JuMP

The JuMP core developers and contributors

May 20, 2025

Contents

Co	ntents		ii
i.	Introduct	tion	1
1	Introdu	ction	2
	1.1	What is JuMP?	2
	1.2	Resources for getting started	3
	1.3	How the documentation is structured	3
	1.4	Citing JuMP	4
	1.5	NumFOCUS	4
	1.6	License	4
2	Should	you use JuMP?	5
	2.1	When should you use JuMP?	5
	2.2	When should you not use JuMP?	6
3	Installa	tion Guide	8
	3.1	Install Julia	8
	3.2	Install JuMP	8
	3.3	Install a solver	9
	3.4	Supported solvers	9
	3.5	AMPL-based solvers	10
	3.6	GAMS-based solvers	11
	3.7	NEOS-based solvers	11
	3.8	Common installation issues	11
н	Tutorials	I	L4
4	Getting	started	15
	4.1	Introduction	15
	4.2	Getting started with Julia	15
	4.3	Getting started with JuMP	33
	4.4	Getting started with sets and indexing	50
	4.5	Getting started with data and plotting	60
	4.6	Debugging	77
	4.7	Tolerances and numerical issues	86
	4.8	Design patterns for larger models	98
	4.9	Performance tips	15
	4.10	Performance problems with sum-if formulations	19
5	Transiti	oning 1	L 26
	5.1	Transitioning from MATLAB	26
6	Linear p	programs 1	L 37
	6.1	Introduction	.37
	6.2	The knapsack problem example	38

	6.3	The diet problem
	6.4	The cannery problem
	6.5	The factory schedule example
	6.6	The multi-commodity flow problem
	6.7	The network multi-commodity flow problem
	6.8	Tips and tricks
	6.9	Approximating nonlinear functions
	6.10	The facility location problem
	6.11	Financial modeling problems
	6.12	Geographical clustering
	6.13	Network flow problems
	6.14	The transportation problem
	6.15	Multi-objective knapsack
	6.16	Simple multi-objective examples
	6.17	Sudoku
	6.18	N-Queens
	6.19	Constraint programming
	6.20	Callbacks
	6.21	Sensitivity analysis of a linear program
	6.22	Basis matrices
	6.23	Computing the duals of a mixed-integer program
	6.24	Finding multiple feasible solutions
7	Nonline	ear programs 277
	7.1	Introduction
	7.2	Simple examples
	7.3	User-defined operators with vector outputs
	7.4	Automatic differentiation of user-defined operators
	7.5	User-defined Hessians
	7.6	Nested optimization problems
	7.7	Computing Hessians
	7.8	Example: mixed complementarity problems
	7.9	Example: classification problems
	7.10	Example: portfolio optimization
	7.11	Example: nonlinear optimal control of a rocket
_	7.12	Example: optimal control for a Space Shuttle reentry trajectory
8	Conic p	rograms 354
	8.1	Introduction
	8.2	Modeling with cones
	8.3	
	8.4	Arbitrary precision arithmetic
	8.5	Primal and dual warm-starts
	8.6	
	8.7	Chordal decomposition
	8.8	Example: logistic regression
	ö.9	Example: experiment design
	8.10	Example: minimal ellipses
	8.11 0.12	Example: empsoid approximation
	8.12	Example: numg of circles and empses
•	ö.13	Example: quantum state discrimination
9	Algoriti	Panders decomposition 446
	9.1	Column concretion
	9.2	Column generation

13.8

	9.3	Traveling Salesperson Problem
	9.4	Rolling horizon problems
	9.5	Parallelism
	9.6	Writing a solver interface
10	Applica	tions 510
	10.1	Power Systems
	10.2	Optimal power flow
	10.3	Serving web apps
	10.4	Two-stage stochastic programs

	Manual		552
11	Models		552
	11 1	Create a medial	552
	11.1		. 555
	11.2	Changing the number types	555
	11.5		
	11.4		. 550
	11.5		. 557
	11.0	Set a time limit	. 33/
	11.7		. 556
	11.8		. 558
	11.9		. 560
	11.10		. 501
	11.11	Backends	. 562
	11.12	Direct mode	. 565
12	Variable	25	567
	12.1	Create a variable	. 567
	12.2	Registered variables	. 570
	12.3	Anonymous variables	. 571
	12.4	Variable names	. 572
	12.5	String names, symbolic names, and bindings	. 574
	12.6	Create, delete, and modify variable bounds	. 575
	12.7	Binary variables	. 577
	12.8	Integer variables	. 578
	12.9	Semi-integer and semi-continuous variables	. 579
	12.10	Start values	. 579
	12.11	Delete a variable	. 580
	12.12	Variable containers	. 581
	12.13	Semidefinite variables	. 587
	12.14	Symmetric variables	. 588
	12.15	The @variables macro	. 590
	12.16	Variables constrained on creation	. 590
	12.17	Parameters	. 594
13	Constra	ints	599
	13.1	Add a constraint	. 599
	13.2	Vectorized constraints	. 600
	13.3	Matrix inequalities	. 602
	13.4	Containers of constraints	. 604
	13.5	Registered constraints	. 605
	13.6	Anonymous constraints	. 606
	13.7	Constraint names	. 607

	13.9	The @constraints macro610
	13.10	Duality
	13.11	Modify a constant term
	13.12	Modify a variable coefficient
	13.13	Delete a constraint
	13.14	Start values
	13.15	Constraint containers
	13.16	Accessing constraints from a model
	13.17	MathOptInterface constraints
	13.18	Set inequality syntax
	13.19	Second-order cone constraints
	13 20	Rotated second-order cone constraints 624
	13 21	Special Ordered Sets of Type 1
	13.21	Special Ordered Sets of Type 2
	13.22	Indicator constraints 625
	12.23	Semidefinite constraints 626
	12.24	
	12.25	
14	15.20 Exercise	
14		Nons 830
	14.1	
	14.2	Quadratic expressions
	14.3	Nonlinear expressions
	14.4	Initializing arrays
15	Objectiv	639
	15.1	Set a linear objective
	15.2	Set a quadratic objective
	15.3	Set a nonlinear objective
	15.4	Query the objective function
	15.5	Evaluate the objective function at a point
	15.6	Query the objective sense
	15.7	Modify an objective
	15.8	Modify an objective coefficient
	15.9	Modify the objective sense
	15.10	Remove an objective
	15.11	Set a vector-valued objective
16	Contain	ers 646
	16.1	Array
	16.2	DenseAxisArray
	16.3	SparseAxisArray
	16.4	Forcing the container type
	16.5	How different container types are chosen
17	Solution	ns 656
	17.1	Check if an optimal solution exists
	17.2	Solutions summary
	17.3	Why did the solver stop?
	17.4	Primal solutions
	17.5	Dual solutions
	17.6	Recommended workflow
	17.7	OptimizeNotCalled errors
	17.8	Accessing attributes
	17.9	Sensitivity analysis for LP
	17.10	Conflicts
	-	

	17.11	Multiple solutions	\$
	17.12	Checking feasibility of solutions)
18	Solver-in	ndependent Callbacks 672	2
	18.1	Available solvers	2
	18.2	Things you can and cannot do during solver-independent callbacks	3
	18.3	Lazy constraints	3
	18.4	User cuts	ļ
	18.5	Heuristic solutions	j
19	Complex	a number support 677	7
	19.1	Complex-valued variables	'
	19.2	Complex-valued variable and start values bounds	3
	19.3	Complex-valued equality constraints)
	19.4	Hermitian PSD Cones)
	19.5	Hermitian PSD constraints	,
20	Nonline	ar Modeling 684	1
	20.1	Set a nonlinear objective 684	Ľ
	20.1	Add a poplinear constraint 684	r I
	20.2	Add a normeter	
	20.5		,
	20.4) ,
	20.5	Common Subexpressions	,
	20.6		
	20.7	Nonlinear expressions in detail	5
	20.8	Function tracing	-
	20.9	User-defined operators	;
21	Nonlinea	ar Modeling (Legacy) 702	2
	21.1	Set a nonlinear objective	!
	21.2	Add a nonlinear constraint	\$
	21.3	Create a nonlinear expression	;
	21.4	Create a nonlinear parameter	ŀ
	21.5	Syntax notes)
	21.6	User-defined Functions	'
	21.7	Factors affecting solution time	
	21.8	Querying derivatives from a JuMP model	-
	21.9	Raw expression input	3
	21.10	Known performance issues	,
	PI Refer	ence 718	
22	Docstrin	as 710	•
~~	22.1	y y y y y y y y y y y y y y y y y y y	Ś
	22.1		:
	22.2		,
VE	Backgrou	nd Information 947	_
23	Algebrai	c modeling languages 948	3
	23.1	What is an algebraic modeling language?	\$
	23.2	From user to solver)
24	Bibliogra	aphy 954	ł
VIC	Develope	r Docs 956	
25	Contribu	iting 957	7
	25.1	How to contribute to JuMP	,

vi

26	Extens	ions	962
	26.1	Extensions	. 962
27	Custon	n binaries	974
	27.1	How to use a custom binary	. 974
28	Style G	iuide	980
	28.1	Style guide and design principles	. 980
29	Conver	ntions	992
	29.1	Conventions for interfacing between JuMP and MathOptInterface	. 992
30	Roadm	ap	994
	30.1	Development roadmap	. 994
31	Checkl	ists	996
	31.1	Checklists	. 996

	1athOptI	nterface	998
32	Introduc	tion	999
	32.1	Introduction	. 999
	32.2	Motivation	. 1000
33	Tutorials	5	1001
	33.1	Solving a problem using MathOptInterface	. 1001
	33.2	Implementing a solver interface	. 1003
	33.3	Transitioning from MathProgBase	. 1017
	33.4	Implementing a constraint bridge	. 1018
	33.5	Manipulating expressions	. 1024
	33.6	Latency	. 1026
34	Manual		1033
	34.1	Standard form problem	. 1033
	34.2	Models	. 1036
	34.3	Variables	. 1038
	34.4	Constraints	. 1039
	34.5	Solutions	. 1043
	34.6	Problem modification	. 1046
35	Backgro	und	1052
	35.1	Duality	. 1052
	35.2	Infeasibility certificates	. 1058
	35.3	Naming conventions	. 1060
36	API Refe	erence	1062
	36.1	Standard form	. 1062
	36.2	Models	. 1107
	36.3	Variables	. 1156
	36.4	Constraints	. 1164
	36.5	Modifications	. 1175
	36.6	Nonlinear programming	. 1179
	36.7	Callbacks	. 1198
	36.8	Errors	. 1204
37	Submod	ules	1210
	37.1	Benchmarks	. 1210
	37.2	Bridges	. 1212
	37.3	FileFormats	. 1294
	37.4	Nonlinear	. 1305
	37.5	Utilities	. 1338
	37.6	Test	1393

38	Develop	er Docs	1403
	38.1	Checklists	. 1403

Part I

Introduction

Chapter 1

Introduction



Welcome to the documentation for JuMP.

Note

This documentation is also available in PDF format: JuMP.pdf.

1.1 What is JuMP?

JuMP is a domain-specific modeling language for mathematical optimization embedded in Julia.

JuMP makes it easy to formulate and solve a range of problem classes, including linear programs, integer programs, conic programs, semidefinite programs, and constrained nonlinear programs. Here's an example:

```
julia> using JuMP, Ipopt
julia> function solve_constrained_least_squares_regression(A::Matrix, b::Vector)
    m, n = size(A)
    model = Model(Ipopt.Optimizer)
    set_silent(model)
    @variable(model, x[1:n])
    @variable(model, residuals[1:m])
    @constraint(model, residuals == A * x - b)
    @constraint(model, sum(x) == 1)
    @objective(model, Min, sum(residuals.^2))
    optimize!(model)
    return value.(x)
    end
solve_constrained_least_squares_regression (generic function with 1 method)
julia> A, b = rand(10, 3), rand(10);
```

```
julia> x = solve_constrained_least_squares_regression(A, b)
3-element Vector{Float64}:
    0.4137624719002825
    0.09707679853084578
    0.48916072956887174
```

Тір

If you aren't sure if you should use JuMP, read Should you use JuMP?.

1.2 Resources for getting started

There are a few ways to get started with JuMP:

- Read the Installation Guide.
- Read the introductory tutorials Getting started with Julia and Getting started with JuMP.
- Browse some of our modeling tutorials, including classics such as The diet problem, or the Maximum likelihood estimation problem using nonlinear programming.

Tip

Need help? Join the community forum to search for answers to commonly asked questions.

Before asking a question, make sure to read the post make it easier to help you, which contains a number of tips on how to ask a good question.

1.3 How the documentation is structured

Having a high-level overview of how this documentation is structured will help you know where to look for certain things.

- **Tutorials** contain worked examples of solving problems with JuMP. Start here if you are new to JuMP, or you have a particular problem class you want to model.
- The Manual contains short code-snippets that explain how to achieve specific tasks in JuMP. Look here
 if you want to know how to achieve a particular task, such as how to Delete a variable or how to Modify
 an objective coefficient.
- The **API Reference** contains a complete list of the functions you can use in JuMP. Look here if you want to know how to use a particular function.
- The Background information section contains background reading material to provide context to JuMP. Look here if you want an understanding of what JuMP is and why we created it, rather than how to use it.



Figure 1.1: NumFOCUS logo

- The **Developer docs** section contains information for people contributing to JuMP development or writing JuMP extensions. Don't worry about this section if you are using JuMP to formulate and solve problems as a user.
- The MathOptInterface section is a self-contained copy of the documentation for MathOptInterface. Look here for functions and constants beginning with MOI., as well as for general information on how MathOptInterface works.

1.4 Citing JuMP

If you find JuMP useful in your work, we kindly request that you cite the following paper (preprint):

```
}
```

1.5 NumFOCUS

JuMP is a Sponsored Project of NumFOCUS, a 501(c)(3) nonprofit charity in the United States. NumFOCUS provides JuMP with fiscal, legal, and administrative support to help ensure the health and sustainability of the project. Visit numfocus.org for more information.

You can support JuMP by donating.

Donations to JuMP are managed by NumFOCUS. For donors in the United States, your gift is tax-deductible to the extent provided by law. As with any donation, you should consult with your tax adviser about your particular tax situation.

JuMP's largest expense is the annual JuMP-dev workshop. Donations will help us provide travel support for JuMP-dev attendees and take advantage of other opportunities that arise to support JuMP development.

1.6 License

JuMP is licensed under the MPL-2.0 software license. Consult the license and the Mozilla FAQ for more information. In addition, JuMP is typically used in conjunction with solver packages and extensions which have their own licences. Consult their package repositories for the specific licenses that apply.

Chapter 2

Should you use JuMP?

JuMP is an algebraic modeling language for mathematical optimization written in the Julia language.

This page explains when you should consider using JuMP, and importantly, when you should not use JuMP.

2.1 When should you use JuMP?

You should use JuMP if you have a constrained optimization problem that is formulated using the language of mathematical programming, that is, the problem has:

- · a set of real- or complex-valued decision variables
- a scalar- or vector-valued real objective function
- a set of constraints.

Key reasons to use JuMP include:

- User friendliness
 - JuMP has syntax that mimics natural mathematical expressions. (See the section on algebraic modeling languages.)
- Solver independence
 - JuMP uses a generic solver-independent interface provided by the MathOptInterface package, making it easy to change between a number of open-source and commercial optimization software packages ("solvers"). The Supported solvers section contains a table of the currently supported solvers.
- · Ease of embedding
 - JuMP itself is written purely in Julia. Solvers are the only binary dependencies.
 - JuMP provides automatic installation of most solvers.
 - Because it is embedded in a general-purpose programming language, JuMP makes it easy to solve
 optimization problems as part of a larger workflow, for example, inside a simulation, behind a
 web server, or as a subproblem in a decomposition algorithm. As a trade-off, JuMP's syntax is
 constrained by the syntax and functionality available in Julia.

- JuMP is MPL licensed, meaning that it can be embedded in commercial software that complies with the terms of the license.
- Speed
 - Benchmarking has shown that JuMP can create problems at similar speeds to special-purpose modeling languages such as AMPL.
 - JuMP communicates with most solvers in memory, avoiding the need to write intermediary files.
- · Access to advanced algorithmic techniques
 - JuMP supports efficient in-memory re-solves of models.
 - JuMP provides access to solver-independent and solver-dependent Callbacks.

2.2 When should you not use JuMP?

JuMP supports a broad range of optimization classes. However, there are still some that it doesn't support, or that are better supported by other software packages.

You want to optimize a complicated Julia function

Packages in Julia compose well. It's common for people to pick two unrelated packages and use them in conjunction to create novel behavior. JuMP isn't one of those packages.

If you want to optimize an ordinary differential equation from DifferentialEquations.jl or tune a neural network from Flux.jl, consider using other packages such as:

- Optim.jl
- Optimization.jl
- NLPModels.jl
- Nonconvex.jl

Black-box, derivative free, or unconstrained optimization

JuMP supports nonlinear programs with constraints and objectives containing user-defined operators. However, the functions must be automatically differentiable, or you need to provide explicit derivatives. (See Userdefined operators for more information.)

If your function is a black-box that is non-differentiable (for example, the function calls a simulation written in C++), JuMP is not the right tool for the job. This also applies if you want to use a derivative free method.

Even if your problem is differentiable, if it is unconstrained there is limited benefit (and downsides in the form of more overhead) to using JuMP over tools which are concerned only with function minimization.

Alternatives to consider are:

- Optim.jl
- Optimization.jl
- NLopt.jl

Disciplined convex programming

JuMP does not support disciplined convex programming (DCP).

Alternatives to consider are:

- Convex.jl
- CVXPY [Python]
- YALMIP [MATLAB]

Note

Convex.jl is also built on MathOptInterface, and shares the same set of underlying solvers. However, you input problems differently, and Convex.jl checks that the problem is DCP.

Stochastic programming

JuMP requires deterministic input data.

If you have stochastic input data, consider using a JuMP extension such as:

- InfiniteOpt.jl
- StochasticPrograms.jl
- SDDP.jl

Polyhedral computations

JuMP does not provide tools for working with the polyhedron formed by the set of linear constraints. Alternatives to consider are:

• Polyhedra.jl (See the documentation to create a polyhedron from a JuMP model.)

Chapter 3

Installation Guide

This guide explains how to install Julia and JuMP. If you have installation troubles, read the Common installation issues section below.

3.1 Install Julia

JuMP is a package for Julia. To use JuMP, first download and install Julia.



Choosing a version

You can install the "Current stable release" or the "Long-term support (LTS) release."

- The "Current stable release" is the latest release of Julia. It has access to newer features, and is likely faster.
- The "Long-term support release" is an older version of Julia that has continued to receive bug and security fixes. However, it may not have the latest features or performance improvements.

For most users, you should install the "Current stable release," and whenever Julia releases a new version of the current stable release, you should update your version of Julia. Note that any code you write on one version of the current stable release will continue to work on all subsequent releases.

For users in restricted software environments (for example, your enterprise IT controls what software you can install), you may be better off installing the long-term support release because you will not have to update Julia as frequently.

3.2 Install JuMP

JuMP is installed using the built-in Julia package manager. Launch Julia, and then enter the following at the julia> prompt:

julia> **import** Pkg

julia> Pkg.add("JuMP")

Тір

We recommend you create a Pkg *environment* for each project you use JuMP for, instead of adding lots of packages to the global environment. The Pkg manager documentation has more information on this topic.

When we release a new version of JuMP, you can update with:

julia> import Pkg

julia> Pkg.update("JuMP")

3.3 Install a solver

JuMP depends on solvers to solve optimization problems. Therefore, you will need to install one before you can solve problems with JuMP.

Install a solver using the Julia package manager, replacing "HiGHS" by the Julia package name as appropriate.

julia> **import** Pkg

julia> Pkg.add("HiGHS")

Once installed, you can use HiGHS as a solver with JuMP as follows, using set_attribute to set solver-specific options:

```
julia> using JuMP
```

julia> using HiGHS

julia> model = Model(HiGHS.Optimizer);

```
julia> set_attribute(model, "output_flag", false)
```

```
julia> set_attribute(model, "primal_feasibility_tolerance", 1e-8)
```

Note

Most packages follow the ModuleName.Optimizer naming convention, but exceptions may exist. See the README of the Julia package's GitHub repository for more details on how to use a particular solver, including any solver-specific options.

3.4 Supported solvers

Most solvers are not written in Julia, and some require commercial licenses to use, so installation is often more complex.

- If a solver has Manual in the Installation column, the solver requires a manual installation step, such as downloading and installing a binary, or obtaining a commercial license. Consult the README of the relevant Julia package for more information.
- If the solver has Manual^M in the Installation column, the solver requires an installation of MATLAB.
- If the Installation column is missing an entry, installing the Julia package will download and install any relevant solver binaries automatically, and you shouldn't need to do anything other than Pkg.add.

Solvers with a missing entry in the Julia Package column are written in Julia. The link in the Solver column is the corresponding Julia package.

Where:

- LP = Linear programming
- QP = Quadratic programming
- SOCP = Second-order conic programming (including problems with convex quadratic constraints or objective)
- MCP = Mixed-complementarity programming
- NLP = Nonlinear programming
- SDP = Semidefinite programming
- (MI)XXX = Mixed-integer equivalent of problem type XXX
- CP-SAT = Constraint programming and Boolean satisfiability

Note

Developed a solver or solver wrapper? This table is open for new contributions. Edit the installation.md file, and use the checklist Adding a new solver to the documentation when opening the pull request.

Note

Developing a solver or solver wrapper? See Models and the MathOptInterface docs for more details on how JuMP interacts with solvers. Please get in touch via the Developer Chatroom with any questions about connecting new solvers with JuMP.

3.5 AMPL-based solvers

Use AmpINLWriter to access solvers that support the NL format.

Some solvers, such as Bonmin, Couenne and SHOT can be installed via the Julia package manager. Others need to be manually installed.

Consult the AMPL documentation for a complete list of supported solvers.

3.6 GAMS-based solvers

Use GAMS.jl to access solvers available through GAMS. Such solvers include: AlphaECP, Antigone, BARON, CONOPT, Couenne, LocalSolver, PATHNLP, SHOT, SNOPT, SoPlex. See a complete list here.

Note

GAMS.jl requires an installation of the commercial software GAMS for which a free community license exists.

3.7 NEOS-based solvers

Use NEOSServer.jl to access solvers available through the NEOS Server.

3.8 Common installation issues

Tip

When in doubt, run import Pkg; Pkg.update() to see if updating your packages fixes the issue. Remember you will need to exit Julia and start a new session for the changes to take effect.

Check the version of your packages

Each package is versioned with a three-part number of the form vX.Y.Z. You can check which versions you have installed with import Pkg; Pkg.status().

This should almost always be the most-recent release. You can check the releases of a package by going to the relevant GitHub page, and navigating to the "releases" page. For example, the list of JuMP releases is available at: https://github.com/jump-dev/JuMP.jl/releases.

If you post on the community forum, please include the output of Pkg.status().

Unsatisfiable requirements detected

Did you get an error like Unsatisfiable requirements detected for package JuMP? The Pkg documentation has a section on how to understand and manage these conflicts.

Installing new packages can make JuMP downgrade to an earlier version

Another common complaint is that after adding a new package, code that previously worked no longer works.

This usually happens because the new package is not compatible with the latest version of JuMP. Therefore, the package manager rolls-back JuMP to an earlier version. Here's an example.

First, we add JuMP:

```
(jump_example) pkg> add JuMP
Resolving package versions...
Updating `~/jump_example/Project.toml`
[4076af6c] + JuMP v0.21.5
Updating `~/jump_example/Manifest.toml`
... lines omitted ...
```

The + JuMP v0.21.5 line indicates that JuMP has been added at version 0.21.5. However, watch what happens when we add JuMPeR:

```
(jump_example) pkg> add JuMPeR
Resolving package versions...
Updating `~/jump_example/Project.toml`
[4076af6c] ↓ JuMP v0.21.5 ⇒ v0.18.6
[707a9f91] + JuMPeR v0.6.0
Updating `~/jump_example/Manifest.toml`
... lines omitted ...
```

JuMPeR gets added at version 0.6.0 (+ JuMPeR v0.6.0), but JuMP gets downgraded from 0.21.5 to 0.18.6 (\downarrow JuMP v0.21.5 \Rightarrow v0.18.6)! The reason for this is that JuMPeR doesn't support a version of JuMP newer than 0.18.6.

Тір

Pay careful attention to the output of the package manager when adding new packages, especially when you see a package being downgraded.

Supports	License	Installation	Julia Package	Solver
(MI)NLP	Triad NS			Alpine.jl
(MI)LP, (MI)SOCP, (MI)NLP	Comm.	Manual	KNITRO.jl	Artelys Knitro
(MI)NLP	Comm.	Manual	BARON.jl	BARON
(MI)NLP	EPL		AmplNLWriter.jl	Bonmin
(MI)LP	EPL		Cbc.jl	Cbc
LP, SOCP, SDP	GPL	Manual [™]	CDCS.jl	CDCS
LP	GPL		CDDLib.jl	CDD
LP, QP, SOCP, SDP	Apache			Clarabel.jl
LP	EPL		Clp.jl	qlD
(MI)LP, SOCP, SDP	Comm.		COPT.il	COPT
LP. OP. SOCP. SDP	Apache		,	COSMO.il
(MI)NLP	EPL		AmpINLWriter.il	Couenne
(MI)LP. (MI)SOCP	Comm.	Manual	CPI FX.il	CPI FX
	FPI		CSDPil	CSDP
(Mixed-binary) OP	MIT			
(MI)NI P	MIT		0501.ji	FAGO il
	GPI		ECOS il	ECOS
	Comm	Manual	Zecos.ji	ELCO Ypross
		Manual		
	GFL	Manual	GLFN.JI Curobi il	GLFK
	Comm.	Manual	Gurobi.ji	Gurobi
(MI)LP, QP	MIT		HIGHS.JI	HIGHS
LP, SOCP, SDP				Hypatia.Ji
LP, QP, NLP	EPL		Ipopt.JI	Ipopt
(MI)SOCP, (MI)NLP	MIT			Juniper.ji
LP, SDP	MIT			Loraine.ji
LP, QP, NLP	MII			MadNLP.JI
(MI)NLP	EPL 2.0		MAINGO.JI	MAINGO
NLP	MIT			Manopt.jl
CP-SAT	MPL-2	Manual	MiniZinc.jl	MiniZinc
(MI)NLP	BSD-like	Manual	AmplNLWriter.jl	Minotaur
(MI)LP, (MI)SOCP, SDP	Comm.	Manual	MosekTools.jl	MOSEK
LP, QP, NLP	GPL		NLopt.jl	NLopt
(MI)NLP	Comm.		AmplNLWriter.jl	Octeract
NLP	MIT			Optim.jl
LP, QP	Apache		OSQP.jl	OSQP
MCP	MIT		PATHSolver.jl	PATH
(MI)NLP, (MI)SOCP, (MI)SDP	MPL-2			Pajarito.jl
(MI)NLP	MPL-2			Pavito.jl
Bilinear SDP	Comm.		Penopt.jl	Penbmi
NLP	MPL-2			Percival.jl
NLP	MIT		PolyJuMP.jl	PolyJuMP.KKT
NLP	MIT		PolyJuMP.jl	PolyJuMP.QCQP
LP, SOCP, SDP	MIT			ProxSDP.jl
(MI)NLP	RAPOSa	Manual	AmplNLWriter.jl	RAPOSa
(MI)LP, (MI)NLP	Apache		SCIP.jl	SCIP
LP, QP, SOCP, SDP	MIT		SCS.jl	SCS
LP, SDP	GPL		SDPA.jl, SDPAFamily.jl	SDPA
LP. SDP	GPL		SDPLR.il	SDPLR
LP. SDP	CC BY-SA	Manual [™]	SDPNAL.il	SDPNAL
LP. SOCP. SDP	GPI	Manual [™]	SDPT3.il	SDPT3
	GPI	Manual [™]	SeDuMi il	SeDuMi
	FPI		AmnINI Writer il	SHOT
	міт		Ampinemittenji	StatusSwitchingOPil
נו, ער ום	MPI_2			Tidin il
LF	1411 L-Z			

Part II

Tutorials

Chapter 4

Getting started

4.1 Introduction

The purpose of these "Getting started" tutorials is to teach new users the basics of Julia and JuMP.

How these tutorials are structured

Having a high-level overview of how this part of the documentation is structured will help you know where to look for certain things.

- The "Getting started with" tutorials are basic introductions to different aspects of JuMP and Julia. If you are new to JuMP and Julia, start by reading them in the following order:
 - Getting started with Julia
 - Getting started with JuMP
 - Getting started with sets and indexing
 - Getting started with data and plotting
- Julia has a reputation for being "fast." Unfortunately, it is also easy to write *slow* Julia code. Performance tips contains a number of important tips on how to improve the performance of models you write in JuMP.
- Design patterns for larger models is a more advanced tutorial that is aimed at users writing large JuMP models. It's in the "Getting started" section to give you an early preview of how JuMP makes it easy to structure larger models. If you are new to JuMP you may want to skip or briefly skim this tutorial, and come back to it once you have written a few JuMP models.

4.2 Getting started with Julia

This tutorial was generated using Literate.jl. Download the source as a .jl file.

Because JuMP is embedded in Julia, knowing some basic Julia is important before you start learning JuMP.

Тір

This tutorial is designed to provide a minimalist crash course in the basics of Julia. You can find resources that provide a more comprehensive introduction to Julia here.

Installing Julia

To install Julia, download the latest stable release, then follow the platform specific install instructions.



Next, you need an IDE to develop in. VS Code is a popular choice, so follow these install instructions.

Julia can also be used with Jupyter notebooks or the reactive notebooks of Pluto.jl.

The Julia REPL

The main way of interacting with Julia is via its REPL (Read Evaluate Print Loop). To access the REPL, start the Julia executable to arrive at the julia> prompt, and then start coding:

```
julia> 1 + 1
2
```

As your programs become larger, write a script as a text file, and then run that file using:

```
julia> include("path/to/file.jl")
```

Warning

Because of Julia's startup latency, running scripts from the command line like the following is slow:

\$ julia path/to/file.jl

Use the REPL or a notebook instead.

Code blocks in this documentation

In this documentation you'll see a mix of code examples with and without the julia>.

The Julia prompt is mostly used to demonstrate short code snippets, and the output is exactly what you will see if run from the REPL.

Blocks without the julia> can be copy-pasted into the REPL, but they are used because they enable richer output like plots or LaTeX to be displayed in the online and PDF versions of the documentation. If you run them from the REPL you may see different output.

Where to get help

- Read the documentation
 - JuMP https://jump.dev/JuMP.jl/stable/
 - Julia https://docs.julialang.org/en/v1/
- Ask (or browse) the Julia community forum: https://discourse.julialang.org

- If the question is JuMP-related, ask in the Optimization (Mathematical) section, or tag your question with "jump"

To access the built-in help at the REPL, type ? to enter help-mode, followed by the name of the function to lookup:

```
help?> print
search: print println printstyled sprint isprint prevind parentindices precision escape_string
```

print([io::**IO**], xs...)

Write to io (or to the default output stream stdout if io is not given) a canonical (un-decorated) text representation. The representation used by print includes minimal formatting and tries to avoid Julia-specific details.

print falls back to calling show, so most types should just define show. Define print if your type has a separate "plain" representation. For example, show displays strings with quotes, and print displays strings without quotes.

string returns the output of print as a string.

Numbers and arithmetic

Since we want to solve optimization problems, we're going to be using a lot of math. Luckily, Julia is great for math, with all the usual operators:

```
julia> 1 + 1
2
julia> 1 - 2
-1
julia> 2 * 2
4
julia> 4 / 5
0.8
julia> 3^2
9
```

Did you notice how Julia didn't print .0 after some of the numbers? Julia is a dynamic language, which means you never have to explicitly declare the type of a variable. However, in the background, Julia is giving each variable a type. Check the type of something using the typeof function:

```
julia> typeof(1)
Int64
julia> typeof(1.0)
Float64
```

Here 1 is an Int64, which is an integer with 64 bits of precision, and 1.0 is a Float64, which is a floating point number with 64-bits of precision.



We create complex numbers using im:

```
julia> x = 2 + 1im
2 + 1im
julia> real(x)
2
julia> imag(x)
1
julia> typeof(x)
Complex{Int64}
julia> x * (1 - 2im)
```

Info

4 - 3im

The curly brackets surround what we call the *parameters* of a type. You can read Complex{Int64} as "a complex number, where the real and imaginary parts are represented by Int64." If we call typeof(1.0 + 2.0im) it will be Complex{Float64}, which a complex number with the parts represented by Float64.

There are also some cool things like an irrational representation of π .

```
julia> π
π = 3.1415926535897...
```

Тір

To make π (and most other Greek letters), type \pi and then press [TAB].

CHAPTER 4. GETTING STARTED

julia> typeof(π)
Irrational{:π}

However, if we do math with irrational numbers, they get converted to Float64:

julia> typeof(2π / 3) Float64

Floating point numbers

Warning

If you aren't familiar with floating point numbers, make sure to read this section carefully.

A Float64 is a floating point approximation of a real number using 64-bits of information.

Because it is an approximation, things we know hold true in mathematics don't hold true in a computer. For example:

julia> 0.1 * 3 == 0.3 false

A more complicated example is:

julia> sin(2π / 3) == $\sqrt{3}$ / 2 false

Тір

Get \checkmark by typing \sqrt then press [TAB].

Let's see what the differences are:

```
julia> 0.1 * 3 - 0.3
5.551115123125783e-17
```

julia> sin(2π / 3) - √3 / 2 1.1102230246251565e-16

They are small, but not zero.

One way of explaining this difference is to consider how we would write 1 / 3 and 2 / 3 using only four digits after the decimal point. We would write 1 / 3 as 0.3333, and 2 / 3 as 0.6667. So, despite the fact that 2 * (1 / 3) = 2 / 3, 2 * 0.3333 = 0.66666 != 0.66667.

Let's try that again using \approx (\approx + [TAB]) instead of ==:

```
julia> 0.1 * 3 \approx 0.3
true
julia> sin(2\pi / 3) \approx \sqrt{3} / 2
true
```

 \approx is a clever way of calling the isapprox function:

```
julia> isapprox(sin(2\pi / 3), \sqrt{3} / 2; atol = 1e-8) true
```

Warning

Floating point is the reason solvers use tolerances when they solve optimization models. A common mistake you're likely to make is checking whether a binary variable is 0 using value(z) == 0. Always remember to use something like isapprox when comparing floating point numbers.

Note that isapprox will always return false if one of the number being compared is 0 and atol is zero (its default value).

```
julia> 1e-300 ≈ 0.0
false
```

so always set a nonzero value of atol if one of the arguments can be zero.

```
julia> isapprox(le-9, 0.0; atol = le-8)
true
```

Тір

Gurobi has a good series of articles on the implications of floating point in optimization if you want to read more.

If you aren't careful, floating point arithmetic can throw up all manner of issues. For example:

```
julia> 1 + 1e-16 == 1
true
```

It even turns out that floating point numbers aren't associative:

```
julia> (1 + 1e-16) - 1e-16 == 1 + (1e-16 - 1e-16) false
```

It's important to note that this issue isn't Julia-specific. It happens in every programming language (try it out in Python).

Vectors, matrices, and arrays

Similar to MATLAB, Julia has native support for vectors, matrices and tensors; all of which are represented by arrays of different dimensions. Vectors are constructed by comma-separated elements surrounded by square brackets:

```
julia> b = [5, 6]
2-element Vector{Int64}:
5
6
```

Matrices can be constructed with spaces separating the columns, and semicolons separating the rows:

```
julia> A = [1.0 2.0; 3.0 4.0]
2×2 Matrix{Float64}:
    1.0 2.0
    3.0 4.0
```

We can do linear algebra:

```
julia> x = A \ b
2-element Vector{Float64}:
    -3.09999999999999997
    4.499999999999999
```

Info

Here is floating point at work again; x is approximately [-4, 4.5].

```
julia> A * x
2-element Vector{Float64}:
    5.0
    6.0
julia> A * x ≈ b
true
```

Note that when multiplying vectors and matrices, dimensions matter. For example, you can't multiply a vector by a vector:

```
julia> b * b
MethodError: no method matching *(::Vector{Int64}, ::Vector{Int64})
The function `*` exists, but no method is defined for this combination of argument types.
Closest candidates are:
 *(::Any, ::Any, !Matched::Any, !Matched::Any...)
@ Base operators.jl:596
 *(!Matched::Type{<:LinearOperatorCollection.ProdOp}, ::Any, !Matched::Any)
@ LinearOperatorCollection ~/.julia/packages/LinearOperatorCollection/qbON0/src/ProdOp.jl:73
 *(!Matched::ChainRulesCore.NotImplemented, ::Any)
```

```
@ ChainRulesCore ~/.julia/packages/ChainRulesCore/U6wNx/src/tangent_arithmetic.jl:37
...
```

But multiplying transposes works:

```
julia> b' * b
61
julia> b * b'
2×2 Matrix{Int64}:
25 30
30 36
```

Other common types

Comments

Although not technically a type, code comments begin with the # character:

```
julia> 1 + 1 # This is a comment
2
```

Multiline comments begin with #= and end with =#:

```
#=
Here is a
multiline comment
=#
```

Comments can even be nested inside expressions. This is sometimes helpful when documenting inputs to functions:

Strings

Double quotes are used for strings:

```
julia> typeof("This is Julia")
String
```

Unicode is fine in strings:

```
julia> typeof("π is about 3.1415")
String
```

Use println to print a string:

julia> println("Hello, World!")
Hello, World!

Use \$() to interpolate values into a string:

julia> x = 123 123

julia> println("The value of x is: \$(x)")
The value of x is: 123

Use triple-quotes for multiline strings:

```
julia> s = """
Here is
a
multiline string
"""
"Here is\na\nmultiline string\n"
```

julia> println(s)

Here is a multiline string

Symbols

Julia Symbols are a data structure from the compiler that represent Julia identifiers (that is, variable names).

```
julia> println("The value of x is: $(eval(:x))")
The value of x is: 123
```

Warning

We used eval here to demonstrate how Julia links Symbols to variables. However, avoid calling eval in your code. It is usually a sign that your code is doing something that could be more easily achieved a different way. The Community Forum is a good place to ask for advice on alternative approaches.

julia> typeof(:x) Symbol You can think of a Symbol as a String that takes up less memory, and that can't be modified.

Convert between String and Symbol using their constructors:

```
julia> String(:abc)
"abc"
julia> Symbol("abc")
```

:abc

Тір

Symbols are often (ab)used to stand in for a String or an Enum, when one of the latter is likely a better choice. The JuMP Style guide recommends reserving Symbols for identifiers. See @enum vs. Symbol for more.

Tuples

Julia makes extensive use of a simple data structure called Tuples. Tuples are immutable collections of values. For example:

```
julia> t = ("hello", 1.2, :foo)
("hello", 1.2, :foo)
```

julia> typeof(t)
Tuple{String, Float64, Symbol}

Tuples can be accessed by index, similar to arrays:

julia> t[2] 1.2

And they can be "unpacked" like so:

```
julia> a, b, c = t
("hello", 1.2, :foo)
julia> b
1.2
```

The values can also be given names, which is a convenient way of making light-weight data structures.

julia> t = (word = "hello", num = 1.2, sym = :foo)
(word = "hello", num = 1.2, sym = :foo)

Values can be accessed using dot syntax:

julia> t.word "hello"

Dictionaries

Similar to Python, Julia has native support for dictionaries. Dictionaries provide a very generic way of mapping keys to values. For example, a map of integers to strings:

```
julia> dl = Dict(1 => "A", 2 => "B", 4 => "D")
Dict{Int64, String} with 3 entries:
    4 => "D"
    2 => "B"
    1 => "A"
```

Info

Type-stuff again: Dict{Int64,String} is a dictionary with Int64 keys and String values.

Looking up a value uses the bracket syntax:

julia> d1[2] "B"

Dictionaries support non-integer keys and can mix data types:

```
julia> Dict("A" => 1, "B" => 2.5, "D" => 2 - 3im)
Dict{String, Number} with 3 entries:
   "B" => 2.5
   "A" => 1
   "D" => 2-3im
```

Info

Julia types form a hierarchy. Here the value type of the dictionary is Number, which is a generalization of Int64, Float64, and Complex{Int}. Leaf nodes in this hierarchy are called "concrete" types, and all others are called "Abstract." In general, having variables with abstract types like Number can lead to slower code, so you should try to make sure every element in a dictionary or vector is the same type. For example, in this case we could represent every element as a Complex{Float64}:

```
julia> Dict("A" => 1.0 + 0.0im, "B" => 2.5 + 0.0im, "D" => 2.0 - 3.0im)
Dict{String, ComplexF64} with 3 entries:
    "B" => 2.5+0.0im
    "A" => 1.0+0.0im
    "D" => 2.0-3.0im
```

Dictionaries can be nested:

julia> d2 = Dict("A" => 1, "B" => 2, "D" => Dict(:foo => 3, :bar => 4))
Dict{String, Any} with 3 entries:
 "B" => 2
 "A" => 1
 "D" => Dict(:bar=>4, :foo=>3)
julia> d2["B"]
2
julia> d2["D"][:foo]
3

Structs

You can define custom datastructures with struct:

By default, these are not mutable

```
julia> a.x = 2
setfield!: immutable struct of type MyStruct cannot be changed
```

However, you can declare a mutable struct which is mutable:

Loops

Julia has native support for for-each style loops with the syntax for <value> in <collection> end:



Ranges are constructed as start:stop, or start:step:stop.

5.6

This for-each loop also works with dictionaries:

Note that in contrast to vector languages like MATLAB and R, loops do not result in a significant performance degradation in Julia.

Control flow

Julia control flow is similar to MATLAB, using the keywords if-elseif-else-end, and the logical operators || and && for **or** and **and** respectively:

```
else
println("$(i) is bigger than 10")
end
end
0 is less than 5
5 is less than 10
the value is 10
15 is bigger than 10
```

Comprehensions

Similar to languages like Haskell and Python, Julia supports the use of simple loops in the construction of arrays and dictionaries, called comprehensions.

A list of increasing integers:

```
julia> [i for i in 1:5]
5-element Vector{Int64}:
1
2
3
4
5
```

Matrices can be built by including multiple indices:

```
julia> [i * j for i in 1:5, j in 5:10]
5×6 Matrix{Int64}:
5 6 7 8 9 10
10 12 14 16 18 20
15 18 21 24 27 30
20 24 28 32 36 40
25 30 35 40 45 50
```

Conditional statements can be used to filter out some values:

julia> [i for i in 1:10 if i % 2 == 1]
5-element Vector{Int64}:
1
3
5
7
9

A similar syntax can be used for building dictionaries:

julia> Dict("\$(i)" => i for i in 1:10 if i % 2 == 1)
Dict{String, Int64} with 5 entries:
 "1" => 1
"5" => 5 "7" => 7 "9" => 9 "3" => 3

Functions

A simple function is defined as follows:

```
julia> function print_hello()
            return println("hello")
            end
print_hello (generic function with 1 method)
julia> print_hello()
```

hello

Arguments can be added to a function:

julia> print_it("hello")
hello

```
julia> print_it(1.234)
1.234
```

```
julia> print_it(:my_id)
my_id
```

Optional keyword arguments are also possible:

The keyword return is used to specify the return values of a function:

Anonymous functions

The syntax input -> output creates an anonymous function. These are most useful when passed to other functions. For example:

```
julia> f = x -> x^2
#11 (generic function with 1 method)
julia> f(2)
4
julia> map(x -> x^2, 1:4)
4-element Vector{Int64}:
    1
    4
    9
    16
```

Type parameters

We can constrain the inputs to a function using type parameters, which are :: followed by the type of the input we want. For example:

```
julia> function foo(x::Int)
    return x^2
    end
foo (generic function with 1 method)
julia> function foo(x::Float64)
    return exp(x)
    end
foo (generic function with 2 methods)
julia> function foo(x::Number)
    return x + 1
    end
foo (generic function with 3 methods)
julia> foo(2)
4
```

julia> foo(2.0**)** 7.38905609893065

julia> foo(1 + lim)
2 + lim

But what happens if we call foo with something we haven't defined it for?

```
julia> foo([1, 2, 3])
MethodError: no method matching foo(::Vector{Int64})
The function `foo` exists, but no method is defined for this combination of argument types.
Closest candidates are:
   foo(!Matched::Float64)
   @ Main REPL[2]:1
   foo(!Matched::Int64)
   @ Main REPL[1]:1
   foo(!Matched::Number)
   @ Main REPL[3]:1
```

A MethodError means that you passed a function something that didn't match the type that it was expecting. In this case, the error message says that it doesn't know how to handle an Vector{Int64}, but it does know how to handle Float64, Int64, and Number.



Read the "Closest candidates" part of the error message carefully to get a hint as to what was expected.

Broadcasting

In the example above, we didn't define what to do if f was passed a Vector. Luckily, Julia provides a convenient syntax for mapping f element-wise over arrays. Just add a . between the name of the function and the opening (. This works for *any* function, including functions with multiple arguments. For example:

```
julia> foo.([1, 2, 3])
3-element Vector{Int64}:
1
4
9
```

Тір

Get a MethodError when calling a function that takes a Vector, Matrix, or Array? Try broadcasting.

Mutable vs immutable objects

Some types in Julia are *mutable*, which means you can change the values inside them. A good example is an array. You can modify the contents of an array without having to make a new array.

In contrast, types like Float64 are immutable. You cannot modify the contents of a Float64.

This is something to be aware of when passing types into functions. For example:

```
julia> function mutability_example(mutable_type::Vector{Int}, immutable_type::Int)
           mutable_type[1] += 1
           immutable_type += 1
           return
       end
mutability_example (generic function with 1 method)
julia> mutable_type = [1, 2, 3]
3-element Vector{Int64}:
1
2
3
julia> immutable_type = 1
1
julia> mutability_example(mutable_type, immutable_type)
julia> println("mutable_type: $(mutable_type)")
mutable_type: [2, 2, 3]
julia> println("immutable_type: $(immutable_type)")
immutable_type: 1
```

Because Vector{Int} is a mutable type, modifying the variable inside the function changed the value outside of the function. In contrast, the change to immutable_type didn't modify the value outside the function.

You can check mutability with the isimmutable function:

```
julia> isimmutable([1, 2, 3])
false
julia> isimmutable(1)
true
```

The package manager

Installing packages

No matter how wonderful Julia's base language is, at some point you will want to use an extension package. Some of these are built-in, for example random number generation is available in the Random package in the standard library. These packages are loaded with the commands using and import.

```
julia> using Random # The equivalent of Python's `from Random import *`
julia> import Random # The equivalent of Python's `import Random`
julia> Random.seed!(33)
Random.TaskLocalRNG()
julia> [rand() for i in 1:10]
```

```
10-element Vector{Float64}:
0.4745319377345316
0.9650392357070774
0.8194019096093067
0.9297749959069098
0.3127122336048005
0.9684448191382753
0.9063743823581542
0.8386731983150535
0.5103924401614957
0.9296414851080324
```

The Package Manager is used to install packages that are not part of Julia's standard library.

For example the following can be used to install JuMP,

using Pkg
Pkg.add("JuMP")

For a complete list of registered Julia packages see the package listing at JuliaHub.

From time to you may wish to use a Julia package that is not registered. In this case a git repository URL can be used to install the package.

```
using Pkg
Pkg.add("https://github.com/user-name/MyPackage.jl.git")
```

Package environments

By default, Pkg.add will add packages to Julia's global environment. However, Julia also has built-in support for virtual environments.

Activate a virtual environment with:

```
import Pkg; Pkg.activate("/path/to/environment")
```

You can see what packages are installed in the current environment with Pkg.status().

Tip

We *strongly* recommend you create a Pkg environment for each project that you create in Julia, and add only the packages that you need, instead of adding lots of packages to the global environment. The Pkg manager documentation has more information on this topic.

4.3 Getting started with JuMP

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial is aimed at providing a quick introduction to writing and solving optimization models with JuMP. If you're new to Julia, start by reading Getting started with Julia.

What is JuMP?

JuMP ("Julia for Mathematical Programming") is an open-source modeling language that is embedded in Julia. It allows users to formulate various classes of optimization problems (linear, mixed-integer, quadratic, conic quadratic, semidefinite, and nonlinear) with easy-to-read code. These problems can then be solved using state-of-the-art open-source and commercial solvers.

JuMP also makes advanced optimization techniques easily accessible from a high-level language.

What is a solver?

A solver is a software package that incorporates algorithms for finding solutions to one or more classes of problem.

For example, HiGHS is a solver for linear programming (LP) and mixed integer programming (MIP) problems. It incorporates algorithms such as the simplex method and the interior-point method.

The Supported-solvers table lists the open-source and commercial solvers that JuMP currently supports.

What is MathOptInterface?

Each solver has its own concepts and data structures for representing optimization models and obtaining results.

MathOptInterface (MOI) is an abstraction layer that JuMP uses to convert from the problem written in JuMP to the solver-specific data structures for each solver.

MOI can be used directly, or through a higher-level modeling interface like JuMP.

Because JuMP is built on top of MOI, you'll often see the MathOptInterface. prefix displayed when JuMP types are printed. However, you'll only need to understand and interact with MOI to accomplish advanced tasks such as creating solver-independent callbacks.

Installation

JuMP is a package for Julia. From Julia, JuMP is installed by using the built-in package manager.

import Pkg
Pkg.add("JuMP")

You also need to include a Julia package which provides an appropriate solver. One such solver is HiGHS.Optimizer, which is provided by the HiGHS.jl package.

import Pkg
Pkg.add("HiGHS")

See Installation Guide for a list of other solvers you can use.

An example

Let's solve the following linear programming problem using JuMP and HiGHS. We will first look at the complete code to solve the problem and then go through it step by step.

Here's the problem:

$$\begin{array}{ll} \min & 12x + 20y \\ \text{s.t.} & 6x + 8y \geq 100 \\ & 7x + 12y \geq 120 \\ & x \geq 0 \\ & y \in [0,3] \end{array}$$

And here's the code to solve this problem:

```
julia> using JuMP
julia> using HiGHS
julia> model = Model(HiGHS.Optimizer)
A JuMP Model
├ solver: HiGHS
bjective_sense: FEASIBILITY_SENSE
- num_variables: 0
- num constraints: 0
L Names registered in the model: none
julia> @variable(model, x >= 0)
Х
julia> @variable(model, 0 <= y <= 3)</pre>
V
julia> @objective(model, Min, 12x + 20y)
12 x + 20 y
julia> @constraint(model, c1, 6x + 8y >= 100)
c1 : 6 \times + 8 \text{ y} \ge 100
julia> @constraint(model, c2, 7x + 12y >= 120)
c2 : 7 \times + 12 y \ge 120
julia> print(model)
Min 12 x + 20 y
Subject to
c1 : 6 \times + 8 y \ge 100
 c2 : 7 \times + 12 y \ge 120
 X \ge 0
 y ≥ 0
 y ≤ 3
julia> optimize!(model)
Running HiGHS 1.10.0 (git hash: fd8665394e): Copyright (c) 2025 HiGHS under MIT licence terms
LP has 2 rows; 2 cols; 4 nonzeros
Coefficient ranges:
 Matrix [6e+00, 1e+01]
  Cost [1e+01, 2e+01]
  Bound [3e+00, 3e+00]
  RHS [1e+02, 1e+02]
```

Presolving model

35

```
2 rows, 2 cols, 4 nonzeros Os
2 rows, 2 cols, 4 nonzeros Os
Presolve : Reductions: rows 2(-0); columns 2(-0); elements 4(-0) - Not reduced
Problem not reduced by presolve: solving the LP
Using EKK dual simplex solver - serial
 Iteration Objective Infeasibilities num(sum)
        0
            0.0000000000e+00 Pr: 2(220) 0s
        2 2.050000000e+02 Pr: 0(0) 0s
Model status : Optimal
Simplex iterations: 2
Objective value : 2.0500000000e+02
Relative P-D gap : 1.3864248503e-16
HiGHS run time
                 :
                          0.00
julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1
```

of TIME. Terminationstatuscode - 1

julia> primal_status(model)
FEASIBLE POINT::ResultStatusCode = 1

julia> dual_status(model)
FEASIBLE_POINT::ResultStatusCode = 1

julia> objective_value(model)
204.999999999999997

julia> value(y)
1.2499999999999996

julia> shadow_price(c1)
-0.249999999999999922

julia> shadow_price(c2)
-1.5000000000000007

Step-by-step

Once JuMP is installed, to use JuMP in your programs write:

julia> using JuMP

We also need to include a Julia package which provides an appropriate solver. We want to use HiGHS.Optimizer here which is provided by the HiGHS.jl package:

julia> using HiGHS

JuMP builds problems incrementally in a Model object. Create a model by passing an optimizer to the Model function:

```
julia> model = Model(HiGHS.Optimizer)
```

```
A JuMP Model
    solver: HiGHS
    objective_sense: FEASIBILITY_SENSE
    num_variables: 0
    num_constraints: 0
    Names registered in the model: none
```

Variables are modeled using @variable:

```
julia> @variable(model, x >= 0)
x
```

Info

The macro creates a new Julia object, x, in the current scope. We could have made this more explicit by writing:

x = @variable(model, x >= 0)

but the macro does this automatically for us to save writing x twice.

Variables can have lower and upper bounds:

julia> @variable(model, 0 <= y <= 30)
y</pre>

The objective is set using <a>left <a>

```
julia> @objective(model, Min, 12x + 20y)
12 x + 20 y
```

Constraints are modeled using @constraint. Here, c1 and c2 are the names of our constraint:

julia> @constraint(model, c1, 6x + 8y >= 100) c1 : 6 x + 8 y ≥ 100 julia> @constraint(model, c2, 7x + 12y >= 120) c2 : 7 x + 12 y ≥ 120

Call print to display the model:

```
julia> print(model)
Min 12 x + 20 y
Subject to
c1 : 6 x + 8 y ≥ 100
c2 : 7 x + 12 y ≥ 120
```

CHAPTER 4. GETTING STARTED

 $\begin{array}{rcl} x & \geq & 0 \\ y & \geq & 0 \\ y & \leq & 30 \end{array}$

To solve the optimization problem, call the optimize! function:

```
julia> optimize!(model)
Running HiGHS 1.10.0 (git hash: fd8665394e): Copyright (c) 2025 HiGHS under MIT licence terms
LP has 2 rows; 2 cols; 4 nonzeros
Coefficient ranges:
 Matrix [6e+00, 1e+01]
 Cost [1e+01, 2e+01]
 Bound [3e+01, 3e+01]
 RHS [1e+02, 1e+02]
Presolving model
2 rows, 2 cols, 4 nonzeros Os
2 rows, 2 cols, 4 nonzeros Os
Presolve : Reductions: rows 2(-0); columns 2(-0); elements 4(-0) - Not reduced
Problem not reduced by presolve: solving the LP
Using EKK dual simplex solver - serial
 Iteration Objective Infeasibilities num(sum)
      0 0.0000000000e+00 Pr: 2(220) 0s
        2 2.050000000e+02 Pr: 0(0) 0s
Model status : Optimal
Simplex iterations: 2
Objective value : 2.0500000000e+02
Relative P-D gap : 0.0000000000e+00
HiGHS run time
               :
                            0.00
```

Info

The ! after optimize is part of the name. It's nothing special. Julia has a convention that functions which mutate their arguments should end in !. A common example is push!.

Now let's see what information we can query about the solution, starting with is_solved_and_feasible:

```
julia> is_solved_and_feasible(model)
true
```

We can get more information about the solution by querying the three types of statuses.

termination_status tells us why the solver stopped:

```
julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1
```

In this case, the solver found an optimal solution.

Check primal_status to see if the solver found a primal feasible point:

CHAPTER 4. GETTING STARTED

julia> primal_status(model)
FEASIBLE_POINT::ResultStatusCode = 1

and dual_status to see if the solver found a dual feasible point:

julia> dual_status(model)
FEASIBLE POINT::ResultStatusCode = 1

Now we know that our solver found an optimal solution, and that it has a primal and a dual solution to query.

Query the objective value using objective_value:

julia> objective_value(model)
205.0

the primal solution using value:

julia> value(x)
15.00000000000000004

julia> value(y)
1.24999999999999976

and the dual solution using shadow_price:

```
julia> shadow_price(c1)
-0.249999999999999917
```

julia> shadow_price(c2)
-1.5000000000000007

Warning

You should always check whether the solver found a solution before calling solution functions like value or objective_value. A common workflow is:

```
optimize!(model)
if !is_solved_and_feasible(model)
    error("Solver did not find an optimal solution")
end
```

That's it for our simple model. In the rest of this tutorial, we expand on some of the basic JuMP operations.

Model basics

Create a model by passing an optimizer:

```
julia> model = Model(HiGHS.Optimizer)
```

```
A JuMP Model
    solver: HiGHS
    objective_sense: FEASIBILITY_SENSE
    num_variables: 0
    num_constraints: 0
    Names registered in the model: none
```

Alternatively, call set_optimizer at any point before calling optimize!:

```
julia> model = Model()
A JuMP Model
+ solver: none
+ objective_sense: FEASIBILITY_SENSE
+ num_variables: 0
+ num_constraints: 0
L Names registered in the model: none
```

julia> set_optimizer(model, HiGHS.Optimizer)

For some solvers, you can also use direct_model, which offers a more efficient connection to the underlying solver:

```
julia> model = direct_model(HiGHS.Optimizer())
```

```
A JuMP Model
    mode: DIRECT
    solver: HiGHS
    objective_sense: FEASIBILITY_SENSE
    num_variables: 0
    h num_constraints: 0
    L Names registered in the model: none
```

Warning

Some solvers do not support direct_model!

Solver Options

Pass options to solvers with optimizer_with_attributes:

Note

These options are solver-specific. To find out the various options available, see the GitHub README of the individual solver packages. The link to each solver's GitHub page is in the Supported solvers table.

You can also pass options with set_attribute:

```
julia> model = Model(HiGHS.Optimizer)
A JuMP Model
+ solver: HiGHS
+ objective_sense: FEASIBILITY_SENSE
+ num_variables: 0
+ num_constraints: 0
L Names registered in the model: none
```

julia> set_attribute(model, "output_flag", false)

Solution basics

We saw above how to use termination_status and primal_status to understand the solution returned by the solver.

However, only query solution attributes like value and objective_value if there is an available solution. Here's a recommended way to check:

```
julia> function solve_infeasible()
    model = Model(HiGHS.Optimizer)
    @variable(model, 0 <= x <= 1)
    @variable(model, 0 <= y <= 1)
    @constraint(model, x + y >= 3)
    @objective(model, Max, x + 2y)
    optimize!(model)
    if !is_solved_and_feasible(model)
        @warn("The model was not solved correctly.")
        return
    end
    return value(x), value(y)
end
```

solve infeasible (generic function with 1 method)

julia> solve_infeasible()

```
Running HiGHS 1.10.0 (git hash: fd8665394e): Copyright (c) 2025 HiGHS under MIT licence terms

LP has 1 rows; 2 cols; 2 nonzeros

Coefficient ranges:

Matrix [1e+00, 1e+00]

Cost [1e+00, 2e+00]

Bound [1e+00, 1e+00]

RHS [3e+00, 3e+00]

Presolving model

Problem status detected on presolve: Infeasible

Model status : Infeasible

Objective value : 0.000000000e+00

HiGHS run time : 0.00
```

```
Solving LP to try to compute dual ray
LP has 1 rows; 2 cols; 2 nonzeros
Coefficient ranges:
 Matrix [1e+00, 1e+00]
 Cost [0e+00, 0e+00]
 Bound [1e+00, 1e+00]
 RHS [3e+00, 3e+00]
Solving LP without presolve, or with basis, or unconstrained
Using EKK dual simplex solver - serial
Iteration Objective Infeasibilities num(sum)
        0 -0.0000000000e+00 Pr: 1(3) 0s
        2 -1.7393294116e-06 0s
Model status : Infeasible
Simplex iterations: 2
Objective value : 0.000000000e+00
Relative P-D gap : 1.7393294116e-06
HiGHS run time
                           0.00
Solving linear system to compute dual ray
_{\mbox{\scriptsize \Gamma}} Warning: The model was not solved correctly.
L @ Main REPL[1]:9
```

Variable basics

Let's create a new empty model to explain some of the variable syntax:

```
julia> model = Model()
A JuMP Model
| solver: none
| objective_sense: FEASIBILITY_SENSE
| num_variables: 0
| num_constraints: 0
L Names registered in the model: none
```

Variable bounds

All of the variables we have created till now have had a bound. We can also create a free variable.

```
julia> @variable(model, free_x)
free_x
```

While creating a variable, instead of using the <= and >= syntax, we can also use the lower_bound and upper_bound keyword arguments.

```
julia> @variable(model, keyword_x, lower_bound = 1, upper_bound = 2)
keyword_x
```

We can query whether a variable has a bound using the has_lower_bound and has_upper_bound functions. The values of the bound can be obtained using the lower_bound and upper_bound functions.

```
julia> has_upper_bound(keyword_x)
true
julia> upper_bound(keyword_x)
2.0
```

Note querying the value of a bound that does not exist will result in an error.

```
julia> lower_bound(free_x)
Variable free_x does not have a lower bound.
```

Containers

We have already seen how to add a single variable to a model using the <code>@variable</code> macro. Now let's look at ways to add multiple variables to a model.

JuMP provides data structures for adding collections of variables to a model. These data structures are referred to as *containers* and are of three types: Arrays, DenseAxisArrays, and SparseAxisArrays.

Arrays

JuMP arrays are created when you have integer indices that start at 1:

```
julia> @variable(model, a[1:2, 1:2])
2×2 Matrix{VariableRef}:
    a[1,1]    a[1,2]
    a[2,1]    a[2,2]
```

Index elements in a as follows:

julia> a[1, 1]
a[1,1]

```
julia> a[2, :]
2-element Vector{VariableRef}:
a[2,1]
a[2,2]
```

Create an n-dimensional variable $x \in R^n$ with bounds $l \leq x \leq u$ ($l, u \in R^n$) as follows:

julia> n = 10
10
julia> l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
julia> u = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19];
julia> @variable(model, l[i] <= x[i = 1:n] <= u[i])</pre>

10-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
 x[4]
 x[5]
 x[6]
 x[7]
 x[8]
 x[9]
 x[10]

We can also create variable bounds that depend upon the indices:

```
julia> @variable(model, y[i = 1:2, j = 1:2] >= 2i + j)
2×2 Matrix{VariableRef}:
    y[1,1]    y[1,2]
    y[2,1]    y[2,2]
```

DenseAxisArrays

DenseAxisArrays are used when the indices are not one-based integer ranges. The syntax is similar except with an arbitrary vector as an index as opposed to a one-based range:

```
julia> @variable(model, z[i = 2:3, j = 1:2:3] >= 0)
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, 2:3
    Dimension 2, 1:2:3
And data, a 2×2 Matrix{VariableRef}:
    z[2,1] z[2,3]
    z[3,1] z[3,3]
```

Indices do not have to be integers. They can be any Julia type:

```
julia> @variable(model, w[1:5, ["red", "blue"]] <= 1)
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, Base.OneTo(5)
    Dimension 2, ["red", "blue"]
And data, a 5×2 Matrix{VariableRef}:
    w[1,red] w[1,blue]
    w[2,red] w[2,blue]
    w[3,red] w[3,blue]
    w[4,red] w[4,blue]
    w[5,red] w[5,blue]</pre>
```

Index elements in a DenseAxisArray as follows:

julia> z[2, 1] z[2,1]

```
julia> w[2:3, ["red", "blue"]]
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, [2, 3]
    Dimension 2, ["red", "blue"]
And data, a 2×2 Matrix{VariableRef}:
    w[2,red] w[2,blue]
    w[3,red] w[3,blue]
```

See Forcing the container type for more details.

SparseAxisArrays

SparseAxisArrays are created when the indices do not form a Cartesian product. For example, this applies when indices have a dependence upon previous indices (called triangular indexing):

```
julia> @variable(model, u[i = 1:2, j = i:3])
SparseAxisArray{VariableRef, 2, Tuple{Int64, Int64}} with 5 entries:
  [1, 1] = u[1,1]
  [1, 2] = u[1,2]
  [1, 3] = u[1,3]
  [2, 2] = u[2,2]
  [2, 3] = u[2,3]
```

We can also conditionally create variables by adding a comparison check that depends upon the named indices and is separated from the indices by a semi-colon ;:

```
julia> @variable(model, v[i = 1:9; mod(i, 3) == 0])
SparseAxisArray{VariableRef, 1, Tuple{Int64}} with 3 entries:
[3] = v[3]
[6] = v[6]
[9] = v[9]
```

Index elements in a DenseAxisArray as follows:

```
julia> u[1, 2]
u[1,2]
```

```
julia> v[[3, 6]]
SparseAxisArray{VariableRef, 1, Tuple{Int64}} with 2 entries:
  [3] = v[3]
  [6] = v[6]
```

Integrality

JuMP can create binary and integer variables. Binary variables are constrained to the set $\{0, 1\}$, and integer variables are constrained to the set \mathbb{Z} .

Integer variables

Create an integer variable by passing Int:

```
julia> @variable(model, integer_x, Int)
integer_x
```

or setting the integer keyword to true:

```
julia> @variable(model, integer_z, integer = true)
integer_z
```

Binary variables

Create a binary variable by passing Bin:

julia> @variable(model, binary_x, Bin)
binary_x

or setting the binary keyword to true:

```
julia> @variable(model, binary_z, binary = true)
binary_z
```

Constraint basics

We'll need a new model to explain some of the constraint basics:

```
julia> model = Model()
A JuMP Model
+ solver: none
+ objective_sense: FEASIBILITY_SENSE
+ num_variables: 0
+ num_constraints: 0
L Names registered in the model: none
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> @variable(model, z[1:10]);
```

Containers

Just as we had containers for variables, JuMP also provides Arrays, DenseAxisArrays, and SparseAxisArrays for storing collections of constraints. Examples for each container type are given below.

Arrays

Create an Array of constraints:

DenseAxisArrays

Create an DenseAxisArray of constraints:

```
julia> @constraint(model, [i = 1:2, j = 2:3], i * x <= j + 1)
2-dimensional DenseAxisArray{ConstraintRef{Model,

→ MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

→ MathOptInterface.LessThan{Float64}, ScalarShape},2,...} with index sets:

Dimension 1, Base.OneTo(2)

Dimension 2, 2:3

And data, a 2×2 Matrix{ConstraintRef{Model,

→ MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

→ MathOptInterface.LessThan{Float64}, ScalarShape}}:

x ≤ 3 x ≤ 4

2 x ≤ 3 2 x ≤ 4
```

SparseAxisArrays

Create an SparseAxisArray of constraints:

Constraints in a loop

We can add constraints using regular Julia loops:

```
julia> for i in 1:3
    @constraint(model, 6x + 4y >= 5i)
    end
```

or use for each loops inside the @constraint macro:

We can also create constraints such as $\sum_{i=1}^{10} z_i \leq 1$:

```
julia> @constraint(model, sum(z[i] for i in 1:10) <= 1)
z[1] + z[2] + z[3] + z[4] + z[5] + z[6] + z[7] + z[8] + z[9] + z[10] \leq 1
```

Objective functions

Set an objective function with @objective:

```
julia> model = Model(HiGHS.Optimizer)
A JuMP Model
+ solver: HiGHS
+ objective_sense: FEASIBILITY_SENSE
+ num_variables: 0
+ num_constraints: 0
L Names registered in the model: none
julia> @variable(model, x >= 0)
x
julia> @variable(model, y >= 0)
y
julia> @objective(model, Min, 2x + y)
2 x + y
```

Create a maximization objective using Max:

```
julia> @objective(model, Max, 2x + y)
2 x + y
```

Тір

Calling <code>@objective</code> multiple times will over-write the previous objective. This can be useful when you want to solve the same problem with different objectives.

Vectorized syntax

We can also add constraints and an objective to JuMP using vectorized linear algebra. We'll illustrate this by solving an LP in standard form that is,

```
\begin{array}{ll} \min & c^T x \\ \text{s.t.} & Ax = b \\ & x \ge 0 \end{array}
```

```
julia> vector_model = Model(HiGHS.Optimizer)
A JuMP Model
- solver: HiGHS
b objective_sense: FEASIBILITY_SENSE
- num_variables: 0
hum_constraints: 0
<sup>L</sup> Names registered in the model: none
julia> A = [1 1 9 5; 3 5 0 8; 2 0 6 13]
3×4 Matrix{Int64}:
1 1 9 5
3 5 0 8
2 0 6 13
julia> b = [7, 3, 5]
3-element Vector{Int64}:
7
3
5
julia> c = [1, 3, 5, 2]
4-element Vector{Int64}:
1
3
5
2
julia> @variable(vector_model, x[1:4] >= 0)
4-element Vector{VariableRef}:
x[1]
x[2]
x[3]
x[4]
julia> @constraint(vector_model, A * x .== b)
3-element Vector{ConstraintRef{Model,
\hookrightarrow \mbox{MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64}, }
→ MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
x[1] + x[2] + 9 x[3] + 5 x[4] = 7
3 \times [1] + 5 \times [2] + 8 \times [4] = 3
2 \times [1] + 6 \times [3] + 13 \times [4] = 5
julia> @objective(vector_model, Min, c' * x)
x[1] + 3 x[2] + 5 x[3] + 2 x[4]
```

julia> optimize!(vector_model)

```
Running HiGHS 1.10.0 (git hash: fd8665394e): Copyright (c) 2025 HiGHS under MIT licence terms LP has 3 rows; 4 cols; 10 nonzeros Coefficient ranges:
```

```
Matrix [1e+00, 1e+01]
 Cost [1e+00, 5e+00]
 Bound [0e+00, 0e+00]
 RHS [3e+00, 7e+00]
Presolving model
3 rows, 4 cols, 10 nonzeros Os
Dependent equations search running on 3 equations with time limit of 1000.00s
Dependent equations search removed 0 rows and 0 nonzeros in 0.00s (limit = 1000.00s)
3 rows, 4 cols, 10 nonzeros Os
Presolve : Reductions: rows 3(-0); columns 4(-0); elements 10(-0) - Not reduced
Problem not reduced by presolve: solving the LP
Using EKK dual simplex solver - serial
 Iteration
              Objective
                             Infeasibilities num(sum)
       0 0.000000000e+00 Pr: 3(13.5) 0s
        4 4.9230769231e+00 Pr: 0(0) 0s
Model status
               : Optimal
Simplex iterations: 4
Objective value : 4.9230769231e+00
Relative P-D gap : 0.000000000e+00
HiGHS run time
                 :
                            0.00
```

```
julia> assert_is_solved_and_feasible(vector_model)
```

```
julia> objective_value(vector_model)
4.923076923076922
```

4.4 Getting started with sets and indexing

This tutorial was generated using Literate.jl. Download the source as a .jl file.

Most introductory courses to linear programming will teach you to identify sets over which the decision variables and constraints are indexed. Therefore, it is common to write variables such as x_i for all $i \in I$.

A common stumbling block for new users to JuMP is that JuMP does not provide specialized syntax for constructing and manipulating these sets.

We made this decision because Julia already provides a wealth of data structures for working with sets.

In contrast, because tools like AMPL are stand-alone software packages, they had to define their own syntax for set construction and manipulation. Indeed, the AMPL Book has two entire chapters devoted to sets and indexing (Chapter 5, "Simple Sets and Indexing," and Chapter 6, "Compound Sets and Indexing").

The purpose of this tutorial is to demonstrate a variety of ways in which you can construct and manipulate sets for optimization models.

If you haven't already, you should first read Getting started with JuMP.

using JuMP

Unordered sets

Unordered sets are useful to describe non-numeric indices, such as the names of cities or types of products.

The most common way to construct a set is by creating a vector:

```
animals = ["dog", "cat", "chicken", "cow", "pig"]
model = Model()
@variable(model, x[animals])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ["dog", "cat", "chicken", "cow", "pig"]
And data, a 5-element Vector{VariableRef}:
    x[dog]
    x[cat]
    x[chicken]
    x[cow]
    x[pig]
```

We can also use things like the keys of a dictionary:

```
weight_of_animals = Dict(
    "dog" => 20.0,
    "cat" => 5.0,
    "chicken" => 2.0,
    "cow" => 720.0,
    "pig" => 150.0,
)
animal_keys = keys(weight_of_animals)
model = Model()
@variable(model, x[animal_keys])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ["cow", "chicken", "cat", "pig", "dog"]
And data, a 5-element Vector{VariableRef}:
    x[cow]
    x[chicken]
    x[cat]
    x[pig]
    x[dog]
```

A third option is to use Julia's Set object.

```
animal_set = Set()
for animal in keys(weight_of_animals)
        push!(animal_set, animal)
end
animal_set
```

Set{Any} with 5 elements:
 "cow"
 "chicken"

"cat" "pig" "dog"

The nice thing about Sets is that they automatically remove duplicates:

push!(animal_set, "dog")
animal_set

```
Set{Any} with 5 elements:
    "cow"
    "chicken"
    "cat"
    "pig"
    "dog"
```

Note how dog does not appear twice.

model = Model()
@variable(model, x[animal_set])

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ["cow", "chicken", "cat", "pig", "dog"]
And data, a 5-element Vector{VariableRef}:
    x[cow]
    x[chicken]
    x[cat]
    x[pig]
    x[dog]
```

Sets of numbers

Sets of numbers are useful to describe sets that are ordered, such as years or elements in a vector. The easiest way to create sets of numbers is to use Julia's range syntax.

These can start at 1:

model = Model()
@variable(model, x[1:4])

4-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
 x[4]

but they don't have to:

model = Model()
@variable(model, x[2012:2021])

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, 2012:2021
And data, a 10-element Vector{VariableRef}:
    x[2012]
    x[2013]
    x[2014]
    x[2015]
    x[2016]
    x[2017]
    x[2018]
    x[2019]
    x[2020]
    x[2021]
```

Ranges also have a start:step:stop syntax. So the Olympic years are:

model = Model()
@variable(model, x[1896:4:2020])

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
   Dimension 1, 1896:4:2020
And data, a 32-element Vector{VariableRef}:
x[1896]
x[1900]
x[1904]
x[1908]
x[1912]
x[1916]
x[1920]
x[1924]
x[1928]
x[1932]
x[1988]
x[1992]
x[1996]
x[2000]
x[2004]
x[2008]
x[2012]
x[2016]
x[2020]
```

Sets of other things

An important observation is that you can have *any* Julia type as the element of a set. It doesn't have to be a String or a Number. For example, you can have tuples:

```
sources = ["A", "B", "C"]
sinks = ["D", "E"]
S = [(source, sink) for source in sources, sink in sinks]
model = Model()
@variable(model, x[S])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, [("A", "D"), ("B", "D"), ("C", "D"), ("A", "E"), ("B", "E"), ("C", "E")]
And data, a 6-element Vector{VariableRef}:
    x[("A", "D")]
    x[("B", "D")]
    x[("C", "D")]
    x[("C", "E")]
    x[("B", "E")]
    x[("C", "E")]
```

```
x[("A", "D")]
```

 $x_{("A","D")}$

For multi-dimensional sets, you can use JuMP's syntax for constructing Containers:

model = Model()
@variable(model, x[sources, sinks])

```
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, ["A", "B", "C"]
    Dimension 2, ["D", "E"]
And data, a 3×2 Matrix{VariableRef}:
    x[A,D] x[A,E]
    x[B,D] x[B,E]
    x[C,D] x[C,E]
```

x["A", "D"]

Info

 $x_{A,D}$

Note how we indexed x["A", "D"] instead of x[("A", "D")] as above.

Sets to watch out for

JuMP supports any sets which are iterable, that is, the set set supports a for-loop like: [i for i in set]. This causes a few common errors.

First, if T = 3, you may pass the integer T by mistake instead of a range like 1:T:

```
model = Model()
T = 3
@variable(model, x[T])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, [3]
And data, a 1-element Vector{VariableRef}:
    x[3]
```

This results in a single variable being created, instead of three as desired. Because this is a common error, a warning is printed, advising you to pass a Vector{Int} instead:

@variable(model, x_fixed[[T]])

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, [3]
And data, a 1-element Vector{VariableRef}:
    x_fixed[3]
```

Second, because Strings are iterable, passing a "index" as a singleton index is the same as passing ['i', 'n', 'd', 'e', 'x']:

@variable(model, y["index"])

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ['i', 'n', 'd', 'e', 'x']
And data, a 5-element Vector{VariableRef}:
    y[i]
    y[n]
    y[d]
    y[d]
    y[e]
    y[x]
```

This time, a warning is not printed, but the work-around is similar, pass a Vector{String} instead:

@variable(model, y_fixed[["index"]])

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ["index"]
And data, a 1-element Vector{VariableRef}:
    y_fixed[index]
```

Тір

As a rule of thumb, if you want an index with one element, avoid confusion by passing [index] instead of index.

Set operations

Julia has built-in support for set operations such as union, intersect, and setdiff.

Therefore, to create a set of all years in which the summer Olympics were held, we can use:

```
baseline = 1896:4:2020
cancelled = [1916, 1940, 1944, 2020]
off_year = [2021]
olympic_years = union(setdiff(baseline, cancelled), off_year)
```

29-element	Vector{Int64}:
1896	
1900	
1904	
1908	
1912	
1920	
1924	
1928	
1932	
1936	
1988	
1992	
1996	
2000	
2004	
2008	
2012	
2016	
2021	

You can also find the number of elements (that is, the cardinality) in a set using length:

length(olympic_years)

29

Set membership operations

To compute membership of sets, use the in function.

2000 in olympic_years

true

2001 in olympic_years

false

Indexing expressions

Use Julia's generator syntax to compute new sets, such as the list of Olympic years that are divisible by 3:

```
olympic_3_years = [year for year in olympic_years if mod(year, 3) == 0]
model = Model()
@variable(model, x[olympic_3_years])
```

<pre>1-dimensional DenseAxisArray{VariableRef,1,} with index sets:</pre>
Dimension 1, [1896, 1908, 1920, 1932, 1956, 1968, 1980, 1992, 2004, 2016]
And data, a 10-element Vector{VariableRef}:
x[1896]
x[1908]
x[1920]
x[1932]
x[1956]
x[1968]
x[1980]
x[1992]
x[2004]
x[2016]

Alternatively, use JuMP's syntax for constructing Containers:

```
model = Model()
@variable(model, x[year in olympic_years; mod(year, 3) == 0])
```

Compound sets

Consider a transportation problem in which we need to ship goods between cities. We have been provided a list of cities:

cities = ["Auckland", "Wellington", "Christchurch", "Dunedin"]

```
4-element Vector{String}:
   "Auckland"
   "Wellington"
   "Christchurch"
   "Dunedin"
```

and a distance matrix which records the shipping distance between pairs of cities. If we can't ship between two cities, the distance is 0.

distances = [0 643 1071 1426; 0 0 436 790; 0 0 0 360; 1426 0 0 0]

4×4 Matrix{Int64}: 0 643 1071 1426 0 0 436 790 0 0 360 1426 0 0 0

Let's have a look at ways we could write a model with an objective function to minimize the total shipping cost. For simplicity, we'll ignore all constraints.

Fix unused variables

One approach is to fix all variables that we can't use to zero. Most solvers are smart-enough to remove these during a presolve phase, so it has a very small impact on performance:

```
N = length(cities)
model = Model()
@variable(model, x[1:N, 1:N] >= 0)
for i in 1:N, j in 1:N
    if distances[i, j] == 0
        fix(x[i, j], 0.0; force = true)
    end
end
@objective(model, Min, sum(distances[i, j] * x[i, j] for i in 1:N, j in 1:N))
```

 $643x_{1,2} + 1071x_{1,3} + 1426x_{1,4} + 436x_{2,3} + 790x_{2,4} + 360x_{3,4} + 1426x_{4,1}$

Filtered summation

Another approach is to define filters whenever we want to sum over our decision variables:

```
N = length(cities)
model = Model()
@variable(model, x[1:N, 1:N] >= 0)
@objective(
    model,
    Min,
    sum(
        distances[i, j] * x[i, j] for i in 1:N, j in 1:N if distances[i, j] > 0
    ),
)
```

 $643x_{1,2} + 1071x_{1,3} + 1426x_{1,4} + 436x_{2,3} + 790x_{2,4} + 360x_{3,4} + 1426x_{4,1}$

Filtered indexing

We could also use JuMP's support for Containers:

```
N = length(cities)
model = Model()
@variable(model, x[i = 1:N, j = 1:N; distances[i, j] > 0])
@objective(model, Min, sum(distances[i...] * x[i] for i in eachindex(x)))
```

 $643x_{1,2} + 1071x_{1,3} + 1426x_{1,4} + 436x_{2,3} + 790x_{2,4} + 360x_{3,4} + 1426x_{4,1}$

Note

The i... is called a "splat." It converts a tuple like (1, 2) into two indices like distances [1, 2].

Converting to a different data structure

Another approach, and one that is often the most readable, is to convert the data you have into something that is easier to work with. Originally, we had a vector of strings and a matrix of distances. What we really need is something that maps usable origin-destination pairs to distances. A dictionary is an obvious choice:

```
routes = Dict(
  (a, b) => distances[i, j] for
  (i, a) in enumerate(cities), (j, b) in enumerate(cities) if
  distances[i, j] > 0
)
```

```
Dict{Tuple{String, String}, Int64} with 7 entries:
  ("Auckland", "Wellington") => 643
  ("Wellington", "Christchurch") => 436
  ("Wellington", "Dunedin") => 790
  ("Christchurch", "Dunedin") => 360
  ("Auckland", "Dunedin") => 1426
  ("Dunedin", "Auckland") => 1426
  ("Auckland", "Christchurch") => 1071
```

Then, we can create our model like so:

```
model = Model()
@variable(model, x[keys(routes)])
@objective(model, Min, sum(v * x[k] for (k, v) in routes))
```

 $\frac{643x("Auckland","Wellington") + 436x("Wellington","Christchurch") + 790x("Wellington","Dunedin") + 360x("Christchurch","Dunedin") + 1426x("Auckland","Dunedin") + 1071x("Auckland","Christchurch") + 1071x("Auckland","Christc$

This has a number of benefits over the other approaches, including a compacter algebraic model and variables that are named in a more meaningful way.

Tip

If you're struggling to formulate a problem using the available syntax in JuMP, it's probably a sign that you should convert your data into a different form.

Next steps

The purpose of this tutorial was to show how JuMP does not have specialized syntax for set creation and manipulation. Instead, you should use the tools provided by Julia itself.

This is both an opportunity and a challenge, because you are free to pick the syntax and data structures that best suit your problem, but for new users it can be daunting to decide which structure to use.

Read through some of the other JuMP tutorials to get inspiration and ideas for how you can use Julia's syntax and data structures to your advantage.

4.5 Getting started with data and plotting

This tutorial was generated using Literate.jl. Download the source as a .jl file.

In this tutorial we will learn how to read tabular data into Julia, and some of the basics of plotting.

If you're new to Julia, start by reading Getting started with Julia and Getting started with JuMP first.

Note

There are multiple ways to read the same kind of data into Julia. This tutorial focuses on DataFrames.jl because it provides the ecosystem to work with most of the required file types in a straightforward manner.

Before we get started, we need this constant to point to where the data files are.

```
import JuMP
const DATA_DIR = joinpath(
    dirname(pathof(JuMP)),
    joinpath("..", "docs", "src", "tutorials", "getting_started", "data"),
);
```

Where to get help

Read the documentation

- Plots.jl: http://docs.juliaplots.org/latest/
- CSV.jl: http://csv.juliadata.org/stable
- DataFrames.jl: https://dataframes.juliadata.org/stable/

Preliminaries

To get started, we need to install some packages.

DataFrames.jl

The DataFrames package provides a set of tools for working with tabular data. It is available through the Julia package manager.

using Pkg
Pkg.add("DataFrames")

import DataFrames

What is a DataFrame?

A DataFrame is a data structure like a table or spreadsheet. You can use it for storing and exploring a set of related data values. Think of it as a smarter array for holding tabular data.

Plots.jl

The Plots package provides a set of tools for plotting. It is available through the Julia package manager.

CHAPTER 4. GETTING STARTED

using Pkg
Pkg.add("Plots")

import Plots

CSV .jl

CSV and other delimited text files can be read by the CSV.jl package.

Pkg.add("CSV")

import CSV

DataFrame basics

To read a CSV file into a DataFrame, we use the CSV. read function.

csv_df = CSV.read(joinpath(DATA_DIR, "StarWars.csv"), DataFrames.DataFrame)

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died	Jedi	Species	Weapon
	String31	String7	Float64	String7	String15	String7	String15	String15	String15	String15	String7	String15	String15
1	Anakin Skywalker	male	1.88	84	blue	blond	fair	Tatooine	41.9BBY	4ABY	jedi	human	lightsaber
2	Padme Amidala	female	1.65	45	brown	brown	light	Naboo	46BBY	19BBY	no_jedi	human	unarmed
3	Luke Skywalker	male	1.72	77	blue	blond	fair	Tatooine	19BBY	unk_died	jedi	human	lightsaber
4	Leia Skywalker	female	1.5	49	brown	brown	light	Alderaan	19BBY	unk_died	no_jedi	human	blaster
5	Qui-Gon Jinn	male	1.93	88.5	blue	brown	light	unk_planet	92BBY	32BBY	jedi	human	lightsaber
6	Obi-Wan Kenobi	male	1.82	77	bluegray	auburn	fair	Stewjon	57BBY	0BBY	jedi	human	lightsaber
7	Han Solo	male	1.8	80	brown	brown	light	Corellia	29BBY	unk_died	no_jedi	human	blaster
8	Sheev Palpatine	male	1.73	75	blue	red	pale	Naboo	82BBY	10ABY	no_jedi	human	force-lightning
9	R2-D2	male	0.96	32	NA	NA	NA	Naboo	33BBY	unk_died	no_jedi	droid	unarmed
10	C-3PO	male	1.67	75	NA	NA	NA	Tatooine	112BBY	3ABY	no_jedi	droid	unarmed
11	Yoda	male	0.66	17	brown	brown	green	unk_planet	896BBY	4ABY	jedi	yoda	lightsaber
12	Darth Maul	male	1.75	80	yellow	none	red	Dathomir	54BBY	unk_died	no_jedi	dathomirian	lightsaber
13	Dooku	male	1.93	86	brown	brown	light	Serenno	102BBY	19BBY	jedi	human	lightsaber
14	Chewbacca	male	2.28	112	blue	brown	NA	Kashyyyk	200BBY	25ABY	no_jedi	wookiee	bowcaster
15	Jabba	male	3.9	NA	yellow	none	tan-green	Tatooine	unk_born	4ABY	no_jedi	hutt	unarmed
16	Lando Calrissian	male	1.78	79	brown	blank	dark	Socorro	31BBY	unk_died	no_jedi	human	blaster
17	Boba Fett	male	1.83	78	brown	black	brown	Kamino	31.5BBY	unk_died	no_jedi	human	blaster
18	Jango Fett	male	1.83	79	brown	black	brown	ConcordDawn	66BBY	22BBY	no_jedi	human	blaster
19	Grievous	male	2.16	159	gold	black	orange	Kalee	unk_born	19BBY	no_jedi	kaleesh	slugthrower
20	Chief Chirpa	male	1.0	50	black	gray	brown	Endor	unk_born	4ABY	no_jedi	ewok	spear

Let's try plotting some of this data

```
Plots.scatter(
    csv_df.Weight,
    csv_df.Height;
    xlabel = "Weight",
    ylabel = "Height",
)
```



That doesn't look right. What happened? If you look at the dataframe above, it read Weight in as a String column because there are "NA" fields. Let's correct that, by telling CSV to consider "NA" as missing.

```
csv_df = CSV.read(
    joinpath(DATA_DIR, "StarWars.csv"),
    DataFrames.DataFrame;
    missingstring = "NA",
)
```

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died	Jedi	Species	Weapon
	String31	String7	Float64	Float64?	String15?	String7?	String15?	String15	String15	String15	String7	String15	String15
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY	jedi	human	lightsaber
2	Padme Amidala	female	1.65	45.0	brown	brown	light	Naboo	46BBY	19BBY	no_jedi	human	unarmed
3	Luke Skywalker	male	1.72	77.0	blue	blond	fair	Tatooine	19BBY	unk_died	jedi	human	lightsaber
4	Leia Skywalker	female	1.5	49.0	brown	brown	light	Alderaan	19BBY	unk_died	no_jedi	human	blaster
5	Qui-Gon Jinn	male	1.93	88.5	blue	brown	light	unk_planet	92BBY	32BBY	jedi	human	lightsaber
6	Obi-Wan Kenobi	male	1.82	77.0	bluegray	auburn	fair	Stewjon	57BBY	0BBY	jedi	human	lightsaber
7	Han Solo	male	1.8	80.0	brown	brown	light	Corellia	29BBY	unk_died	no_jedi	human	blaster
8	Sheev Palpatine	male	1.73	75.0	blue	red	pale	Naboo	82BBY	10ABY	no_jedi	human	force-lightning
9	R2-D2	male	0.96	32.0	missing	missing	missing	Naboo	33BBY	unk_died	no_jedi	droid	unarmed
10	C-3PO	male	1.67	75.0	missing	missing	missing	Tatooine	112BBY	3ABY	no_jedi	droid	unarmed
11	Yoda	male	0.66	17.0	brown	brown	green	unk_planet	896BBY	4ABY	jedi	yoda	lightsaber
12	Darth Maul	male	1.75	80.0	yellow	none	red	Dathomir	54BBY	unk_died	no_jedi	dathomirian	lightsaber
13	Dooku	male	1.93	86.0	brown	brown	light	Serenno	102BBY	19BBY	jedi	human	lightsaber
14	Chewbacca	male	2.28	112.0	blue	brown	missing	Kashyyyk	200BBY	25ABY	no_jedi	wookiee	bowcaster
15	Jabba	male	3.9	missing	yellow	none	tan-green	Tatooine	unk_born	4ABY	no_jedi	hutt	unarmed
16	Lando Calrissian	male	1.78	79.0	brown	blank	dark	Socorro	31BBY	unk_died	no_jedi	human	blaster
17	Boba Fett	male	1.83	78.0	brown	black	brown	Kamino	31.5BBY	unk_died	no_jedi	human	blaster
18	Jango Fett	male	1.83	79.0	brown	black	brown	ConcordDawn	66BBY	22BBY	no_jedi	human	blaster
19	Grievous	male	2.16	159.0	gold	black	orange	Kalee	unk_born	19BBY	no_jedi	kaleesh	slugthrower
20	Chief Chirpa	male	1.0	50.0	black	gray	brown	Endor	unk_born	4ABY	no_jedi	ewok	spear

Then let's re-plot our data

Plots.scatter(
csv_df.Wei	ght,				
csv_df.Hei	ght;				
title = "H	eight vs	Weight	of	StarWars	characters",
xlabel = "	Weight",				
ylabel = "	Height",				
label = fa	lse,				
ylims = (0	, 3),				
)					



That looks better.

Тір

Read the CSV documentation for other parsing options.

DataFrames.jl supports manipulation using functions similar to pandas. For example, split the dataframe into groups based on eye-color:

by_eyecolor = DataFrames.groupby(csv_df, :Eyecolor)

GroupedDataFrame with 7 groups based on key: Eyecolor

First Group (5 rows): Eyecolor = "blue"
	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Die
	String31	String7	Float64	Float64?	String15?	String7?	String15?	String15	String15	String
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	4AB
2	Luke Skywalker	male	1.72	77.0	blue	blond	fair	Tatooine	19BBY	unk_d
3	Qui-Gon Jinn	male	1.93	88.5	blue	brown	light	unk_planet	92BBY	32BI
4	Sheev Palpatine	male	1.73	75.0	blue	red	pale	Naboo	82BBY	10AI
5	Chewbacca	male	2.28	112.0	blue	brown	missing	Kashyyyk	200BBY	25AI

Last Group (1 row): Eyecolor = "black"

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died	
	String31	String7	Float64	Float64?	String15?	String7?	String15?	String15	String15	String15	
1	Chief Chirpa	male	1.0	50.0	black	gray	brown	Endor	unk_born	4ABY	

Then recombine into a single dataframe based on a function operating over the split dataframes:

```
eyecolor_count = DataFrames.combine(by_eyecolor) do df
    return DataFrames.nrow(df)
```

end

	Eyecolor	x1
	String15?	Int64
1	blue	5
2	brown	8
3	bluegray	1
4	missing	2
5	yellow	2
6	gold	1
7	black	1

We can rename columns:

DataFrames.rename!(eyecolor_count, :x1 => :count)

	Eyecolor	count
	String15?	Int64
1	blue	5
2	brown	8
3	bluegray	1
4	missing	2
5	yellow	2
6	gold	1
7	black	1

Drop some missing rows:

DataFrames.dropmissing!(eyecolor_count, :Eyecolor)

Then we can visualize the data:

	Eyecolor	count
	String15	Int64
1	blue	5
2	brown	8
3	bluegray	1
4	yellow	2
5	gold	1
6	black	1

```
sort!(eyecolor_count, :count; rev = true)
Plots.bar(
    eyecolor_count.Eyecolor,
    eyecolor_count.count;
    xlabel = "Eye color",
    ylabel = "Number of characters",
    label = false,
)
```



Other Delimited Files

We can also use the CSV.jl package to read any other delimited text file format.

By default, CSV.File will try to detect a file's delimiter from the first 10 lines of the file.

Candidate delimiters include ',', '\t', ' ', '|', ';', and ':'. If it can't auto-detect the delimiter, it will assume ','.

Let's take the example of space separated data.

	Name	Cups	Calories	Carbs	Fat	Fiber	Potassium	Protein	Sodium	Sugars
	String31	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
1	CapnCrunch	0.75	120	12.0	2	0.0	35	1	220	12
2	CocoaPuffs	1.0	110	12.0	1	0.0	55	1	180	13
3	Trix	1.0	110	13.0	1	0.0	25	1	140	12
4	AppleJacks	1.0	110	11.0	0	1.0	30	2	125	14
5	CornChex	1.0	110	22.0	0	0.0	25	2	280	3
6	CornFlakes	1.0	100	21.0	0	1.0	35	2	290	2
7	Nut&Honey	0.67	120	15.0	1	0.0	40	2	190	9
8	Smacks	0.75	110	9.0	1	1.0	40	2	70	15
9	MultiGrain	1.0	100	15.0	1	2.0	90	2	220	6
10	CracklinOat	0.5	110	10.0	3	4.0	160	3	140	7
11	GrapeNuts	0.25	110	17.0	0	3.0	90	3	179	3
12	HoneyNutCheerios	0.75	110	11.5	1	1.5	90	3	250	10
13	NutriGrain	0.67	140	21.0	2	3.0	130	3	220	7
14	Product19	1.0	100	20.0	0	1.0	45	3	320	3
15	TotalRaisinBran	1.0	140	15.0	1	4.0	230	3	190	14
16	WheatChex	0.67	100	17.0	1	3.0	115	3	230	3
17	Oatmeal	0.5	130	13.5	2	1.5	120	3	170	10
18	Life	0.67	100	12.0	2	2.0	95	4	150	6
19	Мауро	1.0	100	16.0	1	0.0	95	4	0	3
20	QuakerOats	0.5	100	14.0	1	2.0	110	4	135	6
21	Muesli	1.0	150	16.0	3	3.0	170	4	150	11
22	Cheerios	1.25	110	17.0	2	2.0	105	6	290	1
23	SpecialK	1.0	110	16.0	0	1.0	55	6	230	3

ss_df =	CSV.read(joinpath(DATA_	DIR,	"Cereal.txt"),	DataFrames.DataFrame)
---------	-------------------------	------	----------------	-----------------------

We can also specify the delimiter as follows:

```
delim_df = CSV.read(
    joinpath(DATA_DIR, "Soccer.txt"),
    DataFrames.DataFrame;
    delim = "::",
)
```

Working with DataFrames

Now that we have read the required data into a DataFrame, let us look at some basic operations we can perform on it.

Querying Basic Information

The size function gets us the dimensions of the DataFrame:

DataFrames.size(ss_df)

(23, 10)

	Team	Played	Wins	Draws	Losses	Goals_for	Goals_against
	String31	Int64	Int64	Int64	Int64	String15	String15
1	Barcelona	38	30	4	4	110 goals	21 goals
2	Real Madrid	38	30	2	6	118 goals	38 goals
3	Atletico Madrid	38	23	9	6	67 goals	29 goals
4	Valencia	38	22	11	5	70 goals	32 goals
5	Seville	38	23	7	8	71 goals	45 goals
6	Villarreal	38	16	12	10	48 goals	37 goals
7	Athletic Bilbao	38	15	10	13	42 goals	41 goals
8	Celta Vigo	38	13	12	13	47 goals	44 goals
9	Malaga	38	14	8	16	42 goals	48 goals
10	Espanyol	38	13	10	15	47 goals	51 goals
11	Rayo Vallecano	38	15	4	19	46 goals	68 goals
12	Real Sociedad	38	11	13	14	44 goals	51 goals
13	Elche	38	11	8	19	35 goals	62 goals
14	Levante	38	9	10	19	34 goals	67 goals
15	Getafe	38	10	7	21	33 goals	64 goals
16	Deportivo La Coruna	38	7	14	17	35 goals	60 goals
17	Granada	38	7	14	17	29 goals	64 goals
18	Eibar	38	9	8	21	34 goals	55 goals
19	Almeria	38	8	8	22	35 goals	64 goals
20	Cordoba	38	3	11	24	22 goals	68 goals

We can also use the nrow and ncol functions to get the number of rows and columns respectively:

DataFrames.nrow(ss_df), DataFrames.ncol(ss_df)

(23, 10)

The describe function gives basic summary statistics of data in a DataFrame:

DataFrames.describe(ss_df)

	variable	mean	min	median	max	nmissing	eltype
	Symbol	Union	Any	Union	Any	Int64	DataType
1	Name		AppleJacks		WheatChex	0	String31
2	Cups	0.823043	0.25	1.0	1.25	0	Float64
3	Calories	113.043	100	110.0	150	0	Int64
4	Carbs	15.0435	9.0	15.0	22.0	0	Float64
5	Fat	1.13043	0	1.0	3	0	Int64
6	Fiber	1.56522	0.0	1.5	4.0	0	Float64
7	Potassium	86.3043	25	90.0	230	0	Int64
8	Protein	2.91304	1	3.0	6	0	Int64
9	Sodium	189.957	0	190.0	320	0	Int64
10	Sugars	7.52174	1	7.0	15	0	Int64

Names of every column can be obtained by the names function:

DataFrames.names(ss_df)

```
10-element Vector{String}:
    "Name"
    "Cups"
    "Calories"
    "Carbs"
    "Fat"
    "Fiber"
    "Potassium"
    "Protein"
    "Sodium"
    "Sugars"
```

Corresponding data types are obtained using the broadcasted eltype function:

eltype.(ss_df)

	Name	Cups	Calories	Carbs	Fat	Fiber	Potassium	Protein	Sodium	Sugars
	DataType	DataType	DataType	DataType						
1	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
2	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
3	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
4	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
5	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
6	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
7	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
8	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
9	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
10	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
11	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
12	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
13	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
14	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
15	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
16	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
17	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
18	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
19	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
20	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
21	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
22	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
23	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64

Accessing the Data

Similar to regular arrays, we use numerical indexing to access elements of a DataFrame:

csv_df[1, 1]

"Anakin Skywalker"

The following are different ways to access a column:

csv_df[!, 1]

<pre>20-element Vector{InlineStrings.String31}:</pre>
"Anakin Skywalker"
"Padme Amidala"
"Luke Skywalker"
"Leia Skywalker"
"Qui-Gon Jinn"
"Obi-Wan Kenobi"
"Han Solo"
"Sheev Palpatine"
"R2-D2"
"C - 3P0"
"Yoda"
"Darth Maul"
"Dooku"
"Chewbacca"
"Jabba"
"Lando Calrissian"
"Boba Fett"
"Jango Fett"
"Grievous"
"Chief Chirpa"

csv_df[!, :Name]

20-element Vector{InlineStrings.String31}: "Anakin Skywalker" "Padme Amidala" "Luke Skywalker" "Leia Skywalker" "Qui-Gon Jinn" "Obi-Wan Kenobi" "Han Solo" "Sheev Palpatine" "R2-D2" "C-3P0" "Yoda" "Darth Maul" "Dooku" "Chewbacca" "Jabba" "Lando Calrissian" "Boba Fett"

"Jango Fett" "Grievous" "Chief Chirpa"

csv_df.Name

<pre>20-element Vector{InlineStrings.String31}:</pre>
"Anakin Skywalker"
"Padme Amidala"
"Luke Skywalker"
"Leia Skywalker"
"Qui-Gon Jinn"
"Obi-Wan Kenobi"
"Han Solo"
"Sheev Palpatine"
"R2-D2"
"C-3P0"
"Yoda"
"Darth Maul"
"Dooku"
"Chewbacca"
"Jabba"
"Lando Calrissian"
"Boba Fett"
"Jango Fett"
"Grievous"
"Chief Chirpa"

csv_df[:, 1] # Note that this creates a copy.

```
20-element Vector{InlineStrings.String31}:
"Anakin Skywalker"
"Padme Amidala"
 "Luke Skywalker"
"Leia Skywalker"
 "Qui-Gon Jinn"
"Obi-Wan Kenobi"
"Han Solo"
"Sheev Palpatine"
"R2-D2"
"C-3P0"
"Yoda"
 "Darth Maul"
 "Dooku"
 "Chewbacca"
 "Jabba"
 "Lando Calrissian"
 "Boba Fett"
```

"Jango Fett" "Grievous" "Chief Chirpa"

The following are different ways to access a row:

csv_df[1:1, :]

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died	Jedi	Species	Weapon
	String31	String7	Float64	Float64?	String15?	String7?	String15?	String15	String15	String15	String7	String15	String15
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY	jedi	human	lightsaber

csv_df[1, :] # This produces a DataFrameRow.

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died	Jedi	Species	Weapon
	String31	String7	Float64	Float64?	String15?	String7?	String15?	String15	String15	String15	String7	String15	String15
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY	jedi	human	lightsaber

We can change the values just as we normally assign values.

Assign a range to scalar:

csv_df[1:3, :Height] .= 1.83

```
3-element view(::Vector{Float64}, 1:3) with eltype Float64:
1.83
1.83
1.83
```

Assign a vector:

csv_df[4:6, :Height] = [1.8, 1.6, 1.8]

3-element Vector{Float64}:
1.8
1.6
1.8

csv_df

Tip

There are a lot more things which can be done with a DataFrame. Read the docs for more information.

For information on dplyr-type syntax:

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died	Jedi	Species	Weapon
	String31	String7	Float64	Float64?	String15?	String7?	String15?	String15	String15	String15	String7	String15	String15
1	Anakin Skywalker	male	1.83	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY	jedi	human	lightsaber
2	Padme Amidala	female	1.83	45.0	brown	brown	light	Naboo	46BBY	19BBY	no_jedi	human	unarmed
3	Luke Skywalker	male	1.83	77.0	blue	blond	fair	Tatooine	19BBY	unk_died	jedi	human	lightsaber
4	Leia Skywalker	female	1.8	49.0	brown	brown	light	Alderaan	19BBY	unk_died	no_jedi	human	blaster
5	Qui-Gon Jinn	male	1.6	88.5	blue	brown	light	unk_planet	92BBY	32BBY	jedi	human	lightsaber
6	Obi-Wan Kenobi	male	1.8	77.0	bluegray	auburn	fair	Stewjon	57BBY	0BBY	jedi	human	lightsaber
7	Han Solo	male	1.8	80.0	brown	brown	light	Corellia	29BBY	unk_died	no_jedi	human	blaster
8	Sheev Palpatine	male	1.73	75.0	blue	red	pale	Naboo	82BBY	10ABY	no_jedi	human	force-lightning
9	R2-D2	male	0.96	32.0	missing	missing	missing	Naboo	33BBY	unk_died	no_jedi	droid	unarmed
10	C-3PO	male	1.67	75.0	missing	missing	missing	Tatooine	112BBY	3ABY	no_jedi	droid	unarmed
11	Yoda	male	0.66	17.0	brown	brown	green	unk_planet	896BBY	4ABY	jedi	yoda	lightsaber
12	Darth Maul	male	1.75	80.0	yellow	none	red	Dathomir	54BBY	unk_died	no_jedi	dathomirian	lightsaber
13	Dooku	male	1.93	86.0	brown	brown	light	Serenno	102BBY	19BBY	jedi	human	lightsaber
14	Chewbacca	male	2.28	112.0	blue	brown	missing	Kashyyyk	200BBY	25ABY	no_jedi	wookiee	bowcaster
15	Jabba	male	3.9	missing	yellow	none	tan-green	Tatooine	unk_born	4ABY	no_jedi	hutt	unarmed
16	Lando Calrissian	male	1.78	79.0	brown	blank	dark	Socorro	31BBY	unk_died	no_jedi	human	blaster
17	Boba Fett	male	1.83	78.0	brown	black	brown	Kamino	31.5BBY	unk_died	no_jedi	human	blaster
18	Jango Fett	male	1.83	79.0	brown	black	brown	ConcordDawn	66BBY	22BBY	no_jedi	human	blaster
19	Grievous	male	2.16	159.0	gold	black	orange	Kalee	unk_born	19BBY	no_jedi	kaleesh	slugthrower
20	Chief Chirpa	male	1.0	50.0	black	gray	brown	Endor	unk_born	4ABY	no_jedi	ewok	spear

- Read the DataFrames.jl documentation
- Check out DataFramesMeta.jl

Example: the passport problem

Let's now apply what we have learned to solve a real problem.

Data manipulation

The Passport Index Dataset lists travel visa requirements for 199 countries, in .csv format. Our task is to find the minimum number of passports required to visit all countries.

```
passport_data = CSV.read(
    joinpath(DATA_DIR, "passport-index-matrix.csv"),
    DataFrames.DataFrame,
);
```

In this dataset, the first column represents a passport (=from) and each remaining column represents a foreign country (=to).

The values in each cell are as follows:

- 3 = visa-free travel
- 2 = eTA is required
- 1 = visa can be obtained on arrival
- 0 = visa is required
- -1 is for all instances where passport and destination are the same

Our task is to find out the minimum number of passports needed to visit every country without requiring a visa.

The values we are interested in are -1 and 3. Let's modify the dataframe so that the -1 and 3 are 1 (true), and all others are 0 (false):

```
function modifier(x)
    if x == -1 || x == 3
        return 1
    else
        return 0
    end
end
for country in passport_data.Passport
    passport_data[!, country] = modifier.(passport_data[!, country])
end
```

The values in the cells now represent:

- 1 = no visa required for travel
- 0 = visa required for travel

JuMP Modeling

To model the problem as a mixed-integer linear program, we need a binary decision variable x_c for each country c. x_c is 1 if we select passport c and 0 otherwise. Our objective is to minimize the sum $\sum x_c$ over all countries.

Since we wish to visit all the countries, for every country, we must own at least one passport that lets us travel to that country visa free. For one destination, this can be mathematically represented as $\sum_{c \in C} a_{c,d} \cdot x_d \ge 1$, where a is the passport_data dataframe.

Thus, we can represent this problem using the following model:

$$\begin{array}{ll} \min & \displaystyle \sum_{c \in C} x_c \\ \text{s.t.} & \displaystyle \sum_{c \in C} a_{c,d} x_c \geq 1 \quad \forall d \in C \\ & \displaystyle x_c \in \{0,1\} \quad \forall c \in C. \end{array}$$

We'll now solve the problem using JuMP:

using JuMP import HiGHS

First, create the set of countries:

C = passport_data.Passport

199-element Vector{String}: "Afghanistan" "Albania" "Algeria"

```
"Andorra"
"Angola"
"Antigua and Barbuda"
"Argentina"
"Armenia"
"Australia"
"Austria"
"Uruguay"
"Uzbekistan"
"Vanuatu"
"Vatican"
"Venezuela"
"Viet Nam"
"Yemen"
"Zambia"
"Zimbabwe"
```

Then, create the model and initialize the decision variables:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[C], Bin)
@objective(model, Min, sum(x))
@constraint(model, [d in C], passport_data[!, d]' * x >= 1)
model
```

```
A JuMP Model
    solver: HiGHS
    objective_sense: MIN_SENSE
    L objective_function_type: AffExpr
    num_variables: 199
    num_constraints: 398
    AffExpr in MOI.GreaterThan{Float64}: 199
    L VariableRef in MOI.ZeroOne: 199
    Names registered in the model
    L :x
```

Now optimize:

optimize!(model)

We can use the solution_summary function to get an overview of the solution:

solution_summary(model)

```
solution_summary(; result = 1, verbose = false)
| solver_name : HiGHS
| Termination
| | termination_status : OPTIMAL
| | result_count : 1
| | raw_status : kHighsModelStatusOptimal
| L objective_bound : 2.30000e+01
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : N0_SOLUTION
| | objective_value : 2.30000e+01
| | dual_objective_value : NaN
| L relative_gap : 0.00000e+00
L Work counters
| solve_time (sec) : 7.18999e-03
| simplex_iterations : 26
| barrier_iterations : -1
L node_count : 1
```

Just to be sure, check that the solver found an optimal solution:

```
assert_is_solved_and_feasible(model)
```

Solution

Let's have a look at the solution in more detail:

println("Minimum number of passports needed: ", objective_value(model))

Minimum number of passports needed: 23.0

```
println("Optimal passports:")
for c in C
    if value(x[c]) > 0.5
        println(" * ", c)
    end
end
```

Optimal passports: * Afghanistan

- * Chad
- * Comoros
- * Djibouti
- * Georgia
- * Hong Kong
- * India

* Luxembourg
* Madagascar
* Maldives
* Mali
* New Zealand
* North Korea
* Papua New Guinea
* Singapore
* Somalia
* Sri Lanka
* Tunisia
* Turkey
* Uganda
* United Arab Emirates
* United States
* Zimbabwe

We need some passports, like New Zealand and the United States, which have widespread access to a large number of countries. However, we also need passports like North Korea which only have visa-free access to a very limited number of countries.

Note

```
We use value(x[c]) > 0.5 rather than value(x[c]) == 1 to avoid excluding solutions like x[c] = 0.99999 that are "1" to some tolerance.
```

4.6 Debugging

This tutorial was generated using Literate.jl. Download the source as a .jl file.

Dealing with bugs is an unavoidable part of coding optimization models in any framework, including JuMP. Sources of bugs include not only generic coding errors (method errors, typos, off-by-one issues), but also semantic mistakes in the formulation of an optimization problem and the incorrect use of a solver.

This tutorial explains some common sources of bugs and modeling issues that you might encounter when writing models in JuMP, and it suggests a variety of strategies to deal with them.

Tip

This tutorial is more advanced than the other "Getting started" tutorials. It's in the "Getting started" section to give you an early preview of how to debug JuMP models. However, if you are new to JuMP, you may want to briefly skim the tutorial, and come back to it once you have written a few JuMP models.

julia> using JuMP

julia> import HiGHS

Getting help

Debugging can be a frustrating part of modeling, particularly if you're new to optimization and programming. If you're stuck, join the community forum to search for answers to commonly asked questions.

Before asking a new question, make sure to read the post Make it easier to help you, which contains a number of tips on how to ask a good question.

Above all else, take time to simplify your code as much as possible. The fewer lines of code you can post that reproduces the same issue, the faster someone can answer your question.

Debugging Julia code

Read the Debugging chapter in the book ThinkJulia.jl. It has a number of great tips and tricks for debugging Julia code.

Solve failures

When a solver experiences an issue that prevents it from finding an optimal solution (or proving that one does not exist), JuMP may return one of a number of termination statuses.

For example, if the solver found a solution, but experienced numerical imprecision, it may return a status such as ALMOST_OPTIMAL or ALMOST_LOCALLY_SOLVED indicating that the problem was solved to a relaxed set of tolerances. Alternatively, the solver may return a problematic status such as NUMERICAL_ERROR, SLOW_PROGRESS, or OTHER_ERROR, indicating that it could not find a solution to the problem.

Most solvers can experience numerical imprecision because they use floating-point arithmetic to perform operations such as addition, subtraction, and multiplication. These operations aren't exact, and small errors can accrue between the theoretical value and the value that the computer computes. For example:

```
julia> 0.1 * 3 == 0.3 false
```

Тір

Read the Guidelines for numerical issues section of the Gurobi documentation, along with the Debugging numerical problems section of the YALMIP documentation.

Common sources

Common sources of solve failures are:

- Very large numbers and very small numbers as problem coefficients. Exactly what "large" is depends on the solver and the problem, but in general, values above 1e6 or smaller than 1e-6 cause problems.
- Nonlinear problems with functions that are not defined in parts of their domain. For example, minimizing log(x) where x >= 0 is undefined when x = 0 (a common starting value).

Strategies

Strategies to debug sources of solve failures include:

- Rescale variables in the problem and their associated coefficients to make the magnitudes of all coefficients in the 1e-4 to 1e4 range. For example, that might mean rescaling a variable from measuring distance in centimeters to kilometers.
- Try a different solver. Some solvers might be more robust than others for a particular problem.

- Read the documentation of your solver, and try settings that encourage numerical robustness.
- Set bounds or add constraints so that all nonlinear functions are defined across all of the feasible region.
 This particularly applies for functions like 1 / x and log(x) which are not defined for x = 0.

Incorrect results

Sometimes, you might find that the solver returns an "optimal" solution that is incorrect according to the model you are trying to solve (perhaps the solution is suboptimal, or it doesn't satisfy some of the constraints).

Incorrect results can be hard to detect and debug, because the solver gives no hints that there is a problem. Indeed, the termination_status will likely be OPTIMAL and a solution will be available.

Common sources

Common sources of incorrect results are:

- A modeling error, so that your JuMP model does not match the formulation you have on paper
- Not accounting for the tolerances that solvers use (for example, if x is binary, a value like x = 1.0000001 may still be considered feasible)
- A bug in JuMP or the solver.

The probability of the issue being a bug in JuMP or the solver is much smaller than a modeling error. When in doubt, first assume there is a bug in your code before assuming that there is a bug in JuMP.

Strategies

Strategies to debug sources of incorrect results include:

- Print your JuMP model to see if it matches the formulation you have on paper. Look out for incorrect signs + instead of -, and off-by-one errors such as x[t] instead of x[t-1].
- Check that you are not using exact comparisons like value(x) == 1.0; always use isapprox(value(x), 1.0; atol = 1e-6) where you manually specify the comparison tolerance.
- Try a different solver. If one solver succeeds where another doesn't this is a sign that the problem is a numerical issue or a bug in the solver.

Debugging an infeasible model

A model is infeasible if there is no primal solution that satisfies all of the constraints. In general, an infeasible model means one of two things:

- Your problem really has no feasible solution
- There is a mistake in your model.

Example

A simple example of an infeasible model is:

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x >= 0)
x
julia> @objective(model, Max, 2x + 1)
2 x + 1
julia> @constraint(model, con, 2x - 1 <= -2)
con : 2 x ≤ -1
```

because the bound says that $x \ge 0$, but we can rewrite the constraint to be $x \le -1/2$. When the problem is infeasible, JuMP may return one of a number of statuses. The most common is INFEASIBLE:

```
julia> optimize!(model)
julia> termination_status(model)
INFEASIBLE::TerminationStatusCode = 2
```

Depending on the solver, you may also receive INFEASIBLE OR UNBOUNDED or LOCALLY INFEASIBLE.

A termination status of INFEASIBLE_OR_UNBOUNDED means that the solver could not prove if the solver was infeasible or unbounded, only that the model does not have a finite feasible optimal solution.

Nonlinear optimizers such as Ipopt may return the status LOCALLY_INFEASIBLE. This does not mean that the solver *proved* no feasible solution exists, only that it could not find one. If you know a primal feasible point, try providing it as a starting point using set_start_value and re-optimize.

Common sources

Common sources of infeasibility are:

- · Incorrect units, for example, using a lower bound of megawatts and an upper bound of kilowatts
- Using + instead of in a constraint
- Off-by-one and related errors, for example, using x[t] instead of x[t-1] in part of a constraint
- · Otherwise invalid mathematical formulations

Strategies

Strategies to debug sources of infeasibility include:

- Iteratively comment out a constraint (or block of constraints) and re-solve the problem. When you find a constraint that makes the problem infeasible when added, check the constraint carefully for errors.
- If the problem is still infeasible with all constraints commented out, check all variable bounds. Do they use the right data?
- If you have a known feasible solution, use primal_feasibility_report to evaluate the constraints and check for violations. You'll probably find that you have a typo in one of the constraints.

Try a different solver. Sometimes, solvers have bugs, and they can incorrectly report a problem as
infeasible when it isn't. If you find such a case where one solver reports the problem is infeasible and
another can find an optimal solution, please report it by opening an issue on the GitHub repository of
the solver that reports infeasibility.

Тір

Some solvers also have specialized support for debugging sources of infeasibility via an irreducible infeasible subsystem. To see if your solver has support, try calling compute_conflict!:

julia> compute_conflict!(model)
ERROR: ArgumentError: The optimizer HiGHS.Optimizer does not support `compute_conflict!`

In this case, HiGHS does not support computing conflicts, but other solvers such as Gurobi and CPLEX do. If the solver does support computing conflicts, read Conflicts for more details.

Penalty relaxation

Another strategy to debug sources of infeasibility is the relax_with_penalty! function.

The penalty relaxation modifies constraints of the form $f(x) \in S$ into $f(x) + y - z \in S$, where $y, z \ge 0$, and then it introduces a penalty term into the objective of $a \times (y + z)$ (if minimizing, else -a), where a is a penalty.

Here map is a dictionary which maps constraint indices to an affine expression representing (y + z).

If we optimize the relaxed model, this time we get a feasible solution:

```
julia> optimize!(model)
```

```
julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1
```

Iterate over the contents of map to see which constraints are violated:

```
julia> for (con, penalty) in map
     violation = value(penalty)
     if violation > 0
        println("Constraint `$(name(con))` is violated by $violation")
     end
```



Once you find a violated constraint in the relaxed problem, take a look to see if there is a typo or other common mistake in that particular constraint.

Consult the docstring relax_with_penalty! for information on how to modify the penalty cost term a, either for every constraint in the model or a particular subset of the constraints.

When using relax_with_penalty!, you should be aware that:

- Variable bounds and integrality restrictions are not relaxed. If the problem is still infeasible after calling relax_with_penalty!, check the variable bounds.
- You cannot undo the penalty relaxation. If you need an unmodified model, rebuild the problem, or call copy model before calling relax with penalty!.

Debugging an unbounded model

A model is unbounded if there is no limit on how good the objective value can get. Most often, an unbounded model means that you have an error in your modeling, because all physical systems have limits. (You cannot make an infinite amount of profit.)

Example

A simple example of an unbounded model is:

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x >= 0)
x
julia> @objective(model, Max, 2x + 1)
2 x + 1
```

because we can increase x without limit, and the objective value 2x + 1 gets better as x increases.

When the problem is unbounded, JuMP may return one of a number of statuses. The most common is DUAL_INFEASIBLE:

```
julia> optimize!(model)
```

```
julia> termination_status(model)
DUAL_INFEASIBLE::TerminationStatusCode = 3
```

Depending on the solver, you may also receive INFEASIBLE_OR_UNBOUNDED or an error code like NORM_LIMIT.

Common sources

Common sources of unboundedness are:

- Using Max instead of Min
- Omitting variable bounds, such as 0 <= x <= 1
- Using + instead of in a term of the objective function.

Strategies

Strategies to debug sources of unboundedness include:

- Double check whether you intended Min or Max in the <a>@objective line.
- Print the objective function with print(objective_function(model)) and verify that the value and sign of each coefficient is as you expect.
- Add large bounds to all variables that are free or have one-sided bounds, then re-solve the problem. Because all variables are now bounded, the problem will have a finite optimal solution. Look at the value of each variable in the optimal solution to see if it is at one of the new bounds. If it is, you either need to specify a better bound for that variable, or there might be a mistake in the objective function associated with that variable (for example, a + instead of a -).

If there are too many variables to add bounds to, or there are too many terms to examine by hand, another strategy is to create a new variable with a large upper bound (if maximizing, lower bound if minimizing) and a constraint that the variable must be less-than or equal to the expression of the objective function. For example:

This new model has a finite optimal solution, so we can solve it and then look for variables with large positive or negative values in the optimal solution.

```
julia> optimize!(model)
julia> assert_is_solved_and_feasible(model)
```

```
julia> for var in all_variables(model)
    if var == objective
        continue
    end
    if abs(value(var)) > 1e3
        println("Variable `$(name(var))` may be unbounded")
    end
    end
Variable `x` may be unbounded
```

Debugging performance problems

There are two common sources for a model that takes a long time to solve:

- 1. JuMP builds the problem quickly, but the solver takes a long time to run or prove optimality. This commonly happens for mixed-integer programs, and you should see the solver print logs indicating slow but steady progress. There is no easy fix for this, other than choosing a different solver or reformulating your model.
- JuMP builds the problem slowly, and even if you wait a while, the solver may ever start running or displaying output.

This section explains how to debug the second case.

As a rule of thumb, we never expect JuMP to be the bottleneck in the solution process. If your model takes longer to build than to solve, or if it takes longer than a small number of minutes to build, then you have a fixable performance problem. JuMP models should never take hours to build.

Common sources

Common sources of performance problems are:

- Writing Julia code that has $O(N^2)$ or worse scaling behavior. As one common example, see Performance problems with sum-if formulations.
- Building a model with a very large (> 10^8) number of variables or constraints. Here the most likely cause of the performance problem is that you do not have enough memory to store the model. Use a computer with more RAM.

Macro timing

JuMP has a built-in feature that can measure the time spent in each macro. Turn it on using set_macro_timing, build the model, and then use print_macro_timing_summary to print a summary. Here's an example:

```
julia> begin
    N = 200
    demand = vcat(-1.0, zeros(N - 2), 1.0)
    edges = [(i, j) for i in 1:N for j in 1:N if i < j]
    model = Model()
    set_macro_timing(model, true)
    @variable(model, flows[e in edges] >= 0)
    @constraint(
```

In this case, you can see that the @constraint call dominates the runtime. If it isn't obvious why that is, read the Performance problems with sum-if formulations tutorial.

Note that the macro timing feature measures only the time spent inside JuMP macros. It does not measure regular Julia code outside the macros.

Other strategies

If the macro timing feature does not reveal the bottleneck, it means that your issue is in regular Julia code that is not inside a JuMP macro.

The strategy to debug JuMP models that have performance problems depends on how long your model takes to build.

As a first step, encapsulate everything you want to debug into a single function foo, so you can run it with @time foo().

Can you run @time foo() in seconds to minutes?

If the answer is "yes," then you can use ProfileView.jl to find the bottleneck.

To use ProfileView, do:

julia> **using** ProfileView

julia> @profview foo(); # run once to trigger compilation. Ignore the output.

```
julia> @profview foo()
```

This will open a flamegraph. The x-axis of the graph is time, so that wider bars take more time. The bars are stacked so that the foo() call is on the bottom, and subsequent calls within foo are stacked on top.

Reading a flamegraph can take some experience, but if you click on a bar it will print the line number to the REPL. Hunt around until you find the widest bar that points to a line of code that you have written, then ask yourself if it makes sense for this line to be the bottleneck.

If a wide bar points to code inside JuMP or a related Julia package, please open an issue on GitHub or post on the community forum.

If @time foo() takes longer than a few minutes to run, then either make the problem smaller by using a smaller dataset, or do the following.

- 1. Comment out everything in the function, then, line by line (or block by block):
- 2. Un-comment some code and re-run @time foo()
- 3. If the time increases by a lot (from seconds or minutes to hours), look for $O(N^2)$ or worse scaling behavior. Is there a better way to write the code that you are trying to execute?
- If the time increases by more than expected, but it still takes seconds or minutes to execute, use ProfileView to look for obvious bottlenecks.

4.7 Tolerances and numerical issues

This tutorial was generated using Literate.jl. Download the source as a .jl file.

Optimization solvers can seem like magic black boxes that take in an algebraic formulation of a problem and return a solution. It is tempting to treat their solutions at face value, since we often have little ability to verify that the solution is in fact optimal. However, like all numerical algorithms that use floating point arithmetic, optimization solvers use tolerances to check whether a solution satisfies the constraints. In the best case, the solution satisfies the original constraints to machine precision. In most cases, the solution satisfies the constraints to some very small tolerance that has no noticeable impact on the quality of the optimal solution. In the worst case, the solver can return a "wrong" solution, or fail to find one even if it exists. (The solution is "wrong" only in the sense of user expectation. It will satisfy the solution to the tolerances that are provided.)

The purpose of this tutorial is to explain the various types of tolerances that are used in optimization solvers and what you can reasonably expect from a solution.

There are a few sources of additional information:

- Ambros Gleixner has an excellent YouTube talk Numerics in LP & MIP Solvers
- · Gurobi has a series of articles in their documentation called Guidelines for Numerical Issues

Tip

This tutorial is more advanced than the other "Getting started" tutorials. It's in the "Getting started" section to give you an early preview of how tolerances affect JuMP models. However, if you are new to JuMP, you may want to briefly skim the tutorial, and come back to it once you have written a few JuMP models.

Required packages

This tutorial uses the following packages:

using JuMP import HiGHS import SCS

Background

Optimization solvers use tolerances to check the feasibility of constraints.

There are four main types of tolerances:

- 1. primal feasibility: controls how feasibility of the primal solution is measured
- 2. dual feasibility: controls how feasibility of the dual solution is measured
- 3. integrality: controls how feasibility of the binary and integer variables are measured
- 4. optimality: controls how close the primal and dual solutions must be.

Solvers may use absolute tolerances, relative tolerances, or some mixture of both. The definition and default value of each tolerance is solver-dependent.

The dual feasibility tolerance is much the same as the primal feasibility tolerance, only that operates on the space of dual solutions instead of the primal. HiGHS has dual_feasibility_tolerance, but some solvers have only a single feasibility tolerance that uses the same value for both.

The optimality tolerance is a more technical tolerance that is used to test the equivalence of the primal and dual objectives in the KKT system if you are solving a continuous problem via interior point. HiGHS has ipm_optimality_tolerance, but some solvers will not have such a tolerance. Note that the optimality tolerance is different to the relative MIP gap that controls early termination of a MIP solution during branch-and-bound.

Because the dual and optimality tolerances are less used, this tutorial focuses on the primal feasibility and integrality tolerances.

Primal feasibility

The primal feasibility tolerance controls how primal constraints are evaluated. For example, the constraint 2x = 1 is actually implemented as $|2x - 1| \le \varepsilon$, where ε is a small solver-dependent primal feasibility tolerance that is typically on the order of 1e-8.

Here's an example in practice. This model should be infeasible, since x must be non-negative, but there is also an equality constraint that x is equal to a small negative number. Yet when we solve this problem, we get:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x >= 0)
@constraint(model, x == -le-8)
optimize!(model)
is_solved_and_feasible(model)
```

true

value(x)

0.0

In other words, HiGHS thinks that the solution x = 0 satisfies the constraint x == -1e-8. The value of ε in HiGHS is controlled by the primal_feasibility_tolerance option. The default is 1e-7. If we set this to a smaller value, HiGHS will now correctly deduce that the problem is infeasible:

```
set_attribute(model, "primal_feasibility_tolerance", le-10)
optimize!(model)
is_solved_and_feasible(model)
```

false

Realistic example

Here's a more realistic example, which was reported in the SCS.jl repository:

```
n, ε = 13, 0.0234
N = 2^n
model = Model(SCS.Optimizer)
@variable(model, x[1:N] >= 0)
@objective(model, Min, x[1])
@constraint(model, sum(x) == 1)
z = [(-1)^((i & (1 << j)) >> j) for j in 0:n-1, i in 0:N-1]
@constraint(model, z * x .>= 1 - ε)
optimize!(model)
```

```
_____
^^I
     SCS v3.2.7 - Splitting Conic Solver
^^I(c) Brendan O'Donoghue, Stanford University, 2012
-----
problem: variables n: 8192, constraints m: 8206
cones: ^^I z: primal zero / dual free vars: 1
^^I l: linear vars: 8205
settings: eps_abs: 1.0e-04, eps_rel: 1.0e-04, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive_scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration_lookback: 10, acceleration_interval: 10
\ensuremath{^{\mbox{n}}}\xspace compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-qdldl
^^I nnz(A): 122880, nnz(P): 0
-----
iter | pri res | dua res | gap | obj | scale | time (s)
-----
   0| 2.00e+01 1.00e+00 2.00e+01 -9.98e+00 1.00e-01 3.50e-02
  100 | 6.92e-05 7.92e-05 7.33e-06 2.41e-05 1.00e-01 9.30e-02
-----
status: solved
timings: total: 9.31e-02s = setup: 3.39e-02s + solve: 5.92e-02s
```

```
^^I lin-sys: 5.02e-02s, cones: 2.16e-03s, accel: 7.29e-04s
objective = 0.000024
```

SCS reports that it solved the problem to optimality:

is_solved_and_feasible(model)

true

and that the solution for x[1] is nearly zero:

value(x[1])

2.04406873858532e-5

However, the analytic solution for x[1] is:

1 - n * ε / 2

0.8479

The answer is very wrong, and there is no indication from the solver that anything untoward happened. What's going on?

One useful debugging tool is primal_feasibility_report:

report = primal_feasibility_report(model)

Dict{Any, Float64} with 8192 entries: x[3585] ≥ 0 => 1.25513e-5 x[4278] ≥ 0 => 1.71395e-5 x[6729] ≥ 0 => 1.56101e-5 x[3588] ≥ 0 => 1.56101e-5 x[7518] ≥ 0 => 2.17278e-5 x[7166] ≥ 0 => 2.47866e-5 x[4358] ≥ 0 => 1.40807e-5 x[5071] ≥ 0 => 2.01984e-5 x[1948] ≥ 0 => 2.01984e-5 x[3811] ≥ 0 => 1.8669e-5 $\begin{array}{cccc} x[6649] \geq 0 & => & 2.01984e-5 \\ x[502] \geq 0 & => & 1.8669e-5 \\ x[2171] \geq 0 & => & 1.71395e-5 \\ x[2205] \geq 0 & => & 1.56101e-5 \\ x[4902] \geq 0 & => & 1.71395e-5 \\ x[6782] \geq 0 & => & 2.17278e-5 \\ x[7494] \geq 0 & => & 1.8669e-5 \\ x[1102] \geq 0 & => & 1.56101e-5 \\ x[5407] \geq 0 & => & 1.8669e-5 \\ \end{array}$

report is a dictionary which maps constraints to the violation. The largest violation is approximately 1e-5:

maximum(values(report))

6.92133754155444e-5

This makes sense, because the default primal feasibility tolerance for SCS is 1e-4.

Most of the entries are lower bound constraints on the variables. Here are all the variables which violate their lower bound:

violated_variables = filter(xi -> value(xi) < 0, x)</pre>

```
8178-element Vector{VariableRef}:
x[4]
x[6]
x[7]
x[8]
x[10]
x[11]
x[12]
x[13]
x[14]
x[15]
x[8184]
x[8185]
x[8186]
x[8187]
x[8188]
x[8189]
x[8190]
x[8191]
x[8192]
```

y = first(violated_variables)

 x_4

It has a primal value of:

value(y)

-1.1021914231743998e-5

which matches the value in the feasibility report:

report[LowerBoundRef(y)]

1.1021914231743998e-5

Despite the small primal feasibility tolerance and the small actual violations of the constraints, our optimal solution is very far from the theoretical optimum.

We can "fix" our model by decreasing eps_abs and eps_rel, which SCS uses to control the absolute and relative feasibility tolerances. Now the solver finds the correct solution:

```
set_attribute(model, "eps_abs", 1e-5)
set_attribute(model, "eps_rel", 1e-5)
optimize!(model)
```

```
_____
^^I
      SCS v3.2.7 - Splitting Conic Solver
^^I(c) Brendan O'Donoghue, Stanford University, 2012
_____
problem: variables n: 8192, constraints m: 8206
cones: ^^I z: primal zero / dual free vars: 1
^^I l: linear vars: 8205
settings: eps_abs: 1.0e-05, eps_rel: 1.0e-05, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive_scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration lookback: 10, acceleration interval: 10
^^I compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-qdldl
^^I nnz(A): 122880, nnz(P): 0
_____
iter | pri res | dua res | gap | obj | scale | time (s)
_____
   0| 2.00e+01 1.00e+00 2.00e+01 -9.98e+00 1.00e-01 3.42e-02
  250| 2.01e-02 2.85e-04 2.00e-02 3.01e-02 3.86e-01 1.83e-01
  500| 3.69e-04 5.93e-04 8.84e-05 8.48e-01 6.13e-01 3.34e-01
```

```
550| 2.66e-06 6.58e-10 1.27e-05 8.48e-01 6.13e-01 3.64e-01
status: solved
timings: total: 3.64e-01s = setup: 3.31e-02s + solve: 3.31e-01s
^I lin-sys: 2.71e-01s, cones: 1.16e-02s, accel: 5.70e-03s
objective = 0.847906
```

assert_is_solved_and_feasible(model)
value(x[1])

0.8479127435814551

Why you shouldn't use a small tolerance

There is no direct relationship between the size of feasibility tolerance and the quality of the solution.

You might surmise from this section that you should set the tightest feasibility tolerance possible. However, tighter tolerances come at the cost of increased solve time.

For example, SCS is a first-order solver. This means it uses only local gradient information at update each iteration. SCS took 100 iterations to solve the problem with the default tolerance of 1e-4, and 550 iterations to solve the problem with 1e-5. SCS may not be able to find a solution to our problem with a tighter tolerance in a reasonable amount of time.

Integrality

Integrality tolerances control how the solver decides if a variable satisfies an integrality or binary constraint. The tolerance is typically defined as: $|x - \lfloor x + 0.5 \rfloor| \le \varepsilon$, which you can read as the absolute distance to the nearest integer.

Here's a simple example:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
set_attribute(model, "presolve", "off")
@variable(model, x == 1 + 1e-6, Int)
optimize!(model)
is_solved_and_feasible(model)
```

true

HiGHS found an optimal solution, and the value of x is:

value(x)

```
1.000001
```

In other words, HiGHS thinks that the solution x = 1.000001 satisfies the constraint that x must be an integer. primal feasibility report shows that indeed, the integrality constraint is violated:

```
primal feasibility report(model)
```

Dict{Any, Float64} with 1 entry: x integer => 1.0e-6

The value of ε in HiGHS is controlled by the mip_feasibility_tolerance option. The default is 1e-6. If we set the attribute to a smaller value, HiGHS will now correctly deduce that the problem is infeasible:

```
set_attribute(model, "mip_feasibility_tolerance", 1e-10)
optimize!(model)
is_solved_and_feasible(model)
```

false

Realistic example

Integrality tolerances are particularly important when you have big-M type constraints. Small non-integer values in the integer variables can cause "leakage" flows even when the big-M switch is "off." Consider this example:

```
M = 1e6
model = Model()
@variable(model, x >= 0)
@variable(model, y, Bin)
@constraint(model, x <= M * y)
print(model)</pre>
```

Feasibility Subject to $x - 1000000 y \le 0$ $x \ge 0$ y binary

This model has a feasible solution (to tolerances) of (x, y) = (1, 1e-6); there can be a non-zero value of x even when y is (approximately) 0.

```
primal_feasibility_report(model, Dict(x => 1.0, y => 1e-6))
```

```
Dict{Any, Float64} with 1 entry:
  y binary => 1.0e-6
```

Rounding the solution

Integrality tolerances are the reason why JuMP does not return :: Int for value(x) of an integer variable or :: Bool for value(x) of a binary variable.

In most cases, it is safe to post-process the solution using $y_{int} = round(Int, value(y))$. However, in some cases "fixing" the integrality like this can cause violations in primal feasibility that exceed the primal feasibility tolerance. For example, if we rounded our (x, y) = (1, 1e-6) solution to (x, y) = (1, 0), then the constraint $x \le M * y$ is now violated by a value of 1.0, which is much greater than a typical feasibility tolerance of 1e-8.

```
primal_feasibility_report(model, Dict(x => 1.0, y => 0.0))
```

```
Dict{Any, Float64} with 1 entry:
    x - 1000000 y ≤ 0 => 1.0
```

Why you shouldn't use a small tolerance

Just like primal feasibility tolerances, using a smaller value for the integrality tolerance can lead to greatly increased solve times.

Contradictory results

The distinction between feasible and infeasible can be surprisingly nuanced. Solver A might decide the problem is feasible while solver B might decide it is infeasible. Different algorithms *within* solver A (like simplex and barrier) may also come to different conclusions. Even changing settings like turning presolve on and off can make a difference.

Here is an example where HiGHS reports the problem is infeasible, but there exists a feasible (to tolerance) solution:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x >= 0)
@variable(model, y >= 0)
@constraint(model, x + 1e8 * y == -1)
optimize!(model)
is_solved_and_feasible(model)
```

~		-		
-+	2		~	0
	а		~	-
	~	-	-	-

The feasible solution (x, y) = (0.0, -1e-8) has a maximum primal violation of 1e-8 which is smaller than the HiGHS feasibility tolerance:

```
primal_feasibility_report(model, Dict(x => 0.0, y => -le-8))
```

Dict{Any, Float64} with 1 entry: $y \ge 0 \implies 1.0e-8$

This happens because there are two basic solutions. The first is infeasible at (x, y) = (-1, 0) and the second is feasible (x, y) = (0, -1e-8). Different algorithms may terminate at either of these bases.

Another example is a variation on our integrality example, but this time, there are constraints that $x \ge 1$ and $y \le 0.5$:

```
M = 1e6
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x >= 1)
@variable(model, y, Bin)
@constraint(model, y <= 0.5)
@constraint(model, x <= M * y)
optimize!(model)
is_solved_and_feasible(model)
```

false

HiGHS reports the problem is infeasible, but there is a feasible (to tolerance) solution of:

```
primal_feasibility_report(model, Dict(x => 1.0, y => 1e-6))
```

```
Dict{Any, Float64} with 1 entry:
    y binary => 1.0e-6
```

This happens because the presolve routine deduces that the y <= 0.5 constraint forces the binary variable y to take the value 0. Substituting the value for y into the last constraint, presolve may also deduce that x <= 0, which violates the bound of x >= 1 and so the problem is infeasible.

We can work around this by providing HiGHS with the feasible starting solution:

set_start_value(x, 1)
set_start_value(y, 1e-6)

Now HiGHS will report that the problem is feasible:

```
optimize!(model)
is_solved_and_feasible(model)
```

true

Contradictory results are not a bug in the solver

These contradictory examples are not bugs in the HiGHS solver. They are an expected result of the interaction between the tolerances and the solution algorithm. There will always be models in the gray boundary at the edge of feasibility, for which the question of feasibility is not a clear true or false.

Problem scaling

Problem scaling refers to the absolute magnitudes of the data in your problem. The data is any numbers in the objective, the constraints, or the variable bounds.

We say that a problem is poorly scaled if there are very small ($< 10^{-3}$) or very large ($> 10^{6}$) coefficients in the problem, or if the ratio of the largest to smallest coefficient is large.

Numerical issues related to the feasibility tolerances most commonly arise because of poor problem scaling. The next examples assume a primal feasibility tolerance of 1e-8, but actual tolerances may vary from one solver to another.

Small magnitudes

If the problem data is too small, then the feasibility tolerance can be too easily satisfied. For example, consider:

```
model = Model()
@variable(model, x)
@constraint(model, le-8 * x == le-4)
```

$$1.0 \times 10^{-8} x = 0.0001$$

This should have the solution that $x = 10^4$, but because the feasibility tolerance of this constraint is $|10^{-4} - 10^{-8}x| < 10^{-8}$, it actually permits any value of x between 9999 and 10,001, which is a larger range of feasible values than you might have expected.

Large magnitudes

If the problem data is too large, then the feasibility tolerance can be too difficult to satisfy.

```
model = Model()
@variable(model, x)
@constraint(model, lel2 * x == le4)
```

1000000000000x = 10000

This should have the solution that $x = 10^{-8}$, but because the feasibility tolerance of this constraint is $|10^{12}x - 10^4| < 10^{-8}$, it actually permits any value of x in $10^{-8} \pm 10^{-20}$, which is a smaller range of feasible values than you might have expected.

Large magnitude ratios

If the ratio of the smallest to the largest magnitude is too large, then the tolerances or small changes in the input data can lead to large changes in the optimal solution. We have already seen an example with the integrality tolerance, but we can exacerbate the behavior by putting a small coefficient on x:

model = Model()
@variable(model, x >= 0)
@variable(model, y, Bin)
@constraint(model, le-6x <= le6 * y)</pre>

 $1.0 \times 10^{-6} x - 1000000 y \le 0$

This problem has a feasible (to tolerance) solution of:

primal_feasibility_report(model, Dict(x => 1_000_000.01, y => 1e-6))

Dict{Any, Float64} with 2 entries: y binary => 1.0e-6 1.0e-6 x - 1000000 y ≤ 0 => 1.0e-8

If you intended the constraint to read that if x is non-zero then y = 1, this solution might be unexpected.

Recommended values

There are no hard rules that you must follow, and the interaction between tolerances, problem scaling, and the solution is problem dependent. You should always check the solution returned by the solver to check it makes sense for your application.

With that caveat in mind, a general rule of thumb to follow is:

Try to keep the ratio of the smallest to largest coefficient less than 10^6 in any row and column, and try to keep most values between 10^{-3} and 10^6 .

Choosing the correct units

The best way to fix problem scaling is by changing the units of your variables and constraints. Here's an example. Suppose we are choosing the level of capacity investment in a new power plant. We can install up to 1 GW of capacity at a cost of \$1.78/W, and we have a budget of \$200 million.

```
model = Model()
@variable(model, 0 <= x_capacity_W <= 10^9)
@constraint(model, 1.78 * x_capacity_W <= 200e6)</pre>
```

 $1.78x_capacity_W \le 200000000$

This constraint violates the recommendations because there are values greater than 10^6 , and the ratio of the coefficients in the constraint is 10^8 .

One fix is the convert our capacity variable from Watts to Megawatts. This yields:

```
model = Model()
@variable(model, 0 <= x_capacity_MW <= 10^3)
@constraint(model, 1.78e6 * x_capacity_MW <= 200e6)</pre>
```

$1780000x_{capacity_MW} \le 200000000$

We can improve our model further by dividing the constraint by 10^6 to change the units from dollars to million dollars.

```
model = Model()
@variable(model, 0 <= x_capacity_MW <= 10^3)
@constraint(model, 1.78 * x_capacity_MW <= 200)</pre>
```

 $1.78x_capacity_MW \le 200$

This problem is equivalent to the original problem, but it has much better problem scaling.

As a general rule, to fix problem scaling you must simultaneously scale both variables and constraints. It is usually not sufficient to scale variables or constraints in isolation.

4.8 Design patterns for larger models

This tutorial was generated using Literate.jl. Download the source as a .jl file.

JuMP makes it easy to build and solve optimization models. However, once you start to construct larger models, and especially ones that interact with external data sources or have customizable sets of variables and constraints based on client choices, you may find that your scripts become unwieldy. This tutorial demonstrates a variety of ways in which you can structure larger JuMP models to improve their readability and maintainability.

Tip

This tutorial is more advanced than the other "Getting started" tutorials. It's in the "Getting started" section to give you an early preview of how JuMP makes it easy to structure larger models. However, if you are new to JuMP you may want to briefly skim the tutorial, and come back to it once you have written a few JuMP models.

Overview

This tutorial uses explanation-by-example. We're going to start with a simple knapsack model, and then expand it to add various features and structure.

A simple script

Your first prototype of a JuMP model is probably a script that uses a small set of hard-coded data.

```
using JuMP, HiGHS
profit = [5, 3, 2, 7, 4]
weight = [2, 8, 4, 2, 5]
capacity = 10
N = 5
model = Model(HiGHS.Optimizer)
@variable(model, x[1:N], Bin)
@objective(model, Max, sum(profit[i] * x[i] for i in 1:N))
@constraint(model, sum(weight[i] * x[i] for i in 1:N) <= capacity)
optimize!(model)
assert_is_solved_and_feasible(model)
value.(x)</pre>
```

```
5-element Vector{Float64}:

1.0

0.0

-0.0

1.0

1.0
```

The benefits of this approach are:

- it is quick to code
- it is quick to make changes.

The downsides include:

- all variables are global (read Performance tips)
- it is easy to introduce errors, for example, having profit and weight be vectors of different lengths, or not match N
- the solution, x[i], is hard to interpret without knowing the order in which we provided the data.

Wrap the model in a function

A good next step is to wrap your model in a function. This is useful for a few reasons:

- · it removes global variables
- it encapsulates the JuMP model and forces you to clarify your inputs and outputs
- we can add some error checking.

```
function solve_knapsack_1(profit::Vector, weight::Vector, capacity::Real)
    if length(profit) != length(weight)
        throw(DimensionMismatch("profit and weight are different sizes"))
    end
    N = length(weight)
    model = Model(HiGHS.Optimizer)
    @variable(model, x[1:N], Bin)
    @objective(model, Max, sum(profit[i] * x[i] for i in 1:N))
    @constraint(model, sum(weight[i] * x[i] for i in 1:N) <= capacity)
    optimize!(model)
    assert_is_solved_and_feasible(model)
    return value.(x)
end</pre>
```

solve_knapsack_1([5, 3, 2, 7, 4], [2, 8, 4, 2, 5], 10)

```
5-element Vector{Float64}:

1.0

0.0

-0.0

1.0

1.0
```

Create better data structures

Although we can check for errors like mis-matched vector lengths, if you start to develop models with a lot of data, keeping track of vectors and lengths and indices is fragile and a common source of bugs. A good solution is to use Julia's type system to create an abstraction over your data.

For example, we can create a struct that represents a single object, with a constructor that lets us validate assumptions on the input data:

as well as a struct that holds a dictionary of objects and the knapsack's capacity:

```
struct KnapsackData
    objects::Dict{String,KnapsackObject}
    capacity::Float64
end
```
Here's what our data might look like now:

```
objects = Dict(
    "apple" => KnapsackObject(5.0, 2.0),
    "banana" => KnapsackObject(3.0, 8.0),
    "cherry" => KnapsackObject(2.0, 4.0),
    "date" => KnapsackObject(7.0, 2.0),
    "eggplant" => KnapsackObject(4.0, 5.0),
)
data = KnapsackData(objects, 10.0)
```

```
Main.KnapsackData(Dict{String, Main.KnapsackObject}("cherry" => Main.KnapsackObject(2.0, 4.0),

→ "banana" => Main.KnapsackObject(3.0, 8.0), "date" => Main.KnapsackObject(7.0, 2.0), "eggplant"

→ => Main.KnapsackObject(4.0, 5.0), "apple" => Main.KnapsackObject(5.0, 2.0)), 10.0)
```

If you want, you can add custom printing to make it easier to visualize:

```
function Base.show(io::IO, data::KnapsackData)
    println(io, "A knapsack with capacity $(data.capacity) and possible items:")
    for (k, v) in data.objects
        println(
            io,
            " $(rpad(k, 8)) : profit = $(v.profit), weight = $(v.weight)",
            )
    end
    return
end
data
```

A knapsack with capacity 10.0 and possible items: cherry : profit = 2.0, weight = 4.0 banana : profit = 3.0, weight = 8.0 date : profit = 7.0, weight = 2.0 eggplant : profit = 4.0, weight = 5.0 apple : profit = 5.0, weight = 2.0

Then, we can re-write our solve_knapsack function to take our KnapsackData as input:

```
function solve_knapsack_2(data::KnapsackData)
model = Model(HiGHS.Optimizer)
@variable(model, x[keys(data.objects)], Bin)
@objective(model, Max, sum(v.profit * x[k] for (k, v) in data.objects))
@constraint(
            model,
            sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,
)
optimize!(model)</pre>
```

```
assert_is_solved_and_feasible(model)
    return value.(x)
end
```

solve_knapsack_2(data)

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
    -0.0
    0.0
    1.0
    1.0
    1.0
```

Read in data from files

Having a data structure is a good step. But it is still annoying that we have to hard-code the data into Julia. A good next step is to separate the data into an external file format; JSON is a common choice.

```
json_data = """
{
    "objects": {
        "apple": {"profit": 5.0, "weight": 2.0},
        "banana": {"profit": 3.0, "weight": 8.0},
        "cherry": {"profit": 2.0, "weight": 4.0},
        "date": {"profit": 7.0, "weight": 2.0},
        "eggplant": {"profit": 4.0, "weight": 5.0}
   },
    "capacity": 10.0
}
.....
temp_dir = mktempdir()
knapsack_json_filename = joinpath(temp_dir, "knapsack.json")
# Instead of writing a new file here you could replace `knapsack_json_filename`
# with the path to a local file.
write(knapsack_json_filename, json_data);
```

Now let's write a function that reads this file and builds a KnapsackData object:

) end

data = read_data(knapsack_json_filename)

```
A knapsack with capacity 10.0 and possible items:

cherry : profit = 2.0, weight = 4.0

banana : profit = 3.0, weight = 8.0

date : profit = 7.0, weight = 2.0

eggplant : profit = 4.0, weight = 5.0

apple : profit = 5.0, weight = 2.0
```

Add options via if-else

At this point, we have data in a file format which we can load and solve a single problem. For many users, this might be sufficient. However, at some point you may be asked to add features like "but what if we want to take more than one of a particular item?"

If this is the first time that you've been asked to add a feature, adding options via if-else statements is a good approach. For example, we might write:

```
function solve_knapsack_3(data::KnapsackData; binary_knapsack::Bool)
   model = Model(HiGHS.Optimizer)
   if binary_knapsack
        @variable(model, x[keys(data.objects)], Bin)
   else
        @variable(model, x[keys(data.objects)] >= 0, Int)
   end
   @objective(model, Max, sum(v.profit * x[k] for (k, v) in data.objects))
   @constraint(
        model,
        sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,</pre>
   )
   optimize!(model)
   assert_is_solved_and_feasible(model)
    return value.(x)
end
```

solve_knapsack_3 (generic function with 1 method)

Now we can solve the binary knapsack:

solve_knapsack_3(data; binary_knapsack = true)

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
    -0.0
    0.0
    1.0
    1.0
    1.0
```

And an integer knapsack where we can take more than one copy of each item:

solve_knapsack_3(data; binary_knapsack = false)

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
    0.0
    0.0
    5.0
    0.0
    0.0
    0.0
```

Add configuration options via dispatch

If you get repeated requests to add different options, you'll quickly find yourself in a mess of different flags and if-else statements. It's hard to write, hard to read, and hard to ensure you haven't introduced any bugs. A good solution is to use Julia's type dispatch to control the configuration of the model. The easiest way to explain this is by example.

First, start by defining a new abstract type, as well as new subtypes for each of our options. These types are going to control the configuration of the knapsack model.

```
abstract type AbstractConfiguration end
struct BinaryKnapsackConfig <: AbstractConfiguration end
struct IntegerKnapsackConfig <: AbstractConfiguration end</pre>
```

Then, we rewrite our solve_knapsack function to take a config argument, and we introduce an add_knapsack_variables function to abstract the creation of our variables.

```
function solve_knapsack_4(data::KnapsackData, config::AbstractConfiguration)
model = Model(HiGHS.Optimizer)
x = add_knapsack_variables(model, data, config)
@objective(model, Max, sum(v.profit * x[k] for (k, v) in data.objects))
@constraint(
            model,
            sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,</pre>
```

```
)
optimize!(model)
assert_is_solved_and_feasible(model)
return value.(x)
end
```

solve_knapsack_4 (generic function with 1 method)

For the binary knapsack problem, add_knapsack_variables looks like this:

```
function add_knapsack_variables(
    model::Model,
    data::KnapsackData,
    ::BinaryKnapsackConfig,
)
    return @variable(model, x[keys(data.objects)], Bin)
end
```

add_knapsack_variables (generic function with 1 method)

For the integer knapsack problem, add_knapsack_variables looks like this:

```
function add_knapsack_variables(
    model::Model,
    data::KnapsackData,
    ::IntegerKnapsackConfig,
)
    return @variable(model, x[keys(data.objects)] >= 0, Int)
end
```

add_knapsack_variables (generic function with 2 methods)

Now we can solve the binary knapsack:

```
solve_knapsack_4(data, BinaryKnapsackConfig())
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
    -0.0
    0.0
    1.0
    1.0
    1.0
```

and the integer knapsack problem:

```
solve_knapsack_4(data, IntegerKnapsackConfig())
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
    0.0
    0.0
    5.0
    0.0
    0.0
    0.0
```

The main benefit of the dispatch approach is that you can quickly add new options without needing to modify the existing code. For example:

```
struct UpperBoundedKnapsackConfig <: AbstractConfiguration
    limit::Int
end
function add_knapsack_variables(
    model::Model,
    data::KnapsackData,
    config::UpperBoundedKnapsackConfig,
)
    return @variable(model, 0 <= x[keys(data.objects)] <= config.limit, Int)
end</pre>
```

solve_knapsack_4(data, UpperBoundedKnapsackConfig(3))

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
    0.0
    0.0
    3.0
    0.0
    2.0
```

Generalize constraints and objectives

It's easy to extend the dispatch approach to constraints and objectives as well. The key points to notice in the next two functions are that:

- we can access registered variables via model[:x]
- we can define generic functions which accept any AbstractConfiguration as a configuration argument. That means we can implement a single method and have it apply to multiple configuration types.

```
function add_knapsack_constraints(
    model::Model,
    data::KnapsackData,
    ::AbstractConfiguration,
)
   x = model[:x]
   @constraint(
        model,
        capacity_constraint,
        sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,</pre>
   )
    return
end
function add_knapsack_objective(
   model::Model,
   data::KnapsackData,
   ::AbstractConfiguration,
)
   x = model[:x]
   @objective(model, Max, sum(v.profit * x[k] for (k, v) in data.objects))
    return
end
function solve_knapsack_5(data::KnapsackData, config::AbstractConfiguration)
   model = Model(HiGHS.Optimizer)
    add_knapsack_variables(model, data, config)
    add_knapsack_constraints(model, data, config)
    add_knapsack_objective(model, data, config)
    optimize!(model)
    assert_is_solved_and_feasible(model)
    return value.(model[:x])
end
solve_knapsack_5(data, BinaryKnapsackConfig())
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
    -0.0
    0.0
    1.0
    1.0
    1.0
```

Function barriers

Querying a variable like x = model[:x] is not type stable. The lack of type stability means that Julia cannot statically prove what the type of x will be within a function. As a consequence, the functions add_knapsack_constraints and add_knapsack_objective are slower than they would be if Julia could prove that x was a Containers.DenseAxisArray.

One solution to the problem of type stability is to use a function barrier:

```
function add_knapsack_constraints(
    model::Model,
    data::KnapsackData,
    config::AbstractConfiguration,
)
    return add_knapsack_constraints_inner(model, data, config, model[:x])
end
function add_knapsack_constraints_inner(
    model::Model,
    data::KnapsackData,
    ::AbstractConfiguration,
    х,
)
    @constraint(
        model,
        capacity_constraint,
        sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,</pre>
    )
    return
end
```

add_knapsack_constraints_inner (generic function with 1 method)

Now, add_knapsack_constraints_inner is faster because Julia can compile a specific version that depends on the type of x passed as an input argument.

The downside of function barriers is that they require more code. The upside is faster code and reduced memory allocations. In general, you should add a function barrier if you have a benchmark showing that the difference is meaningful for your code base. Function barriers are more likely to be useful if the _inner function does a lot of computational work, for example, it adds thousands of constraints or has a summation over thousands of items.

Remove solver dependence, add error checks

Compared to where we started, our knapsack model is now significantly different. We've wrapped it in a function, defined some data types, and introduced configuration options to control the variables and constraints that get added. There are a few other steps we can do to further improve things:

- remove the dependence on HiGHS
- add checks that we found an optimal solution
- add a helper function to avoid the need to explicitly construct the data.

```
function solve_knapsack_6(
    optimizer,
    data::KnapsackData,
    config::AbstractConfiguration,
)
    model = Model(optimizer)
```

```
add_knapsack_variables(model, data, config)
   add_knapsack_constraints(model, data, config)
   add_knapsack_objective(model, data, config)
   optimize!(model)
   if !is_solved_and_feasible(model)
        @warn("Model not solved to optimality")
        return nothing
   end
   return value.(model[:x])
end
function solve_knapsack_6(
   optimizer,
   data::String,
   config::AbstractConfiguration,
)
    return solve_knapsack_6(optimizer, read_data(data), config)
end
solution = solve_knapsack_6(
   HiGHS.Optimizer,
   knapsack_json_filename,
   BinaryKnapsackConfig(),
)
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
    -0.0
    0.0
    1.0
    1.0
    1.0
```

Create a module

Now we're ready to expose our model to the wider world. That might be as part of a larger Julia project that we're contributing to, or as a stand-alone script that we can run on-demand. In either case, it's good practice to wrap everything in a module. This further encapsulates our code into a single namespace, and we can add documentation in the form of docstrings.

Some good rules to follow when creating a module are:

- use import in a module instead of using to make it clear which functions are from which packages
- use _ to start function and type names that are considered private
- add docstrings to all public variables and functions.

```
module KnapsackModel
import JuMP
import JSON
struct _KnapsackObject
    profit::Float64
    weight::Float64
    function _KnapsackObject(profit::Float64, weight::Float64)
        if weight < 0
            throw(DomainError("Weight of object cannot be negative"))
        end
        return new(profit, weight)
    end
end
struct _KnapsackData
    objects::Dict{String,_KnapsackObject}
    capacity::Float64
end
function _read_data(filename)
    d = JSON.parsefile(filename)
    return KnapsackData(
        Dict(
            k => _KnapsackObject(v["profit"], v["weight"]) for
            (k, v) in d["objects"]
        ),
        d["capacity"],
    )
end
abstract type _AbstractConfiguration end
.....
    BinaryKnapsackConfig()
Create a binary knapsack problem where each object can be taken 0 or 1 times.
.....
struct BinaryKnapsackConfig <: _AbstractConfiguration end</pre>
.....
    IntegerKnapsackConfig()
Create an integer knapsack problem where each object can be taken any number of
times.
.....
struct IntegerKnapsackConfig <: _AbstractConfiguration end</pre>
function add knapsack variables(
    model::JuMP.Model,
    data::_KnapsackData,
    ::BinaryKnapsackConfig,
)
    return JuMP.@variable(model, x[keys(data.objects)], Bin)
```

end

```
function _add_knapsack_variables(
   model::JuMP.Model,
    data::_KnapsackData,
    ::IntegerKnapsackConfig,
)
    return JuMP.@variable(model, x[keys(data.objects)] >= 0, Int)
end
function _add_knapsack_constraints(
   model::JuMP.Model,
   data::_KnapsackData,
    config::_AbstractConfiguration,
)
    return _add_knapsack_constraints_inner(model, data, config, model[:x])
end
function _add_knapsack_constraints_inner(
   model::JuMP.Model,
   data::_KnapsackData,
    ::_AbstractConfiguration,
   х,
)
   JuMP.@constraint(
        model,
        capacity_constraint,
        sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,</pre>
    )
    return
end
function _add_knapsack_objective(
   model::JuMP.Model,
   data::_KnapsackData,
    config::_AbstractConfiguration,
)
    return _add_knapsack_objective_inner(model, data, config, model[:x])
end
function _add_knapsack_objective_inner(
   model::JuMP.Model,
   data::_KnapsackData,
    ::_AbstractConfiguration,
   х,
)
   JuMP.@objective(model, Max, sum(v.profit * x[k] for (k, v) in data.objects))
    return
end
function _solve_knapsack(
   optimizer,
   data::_KnapsackData,
   config::_AbstractConfiguration,
)
```

```
model = JuMP.Model(optimizer)
   _add_knapsack_variables(model, data, config)
   _add_knapsack_constraints(model, data, config)
   _add_knapsack_objective(model, data, config)
   JuMP.optimize!(model)
   if !JuMP.is_solved_and_feasible(model)
        @warn("Model not solved to optimality")
        return nothing
   end
   return JuMP.value.(model[:x])
end
.....
   solve_knapsack(
        optimizer,
        knapsack_json_filename::String,
        config:: AbstractConfiguration,
    )
```

Solve the knapsack problem and return the optimal primal solution

```
# Arguments
```

- * `optimizer` : an object that can be passed to `JuMP.Model` to construct a new JuMP model.
- * `knapsack_json_filename` : the filename of a JSON file containing the data for the problem.
- * `config` : an object to control the type of knapsack model constructed. Valid options are:

 - * `BinaryKnapsackConfig()`
 - * `IntegerKnapsackConfig()`

```
# Returns
```

- * If an optimal solution exists: a `JuMP.DenseAxisArray` that maps the `String` name of each object to the number of objects to pack into the knapsack.
- * Otherwise, `nothing`, indicating that the problem does not have an optimal solution.

```
# Example
```

```
```julia
solution = solve_knapsack(
 HiGHS.Optimizer,
 "path/to/data.json",
 BinaryKnapsackConfig(),
)
```
```julia
solution = solve_knapsack(
 MOI.OptimizerWithAttributes(HiGHS.Optimizer, "output_flag" => false),
 "path/to/data.json",
 IntegerKnapsackConfig(),
)
```

```
function solve_knapsack(
 optimizer,
 knapsack_json_filename::String,
 config::_AbstractConfiguration,
)
 data = _read_data(knapsack_json_filename)
 return _solve_knapsack(optimizer, data, config)
end
```

end

Main.KnapsackModel

Finally, you can call your model:

```
import .KnapsackModel
```

```
KnapsackModel.solve_knapsack(
 HiGHS.Optimizer,
 knapsack_json_filename,
 KnapsackModel.BinaryKnapsackConfig(),
)
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
 Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
 -0.0
 0.0
 1.0
 1.0
 1.0
 1.0
```

#### Note

The . in .KnapsackModel denotes that it is a submodule and not a separate package that we installed with Pkg.add. If you put the KnapsackModel in a separate file, load it with:

```
include("path/to/KnapsackModel.jl")
import .KnapsackModel
```

# Add tests

As a final step, you should add tests for your model. This often means testing on a small problem for which you can work out the optimal solution by hand. The Julia standard library Test has good unit-testing functionality.

```
import .KnapsackModel
using Test
@testset "KnapsackModel" begin
 @testset "feasible_binary_knapsack" begin
 x = KnapsackModel.solve_knapsack(
 HiGHS.Optimizer,
 knapsack_json_filename,
 KnapsackModel.BinaryKnapsackConfig(),
)
 @test isapprox(x["apple"], 1, atol = 1e-5)
 @test isapprox(x["banana"], 0, atol = 1e-5)
 @test isapprox(x["cherry"], 0, atol = 1e-5)
 @test isapprox(x["date"], 1, atol = 1e-5)
 @test isapprox(x["eggplant"], 1, atol = 1e-5)
 end
 @testset "feasible integer knapsack" begin
 x = KnapsackModel.solve knapsack(
 HiGHS.Optimizer,
 knapsack_json_filename,
 KnapsackModel.IntegerKnapsackConfig(),
)
 @test isapprox(x["apple"], 0, atol = 1e-5)
 @test isapprox(x["banana"], 0, atol = 1e-5)
 @test isapprox(x["cherry"], 0, atol = 1e-5)
 @test isapprox(x["date"], 5, atol = 1e-5)
 @test isapprox(x["eggplant"], 0, atol = 1e-5)
 end
 @testset "infeasible_binary_knapsack" begin
 dir = mktempdir()
 infeasible_filename = joinpath(dir, "infeasible.json")
 write(
 infeasible_filename,
 "objects": {
 "apple": {"profit": 5.0, "weight": 2.0},
 "banana": {"profit": 3.0, "weight": 8.0},
 "cherry": {"profit": 2.0, "weight": 4.0},
 "date": {"profit": 7.0, "weight": 2.0},
 "eggplant": {"profit": 4.0, "weight": 5.0}
 },
 "capacity": -10.0
 }""",
)
 x = KnapsackModel.solve_knapsack(
 HiGHS.Optimizer,
 infeasible_filename,
 KnapsackModel.BinaryKnapsackConfig(),
)
 @test x === nothing
 end
end
```

Test.DefaultTestSet("KnapsackModel", Any[Test.DefaultTestSet("feasible\_binary\_knapsack", Any[], 5,

- $\hookrightarrow \mbox{ false, true, 1.747779948576879e9, 1.747779948578839e9, false,}$
- $\hookrightarrow \ \ \text{Any[], 5, false, false, true, 1.747779948578863e9, 1.747779948751315e9, false, }$
- → Any[], 1, false, false, true, 1.747779948751364e9, 1.747779948753238e9, false,
- $\rightarrow$  "design\_patterns\_for\_larger\_models.md")], 0, false, false, true, 1.747779948576843e9,
- $\hookrightarrow$  1.747779948753245e9, false, "design\_patterns\_for\_larger\_models.md")

#### Тір

Place these tests in a separate file test\_knapsack\_model.jl so that you can run the tests by adding include("test\_knapsack\_model.jl") to any file where needed.

## Next steps

We've only briefly scratched the surface of ways to create and structure large JuMP models, so consider this tutorial a starting point, rather than a comprehensive list of all the possible ways to structure JuMP models. If you are embarking on a large project that uses JuMP, a good next step is to look at ways people have written large JuMP projects "in the wild."

Here are some good examples (all co-incidentally related to energy):

- AnyMOD.jl
  - JuMP-dev 2021 talk
  - source code
- PowerModels.jl
  - JuMP-dev 2021 talk
  - source code
- PowerSimulations.jl
  - JuliaCon 2021 talk
  - source code
- UnitCommitment.jl
  - JuMP-dev 2021 talk
  - source code

#### 4.9 Performance tips

This tutorial was generated using Literate.jl. Download the source as a .jl file.

By now you should have read the other "getting started" tutorials. You're almost ready to write your own models, but before you do so there are some important things to be aware of.

The Julia manual has an excellent section on Performance tips. The purpose of this tutorial is to highlight a number of performance issues that are specific to JuMP.

# **Required packages**

This tutorial uses the following packages:

julia> using JuMP

julia> import HiGHS

#### Use macros to build expressions

Use JuMP's macros to build expressions.

Constructing an expression outside the macro results in intermediate copies of the expression. For example,

x[1] + x[2] + x[3]

is equivalent to

a = x[1] b = a + x[2]c = b + x[3]

Since we only care about c, the a and b expressions are not needed and constructing them slows the program down.

JuMP's macros rewrite the expressions to operate in-place and avoid these extra copies. Because they allocate less memory, they are faster, particularly for large expressions.

Here's an example.

```
julia> model = Model()
A JuMP Model
+ solver: none
+ objective_sense: FEASIBILITY_SENSE
+ num_variables: 0
+ num_constraints: 0
L Names registered in the model: none
julia> @variable(model, x[1:3])
```

# 3-element Vector{VariableRef}: x[1] x[2] x[3]

Here's what happens if we construct the expression outside the macro:

julia> @allocated x[1] + x[2] + x[3]
1296

# Info

The @allocated measures how many bytes were allocated during the evaluation of an expression. Fewer is better.

If we use the @expression macro, we get many fewer allocations:

```
julia> @allocated @expression(model, x[1] + x[2] + x[3])
736
```

# Use add\_to\_expression! to build summations

If you don't want to use the expression macros, use add\_to\_expression! to build summations. For example, instead of:

# do

**julia> expr** x[1] + x[2] + x[3]

The former is equivalent to:

```
julia> expr0 = zero(AffExpr)
0
julia> expr1 = expr0 + x[1]
x[1]
julia> expr2 = expr1 + x[2]
x[1] + x[2]
julia> expr = expr2 + x[3]
x[1] + x[2] + x[3]
```

CHAPTER 4. GETTING STARTED

which allocates four unique AffExpr objects. The latter efficiently updates expr in-place so that only one AffExpr object is allocated.

The function  $add_to_expression!$  also supports terms like y += a \* x where a is a constant. For example, instead of:

# do

julia> expr
x[1] + 2 x[2] + 3 x[3]

Don't do this, because i \* x[i] will allocate a new AffExpr in each iteration:

# **Disable string names**

By default, JuMP creates String names for variables and constraints and passes these to the solver. The benefit of passing names is that it improves the readability of log messages from the solver (for example, "variable x has invalid bounds" instead of "variable v1203 has invalid bounds"), but for larger models the overhead of passing names can be non-trivial.

Disable the creation of String names by setting set\_string\_name = false in the @variable and @constraint macros, or by calling set\_string\_names\_on\_creation to disable all names for a particular model:

```
julia> model = Model();
julia> set_string_names_on_creation(model, false)
julia> @variable(model, x)
_[1]
julia> @constraint(model, c, 2x <= 1)
2 _[1] ≤ 1
```

Note that this doesn't change how symbolic names and bindings are stored:

```
julia> x
_[1]
julia> model[:x]
_[1]
julia> x === model[:x]
true
```

But you can no longer look up the variable by the string name:

```
julia> variable_by_name(model, "x") === nothing
true
```

#### Info

For more information on the difference between string names, symbolic names, and bindings, see String names, symbolic names, and bindings.

# 4.10 Performance problems with sum-if formulations

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to explain a common performance issue that can arise with summations like sum(x[a] for a in list if condition(a)). This issue is particularly common in models with graph or network structures.

#### Tip

This tutorial is more advanced than the other "Getting started" tutorials. It's in the "Getting started" section because it is one of the most common causes of performance problems that users experience when they first start using JuMP to write large scale programs. If you are new to JuMP, you may want to briefly skim the tutorial and come back to it once you have written a few JuMP models.

# **Required packages**

This tutorial uses the following packages

using JuMP import Plots

# Data

As a motivating example, we consider a network flow problem, like the examples in Network flow problems or The network multi-commodity flow problem.

Here is a function that builds a random graph. The specifics do not matter.

```
function build_random_graph(num_nodes::Int, num_edges::Int)
 nodes = 1:num_nodes
 edges = Pair{Int,Int}[i - 1 => i for i in 2:num_nodes]
 while length(edges) < num_edges</pre>
 edge = rand(nodes) => rand(nodes)
 if !(edge in edges)
 push!(edges, edge)
 end
 end
 function demand(n)
 if n == 1
 return -1
 elseif n == num_nodes
 return 1
 else
 return 0
 end
 end
 return nodes, edges, demand
end
nodes, edges, demand = build_random_graph(4, 8)
```

(1:4, [1 => 2, 2 => 3, 3 => 4, 2 => 2, 3 => 3, 1 => 1, 3 => 2, 1 => 4], Main.demand)

The goal is to decide the flow of a commodity along each edge in edges to satisfy the demand (n) of each node n in nodes.

The mathematical formulation is:

s.t. 
$$\sum_{(i,n)\in E} x_{i,n} - \sum_{(n,j)\in E} x_{n,j} = d_n \quad \forall n \in N$$
$$x_e \ge 0 \quad \forall e \in E$$

# Naïve model

The first model you might write down is:

```
model = Model()
@variable(model, flows[e in edges] >= 0)
@constraint(
 model,
 [n in nodes],
 sum(flows[(i, j)] for (i, j) in edges if j == n) -
 sum(flows[(i, j)] for (i, j) in edges if i == n) == demand(n)
);
```

The benefit of this formulation is that it looks very similar to the mathematical formulation of a network flow problem.

The downside to this formulation is subtle. Behind the scenes, the JuMP @constraint macro expands to something like:

```
model = Model()
@variable(model, flows[e in edges] >= 0)
for n in nodes
 flow_in = AffExpr(0.0)
 for (i, j) in edges
 if j == n
 add_to_expression!(flow_in, flows[(i, j)])
 end
 end
 flow_out = AffExpr(0.0)
 for (i, j) in edges
 if i == n
 add_to_expression!(flow_out, flows[(i, j)])
 end
 end
 @constraint(model, flow_in - flow_out == demand(n))
end
```

This formulation includes two for-loops, with a loop over every edge (twice) for every node. The big-O notation of the runtime is  $O(|nodes| \times |edges|)$ . If you have a large number of nodes and a large number of edges, the runtime of this loop can be large.

Let's build a function to benchmark our formulation:

```
function build_naive_model(nodes, edges, demand)
model = Model()
@variable(model, flows[e in edges] >= 0)
@constraint(
 model,
 [n in nodes],
 sum(flows[(i, j)] for (i, j) in edges if j == n) -
 sum(flows[(i, j)] for (i, j) in edges if i == n) == demand(n)
)
return model
end
nodes, edges, demand = build_random_graph(1_000, 2_000)
@elapsed build_naive_model(nodes, edges, demand)
```

121

#### 0.125310643

A good way to benchmark is to measure the runtime across a wide range of input sizes. From our big-O analysis, we should expect that doubling the number of nodes and edges results in a 4x increase in the runtime.

```
run_times = Float64[]
factors = 1:10
for factor in factors
 graph = build_random_graph(1_000 * factor, 5_000 * factor)
 push!(run_times, @elapsed build_naive_model(graph...))
end
Plots.plot(; xlabel = "Factor", ylabel = "Runtime [s]")
Plots.scatter!(factors, run_times; label = "Actual")
a, b = hcat(ones(10), factors .^ 2) \ run_times
Plots.plot!(factors, a .+ b * factors .^ 2; label = "Quadratic fit")
```



As expected, the runtimes demonstrate quadratic scaling: if we double the number of nodes and edges, the runtime increases by a factor of four.

# Caching

We can improve our formulation by caching the list of incoming and outgoing nodes for each node n:

```
out_nodes = Dict(n => Int[] for n in nodes)
in_nodes = Dict(n => Int[] for n in nodes)
for (i, j) in edges
 push!(out_nodes[i], j)
 push!(in_nodes[j], i)
end
```

```
end
```

with the corresponding change to our model:

```
model = Model()
@variable(model, flows[e in edges] >= 0)
@constraint(
 model,
 [n in nodes],
 sum(flows[(i, n)] for i in in_nodes[n]) -
 sum(flows[(n, j)] for j in out_nodes[n]) == demand(n)
);
```

The benefit of this formulation is that we now loop over out\_nodes[n] rather than edges for each node n, and so the runtime is O(|edges|).

Let's build a new function to benchmark our formulation:

```
function build cached model(nodes, edges, demand)
 out nodes = Dict(n => Int[] for n in nodes)
 in_nodes = Dict(n => Int[] for n in nodes)
 for (i, j) in edges
 push!(out_nodes[i], j)
 push!(in_nodes[j], i)
 end
 model = Model()
 @variable(model, flows[e in edges] >= 0)
 @constraint(
 model,
 [n in nodes],
 sum(flows[(i, n)] for i in in_nodes[n]) -
 sum(flows[(n, j)] for j in out_nodes[n]) == demand(n)
)
 return model
end
```

```
nodes, edges, demand = build_random_graph(1_000, 2_000)
@elapsed build_cached_model(nodes, edges, demand)
```

# 0.174810538

#### Analysis

Now we can analyse the difference in runtime of the two formulations:

```
run_times_naive = Float64[]
run_times_cached = Float64[]
factors = 1:10
for factor in factors
 graph = build_random_graph(1_000 * factor, 5_000 * factor)
 push!(run_times_naive, @elapsed build_naive_model(graph...))
 push!(run_times_cached, @elapsed build_cached_model(graph...))
end
Plots.plot(; xlabel = "Factor", ylabel = "Runtime [s]")
Plots.scatter!(factors, run_times_naive; label = "Actual")
a, b = hcat(ones(10), factors .^ 2) \ run_times_naive
Plots.plot!(factors, a .+ b * factors .^ 2; label = "Quadratic fit")
Plots.scatter!(factors, run_times_cached; label = "Cached")
a, b = hcat(ones(10), factors) \ run_times_cached
Plots.plot!(factors, a .+ b * factors; label = "Linear fit")
```



Even though the cached model needs to build in\_nodes and out\_nodes, it is asymptotically faster than the naïve model, scaling linearly with factor rather than quadratically.

# Lesson

If you write code with sum-if type conditions, for example, @constraint(model, [a in set], sum(x[b] for b in list if condition(a, b)), you can improve the performance by caching the elements for which condition(a, b) is true.

# CHAPTER 4. GETTING STARTED

Finally, you should understand that this behavior is not specific to JuMP, and that it applies more generally to all computer programs you might write. (Python programs that use Pyomo or gurobipy would similarly benefit from this caching approach.)

Understanding big-O notation and algorithmic complexity is a useful debugging skill to have, regardless of the type of program that you are writing.

# **Chapter 5**

# Transitioning

# 5.1 Transitioning from MATLAB

This tutorial was generated using Literate.jl. Download the source as a .jl file.

YALMIP and CVX are two packages for mathematical optimization in MATLAB®. They are independently developed and are in no way affiliated with JuMP.

The purpose of this tutorial is to help new users to JuMP who have previously used YALMIP or CVX by comparing and contrasting their different features.

Тір

If you have not used Julia before, read the Getting started with Julia tutorial.

#### Namespaces

Julia has namespaces, which MATLAB lacks. Therefore one needs to either use the command:

using JuMP

in order bring all names exported by JuMP into scope, or:

import JuMP

in order to merely make the JuMP package available. import requires prefixing everything you use from JuMP with JuMP.. In this tutorial we use the former.

#### Models

YALMIP and CVX have a single, implicit optimization model that you build by defining variables and constraints.

In JuMP, we create an explicit model first, and then, when you declare variables, constraints, or the objective function, you specify to which model they are being added.

Create a new JuMP model with the command:

model = Model()

```
A JuMP Model
 solver: none
 objective_sense: FEASIBILITY_SENSE
 num_variables: 0
 num_constraints: 0
 L Names registered in the model: none
```

# Variables

In most cases there is a direct translation between variable declarations. The following table shows some common examples:

JuMP	YALMIP	CVX
@variable(model, x)	x = sdpvar	variable x
<pre>@variable(model, x, Int)</pre>	x = intvar	variable x integer
@variable(model, x, Bin)	x = binvar	variable x binary
<pre>@variable(model, v[1:d])</pre>	v = sdpvar(d, 1)	variable v(d)
<pre>@variable(model, m[1:d, 1:d])</pre>	<pre>m = sdpvar(d,d,'full')</pre>	variable m(d, d)
<pre>@variable(model, m[1:d, 1:d] in</pre>	m =	variable m(d,d)
ComplexPlane())	<pre>sdpvar(d,d,'full','complex')</pre>	complex
<pre>@variable(model, m[1:d, 1:d],</pre>	m = sdpvar(d)	variable m(d,d)
Symmetric)		symmetric
<pre>@variable(model, m[1:d, 1:d],</pre>	m =	variable m(d,d)
Hermitian)	<pre>sdpvar(d,d,'hermitian','complexh@rmitian</pre>	

Like CVX, but unlike YALMIP, JuMP can also constrain variables upon creation:

JuMP	CVX
<pre>@variable(model, v[1:d] &gt;= 0) @variable(model, m[1:d, 1:d], PSD) @variable(model, m[1:d, 1:d] in PSDCone()) @variable(model, m[1:d, 1:d] in WarmiticsPCPCase())</pre>	<pre>variable v(d) nonnegative variable m(d,d) semidefinite variable m(d,d) semidefinite variable m(d,d) complex</pre>
HermitianPSDCone())	semidefinite

JuMP can additionally set variable bounds, which may be handled more efficiently by a solver than an equivalent linear constraint. For example:

@variable(model, -1 <= x[i in 1:3] <= i)
upper\_bound.(x)</pre>

3-element Vector{Float64}: 1.0 2.0 3.0 A more interesting case is when you want to declare, for example, n real symmetric matrices. Both YALMIP and CVX allow you to put the matrices as the slices of a 3-dimensional array, via the commands m = sdpvar(d, d, n) and variable m(d, d, n) symmetric, respectively. With JuMP this is not possible. Instead, to achieve the same result one needs to declare a vector of n matrices:

```
d, n = 3, 2
m = [@variable(model, [1:d, 1:d], Symmetric) for _ in 1:n]
```

```
2-element Vector{LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 [_[4] _[5] _[7]; _[5] _[6] _[8]; _[7] _[8] _[9]]
 [_[10] _[11] _[13]; _[11] _[12] _[14]; _[13] _[14] _[15]]
```

#### m[1]

```
3×3 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 _[4] _[5] _[7]
 _[5] _[6] _[8]
 _[7] _[8] _[9]
```

**m[**2]

3×3 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 \_[10] \_[11] \_[13]
 \_[11] \_[12] \_[14]
 \_[13] \_[14] \_[15]

The analogous construct in MATLAB would be a cell array containing the optimization variables, which every discerning programmer avoids as cell arrays are rather slow. This is not a problem in Julia: a vector of matrices is almost as fast as a 3-dimensional array.

#### Constraints

As in the case of variables, in most cases there is a direct translation between the packages:

Like YALMIP and CVX, JuMP is smart enough to not generate redundant constraints when declaring equality constraints between Symmetric or Hermitian matrices. In these cases @constraint(model, m == c) will not generate constraints for the lower diagonal and the imaginary part of the diagonal (in the complex case).

Experienced MATLAB users will probably be relieved to see that you must pass PSDCone() or HermitianPSDCone() to make a matrix positive semidefinite, because the >= ambiguity in YALMIP and CVX is common source of bugs.

#### Setting the objective

Like CVX, but unlike YALMIP, JuMP has a specific command for setting an objective function:

JuMP	YALMIP	CVX
<pre>@constraint(model, v == c)</pre>	V == C	V == C
<pre>@constraint(model, v &gt;= 0)</pre>	v >= 0	v >= 0
<pre>@constraint(model, m &gt;= 0, PSDCone())</pre>	m >= 0	<pre>m == semidefinite(length(m))</pre>
<pre>@constraint(model, m &gt;= 0,</pre>	m >= 0	m ==
HermitianPSDCone())		<pre>hermitian_semidefinite(length(m))</pre>
<pre>@constraint(model, [t; v] in</pre>	cone(v, t)	<pre>{v, t} == lorentz(length(v))</pre>
SecondOrderCone())		
<pre>@constraint(model, [x, y, z] in</pre>	expcone([x,	<pre>{x, y, z} == exponential(1)</pre>
<pre>MOI.ExponentialCone())</pre>	y, z])	

@objective(model, Min, sum(i \* x[i] for i in 1:3))

 $x_1 + 2x_2 + 3x_3$ 

Here the third argument is any expression you want to optimize, and Min is an objective sense (the other possibility is Max).

# Setting solver and options

In order to set an optimizer with JuMP, do:

import Clarabel
set\_optimizer(model, Clarabel.Optimizer)

where "Clarabel" is an example solver. See the list of Supported solvers for other choices.

To configure the solver options you use the command:

set\_attribute(model, "verbose", true)

where verbose is an option specific to Clarabel.

A crucial difference is that with JuMP you must explicitly choose a solver before optimizing. Both YALMIP and CVX allow you to leave it empty and will try to guess an appropriate solver for the problem.

#### Optimizing

Like YALMIP, but unlike CVX, with JuMP you need to explicitly start the optimization, with the command:

optimize!(model)

```
Clarabel.jl v0.10.0 - Clever Acronym
(c) Paul Goulart
University of Oxford, 2022
```

```
problem:
 variables = 15
 constraints = 6
 nnz(P)
 = 0
 nnz(A)
 = 6
 cones (total) = 1
 : Nonnegative = 1, numel = 6
settings:
 linear algebra: direct / qdldl, precision: Float64
 max iter = 200, time limit = Inf, max step = 0.990
 tol_feas = 1.0e-08, tol_gap_abs = 1.0e-08, tol_gap_rel = 1.0e-08,
 static reg : on, \epsilon 1 = 1.0e-08, \epsilon 2 = 4.9e-32
 dynamic reg: on, \varepsilon = 1.0e-13, \delta = 2.0e-07
 iter refine: on, reltol = 1.0e-13, abstol = 1.0e-12,
 max iter = 10, stop ratio = 5.0
 equilibrate: on, min scale = 1.0e-04, max scale = 1.0e+04
 max iter = 10
iter pcost
 dcost gap pres
 dres
 k/t μ
 step

 0 1.0000e+01 -1.2500e+01 2.25e+00 0.00e+00 0.00e+00 1.00e+00 3.36e+00 -----
 1 3.9744e+00 -5.5968e-01 4.53e+00 1.43e-16 1.27e-16 3.10e-01 6.92e-01 8.38e-01
 2 1.1590e-01 -1.2437e-01 2.40e-01 4.88e-17 3.27e-17 2.81e-02 3.83e-02 9.73e-01
 3 1.1746e-03 -1.2507e-03 2.43e-03 1.06e-16 7.36e-17 2.83e-04 3.87e-04 9.90e-01
 4 1.1746e-05 -1.2507e-05 2.43e-05 1.44e-16 3.68e-17 2.83e-06 3.87e-06 9.90e-01
 5 1.1746e-07 -1.2507e-07 2.43e-07 5.05e-15 4.78e-15 2.83e-08 3.87e-08 9.90e-01
 6 1.1746e-09 -1.2507e-09 2.43e-09 1.59e-16 6.59e-17 2.83e-10 3.87e-10 9.90e-01

Terminated with status = solved
solve time = 623\mu s
```

The exclamation mark here is a Julia-ism that means the function is modifying its argument, model.

#### Querying solution status

After the optimization is done, you should check for the solution status to see what solution (if any) the solver found.

Like YALMIP and CVX, JuMP provides a solver-independent way to check it, via the command:

```
is_solved_and_feasible(model)
```

true

If the return value is false, you should investigate with termination\_status, primal\_status, and raw\_status, See Solutions for more details on how to query and interpret solution statuses.

#### **Extracting variables**

Like YALMIP, but unlike CVX, with JuMP you need to explicitly ask for the value of your variables after optimization is done, with the function call value(x) to obtain the value of variable x.

value.(m[1][1, 1])

A subtlety is that, unlike YALMIP, the function value is only defined for scalars. For vectors and matrices you need to use Julia broadcasting: value.(v).

value.(m[1])

0.0

3×3 Matrix{Float64}: 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

There is also a specialized function for extracting the value of the objective, objective\_value(model), which is useful if your objective doesn't have a convenient expression.

objective\_value(model)

-5.99999998825352

#### **Dual variables**

Like YALMIP and CVX, JuMP allows you to recover the dual variables. In order to do that, the simplest method is to name the constraint you're interested in, for example, @constraint(model, bob, sum(v) == 1) and then, after the optimzation is done, call dual(bob). See Duality for more details.

#### **Reformulating problems**

Perhaps the biggest difference between JuMP and YALMIP and CVX is how far the package is willing to go in reformulating the problems you give to it.

CVX is happy to reformulate anything it can, even using approximations if your solver cannot handle the problem.

YALMIP will only do exact reformulations, but is still fairly adventurous, for example, being willing to reformulate a nonlinear objective in terms of conic constraints.

JuMP does no such thing: it only reformulates objectives into objectives, and constraints into constraints, and is fairly conservative at that. As a result, you might need to do some reformulations manually, for which a good guide is the Modeling with cones tutorial.

# Vectorization

In MATLAB, it is absolutely essential to "vectorize" your code to obtain acceptable performance. This is because MATLAB is a slow interpreted language, which sends your commands to fast libraries. When you "vectorize" your code you are minimizing the MATLAB part of the work and sending it to the fast libraries instead.

There's no such duality with Julia.

Everything you write and most libraries you use will compile down to LLVM, so "vectorization" has no effect.

For example, if you are writing a linear program in MATLAB and instead of the usual constraints =  $[v \ge 0]$  you write:

```
for i = 1:n
 constraints = [constraints, v(i) >= 0];
end
```

performance will be poor.

With Julia, on the other hand, there is hardly any difference between

 $(constraint(model, v \ge 0))$ 

and

```
for i in 1:n
 @constraint(model, v[i] >= 0)
end
```

#### Symmetric and Hermitian matrices

Julia has specialized support for symmetric and Hermitian matrices in the LinearAlgebra package:

import LinearAlgebra

If you have a matrix that is numerically symmetric:

x = [1 2; 2 3]

2×2 Matrix{Int64}:
 1 2
 2 3

LinearAlgebra.issymmetric(x)

true

then you can wrap it in a LinearAlgebra.Symmetric matrix to tell Julia's type system that the matrix is symmetric.

LinearAlgebra.Symmetric(x)

```
2×2 LinearAlgebra.Symmetric{Int64, Matrix{Int64}}:
1 2
2 3
```

Using a Symmetric matrix lets Julia and JuMP use more efficient algorithms when they are working with symmetric matrices.

If you have a matrix that is nearly but not exactly symmetric:

x = [1.0 2.0; 2.001 3.0] LinearAlgebra.issymmetric(x)

false

then you could, as you might do in MATLAB, make it numerically symmetric as follows:

 $x_sym = 0.5 * (x + x')$ 

```
2×2 Matrix{Float64}:
1.0 2.0005
2.0005 3.0
```

In Julia, you can explicitly choose whether to use the lower or upper triangle of the matrix:

```
x_sym = LinearAlgebra.Symmetric(x, :L)
```

x\_sym = LinearAlgebra.Symmetric(x, :U)

```
2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
1.0 2.0
2.0 3.0
```

The same applies for Hermitian matrices, using LinearAlgebra.Hermitian and LinearAlgebra.ishermitian.

#### Primal versus dual form

When you translate some optimization problems from YALMIP or CVX to JuMP, you might be surprised to see it get much faster or much slower, even if you're using exactly the same solver. The most likely reason is that YALMIP will always interpret the problem as the dual form, whereas CVX and JuMP will try to interpret the problem in the form most appropriate to the solver. If the problem is more naturally formulated in the primal form it is likely that YALMIP's performance will suffer, or if JuMP gets it wrong, its performance will suffer. It might be worth trying both primal and dual forms if you're having trouble, which can be done automatically with the package Dualization.jl.

For an in-depth explanation of this issue, see the Dualization tutorial.

# **Rosetta stone**

In this section, we show a complete example of the same optimization problem being solved with JuMP, YALMIP, and CVX. It is a semidefinite program that computes a lower bound on the random robustness of entanglement using the partial transposition criterion.

The code is complete, apart from the function that does partial transposition. With both YALMIP and CVX we use the function PartialTranspose from QETLAB. With JuMP, we could use the function Convex.partialtranspose from Convex.jl, but we reproduce it here for simplicity:

```
function partial_transpose(x::AbstractMatrix, sys::Int, dims::Vector)
 @assert size(x, 1) == size(x, 2) == prod(dims)
 @assert 1 <= sys <= length(dims)
 n = length(dims)
 s = n - sys + 1
 p = collect(1:2n)
 p[s], p[n+s] = n + s, s
 r = reshape(x, (reverse(dims)..., reverse(dims)...))
 return reshape(permutedims(r, p), size(x))
end</pre>
```

partial\_transpose (generic function with 1 method)

#### JuMP

The JuMP code to solve this problem is:

using JuMP import Clarabel import LinearAlgebra

```
function random_state_pure(d)
 x = randn(Complex{Float64}, d)
 y = x * x'
 return LinearAlgebra.Hermitian(y / LinearAlgebra.tr(y))
end
function robustness_jump(d)
 rho = random_state_pure(d^2)
 id = LinearAlgebra.Hermitian(LinearAlgebra.I(d²))
 rhoT = LinearAlgebra.Hermitian(partial_transpose(rho, 1, [d, d]))
 model = Model(Clarabel.Optimizer)
 set_attribute(model, "verbose", false)
 @variable(model, \lambda)
 @constraint(model, PPT, rhoT + \lambda * id in HermitianPSDCone())
 (dobjective(model, Min, \lambda)
 optimize!(model)
 assert_is_solved_and_feasible(model; allow_almost = true)
 WT = dual(PPT)
 return value(λ), real(LinearAlgebra.dot(WT, rhoT))
end
```

```
robustness_jump(3)
```

(0.43174978781546347, -0.43174978731622865)

#### YALMIP

The corresponding YALMIP code is:

```
function robustness_yalmip(d)
 rho = random_state_pure(d^2);
 % PartialTranspose from https://github.com/nathanieljohnston/QETLAB
 rhoT = PartialTranspose(rho, 1, [d d]);
 lambda = sdpvar;
 constraints = [(rhoT + lambda*eye(d^2) >= 0):'PPT'];
 ops = sdpsettings(sdpsettings, 'verbose', 0, 'solver', 'sedumi');
 sol = optimize(constraints, lambda, ops);
 if sol.problem == 0
 WT = dual(constraints('PPT'));
 value(lambda)
 real(WT(:)' * rhoT(:))
 else
 display(['Something went wrong: ', sol.info])
 end
end
function rho = random_state_pure(d)
 x = randn(d, 1) + 1i * randn(d, 1);
 y = x * x';
 rho = y / trace(y);
end
```

# сvх

The corresponding CVX code is:

```
function robustness_cvx(d)
 rho = random_state_pure(d^2);
 % PartialTranspose from https://github.com/nathanieljohnston/QETLAB
 rhoT = PartialTranspose(rho, 1, [d d]);
 cvx begin
 variable lambda
 dual variable WT
 WT : rhoT + lambda * eye(d^2) == hermitian_semidefinite(d^2)
 minimise lambda
 cvx_end
 if strcmp(cvx_status, 'Solved')
 lambda
 real(WT(:)' * rhoT(:))
 else
 display('Something went wrong.')
 end
end
function rho = random_state_pure(d)
 x = randn(d, 1) + 1i * randn(d, 1);
 y = x * x';
 rho = y / trace(y);
end
```
# **Chapter 6**

# Linear programs

# 6.1 Introduction

Linear programs (LPs) are a fundamental class of optimization problems of the form:

$$\min_{x \in \mathbb{R}^n} \sum_{i=1}^n c_i x_i \tag{6.1}$$

$$\mathsf{s.t.}l_j \le \sum_{i=1}^n a_{ij} x_i \le u_j \qquad \qquad j = 1 \dots m \tag{6.2}$$

$$l_i \le x_i \le u_i \qquad \qquad i = 1 \dots n. \tag{6.3}$$

The most important thing to note is that all terms are of the form coefficient \* variable, and that there are no nonlinear terms or multiplications between variables.

Mixed-integer linear programs (MILPs) are extensions of linear programs in which some (or all) of the decision variables take discrete values.

#### How to choose a solver

Almost all solvers support linear programs; look for "LP" in the list of Supported solvers. However, fewer solvers support mixed-integer linear programs. Solvers supporting discrete variables start with "(MI)" in the list of Supported solvers.

#### How these tutorials are structured

Having a high-level overview of how this part of the documentation is structured will help you know where to look for certain things.

- The following tutorials are worked examples that present a problem in words, then formulate it in mathematics, and then solve it in JuMP. This usually involves some sort of visualization of the solution. Start here if you are new to JuMP.
  - The diet problem
  - The cannery problem
  - The facility location problem

- Financial modeling problems
- Network flow problems
- N-Queens
- Sudoku
- The Tips and tricks tutorial contains a number of helpful reformulations and tricks you can use when modeling linear programs. Look here if you are stuck trying to formulate a problem as a linear program.
- The Sensitivity analysis of a linear program tutorial explains how to create sensitivity reports like those produced by the Excel Solver.
- The Callbacks tutorial explains how to write a variety of solver-independent callbacks. Look here if you want to write a callback.
- The remaining tutorials are less verbose and styled in the form of short code examples. These tutorials have less explanation, but may contain useful code snippets, particularly if they are similar to a problem you are trying to solve.

## 6.2 The knapsack problem example

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to demonstrate how to formulate and solve a simple optimization problem.

#### **Required packages**

This tutorial requires the following packages:

using JuMP import HiGHS

# Formulation

The knapsack problem is a classical optimization problem: given a set of items and a container with a fixed capacity, choose a subset of items having the greatest combined value that will fit within the container without exceeding the capacity.

The name of the problem suggests its analogy to packing for a trip, where the baggage weight limit is the capacity and the goal is to pack the most profitable combination of belongings.

We can formulate the knapsack problem as the integer linear program:

$$\max \sum_{i=1}^{n} c_i x_i$$
  
s.t. 
$$\sum_{i=1}^{n} w_i x_i \le C,$$
$$x_i \in \{0,1\}, \quad \forall i = 1, \dots, n,$$

where C is the capacity, and there is a choice between n items, with item i having weight  $w_i$ , profit  $c_i$ . Decision variable  $x_i$  is equal to 1 if the item is chosen and 0 if not.

This formulation can be written more compactly as:

$$\max c^{\top} x$$
  
s.t.  $w^{\top} x \le C$   
 $x$  binary.

## Data

The data for the problem consists of two vectors (one for the profits and one for the weights) along with a capacity.

There are five objects:

n = 5;

For our example, we use a capacity of 10 units:

capacity = 10.0;

and the profit and cost data:

profit = [5.0, 3.0, 2.0, 7.0, 4.0]; weight = [2.0, 8.0, 4.0, 2.0, 5.0];

#### **JuMP formulation**

Let's begin constructing the JuMP model for our knapsack problem.

First, we'll create a Model object for holding model elements as we construct each part. We'll also set the solver that will ultimately be called to solve the model, once it's constructed.

model = Model(HiGHS.Optimizer)

```
A JuMP Model

 solver: HiGHS

 objective_sense: FEASIBILITY_SENSE

 hum_variables: 0

 hum_constraints: 0

 Vames registered in the model: none
```

Next we need the decision variables representing which items are chosen:

@variable(model, x[1:n], Bin)

```
5-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
 x[4]
 x[5]
```

We now want to constrain those variables so that their combined weight is less than or equal to the given capacity:

@constraint(model, sum(weight[i] \* x[i] for i in 1:n) <= capacity)</pre>

 $2x_1 + 8x_2 + 4x_3 + 2x_4 + 5x_5 \le 10$ 

Finally, our objective is to maximize the combined profit of the chosen items:

@objective(model, Max, sum(profit[i] \* x[i] for i in 1:n))

 $5x_1 + 3x_2 + 2x_3 + 7x_4 + 4x_5$ 

Let's print a human-readable description of the model and check that the model looks as expected:

print(model)

Max 5 x[1] + 3 x[2] + 2 x[3] + 7 x[4] + 4 x[5]Subject to 2  $x[1] + 8 x[2] + 4 x[3] + 2 x[4] + 5 x[5] \le 10$ x[1] binary x[2] binary x[3] binary x[4] binary x[5] binary

We can now solve the optimization problem and inspect the results.

optimize!(model)
assert\_is\_solved\_and\_feasible(model)
solution\_summary(model)

```
solution_summary(; result = 1, verbose = false)
+ solver_name : HiGHS
+ Termination
| + termination_status : OPTIMAL
| + result_count : 1
```

```
| | raw_status : kHighsModelStatusOptimal
| L objective_bound : 1.60000e+01
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : N0_SOLUTION
| | objective_value : 1.60000e+01
| | dual_objective_value : NaN
| L relative_gap : 0.00000e+00
L Work counters
| solve_time (sec) : 5.80549e-04
| simplex_iterations : 1
| barrier_iterations : -1
L node_count : 1
```

The items chosen are

items\_chosen = [i for i in 1:n if value(x[i]) > 0.5]

3-element Vector{Int64}:
1
4
5

## Writing a function

After working interactively, it is good practice to implement your model in a function.

The function can be used to ensure that the model is given well-defined input data with validation checks, and that the solution process went as expected.

```
function solve_knapsack_problem(;
 profit::Vector{Float64},
 weight::Vector{Float64},
 capacity::Float64,
)
 n = length(weight)
 # The profit and weight vectors must be of equal length.
 @assert length(profit) == n
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, x[1:n], Bin)
 @objective(model, Max, profit' * x)
 @constraint(model, weight' * x <= capacity)</pre>
 optimize!(model)
 assert_is_solved_and_feasible(model)
 println("Objective is: ", objective_value(model))
 println("Solution is:")
 for i in 1:n
 print("x[$i] = ", round(Int, value(x[i])))
 println(", c[$i] / w[$i] = ", profit[i] / weight[i])
```

```
end
chosen_items = [i for i in 1:n if value(x[i]) > 0.5]
return chosen_items
end
```

solve\_knapsack\_problem(; profit = profit, weight = weight, capacity = capacity)

```
3-element Vector{Int64}:
1
4
5
```

We observe that the chosen items (1, 4, and 5) have the best profit to weight ratio in this particular example.

#### Next steps

Here are some things to try next:

- Call the function with different data. What happens as the capacity increases?
- What happens if the profit and weight vectors are different lengths?
- Instead of creating a binary variable with Bin, we could have written @variable(model, 0 <= x[1:n]</li>
   = 1, Int). Verify that this formulation finds the same solution. What happens if we are allowed to take more than one of each item?

## 6.3 The diet problem

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to demonstrate how to incorporate DataFrames into a JuMP model. As an example, we use classic Stigler diet problem.

#### **Required packages**

This tutorial requires the following packages:

```
using JuMP
import CSV
import DataFrames
import HiGHS
import Test
```

#### Formulation

We wish to cook a nutritionally balanced meal by choosing the quantity of each food f to eat from a set of foods F in our kitchen.

Each food f has a cost,  $c_f$ , as well as a macro-nutrient profile  $a_{m,f}$  for each macro-nutrient  $m \in M$ .

Because we care about a nutritionally balanced meal, we set some minimum and maximum limits for each nutrient, which we denote  $l_m$  and  $u_m$  respectively.

Furthermore, because we are optimizers, we seek the minimum cost solution.

With a little effort, we can formulate our dinner problem as the following linear program:

$$\min \sum_{f \in F} c_f x_f$$
  
s.t.  $l_m \leq \sum_{f \in F} a_{m,f} x_f \leq u_m, \quad \forall m \in M$   
 $x_f \geq 0, \qquad \forall f \in F.$ 

In the rest of this tutorial, we will create and solve this problem in JuMP, and learn what we should cook for dinner.

#### Data

First, we need some data for the problem. For this tutorial, we'll write CSV files to a temporary directory from Julia. If you have existing files, you could change the filenames to point to them instead.

```
dir = mktempdir()
```

#### "/tmp/jl\_yP2LWr"

The first file is a list of foods with their macro-nutrient profile:

```
food_csv_filename = joinpath(dir, "diet_foods.csv")
open(food_csv_filename, "w") do io
 write(
 io,

 name,cost,calories,protein,fat,sodium
 hamburger, 2.49, 410, 24, 26, 730
 chicken, 2.89, 420, 32, 10, 1190
 hot dog, 1.50, 560, 20, 32, 1800
 fries, 1.89, 380, 4, 19, 270
 macaroni,2.09,320,12,10,930
 pizza, 1.99, 320, 15, 12, 820
 salad, 2.49, 320, 31, 12, 1230
 milk,0.89,100,8,2.5,125
 ice cream, 1.59, 330, 8, 10, 180
 ••••,
)
 return
end
foods = CSV.read(food_csv_filename, DataFrames.DataFrame)
```

Here, F is foods.name and  $c_f$  is foods.cost. (We're also playing a bit loose the term "macro-nutrient" by including calories and sodium.)

	name	cost	calories	protein	fat	sodium
	String15	Float64	Int64	Int64	Float64	Int64
1	hamburger	2.49	410	24	26.0	730
2	chicken	2.89	420	32	10.0	1190
3	hot dog	1.5	560	20	32.0	1800
4	fries	1.89	380	4	19.0	270
5	macaroni	2.09	320	12	10.0	930
6	pizza	1.99	320	15	12.0	820
7	salad	2.49	320	31	12.0	1230
8	milk	0.89	100	8	2.5	125
9	ice cream	1.59	330	8	10.0	180

We also need our minimum and maximum limits:

```
nutrient_csv_filename = joinpath(dir, "diet_nutrient.csv")
open(nutrient_csv_filename, "w") do io
 write(
 io,
 """
 nutrient,min,max
 calories,1800,2200
 protein,91,
 fat,0,65
 sodium,0,1779
 """,
)
 return
end
```

limits = CSV.read(nutrient\_csv\_filename, DataFrames.DataFrame)

	nutrient	min	max
	String15	Int64	Int64?
1	calories	1800	2200
2	protein	91	missing
3	fat	0	65
4	sodium	0	1779

Protein is missing data for the maximum. Let's fix that using coalesce:

```
limits.max = coalesce.(limits.max, Inf)
limits
```

	nutrient	min	max
	String15	Int64	Real
1	calories	1800	2200
2	protein	91	Inf
3	fat	0	65
4	sodium	0	1779

## **JuMP** formulation

Now we're ready to convert our mathematical formulation into a JuMP model.

First, create a new JuMP model. Since we have a linear program, we'll use HiGHS as our optimizer:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
```

Next, we create a set of decision variables x, with one element for each row in the DataFrame, and each x has a lower bound of  $\theta$ :

@variable(model, x[foods.name] >= 0)

To simplify things later on, we store the vector as a new column x in the DataFrame foods. Since x is a DenseAxisArray, we first need to convert it to an Array:

foods.x = Array(x)

```
9-element Vector{VariableRef}:
 x[hamburger]
 x[chicken]
 x[hot dog]
 x[fries]
 x[macaroni]
 x[pizza]
 x[salad]
 x[milk]
 x[ice cream]
```

Our objective is to minimize the total cost of purchasing food:

@objective(model, Min, sum(foods.cost .\* foods.x));

For the next component, we need to add a constraint that our total intake of each component is within the limits contained in the limits DataFrame:

```
@constraint(
 model,
 [row in eachrow(limits)],
 row.min <= sum(foods[!, row.nutrient] .* foods.x) <= row.max,
);</pre>
```

What does our model look like?

print(model)

```
Min 2.49 x[hamburger] + 2.89 x[chicken] + 1.5 x[hot dog] + 1.89 x[fries] + 2.09 x[macaroni] + 1.99
→ x[pizza] + 2.49 x[salad] + 0.89 x[milk] + 1.59 x[ice cream]
Subject to
410 x[hamburger] + 420 x[chicken] + 560 x[hot dog] + 380 x[fries] + 320 x[macaroni] + 320 x[pizza]
 → + 320 x[salad] + 100 x[milk] + 330 x[ice cream] ∈ [1800, 2200]
24 x[hamburger] + 32 x[chicken] + 20 x[hot dog] + 4 x[fries] + 12 x[macaroni] + 15 x[pizza] + 31
 \hookrightarrow x[salad] + 8 x[milk] + 8 x[ice cream] \in [91, Inf]
26 x[hamburger] + 10 x[chicken] + 32 x[hot dog] + 19 x[fries] + 10 x[macaroni] + 12 x[pizza] + 12
 \hookrightarrow x[salad] + 2.5 x[milk] + 10 x[ice cream] \in [0, 65]
730 x[hamburger] + 1190 x[chicken] + 1800 x[hot dog] + 270 x[fries] + 930 x[macaroni] + 820

 → x[pizza] + 1230 x[salad] + 125 x[milk] + 180 x[ice cream] ∈ [0, 1779]

x[hamburger] \ge 0
x[chicken] \ge 0
x[hot dog] \ge 0
x[fries] \ge 0
x[macaroni] \ge 0
x[pizza] \ge 0
x[salad] \ge 0
x[milk] \ge 0
x[ice cream] \ge 0
```

#### Solution

Let's optimize and take a look at the solution:

```
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : HiGHS
| Termination
| | termination_status : OPTIMAL
| | result_count : 1
| | raw_status : kHighsModelStatusOptimal
```

```
| L objective_bound : 1.18289e+01
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | dual_objective_value : 1.18289e+01
| L dual_objective_value : 1.18289e+01
| L relative_gap : 0.00000e+00
L Work counters
| solve_time (sec) : 2.11477e-04
| simplex_iterations : 6
| barrier_iterations : 0
L node_count : -1
```

We found an optimal solution. Let's see what the optimal solution is:

```
for row in eachrow(foods)
 println(row.name, " = ", value(row.x))
end
```

```
hamburger = 0.6045138888888871

chicken = 0.0

hot dog = 0.0

fries = 0.0

macaroni = 0.0

pizza = 0.0

salad = 0.0

milk = 6.9701388888888935

ice cream = 2.591319444444447
```

That's a lot of milk and ice cream, and sadly, we only get 0.6 of a hamburger.

We can also use the function Containers.rowtable to easily convert the result into a DataFrame:

table = Containers.rowtable(value, x; header = [:food, :quantity])
solution = DataFrames.DataFrame(table)

	food	quantity
	String15	Float64
1	hamburger	0.604514
2	chicken	0.0
3	hot dog	0.0
4	fries	0.0
5	macaroni	0.0
6	pizza	0.0
7	salad	0.0
8	milk	6.97014
9	ice cream	2.59132

This makes it easy to perform analyses our solution:

```
filter!(row -> row.quantity > 0.0, solution)
```

	food	quantity
	String15	Float64
1	hamburger	0.604514
2	milk	6.97014
3	ice cream	2.59132

## **Problem modification**

JuMP makes it easy to take an existing model and modify it by adding extra constraints. Let's see what happens if we add a constraint that we can buy at most 6 units of milk or ice cream combined.

```
dairy_foods = ["milk", "ice cream"]
is_dairy = map(name -> name in dairy_foods, foods.name)
dairy_constraint = @constraint(model, sum(foods[is_dairy, :x]) <= 6)
optimize!(model)
Test.@test !is_solved_and_feasible(model)
Test.@test termination_status(model) == INFEASIBLE
Test.@test primal_status(model) == NO_SOLUTION
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
F solver_name : HiGHS
├ Termination
| + termination_status : INFEASIBLE
| | result_count : 1
| | raw_status : kHighsModelStatusInfeasible
| L objective bound : 0.00000e+00
Folution (result = 1)
| | primal_status : NO_SOLUTION
| | dual_status : INFEASIBILITY_CERTIFICATE
| | objective_value : 1.18289e+01
| | dual_objective_value : 3.56146e+00
| ^L relative_gap : Inf
^L Work counters
 solve_time (sec) : 1.45435e-04
 > simplex_iterations : 0
 barrier_iterations : 0
 L node_count
 : -1
```

There exists no feasible solution to our problem. Looks like we're stuck eating ice cream for dinner.

#### Next steps

- You can delete a constraint using delete(model, dairy\_constraint). Can you add a different constraint to provide a diet with less dairy?
- Some food items (like hamburgers) are discrete. You can use set\_integer to force a variable to take integer values. What happens to the solution if you do?

# 6.4 The cannery problem

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by Louis Luangkesorn.

This tutorial solves the cannery problem from Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963. This class of problem is known as a transshipment problem.

The purpose of this tutorial is to demonstrate how to use JSON data in the formulation of a JuMP model.

## **Required packages**

This tutorial requires the following packages:

using JuMP import HiGHS import JSON import Test

## Formulation

The cannery problem assumes we are optimizing the shipment of cases of cans from production plants  $p \in P$  to markets  $m \in M$ .

Each production plant p has a capacity  $c_p$ , and each market m has a demand  $d_m$ . The shipping cost per case of cans from plant p to market m is  $d_{p,m}$ .

We wish to find the distribution plan  $x_{p,m}$ , the number of cases of cans to ship from plant p to market m, for  $p \in P$  and  $m \in M$  that minimizes the shipping costs. We can formulate our problem as the following linear program:

$$\begin{split} \min \sum_{p \in P} \sum_{m \in M} d_{p,m} x_{p,m} \\ \text{s.t.} \sum_{m \in M} x_{p,m} \leq c_p, \qquad \forall p \in P \\ \sum_{p \in P} x_{p,m} \geq d_m, \qquad \forall m \in M \\ x_{p,m} \geq 0, \qquad \forall p \in P, m \in M \end{split}$$

#### Data

A key feature of the tutorial is to demonstrate how to load data from JSON.

For simplicity, we've hard-coded it below. But if the data was available as a .json file, we could use data = JSON.parsefile(filename) to read in the data.

```
data = JSON.parse("""
{
 "plants": {
 "Seattle": {"capacity": 350},
 "San-Diego": {"capacity": 600}
 },
```

```
"markets": {
 "New-York": {"demand": 300},
 "Chicago": {"demand": 300},
 "Topeka": {"demand": 300}
 },
 "distances": {
 "Seattle => New-York": 2.5,
 "Seattle => Chicago": 1.7,
 "Seattle => Topeka": 1.8,
 "San-Diego => New-York": 2.5,
 "San-Diego => Chicago": 1.8,
 "San-Diego => Topeka": 1.4
 }
}
```

```
Dict{String, Any} with 3 entries:
 "plants" => Dict{String, Any}("Seattle"=>Dict{String, Any}("capacity"=>350...
 "distances" => Dict{String, Any}("San-Diego => New-York"=>2.5, "Seattle => To...
 "markets" => Dict{String, Any}("Chicago"=>Dict{String, Any}("demand"=>300),...
```

Create the set of plants:

```
P = keys(data["plants"])
```

```
KeySet for a Dict{String, Any} with 2 entries. Keys:
 "Seattle"
 "San-Diego"
```

Create the set of markets:

M = keys(data["markets"])

```
KeySet for a Dict{String, Any} with 3 entries. Keys:
 "Chicago"
 "Topeka"
 "New-York"
```

We also need a function to compute the distance from plant to market:

distance(p::String, m::String) = data["distances"]["\$(p) => \$(m)"]

distance (generic function with 1 method)

#### **JuMP** formulation

Now we're ready to convert our mathematical formulation into a JuMP model.

First, create a new JuMP model. Since we have a linear program, we'll use HiGHS as our optimizer:

```
model = Model(HiGHS.Optimizer)
```

```
A JuMP Model
 solver: HiGHS
 objective_sense: FEASIBILITY_SENSE
 num_variables: 0
 num_constraints: 0
 L Names registered in the model: none
```

Our decision variables are indexed over the set of plants and markets:

@variable(model, x[P, M] >= 0)

```
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
 Dimension 1, ["Seattle", "San-Diego"]
 Dimension 2, ["Chicago", "Topeka", "New-York"]
And data, a 2×3 Matrix{VariableRef}:
 x[Seattle,Chicago] x[Seattle,Topeka] x[Seattle,New-York]
 x[San-Diego,Chicago] x[San-Diego,Topeka] x[San-Diego,New-York]
```

We need a constraint that each plant can ship no more than its capacity:

@constraint(model, [p in P], sum(x[p, :]) <= data["plants"][p]["capacity"])</pre>

```
1-dimensional DenseAxisArray{ConstraintRef{Model,

→ MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

→ MathOptInterface.LessThan{Float64}, ScalarShape},1,...} with index sets:

Dimension 1, ["Seattle", "San-Diego"]

And data, a 2-element Vector{ConstraintRef{Model,

→ MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

→ MathOptInterface.LessThan{Float64}}, ScalarShape}}:

x[Seattle,Chicago] + x[Seattle,Topeka] + x[Seattle,New-York] ≤ 350

x[San-Diego,Chicago] + x[San-Diego,Topeka] + x[San-Diego,New-York] ≤ 600
```

@constraint(model, [m in M], sum(x[:, m]) >= data["markets"][m]["demand"])

```
1-dimensional DenseAxisArray{ConstraintRef{Model,

→ MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

→ MathOptInterface.GreaterThan{Float64}}, ScalarShape},1,...} with index sets:

Dimension 1, ["Chicago", "Topeka", "New-York"]

And data, a 3-element Vector{ConstraintRef{Model,

→ MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

→ MathOptInterface.GreaterThan{Float64}}, ScalarShape}:

x[Seattle,Chicago] + x[San-Diego,Chicago] ≥ 300

x[Seattle,Topeka] + x[San-Diego,Topeka] ≥ 300

x[Seattle,New-York] + x[San-Diego,New-York] ≥ 300
```

Finally, our objective is to minimize the transportation distance:

@objective(model, Min, sum(distance(p, m) \* x[p, m] for p in P, m in M));

## Solution

Let's optimize and look at the solution:

optimize!(model)
solution\_summary(model)

```
solution_summary(; result = 1, verbose = false)
- solver name
 : HiGHS
- Termination
| + result_count : 1
| + raw_status : kHighsModelStatusOptimal
| ^L objective_bound : 1.68000e+03
Solution (result = 1)
| + primal_status : FEASIBLE_POINT
| + dual_status : FEASIBLE_POINT
| + objective_value : 1.68000e+03
| | dual_objective_value : 1.68000e+03
| ^L relative_gap : 0.00000e+00
^L Work counters
 - solve time (sec) : 2.07663e-04
 simplex_iterations : 3
 barrier_iterations : 0
 ^L node_count
 : -1
```

What's the optimal shipment?

```
assert_is_solved_and_feasible(model)
for p in P, m in M
 println(p, " => ", m, ": ", value(x[p, m]))
end
```

```
Seattle => Chicago: 300.0
Seattle => Topeka: 0.0
Seattle => New-York: 0.0
San-Diego => Chicago: 0.0
San-Diego => Topeka: 300.0
San-Diego => New-York: 300.0
```

# 6.5 The factory schedule example

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by @Crghilardi.

This tutorial is a Julia translation of Part 5 from Introduction to Linear Programming with Python.

The purpose of this tutorial is to demonstrate how to use DataFrames and delimited files, and to structure your code that is robust to infeasibilities and permits running with different datasets.

#### **Required packages**

This tutorial requires the following packages:

```
using JuMP
import CSV
import DataFrames
import HiGHS
import StatsPlots
```

#### Formulation

The Factory Scheduling Problem assumes we are optimizing the production of a good from factories  $f \in F$  over the course of 12 months  $m \in M$ .

If a factory f runs during a month m, a fixed cost of  $a_f$  is incurred, the factory must produce  $x_{m,f}$  units that is within some minimum and maximum production levels  $l_f$  and  $u_f$  respectively, and each unit of production incurs a variable cost  $c_f$ . Otherwise, the factory can be shut for the month with zero production and no fixedcost is incurred. We denote the run/not-run decision by  $z_{m,f} \in \{0,1\}$ , where  $z_{m,f}$  is 1 if factory f runs in month m. The factory must produce enough units to satisfy demand  $d_m$ .

With a little effort, we can formulate our problem as the following linear program:

$$\begin{split} \min \sum_{f \in F, m \in M} a_f z_{m,f} + c_f x_{m,f} \\ \text{s.t.} x_{m,f} &\leq u_f z_{m,f} & \forall f \in F, m \in M \\ x_{m,f} &\geq l_f z_{m,f} & \forall f \in F, m \in M \\ \sum_{f \in F} x_{m,f} &= d_m & \forall f \in F, m \in M \\ z_{m,f} \in \{0,1\} & \forall f \in F, m \in M. \end{split}$$

However, this formulation has a problem: if demand is too high, we may be unable to satisfy the demand constraint, and the problem will be infeasible.

#### Tip

When modeling, consider ways to formulate your model such that it always has a feasible solution. This greatly simplifies debugging data errors that would otherwise result in an infeasible solution. In practice, most practical decisions have a feasible solution. In our case, we could satisfy demand (at a high cost) by buying replacement items for the buyer, or running the factories in overtime to make up the difference.

We can improve our model by adding a new variable,  $\delta_m$ , which represents the quantity of unmet demand in each month m. We penalize  $\delta_m$  by an arbitrarily large value of \$10,000/unit in the objective.

$$\begin{split} \min \sum_{f \in F, m \in M} a_f z_{m,f} + c_f x_{m,f} + \sum_{m \in M} 10000 \delta_m \\ \text{s.t.} x_{m,f} &\leq u_f z_{m,f} & \forall f \in F, m \in M \\ x_{m,f} &\geq l_f z_{m,f} & \forall f \in F, m \in M \\ \sum_{f \in F} x_{m,f} - \delta_m &= d_m & \forall f \in F, m \in M \\ z_{m,f} \in \{0,1\} & \forall f \in F, m \in M \\ \delta_m &\geq 0 & \forall m \in M. \end{split}$$

# Data

The JuMP GitHub repository contains two text files with the data we need for this tutorial.

The first file contains a dataset of our factories, A and B, with their production and cost levels for each month. For the documentation, the file is located at:

factories\_filename = joinpath(@\_\_DIR\_\_, "factory\_schedule\_factories.txt");

To run locally, download factory\_schedule\_factories.txt and update factories\_filename appropriately.

The file has the following contents:

print(read(factories\_filename, String))

factory	month	<pre>min_production</pre>	<pre>max_production</pre>	fixed_cost	variable_cost
Α	1	20000	100000	500	10
Α	2	20000	110000	500	11
А	3	20000	120000	500	12
А	4	20000	145000	500	9
А	5	20000	160000	500	8
Α	6	20000	140000	500	8
А	7	20000	155000	500	5
Α	8	20000	200000	500	7
Α	9	20000	210000	500	9
Α	10	20000	197000	500	10
Α	11	20000	80000	500	8
Α	12	20000	150000	500	8
В	1	20000	50000	600	5
В	2	20000	55000	600	4
В	3	20000	60000	600	3
В	4	20000	100000	600	5
В	5	Θ	Θ	Θ	0
В	6	20000	70000	600	6
В	7	20000	60000	600	4
В	8	20000	100000	600	6
В	9	20000	100000	600	8
В	10	20000	100000	600	11
В	11	20000	120000	600	10
В	12	20000	150000	600	12

We use the CSV and DataFrames packages to read it into Julia:

```
factory_df = CSV.read(
 factories_filename,
 DataFrames.DataFrame;
 delim = ' ',
 ignorerepeated = true,
)
```

The second file contains the demand data by month:

demand\_filename = joinpath(@\_\_DIR\_\_, "factory\_schedule\_demand.txt");

To run locally, download factory\_schedule\_demand.txt and update demand\_filename appropriately.

```
demand_df = CSV.read(
 demand_filename,
 DataFrames.DataFrame;
 delim = ' ',
 ignorerepeated = true,
)
```

	factory	month	min_production	max_production	fixed_cost	variable_cost
	String1	Int64	Int64	Int64	Int64	Int64
1	Α	1	20000	100000	500	10
2	А	2	20000	110000	500	11
3	А	3	20000	120000	500	12
4	А	4	20000	145000	500	9
5	А	5	20000	160000	500	8
6	А	6	20000	140000	500	8
7	А	7	20000	155000	500	5
8	А	8	20000	200000	500	7
9	А	9	20000	210000	500	9
10	А	10	20000	197000	500	10
11	А	11	20000	80000	500	8
12	А	12	20000	150000	500	8
13	В	1	20000	50000	600	5
14	В	2	20000	55000	600	4
15	В	3	20000	60000	600	3
16	В	4	20000	100000	600	5
17	В	5	0	0	0	0
18	В	6	20000	70000	600	6
19	В	7	20000	60000	600	4
20	В	8	20000	100000	600	6
21	В	9	20000	100000	600	8
22	В	10	20000	100000	600	11
23	В	11	20000	120000	600	10
24	В	12	20000	150000	600	12
			I			

	month	demand
	Int64	Int64
1	1	120000
2	2	100000
3	3	130000
4	4	130000
5	5	140000
6	6	130000
7	7	150000
8	8	170000
9	9	200000
10	10	190000
11	11	140000
12	12	100000

# Data validation

Before moving on, it's always good practice to validate the data you read from external sources. The more effort you spend here, the fewer issues you will have later. The following function contains a few simple checks, but we could add more. For example, you might want to check that none of the values are too large (or too small), which might indicate a typo or a unit conversion issue (perhaps the variable costs are in \$/1000 units instead of \$/unit).

```
function valiate_data(
 demand_df::DataFrames.DataFrame,
 factory_df::DataFrames.DataFrame,
)
 # Minimum production must not exceed maximum production.
 @assert all(factory_df.min_production .<= factory_df.max_production)
 # Demand, minimum production, fixed costs, and variable costs must all be
 # non-negative.
 @assert all(demand_df.demand .>= 0)
 @assert all(factory_df.min_production .>= 0)
 @assert all(factory_df.fixed_cost .>= 0)
 @assert all(factory_df.variable_cost .>= 0)
 return
end
```

valiate\_data(demand\_df, factory\_df)

#### JuMP formulation

Next, we need to code our JuMP formulation. As shown in Design patterns for larger models, it's always good practice to code your model in a function that accepts well-defined input and returns well-defined output.

```
function solve_factory_scheduling(
 demand_df::DataFrames.DataFrame,
 factory_df::DataFrames.DataFrame,
)
 # Even though we validated the data above, it's good practice to do it here
 # too.
 valiate_data(demand_df, factory_df)
 months, factories = unique(factory_df.month), unique(factory_df.factory)
 model = Model(HiGHS.Optimizer)
 set silent(model)
 @variable(model, status[months, factories], Bin)
 @variable(model, production[months, factories], Int)
 @variable(model, unmet_demand[months] >= 0)
 # We use `eachrow` to loop through the rows of the dataframe and add the
 # relevant constraints.
 for r in eachrow(factory_df)
 m, f = r.month, r.factory
 @constraint(model, production[m, f] <= r.max_production * status[m, f])</pre>
 @constraint(model, production[m, f] >= r.min_production * status[m, f])
 end
 @constraint(
 model,
 [r in eachrow(demand_df)],
 sum(production[r.month, :]) + unmet_demand[r.month] == r.demand,
)
 @objective(
 model,
 Min,
 10_000 * sum(unmet_demand) + sum(
 r.fixed_cost * status[r.month, r.factory] +
 r.variable_cost * production[r.month, r.factory] for
 r in eachrow(factory_df)
```

```
)
)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 schedules = Dict{Symbol, Vector{Float64}}(
 Symbol(f) => value.(production[:, f]) for f in factories
)
 schedules[:unmet_demand] = value.(unmet_demand)
 return (
 termination_status = termination_status(model),
 cost = objective_value(model),
 # This `select` statement re-orders the columns in the DataFrame.
 schedules = DataFrames.select(
 DataFrames.DataFrame(schedules),
 [:unmet_demand, :A, :B],
),
)
end
```

solve\_factory\_scheduling (generic function with 1 method)

# Solution

Now we can call our solve\_factory\_scheduling function using the data we read in above.

```
solution = solve_factory_scheduling(demand_df, factory_df);
```

Let's see what solution contains:

solution.termination\_status

OPTIMAL::TerminationStatusCode = 1

solution.cost

1.29064e7

solution.schedules

These schedules will be easier to visualize as a graph:

	unmet_demand	А	В
	Float64	Float64	Float64
1	0.0	70000.0	50000.0
2	0.0	45000.0	55000.0
3	0.0	70000.0	60000.0
4	0.0	30000.0	100000.0
5	0.0	140000.0	-0.0
6	0.0	60000.0	70000.0
7	0.0	90000.0	60000.0
8	0.0	70000.0	100000.0
9	0.0	100000.0	100000.0
10	0.0	190000.0	-0.0
11	0.0	80000.0	60000.0
12	0.0	100000.0	-0.0

```
StatsPlots.groupedbar(
 Matrix(solution.schedules);
 bar_position = :stack,
 labels = ["unmet demand" "A" "B"],
 xlabel = "Month",
 ylabel = "Production",
 legend = :topleft,
 color = ["#20326c" "#4063d8" "#a0blec"],
)
```



Note that we don't have any unmet demand.

## What happens if demand increases?

Let's run an experiment by increasing the demand by 50% in all time periods:

demand\_df.demand .\*= 1.5

12-element Vector{Float64}: 180000.0 150000.0 195000.0 210000.0 210000.0 255000.0 255000.0 300000.0 285000.0 210000.0 150000.0

Now we resolve the problem:

```
high_demand_solution = solve_factory_scheduling(demand_df, factory_df);
```

and visualize the solution:

```
StatsPlots.groupedbar(
 Matrix(high_demand_solution.schedules);
 bar_position = :stack,
 labels = ["unmet demand" "A" "B"],
 xlabel = "Month",
 ylabel = "Production",
 legend = :topleft,
 color = ["#20326c" "#4063d8" "#a0blec"],
)
```



Uh oh, we can't satisfy all of the demand.

## How sensitive is the solution to changes in variable cost?

Let's run another experiment, this time seeing how the optimal objective value changes as we vary the variable costs of each factory.

First though, let's reset the demand to it's original level:

demand\_df.demand ./= 1.5;

For our experiment, we're going to scale the variable costs of both factories by a set of values from 0.0 to 1.5:

scale\_factors = 0:0.1:1.5

#### 0.0:0.1:1.5

At a high level, we're going to loop over the scale factors for A, then the scale factors for B, rescale the input data, call our solve\_factory\_scheduling example, and then store the optimal objective value in the following cost matrix:

cost = zeros(length(scale\_factors), length(scale\_factors));

Because we're modifying factory\_df in-place, we need to store the original variable costs in a new column:

```
factory_df[!, :old_variable_cost] = copy(factory_df.variable_cost);
```

Then, we need a function to scale the :variable\_cost column for a particular factory by a value scale:

```
function scale_variable_cost(df, factory, scale)
 rows = df.factory .== factory
 df[rows, :variable_cost] .=
 round.(Int, df[rows, :old_variable_cost] .* scale)
 return
end
```

scale\_variable\_cost (generic function with 1 method)

Our experiment is just a nested for-loop, modifying A and B and storing the cost:

```
for (j, a) in enumerate(scale_factors)
 scale_variable_cost(factory_df, "A", a)
 for (i, b) in enumerate(scale_factors)
 scale_variable_cost(factory_df, "B", b)
 cost[i, j] = solve_factory_scheduling(demand_df, factory_df).cost
 end
end
```

Let's visualize the cost matrix:

```
StatsPlots.contour(
 scale_factors,
 scale_factors,
 cost;
 xlabel = "Scale of factory A",
 ylabel = "Scale of factory B",
)
```



What can you infer from the solution?

## Info

The Power Systems tutorial explains a number of other ways you can structure a problem to perform a parametric analysis of the solution. In particular, you can use in-place modification to reduce the time it takes to build and solve the resulting models.

# 6.6 The multi-commodity flow problem

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by Louis Luangkesorn.

This tutorial is a JuMP implementation of the multi-commodity transportation model described in *AMPL: A Modeling Language for Mathematical Programming*, by R. Fourer, D.M. Gay and B.W. Kernighan.

The purpose of this tutorial is to demonstrate creating a JuMP model from an SQLite database.

#### **Required packages**

This tutorial uses the following packages

```
using JuMP
import DataFrames
import HiGHS
import SQLite
```

#### CHAPTER 6. LINEAR PROGRAMS

import Tables
import Test

const DBInterface = SQLite.DBInterface

DBInterface

## Formulation

The multi-commondity flow problem is a simple extension of The transportation problem to multiple types of products. Briefly, we start with the formulation of the transportation problem:

$$\begin{array}{ll} \min & \sum_{i \in O, j \in D} c_{i,j} x_{i,j} \\ s.t. & \sum_{j \in D} x_{i,j} \leq s_i \qquad \forall i \in O \\ & \sum_{i \in O} x_{i,j} = d_j \qquad \forall j \in D \\ & x_{i,j} \geq 0 \quad \forall i \in O, j \in D \end{array}$$

but introduce a set of products P, resulting in:

$$\begin{array}{ll} \min & \displaystyle \sum_{i \in O, j \in D, k \in P} c_{i,j,k} x_{i,j,k} \\ s.t. & \displaystyle \sum_{j \in D} x_{i,j,k} \leq s_{i,k} & \forall i \in O, k \in P \\ & \displaystyle \sum_{i \in O} x_{i,j,k} = d_{j,k} & \forall j \in D, k \in P \\ & \displaystyle x_{i,j,k} \geq 0 & \forall i \in O, j \in D, k \in P \\ & \displaystyle \sum_{k \in P} x_{i,j,k} \leq u_{i,j} & \forall i \in O, j \in D \end{array}$$

Note that the last constraint is new; it says that there is a maximum quantity of goods (of any type) that can be transported from origin i to destination j.

## Data

For the purpose of this tutorial, the JuMP repository contains an example database called multi.sqlite.

```
filename = joinpath(@__DIR__, "multi.sqlite");
```

To run locally, download multi.sqlite and update filename appropriately.

Load the database using SQLite.DB:

db = SQLite.DB(filename)

SQLite.DB("/home/runner/work/JuMP.jl/JuMP.jl/docs/latex\_build/tutorials/linear/multi.sqlite")

A quick way to see the schema of the database is via SQLite.tables:

SQLite.tables(db)

```
5-element Vector{SQLite.DBTable}:
SQLite.DBTable("locations", Tables.Schema:
:location Union{Missing, String}
 Union{Missing, String})
:type
SQLite.DBTable("products", Tables.Schema:
:product Union{Missing, String})
SQLite.DBTable("supply", Tables.Schema:
:origin Union{Missing, String}
 :product Union{Missing, String}
 :supply Union{Missing, Float64})
SQLite.DBTable("demand", Tables.Schema:
 :destination Union{Missing, String}
 :product Union{Missing, String}
 :demand
 Union{Missing, Float64})
SQLite.DBTable("cost", Tables.Schema:
 :origin Union{Missing, String}
 :destination Union{Missing, String}
 :product
 Union{Missing, String}
 :cost
 Union{Missing, Float64})
```

We interact with the database by executing queries, and then piping the results to an appropriate table. One example is a DataFrame:

DBInterface.execute(db, "SELECT \* FROM locations") |> DataFrames.DataFrame

	location	type
	String	String
1	GARY	origin
2	CLEV	origin
3	PITT	origin
4	FRA	destination
5	DET	destination
6	LAN	destination
7	WIN	destination
8	STL	destination
9	FRE	destination
10	LAF	destination

But other table types are supported, such as Tables.rowtable:

```
DBInterface.execute(db, "SELECT * FROM locations") |> Tables.rowtable
```

```
10-element Vector{@NamedTuple{location::String, type::String}}:
 (location = "GARY", type = "origin")
 (location = "CLEV", type = "origin")
 (location = "PITT", type = "origin")
 (location = "FRA", type = "destination")
 (location = "DET", type = "destination")
 (location = "LAN", type = "destination")
 (location = "WIN", type = "destination")
 (location = "STL", type = "destination")
 (location = "FRE", type = "destination")
 (location = "FRE", type = "destination")
 (location = "LAF", type = "destination")
```

A rowtable is a Vector of NamedTuples.

You can construct more complicated SQL queries:

```
origins =
 DBInterface.execute(
 db,
 "SELECT location FROM locations WHERE type = \"origin\"",
) |> Tables.rowtable
```

```
3-element Vector{@NamedTuple{location::String}}:
 (location = "GARY",)
 (location = "CLEV",)
 (location = "PITT",)
```

But for our purpose, we just want the list of strings:

```
origins = map(y -> y.location, origins)
```

```
3-element Vector{String}:
 "GARY"
 "CLEV"
 "PITT"
```

We can compose these two operations to get a list of destinations:

```
destinations =
 DBInterface.execute(
 db,
 "SELECT location FROM locations WHERE type = \"destination\"",
```

) |> Tables.rowtable |> x -> map(y -> y.location, x)

```
7-element Vector{String}:

"FRA"

"DET"

"LAN"

"WIN"

"STL"

"FRE"

"LAF"
```

And a list of products from our products table:

```
products =
 DBInterface.execute(db, "SELECT product FROM products") |>
 Tables.rowtable |>
 x -> map(y -> y.product, x)
```

```
3-element Vector{String}:
 "bands"
 "coils"
 "plate"
```

#### **JuMP** formulation

We start by creating a model and our decision variables:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[origins, destinations, products] >= 0)
```

```
3-dimensional DenseAxisArray{VariableRef,3,...} with index sets:
 Dimension 1, ["GARY", "CLEV", "PITT"]
 Dimension 2, ["FRA", "DET", "LAN", "WIN", "STL", "FRE", "LAF"]
 Dimension 3, ["bands", "coils", "plate"]
And data, a 3×7×3 Array{VariableRef, 3}:
[:, :, "bands"] =
 x[GARY,FRA,bands] x[GARY,DET,bands] ... x[GARY,LAF,bands]
 x[CLEV,FRA,bands] x[CLEV,DET,bands] ... x[GARY,LAF,bands]
 x[CLEV,FRA,bands] x[CLEV,DET,bands] x[CLEV,LAF,bands]
 x[PITT,FRA,bands] x[PITT,DET,bands] x[PITT,LAF,bands]
[:, :, "coils"] =
 x[GARY,FRA,coils] x[GARY,DET,coils] ... x[GARY,LAF,coils]
 x[CLEV,FRA,coils] x[CLEV,DET,coils] ... x[CLEV,LAF,coils]
```

```
x[PITT,FRA,coils] x[PITT,DET,coils] x[PITT,LAF,coils]
[:, :, "plate"] =
x[GARY,FRA,plate] x[GARY,DET,plate] ... x[GARY,LAF,plate]
x[CLEV,FRA,plate] x[CLEV,DET,plate] x[CLEV,LAF,plate]
x[PITT,FRA,plate] x[PITT,DET,plate] x[PITT,LAF,plate]
```

One approach when working with databases is to extract all of the data into a Julia datastructure. For example, let's pull the cost table into a DataFrame and then construct our objective by iterating over the rows of the DataFrame:

```
cost = DBInterface.execute(db, "SELECT * FROM cost") |> DataFrames.DataFrame
@objective(
 model,
 Max,
 sum(r.cost * x[r.origin, r.destination, r.product] for r in eachrow(cost)),
);
```

If we don't want to use a DataFrame, we can use a Tables.rowtable instead:

```
supply = DBInterface.execute(db, "SELECT * FROM supply") |> Tables.rowtable
for r in supply
 @constraint(model, sum(x[r.origin, :, r.product]) <= r.supply)
end</pre>
```

Another approach is to execute the query, and then to iterate through the rows of the query using Tables . rows:

```
demand = DBInterface.execute(db, "SELECT * FROM demand")
for r in Tables.rows(demand)
 @constraint(model, sum(x[:, r.destination, r.product]) == r.demand)
end
```

#### Warning

Iterating through the rows of a query result works by incrementing a cursor inside the database. As a consequence, you cannot call Tables. rows twice on the same query result.

The SQLite queries can be arbitrarily complex. For example, here's a query which builds every possible origindestination pair:

```
od_pairs = DBInterface.execute(
 db,
 """
 SELECT a.location as 'origin',
 b.location as 'destination'
 FROM locations a
 INNER JOIN locations b
 ON a.type = 'origin' AND b.type = 'destination'
 """,
)
```

SQLite.Query{false}(SQLite.Stmt(SQLite.DB("/home/runner/work/JuMP.jl/JuMP.jl/docs/latex\_build/tutorials/linear/multi.s → Base.RefValue{Ptr{SQLite.C.sqlite3\_stmt}}(Ptr{SQLite.C.sqlite3\_stmt} @0x0000000515cb9d8),

```
\hookrightarrow \mbox{ Dict{Int64, Any}()), Base.RefValue{Int32}(100), [:origin, :destination], Type[Union{Missing, the test of test o
```

```
\hookrightarrow String}, Union{Missing, String}], Dict(:origin => 1, :destination => 2),
```

```
\hookrightarrow Base.RefValue{Int64}(0))
```

With a constraint that we cannot send more than 625 units between each pair:

```
for r in Tables.rows(od_pairs)
 @constraint(model, sum(x[r.origin, r.destination, :]) <= 625)
end</pre>
```

## Solution

Finally, we can optimize the model:

optimize!(model)
assert\_is\_solved\_and\_feasible(model)
solution\_summary(model)

```
solution_summary(; result = 1, verbose = false)
- solver_name
 : HiGHS
├ Termination
| | termination_status : OPTIMAL
| | raw_status : kHighsModelStatusOptimal
| ^L objective_bound : 2.25700e+05
Solution (result = 1)
| + primal_status : FEASIBLE_POINT
| + dual_status : FEASIBLE_POINT
| + objective_value : 2.25700e+05
| + dual_objective_value : 2.25700e+05
| L relative gap : 0.00000e+00
L Work counters
 solve_time (sec) : 8.02755e-04
 simplex_iterations : 54
 barrier_iterations : 0
 L node_count
 : -1
```

and print the solution:

```
begin
 println(" ", join(products, ' '))
 for o in origins, d in destinations
 v = lpad.([round(Int, value(x[o, d, p])) for p in products], 5)
 println(o, " ", d, " ", join(replace.(v, " 0" => " . "), " "))
 end
end
```

		bands	coils	plate
GARY	FRA	25	500	100
GARY	DET	125		50
GARY	LAN			
GARY	WIN			50
GARY	STL	250	300	
GARY	FRE			
GARY	LAF			
CLEV	FRA	275		
CLEV	DET	100	200	50
CLEV	LAN	100		
CLEV	WIN			
CLEV	STL		625	
CLEV	FRE	225	400	
CLEV	LAF		375	250
PITT	FRA			
PITT	DET	75	550	
PITT	LAN		400	
PITT	WIN	75	250	
PITT	STL	400	25	200
PITT	FRE		450	100
PITT	LAF	250	125	

## 6.7 The network multi-commodity flow problem

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial is a variation of The multi-commodity flow problem where the graph is a network instead of a bipartite graph.

The purpose of this tutorial is to demonstrate a style of modeling that uses relational algebra.

# **Required packages**

This tutorial uses the following packages:

```
using JuMP
import DataFrames
import HiGHS
import SQLite
import SQLite.DBInterface
import Test
```

# Formulation

The network multi-commondity flow problem is an extension of the The multi-commodity flow problem, where instead of having a bipartite graph of supply and demand nodes, the graph can contains a set of nodes,  $i \in \mathcal{N}$ , which each have a (potentially zero) supply capacity,  $u_{i,p}^s$ , and (potentially zero) a demand,  $d_{i,p}$  for each commodity  $p \in P$ . The nodes are connected by a set of edges  $(i, j) \in \mathcal{E}$ , which have a shipment cost  $c_{i,j,p}^x$  and a total flow capacity of  $u_{i,i}^x$ .

Our take is to choose an optimal supply for each node  $s_{i,p}$ , as well as the optimal transshipment  $x_{i,j,p}$  that minimizes the total cost.

The mathematical formulation is:

$$\begin{array}{ll} \min & \sum_{(i,j)\in\mathcal{E},p\in P} c_{i,j,p}^{x} x_{i,j,p} + \sum_{i\in\mathcal{N},p\in P} c_{i,p}^{s} s_{i,p} \\ s.t. & s_{i,p} + \sum_{(j,i)\in\mathcal{E}} x_{j,i,p} - \sum_{(i,j)\in\mathcal{E}} x_{i,j,p} = d_{i,p} & \forall i\in\mathcal{N}, p\in P \\ & x_{i,j,p} \geq 0 & \forall (i,j)\in\mathcal{E}, p\in P \\ & \sum_{p\in P} x_{i,j,p} \leq u_{i,j}^{x} & \forall (i,j)\in\mathcal{E} \\ & 0 \leq s_{i,p} \leq u_{i,p}^{s} & \forall i\in\mathcal{N}, p\in P. \end{array}$$

The purpose of this tutorial is to demonstrate how this model can be built using relational algebra instead of a direct math-to-code translation of the summations.

## Data

For the purpose of this tutorial, the JuMP repository contains an example database called commodity\_nz.db:

filename = joinpath(@\_\_DIR\_\_, "commodity\_nz.db");

To run locally, download commodity\_nz.db and update filename appropriately.

Load the database using SQLite.DB:

db = SQLite.DB(filename)

SQLite.DB("/home/runner/work/JuMP.jl/JuMP.jl/docs/latex\_build/tutorials/linear/commodity\_nz.db")

A quick way to see the schema of the database is via SQLite.tables:

SQLite.tables(db)

```
4-element Vector{SQLite.DBTable}:
SQLite.DBTable("products", Tables.Schema:
:product Union{Missing, String}
:cost_per_km Union{Missing, Float64})
SQLite.DBTable("shipping", Tables.Schema:
:origin Union{Missing, String}
:destination Union{Missing, String}
:product Union{Missing, String}
:distance_km Union{Missing, Float64})
SQLite.DBTable("supply", Tables.Schema:
```

```
:origin Union{Missing, String}
:product Union{Missing, String}
:capacity Union{Missing, Float64}
:cost Union{Missing, Float64})
SQLite.DBTable("demand", Tables.Schema:
:destination Union{Missing, String}
:product Union{Missing, String}
:demand Union{Missing, Float64})
```

We interact with the database by executing queries and then loading the results into a DataFrame:

```
function get_table(db, table)
 query = DBInterface.execute(db, "SELECT * FROM $table")
 return DataFrames.DataFrame(query)
end
```

get\_table (generic function with 1 method)

The shipping table contains the set of arcs and their distances:

df\_shipping = get\_table(db, "shipping")

	origin	destination	product	distance_km
	String	String	String	Float64
1	auckland	waikato	milk	112.0
2	auckland	tauranga	milk	225.0
3	auckland	christchurch	milk	1070.0
4	waikato	auckland	milk	112.0
5	waikato	tauranga	milk	107.0
6	waikato	wellington	milk	392.0
7	tauranga	auckland	milk	225.0
8	tauranga	waikato	milk	107.0
9	christchurch	auckland	milk	1070.0
10	auckland	waikato	kiwifruit	112.0
11	auckland	christchurch	kiwifruit	1070.0
12	waikato	auckland	kiwifruit	112.0
13	waikato	wellington	kiwifruit	392.0
14	tauranga	auckland	kiwifruit	225.0
15	tauranga	waikato	kiwifruit	107.0
16	christchurch	auckland	kiwifruit	1070.0

The products table contains the shipping cost per kilometer of each product:

df\_products = get\_table(db, "products")

The supply table contains the supply capacity of each node, as well as the cost:
	product	cost_per_km
	String	Float64
1	milk	0.001
2	kiwifruit	0.01

```
df_supply = get_table(db, "supply")
```

	origin	product	capacity	cost
	String	String	Float64	Float64
1	waikato	milk	10.0	0.5
2	tauranga	milk	6.0	1.0
3	tauranga	kiwifruit	26.0	1.0
4	christchurch	milk	10.0	0.6

The demand table contains the demand of each node:

```
df_demand = get_table(db, "demand")
```

	destination	product	demand
	String	String	Float64
1	auckland	milk	16.0
2	auckland	kiwifruit	16.0
3	tauranga	milk	2.0
4	tauranga	kiwifruit	2.0
5	wellington	milk	2.0
6	wellington	kiwifruit	2.0
7	christchurch	milk	4.0
8	christchurch	kiwifruit	4.0

## **JuMP formulation**

We start by creating a model and our decision variables:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
```

For the shipping decisions, we create a new column in df\_shipping called x\_flow, which has one non-negative decision variable for each row:

```
df_shipping.x_flow = @variable(model, x[1:size(df_shipping, 1)] >= 0)
df_shipping
```

For the supply, we add a variable to each row, and then set the upper bound to the capacity of each supply node:

	origin	destination	product	distance_km	x_flow
	String	String	String	Float64	GenericV
1	auckland	waikato	milk	112.0	$x_1$
2	auckland	tauranga	milk	225.0	$x_2$
3	auckland	christchurch	milk	1070.0	$x_3$
4	waikato	auckland	milk	112.0	$x_4$
5	waikato	tauranga	milk	107.0	$x_5$
6	waikato	wellington	milk	392.0	$x_6$
7	tauranga	auckland	milk	225.0	$x_7$
8	tauranga	waikato	milk	107.0	$x_8$
9	christchurch	auckland	milk	1070.0	$x_9$
10	auckland	waikato	kiwifruit	112.0	$x_{10}$
11	auckland	christchurch	kiwifruit	1070.0	$x_{11}$
12	waikato	auckland	kiwifruit	112.0	$x_{12}$
13	waikato	wellington	kiwifruit	392.0	$x_{13}$
14	tauranga	auckland	kiwifruit	225.0	$x_{14}$
15	tauranga	waikato	kiwifruit	107.0	$x_{15}$
16	christchurch	auckland	kiwifruit	1070.0	$x_{16}$

df\_supply.x\_supply = @variable(model, s[1:size(df\_supply, 1)] >= 0)
set\_upper\_bound.(df\_supply.x\_supply, df\_supply.capacity)
df\_supply

	origin	product	capacity	cost	x_supply
	String	String	Float64	Float64	GenericV
1	waikato	milk	10.0	0.5	$s_1$
2	tauranga	milk	6.0	1.0	$s_2$
3	tauranga	kiwifruit	26.0	1.0	$s_3$
4	christchurch	milk	10.0	0.6	$s_4$

Our objective is to minimize the shipping cost plus the supply cost. To compute the flow cost, we need to join the shipping table, which contains distance\_km with the products table, which contains cost\_per\_km:

```
df_cost = DataFrames.leftjoin(df_shipping, df_products; on = [:product])
df_cost.flow_cost = df_cost.cost_per_km .* df_cost.distance_km
df_cost
```

Then we can use linear algebra to compute the inner product between two columns:

```
@objective(
 model,
 Min,
 df_cost.flow_cost' * df_shipping.x_flow +
 df_supply.cost' * df_supply.x_supply
);
```

For the flow capacities on each arc, we use DataFrames.groupby to partition the flow variables based on :origin and :destination, and then we constrain their sum to be less than a fixed capacity.

	origin	destination	product	distance_km	x_flow	cost_per_km	flow_cost
	String	String	String	Float64	GenericV	Float64?	Float64
1	auckland	waikato	milk	112.0	$x_1$	0.001	0.112
2	auckland	tauranga	milk	225.0	$x_2$	0.001	0.225
3	auckland	christchurch	milk	1070.0	$x_3$	0.001	1.07
4	waikato	auckland	milk	112.0	$x_4$	0.001	0.112
5	waikato	tauranga	milk	107.0	$x_5$	0.001	0.107
6	waikato	wellington	milk	392.0	$x_6$	0.001	0.392
7	tauranga	auckland	milk	225.0	$x_7$	0.001	0.225
8	tauranga	waikato	milk	107.0	$x_8$	0.001	0.107
9	christchurch	auckland	milk	1070.0	$x_9$	0.001	1.07
10	auckland	waikato	kiwifruit	112.0	$x_{10}$	0.01	1.12
11	auckland	christchurch	kiwifruit	1070.0	$x_{11}$	0.01	10.7
12	waikato	auckland	kiwifruit	112.0	$x_{12}$	0.01	1.12
13	waikato	wellington	kiwifruit	392.0	$x_{13}$	0.01	3.92
14	tauranga	auckland	kiwifruit	225.0	$x_{14}$	0.01	2.25
15	tauranga	waikato	kiwifruit	107.0	$x_{15}$	0.01	1.07
16	christchurch	auckland	kiwifruit	1070.0	$x_{16}$	0.01	10.7

```
capacity = 30
for df in DataFrames.groupby(df_shipping, [:origin, :destination])
 @constraint(model, sum(df.x_flow) <= capacity)
end</pre>
```

For each node in the graph, we need to compute a mass balance constraint which says that for each product, the supply, plus the flow into the node, and less the flow out of the node is equal to the demand.

We can compute an expression for the flow out of each node using DataFrames.groupby on the origin and product columns of the df\_shipping table:

```
df_flow_out = DataFrames.DataFrame(
 (node = i.origin, product = i.product, x_flow_out = sum(df.x_flow)) for
 (i, df) in pairs(DataFrames.groupby(df_shipping, [:origin, :product]))
)
```

	node	product	x_flow_out
	String	String	AffExpr
1	auckland	milk	$x_1 + x_2 + x_3$
2	waikato	milk	$x_4 + x_5 + x_6$
3	tauranga	milk	$x_7 + x_8$
4	christchurch	milk	$x_9$
5	auckland	kiwifruit	$x_{10} + x_{11}$
6	waikato	kiwifruit	$x_{12} + x_{13}$
7	tauranga	kiwifruit	$x_{14} + x_{15}$
8	christchurch	kiwifruit	$x_{16}$

We can compute an expression for the flow into each node using DataFrames.groupby on the destination and product columns of the df\_shipping table:

```
df_flow_in = DataFrames.DataFrame(
 (node = i.destination, product = i.product, x_flow_in = sum(df.x_flow))
 for (i, df) in
 pairs(DataFrames.groupby(df_shipping, [:destination, :product]))
)
```

node	product	x_flow_in
String	String	AffExpr
waikato	milk	$x_1 + x_8$
tauranga	milk	$x_2 + x_5$
christchurch	milk	$x_3$
auckland	milk	$x_4 + x_7 + x_9$
wellington	milk	$x_6$
waikato	kiwifruit	$x_{10} + x_{15}$
christchurch	kiwifruit	$x_{11}$
auckland	kiwifruit	$x_{12} + x_{14} + x_{16}$
wellington	kiwifruit	$x_{13}$
	node String waikato tauranga christchurch auckland wellington waikato christchurch auckland wellington	nodeproductStringStringwaikatomilktaurangamilkchristchurchmilkaucklandmilkwellingtonmilkwaikatokiwifruitchristchurchkiwifruitaucklandkiwifruitwellingtonkiwifruitaucklandkiwifruitwellingtonkiwifruit

We can join the two tables together using DataFrames.outerjoin. We need to use outerjoin here because there might be missing rows.

df = DataFrames.outerjoin(df\_flow\_in, df\_flow\_out; on = [:node, :product])

	node	product	x_flow_in	x_flow_out
	String	String	AffExpr?	AffExpr?
1	waikato	milk	$x_1 + x_8$	$x_4 + x_5 + x_6$
2	tauranga	milk	$x_2 + x_5$	$x_7 + x_8$
3	christchurch	milk	$x_3$	$x_9$
4	auckland	milk	$x_4 + x_7 + x_9$	$x_1 + x_2 + x_3$
5	waikato	kiwifruit	$x_{10} + x_{15}$	$x_{12} + x_{13}$
6	christchurch	kiwifruit	$x_{11}$	$x_{16}$
7	auckland	kiwifruit	$x_{12} + x_{14} + x_{16}$	$x_{10} + x_{11}$
8	wellington	milk	$x_6$	missing
9	wellington	kiwifruit	$x_{13}$	missing
10	tauranga	kiwifruit	missing	$x_{14} + x_{15}$

Next, we need to join the supply column:

```
df = DataFrames.leftjoin(
 df,
 DataFrames.select(df_supply, [:origin, :product, :x_supply]);
 on = [:node => :origin, :product],
)
```

and then the demand column

	node	product	x_flow_in	x_flow_out	x_supply
	String	String	AffExpr?	AffExpr?	GenericV?
1	waikato	milk	$x_1 + x_8$	$x_4 + x_5 + x_6$	$s_1$
2	tauranga	milk	$x_2 + x_5$	$x_7 + x_8$	$s_2$
3	christchurch	milk	$x_3$	$x_9$	$s_4$
4	tauranga	kiwifruit	missing	$x_{14} + x_{15}$	$s_3$
5	auckland	milk	$x_4 + x_7 + x_9$	$x_1 + x_2 + x_3$	missing
6	waikato	kiwifruit	$x_{10} + x_{15}$	$x_{12} + x_{13}$	missing
7	christchurch	kiwifruit	$x_{11}$	$x_{16}$	missing
8	auckland	kiwifruit	$x_{12} + x_{14} + x_{16}$	$x_{10} + x_{11}$	missing
9	wellington	milk	$x_6$	missing	missing
10	wellington	kiwifruit	$x_{13}$	missing	missing

```
DataFrames.select(df_demand, [:destination, :product, :demand]);
on = [:node => :destination, :product],
```

```
)
```

	node	product	x_flow_in	x_flow_out	x_supply	demand
	String	String	AffExpr?	AffExpr?	GenericV?	Float64?
1	tauranga	milk	$x_2 + x_5$	$x_7 + x_8$	$s_2$	2.0
2	christchurch	milk	$x_3$	$x_9$	$s_4$	4.0
3	tauranga	kiwifruit	missing	$x_{14} + x_{15}$	$s_3$	2.0
4	auckland	milk	$x_4 + x_7 + x_9$	$x_1 + x_2 + x_3$	missing	16.0
5	christchurch	kiwifruit	$x_{11}$	$x_{16}$	missing	4.0
6	auckland	kiwifruit	$x_{12} + x_{14} + x_{16}$	$x_{10} + x_{11}$	missing	16.0
7	wellington	milk	$x_6$	missing	missing	2.0
8	wellington	kiwifruit	$x_{13}$	missing	missing	2.0
9	waikato	milk	$x_1 + x_8$	$x_4 + x_5 + x_6$	$s_1$	missing
10	waikato	kiwifruit	$x_{10} + x_{15}$	$x_{12} + x_{13}$	missing	missing

Now we're ready to add our mass balance constraint. Because some rows contain missing values, we need to use coalesce to convert any missing into a numeric value:

```
@constraint(
```

```
model,
[r in eachrow(df)],
coalesce(r.x_supply, 0.0) + coalesce(r.x_flow_in, 0.0) -
coalesce(r.x_flow_out, 0.0) == coalesce(r.demand, 0.0),
.
```

```
);
```

# Solution

Finally, we can optimize the model:

```
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : HiGHS
| Termination
| | termination_status : OPTIMAL
| | result_count : 1
| | raw_status : kHighsModelStatusOptimal
| L objective_bound : 1.43228e+02
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | dual_objective_value : 1.43228e+02
| L relative_gap : 3.96874e-16
L Work counters
| solve_time (sec) : 3.64065e-04
| simplex_iterations : 8
| barrier_iterations : 0
L node_count : -1
```

update the solution in the DataFrames:

```
df_shipping.x_flow = value.(df_shipping.x_flow)
df_supply.x_supply = value.(df_supply.x_supply);
```

and display the optimal solution for flows:

```
DataFrames.select(
 filter!(row -> row.x_flow > 0.0, df_shipping),
 [:origin, :destination, :product, :x_flow],
)
```

	origin	destination	product	x_flow
	String	String	String	Float64
1	waikato	auckland	milk	10.0
2	waikato	wellington	milk	2.0
3	tauranga	auckland	milk	2.0
4	tauranga	waikato	milk	2.0
5	christchurch	auckland	milk	4.0
6	auckland	christchurch	kiwifruit	4.0
7	waikato	auckland	kiwifruit	20.0
8	waikato	wellington	kiwifruit	2.0
9	tauranga	waikato	kiwifruit	22.0

## 6.8 Tips and tricks

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial was originally contributed by Arpit Bhatia.

# Тір

A good source of tips is the Mosek Modeling Cookbook.

This tutorial collates some tips and tricks you can use when formulating mixed-integer programs. It uses the following packages:

julia> using JuMP

## Absolute value

To model the absolute value function  $t \ge |x|$ , there are a few options. In all cases, these reformulations only work if you are minimizing t "down" into |x|. They do not work if you are trying to maximize |x|.

## Option 1

This option adds two linear inequality constraints:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, t)
t
julia> @constraint(model, t >= x)
-x + t ≥ 0
julia> @constraint(model, t >= -x)
x + t ≥ 0
```

#### **Option 2**

This option uses two non-negative variables and forms expressions for x and t:

```
julia> model = Model();
julia> @variable(model, z[1:2] >= 0)
2-element Vector{VariableRef}:
 z[1]
 z[2]
julia> @expression(model, t, z[1] + z[2])
 z[1] + z[2]
julia> @expression(model, x, z[1] - z[2])
 z[1] - z[2]
```

## **Option 3**

This option uses MOI.NormOneCone and lets JuMP choose the reformulation:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, t)
t
julia> @constraint(model, [t; x] in MOI.NormOneCone(2))
[t, x] & MathOptInterface.NormOneCone(2)
```

## L1-norm

To model  $\min ||x||_1$ , that is,  $\min \sum_i |x_i|$ , use the <code>MOI.NormOneCone</code>:

```
julia> model = Model();
julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
julia> @variable(model, t)
t
julia> @constraint(model, [t; x] in MOI.NormOneCone(1 + length(x)))
[t, x[1], x[2], x[3]] ∈ MathOptInterface.NormOneCone(4)
julia> @objective(model, Min, t)
```

## Infinity-norm

t

To model  $\min ||x||_\infty$ , that is,  $\min \max |x_i|$ , use the <code>MOI.NormInfinityCone</code>:

```
julia> model = Model();
julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
julia> @variable(model, t)
t
julia> @constraint(model, [t; x] in MOI.NormInfinityCone(1 + length(x)))
```

[t, x[1], x[2], x[3]] ∈ MathOptInterface.NormInfinityCone(4)

```
julia> @objective(model, Min, t)
t
```

## Max

```
To model t \ge \max\{x, y\}, do:
```

```
julia> model = Model();
julia> @variable(model, t)
t
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> @constraint(model, t >= x)
t - x ≥ 0
julia> @constraint(model, t >= y)
t - y ≥ 0
```

This reformulation does not work for  $t \ge \min\{x, y\}$ .

# Min

```
To model t \leq \min\{x, y\}, do:
```

```
julia> model = Model();
julia> @variable(model, t)
t
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> @constraint(model, t <= x)
t - x ≤ 0
julia> @constraint(model, t <= y)
t - y ≤ 0
```

This reformulation does not work for  $t \leq \max\{x, y\}$ .

## Modulo

To model  $y = x \mod n$ , where n is a constant modulus, we use the relationship  $x = n \cdot z + y$ , where  $z \in \mathbb{Z}_+$  is the number of times that n can be divided by x and y is the remainder.

```
julia> n = 4
4
julia> model = Model();
julia> @variable(model, x >= 0, Int)
x
julia> @variable(model, 0 <= y <= n - 1, Int)
y
julia> @variable(model, z >= 0, Int)
z
julia> @constraint(model, x == n * z + y)
x - y - 4 z = 0
```

The modulo reformulation is often useful for subdividing a time increment into units of time like hours and days:

```
julia> model = Model();
julia> @variable(model, t >= 0, Int)
t
julia> @variable(model, 0 <= hours <= 23, Int)
hours
julia> @variable(model, days >= 0, Int)
days
julia> @constraint(model, t == 24 * days + hours)
```

#### **Boolean operators**

t - hours - 24 days = 0

Binary variables can be used to construct logical operators. Here are some example.

Or

```
x_3 = x_1 \lor x_2
```

x[2] x[3]

And

## $x_3 = x_1 \wedge x_2$

Not

```
x_1 \neg x_2
```

julia> model = Model();

```
julia> @variable(model, x[1:2], Bin)
2-element Vector{VariableRef}:
 x[1]
 x[2]
julia> @constraint(model, x[1] == 1 - x[2])
```

x[1] + x[2] = 1

## Implies

 $x_1 \implies x_2$ 

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2], Bin)
2-element Vector{VariableRef}:
 x[1]
 x[2]
julia> @constraint(model, x[1] <= x[2])
x[1] - x[2] ≤ 0</pre>
```

#### Disjunctions

#### Problem

Suppose that we have two constraints  $a^{\top}x \leq b$  and  $c^{\top}x \leq d$ , and we want at least one to hold.

## Trick 1

Use an indicator constraint.

```
Example Either x_1 \leq 1 or x_2 \leq 2.
```

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @variable(model, y[1:2], Bin)
2-element Vector{VariableRef}:
 y[1]
 y[2]

julia> @constraint(model, y[1] --> {x[1] <= 1})
y[1] --> {x[1] ≤ 1}

julia> @constraint(model, y[2] --> {x[2] <= 2})
y[2] --> {x[2] ≤ 2}

julia> @constraint(model, sum(y) == 1) # Exactly one branch must be true
y[1] + y[2] = 1
```

## Trick 2

Introduce a "big-M" multiplied by a binary variable to relax one of the constraints.

**Example** Either  $x_1 \leq 1$  or  $x_2 \leq 2$ .

```
julia> model = Model();
```

julia> @variable(model, x[1:2] <= 10)
2-element Vector{VariableRef}:</pre>

x[1] x[2]

```
julia> @variable(model, y[1:2], Bin)
2-element Vector{VariableRef}:
 y[1]
 y[2]
julia> M = 100
100
julia> @constraint(model, x[1] <= 1 + M * y[1])
x[1] - 100 y[1] ≤ 1
julia> @constraint(model, x[2] <= 2 + M * y[2])
x[2] - 100 y[2] ≤ 2
julia> @constraint(model, sum(y) == 1)
y[1] + y[2] = 1
```

#### Warning

If M is too small, the solution may be suboptimal. If M is too big, the solver may encounter numerical issues. Try to use domain knowledge to choose an M that is just right. Gurobi has a good documentation section on this topic.

## Indicator constraints

## Problem

Suppose we want to model that a certain linear inequality must be satisfied when some other event occurs, that is, for a binary variable z, we want to model the implication:

$$z = 1 \implies a^{\top} x < b$$

## Trick 1

Some solvers have native support for indicator constraints. In addition, if the variables involved have finite domains, then JuMP can automatically reformulate an indicator into a mixed-integer program.

Example  $x_1 + x_2 \leq 1$  if z = 1.

```
julia> model = Model();
julia> @variable(model, 0 <= x[1:2] <= 10)
2-element Vector{VariableRef}:
 x[1]
 x[2]
julia> @variable(model, z, Bin)
z
```

```
julia> @constraint(model, z --> {sum(x) <= 1})
z --> {x[1] + x[2] ≤ 1}
```

```
Example x_1 + x_2 \leq 1 if z = 0.
```

## Trick 2

If the solver doesn't support indicator constraints and the variables do not have a finite domain, you can use the big-M trick.

Example  $x_1 + x_2 \leq 1$  if z = 1.

```
julia> model = Model();
julia> @variable(model, x[1:2] <= 10)
2-element Vector{VariableRef}:
 x[1]
 x[2]
julia> @variable(model, z, Bin)
z
julia> M = 100
100
julia> @constraint(model, sum(x) <= 1 + M * (1 - z))
 x[1] + x[2] + 100 z ≤ 101
```

```
Example x_1 + x_2 \leq 1 if z = 0.
```

```
julia> model = Model();
julia> @variable(model, x[1:2] <= 10)
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

julia> @variable(model, z, Bin)

z
julia> M = 100
100
julia> @constraint(model, sum(x) <= 1 + M \* z)
x[1] + x[2] - 100 z ≤ 1</pre>

## Semi-continuous variables

A semi-continuous variable is a continuous variable between bounds [l, u] that also can assume the value zero, that is:  $x \in \{0\} \cup [l, u]$ .

```
Example x \in \{0\} \cup [1,2]

julia> model = Model();

julia> @variable(model, x in Semicontinuous(1.0, 2.0))

x
```

You can also represent a semi-continuous variable using the reformulation:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, z, Bin)
z
julia> @constraint(model, x <= 2 * z)
x - 2 z ≤ 0
julia> @constraint(model, x >= 1 * z)
x - z ≥ 0
```

When z = 0 the two constraints are equivalent to  $0 \le x \le 0$ . When z = 1, the two constraints are equivalent to  $1 \le x \le 2$ .

## Semi-integer variables

julia> model = Model();

A semi-integer variable is a variable which assumes integer values between bounds [l, u] and can also assume the value zero:  $x \in \{0\} \cup [l, u] \cap \mathbb{Z}$ .

```
julia> @variable(model, x in Semiinteger(5.0, 10.0))
x
```

You can also represent a semi-integer variable using the reformulation:

```
julia> model = Model();
julia> @variable(model, x, Int)
x
julia> @variable(model, z, Bin)
z
julia> @constraint(model, x <= 10 * z)
x - 10 z ≤ 0
julia> @constraint(model, x >= 5 * z)
x - 5 z ≥ 0
```

When z = 0 the two constraints are equivalent to  $0 \le x \le 0$ . When z = 1, the two constraints are equivalent to  $5 \le x \le 10$ .

#### Special Ordered Sets of Type 1

A Special Ordered Set of Type 1 is a set of variables, at most one of which can take a non-zero value, all others being at 0.

They most frequently apply where a set of variables are actually binary variables. In other words, we have to choose at most one from a set of possibilities.

You can optionally pass S0S1 a weight vector like

```
julia> @constraint(model, x in SOS1([0.2, 0.5, 0.3]))
[x[1], x[2], x[3]] ∈ MathOptInterface.SOS1{Float64}([0.2, 0.5, 0.3])
```

If the decision variables are related and have a physical ordering, then the weight vector, although not used directly in the constraint, can help the solver make a better decision in the solution process.

#### **Special Ordered Sets of Type 2**

A Special Ordered Set of type 2 is a set of non-negative variables, of which at most two can be non-zero, and if two are non-zero these must be consecutive in their ordering.

```
julia> model = Model();
```

```
julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
julia> @constraint(model, x in SOS2([3.0, 1.0, 2.0]))
[x[1], x[2], x[3]] ∈ MathOptInterface.SOS2{Float64}([3.0, 1.0, 2.0])
```

The ordering provided by the weight vector is more important in this case as the variables need to be consecutive according to the ordering. For example, in the above constraint, the possible pairs are:

- Consecutive
  - (x[1] and x[3]) as they correspond to 3 and 2 resp. and thus can be non-zero
  - (x[2] and x[3]) as they correspond to 1 and 2 resp. and thus can be non-zero
- Non-consecutive
  - (x[1] and x[2]) as they correspond to 3 and 1 resp. and thus cannot be non-zero

#### **Piecewise linear approximations**

SOSII constraints are most often used to form piecewise linear approximations of a function.

Given a set of points for x:

**julia> x̂** = -1:0.5:2 -1.0:0.5:2.0

and a set of corresponding points for y:

```
julia> ŷ = x̂ .^ 2
7-element Vector{Float64}:
 1.0
 0.25
 0.0
 0.25
 1.0
 2.25
 4.0
```

the piecewise linear approximation is constructed by representing x and y as convex combinations of  $\hat{x}$  and  $\hat{y}$ .

```
julia> N = length(x)
7
julia> model = Model();
julia> @variable(model, -1 <= x <= 2)
x</pre>
```

```
julia> @variable(model, y)
у
julia> @variable(model, 0 \le \lambda[1:N] \le 1)
7-element Vector{VariableRef}:
 \lambda[1]
 λ[2]
 λ[3]
 λ[4]
 λ[5]
 \lambda[6]
 λ[7]
julia> @objective(model, Max, y)
julia> @constraints(model, begin
 x == sum(\hat{x}[i] * \lambda[i] \text{ for } i \text{ in } 1:N)
 y == sum(\hat{y}[i] * \lambda[i] \text{ for } i \text{ in } 1:N)
 sum(\lambda) == 1
 \lambda in SOS2()
 end)
(x + \lambda[1] + 0.5 \lambda[2] - 0.5 \lambda[4] - \lambda[5] - 1.5 \lambda[6] - 2 \lambda[7] = 0, y - \lambda[1] - 0.25 \lambda[2] - 0.25 \lambda[4] - 0
\hookrightarrow \lambda[5] - 2.25 \lambda[6] - 4 \lambda[7] = 0, \lambda[1] + \lambda[2] + \lambda[3] + \lambda[4] + \lambda[5] + \lambda[6] + \lambda[7] = 1, [\lambda[1], \lambda[2], \lambda[2], \lambda[2], \lambda[2] + \lambda[3] + \lambda[4] + \lambda[5] + \lambda[6] + \lambda[7] = 1, \lambda[1], \lambda[2], \lambda[2], \lambda[3] + \lambda[4] + \lambda[5] + \lambda[6] + \lambda[6] + \lambda[7] = 1, \lambda[6] + \lambda[7] + \lambda[6] +
\hookrightarrow \ \lambda[3], \ \lambda[4], \ \lambda[5], \ \lambda[6], \ \lambda[7]] \in MathOptInterface.SOS2{Float64}([1.0, \ 2.0, \ 3.0, \ 4.0, \ 5.0, \ 6
\hookrightarrow 7.0]))
```

## 6.9 Approximating nonlinear functions

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to explain how to approximate nonlinear functions with a mixed-integer linear program.

This tutorial uses the following packages:

using JuMP import HiGHS import Plots

#### Minimizing a convex function (outer approximation)

If the function you are approximating is convex, and you want to minimize "down" onto it, then you can use an outer approximation.

For example,  $f(x) = x^2$  is a convex function:

```
f(x) = x^2
∇f(x) = 2 * x
plot = Plots.plot(f, -2:0.01:2; ylims = (-0.5, 4), label = false, width = 3)
```



Because f is convex, we know that for any  $x_k$ , we have:  $f(x) \ge f(x_k) + \nabla f(x_k) \cdot (x - x_k)$ 

```
for x_k in -2:1:2 ## Tip: try changing the number of points x_k

g = x \rightarrow f(x_k) + \nabla f(x_k) * (x - x_k)

Plots.plot!(plot, g, -2:0.01:2; color = :red, label = false, width = 3)

end

plot
```



We can use these *tangent planes* as constraints in our model to create an outer approximation of the function. As we add more planes, the error between the true function and the piecewise linear outer approximation decreases.

```
Here is the model in JuMP:
```

outer\_approximate\_x\_squared (generic function with 1 method)

Here are a few values:



# Note This formulation does not work if we want to maximize y.

## Maximizing a concave function (outer approximation)

The outer approximation also works if we want to maximize "up" into a concave function.

```
f(x) = log(x)

∇f(x) = 1 / x
X = 0.1:0.1:1.6
plot = Plots.plot(
 f,
 X;
 xlims = (0.1, 1.6),
 ylims = (-3, log(1.6)),
 label = false,
 width = 3,
)
```

```
for x_k in 0.1:0.5:1.6 ## Tip: try changing the number of points x_k
 g = x -> f(x_k) + \nabla f(x_k) * (x - x_k)
 Plots.plot!(plot, g, X; color = :red, label = false, width = 3)
end
plot
```



Here is the model in JuMP:

```
function outer_approximate_log(x)
f(x) = log(x)
Vf(x) = 1 / x
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 0.1 <= x <= 1.6)
@variable(model, y)
Tip: try changing the number of points x_k
@constraint(model, [x_k in 0.1:0.5:2], y <= f(x_k) + \nabla f(x_k) * (x - x_k))
@objective(model, Max, y)
@constraint(model, x == x) # <-- a trivial constraint just for testing.
optimize!(model)
assert_is_solved_and_feasible(model)
return value(y)
end</pre>
```

```
outer_approximate_log (generic function with 1 method)
```

Here are a few values:



# Note

This formulation does not work if we want to minimize y.

## Minimizing a convex function (inner approximation)

Instead of creating an outer approximation, we can also create an inner approximation. For example, given  $f(x) = x^2$ , we may want to approximate the true function by the red piecewise linear function:

```
f(x) = x²

$ = -2:0.8:2 ## Tip: try changing the number of points in x[^]

plot = Plots.plot(f, -2:0.01:2; ylims = (-0.5, 4), label = false, linewidth = 3)
```



Plots.plot!(plot, f,  $\hat{x}$ ; label = false, color = :red, linewidth = 3) plot

To do so, we represent the decision variables (x,y) by the convex combination of a set of discrete points  $\{x_k,y_k\}_{k=1}^K$ :

$$x = \sum_{k=1}^{K} \lambda_k x_k$$
$$y = \sum_{k=1}^{K} \lambda_k y_k$$
$$\sum_{k=1}^{K} \lambda_k = 1$$
$$\ge 0, k = 1, \dots, k$$

The feasible region of the convex combination actually allows any (x,y) point inside this shaded region:

I = [1, 2, 3, 4, 5, 6, 1]
Plots.plot!(x[I], f.(x[I]); fill = (0, 0, "#f004"), width = 0, label = false)
plot

 $\lambda_k$ 



Thus, this formulation does not work if we want to maximize y.

Here is the model in JuMP:

```
function inner_approximate_x_squared(x)
 f(x) = x^2
 \nabla f(x) = 2x
 \boldsymbol{\hat{x}} = -2:0.8:2 ## Tip: try changing the number of points in x^
 \hat{y} = f.(\hat{x})
 n = length(\hat{x})
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, -2 <= x <= 2)</pre>
 @variable(model, y)
 @variable(model, 0 \le \lambda[1:n] \le 1)
 @constraint(model, x == sum(\lambda[i] * \hat{x}[i] \text{ for } i \text{ in } 1:n))
 @constraint(model, y == sum(\lambda[i] * \hat{y}[i] for i in 1:n))
 (constraint(model, sum(\lambda) == 1))
 @objective(model, Min, y)
 @constraint(model, x == \bar{x}) # <-- a trivial constraint just for testing.
 optimize!(model)
 assert_is_solved_and_feasible(model)
 return value(y)
end
```

inner\_approximate\_x\_squared (generic function with 1 method)

Here are a few values:



## Maximizing a convex function (inner approximation)

The inner approximation also works if we want to maximize "up" into a concave function.



Here is the model in JuMP:

```
function inner_approximate_log(x)
 f(x) = log(x)
 \hat{y} = f.(\hat{x})
 n = length(\hat{x})
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, 0.1 <= x <= 1.6)</pre>
 @variable(model, y)
 @variable(model, 0 <= \lambda[1:n] <= 1)
 (constraint(model, sum(\lambda) == 1))
 \texttt{@constraint(model, x == sum(\lambda[i] * \hat{x}[i] for i in 1:n))}
 \texttt{@constraint(model, y == sum(\lambda[i] * \hat{y}[i] for i in 1:n))}
 @objective(model, Max, y)
 @constraint(model, x == \bar{x}) # <-- a trivial constraint just for testing.
 optimize!(model)
 assert_is_solved_and_feasible(model)
 return value(y)
end
```

inner\_approximate\_log (generic function with 1 method)

Here are a few values:



## **Piecewise linear approximation**

If the model is non-convex (or non-concave), then we cannot use an outer approximation, and the inner approximation allows a solution far from the true function. For example, for f(x) = sin(x), we have:



We can force the inner approximation to stay on the red line by adding the constraint  $\lambda$  in S0S2(). The S0S2 set is a Special Ordered Set of Type 2, and it ensures that at most two elements of  $\lambda$  can be non-zero, and if they are, that they must be adjacent. This prevents the model from taking a convex combination of points 1 and 5 to end up on the lower boundary of the shaded red area.

Here is the model in JuMP:

```
function piecewise_linear_sin(x)
 f(x) = sin(x)
 # Tip: try changing the number of points in x^{\uparrow}
 \hat{x} = range(; start = 0, stop = 2\pi, length = 7)
 \hat{y} = f.(\hat{x})
 n = length(\hat{x})
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, 0 \le x \le 2\pi)
 @variable(model, y)
 @variable(model, 0 \le \lambda[1:n] \le 1)
 @constraints(model, begin
 x = sum(\lambda[i] * \hat{x}[i] \text{ for } i \text{ in } 1:n)
 y = sum(\lambda[i] * \hat{y}[i] \text{ for } i \text{ in } 1:n)
 sum(\lambda) == 1
 \lambda in SOS2() # <-- this is new
 end)
 @constraint(model, x == \bar{x}) # <-- a trivial constraint just for testing.
 optimize!(model)
 assert_is_solved_and_feasible(model)
 return value(y)
```

end

piecewise\_linear\_sin (generic function with 1 method)

```
Here are a few values:
```



# 6.10 The facility location problem

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial was originally contributed by Mathieu Tanneau and Alexis Montoison.

## **Required packages**

This tutorial requires the following packages:

CHAPTER 6. LINEAR PROGRAMS

using JuMP import HiGHS import LinearAlgebra import Plots import Random

#### **Uncapacitated facility location**

## **Problem description**

We are given

- A set  $M = \{1, \ldots, m\}$  of clients
- A set  $N = \{1, \dots, n\}$  of sites where a facility can be built

Decision variables Decision variables are split into two categories:

- Binary variable  $y_j$  indicates whether facility j is built or not
- Binary variable  $x_{i,j}$  indicates whether client i is assigned to facility j

**Objective** The objective is to minimize the total cost of serving all clients. This costs breaks down into two components:

• Fixed cost of building a facility.

In this example, this cost is  $f_j = 1, \ \forall j$ .

· Cost of serving clients from the assigned facility.

In this example, the cost  $c_{i,j}$  of serving client *i* from facility *j* is the Euclidean distance between the two.

#### Constraints

- · Each customer must be served by exactly one facility
- A facility cannot serve any client unless it is open

## **MILP** formulation

The problem can be formulated as the following MILP:

$$\min_{x,y} \sum_{i,j} c_{i,j} x_{i,j} + \sum_{j} f_{j} y_{j}$$

$$s.t. \sum_{j} x_{i,j} = 1, \qquad \forall i \in M$$

$$x_{i,j} \leq y_{j}, \qquad \forall i \in M, j \in N$$

$$x_{i,j}, y_{j} \in \{0,1\}, \qquad \forall i \in M, j \in N$$

where the first set of constraints ensures that each client is served exactly once, and the second set of constraints ensures that no client is served from an unopened facility.

## **Problem data**

To ensure reproducibility, we set the random number seed:

Random.seed!(314)

Random.TaskLocalRNG()

Here's the data we need:

```
Number of clients
m = 12
Number of facility locations
n = 5
Clients' locations
x_c, y_c = rand(m), rand(m)
Facilities' potential locations
x_f, y_f = rand(n), rand(n)
Fixed costs
f = ones(n);
Distance
c = zeros(m, n)
for i in 1:m
 for j in 1:n
 c[i, j] = LinearAlgebra.norm([x_c[i] - x_f[j], y_c[i] - y_f[j]], 2)
 end
end
```

```
Display the data
```

```
Plots.scatter(
 x_c,
 y_c;
 label = "Clients",
 markershape = :circle,
 markercolor = :blue,
)
Plots.scatter!(
 x_f,
 y_f;
 label = "Facility",
 markershape = :square,
 markercolor = :white,
 markersize = 6,
 markerstrokecolor = :red,
 markerstrokewidth = 2,
)
```



## **JuMP** implementation

Create a JuMP model

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, y[1:n], Bin);
@variable(model, x[1:m, 1:n], Bin);
Each client is served exactly once
@constraint(model, client_service[i in 1:m], sum(x[i, j] for j in 1:n) == 1);
A facility must be open to serve a client
@constraint(model, open_facility[i in 1:m, j in 1:n], x[i, j] <= y[j]);
@objective(model, Min, f' * y + sum(c .* x));</pre>
```

Solve the uncapacitated facility location problem with HiGHS

```
optimize!(model)
assert_is_solved_and_feasible(model)
println("Optimal value: ", objective_value(model))
```

Optimal value: 5.7018394545724185

## Visualizing the solution

The threshold 1e-5 ensure that edges between clients and facilities are drawn when  $x[i, j] \approx 1$ .

```
x_is_selected = isapprox.(value.(x), 1; atol = 1e-5);
y_is_selected = isapprox.(value.(y), 1; atol = 1e-5);
p = Plots.scatter(
 x_c,
 y_c;
 markershape = :circle,
 markercolor = :blue,
 label = nothing,
)
Plots.scatter!(
 x_f,
 y_f;
 markershape = :square,
 markercolor = [(y_is_selected[j] ? :red : :white) for j in 1:n],
 markersize = 6,
 markerstrokecolor = :red,
 markerstrokewidth = 2,
 label = nothing,
)
for i in 1:m, j in 1:n
 if x_is_selected[i, j]
 Plots.plot!(
 [x_c[i], x_f[j]],
 [y_c[i], y_f[j]];
 color = :black,
 label = nothing,
)
 end
end
р
```

206



## **Capacitated facility location**

## **Problem formulation**

The capacitated variant introduces a capacity constraint on each facility, that is, clients have a certain level of demand to be served, while each facility only has finite capacity which cannot be exceeded.

Specifically,

- The demand of client i is denoted by  $a_i \geq 0$
- The capacity of facility j is denoted by  $q_j \geq 0$

The capacity constraints then write

$$\sum_{i} a_i x_{i,j} \le q_j y_j \quad \forall j \in N$$

Note that, if  $y_j$  is set to 0, the capacity constraint above automatically forces  $x_{i,j}$  to 0.

Thus, the capacitated facility location can be formulated as follows

$$\min_{x,y} \sum_{i,j} c_{i,j} x_{i,j} + \sum_{j} f_{j} y_{j}$$

$$s.t. \sum_{j} x_{i,j} = 1, \qquad \forall i \in M$$

$$\sum_{i} a_{i} x_{i,j} \leq q_{j} y_{j}, \qquad \forall j \in N$$

$$x_{i,j}, y_{j} \in \{0,1\}, \qquad \forall i \in M, j \in N$$

For simplicity, we will assume that there is enough capacity to serve the demand, that is, there exists at least one feasible solution.

We need some new data:

# Demands
a = rand(1:3, m);
# Capacities
q = rand(5:10, n);

#### Display the data

```
Plots.scatter(
 x_c,
 y_c;
 label = nothing,
 markershape = :circle,
 markercolor = :blue,
 markersize = 2 .* (2 .+ a),
)
Plots.scatter!(
 x_f,
 y_f;
 label = nothing,
 markershape = :rect,
 markercolor = :white,
 markersize = q,
 markerstrokecolor = :red,
 markerstrokewidth = 2,
)
```


## **JuMP** implementation

Create a JuMP model

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, y[1:n], Bin);
@variable(model, x[1:m, 1:n], Bin);
Each client is served exactly once
@constraint(model, client_service[i in 1:m], sum(x[i, :]) == 1);
Capacity constraint
@constraint(model, capacity, x' * a .<= (q .* y));
Objective
@objective(model, Min, f' * y + sum(c .* x));</pre>
```

Solve the problem

```
optimize!(model)
assert_is_solved_and_feasible(model)
println("Optimal value: ", objective_value(model))
```

Optimal value: 6.1980444155009975

### Visualizing the solution

The threshold 1e-5 ensure that edges between clients and facilities are drawn when  $x[i, j] \approx 1$ .

```
x_is_selected = isapprox.(value.(x), 1; atol = 1e-5);
y_is_selected = isapprox.(value.(y), 1; atol = 1e-5);
```

```
Display the solution
```

```
p = Plots.scatter(
 x_c,
 y_c;
 label = nothing,
 markershape = :circle,
 markercolor = :blue,
 markersize = 2 . * (2 . + a),
)
Plots.scatter!(
 x_f,
 y_f;
 label = nothing,
 markershape = :rect,
 markercolor = [(y_is_selected[j] ? :red : :white) for j in 1:n],
 markersize = q,
 markerstrokecolor = :red,
 markerstrokewidth = 2,
)
for i in 1:m, j in 1:n
 if x_is_selected[i, j]
 Plots.plot!(
 [x_c[i], x_f[j]],
 [y_c[i], y_f[j]];
 color = :black,
 label = nothing,
)
 end
end
р
```



## 6.11 Financial modeling problems

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by Arpit Bhatia.

Optimization models play an increasingly important role in financial decisions. Many computational finance problems can be solved efficiently using modern optimization techniques.

In this tutorial we will discuss two such examples taken from (Cornuéjols et al., 2018).

This tutorial uses the following packages

using JuMP import HiGHS

### Short-term financing

Corporations routinely face the problem of financing short term cash commitments such as the following:

Month	Jan	Feb	Mar	Apr	May	Jun
Net Cash Flow	-150	-100	200	-200	50	300

Net cash flow requirements are given in thousands of dollars. The company has the following sources of funds:

• A line of credit of up to \$100K at an interest rate of 1% per month,

- In any one of the first three months, it can issue 90-day commercial paper bearing a total interest of 2% for the 3-month period,
- Excess funds can be invested at an interest rate of 0.3% per month.

Our task is to find out the most economical way to use these 3 sources such that we end up with the most amount of money at the end of June.

We model this problem in the following manner:

We will use the following decision variables:

- the amount  $u_i$  drawn from the line of credit in month i
- the amount  $v_i$  of commercial paper issued in month i
- the excess funds  $w_i$  in month i

Here we have three types of constraints:

- 1. for every month, cash inflow = cash outflow for each month
- 2. upper bounds on  $u_i$
- 3. nonnegativity of the decision variables  $u_i$ ,  $v_i$  and  $w_i$ .

Our objective will be to simply maximize the company's wealth in June, which say we represent with the variable m.

```
financing = Model(HiGHS.Optimizer)
@variables(financing, begin
 0 <= u[1:5] <= 100
 0 <= v[1:3]
 0 <= w[1:5]
 m
end)
@objective(financing, Max, m)
@constraints(
 financing,
 begin
 u[1] + v[1] - w[1] == 150 # January
 u[2] + v[2] - w[2] - 1.01u[1] + 1.003w[1] == 100 # February
 u[3] + v[3] - w[3] - 1.01u[2] + 1.003w[2] == -200 # March
 u[4] - w[4] - 1.02v[1] - 1.01u[3] + 1.003w[3] == 200 # April
 u[5] - w[5] - 1.02v[2] - 1.01u[4] + 1.003w[4] == -50 # May
 -m - 1.02v[3] - 1.01u[5] + 1.003w[5] == -300 # June
 end
)
optimize!(financing)
assert_is_solved_and_feasible(financing)
objective_value(financing)
```

#### 92.49694915254233

#### **Combinatorial auctions**

In many auctions, the value that a bidder has for a set of items may not be the sum of the values that he has for individual items.

Examples are equity trading, electricity markets, pollution right auctions and auctions for airport landing slots.

To take this into account, combinatorial auctions allow the bidders to submit bids on combinations of items.

Let  $M = \{1, 2, ..., m\}$  be the set of items that the auctioneer has to sell. A bid is a pair  $B_j = (S_j, p_j)$  where  $S_j \subseteq M$  is a nonempty set of items and  $p_j$  is the price offer for this set.

Suppose that the auctioneer has received n bids  $B_1, B_2, \ldots, B_n$ . The goal of this problem is to help an auctioneer determine the winners in order to maximize his revenue.

We model this problem by taking a decision variable  $y_j$  for every bid. We add a constraint that each item i is sold at most once. This gives us the following model:

$$\begin{array}{ll} \max & \sum_{i=1}^n p_j y_j \\ \text{s.t.} & \sum_{j:i \in S_j} y_j \leq 1 \quad \forall i = \{1, 2 \dots m\} \\ & y_j \in \{0, 1\} \quad \forall j \in \{1, 2 \dots n\} \end{array}$$

bid\_values = [6 3 12 12 8 16] bid\_items = [[1], [2], [3 4], [1 3], [2 4], [1 3 4]]

```
auction = Model(HiGHS.Optimizer)
@variable(auction, y[1:6], Bin)
@objective(auction, Max, sum(y' .* bid_values))
for i in 1:6
 @constraint(auction, sum(y[j] for j in 1:6 if i in bid_items[j]) <= 1)
end
optimize!(auction)
assert_is_solved_and_feasible(auction)
objective_value(auction)</pre>
```

21.0

value.(y)

```
6-element Vector{Float64}:

1.0

1.0

-0.0

-0.0

0.0
```

## 6.12 Geographical clustering

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by Matthew Helm and Mathieu Tanneau.

The goal of this exercise is to cluster n cities into k groups, minimizing the total pairwise distance between cities *and* ensuring that the variance in the total populations of each group is relatively small.

This tutorial uses the following packages:

using JuMP import DataFrames import HiGHS import LinearAlgebra

For this example, we'll use the 20 most populous cities in the United States.

```
cities = DataFrames.DataFrame(
 Union{String,Float64}[
 "New York, NY" 8.405 40.7127 -74.0059
 "Los Angeles, CA" 3.884 34.0522 -118.2436
 "Chicago, IL" 2.718 41.8781 -87.6297
 "Houston, TX" 2.195 29.7604 -95.3698
 "Philadelphia, PA" 1.553 39.9525 -75.1652
 "Phoenix, AZ" 1.513 33.4483 -112.0740
 "San Antonio, TX" 1.409 29.4241 -98.4936
 "San Diego, CA" 1.355 32.7157 -117.1610
 "Dallas, TX" 1.257 32.7766 -96.7969
 "San Jose, CA" 0.998 37.3382 -121.8863
 "Austin, TX" 0.885 30.2671 -97.7430
 "Indianapolis, IN" 0.843 39.7684 -86.1580
 "Jacksonville, FL" 0.842 30.3321 -81.6556
 "San Francisco, CA" 0.837 37.7749 -122.4194
 "Columbus, OH" 0.822 39.9611 -82.9987
 "Charlotte, NC" 0.792 35.2270 -80.8431
 "Fort Worth, TX" 0.792 32.7554 -97.3307
 "Detroit, MI" 0.688 42.3314 -83.0457
 "El Paso, TX" 0.674 31.7775 -106.4424
 "Memphis, TN" 0.653 35.1495 -90.0489
],
 ["city", "population", "lat", "lon"],
)
```

city	population	lat	lon
Union	Union	Union	Union
New York, NY	8.405	40.7127	-74.0059
Los Angeles, CA	3.884	34.0522	-118.244
Chicago, IL	2.718	41.8781	-87.6297
Houston, TX	2.195	29.7604	-95.3698
Philadelphia, PA	1.553	39.9525	-75.1652
Phoenix, AZ	1.513	33.4483	-112.074
San Antonio, TX	1.409	29.4241	-98.4936
San Diego, CA	1.355	32.7157	-117.161
Dallas, TX	1.257	32.7766	-96.7969
San Jose, CA	0.998	37.3382	-121.886
Austin, TX	0.885	30.2671	-97.743
Indianapolis, IN	0.843	39.7684	-86.158
Jacksonville, FL	0.842	30.3321	-81.6556
San Francisco, CA	0.837	37.7749	-122.419
Columbus, OH	0.822	39.9611	-82.9987
Charlotte, NC	0.792	35.227	-80.8431
Fort Worth, TX	0.792	32.7554	-97.3307
Detroit, MI	0.688	42.3314	-83.0457
El Paso, TX	0.674	31.7775	-106.442
Memphis, TN	0.653	35.1495	-90.0489
	city Union New York, NY Los Angeles, CA Chicago, IL Houston, TX Philadelphia, PA Phoenix, AZ San Antonio, TX San Diego, CA Dallas, TX San Jose, CA Austin, TX Indianapolis, IN Jacksonville, FL San Francisco, CA Columbus, OH Charlotte, NC Fort Worth, TX Detroit, MI El Paso, TX Memphis, TN	city         population           Union         Union           New York, NY         8.405           Los Angeles, CA         3.884           Chicago, IL         2.718           Houston, TX         2.195           Philadelphia, PA         1.553           Phoenix, AZ         1.513           San Antonio, TX         1.409           San Diego, CA         1.355           Dallas, TX         1.257           San Jose, CA         0.998           Austin, TX         0.885           Indianapolis, IN         0.843           Jacksonville, FL         0.842           San Francisco, CA         0.837           Columbus, OH         0.822           Charlotte, NC         0.792           Fort Worth, TX         0.792           Detroit, MI         0.688           El Paso, TX         0.674           Memphis, TN         0.653	citypopulationlatUnionUnionUnionNew York, NY8.40540.7127Los Angeles, CA3.88434.0522Chicago, IL2.71841.8781Houston, TX2.19529.7604Philadelphia, PA1.55339.9525Phoenix, AZ1.51333.4483San Antonio, TX1.40929.4241San Diego, CA1.35532.7157Dallas, TX1.25732.7766San Jose, CA0.99837.3382Austin, TX0.88530.2671Indianapolis, IN0.84339.7684Jacksonville, FL0.84230.3321San Francisco, CA0.83737.7749Columbus, OH0.82239.9611Charlotte, NC0.79235.227Fort Worth, TX0.79232.7554Detroit, MI0.668842.3314El Paso, TX0.67431.7775Memphis, TN0.65335.1495

#### **Model Specifics**

We will cluster these 20 cities into 3 different groups and we will assume that the ideal or target population P for a group is simply the total population of the 20 cities divided by 3:

n = size(cities, 1)
k = 3
P = sum(cities.population) / k

### 11.038333333333334

### Obtaining the distances between each city

Let's compute the pairwise Haversine distance between each of the cities in our data set and store the result in a variable we'll call dm:

```
"""
haversine(lat1, long1, lat2, long2, r = 6372.8)
Compute the haversine distance between two points on a sphere of radius `r`,
where the points are given by the latitude/longitude pairs `lat1/long1` and
`lat2/long2` (in degrees).
"""
function haversine(lat1, long1, lat2, long2, r = 6372.8)
 lat1, long1 = deg2rad(lat1), deg2rad(long1)
 lat2, long2 = deg2rad(lat2), deg2rad(long2)
```

```
hav(a, b) = sin((b - a) / 2)^2
inner_term = hav(lat1, lat2) + cos(lat1) * cos(lat2) * hav(long1, long2)
d = 2 * r * asin(sqrt(inner_term))
Round distance to nearest kilometer.
return round(Int, d)
end
```

```
Main.haversine
```

Our distance matrix is symmetric so we'll convert it to a LowerTriangular matrix so that we can better interpret the objective value of our model:

```
dm = LinearAlgebra.LowerTriangular([
 haversine(cities.lat[i], cities.lon[i], cities.lat[j], cities.lon[j])
 for i in 1:n, j in 1:n
])
```

20×20	Linear	Algebr	a.Lowe	rTrian	gular{	Ма	trix{I	nt64}}	:				
0	•	•	•	•	•	•		•		•	•		•
3937	0	•	•	•	•	•		•	•	•	•	•	•
1145	2805	Θ	•	•	•	•		•	•	•	•	•	•
2282	2207	1516	Θ	•	•	•		•	•	•	•	•	·
130	3845	1068	2157	Θ	•	•		•	•	•	•	•	•
3445	574	2337	1633	3345	Θ	•		•	•	•	•	•	·
2546	1934	1695	304	2423	1363	0		•	•	•	•	•	·
3908	179	2787	2094	3812	481	1813		•	•	•	•	•	·
2206	1993	1295	362	2089	1424	406		•		•	•	•	•
4103	492	2958	2588	4023	989	2336		•	•	•	•	•	•
2432	1972	1577	235	2310	1398	118		•		•	•	•	•
1036	2907	265	1394	938	2409	1609		•	•	•	•	•	·
1345	3450	1391	1321	1221	2883	1626				•	•	•	•
4130	559	2986	2644	4052	1051	2394		•	•	•	•	•	•
767	3177	444	1598	668	2679	1834		Θ	•	•	•	•	·
855	3405	946	1490	725	2863	1777		560	Θ	•	•	•	•
2251	1944	1327	382	2134	1375	387		1511	1543	Θ	•	•	•
774	3186	382	1780	711	2716	1994		264	813	1646	0	•	•
3054	1130	2010	1081	2945	559	804		2292	2398	864	2374	0	·
1534	2576	777	780	1415	2028	1017		820	837	722	1003	1565	0

#### **Build the model**

Now that we have the basics taken care of, we can set up our model, create decision variables, add constraints, and then solve.

First, we'll set up a model that leverages the Cbc solver. Next, we'll set up a binary variable  $x_{i,k}$  that takes the value 1 if city i is in group k and 0 otherwise. Each city must be in a group, so we'll add the constraint  $\sum_k x_{i,k} = 1$  for every i.

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:n, 1:k], Bin)
@constraint(model, [i = 1:n], sum(x[i, :]) == 1);
To reduce symmetry, we fix the first city to belong to the first group.
fix(x[1, 1], 1; force = true)
```

The total population of a group k is  $Q_k = \sum_i x_{i,k}q_i$  where  $q_i$  is simply the *i*-th value from the population column in our cities DataFrame. Let's add constraints so that  $\alpha \leq (Q_k - P) \leq \beta$ . We'll set  $\alpha$  equal to -3 million and  $\beta$  equal to 3. By adjusting these thresholds you'll find that there is a tradeoff between having relatively even populations between groups and having geographically close cities within each group. In other words, the larger the absolute values of  $\alpha$  and  $\beta$ , the closer together the cities in a group will be but the variance between the group populations will be higher.

@variable(model, -3 <= population\_diff[1:k] <= 3)
@constraint(model, population diff .== x' \* cities.population .- P)</pre>

```
3-element Vector{ConstraintRef{Model,

→ MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

→ MathOptInterface.EqualTo{Float64}, ScalarShape}:

-8.405 x[1,1] - 3.884 x[2,1] - 2.718 x[3,1] - 2.195 x[4,1] - 1.553 x[5,1] - 1.513 x[6,1] - 1.409

→ x[7,1] - 1.355 x[8,1] - 1.257 x[9,1] - 0.998 x[10,1] - 0.885 x[11,1] - 0.843 x[12,1] - 0.842

→ x[13,1] - 0.837 x[14,1] - 0.822 x[15,1] - 0.792 x[16,1] - 0.792 x[17,1] - 0.668 x[18,1] -

→ 0.674 x[19,1] - 0.653 x[20,1] + population_diff[1] = -11.0383333333334

-8.405 x[1,2] - 3.884 x[2,2] - 2.718 x[3,2] - 2.195 x[4,2] - 1.553 x[5,2] - 1.513 x[6,2] - 1.409

→ x[7,2] - 1.355 x[8,2] - 1.257 x[9,2] - 0.998 x[10,2] - 0.885 x[11,2] - 0.843 x[12,2] - 0.842

→ x[13,2] - 0.837 x[14,2] - 0.822 x[15,2] - 0.792 x[16,2] - 0.792 x[17,2] - 0.688 x[18,2] -

→ 0.674 x[19,2] - 0.653 x[20,2] + population_diff[2] = -11.0383333333334

-8.405 x[1,3] - 3.884 x[2,3] - 2.718 x[3,3] - 2.195 x[4,3] - 1.553 x[5,3] - 1.513 x[6,3] - 1.409

→ x[7,3] - 1.355 x[8,3] - 1.257 x[9,3] - 0.998 x[10,3] - 0.885 x[11,3] - 0.843 x[12,2] - 0.842

→ x[13,3] - 0.633 x[20,2] + population_diff[2] = -11.0383333333334

-8.405 x[1,3] - 3.884 x[2,3] - 2.718 x[3,3] - 2.195 x[4,3] - 1.553 x[5,3] - 1.513 x[6,3] - 1.409

→ x[7,3] - 1.355 x[8,3] - 1.257 x[9,3] - 0.998 x[10,3] - 0.885 x[11,3] - 0.843 x[12,3] - 0.842

→ x[13,3] - 0.637 x[14,3] - 0.822 x[15,3] - 0.792 x[16,3] - 1.553 x[5,3] - 1.513 x[6,3] - 1.409

→ x[7,3] - 1.355 x[8,3] - 1.257 x[9,3] - 0.998 x[10,3] - 0.885 x[11,3] - 0.688 x[18,3] - 0.842

→ x[13,3] - 0.837 x[14,3] - 0.822 x[15,3] - 0.792 x[16,3] - 1.553 x[5,3] - 1.513 x[6,3] - 1.409

→ x[7,3] - 1.355 x[8,3] - 1.257 x[9,3] - 0.998 x[10,3] - 0.885 x[11,3] - 0.688 x[18,3] - 0.674 x[19,3] - 0.653 x[20,3] + population diff[3] = -11.0383333333334
```

Now we need to add one last binary variable  $z_{i,j}$  to our model that we'll use to compute the total distance between the cities in our groups, defined as  $\sum_{i,j} d_{i,j} z_{i,j}$ . Variable  $z_{i,j}$  will equal 1 if cities *i* and *j* are in the same group, and 0 if they are not in the same group.

To ensure that  $z_{i,j} = 1$  if and only if cities i and j are in the same group, we add the constraints  $z_{i,j} \ge x_{i,k} + x_{j,k} - 1$  for every pair i, j and every k:

```
@variable(model, z[i = 1:n, j = 1:i], Bin)
for k in 1:k, i in 1:n, j in 1:i
 @constraint(model, z[i, j] >= x[i, k] + x[j, k] - 1)
end
```

We can now add an objective to our model which will simply be to minimize the dot product of z and our distance matrix, dm.

```
@objective(model, Min, sum(dm[i, j] * z[i, j] for i in 1:n, j in 1:i));
```

We can then call optimize! and review the results.

optimize!(model)
assert\_is\_solved\_and\_feasible(model)

#### **Reviewing the Results**

Now that we have results, we can add a column to our cities DataFrame for the group and then loop through our x variable to assign each city to its group. Once we have that, we can look at the total population for each group and also look at the cities in each group to verify that they are grouped by geographic proximity.

```
cities.group = zeros(n)
for i in 1:n, j in 1:k
 if round(Int, value(x[i, j])) == 1
 cities.group[i] = j
 end
end
for group in DataFrames.groupby(cities, :group)
 @show group
 println("")
 @show sum(group.population)
 println("")
```

```
end
```

group	group = 7×5 SubDataFrame														
Row	city	population	lat	lon	group										
	Union…	Union…	Union…	Union…	Float64										
1	New York, NY	8.405	40.7127	-74.0059	1.0										
2	Philadelphia, PA	1.553	39.9525	-75.1652	1.0										
3	Indianapolis, IN	0.843	39.7684	-86.158	1.0										
4	Jacksonville, FL	0.842	30.3321	-81.6556	1.0										
5	Columbus, OH	0.822	39.9611	-82.9987	1.0										
6	Charlotte, NC	0.792	35.227	-80.8431	1.0										
7	Detroit, MI	0.688	42.3314	-83.0457	1.0										

sum(group.population) = 13.94499999999999

group	= 7×5 SubDataFram	e				
Row	city	population	lat	lon	group	
	Union…	Union…	Union…	Union…	Float64	
1	Chicago, IL	2.718	41.8781	-87.6297	2.0	
2	Houston, TX	2.195	29.7604	-95.3698	2.0	
3	San Antonio, TX	1.409	29.4241	-98.4936	2.0	
4	Dallas, TX	1.257	32.7766	-96.7969	2.0	

5	Austin, TX	0.885	30.2671	-97.743	2.0
6	Fort Worth, TX	0.792	32.7554	-97.3307	2.0
7	Memphis, TN	0.653	35.1495	-90.0489	2.0

sum(group.population) = 9.909

 $group = 6 \times 5$  SubDataFrame

Row   city	population	lat	lon	group
Union	Union…	Union…	Union…	Float64
1   Los Angeles, CA 2   Phoenix, AZ 3   San Diego, CA 4   San Jose, CA 5   San Francisco, CA 6   El Paso. TX	3.884 1.513 1.355 0.998 0.837 0.674	34.0522 33.4483 32.7157 37.3382 37.7749 31.7775	-118.244 -112.074 -117.161 -121.886 -122.419 -106.442	3.0 3.0 3.0 3.0 3.0 3.0 3.0

sum(group.population) = 9.26100000000001

### 6.13 Network flow problems

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by Arpit Bhatia.

In graph theory, a flow network (also known as a transportation network) is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge.

Often in operations research, a directed graph is called a network, the vertices are called nodes and the edges are called arcs.

A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, unless it is a source, which has only outgoing flow, or sink, which has only incoming flow.

A network can be used to model traffic in a computer network, circulation with demands, fluids in pipes, currents in an electrical circuit, or anything similar in which something travels through a network of nodes.

This tutorial requires the following packages:

using JuMP import HiGHS

#### The shortest path problem

Suppose that each arc (i, j) of a graph is assigned a scalar cost  $a_{i,j}$ , and suppose that we define the cost of a forward path to be the sum of the costs of its arcs.

Given a pair of nodes, the shortest path problem is to find a forward path that connects these nodes and has minimum cost.

$$\min \sum_{\forall e(i,j) \in E} a_{i,j} \times x_{i,j}$$
  
s.t. 
$$\sum_{j} x_{ij} - \sum_{k} x_{ki} = b_{i} \quad \forall i$$
$$x_{e} \in \{0,1\} \quad \forall e \in E$$

where  $b_i$  is 1 if *i* is the starting node, -1 if *i* is the ending node, and 0 otherwise.

```
G = [
 0 100 30 0 0
 0 0 20 0 0
 0 0 0 10 60
 0 15 0 0 50
 \bigcirc \bigcirc \bigcirc \bigcirc \bigcirc \bigcirc \bigcirc \bigcirc
]
n = size(G)[1]
b = [1, -1, 0, 0, 0]
shortest_path = Model(HiGHS.Optimizer)
set_silent(shortest_path)
@variable(shortest_path, x[1:n, 1:n], Bin)
Arcs with zero cost are not a part of the path as they do no exist
\texttt{@constraint(shortest_path, [i = 1:n, j = 1:n; G[i, j] == 0], x[i, j] == 0)}
Flow conservation constraint
@constraint(shortest_path, [i = 1:n], sum(x[i, :]) - sum(x[:, i]) == b[i],)
@objective(shortest_path, Min, sum(G .* x))
optimize!(shortest_path)
assert_is_solved_and_feasible(shortest_path)
objective_value(shortest_path)
```



value.(x)

```
5×5 Matrix{Float64}:

0.0 0.0 1.0 0.0 0.0

0.0 0.0 0.0 0.0 0.0

0.0 0.0 0.0 1.0 0.0

0.0 1.0 0.0 0.0 0.0

0.0 0.0 0.0 0.0 0.0
```

## The assignment problem

Suppose that there are n persons and n objects that we have to match on a one-to-one basis. There is a benefit or value  $a_{i,j}$  for matching person i with object j, and we want to assign persons to objects so as to maximize the total benefit.

There is also a restriction that person i can be assigned to object j only if (i, j) belongs to a given set of pairs A.

Mathematically, we want to find a set of person-object pairs  $(1, j_1), ..., (n, j_n)$  from A such that the objects  $j_1, ..., j_n$  are all distinct, and the total benefit  $\sum_{i=1}^{y} a_{ij_i}$  is maximized.

$$\begin{array}{ll} \max & \sum_{(i,j)\in A} a_{i,j} \times y_{i,j} \\ s.t. & \sum_{\{j \mid (i,j)\in A\}} y_{i,j} = 1 & \quad \forall i = \{1,2....n\} \\ & \sum_{\{i \mid (i,j)\in A\}} y_{i,j} = 1 & \quad \forall j = \{1,2....n\} \\ & y_{i,j} \in \{0,1\} & \forall (i,j) \in \{1,2...k\} \end{array}$$

G = [ 6450 0360 5043 7 5 5 5 ] n = size(G)[1]assignment = Model(HiGHS.Optimizer) set\_silent(assignment) @variable(assignment, y[1:n, 1:n], Bin) # One person can only be assigned to one object @constraint(assignment, [i = 1:n], sum(y[:, i]) == 1) # One object can only be assigned to one person @constraint(assignment, [j = 1:n], sum(y[j, :]) == 1) @objective(assignment, Max, sum(G .\* y)) optimize!(assignment) assert\_is\_solved\_and\_feasible(assignment) objective\_value(assignment)

20.0

value.(y)

4×4 Matrix{Float64}: -0.0 1.0 -0.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0 0.0 -0.0 -0.0 0.0 0.0 1.0

### The max-flow problem

In the max-flow problem, we have a graph with two special nodes: the source, denoted by s, and the sink, denoted by t.

The objective is to move as much flow as possible from s into t while observing the capacity constraints.

$$\begin{aligned} \max & \sum_{v:(s,v)\in E} f(s,v) \\ s.t. & \sum_{u:(u,v)\in E} f(u,v) = \sum_{w:(v,w)\in E} f(v,w) \quad \forall v\in V-\{s,t\} \\ & f(u,v)\leq c(u,v) \quad \quad \forall (u,v)\in E \\ & f(u,v)\geq 0 \quad \quad \forall (u,v)\in E \end{aligned}$$

```
G = [
```

```
0 3 2 2 0 0 0 0
 00005100
 00001310
 0 0 0 0 0 0 1 0 0
 0 0 0 0 0 0 0 4
 00000002
 0 0 0 0 0 0 0 4
 0 0 0 0 0 0 0 0
]
n = size(G)[1]
max_flow = Model(HiGHS.Optimizer)
@variable(max_flow, f[1:n, 1:n] >= 0)
Capacity constraints
@constraint(max_flow, [i = 1:n, j = 1:n], f[i, j] <= G[i, j])</pre>
Flow conservation constraints
@constraint(max_flow, [i = 1:n; i != 1 && i != 8], sum(f[i, :]) == sum(f[:, i]))
@objective(max_flow, Max, sum(f[1, :]))
optimize!(max_flow)
assert_is_solved_and_feasible(max_flow)
objective_value(max_flow)
```

6.0

value.(f)

8×8 Matrix{Float64}: -0.0 3.0 2.0 1.0 -0.0 -0.0 -0.0 -0.0 0.0 0.0 0.0 0.0 2.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 -0.0 1.0 0.0 -0.0 -0.0 -0.0 0.0 -0.0 1.0 -0.0 -0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 3.0 0.0 0.0 0.0 0.0 0.0 0.0 2.0

0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

### 6.14 The transportation problem

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by Louis Luangkesorn.

This tutorial is an adaptation of the transportation problem described in *AMPL: A Modeling Language for Mathematical Programming*, by R. Fourer, D.M. Gay and B.W. Kernighan.

The purpose of this tutorial is to demonstrate how to create a JuMP model from an ad-hoc structured text file.

### **Required packages**

This tutorial uses the following packages:

using JuMP import DelimitedFiles import HiGHS

#### Formulation

Suppose that we have a set of factories that produce pogo sticks, and a set of retail stores in which to sell them. Each factory has a maximum number of pogo sticks that it can produce, and each retail store has a demand of pogo sticks that it can sell.

In the transportation problem, we want to choose the number of pogo sticks to make and ship from each factory to each retail store that minimizes the total shipping cost.

Mathematically, we represent our set of factories by a set of origins  $i \in O$  and our retail stores by a set of destinations  $j \in D$ . The maximum supply at each factory is  $s_i$  and the demand from each retail store is  $d_j$ . The cost of shipping one pogo stick from i to j is  $c_{i,j}$ .

With a little effort, we can model the transportation problem as the following linear program:

$$\begin{array}{ll} \min & \sum_{i \in O, j \in D} c_{i,j} x_{i,j} \\ s.t. & \sum_{j \in D} x_{i,j} \leq s_i \qquad \forall i \in O \\ & \sum_{i \in O} x_{i,j} = d_j \qquad \forall j \in D \\ & x_{i,j} \geq 0 \quad \forall i \in O, j \in D \end{array}$$

Data

We assume our data is in the form of a text file that has the following form. In practice, we would obtain this text file from the user as input, but for the purpose of this tutorial we're going to create it from Julia.

```
open(joinpath(@_DIR_, "transp.txt"), "w") do io
print(
 io,
 """
 . FRA DET LAN WIN STL FRE LAF SUPPLY
 GARY 39 14 11 14 16 82 8 1400
 CLEV 27 . 12 . 26 95 17 2600
 PITT 24 14 17 13 28 99 20 2900
 DEMAND 900 1200 600 400 1700 1100 1000 0
 """,
)
 return
end
```

Here the rows are the origins, the columns are the destinations, and the values are the cost of shipping one pogo stick from the origin to the destination. If pogo stick cannot be transported from a source to a destination, then the value is ... The final row and final column are the demand and supply of each location respectively.

We didn't account for arcs which do not exist in our formulation, but we can make a small change and fix  $x_{i,j} = 0$  if  $c_{i,j} = "."$ 

Our first step is to convert this text format into an appropriate Julia datastructure that we can work with. Since our data is tabular with named rows and columns, one option is JuMP's Containers.DenseAxisArray object:

```
function read_data(filename::String)
 data = DelimitedFiles.readdlm(filename)
 rows, columns = data[2:end, 1], data[1, 2:end]
 return Containers.DenseAxisArray(data[2:end, 2:end], rows, columns)
end
```

data = read\_data(joinpath(@\_\_DIR\_\_, "transp.txt"))

```
2-dimensional DenseAxisArray{Any,2,...} with index sets:
 Dimension 1, Any["GARY", "CLEV", "PITT", "DEMAND"]
 Dimension 2, Any["FRA", "DET", "LAN", "WIN", "STL", "FRE", "LAF", "SUPPLY"]
And data, a 4×8 Matrix{Any}:
 39
 14 11 14 16 82
 8 1400
 "." 12 "."
 26 95 17 2600
 27
 17 13 28 99 20 2900
 14
 24
 600 400 1700 1100 1000
900 1200
 Θ
```

#### **JuMP** formulation

Following Design patterns for larger models, we code our JuMP model as a function which takes in an input. In this example, we print the output to stdout:

function solve\_transportation\_problem(data::Containers.DenseAxisArray)
# Get the set of supplies and demands
0, D = axes(data)
# Drop the SUPPLY and DEMAND nodes from our sets
0, D = setdiff(0, ["DEMAND"]), setdiff(D, ["SUPPLY"])

```
model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, x[o in 0, d in D] >= 0)
 # Remove arcs with "." cost by fixing them to 0.0.
 for o in O, d in D
 if data[o, d] == "."
 fix(x[o, d], 0.0; force = true)
 end
 end
 @objective(
 model,
 Min,
 sum(data[o, d] * x[o, d] for o in 0, d in D if data[o, d] != "."),
)
 @constraint(model, [o in 0], sum(x[o, :]) <= data[o, "SUPPLY"])</pre>
 @constraint(model, [d in D], sum(x[:, d]) == data["DEMAND", d])
 optimize!(model)
 assert_is_solved_and_feasible(model)
 # Pretty print the solution in the format of the input
 print(" ", join(lpad.(D, 7, ' ')))
 for o in O
 print("\n", o)
 for d in D
 if isapprox(value(x[o, d]), 0.0; atol = 1e-6)
 print("
 .")
 else
 print(" ", lpad(value(x[o, d]), 6, ' '))
 end
 end
 end
 return
end
```

solve\_transportation\_problem (generic function with 1 method)

## Solution

Let's solve and view the solution:

solve\_transportation\_problem(data)

	FRA	DET	LAN	WIN	STL	FRE	LAF
GARY					300.0	1100.0	
CLEV			600.0		1000.0		1000.0
PITT	900.0	1200.0	•	400.0	400.0	•	

## 6.15 Multi-objective knapsack

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to demonstrate how to create and solve a multi-objective linear program. In addition, it demonstrates how to work with solvers which return multiple solutions.

### **Required packages**

This tutorial requires the following packages:

using JuMP
import HiGHS
import MultiObjectiveAlgorithms as MOA
import Plots

MultiObjectiveAlgorithms.jl is a package which implements a variety of algorithms for solving multi-objective optimization problems. Because it is a long package name, we import it instead as MOA.

### Formulation

The knapsack problem is a classic problem in mixed-integer programming. Given a collection of items  $i \in I$ , each of which has an associated weight,  $w_i$ , and profit,  $p_i$ , the knapsack problem determines which profitmaximizing subset of items to pack into a knapsack such that the total weight is less than a capacity c. The mathematical formulation is:

$$\max \sum_{i \in I} p_i x_i$$
  
s.t. 
$$\sum_{i \in I} w_i x_i \le c$$
$$x_i \in \{0, 1\} \qquad \forall i \in I$$

where  $x_i$  is 1 if we pack item i into the knapsack and 0 otherwise.

For this tutorial, we extend the single-objective knapsack problem by adding another objective: given a desirability rating,  $r_i$ , we wish to maximize the total desirability of the items in our knapsack. Thus, our mathematical formulation is now:

$$\max \sum_{i \in I} p_i x_i$$
$$\sum_{i \in I} r_i x_i$$
s.t. 
$$\sum_{i \in I} w_i x_i \le c$$
$$x_i \in \{0, 1\} \quad \forall i$$

 $\in I$ 

Data

The data for this example was taken from vOptGeneric, and the original author was @xgandibleux.

### CHAPTER 6. LINEAR PROGRAMS

```
profit = [77, 94, 71, 63, 96, 82, 85, 75, 72, 91, 99, 63, 84, 87, 79, 94, 90]
desire = [65, 90, 90, 77, 95, 84, 70, 94, 66, 92, 74, 97, 60, 60, 65, 97, 93]
weight = [80, 87, 68, 72, 66, 77, 99, 85, 70, 93, 98, 72, 100, 89, 67, 86, 91]
capacity = 900
N = length(profit)
```

```
17
```

Comparing the capacity to the total weight of all the items:

capacity / sum(weight)

0.6428571428571429

shows that we can take approximately 64% of the items.

Plotting the items, we see that there are a range of items with different profits and desirability. Some items have a high profit and a high desirability, others have a low profit and a high desirability (and vice versa).

```
Plots.scatter(
 profit,
 desire;
 xlabel = "Profit",
 ylabel = "Desire",
 legend = false,
)
```



The goal of the bi-objective knapsack problem is to choose a subset which maximizes both objectives.

#### **JuMP** formulation

Our JuMP formulation is a direct translation of the mathematical formulation:

```
model = Model()
@variable(model, x[1:N], Bin)
@constraint(model, sum(weight[i] * x[i] for i in 1:N) <= capacity)
@expression(model, profit_expr, sum(profit[i] * x[i] for i in 1:N))
@expression(model, desire_expr, sum(desire[i] * x[i] for i in 1:N))
@objective(model, Max, [profit_expr, desire_expr])</pre>
```

```
2-element Vector{AffExpr}:

77 x[1] + 94 x[2] + 71 x[3] + 63 x[4] + 96 x[5] + 82 x[6] + 85 x[7] + 75 x[8] + 72 x[9] + 91 x[10]

\rightarrow + 99 x[11] + 63 x[12] + 84 x[13] + 87 x[14] + 79 x[15] + 94 x[16] + 90 x[17]

65 x[1] + 90 x[2] + 90 x[3] + 77 x[4] + 95 x[5] + 84 x[6] + 70 x[7] + 94 x[8] + 66 x[9] + 92 x[10]

\rightarrow + 74 x[11] + 97 x[12] + 60 x[13] + 60 x[14] + 65 x[15] + 97 x[16] + 93 x[17]
```

Note how we form a multi-objective program by passing a vector of scalar objective functions.

### Solution

To solve our model, we need an optimizer which supports multi-objective linear programs. One option is to use the MultiObjectiveAlgorithms.jl package.

```
set_optimizer(model, () -> MOA.Optimizer(HiGHS.Optimizer))
set_silent(model)
```

MultiObjectiveAlgorithms.jl supports many different algorithms for solving multiobjective optimization problems. One option is the epsilon-constraint method:

```
set_attribute(model, MOA.Algorithm(), MOA.EpsilonConstraint())
```

Let's solve the problem and see the solution

optimize!(model)
assert\_is\_solved\_and\_feasible(model)
solution\_summary(model)

```
solution_summary(; result = 1, verbose = false)
| solver_name : MOA[algorithm=MultiObjectiveAlgorithms.EpsilonConstraint, optimizer=HiGHS]
| Termination
| termination_status : OPTIMAL
| result_count : 9
| traw_status : Solve complete. Found 9 solution(s)
| tobjective_bound : [9.55000e+02,9.83000e+02]
+ Solution (result = 1)
| terminal_status : FEASIBLE_POINT
| tobjective_value : [9.18000e+02,9.83000e+02]
+ Work counters
 t_solve_time (sec) : 9.27551e-02
```

There are 9 solutions available. We can also use result\_count to see how many solutions are available:

result\_count(model)

9

### Accessing multiple solutions

Access the nine different solutions in the model using the result keyword to solution\_summary, value, and objective\_value:

solution\_summary(model; result = 5)

```
solution_summary(; result = 5, verbose = false)
L Solution (result = 5)
primal_status : FEASIBLE_POINT
dual_status : N0_SOLUTION
L objective_value : [9.36000e+02,9.42000e+02]
```

```
@assert primal_status(model; result = 5) == FEASIBLE_POINT
```

```
assert_is_solved_and_feasible(model; result = 5)
```

objective\_value(model; result = 5)

2-element Vector{Float64}: 936.0 942.0

Note that because we set a vector of two objective functions, the objective value is a vector with two elements. We can also query the value of each objective separately:

```
value(profit_expr; result = 5)
```

936.0

### Visualizing objective space

Unlike single-objective optimization problems, multi-objective optimization problems do not have a single optimal solution. Instead, the solutions returned represent possible trade-offs that the decision maker can choose between the two objectives. A common way to visualize this is by plotting the objective values of each of the solutions:

```
plot = Plots.scatter(
 [value(profit_expr; result = i) for i in 1:result_count(model)],
 [value(desire_expr; result = i) for i in 1:result_count(model)];
 xlabel = "Profit",
 ylabel = "Desire",
 title = "Objective space",
 label = "",
 xlims = (915, 960),
)
for i in 1:result_count(model)
 y = objective_value(model; result = i)
 Plots.annotate!(y[1] - 1, y[2], (i, 10))
```

#### end

ideal\_point = objective\_bound(model)
Plots.scatter!([ideal\_point[1]], [ideal\_point[2]]; label = "Ideal point")



Visualizing the objective space lets the decision maker choose a solution that suits their personal preferences. For example, result #7 is close to the maximum value of profit, but offers significantly higher desirability compared with solutions #8 and #9.

The set of items that are chosen in solution #7 are:

items\_chosen = [i for i in 1:N if value(x[i]; result = 7) > 0.9]

#### Next steps

MultiObjectiveAlgorithms.jl implements a number of different algorithms. Try solving the same problem using MOA.Dichotomy(). Does it find the same solution?

### 6.16 Simple multi-objective examples

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial contains a number of examples of multi-objective programs from the literature.

## **Required packages**

This tutorial requires the following packages:

using JuMP
import HiGHS
import MultiObjectiveAlgorithms as MOA

#### **Bi-objective linear problem**

This example is taken from Example 6.3 (from Steuer, 1985), page 154 of Ehrgott, M. (2005). *Multicriteria Optimization*. Springer, Berlin. The code was adapted from an example in vOptGeneric by @xgandibleux.

```
model = Model()
set_silent(model)
@variable(model, x1 >= 0)
@variable(model, 0 <= x2 <= 3)
@objective(model, Min, [3x1 + x2, -x1 - 2x2])
@constraint(model, 3x1 - x2 <= 6)
set_optimizer(model, () -> MOA.Optimizer(HiGHS.Optimizer))
set_attribute(model, MOA.Algorithm(), MOA.Lexicographic())
optimize!(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : MOA[algorithm=MultiObjectiveAlgorithms.Lexicographic, optimizer=HiGHS]
| Termination
| termination_status : OPTIMAL
| result_count : 2
| raw_status : Solve complete. Found 2 solution(s)
| tobjective_bound : [0.00000e+00, -9.00000e+00]
| Solution (result = 1)
| primal_status : FEASIBLE_POINT
| tobjective_value : [0.00000e+00,0.00000e+00]
| tobjective_value : [0.00000e+00,0.00000e+00]
| Vork counters
| solve_time (sec) : 1.22094e-03
```

```
for i in 1:result_count(model)
 assert_is_solved_and_feasible(model; result = i)
 print(i, ": z = ", round.(Int, objective_value(model; result = i)), " | ")
 println("x = ", value.([x1, x2]; result = i))
end
```

1: z = [0, 0] | x = [0.0, -0.0]2: z = [12, -9] | x = [3.0, 3.0]

#### Bi-objective linear assignment problem

This example is taken from Example 9.38 (from Ulungu and Teghem, 1994), page 255 of Ehrgott, M. (2005). *Multicriteria Optimization*. Springer, Berlin. The code was adapted from an example in vOptGeneric by @xgandibleux.

```
C1 = [5 1 4 7; 6 2 2 6; 2 8 4 4; 3 5 7 1]
C2 = [3 6 4 2; 1 3 8 3; 5 2 2 3; 4 2 3 5]
n = size(C2, 1)
model = Model()
set_silent(model)
@variable(model, x[1:n, 1:n], Bin)
@objective(model, Min, [sum(C1 .* x), sum(C2 .* x)])
@constraint(model, [i = 1:n], sum(x[i, :]) == 1)
@constraint(model, [j = 1:n], sum(x[:, j]) == 1)
set_optimizer(model, () -> MOA.Optimizer(HiGHS.Optimizer))
set_attribute(model, MOA.Algorithm(), MOA.EpsilonConstraint())
optimize!(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
- solver_name
 : MOA[algorithm=MultiObjectiveAlgorithms.EpsilonConstraint, optimizer=HiGHS]
F Termination
| | result_count : 6
 : Solve complete. Found 6 solution(s)
L objective_bound : [6.00000e+00,7.00000e+00]
Solution (result = 1)
: FEASIBLE_POINT
 : NO_SOLUTION
| ^L objective_value
 : [6.00000e+00,2.40000e+01]
^L Work counters
 L solve_time (sec) : 6.44803e-03
```

```
for i in 1:result_count(model)
 assert_is_solved_and_feasible(model; result = i)
 print(i, ": z = ", round.(Int, objective_value(model; result = i)), " | ")
 println("x = ", round.(Int, value.(x; result = i)))
end
```

1: z = [6, 24] | x = [0 1 0 0; 0 0 1 0; 1 0 0 0; 0 0 0 1]2: z = [9, 17] | x = [0 0 1 0; 0 1 0 0; 1 0 0; 0 0 0; 0 0 0 1]3: z = [12, 13] | x = [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1]4: z = [16, 11] | x = [0 0 0 1; 0 1 0 0; 0 0 1 0; 1 0 0]5: z = [19, 10] | x = [0 0 1 0; 1 0 0; 0 0 1; 0 1 00]6: z = [22, 7] | x = [0 0 0 1; 1 0 0 0; 0 0 1 0; 0 1 0]

#### **Bi-objective shortest path problem**

This example is taken from Exercise 9.5 page 269 of Ehrgott, M. (2005). *Multicriteria Optimization*. Springer, Berlin. The code was adapted from an example in vOptGeneric by @xgandibleux.

```
M = 50
C1 = [
 M 4 5 M M M
 MM2127
 M M M 5 2 M
 M M 5 M M 3
 M M M M M 4
 ммммм
1
C2 = [
 M 3 1 M M M
 MM1422
 M M M 1 7 M
 M M 1 M M 2
 M M M M M 2
 ммммм
]
n = size(C2, 1)
model = Model()
set_silent(model)
@variable(model, x[1:n, 1:n], Bin)
@objective(model, Min, [sum(C1 .* x), sum(C2 .* x)])
@constraint(model, sum(x[1, :]) == 1)
@constraint(model, sum(x[:, n]) == 1)
@constraint(model, [i = 2:n-1], sum(x[i, :]) - sum(x[:, i]) == 0)
set_optimizer(model, () -> MOA.Optimizer(HiGHS.Optimizer))
set_attribute(model, MOA.Algorithm(), MOA.EpsilonConstraint())
optimize!(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
+ solver_name : MOA[algorithm=MultiObjectiveAlgorithms.EpsilonConstraint, optimizer=HiGHS]
+ Termination
| + termination_status : OPTIMAL
| + result_count : 4
| + result_count : 4
| + raw_status : Solve complete. Found 4 solution(s)
| L objective_bound : [8.00000e+00,4.00000e+00]
+ Solution (result = 1)
| + primal_status : FEASIBLE_POINT
```

```
| | dual_status : N0_SOLUTION
| L objective_value : [8.00000e+00,9.00000e+00]
L Work counters
L solve_time (sec) : 4.02594e-03
```

```
for i in 1:result_count(model)
 assert_is_solved_and_feasible(model; result = i)
 print(i, ": z = ", round.(Int, objective_value(model; result = i)), " | ")
 X = round.(Int, value.(x; result = i))
 print("Path:")
 for ind in findall(val -> val ≈ 1, X)
 i, j = ind.I
 print(" $i->$j")
 end
 println()
end
```

1: z = [8, 9] | Path: 1->2 2->4 4->6 2: z = [10, 7] | Path: 1->2 2->5 5->6 3: z = [11, 5] | Path: 1->2 2->6 4: z = [13, 4] | Path: 1->3 3->4 4->6

# 6.17 Sudoku

This tutorial was generated using Literate.jl. Download the source as a .jl file.

### This tutorial was originally contributed by lain Dunning.

Sudoku is a popular number puzzle. The goal is to place the digits 1 to 9 on a nine-by-nine grid, with some of the digits already filled in. Your solution must satisfy the following rules:

- The numbers 1 to 9 must appear in each 3x3 square
- The numbers 1 to 9 must appear in each row
- The numbers 1 to 9 must appear in each column

Here is a partially solved Sudoku problem:

Solving a Sudoku isn't an optimization problem with an objective; its actually a *feasibility* problem: we wish to find a feasible solution that satisfies these rules. You can think of it as an optimization problem with an objective of 0.

### Mixed-integer linear programming formulation

We can model this problem using 0-1 integer programming: a problem where all the decision variables are binary. We'll use JuMP to create the model, and then we can solve it with any integer programming solver.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 6.1: Partially solved Sudoku

using JuMP using HiGHS

We will define a binary variable (a variable that is either 0 or 1) for each possible number in each possible cell. The meaning of each variable is as follows: x[i,j,k] = 1 if and only if cell (i,j) has number k, where i is the row and j is the column.

Create a model

sudoku = Model(HiGHS.Optimizer)
set\_silent(sudoku)

Create our variables

```
@variable(sudoku, x[i = 1:9, j = 1:9, k = 1:9], Bin);
```

Now we can begin to add our constraints. We'll actually start with something obvious to us as humans, but what we need to enforce: that there can be only one number per cell.

```
for i in 1:9 # For each row
 for j in 1:9 # and each column
 # Sum across all the possible digits. One and only one of the digits
 # can be in this cell, so the sum must be equal to one.
 @constraint(sudoku, sum(x[i, j, k] for k in 1:9) == 1)
 end
end
```

Next we'll add the constraints for the rows and the columns. These constraints are all very similar, so much so that we can actually add them at the same time.

```
for ind in 1:9 # Each row, OR each column
 for k in 1:9 # Each digit
 # Sum across columns (j) - row constraint
 @constraint(sudoku, sum(x[ind, j, k] for j in 1:9) == 1)
 # Sum across rows (i) - column constraint
 @constraint(sudoku, sum(x[i, ind, k] for i in 1:9) == 1)
 end
end
```

Finally, we have the to enforce the constraint that each digit appears once in each of the nine 3x3 sub-grids. Our strategy will be to index over the top-left corners of each 3x3 square with for loops, then sum over the squares.

```
for i in 1:3:7
 for j in 1:3:7
 for k in 1:9
 # i is the top left row, j is the top left column.
 # We'll sum from i to i+2, for example, i=4, r=4, 5, 6.
 @constraint(
 sudoku,
 sum(x[r, c, k] for r in i:(i+2), c in j:(j+2)) == 1
)
 end
 end
end
end
```

The final step is to add the initial solution as a set of constraints. We'll solve the problem that is in the picture at the start of the tutorial. We'll put a 0 if there is no digit in that location.

The given digits

```
init_sol = [
 5 3 0 0 7 0 0 0 0
 6 0 0 1 9 5 0 0 0
```

```
0 9 8 0 0 0 0 6 0
 8 0 0 0 6 0 0 0 3
 4 0 0 8 0 3 0 0 1
 7 0 0 0 2 0 0 0 6
 0 6 0 0 0 0 2 8 0
 0 0 0 4 1 9 0 0 5
 0 0 0 0 8 0 0 7 9
]
for i in 1:9
 for j in 1:9
 # If the space isn't empty
 if init_sol[i, j] != 0
 # Then the corresponding variable for that digit and location must
 # be 1.
 fix(x[i, j, init_sol[i, j]], 1; force = true)
 end
 end
end
```

solve problem

optimize!(sudoku)
assert\_is\_solved\_and\_feasible(sudoku)

Extract the values of x

x\_val = value.(x);

Create a matrix to store the solution

Display the solution

sol

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	S	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	З	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 6.2: Solved Sudoku

9×9	Ma	tri	×{I	nt6	4}:				
5	3	4	6	7	8	9	1	2	
6	7	2	1	9	5	3	4	8	
1	9	8	3	4	2	5	6	7	
8	5	9	7	6	1	4	2	3	
4	2	6	8	5	3	7	9	1	
7	1	3	9	2	4	8	5	6	
9	6	1	5	3	7	2	8	4	
2	8	7	4	1	9	6	3	5	
3	4	5	2	8	6	1	7	9	

Which is the correct solution:

### **Constraint programming formulation**

We can also model this problem using constraint programming and the all-different constraint, which says that no two elements of a vector can take the same value.

Because of the reformulation system in MathOptInterface, we can still solve this problem using HiGHS.

```
model = Model(HiGHS.Optimizer)
set_silent(model)
HiGHS v1.2 has a bug in presolve which causes the problem to be classified as
infeasible.
set_attribute(model, "presolve", "off")
```

Instead of the binary variables, we directly define a 9x9 grid of integer values between 1 and 9:

@variable(model, 1 <= x[1:9, 1:9] <= 9, Int);</pre>

Then, we enforce that the values in each row must be all-different:

@constraint(model, [i = 1:9], x[i, :] in MOI.AllDifferent(9));

That the values in each column must be all-different:

@constraint(model, [j = 1:9], x[:, j] in MOI.AllDifferent(9));

And that the values in each 3x3 sub-grid must be all-different:

```
for i in (0, 3, 6), j in (0, 3, 6)
 @constraint(model, vec(x[i.+(1:3), j.+(1:3)]) in MOI.AllDifferent(9))
end
```

Finally, as before we set the initial solution and optimize:

```
for i in 1:9, j in 1:9
 if init_sol[i, j] != 0
 fix(x[i, j], init_sol[i, j]; force = true)
 end
end
optimize!(model)
assert_is_solved_and_feasible(model)
```

Display the solution

csp\_sol = round.(Int, value.(x))

9x9 Matrix{Int64}:
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9

Which is the same as we found before:

sol == csp\_sol

true

## 6.18 N-Queens

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by Matthew Helm and Mathieu Tanneau.

The N-Queens problem involves placing N queens on an N x N chessboard such that none of the queens attacks another. In chess, a queen can move vertically, horizontally, and diagonally so there cannot be more than one queen on any given row, column, or diagonal.

Note that none of the queens above are able to attack any other as a result of their careful placement.

```
using JuMP
import HiGHS
import LinearAlgebra
```

N-Queens

N = 8

```
model = Model(HiGHS.Optimizer)
set_silent(model)
```

Next, let's create an N x N chessboard of binary values. 0 will represent an empty space on the board and 1 will represent a space occupied by one of our queens:

@variable(model, x[1:N, 1:N], Bin);

Now we can add our constraints:

There must be exactly one queen in a given row/column



Figure 6.3: Four Queens

```
for i in 1:N
 @constraint(model, sum(x[i, :]) == 1)
 @constraint(model, sum(x[:, i]) == 1)
end
```

There can only be one queen on any given diagonal

```
for i in -(N - 1):(N-1)
 @constraint(model, sum(LinearAlgebra.diag(x, i)) <= 1)
 @constraint(model, sum(LinearAlgebra.diag(reverse(x; dims = 1), i)) <= 1)
end</pre>
```

We are ready to put our model to work and see if it is able to find a feasible solution:

optimize!(model)
assert\_is\_solved\_and\_feasible(model)

We can now review the solution that our model found:

```
solution = round.(Int, value.(x))
```

```
8×8 Matrix{Int64}:
0 0 0 0 1 0 0 0
```

0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	0

## 6.19 Constraint programming

This tutorial was generated using Literate.jl. Download the source as a .jl file.

JuMP supports a range of constraint-programming type constraints via the corresponding sets in MathOptInterface. For most constraints, there are reformulations built-in that convert the constraint programming constraint into a mixed-integer programming equivalent.

Because of this reformulation, all variables must be integer, and they must typically have finite bounds. An error will be thrown if the reformulation requires finiteness and you have a variable with non-finite bounds.

This tutorial uses the following packages:

using JuMP import HiGHS

#### AllDifferent

The MOI.AllDifferent set ensures that every element in a list takes a different integer value.

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 1 <= x[1:4] <= 4, Int)
@constraint(model, x in MOI.AllDifferent(4))
optimize!(model)
assert_is_solved_and_feasible(model)
value.(x)</pre>
```

```
4-element Vector{Float64}:

1.0

4.0

3.0000000000000004

1.999999999999999
```

## BinPacking

The MOI.BinPacking set can be used to divide up a set of items into different groups, such that the sum of their weights does not exceed the capacity of a bin.

```
weights, capacity = Float64[1, 1, 2, 2, 3], 3.0;
number_of_bins = 3
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 1 <= x[1:length(weights)] <= number_of_bins, Int)
@constraint(model, x in MOI.BinPacking(capacity, weights))
optimize!(model)
assert_is_solved_and_feasible(model)
value.(x)
```

5-element Vector{Float64}: 2.0 1.0 2.0 1.0 3.0

Here, the value of x[i] is the bin that item i was placed into.

## Circuit

The MOI.Circuit set is used to construct a tour of a list of N variables. They will each be assigned an integer from 1 to N, that describes the successor to each variable in the list:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:4], Int)
@constraint(model, x in MOI.Circuit(4))
optimize!(model)
assert_is_solved_and_feasible(model)
```

Let's see what tour was found, starting at node number 1:

```
y = round.(Int, value.(x))
tour = Int[1]
while length(tour) < length(y)
 push!(tour, y[tour[end]])
end
tour</pre>
```

4-element Vector{Int64}:
1
4
3
2
CHAPTER 6. LINEAR PROGRAMS

## CountAtLeast

The MOI.CountAtLeast set is used to ensure that at least n elements in a set of variables belong to a set of values.

For example, here is a model with three variables, constrained between 0 and 5:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 0 <= x[1:3] <= 5, Int)</pre>
```

```
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
```

If we want to ensure that at least one element of each set  $\{x[1], x[2]\}$  and  $\{x[2], x[3]\}$  is in the set  $\{3\}$ , then we create a list of variables by concatenating the sets together:

```
variables = [x[1], x[2], x[2], x[3]]
```

```
4-element Vector{VariableRef}:
 x[1]
 x[2]
 x[2]
 x[3]
```

Then we need a partition list that contains the number of elements in each set of variables:

partitions = [2, 2]

2-element Vector{Int64}:
2
2

Finally, we need a set of values that the elements must be a part of:

```
values = Set([3])
```

Set{Int64} with 1 element:
 3

And the number of elements that must be part of the set values:

n = 1

1

The constraint is:

@constraint(model, variables in MOI.CountAtLeast(n, partitions, values))

 $[x_1, x_2, x_2, x_3] \in MathOptInterface.CountAtLeast(1, [2, 2], Set([3]))$ 

To ensure the uniqueness of the solution, we'll add a constraint that x[2] must be <= 2. This ensures that the only feasible solution is for x[1] and x[3] to be 3:

@constraint(model, x[2] <= 2)</pre>

 $x_2 \leq 2$ 

Let's check that we found a valid solution:

```
optimize!(model)
assert_is_solved_and_feasible(model)
value.(x)
```

```
3-element Vector{Float64}:
3.0
0.0
3.0
```

## CountBelongs

The MOI. CountBelongs set is used to count how many elements in a set of variables belong to a set of values.

For example, to count how many elements in a set of 4 variables belong to the set {2, 3}, do:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 0 <= x[i = 1:4] <= i, Int)
@variable(model, n, Int)
@objective(model, Max, sum(x))
set = Set([2, 3])
@constraint(model, [n; x] in MOI.CountBelongs(1 + length(x), set))
optimize!(model)
assert_is_solved_and_feasible(model)
value(n), value.(x)</pre>
```

(2.0, [1.0, 2.0, 3.0, 4.0])

## CountDistinct

The MOI. CountDistinct set is used to count the number of distinct elements in a set of variables.

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 0 <= x[i = 1:4] <= i, Int)
@variable(model, n, Int)
@objective(model, Max, sum(x))
@constraint(model, [n; x] in MOI.CountDistinct(1 + length(x)))
optimize!(model)
assert_is_solved_and_feasible(model)
value(n), value.(x)</pre>
```

(4.0, [1.0, 2.0, 3.0, 4.0])

#### CountGreaterThan

The MOI.CountGreaterThan set is used to strictly upper-bound the number of distinct elements in a set of variables that have a value equal to another variable.

For example, to count the number n of times that y appears in the vector x, use:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 0 <= x[i = 1:4] <= i, Int)
@variable(model, n, Int)
@variable(model, 3 <= y <= 4, Int)
@objective(model, Max, sum(x))
@constraint(model, [n; y; x] in MOI.CountGreaterThan(1 + 1 + length(x)))
optimize!(model)
assert_is_solved_and_feasible(model)
value(n), value(y), value.(x)</pre>
```

```
(2.0, 3.0, [1.0, 2.0, 3.0, 4.0])
```

Here n is strictly greater than the count, and there is no limit on how large n could be. For example, n = 100 is also a feasible solution. The only constraint is that n cannot be equal to or smaller than the number of times that y appears.

## Table

The MOI. Table set is used to select a single row from a matrix of values.

For example, given a matrix:

table = Float64[1 1 0; 0 1 1; 1 0 1; 1 1 1]

4×3 Matrix{Float64}:
 1.0 1.0 0.0
 0.0 1.0 1.0
 1.0 0.0 1.0
 1.0 1.0

we can constraint a 3-element vector x to equal one of the rows in table via:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[i = 1:3], Int)
@constraint(model, x in MOI.Table(table))
optimize!(model)
assert_is_solved_and_feasible(model)
value.(x)
```

```
3-element Vector{Float64}:
1.0
1.0
1.0
```

## 6.20 Callbacks

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of the tutorial is to demonstrate the various solver-independent and solver-dependent callbacks that are supported by JuMP.

The tutorial uses the following packages:

```
using JuMP
import Gurobi
import Random
import Test
```

#### Info

This tutorial uses the MathOptInterface API. By default, JuMP exports the MOI symbol as an alias for the MathOptInterface.jl package. We recommend making this more explicit in your code by adding the following lines:

import MathOptInterface as MOI

#### Lazy constraints

An example using a lazy constraint callback.

```
function example_lazy_constraint()
 model = Model(Gurobi.Optimizer)
 set_silent(model)
 @variable(model, 0 <= x <= 2.5, Int)</pre>
 @variable(model, 0 <= y <= 2.5, Int)</pre>
 @objective(model, Max, y)
 lazy_called = false
 function my_callback_function(cb_data)
 lazy_called = true
 x_val = callback_value(cb_data, x)
 y_val = callback_value(cb_data, y)
 println("Called from (x, y) = ($x_val, $y_val)")
 status = callback_node_status(cb_data, model)
 if status == MOI.CALLBACK_NODE_STATUS_FRACTIONAL
 println(" - Solution is integer infeasible!")
 elseif status == MOI.CALLBACK_NODE_STATUS_INTEGER
 println(" - Solution is integer feasible!")
 else
 @assert status == MOI.CALLBACK_NODE_STATUS_UNKNOWN
 println(" - I don't know if the solution is integer feasible :(")
 end
 if y_val - x_val > 1 + 1e-6
 con = @build_constraint(y - x <= 1)</pre>
 println("Adding $(con)")
 MOI.submit(model, MOI.LazyConstraint(cb_data), con)
 elseif y_val + x_val > 3 + 1e-6
 con = @build_constraint(y + x <= 3)</pre>
 println("Adding $(con)")
 MOI.submit(model, MOI.LazyConstraint(cb_data), con)
 end
 return
 end
 set_attribute(model, MOI.LazyConstraintCallback(), my_callback_function)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 Test.@test lazy called
 Test.@test value(x) == 1
 Test.@test value(y) == 2
 println("Optimal solution (x, y) = ($(value(x)), $(value(y)))")
 return
end
```

example\_lazy\_constraint()

```
Set parameter WLSAccessID
Set parameter WLSSecret
Set parameter LicenseID to value 722777
WLS license 722777 - registered to JuMP Development
Called from (x, y) = (-0.0, 2.0)
- Solution is integer feasible!
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(y - x,
→ MathOptInterface.LessThan{Float64}(1.0))
```

```
Called from (x, y) = (2.0, 2.0)
- Solution is integer feasible!
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(y + x,
\rightarrow MathOptInterface.LessThan{Float64}(3.0))
Called from (x, y) = (2.0, 2.0)
 - Solution is integer feasible!
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(y + x,
\hookrightarrow MathOptInterface.LessThan{Float64}(3.0))
Called from (x, y) = (2.0, 2.0)
- Solution is integer feasible!
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(y + x,
\rightarrow MathOptInterface.LessThan{Float64}(3.0))
Called from (x, y) = (-0.0, 2.0)
 - Solution is integer feasible!
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(y - x,
→ MathOptInterface.LessThan{Float64}(1.0))
Called from (x, y) = (1.0, 2.0)
 - Solution is integer feasible!
Optimal solution (x, y) = (1.0, 2.0)
```

## **User-cuts**

An example using a user-cut callback.

```
function example_user_cut_constraint()
 Random.seed!(1)
 N = 30
 item_weights, item_values = rand(N), rand(N)
 model = Model(Gurobi.Optimizer)
 set silent(model)
 # Turn off "Cuts" parameter so that our new one must be called. In real
 # models, you should leave "Cuts" turned on.
 set_attribute(model, "Cuts", 0)
 @variable(model, x[1:N], Bin)
 @constraint(model, sum(item_weights[i] * x[i] for i in 1:N) <= 10)</pre>
 @objective(model, Max, sum(item_values[i] * x[i] for i in 1:N))
 callback_called = false
 function my_callback_function(cb_data)
 callback_called = true
 x_vals = callback_value.(Ref(cb_data), x)
 accumulated = sum(item_weights[i] for i in 1:N if x_vals[i] > 1e-4)
 println("Called with accumulated = $(accumulated)")
 n_terms = sum(1 for i in 1:N if x_vals[i] > 1e-4)
 if accumulated > 10
 con = @build_constraint(
 sum(x[i] for i in 1:N if x_vals[i] > 0.5) <= n_terms - 1</pre>
)
 println("Adding $(con)")
 MOI.submit(model, MOI.UserCut(cb_data), con)
 end
 end
 set_attribute(model, MOI.UserCutCallback(), my_callback_function)
 optimize!(model)
 assert_is_solved_and_feasible(model)
```

Test.@test callback\_called @show callback\_called return

example\_user\_cut\_constraint()

end

```
Set parameter WLSAccessID
Set parameter WLSSecret
Set parameter LicenseID to value 722777
WLS license 722777 - registered to JuMP Development
Called with accumulated = 10.37975831721494
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[3] + x[4] + x[
 \rightarrow x[5] + x[7] + x[8] + x[9] + x[10] + x[11] + x[12] + x[13] + x[14] + x[16] + x[17] + x[18] + x[18]
\rightarrow x[20] + x[22] + x[23] + x[25] + x[26] + x[28] + x[29] + x[30],
 \rightarrow MathOptInterface.LessThan{Float64}(23.0))
Called with accumulated = 10.37975831721494
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[3] + x[4] + x[
\rightarrow x[5] + x[7] + x[8] + x[9] + x[10] + x[12] + x[13] + x[14] + x[16] + x[17] + x[18] + x[20] + x[20]
\hookrightarrow x[22] + x[23] + x[25] + x[26] + x[28] + x[29] + x[30],
 \rightarrow MathOptInterface.LessThan{Float64}(23.0))
Called with accumulated = 10.37975831721494
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[3] + x[4] + x[4] + x[2] + x[3] + x[4] + x[
 \ \, \hookrightarrow \ \, x[5] \, + \, x[7] \, + \, x[8] \, + \, x[9] \, + \, x[10] \, + \, x[11] \, + \, x[12] \, + \, x[13] \, + \, x[14] \, + \, x[16] \, + \, x[17] \, + \, x[18]
\hookrightarrow x[20] + x[22] + x[23] + x[25] + x[26] + x[29] + x[30],
\rightarrow MathOptInterface.LessThan{Float64}(23.0))
Called with accumulated = 10.37975831721494
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[3] + x[4] +
\hookrightarrow x[22] + x[23] + x[25] + x[26] + x[28] + x[29] + x[30],
\hookrightarrow MathOptInterface.LessThan{Float64}(23.0))
Called with accumulated = 10.37975831721494
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[3] + x[4] + x[
\rightarrow x[5] + x[7] + x[8] + x[9] + x[10] + x[12] + x[13] + x[14] + x[16] + x[17] + x[18] + x[20] + x[20]
\rightarrow x[22] + x[23] + x[25] + x[26] + x[28] + x[29] + x[30],
 \rightarrow MathOptInterface.LessThan{Float64}(23.0))
Called with accumulated = 10.37975831721494
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[3] + x[4] + x[
 \rightarrow x[5] + x[7] + x[8] + x[9] + x[10] + x[12] + x[13] + x[14] + x[16] + x[17] + x[18] + x[20] + x[20]
\hookrightarrow x[22] + x[23] + x[25] + x[26] + x[28] + x[29] + x[30],
 \rightarrow MathOptInterface.LessThan{Float64}(23.0))
Called with accumulated = 10.585271197221452
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[3] + x[4] + x[
 \ \, \hookrightarrow \ \, x[5] \, + \, x[7] \, + \, x[8] \, + \, x[9] \, + \, x[10] \, + \, x[11] \, + \, x[12] \, + \, x[13] \, + \, x[14] \, + \, x[16] \, + \, x[17] \, + \, x[18] \, + \, x[16] \, + \, x[17] \, + \, x[18] \, + \, x[16] \, + \, x[17] \, + \, x[18]
\hookrightarrow x[20] + x[22] + x[23] + x[25] + x[26] + x[29] + x[30],
 \rightarrow MathOptInterface.LessThan{Float64}(23.0))
Called with accumulated = 10.37975831721494
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[3] + x[4] + x
\rightarrow x[5] + x[7] + x[8] + x[9] + x[10] + x[12] + x[13] + x[14] + x[16] + x[17] + x[18] + x[20] + x[20]
\hookrightarrow \ x[22] \, + \, x[23] \, + \, x[25] \, + \, x[26] \, + \, x[28] \, + \, x[29] \, + \, x[30] \, ,
\hookrightarrow MathOptInterface.LessThan{Float64}(23.0))
callback_called = true
```

## **Heuristic solutions**

An example using a heuristic solution callback.

```
function example_heuristic_solution()
 Random.seed!(1)
 N = 30
 item_weights, item_values = rand(N), rand(N)
 model = Model(Gurobi.Optimizer)
 set silent(model)
 # Turn off "Heuristics" parameter so that our new one must be called. In
 # real models, you should leave "Heuristics" turned on.
 set_attribute(model, "Heuristics", 0)
 @variable(model, x[1:N], Bin)
 @constraint(model, sum(item_weights[i] * x[i] for i in 1:N) <= 10)</pre>
 @objective(model, Max, sum(item_values[i] * x[i] for i in 1:N))
 callback called = false
 function my callback function(cb data)
 callback_called = true
 x_vals = callback_value.(Ref(cb_data), x)
 ret =
 MOI.submit(model, MOI.HeuristicSolution(cb_data), x, floor.(x_vals))
 println("Heuristic solution status = $(ret)")
 Test.@test ret in (
 MOI.HEURISTIC_SOLUTION_ACCEPTED,
 MOI.HEURISTIC_SOLUTION_REJECTED,
)
 end
 set_attribute(model, MOI.HeuristicCallback(), my_callback_function)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 Test.@test callback_called
 return
end
```

```
example_heuristic_solution()
```

Set parameter WLSAccessID Set parameter WLSSecret Set parameter LicenseID to value 722777 WLS license 722777 - registered to JuMP Development Heuristic solution status = HEURISTIC\_SOLUTION\_ACCEPTED Heuristic solution status = HEURISTIC\_SOLUTION\_REJECTED

#### Gurobi solver-dependent callback

An example using Gurobi's solver-dependent callback.

```
function example_solver_dependent_callback()
model = direct_model(Gurobi.Optimizer())
@variable(model, 0 <= x <= 2.5, Int)
@variable(model, 0 <= y <= 2.5, Int)</pre>
```

```
@objective(model, Max, y)
cb_calls = Cint[]
function my_callback_function(cb_data, cb_where::Cint)
 # You can reference variables outside the function as normal
 push!(cb_calls, cb_where)
 # You can select where the callback is run
 if cb_where == Gurobi.GRB_CB_MIPNODE
 # You can query a callback attribute using GRBcbget
 resultP = Ref{Cint}()
 Gurobi.GRBcbget(
 cb data,
 cb_where,
 Gurobi.GRB_CB_MIPNODE_STATUS,
 resultP,
)
 if resultP[] != Gurobi.GRB_OPTIMAL
 return # Solution is something other than optimal.
 end
 elseif cb_where != Gurobi.GRB_CB_MIPSOL
 return
 end
 # Before querying `callback_value`, you must call:
 Gurobi.load_callback_variable_primal(cb_data, cb_where)
 x_val = callback value(cb_data, x)
 y val = callback value(cb data, y)
 # You can submit solver-independent MathOptInterface attributes such as
 # lazy constraints, user-cuts, and heuristic solutions.
 if y_val - x_val > 1 + 1e-6
 con = @build_constraint(y - x <= 1)</pre>
 MOI.submit(model, MOI.LazyConstraint(cb_data), con)
 elseif y_val + x_val > 3 + 1e-6
 con = @build_constraint(y + x <= 3)</pre>
 MOI.submit(model, MOI.LazyConstraint(cb_data), con)
 end
 # You can terminate the callback as follows:
 Gurobi.GRBterminate(backend(model))
 return
end
You _must_ set this parameter if using lazy constraints.
set_attribute(model, "LazyConstraints", 1)
set_attribute(model, Gurobi.CallbackFunction(), my_callback_function)
optimize!(model)
Test.@test termination_status(model) == MOI.INTERRUPTED
return
```

example\_solver\_dependent\_callback()

end

```
Set parameter WLSAccessID
Set parameter WLSSecret
Set parameter LicenseID to value 722777
WLS license 722777 - registered to JuMP Development
Set parameter LazyConstraints to value 1
Gurobi Optimizer version 12.0.2 build v12.0.2rc0 (linux64 - "Ubuntu 24.04.2 LTS")
```

```
CPU model: AMD EPYC 7763 64-Core Processor, instruction set [SSE2|AVX|AVX2]
Thread count: 2 physical cores, 4 logical processors, using up to 4 threads
Non-default parameters:
LazyConstraints 1
WLS license 722777 - registered to JuMP Development
Optimize a model with 0 rows, 2 columns and 0 nonzeros
Model fingerprint: 0x1cb4e750
Variable types: 0 continuous, 2 integer (0 binary)
Coefficient statistics:
 Matrix range [0e+00, 0e+00]
 Objective range [1e+00, 1e+00]

 Bounds range
 [2e+00, 2e+00]

 RHS range
 [0e+00, 0e+00]

Presolve time: 0.00s
Explored 0 nodes (0 simplex iterations) in 0.00 seconds (0.00 work units)
Thread count was 1 (of 4 available processors)
Solution count 0
Solve interrupted
Best objective -, best bound -, gap -
User-callback calls 31, time in user-callback 0.03 sec
```

## 6.21 Sensitivity analysis of a linear program

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial explains how to use the lp\_sensitivity\_report function to create sensitivity reports like those that are produced by the Excel Solver. This is most often used in introductory classes to linear programming.

In brief, sensitivity analysis of a linear program is about asking two questions:

- 1. Given an optimal solution, how much can the objective coefficients change by before a different solution becomes optimal?
- 2. Given an optimal solution, how much can the right-hand side of a linear constraint change by before a different solution becomes optimal?

JuMP provides a function, lp\_sensitivity\_report, to help us compute these values, but this tutorial extends that to create more informative tables in the form of a DataFrame.

#### Setup

This tutorial uses the following packages:

using JuMP import HiGHS import DataFrames as well as this small linear program:

```
model = Model(HiGHS.Optimizer)
@variable(model, x >= 0)
@variable(model, 0 <= y <= 3)
@variable(model, z <= 1)
@objective(model, Min, 12x + 20y - z)
@constraint(model, c1, 6x + 8y >= 100)
@constraint(model, c2, 7x + 12y >= 120)
@constraint(model, c3, x + y <= 20)
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model; verbose = true)</pre>
```

```
solution_summary(; result = 1, verbose = true)
F solver_name : HiGHS
- Termination
| + result_count : 1
| + raw_status : kHighsModelStatusOptimal
| L objective_bound : 2.04000e+02
Folution (result = 1)

 | + primal_status
 : FEASIBLE_POINT

 | + dual_status
 : FEASIBLE_POINT

 | + objective_value
 : 2.04000e+02

| | dual_objective_value : 2.04000e+02
| | value
| | | y : 1.25000e+00
| | ^L z : 1.00000e+00
| ^L dual
| | c1 : 2.50000e-01
| | c2 : 1.50000e+00
 ^L c3 : 0.00000e+00
L Work counters
 + solve_time (sec) : 2.57969e-04
 simplex_iterations : 2
 barrier_iterations : 0
 L node_count
 : -1
```

Can you identify:

- The objective coefficient of each variable?
- The right-hand side of each constraint?
- The optimal primal and dual solutions?

## **Sensitivity reports**

Now let's call lp\_sensitivity\_report:

report = lp\_sensitivity\_report(model)

It returns a SensitivityReport object, which maps:

- Every variable reference to a tuple (d\_lo, d\_hi)::Tuple{Float64, Float64}, explaining how much the
  objective coefficient of the corresponding variable can change by, such that the original basis remains
  optimal.
- Every constraint reference to a tuple (d\_lo, d\_hi)::Tuple{Float64, Float64}, explaining how much the right-hand side of the corresponding constraint can change by, such that the basis remains optimal.

Both tuples are relative, rather than absolute. So, given an objective coefficient of 1.0 and a tuple (-0.5, 0.5), the objective coefficient can range between 1.0 - 0.5 an 1.0 + 0.5.

For example:

report[x]

(-0.333333333333333, 3.0)

indicates that the objective coefficient on x, that is, 12, can decrease by -0.333 or increase by 3.0 and the primal solution (15, 1.25) will remain optimal. In addition:

report[c1]

(-4.0, 2.857142857142857)

means that the right-hand side of the c1 constraint (100), can decrease by 4 units, or increase by 2.85 units, and the primal solution (15, 1.25) will remain optimal.

#### Variable sensitivity

By themselves, the tuples aren't informative. Let's put them in context by collating a range of other information about a variable:

```
function variable_report(xi)
 return (
 name = name(xi),
 lower_bound = has_lower_bound(xi) ? lower_bound(xi) : -Inf,
 value = value(xi),
 upper_bound = has_upper_bound(xi) ? upper_bound(xi) : Inf,
 reduced_cost = reduced_cost(xi),
 obj_coefficient = coefficient(objective_function(model), xi),
 allowed_decrease = report[xi][1],
 allowed_increase = report[xi][2],
)
end
```

variable\_report (generic function with 1 method)

Calling our function on x:

x\_report = variable\_report(x)

That's a bit hard to read, so let's call this on every variable in the model and put things into a DataFrame:

```
variable_df =
DataFrames.DataFrame(variable_report(xi) for xi in all_variables(model))
```

	name	lower_bound	value	upper_bound	reduced_cost	obj_coefficient	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	х	0.0	15.0	Inf	0.0	12.0	-0.333333	3.0
2	У	0.0	1.25	3.0	0.0	20.0	-4.0	0.571429
3	z	-Inf	1.0	1.0	-1.0	-1.0	-Inf	1.0

## **Constraint sensitivity**

We can do something similar with constraints:

```
function constraint_report(c::ConstraintRef)
return (
 name = name(c),
 value = value(c),
 rhs = normalized_rhs(c),
 slack = normalized_rhs(c) - value(c),
 shadow_price = shadow_price(c),
 allowed_decrease = report[c][1],
 allowed_increase = report[c][2],
```

) end

cl\_report = constraint\_report(cl)

```
(name = "c1", value = 100.0, rhs = 100.0, slack = 0.0, shadow_price = -0.25, allowed_decrease =

→ -4.0, allowed_increase = 2.857142857142857)
```

That's a bit hard to read, so let's call this on every variable in the model and put things into a DataFrame:

```
constraint_df = DataFrames.DataFrame(
 constraint_report(ci) for (F, S) in list_of_constraint_types(model) for
 ci in all_constraints(model, F, S) if F == AffExpr
)
```

	name	value	rhs	slack	shadow_price	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64
1	c1	100.0	100.0	0.0	-0.25	-4.0	2.85714
2	c2	120.0	120.0	0.0	-1.5	-3.33333	4.66667
3	c3	16.25	20.0	3.75	0.0	-3.75	Inf

#### Analysis questions

Now we can use these dataframes to ask questions of the solution.

For example, we can find basic variables by looking for variables with a reduced cost of 0:

basic = filter(row -> iszero(row.reduced\_cost), variable\_df)

	name	lower_bound	value	upper_bound	reduced_cost	obj_coefficient	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	х	0.0	15.0	Inf	0.0	12.0	-0.333333	3.0
2	У	0.0	1.25	3.0	0.0	20.0	-4.0	0.571429

and non-basic variables by looking for non-zero reduced costs:

non\_basic = filter(row -> !iszero(row.reduced\_cost), variable\_df)

	name	lower_bound	value	upper_bound	reduced_cost	obj_coefficient	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	z	-Inf	1.0	1.0	-1.0	-1.0	-Inf	1.0

we can also find constraints that are binding by looking for zero slacks:

binding = filter(row -> iszero(row.slack), constraint\_df)

	name	value	rhs	slack	shadow_price	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64
1	c1	100.0	100.0	0.0	-0.25	-4.0	2.85714
2	c2	120.0	120.0	0.0	-1.5	-3.33333	4.66667

or non-zero shadow prices:

binding2 = filter(row -> !iszero(row.shadow\_price), constraint\_df)

	name	value	rhs	slack	shadow_price	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64
1	c1	100.0	100.0	0.0	-0.25	-4.0	2.85714
2	c2	120.0	120.0	0.0	-1.5	-3.33333	4.66667

## 6.22 Basis matrices

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial explains how to query the basis of a linear program.

#### Setup

This tutorial uses the following packages:

using JuMP import HiGHS

#### Standard form example

Consider the following example, which is from the Wikipedia article on Basic feasible solutions:

 $\begin{array}{l} \max \ 0 \\ \text{s.t. } 1x_1 + 5x_2 + 3x_3 + 4x_4 + 6x_5 = 14 \\ 0x_1 + 1x_2 + 3x_3 + 5x_4 + 6x_5 = 7 \\ x_i \geq 0, \forall i = 1, \dots, 5. \end{array}$ 

The A matrix is:

A = [1 5 3 4 6; 0 1 3 5 6]

2×5 Matrix{Int64}: 1 5 3 4 6 0 1 3 5 6 and the right-hand side b vector is:

b = [14, 7]

```
2-element Vector{Int64}:
14
7
```

We can create and optimize the problem in the standard form:

```
n = size(A, 2)
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:n] >= 0)
@constraint(model, A * x == b)
optimize!(model)
assert_is_solved_and_feasible(model)
```

This has a solution:

value.(x)

```
5-element Vector{Float64}:
0.0
2.0
0.0
1.0
0.0
```

Query the basis status of a variable using MOI.VariableBasisStatus:

get\_attribute(x[1], MOI.VariableBasisStatus())

NONBASIC\_AT\_LOWER::BasisStatusCode = 2

the result is a MOI.BasisStatusCode. Query all of the basis statuses with the broadcast get\_attribute.(:

get\_attribute.(x, MOI.VariableBasisStatus())

```
5-element Vector{MathOptInterface.BasisStatusCode}:
NONBASIC_AT_LOWER::BasisStatusCode = 2
BASIC::BasisStatusCode = 0
NONBASIC_AT_LOWER::BasisStatusCode = 2
BASIC::BasisStatusCode = 0
NONBASIC_AT_LOWER::BasisStatusCode = 2
```

For this problem, the values are either MOI.NONBASIC\_AT\_LOWER or MOI.BASIC. All of the MOI.NONBASIC\_AT\_LOWER variables have a value at their lower bound. The MOI.BASIC variables correspond to the columns of the optimal basis.

Get the columns using:

indices = get\_attribute.(x, MOI.VariableBasisStatus()) .== MOI.BASIC

```
5-element BitVector:
0
1
0
1
0
```

Filter the basis matrix from A:

```
B = A[:, indices]
```

```
2×2 Matrix{Int64}:
 5 4
 1 5
```

Since the basis matrix is non-singular, solving the system Bx = b must yield the optimal primal solution of the basic variables:

 $\mathsf{B}\,\setminus\,\mathsf{b}$ 

2-element Vector{Float64}: 2.0 0.999999999999999998

value.(x[indices])

```
2-element Vector{Float64}:
2.0
1.0
```

#### A more complicated example

Often, you may want to work with the basis of a model that is not in a nice standard form. For example:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x >= 0)
@variable(model, 0 <= y <= 3)
@variable(model, z <= 1)
@objective(model, Min, 12x + 20y - z)
@constraint(model, c1, 6x + 8y >= 100)
@constraint(model, c2, 7x + 12y >= 120)
@constraint(model, c3, x + y <= 20)
optimize!(model)
assert_is_solved_and_feasible(model)</pre>
```

A common way to query the basis status of every variable is:

```
v_basis = Dict(
 xi => get_attribute(xi, MOI.VariableBasisStatus()) for
 xi in all_variables(model)
)
```

```
Dict{VariableRef, MathOptInterface.BasisStatusCode} with 3 entries:
 y => BASIC
 x => BASIC
 z => NONBASIC_AT_UPPER
```

Despite the model having three constraints, there are only two basic variables. Since the basis matrix must be square, where is the other basic variable?

The answer is that solvers will reformulate inequality constraints:

$$Ax \leq b$$

into the system:

Ax + Is = b

Thus, for every inequality constraint there is a slack variable s.

Query the basis status of the slack variables associated with a constraint using MOI. ConstraintBasisStatus:

```
c_basis = Dict(
 ci => get_attribute(ci, MOI.ConstraintBasisStatus()) for ci in
 all_constraints(model; include_variable_in_set_constraints = false)
)
```

```
Dict{ConstraintRef{Model, C, ScalarShape} where C, MathOptInterface.BasisStatusCode} with 3

\hookrightarrow entries:

c3 : x + y \leq 20 => BASIC

c2 : 7 x + 12 y \geq 120 => NONBASIC

c1 : 6 x + 8 y \geq 100 => NONBASIC
```

Thus, the basis is formed by x, y, and the slack associated with c3.

A simple way to get the A matrix of an unstructured linear program is with lp\_matrix\_data:

```
matrix = lp_matrix_data(model)
matrix.A
```

```
3×3 SparseArrays.SparseMatrixCSC{Float64, Int64} with 6 stored entries:
6.0 8.0 ·
7.0 12.0 ·
1.0 1.0 ·
```

You can check the permutation of the rows and columns using

```
matrix.variables
```

```
3-element Vector{VariableRef}:
 x
 y
 z
```

and

matrix.affine\_constraints

```
3-element Vector{ConstraintRef}:

c1 : 6 x + 8 y ≥ 100

c2 : 7 x + 12 y ≥ 120

c3 : x + y ≤ 20
```

We can construct the slack column associated with c3 as:

```
s_column = zeros(size(matrix.A, 1))
s_column[3] = 1.0
```

## 1.0

The full basis matrix is therefore:

B = hcat(matrix.A[:, [1, 2]], s\_column)

```
3×3 SparseArrays.SparseMatrixCSC{Float64, Int64} with 7 stored entries:
6.0 8.0 ·
7.0 12.0 ·
1.0 1.0 1.0
```

lp\_matrix\_data returns separate vectors for the lower and upper row bounds. Convert to a single right-hand side vector by taking the finite elements:

b = ifelse.(isfinite.(matrix.b\_lower), matrix.b\_lower, matrix.b\_upper)

3-element Vector{Float64}: 100.0 120.0 20.0

Solving the Basis system as before yields:

B∖b

```
3-element Vector{Float64}:
14.999999999999995
1.25000000000004
3.75
```

which is the value of x, y, and the slack associated with c3.

## Identifying degenerate variables

Another common task is identifying degenerate variables. A degenerate variable is a basic variable that has an optimal value at its lower or upper bound.

Here is a function that computes whether a variable is degenerate:

is\_degenerate (generic function with 1 method)

A simple example of a linear program with a degenerate solution is:

```
A, b, c = [1 1; 0 1], [1, 1], [1, 1]
model = Model(HiGHS.Optimizer);
set_silent(model)
@variable(model, x[1:2] >= 0)
@objective(model, Min, c' * x)
@constraint(model, A * x == b)
optimize!(model)
degenerate_variables = filter(is_degenerate, all_variables(model))
```

1-element Vector{VariableRef}:
 x[1]

The solution is degenerate because:

value(x[1])

-0.0

and

```
get_attribute(x[1], MOI.VariableBasisStatus())
```

BASIC::BasisStatusCode = 0

## 6.23 Computing the duals of a mixed-integer program

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial explains how to compute the duals of a mixed-integer linear program by fixing the discrete variables to their optimal solution and resolving as a linear program.

This tutorial uses the following packages:

using JuMP import HiGHS

## The model

Our example model is the unit commitment example from Unit commitment. The details are unimportant, other than to note that there are two types of continuous variables, g and w, representing the quantity of generation from thermal and wind plants, and a discrete variable dispatch, which is 1 if plant i is operating, and  $\theta$  if not.

We are interested in the "dual" of the power\_balance constraint, because it represents the marginal price of electricity that consumers should pay for their consumption.

```
generators = [
 (min = 0.0, max = 1000.0, fixed_cost = 1000.0, variable_cost = 50.0),
 (min = 300.0, max = 1000.0, fixed_cost = 0.0, variable_cost = 100.0),
]
N = length(generators)
model = Model(HiGHS.Optimizer)
set silent(model)
@variables(model, begin
 generators[i].min <= g[i = 1:N] <= generators[i].max</pre>
 0 <= w <= 200
 dispatch[i = 1:N], Bin
end)
@constraints(model, begin
 power_balance, sum(g[i] for i in 1:N) + w == 1500
 [i = 1:N], g[i] <= generators[i].max * dispatch[i]</pre>
 [i = 1:N], g[i] >= generators[i].min * dispatch[i]
end)
@objective(
 model,
 Min,
 sum(
 generators[i].fixed_cost * dispatch[i] +
 generators[i].variable_cost * g[i] for i in 1:N
)
)
print(model)
```

```
Min 1000 dispatch[1] + 50 g[1] + 100 g[2]

Subject to

power_balance : g[1] + g[2] + w = 1500

g[1] \ge 0

g[2] - 300 dispatch[2] \ge 0

g[1] - 1000 dispatch[1] \le 0

g[2] - 1000 dispatch[2] \le 0

g[1] \ge 0

g[2] \ge 300

w \ge 0
```

g[1] ≤ 1000 g[2] ≤ 1000 w ≤ 200 dispatch[1] binary dispatch[2] binary

## Manually fix the variables

If we optimize this model, we obtain a dual\_status of NO\_SOLUTION:

optimize!(model)
assert\_is\_solved\_and\_feasible(model)
dual\_status(model)

NO\_SOLUTION::ResultStatusCode = 0

This is because HiGHS cannot compute the duals of a mixed-integer program. We can work around this problem by fixing the integer variables to their optimal solution, relaxing integrality, and re-solving as a linear program.

```
discrete_values = value.(dispatch)
fix.(dispatch, discrete_values; force = true)
unset_binary.(dispatch)
print(model)
```

```
Min 1000 dispatch[1] + 50 g[1] + 100 g[2]

Subject to

power_balance : g[1] + g[2] + w = 1500

g[1] \geq 0

g[2] - 300 dispatch[2] \geq 0

g[1] - 1000 dispatch[1] \leq 0

g[2] - 1000 dispatch[2] \leq 0

dispatch[1] = 1

dispatch[2] = 1

g[1] \geq 0

g[2] \geq 300

w \geq 0

g[1] \leq 1000

g[2] \leq 1000

w \leq 200
```

Now if we re-solve the problem, we obtain a FEASIBLE\_POINT for the dual:

```
optimize!(model)
assert_is_solved_and_feasible(model)
dual_status(model)
```

FEASIBLE\_POINT::ResultStatusCode = 1

and a marginal price of electricity of \$100/MWh:

dual(power\_balance)

100.0

To reset the problem back to a mixed-integer linear program, we need to unfix and call set\_binary:

unfix.(dispatch)
set\_binary.(dispatch)
print(model)

```
Min 1000 dispatch[1] + 50 g[1] + 100 g[2]
Subject to
 power_balance : g[1] + g[2] + w = 1500
 g[1] ≥ 0
 g[2] - 300 dispatch[2] ≥ 0
 g[1] - 1000 dispatch[1] ≤ 0
 g[2] - 1000 dispatch[2] ≤ 0
 g[1] ≥ 0
 g[2] ≥ 300
 w ≥ 0
 g[1] ≤ 1000
 g[2] ≤ 1000
 w ≤ 200
 dispatch[1] binary
 dispatch[2] binary
```

## **Use** fix\_discrete\_variables

Manually choosing the variables to relax and fix works for our small example, but it becomes more difficult in problems with a larger number of binary and integer variables. To automate the process we just did manually, JuMP provides the fix\_discrete\_variables function:

```
optimize!(model)
assert_is_solved_and_feasible(model)
dual_status(model)
```

NO\_SOLUTION::ResultStatusCode = 0

```
undo = fix_discrete_variables(model);
```

Here undo is a function that, when called with no arguments, returns the model to the original mixed-integer formulation.

After calling fix\_discrete\_variables, you can set a new solver with set\_optimizer if your mixed-integer solver does not support computing a dual solution.

print(model)

Тір

```
Min 1000 dispatch[1] + 50 g[1] + 100 g[2]
Subject to
 power_balance : g[1] + g[2] + w = 1500
 g[1] ≥ 0
 g[2] - 300 dispatch[2] ≥ 0
 g[1] - 1000 dispatch[1] ≤ 0
 g[2] - 1000 dispatch[2] ≤ 0
 dispatch[1] = 1
 dispatch[2] = 1
 g[1] ≥ 0
 g[2] ≥ 300
 w ≥ 0
 g[1] ≤ 1000
 g[2] ≤ 1000
 w ≤ 200
```

optimize!(model)
assert\_is\_solved\_and\_feasible(model)
dual\_status(model)

FEASIBLE\_POINT::ResultStatusCode = 1

dual(power\_balance)

100.0

Finally, call undo to revert the reformulation

CHAPTER 6. LINEAR PROGRAMS

undo() print(model)

```
Min 1000 dispatch[1] + 50 g[1] + 100 g[2]
Subject to
power_balance : g[1] + g[2] + w = 1500
g[1] ≥ 0
g[2] - 300 dispatch[2] ≥ 0
g[1] - 1000 dispatch[1] ≤ 0
g[2] - 1000 dispatch[2] ≤ 0
g[1] ≥ 0
g[2] ≥ 300
w ≥ 0
g[1] ≤ 1000
g[2] ≤ 1000
w ≤ 200
dispatch[1] binary
dispatch[2] binary
```

## 6.24 Finding multiple feasible solutions

This tutorial was generated using Literate.jl. Download the source as a .jl file.

Author: James Foster (@jd-foster)

This tutorial demonstrates how to formulate and solve a combinatorial problem with multiple feasible solutions. In fact, we will see how to find *all* feasible solutions to our problem. We will also see how to enforce an "alldifferent" constraint on a set of integer variables.

## **Required packages**

This tutorial uses the following packages:

using JuMP
import Gurobi
import Test

#### Warning

This tutorial uses Gurobi.jl as the solver because it supports returning multiple feasible solutions, something that open-source MIP solvers such as HiGHS do not currently support. Gurobi is a commercial solver and requires a paid license. However, there are free licenses available for academic and student users. See Gurobi.jl for more details.

#### Symmetric number squares

Symmetric number squares and their sums often arise in recreational mathematics. Here are a few examples:

	1529	2318	5219
	5837	3790	2384
+	2340	+ 1 9 5 6	+ 1 8 6 7
=	9706	= 8 0 6 4	= 9 4 7 0

Notice how all the digits 0 to 9 are used at least once, the first three rows sum to the last row, the columns in each are the same as the corresponding rows (forming a symmetric matrix), and 0 does not appear in the first column.

We will answer the question: how many such squares are there?

#### JuMP model

We now encode the symmetric number square as a JuMP model. First, we need a symmetric matrix of decision variables between 0 and 9 to represent each number:

```
n = 4
optimizer = Gurobi.Optimizer
model = Model(optimizer)
set_silent(model)
@variable(model, 0 <= x_digits[row in 1:n, col in 1:n] <= 9, Int, Symmetric)</pre>
```

```
4×4 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 x_digits[1,1] x_digits[1,2] x_digits[1,3] x_digits[1,4]
 x_digits[1,2] x_digits[2,2] x_digits[2,3] x_digits[2,4]
 x_digits[1,3] x_digits[2,3] x_digits[3,3] x_digits[3,4]
 x_digits[1,4] x_digits[2,4] x_digits[3,4] x_digits[4,4]
```

We modify the lower bound to ensure that the first column cannot contain 0:

```
set_lower_bound.(x_digits[:, 1], 1)
```

4-element Vector{Nothing}:
 nothing
 nothing
 nothing
 nothing

Then, we need a constraint that the sum of the first three rows equals the last row:

```
@expression(model, x_base_10, x_digits * [1_000, 100, 10, 1]);
@constraint(model, sum(x_base_10[i] for i in 1:n-1) == x_base_10[n])
```

 $1000x\_digits_{1,1} + 1100x\_digits_{1,2} + 100x\_digits_{2,2} + 1010x\_digits_{1,3} + 110x\_digits_{2,3} + 10x\_digits_{3,3} - 999x\_digits_{1,4} - 99x\_digits_{1,4} - 90x\_digits_{1,4} - 90$ 

And we use MOI.AllDifferent to ensure that each digit is used exactly once in the upper triangle matrix of  $x_{digits}$ :

```
x_digits_upper = [x_digits[i, j] for j in 1:n for i in 1:j]
@constraint(model, x_digits_upper in MOI.AllDifferent(length(x_digits_upper)));
```

If we optimize this model, we find that Gurobi has returned one solution:

```
optimize!(model)
assert_is_solved_and_feasible(model)
Test.@test result_count(model) == 1
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
- solver_name
 : Gurobi
- Termination
| | termination_status : OPTIMAL
: 1
| - raw_status : Model was solved to optimality (subject to tolerances), and an optimal
\hookrightarrow solution is available.
| L objective_bound : 0.00000e+00
Solution (result = 1)
| + primal_status : FEASIBLE_POINT
| + dual_status : NO_SOLUTION
| + objective_value : 0.00000e+00
| L relative_gap : 0.00000e+00
^L Work counters
 solve_time (sec) : 6.02641e-02
 simplex_iterations : 1587
 barrier_iterations : 0
 L node_count
 : 255
```

To return multiple solutions, we need to set Gurobi-specific parameters to enable the solution pool. Moreover, there is a bug in Gurobi that means the solution pool is not activated if we have already solved the model once. To work around the bug, we need to reset the optimizer. If you turn the solution pool options on before the first solve you do not need to reset the optimizer.

```
set_optimizer(model, optimizer)
```

Set parameter WLSAccessID Set parameter WLSSecret Set parameter LicenseID to value 722777 WLS license 722777 - registered to JuMP Development

The first option turns on the exhaustive search mode for multiple solutions:

CHAPTER 6. LINEAR PROGRAMS

set\_attribute(model, "PoolSearchMode", 2)

The second option sets a limit for the number of solutions found:

set\_attribute(model, "PoolSolutions", 100)

Here the value 100 is an "arbitrary but large enough" whole number for our particular model (and in general will depend on the application).

We can then call optimize! and view the results.

optimize!(model)
assert\_is\_solved\_and\_feasible(model)
solution\_summary(model)

```
solution_summary(; result = 1, verbose = false)
├ solver_name : Gurobi
├ Termination
| + termination_status : OPTIMAL
| | result_count : 20
: Model was solved to optimality (subject to tolerances), and an optimal
\hookrightarrow solution is available.
| L objective_bound : 0.00000e+00
Solution (result = 1)
| + primal_status : FEASIBLE_POINT
| + dual_status : N0_SOLUTION
| + objective_value : 0.00000e+00
| L relative_gap : 0.00000e+00
^L Work counters
 solve_time (sec) : 3.73439e-01
 simplex_iterations : 19526
 barrier_iterations : 0
 L node count : 4661
```

Now Gurobi has found 20 solutions:

result\_count(model)

20

#### Viewing the Results

Access the various feasible solutions by using the value function with the result keyword:

```
solutions =
 [round.(Int, value.(x_digits; result = i)) for i in 1:result_count(model)];
```

Here we have converted the solution to an integer after rounding off very small numerical tolerances.

An example of one feasible solution is:

solutions[1]

4×4 Matrix{Int64}:
 1 5 2 9
5 8 3 7
2 3 4 0
9 7 0 6

and we can nicely print out all the feasible solutions with

```
function solution_string(x::Matrix)
 header = [" ", " ", "+", "="]
 return join([join(vcat(header[i], x[i, :]), " ") for i in 1:4], "\n")
end
for i in 1:result_count(model)
 println("Solution $i: \n", solution_string(solutions[i]), "\n")
end
```

Solution 1:						
	1	5	2	9		
	5	8	3	7		
+	2	3	4	0		
=	9	7	0	6		
_	_			_		
Sc	ι	it:	LOI	ו 2:		
	1	5	2	9		
	5	7	4	6		
+	2	4	0	8		
=	9	6	8	3		
So	οιι	ut:	Lor	ı 3:		
	1	3	2	7		
	3	6	5	4		
+	2	5	0	8		
=	7	4	8	9		
Sc	οιι	ut:	Lor	n 4:		
	2	3	1	7		
	3	5	6	4		
+	1	6	0	8		

Solution 5:
$\begin{array}{c} 1 & 9 & 6 & 7 \\ + & 4 & 6 & 3 & 5 \\ = & 8 & 7 & 5 & 0 \end{array}$
Solution 6: 3 2 1 7 2 9 4 5 + 1 4 0 6 = 7 5 6 8
Solution 7: 1 2 3 7 2 9 6 8 + 3 6 4 5 = 7 8 5 0
Solution 8: 2 3 1 8 3 7 9 0 + 1 9 5 6 = 8 0 6 4
Solution 9: 5 1 3 9 1 0 4 6 + 3 4 8 7 = 9 6 7 2
Solution 10: 5 2 1 9 2 6 8 7 + 1 8 3 4 = 9 7 4 0
Solution 11: 5 2 1 9 2 3 8 4 + 1 8 6 7
= 9 4 7 0 Solution 12: 2 1 6 9 1 3 0 5 + 6 0 7 4 = 9 5 4 8
Solution 13: 1 2 5 9 2 6 4 3 + 5 4 7 8

= 9 3 8 0

S	οιι	ut:	LOI	n :	14:		
	1	2	5	9			
	2	4	3	0			
+	5	3	8	7			
=	9	0	7	6			
So	oli	Jt:	ioi	n :	15:		
	2	1	4	8			
	1	5	6	3			
+	4	6	7	9			
=	8	3	9	0			
So	oli	ut:	LOI	ר ו	16:		
	5	2	1	9			
	2	7	4	3			
+	1	4	0	6			
=	9	3	6	8			
So	oli	ut:	LOI	n :	L7:		
	5	1	3	9			
	1	4	0	6			
+	3	0	8	2			
=	9	6	2	7			
So	olu	ut:	LOI	n :	18:		
	1	2	3	7			
	2	5	6	4			
+	3	6	8	9			
=	7	4	9	0			
So	olı	ut:	LOI	n :	L9:		
	1	4	2	8			
	4	7	5	6			
+	2	5	0	9			
=	8	6	9	3			
So	olı	ut:	LOI	n 2	20:		
	3	2	1	6			
	2	0	4	7			
+	1	4	9	5			
=	6	7	5	8			

The result is the full list of feasible solutions. So the answer to "how many such squares are there?" turns out to be 20.

## Chapter 7

# Nonlinear programs

## 7.1 Introduction

Nonlinear programs (NLPs) are a class of optimization problems in which some of the constraints or the objective function are nonlinear:

$$\min_{x \in \mathbb{R}^n} f_0(x) \tag{7.1}$$

$$\text{s.t.} l_j \le f_j(x) \le u_j \qquad \qquad j = 1 \dots m \tag{7.2}$$

$$l_i \le x_i \le u_i \qquad \qquad i = 1 \dots n. \tag{7.3}$$

Mixed-integer nonlinear linear programs (MINLPs) are extensions of nonlinear programs in which some (or all) of the decision variables take discrete values.

#### How to choose a solver

JuMP supports a range of nonlinear solvers; look for "NLP" in the list of Supported solvers. However, very few solvers support mixed-integer nonlinear linear programs. Solvers supporting discrete variables start with "(MI)" in the list of Supported solvers.

If the only nonlinearities in your model are quadratic terms (that is, multiplication between two decision variables), you can also use second-order cone solvers, which are indicated by "SOCP." In most cases, these solvers are restricted to convex quadratic problems and will error if you pass a nonconvex quadratic function; however, Gurobi has the ability to solve nonconvex quadratic terms.

#### How these tutorials are structured

Having a high-level overview of how this part of the documentation is structured will help you know where to look for certain things.

- The following tutorials are worked examples that present a problem in words, then formulate it in mathematics, and then solve it in JuMP. This usually involves some sort of visualization of the solution. Start here if you are new to JuMP.
  - Example: nonlinear optimal control of a rocket
  - Example: optimal control for a Space Shuttle reentry trajectory
  - Example: portfolio optimization

- The Computing Hessians is an advanced tutorial which explains how to compute the Hessian of the Lagrangian of a nonlinear program. This is useful only in particular cases.
- The remaining tutorials are less verbose and styled in the form of short code examples. These tutorials have less explanation, but may contain useful code snippets, particularly if they are similar to a problem you are trying to solve.

#### 7.2 Simple examples

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial is a collection of examples of small nonlinear programs.

## **Required packages**

This tutorial uses the following packages:

```
using JuMP
import Ipopt
import Random
import Statistics
import Test
```

#### The Rosenbrock function

A nonlinear example of the classical Rosenbrock function.

```
function example_rosenbrock()
 model = Model(Ipopt.Optimizer)
 set_silent(model)
 @variable(model, x)
 @objective(model, Min, (1 - x)^2 + 100 * (y - x^2)^2)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 Test.@test objective_value(model) ≈ 0.0 atol = 1e-10
 Test.@test value(x) ≈ 1.0
 Test.@test value(y) ≈ 1.0
 return
end
```

```
example_rosenbrock()
```

## The cinibeam problem

Based on an AMPL model by Hande Y. Benson

Copyright (C) 2001 Princeton University All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that the copyright notice and this permission notice appear in all supporting documentation.

#### Source:

H. Maurer and H.D. Mittelman, "The non-linear beam via optimal control with bound state variables," Optimal Control Applications and Methods 12, pp. 19-31, 1991.

```
function example_clnlbeam()
 N = 1000
 h = 1 / N
 alpha = 350
 model = Model(Ipopt.Optimizer)
 @variables(model, begin
 -1 <= t[1:(N+1)] <= 1
 -0.05 <= x[1:(N+1)] <= 0.05
 u[1:(N+1)]
 end)
 @objective(
 model.
 Min,
 sum(
 0.5 * h * (u[i+1]² + u[i]²) +
 0.5 * alpha * h * (cos(t[i+1]) + cos(t[i])) for i in 1:N
),
)
 @constraint(
 model,
 [i = 1:N],
 x[i+1] - x[i] - 0.5 * h * (sin(t[i+1]) + sin(t[i])) == 0,
)
 @constraint(
 model,
 [i = 1:N],
 t[i+1] - t[i] - 0.5 * h * u[i+1] - 0.5 * h * u[i] == 0,
)
 optimize!(model)
 println("""
 termination status = $(termination status(model))
 primal_status = $(primal_status(model))
 objective_value = $(objective_value(model))
 """)
 assert_is_solved_and_feasible(model)
 return
end
```

example\_clnlbeam()

```
variables with only upper bounds:
 0
 2000
Total number of equality constraints.....
Total number of inequality constraints.....
 0
 inequality constraints with only lower bounds:
 0
 inequality constraints with lower and upper bounds:
 0
 inequality constraints with only upper bounds:
 0
 inf_pr inf_du lg(mu) ||d|| lg(rg) alpha_du alpha_pr ls
iter
 objective
 0 3.5000000e+02 0.00e+00 0.00e+00 -1.0 0.00e+00 - 0.00e+00 0.00e+00
 0
 1 3.5000000e+02 0.00e+00 0.00e+00 -1.7 0.00e+00
 - 1.00e+00 1.00e+00 0
 2 3.5000000e+02 0.00e+00 0.00e+00 -3.8 0.00e+00 -2.0 1.00e+00 1.00e+00T 0
 3 3.5000000e+02 0.00e+00 0.00e+00 -5.7 0.00e+00 0.2 1.00e+00 1.00e+00T 0
 4 3.5000000e+02 0.00e+00 0.00e+00 -8.6 0.00e+00 -0.2 1.00e+00 1.00e+00T 0
Number of Iterations....: 4
 (scaled)
 (unscaled)
Objective..... 3.50000000000318e+02
 3.50000000000318e+02
0.0000000000000000e+00
0.0000000000000000e+00
Variable bound violation: 0.0000000000000000e+00
 0.00000000000000000e+00
Complementarity..... 2.5059035596802450e-09
 2.5059035596802450e-09
Overall NLP error.....: 2.5059035596802450e-09
 2.5059035596802450e-09
Number of objective function evaluations
 = 5
Number of objective gradient evaluations
 = 5
 = 5
Number of equality constraint evaluations
Number of inequality constraint evaluations
 = 0
Number of equality constraint Jacobian evaluations = 5
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations
 = 4
Total seconds in IPOPT
 = 0.032
EXIT: Optimal Solution Found.
termination status = LOCALLY SOLVED
primal_status = FEASIBLE POINT
objective value = 350.00000000032
```

#### Maximum likelihood estimation

This example uses nonlinear optimization to compute the maximum likelihood estimate (MLE) of the parameters of a normal distribution, a.k.a., the sample mean and variance.

```
function example_mle()
n = 1_000
Random.seed!(1234)
data = randn(n)
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, µ, start = 0.0)
@variable(model, σ >= 0.0, start = 1.0)
@objective(
```
```
model,
 Max,
 n / 2 * log(1 / (2 * \pi * \sigma^2)) -
 sum((data[i] - \mu)² for i in 1:n) / (2 * \sigma²)
)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 println("\mu = ", value(\mu))
 println("mean(data) = ", Statistics.mean(data))
 println("\sigma^2 = ", value(\sigma)^2)
 println("var(data) = ", Statistics.var(data))
 println("MLE objective = ", objective_value(model))
 Test.@test value(\mu) \approx Statistics.mean(data) atol = 1e-3
 Test.@test value(\sigma)^2 \approx Statistics.var(data) atol = 1e-2
 # You can even do constrained MLE!
 (constraint(model, \mu == \sigma^2))
 optimize!(model)
 assert_is_solved_and_feasible(model)
 Test.@test value(\mu) \approx value(\sigma)^2
 println()
 println("With constraint \mu == \sigma^2:")
 = ", value(\mu))
 println("µ
 println(<mark>"o^2</mark>
 = ", value(\sigma)^2)
 println("Constrained MLE objective = ", objective_value(model))
 return
end
```

```
example_mle()
```

## Quadratically constrained programs

A simple quadratically constrained program based on an example from Gurobi.

```
function example_qcp()
 model = Model(Ipopt.Optimizer)
 set_silent(model)
 @variable(model, x)
 @variable(model, y >= 0)
 @variable(model, z >= 0)
 @objective(model, Max, x)
 @constraint(model, x + y + z == 1)
```

```
@constraint(model, x * x + y * y - z * z <= 0)
@constraint(model, x * x - y * z <= 0)
optimize!(model)
assert_is_solved_and_feasible(model)
print(model)
println("Objective value: ", objective_value(model))
println("x = ", value(x))
println("y = ", value(y))
Test.@test objective_value(model) ≈ 0.32699 atol = 1e-5
Test.@test value(x) ≈ 0.32699 atol = 1e-5
Test.@test value(y) ≈ 0.25707 atol = 1e-5
return
end
```

example\_qcp()

```
Max x

Subject to

x + y + z = 1

x^2 + y^2 - z^2 \le 0

x^2 - y^*z \le 0

y \ge 0

z \ge 0

Objective value: 0.32699283491387243

x = 0.32699283491387243

y = 0.2570658388068964
```

## 7.3 User-defined operators with vector outputs

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to demonstrate how to write a user-defined operator with a vector-valued output.

# **Required packages**

This tutorial uses the following packages:

using JuMP
import Ipopt
import Test

# **Motivation**

A common situation is to have a user-defined operator like the following that returns multiple outputs (we define function\_calls to keep track of how many times we call this method):

```
function_calls = 0
function foo(x, y)
global function_calls += 1
```

```
common_term = x^2 + y^2
term_1 = sqrt(1 + common_term)
term_2 = common_term
return term_1, term_2
end
```

foo (generic function with 1 method)

For example, the first term might be used in the objective, and the second term might be used in a constraint, and often they share work that is expensive to evaluate.

This is a problem for JuMP, because it requires user-defined operators to return a single number. One option is to define two separate functions, the first returning the first argument, and the second returning the second argument.

foo\_1(x, y) = foo(x, y)[1]
foo\_2(x, y) = foo(x, y)[2]

foo\_2 (generic function with 1 method)

However, if the common term is expensive to compute, this approach is wasteful because it will evaluate the expensive term twice. Let's have a look at how many times we evaluate  $x^2 + y^2$  during a solve:

```
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, x[1:2] >= 0, start = 0.1)
@operator(model, op_foo_1, 2, foo_1)
@operator(model, op_foo_2, 2, foo_2)
@objective(model, Max, op_foo_1(x[1], x[2]))
@constraint(model, op_foo_2(x[1], x[2]) <= 2)
function_calls = 0
optimize!(model)
assert_is_solved_and_feasible(model)
Test.@test objective_value(model) ≈ √3 atol = 1e-4
Test.@test value.(x) ≈ [1.0, 1.0] atol = 1e-4
println("Naive approach: function_calls = $(function_calls)")</pre>
```

Naive approach: function calls = 44

### Memoization

An alternative approach is to use *memoization*, which uses a cache to store the result of function evaluations. We can write a memoization function as follows:

```
......
 memoize(foo::Function, n_outputs::Int)
Take a function `foo` and return a vector of length `n outputs`, where element
`i` is a function that returns the equivalent of `foo(x...)[i]`.
To avoid duplication of work, cache the most-recent evaluations of `foo`.
Because `foo_i` is auto-differentiated with ForwardDiff, our cache needs to
work when `x` is a `Float64` and a `ForwardDiff.Dual`.
.....
function memoize(foo::Function, n outputs::Int)
 last_x, last_f = nothing, nothing
 last_dx, last_dfdx = nothing, nothing
 function foo_i(i, x::T...) where {T<:Real}</pre>
 if T == Float64
 if x !== last_x
 last_x, last_f = x, foo(x...)
 end
 return last_f[i]::T
 else
 if x !== last_dx
 last_dx, last_dfdx = x, foo(x...)
 end
 return last_dfdx[i]::T
 end
 end
 return [(x...) -> foo_i(i, x...) for i in 1:n_outputs]
end
```

Main.memoize

Let's see how it works. First, construct the memoized versions of foo:

memoized\_foo = memoize(foo, 2)

```
2-element Vector{Main.var"#4#7"{Int64, Main.var"#foo_i#5"{typeof(Main.foo)}}:
#4 (generic function with 1 method)
#4 (generic function with 1 method)
```

Now try evaluating the first element of memoized\_foo.

```
function_calls = 0
memoized_foo[1](1.0, 1.0)
println("function_calls = ", function_calls)
```

```
function_calls = 1
```

As expected, this evaluated the function once. However, if we call the function again, we hit the cache instead of needing to re-compute foo and function\_calls is still 1!

```
memoized_foo[1](1.0, 1.0)
println("function_calls = ", function_calls)
```

```
function_calls = 1
```

Now let's see how this works during a real solve:

```
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, x[1:2] >= 0, start = 0.1)
@operator(model, op_foo_1, 2, memoized_foo[1])
@operator(model, op_foo_2, 2, memoized_foo[2])
@objective(model, Max, op_foo_1(x[1], x[2]))
@constraint(model, op_foo_2(x[1], x[2]) <= 2)</pre>
function calls = \Theta
optimize!(model)
assert_is_solved_and_feasible(model)
Test.@test objective_value(model) \approx \sqrt{3} atol = 1e-4
Test.@test value.(x) \approx [1.0, 1.0] atol = 1e-4
println("Memoized approach: function_calls = $(function_calls)")
```

Memoized approach: function\_calls = 22

Compared to the naive approach, the memoized approach requires half as many function evaluations.

## 7.4 Automatic differentiation of user-defined operators

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to demonstrate how to apply automatic differentiation to User-defined operators.

## Tip

This tutorial is for advanced users. As an alternative, consider using Function tracing instead of creating an operator, and if an operator is necessary, consider using the default of @operator(model, op\_f, N, f) instead of passing explicit Gradients and Hessians.

# **Required packages**

This tutorial uses the following packages:

```
using JuMP
import DifferentiationInterface
import Enzyme
import ForwardDiff
import Ipopt
import Test
```

# **Primal function**

As a simple example, we consider the Rosenbrock function:

f(x::T...) where  $\{T\} = (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2$ 

f (generic function with 1 method)

Here's the value at a random point:

x = rand(2)

2-element Vector{Float64}: 0.8809678954777748 0.5901641516460916

f(x...)

3.4715474597921627

## **Analytic derivative**

If expressions are simple enough, you can provide analytic functions for the gradient and Hessian.

# Gradient

The Rosenbrock function has the gradient vector:

```
function analytic_∇f(g::AbstractVector, x...)
g[1] = 400 * x[1]^3 - 400 * x[1] * x[2] + 2 * x[1] - 2
g[2] = 200 * (x[2] - x[1]^2)
return
end
```

analytic\_ $\nabla f$  (generic function with 1 method)

Let's evaluate it at the same vector x:

analytic\_g = zeros(2)
analytic\_Vf(analytic\_g, x...)
analytic\_g

2-element Vector{Float64}: 65.28490308207542 -37.18805624328958

### Hessian

The Hessian matrix is:

```
function analytic_\[\nu_\] \[\Left(H::AbstractMatrix, x...)
H[1, 1] = 1200 * x[1]^2 - 400 * x[2] + 2
H[1, 2] = -400 * x[1] <-- not needed because Hessian is symmetric
H[2, 1] = -400 * x[1]
H[2, 2] = 200.0
return
end</pre>
```

analytic\_ $\nabla^2 f$  (generic function with 1 method)

Note that because the Hessian is symmetric, JuMP requires that we fill in only the lower triangle.

```
analytic_H = zeros(2, 2) analytic_\nabla^2 f(analytic_H, x...) analytic_H
```

```
2×2 Matrix{Float64}:
697.26 0.0
-352.387 200.0
```

# JuMP example

Putting our analytic functions together, we get:

```
function analytic_rosenbrock()
 model = Model(Ipopt.Optimizer)
 set_silent(model)
 @variable(model, x[1:2])
 @operator(model, op_rosenbrock, 2, f, analytic_∇f, analytic_∇²f)
 @objective(model, Min, op_rosenbrock(x[1], x[2]))
 optimize!(model)
 assert_is_solved_and_feasible(model)
 return value.(x)
end
```

analytic\_rosenbrock()

2-element Vector{Float64}: 0.9999999999999655 0.9999999999999305

#### ForwardDiff

Instead of analytic functions, you can use ForwardDiff.jl to compute derivatives.

#### Info

If you do not specify a gradient or Hessian, JuMP will use ForwardDiff.jl to compute derivatives by default. We provide this section as a worked example of what is going on under the hood.

### **Pros and cons**

The main benefit of ForwardDiff is that it is simple, robust, and works with a broad range of Julia syntax.

The main downside is that f must be a function that accepts arguments of x::Real.... See Common mistakes when writing a user-defined operator for more details.

### Gradient

The gradient can be computed using ForwardDiff.gradient!. Note that ForwardDiff expects a single Vector{T} argument, but we receive x as a tuple, so we need  $y \rightarrow f(y...)$  and collect(x) to get things in the right format.

```
function fdiff_\[Vf(g::AbstractVector{T}, x::Vararg{T,N}) where {T,N}
ForwardDiff.gradient!(g, y -> f(y...), collect(x))
return
end
```

 $fdiff_\nabla f$  (generic function with 1 method)

Let's check that we find the analytic solution:

fdiff\_g = zeros(2)
fdiff\_⊽f(fdiff\_g, x...)
Test.@test ≈(analytic\_g, fdiff\_g)

#### Test Passed

### Hessian

The Hessian is a bit more complicated, but code to implement it is:

```
function fdiff_V²f(H::AbstractMatrix{T}, x::Vararg{T,N}) where {T,N}
 h = ForwardDiff.hessian(y -> f(y...), collect(x))
 for i in 1:N, j in 1:i
 H[i, j] = h[i, j]
 end
 return
end
```

fdiff\_ $\nabla^2 f$  (generic function with 1 method)

Let's check that we find the analytic solution:

fdiff\_H = zeros(2, 2)
fdiff\_∇²f(fdiff\_H, x...)
Test.@test ≈(analytic\_H, fdiff\_H)

Test Passed

### JuMP example

The code for computing the gradient and Hessian using ForwardDiff can be re-used for many operators. Thus, it is helpful to encapsulate it into the function:

```
"""
fdiff_derivatives(f::Function) -> Tuple{Function,Function}
Return a tuple of functions that evaluate the gradient and Hessian of `f` using
ForwardDiff.jl.
"""
function fdiff_derivatives(f::Function)
 function \forbol{f}(g::AbstractVector{T}, x::Vararg{T,N}) where {T,N}
 ForwardDiff.gradient!(g, y -> f(y...), collect(x))
 return
 end
```

```
function ∇²f(H::AbstractMatrix{T}, x::Vararg{T,N}) where {T,N}
 h = ForwardDiff.hessian(y -> f(y...), collect(x))
 for i in 1:N, j in 1:i
 H[i, j] = h[i, j]
 end
 return
end
return ∇f, ∇²f
end
```

Main.fdiff\_derivatives

Here's an example using fdiff\_derivatives:

```
function fdiff_rosenbrock()
 model = Model(Ipopt.Optimizer)
 set_silent(model)
 @variable(model, x[1:2])
 @operator(model, op_rosenbrock, 2, f, fdiff_derivatives(f)...)
 @objective(model, Min, op_rosenbrock(x[1], x[2]))
 optimize!(model)
 assert_is_solved_and_feasible(model)
 return value.(x)
end
```

```
fdiff_rosenbrock()
```

# Enzyme

Another library for automatic differentiation in Julia is Enzyme.jl.

## **Pros and cons**

The main benefit of Enzyme is that it can produce fast derivatives for functions with many input arguments.

The main downsides are that it may throw unusual errors if your code uses an unsupported feature of Julia and that it may have large compile times.

#### Warning

The JuMP developers cannot help you debug error messages related to Enzyme. If the operator works, it works. If not, we suggest you try a different automatic differentiation library. See juliadiff.org for details.

# Gradient

The gradient can be computed using Enzyme.autodiff with the Enzyme.Reverse mode. We need to wrap x in Enzyme.Active to indicate that we want to compute the derivatives with respect to these arguments.

```
function enzyme_Vf(g::AbstractVector{T}, x::Vararg{T,N}) where {T,N}
g .= Enzyme.autodiff(Enzyme.Reverse, f, Enzyme.Active.(x)...)[1]
return
end
```

 $enzyme_{\nabla f}$  (generic function with 1 method)

Let's check that we find the analytic solution:

```
enzyme_g = zeros(2)
enzyme_⊽f(enzyme_g, x...)
Test.@test ≈(analytic_g, enzyme_g)
```

Test Passed

### Hessian

We can compute the Hessian in Enzyme using forward-over-reverse automatic differentiation.

The code to implement the Hessian in Enzyme is complicated, so we will not explain it in detail; see the Enzyme documentation.

```
function enzyme_{\nabla^2}f(H::AbstractMatrix{T}, x::Vararg{T,N}) where {T,N}
 # direction(i) returns a tuple with a `1` in the `i`'th entry and `0`
 # otherwise
 direction(i) = ntuple(j -> Enzyme.Active(T(i == j)), N)
 # As the inner function, compute the gradient using Reverse mode
 \nabla f(x...) = Enzyme.autodiff(Enzyme.Reverse, f, Enzyme.Active, x...)[1]
 # For the outer autodiff, use Forward mode.
 hess = Enzyme.autodiff(
 Enzyme.Forward,
 ⊽f,
 # Compute multiple evaluations of Forward mode, each time using `x` but
 # initializing with a different direction.
 Enzyme.BatchDuplicated.(Enzyme.Active.(x), ntuple(direction, N))...,
)[1]
 # Unpack Enzyme's `hess` data structure into the matrix `H` expected by
 # JuMP.
 for j in 1:N, i in 1:j
 H[j, i] = hess[j][i]
 end
 return
end
```

enzyme\_ $\nabla^2 f$  (generic function with 1 method)

Let's check that we find the analytic solution:

enzyme\_H = zeros(2, 2) enzyme\_∇<sup>2</sup>f(enzyme\_H, x...) Test.@test ≈(analytic\_H, enzyme\_H)

Test Passed

#### JuMP example

The code for computing the gradient and Hessian using Enzyme can be re-used for many operators. Thus, it is helpful to encapsulate it into the function:

```
.....
 enzyme_derivatives(f::Function) -> Tuple{Function,Function}
Return a tuple of functions that evaluate the gradient and Hessian of `f` using
Enzyme.jl.
.....
function enzyme_derivatives(f::Function)
 function \nabla f(g::AbstractVector{T}, x::Vararg{T,N}) where {T,N}
 g .= Enzyme.autodiff(Enzyme.Reverse, f, Enzyme.Active.(x)...)[1]
 return
 end
 function \nabla^2 f(H::AbstractMatrix{T}, x::Vararg{T,N}) where {T,N}
 direction(i) = ntuple(j -> Enzyme.Active(T(i == j)), N)
 \nabla f(x...) = Enzyme.autodiff(Enzyme.Reverse, f, Enzyme.Active, x...)[1]
 hess = Enzyme.autodiff(
 Enzyme.Forward,
 ⊽f,
 Enzyme.BatchDuplicated.(Enzyme.Active.(x), ntuple(direction, N))...,
)[1]
 for j in 1:N, i in 1:j
 H[j, i] = hess[j][i]
 end
 return
 end
 return ⊽f, ⊽²f
end
```

Main.enzyme derivatives

Here's an example using enzyme\_derivatives:

```
function enzyme_rosenbrock()
 model = Model(Ipopt.Optimizer)
 set_silent(model)
 @variable(model, x[1:2])
 @operator(model, op_rosenbrock, 2, f, enzyme_derivatives(f)...)
 @objective(model, Min, op_rosenbrock(x[1], x[2]))
 optimize!(model)
 assert_is_solved_and_feasible(model)
 return value.(x)
end
```

enzyme\_rosenbrock()

# DifferentiationInterface

Julia offers many different autodiff packages. DifferentiationInterface.jl is a package that provides an abstraction layer across a few underlying autodiff libraries.

#### Warning

The JuMP developers cannot help you debug error messages related to DifferentiationInterface. If the operator works, it works. If not, we suggest you directly try using a different automatic differentiation library rather than the DI wrapper. See juliadiff.org for details.

All the necessary information about your choice of underlying autodiff package is encoded in a "backend object" like this one:

DifferentiationInterface.AutoForwardDiff()

ADTypes.AutoForwardDiff()

This type comes from another package called ADTypes.jl, but DifferentiationInterface re-exports it. Other options include AutoZygote() and AutoFiniteDiff().

## Gradient

Apart from providing the backend object, the syntax below remains very similar:

```
function di_Vf(
 g::AbstractVector{T},
 x::Vararg{T,N};
 backend = DifferentiationInterface.AutoForwardDiff(),
) where {T,N}
```

```
DifferentiationInterface.gradient!(splat(f), g, backend, collect(x))
return
end
```

di\_ $\nabla f$  (generic function with 1 method)

Let's check that we find the analytic solution:

```
di_g = zeros(2)
di_⊽f(di_g, x...)
Test.@test ≈(analytic_g, di_g)
```

### Test Passed

### Hessian

The Hessian follows exactly the same logic, except we need only the lower triangle.

```
function di_V²f(
 H::AbstractMatrix{T},
 x::Vararg{T,N};
 backend = DifferentiationInterface.AutoForwardDiff(),
) where {T,N}
 H_dense = DifferentiationInterface.hessian(splat(f), backend, collect(x))
 for i in 1:N, j in 1:i
 H[i, j] = H_dense[i, j]
 end
 return
end
```

di\_ $\nabla^2 f$  (generic function with 1 method)

Let's check that we find the analytic solution:

di\_H = zeros(2, 2) di\_∇²f(di\_H, x...) Test.@test ≈(analytic\_H, di\_H)

#### Test Passed

### JuMP example

The code for computing the gradient and Hessian using DifferentiationInterface can be re-used for many operators. Thus, it is helpful to encapsulate it into the function:

```
.....
 di_derivatives(f::Function; backend) -> Tuple{Function,Function}
Return a tuple of functions that evaluate the gradient and Hessian of `f` using
DifferentiationInterface.jl with any given `backend`.
.....
function di_derivatives(f::Function; backend)
 function ∇f(g::AbstractVector{T}, x::Vararg{T,N}) where {T,N}
 DifferentiationInterface.gradient!(splat(f), g, backend, collect(x))
 return
 end
 function \nabla^2 f(H::AbstractMatrix{T}, x::Vararg{T,N}) where {T,N}
 H_dense =
 DifferentiationInterface.hessian(splat(f), backend, collect(x))
 for i in 1:N, j in 1:i
 H[i, j] = H_dense[i, j]
 end
 return
 end
 return ∇f, ∇²f
end
```

#### Main.di\_derivatives

Here's an example using di\_derivatives:

```
function di_rosenbrock(; backend)
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, x[1:2])
@operator(model, op_rosenbrock, 2, f, di_derivatives(f; backend)...)
@objective(model, Min, op_rosenbrock(x[1], x[2]))
optimize!(model)
assert_is_solved_and_feasible(model)
return value.(x)
end
```

```
di_rosenbrock(; backend = DifferentiationInterface.AutoForwardDiff())
```

# 7.5 User-defined Hessians

This tutorial was generated using Literate.jl. Download the source as a .jl file.

In this tutorial, we explain how to write a user-defined operator (see User-defined operators) with a Hessian matrix explicitly provided by the user.

For a more advanced example, see Nested optimization problems.

## **Required packages**

This tutorial uses the following packages:

using JuMP import Ipopt

#### **Rosenbrock example**

As a simple example, we consider the Rosenbrock function:

rosenbrock(x...) =  $(1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2$ 

rosenbrock (generic function with 1 method)

which has the gradient vector:

```
function ∇rosenbrock(g::AbstractVector, x...)
g[1] = 400 * x[1]^3 - 400 * x[1] * x[2] + 2 * x[1] - 2
g[2] = 200 * (x[2] - x[1]^2)
return
end
```

∇rosenbrock (generic function with 1 method)

and the Hessian matrix:

```
function ∇²rosenbrock(H::AbstractMatrix, x...)
H[1, 1] = 1200 * x[1]^2 - 400 * x[2] + 2
H[1, 2] = -400 * x[1] <-- not needed because Hessian is symmetric
H[2, 1] = -400 * x[1]
H[2, 2] = 200.0
return
end</pre>
```

 $\nabla^2$ rosenbrock (generic function with 1 method)

You may assume the Hessian matrix H is initialized with zeros, and because it is symmetric you need only to fill in the non-zero of the lower-triangular terms.

The matrix type passed in as H depends on the automatic differentiation system, so make sure the first argument to the Hessian function supports an AbstractMatrix (it may be something other than Matrix{Float64}). However, you may assume only that H supports size(H) and setindex!.

Now that we have the function, its gradient, and its Hessian, we can construct a JuMP model, add the operator, and use it in a macro:

```
model = Model(Ipopt.Optimizer)
@variable(model, x[1:2])
@operator(model, op_rosenbrock, 2, rosenbrock, ∇rosenbrock, ∇²rosenbrock)
@objective(model, Min, op_rosenbrock(x[1], x[2]))
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model; verbose = true)
```

```
solution_summary(; result = 1, verbose = true)
- solver_name
 : Ipopt
- Termination
| | result_count : 1
| ^L raw_status
 : Solve_Succeeded
Solution (result = 1)
| | dual_status : FEASIBLE_POINT
| | objective_value : 1.21190e-27
| + dual_objective_value : 0.00000e+00
| ^L value
 + ×[1] : 1.00000e+00
^L x[2] : 1.00000e+00
L Work counters
 - solve_time (sec) : 3.67939e-02
 L barrier iterations : 14
```

# 7.6 Nested optimization problems

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to show how to solve a *nested* optimization problem, where an *upper* problem uses the results from the optimization of a *lower* subproblem.

To model the problem, we define a user-defined operator to handle the decomposition of the lower problem inside the upper one. Finally, we show how to improve the performance by using a cache that avoids resolving the lower problem.

For a simpler example of writing a user-defined operator, see the User-defined Hessians tutorial.

### Info

The JuMP extension BilevelJuMP.jl can also be used to model and solve bilevel optimization problems.

### **Required packages**

This tutorial uses the following packages:

using JuMP import Ipopt

# Formulation

In the rest of this tutorial, our goal is to solve the bilevel optimization problem:

$$\min_{\substack{x,z \\ x,z}} \quad x_1^2 + x_2^2 + z \\ z = \max_{\substack{y \\ y \\ x.t.}} \quad x_1^2 y_1 + x_2^2 y_2 - x_1 y_1^4 - 2x_2 y_2^4 \\ s.t. \quad (y_1 - 10)^2 + (y_2 - 10)^2 \le 25 \\ x \ge 0.$$

This bilevel optimization problem is composed of two nested optimization problems. An *upper* level, involving variables x, and a *lower* level, involving variables y. From the perspective of the lower-level problem, the values of x are fixed parameters, and so the model optimizes y given those fixed parameters. Simultaneously, the upper-level problem optimizes x and z given the response of y.

### Decomposition

There are a few ways to solve this problem, but we are going to use a nonlinear decomposition method. The first step is to write a function to compute the lower-level problem:

$$V(x_1, x_2) = \max_{\substack{y \\ s.t.}} x_1^2 y_1 + x_2^2 y_2 - x_1 y_1^4 - 2x_2 y_2^4$$
  
s.t.  $(y_1 - 10)^2 + (y_2 - 10)^2 \le 25$ 

```
function solve_lower_level(x...)
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, y[1:2])
@objective(
 model,
 Max,
 x[1]^2 * y[1] + x[2]^2 * y[2] - x[1] * y[1]^4 - 2 * x[2] * y[2]^4,
)
@constraint(model, (y[1] - 10)^2 + (y[2] - 10)^2 <= 25)
optimize!(model)
assert_is_solved_and_feasible(model)
return objective_value(model), value.(y)
end</pre>
```

solve\_lower\_level (generic function with 1 method)

The next function takes a value of x and returns the optimal lower-level objective-value and the optimal response y. The reason why we need both the objective and the optimal y will be made clear shortly, but for now let us define:

```
function V(x...)
 f, _ = solve_lower_level(x...)
 return f
end
```

V (generic function with 1 method)

Then, we can substitute V into our full problem to create:

$$\min_{\substack{x \\ s.t.}} \quad x_1^2 + x_2^2 + V(x_1, x_2)$$

This looks like a nonlinear optimization problem with a user-defined operator V! However, because V solves an optimization problem internally, we can't use automatic differentiation to compute the first and second derivatives. Instead, we can use JuMP's ability to pass callback functions for the gradient and Hessian instead.

First up, we need to define the gradient of V with respect to x. In general, this may be difficult to compute, but because x appears only in the objective, we can just differentiate the objective function with respect to x, giving:

```
function \U0377V(g::AbstractVector, x...)
_, y = solve_lower_level(x...)
g[1] = 2 * x[1] * y[1] - y[1]^4
g[2] = 2 * x[2] * y[2] - 2 * y[2]^4
return
end
```

 $\nabla V$  (generic function with 1 method)

Second, we need to define the Hessian of V with respect to x. This is a symmetric matrix, but in our example only the diagonal elements are non-zero:

```
function ∇²V(H::AbstractMatrix, x...)
 _, y = solve_lower_level(x...)
 H[1, 1] = 2 * y[1]
 H[2, 2] = 2 * y[2]
 return
end
```

 $\nabla^2 V$  (generic function with 1 method)

# Info

Providing an explicit Hessian function is optional if first derivatives are already available.

We now have enough to define our bilevel optimization problem:

```
model = Model(Ipopt.Optimizer)
@variable(model, x[1:2] >= 0)
@operator(model, op_V, 2, V, \nabla V, \nabla^2V)
@objective(model, Min, x[1]^2 + x[2]^2 + op_V(x[1], x[2]))
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : Ipopt
| Termination
| | termination_status : LOCALLY_SOLVED
| | result_count : 1
| L raw_status : Solve_Succeeded
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | objective_value : -4.18983e+05
| L dual_objective_value : 0.00000e+00
L Work counters
| solve_time (sec) : 5.09679e-01
L barrier_iterations : 32
```

The optimal objective value is:

objective\_value(model)

-418983.48680640775

and the optimal upper-level decision variables x are:

value.(x)

2-element Vector{Float64}: 154.97862337234338 180.0096143098799 To compute the optimal lower-level decision variables, we need to call solve\_lower\_level with the optimal upper-level decision variables:

```
_, y = solve_lower_level(value.(x)...)
y
```

```
2-element Vector{Float64}:
7.072593961143734
5.94656989283847
```

### Improving performance

Our solution approach works, but it has a performance problem: every time we need to compute the value, gradient, or Hessian of V, we have to re-solve the lower-level optimization problem. This is wasteful, because we will often call the gradient and Hessian at the same point, and so solving the problem twice with the same input repeats work unnecessarily.

We can work around this by using a cache:

```
mutable struct Cache
 x::Any
 f::Float64
 y::Vector{Float64}
end
```

with a function to update the cache if needed:

```
function _update_if_needed(cache::Cache, x...)
 if cache.x !== x
 cache.f, cache.y = solve_lower_level(x...)
 cache.x = x
 end
 return
end
```

\_update\_if\_needed (generic function with 1 method)

Then, we define cached versions of out three functions which call \_updated\_if\_needed and return values from the cache.

```
function cached_f(cache::Cache, x...)
 _update_if_needed(cache, x...)
 return cache.f
end
function cached_⊽f(cache::Cache, g::AbstractVector, x...)
 _update_if_needed(cache, x...)
```

```
g[1] = 2 * x[1] * cache.y[1] - cache.y[1]^4
g[2] = 2 * x[2] * cache.y[2] - 2 * cache.y[2]^4
return
end
function cached_V²f(cache::Cache, H::AbstractMatrix, x...)
_update_if_needed(cache, x...)
H[1, 1] = 2 * cache.y[1]
H[2, 2] = 2 * cache.y[2]
return
end
```

cached\_ $\nabla^2 f$  (generic function with 1 method)

Now we're ready to setup and solve the upper level optimization problem:

```
model = Model(Ipopt.Optimizer)
@variable(model, x[1:2] >= 0)
cache = Cache(Float64[], NaN, Float64[])
@operator(
 model,
 op_cached_f,
 2,
 (x...) -> cached_f(cache, x...),
 (g, x...) -> cached_Vf(cache, g, x...),
 (H, x...) -> cached_V²f(cache, H, x...),
)
@objective(model, Min, x[1]^2 + x[2]^2 + op_cached_f(x[1], x[2]))
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : Ipopt
| Termination
| | termination_status : LOCALLY_SOLVED
| | result_count : 1
| L raw_status : Solve_Succeeded
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | objective_value : -4.18983e+05
| L dual_objective_value : 0.00000e+00
L Work counters
| solve_time (sec) : 1.92476e-01
L barrier_iterations : 32
```

objective\_value(model)

-418983.48680640775

and upper-level decision variable x:

value.(x)

```
2-element Vector{Float64}:
154.97862337234338
180.0096143098799
```

# 7.7 Computing Hessians

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to demonstrate how to compute the Hessian of the Lagrangian of a nonlinear program.

#### Warning

This is an advanced tutorial that interacts with the low-level nonlinear interface of MathOptInterface.

By default, JuMP exports the MOI symbol as an alias for the MathOptInterface.jl package. We recommend making this more explicit in your code by adding the following lines:

import MathOptInterface as MOI

Given a nonlinear program:

$$\min_{x \in \mathbb{R}^n} \qquad \qquad f(x) \tag{7.4}$$

 $l \le g_i(x) \le u \tag{7.5}$ 

the Hessian of the Lagrangian is computed as:

s.t.

$$H(x,\sigma,\mu) = \sigma \nabla^2 f(x) + \sum_{i=1}^m \mu_i \nabla^2 g_i(x)$$

where x is a primal point,  $\sigma$  is a scalar (typically 1), and  $\mu$  is a vector of weights corresponding to the Lagrangian dual of the constraints.

# **Required packages**

This tutorial uses the following packages:

using JuMP import Ipopt import LinearAlgebra import Random import SparseArrays

# The basic model

To demonstrate how to interact with the lower-level nonlinear interface, we need an example model. The exact model isn't important; we use the model from The Rosenbrock function tutorial, with some additional constraints to demonstrate various features of the lower-level interface.

```
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, x[i = 1:2], start = -i)
@constraint(model, g_1, x[1]^2 <= 1)
@constraint(model, g_2, (x[1] + x[2])^2 <= 2)
@objective(model, Min, (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2)
optimize!(model)
assert_is_solved_and_feasible(model)</pre>
```

#### The analytic solution

With a little work, it is possible to analytically derive the correct hessian:

```
function analytic_hessian(x, \sigma, \mu)

g_1_H = [2.0 0.0; 0.0 0.0]

g_2_H = [2.0 2.0; 2.0 2.0]

f_H = zeros(2, 2)

f_H[1, 1] = 2.0 + 1200.0 * x[1]^2 - 400.0 * x[2]

f_H[1, 2] = f_H[2, 1] = -400.0 * x[1]

f_H[2, 2] = 200.0

return \sigma * f_H + \mu' * [g_1_H, g_2_H]

end
```

analytic\_hessian (generic function with 1 method)

Here are various points:

analytic\_hessian([1, 1], 0, [0, 0])

2×2 Matrix{Float64}: 0.0 0.0 0.0 0.0 analytic\_hessian([1, 1], 0, [1, 0])

2×2 Matrix{Float64}:
 2.0 0.0
 0.0 0.0

analytic\_hessian([1, 1], 0, [0, 1])

```
2×2 Matrix{Float64}:
2.0 2.0
2.0 2.0
```

analytic\_hessian([1, 1], 1, [0, 0])

2×2 Matrix{Float64}: 802.0 -400.0 -400.0 200.0

## Create a nonlinear model

JuMP delegates automatic differentiation to the MOI.Nonlinear submodule. Therefore, to compute the Hessian of the Lagrangian, we need to create a MOI.Nonlinear.Model object:

```
rows = Any[]
nlp = MOI.Nonlinear.Model()
for (F, S) in list_of_constraint_types(model)
 if F <: VariableRef
 continue # Skip variable bounds
 end
 for ci in all_constraints(model, F, S)
 push!(rows, ci)
 object = constraint_object(ci)
 MOI.Nonlinear.add_constraint(nlp, object.func, object.set)
 end
end
MOI.Nonlinear.set_objective(nlp, objective_function(model))
nlp</pre>
```

A Nonlinear.Model with: 1 objective 0 parameters 0 expressions 2 constraints It is important that we save the constraint indices in a vector rows, so that we know the order of the constraints in the nonlinear model.

Next, we need to convert our model into an MOI.Nonlinear.Evaluator, specifying an automatic differentiation backend. In this case, we use MOI.Nonlinear.SparseReverseMode:

```
evaluator = MOI.Nonlinear.Evaluator(
 nlp,
 MOI.Nonlinear.SparseReverseMode(),
 index.(all_variables(model)),
)
```

Nonlinear. Evaluator with available features:

\* :Grad

\* :Jac

\* :JacVec

\* :Hess

\* :HessVec

\* :ExprGraph

Before computing anything with the evaluator, we need to initialize it. Use MOI.features\_available to see what we can query:

```
MOI.features_available(evaluator)
```

```
6-element Vector{Symbol}:
:Grad
:Jac
:JacVec
:Hess
:HessVec
:ExprGraph
```

Consult the MOI documentation for specifics, but to obtain the Hessian matrix, we need to initialize :Hess:

MOI.initialize(evaluator, [:Hess])

MOI represents the Hessian as a sparse matrix. Get the sparsity pattern as follows:

hessian\_sparsity = MOI.hessian\_lagrangian\_structure(evaluator)

7-element Vector{Tuple{Int64, Int64}}:

(1, 1)

(2, 2)

(2, 1)

(1, 1)

(1, 1) (2, 2) (2, 1)

The sparsity pattern has a few properties of interest:

- + Each element (i, j) indicates a structural non-zero in row i and column j
- The list may contain duplicates, in which case we should add the values together
- The list does not need to be sorted
- The list may contain any mix of lower- or upper-triangular indices

This format matches Julia's sparse-triplet form of a SparseArray, so we can convert from the sparse Hessian representation to a Julia SparseArray as follows:

```
I = [i for (i, _) in hessian_sparsity]
J = [j for (_, j) in hessian_sparsity]
V = zeros(length(hessian_sparsity))
n = num_variables(model)
H = SparseArrays.sparse(I, J, V, n, n)
```

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 3 stored entries:
0.0 .
0.0 0.0
```

Of course, knowing where the zeros are isn't very interesting. We really want to compute the value of the Hessian matrix at a point.

```
MOI.eval_hessian_lagrangian(evaluator, V, ones(n), 1.0, ones(length(rows)))
H = SparseArrays.sparse(I, J, V, n, n)
```

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 3 stored entries:
806.0 .
-398.0 202.0
```

In practice, we often want to compute the value of the hessian at the optimal solution.

First, we compute the primal solution. To do so, we need a vector of the variables in the order that they were passed to the solver:

x = all\_variables(model)

```
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

Here x[1] is the variable that corresponds to column 1, and so on. Here's the optimal primal solution:

x\_optimal = value.(x)

```
2-element Vector{Float64}:
0.7903587565231842
0.6238546272155127
```

Next, we need the optimal dual solution associated with the nonlinear constraints (this is where it is important to record the order of the constraints as we added them to nlp):

y\_optimal = dual.(rows)

```
2-element Vector{Float64}:
-8.038451738599348e-8
-0.05744089305771262
```

Now we can compute the Hessian at the optimal primal-dual point:

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 3 stored entries:
501.944
-316.258 199.885
```

However, this Hessian isn't quite right because it isn't symmetric. We can fix this by filling in the appropriate off-diagonal terms:

```
function fill_off_diagonal(H)
 ret = H + H'
 row_vals = SparseArrays.rowvals(ret)
 non_zeros = SparseArrays.nonzeros(ret)
 for col in 1:size(ret, 2)
 for i in SparseArrays.nzrange(ret, col)
 if col == row_vals[i]
 non_zeros[i] /= 2
 end
 end
```

end return ret end

fill\_off\_diagonal(H)

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 4 stored entries:
501.944 -316.258
-316.258 199.885
```

Putting everything together:

```
function compute_optimal_hessian(model::Model)
 rows = Any[]
 nlp = MOI.Nonlinear.Model()
 for (F, S) in list_of_constraint_types(model)
 for ci in all_constraints(model, F, S)
 push!(rows, ci)
 object = constraint_object(ci)
 MOI.Nonlinear.add_constraint(nlp, object.func, object.set)
 end
 end
 MOI.Nonlinear.set_objective(nlp, objective_function(model))
 x = all_variables(model)
 backend = MOI.Nonlinear.SparseReverseMode()
 evaluator = MOI.Nonlinear.Evaluator(nlp, backend, index.(x))
 MOI.initialize(evaluator, [:Hess])
 hessian_sparsity = MOI.hessian_lagrangian_structure(evaluator)
 I = [i for (i, _) in hessian_sparsity]
 J = [j for (_, j) in hessian_sparsity]
 V = zeros(length(hessian_sparsity))
 MOI.eval_hessian_lagrangian(evaluator, V, value.(x), 1.0, dual.(rows))
 H = SparseArrays.sparse(I, J, V, length(x), length(x))
 return Matrix(fill_off_diagonal(H))
end
```

H\_star = compute\_optimal\_hessian(model)

2×2 Matrix{Float64}: 501.944 -316.258 -316.258 199.885

If we compare our solution against the analytical solution:

```
analytic_hessian(value.(x), 1.0, dual.([g_1, g_2]))
```

2×2 Matrix{Float64}: 501.944 -316.258 -316.258 199.885

If we look at the eigenvalues of the Hessian:

```
LinearAlgebra.eigvals(H_star)
```

```
2-element Vector{Float64}:
0.4443995924983142
701.3843426037456
```

we see that they are all positive. Therefore, the Hessian is positive definite, and so the solution found by lpopt is a local minimizer.

# Jacobians

In addition to the Hessian, it is also possible to query other parts of the nonlinear model. For example, the Jacobian of the constraints can be queried using MOI.jacobian\_structure and MOI.eval\_constraint\_jacobian.

```
function compute_optimal_jacobian(model::Model)
 rows = Any[]
 nlp = MOI.Nonlinear.Model()
 for (F, S) in list_of_constraint_types(model)
 for ci in all_constraints(model, F, S)
 if !(F <: VariableRef)</pre>
 push!(rows, ci)
 object = constraint_object(ci)
 MOI.Nonlinear.add_constraint(nlp, object.func, object.set)
 end
 end
 end
 MOI.Nonlinear.set_objective(nlp, objective_function(model))
 x = all_variables(model)
 backend = MOI.Nonlinear.SparseReverseMode()
 evaluator = MOI.Nonlinear.Evaluator(nlp, backend, index.(x))
 # Initialize the Jacobian
 MOI.initialize(evaluator, [:Jac])
 # Query the Jacobian structure
 sparsity = MOI.jacobian_structure(evaluator)
 I, J, V = first.(sparsity), last.(sparsity), zeros(length(sparsity))
 # Query the Jacobian values
 MOI.eval constraint jacobian(evaluator, V, value.(x))
 return SparseArrays.sparse(I, J, V, length(rows), length(x))
end
```

compute\_optimal\_jacobian(model)

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 3 stored entries:
1.58072 ·
2.82843 2.82843
```

#### Compare that to the analytic solution:

```
y = value.(x)
[2y[1] 0; 2y[1]+2y[2] 2y[1]+2y[2]]
```

```
2×2 Matrix{Float64}:
1.58072 0.0
2.82843 2.82843
```

# 7.8 Example: mixed complementarity problems

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to provide a collection of mixed complementarity programs.

## **Required packages**

This tutorial uses the following packages:

using JuMP
import PATHSolver

### Background

A mixed complementarity problem has the form:

$$F_i(x) \perp x_i i = 1 \dots n \tag{7.6}$$

$$l_i < x_i < u_i i = 1 \dots n. \tag{7.7}$$

where the  $\perp$  constraint enforces the following relations:

- If  $l_i < x_i < u_i$ , then  $F_i(x) = 0$
- If  $l_i = x_i$ , then  $F_i(x) \ge 0$
- If  $x_i = u_i$ , then  $F_i(x) \leq 0$

You may have seen a complementarity problem written as  $0 \le F(x) \perp x \ge 0$ . This is a special case of a mixed complementarity problem in which  $l_i = 0$  and  $u_i = \infty$ .

Importantly, a mixed complementarity problem does not have an objective, and no other constraint types are present.

#### Linear complementarity

Form a mixed complementarity problem using the perp symbol  $\perp$  (type \perp<tab> in the REPL).

```
M = [0 0 -1 -1; 0 0 1 -2; 1 -1 2 -2; 1 2 -2 4]
q = [2, 2, -2, -6]
model = Model(PATHSolver.Optimizer)
set_silent(model)
@variable(model, 0 <= x[1:4] <= 10, start = 0)
@constraint(model, M * x + q ⊥ x)
optimize!(model)
assert_is_solved_and_feasible(model)
value.(x)</pre>
```

```
4-element Vector{Float64}:
 2.8
 0.0
 0.8
 1.2
```

## Other ways of writing linear complementarity problems

You do not need to use a single vector of variables, and the complementarity constraints can be given in any order. In addition, you can use the perp symbol, the complements (F, x) syntax, or the MOI.Complements set.

```
model = Model(PATHSolver.Optimizer)
set_silent(model)
@variable(model, 0 <= w <= 10, start = 0)
@variable(model, 0 <= x <= 10, start = 0)
@variable(model, 0 <= y <= 10, start = 0)
@variable(model, 0 <= z <= 10, start = 0)
@constraint(model, complements(y - 2z + 2, x))
@constraint(model, [-y - z + 2, w] in MOI.Complements(2))
@constraint(model, w + 2x - 2y + 4z - 6 ± z)
@constraint(model, w - x + 2y - 2z - 2 ± y)
optimize!(model)
assert_is_solved_and_feasible(model)
value.([w, x, y, z])</pre>
```

```
4-element Vector{Float64}:
2.8
0.0
0.8
1.2
```

# Transportation

This example is a reformulation of the transportation problem from Chapter 3.3 of Dantzig, G.B. (1963). *Linear Programming and Extensions*. Princeton University Press, Princeton, New Jersey. It is based on the GAMS model gamslib\_transmcp.

```
capacity = Dict("seattle" => 350, "san-diego" => 600)
demand = Dict("new-york" => 325, "chicago" => 300, "topeka" => 275)
cost = Dict(
 ("seattle" => "new-york") => 90 * 2.5 / 1 000,
 ("seattle" => "chicago") => 90 * 1.7 / 1_000,
 ("seattle" => "topeka") => 90 * 1.8 / 1_000,
 ("san-diego" => "new-york") => 90 * 2.5 / 1_000,
 ("san-diego" => "chicago") => 90 * 1.8 / 1_000,
 ("san-diego" => "topeka") => 90 * 1.4 / 1_000,
)
plants, markets = keys(capacity), keys(demand)
model = Model(PATHSolver.Optimizer)
set_silent(model)
@variable(model, w[i in plants] >= 0)
@variable(model, p[j in markets] >= 0)
@variable(model, x[i in plants, j in markets] >= 0)
@constraints(
 model,
 begin
 [i in plants, j in markets], w[i] + cost[i=>j] - p[j] \perp x[i, j]
 [i in plants], capacity[i] - sum(x[i, :]) \perp w[i]
 [j in markets], sum(x[:, j]) - demand[j] \perp p[j]
 end
)
optimize!(model)
assert_is_solved_and_feasible(model)
value.(p)
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
 Dimension 1, ["new-york", "chicago", "topeka"]
And data, a 3-element Vector{Float64}:
 0.22500000000033224
 0.15299999994933886
 0.126
```

### Expected utility of insurance

This example is taken from a lecture of the course AAE706, given by Thomas F. Rutherford at the University of Wisconsin, Madison. It models the expected coverage of insurance K that a rational actor would obtain to insure a risk that occurs with probability pi and results in a loss of L.

```
pi = 0.01 # Probability of a bad outcome

L = 0.5 # Loss with a bad outcome

\gamma = 0.02 # Premium for coverage

\sigma = 0.5 # Elasticity

\rho = -1 # Risk exponent

U(C) = C^\rho / \rho

MU(C) = C^(\rho - 1)

model = Model(PATHSolver.Optimizer)

set_silent(model)

@variable(model, EU, start = 1) # Expected utility
```

```
@variable(model, EV, start = 1) # Equivalent variation in income
@variable(model, C_G, start = 1) # Consumption on a good day
@variable(model, C_B, start = 1) # Consumption on a bad day
@variable(model, K, start = 1) # Coverage
@constraints(
 model,
 begin
 (1 - pi) * U(C_G) + pi * U(C_B) - EU \perp EU
 100 * (((1 - pi) * C_G^\rho + pi * C_B^\rho)^(1 / \rho) - 1) - EV \perp EV
 1 - \gamma * K - C_G \perp C_G
 1 - L + (1 - \gamma) * K - C_B \perp C_B
 \gamma * ((1 - pi) * MU(C_G) + pi * MU(C_B)) - pi * MU(C_B) \perp K
 end
)
optimize!(model)
assert_is_solved_and_feasible(model)
value(K)
```

0.20474003534537774

### **Electricity consumption**

This example is a mixed complementarity formulation of Example 3.3.1 from (D'Aertrycke et al., 2017).

This example models a risk neutral competitive equilibrium between a producer and a consumer of electricity.

In our example, we assume a producer is looking to invest in a new power plant with capacity x [MW]. This plant has an annualized capital cost of I [ $\notin$ /MW] and an operating cost of C [ $\notin$ /MWh]. There are 8760 hours in a year.

After making the capital investment, there are five possible consumption scenarios,  $\omega$ , which occur with probability  $\theta_{\omega}$ . In each scenario , the producer makes Y MW of electricity.

There is one consumer in the model, who has a quadratic utility function,  $U(Q) = AQ + \frac{BQ^2}{2}$ .

We now build and solve the mixed complementarity problem with a few brief comments. The economic justification for the model would require a larger tutorial than the space available here. Consult (D'Aertrycke *et al.*, 2017) for details.

```
Annualized capital cost
I = 90 000
C = 60
 # Operation cost per MWh
\tau = 8_{760}
 # Hours per year
\theta = [0.2, 0.2, 0.2, 0.2, 0.2] # Scenario probabilities
A = [300, 350, 400, 450, 500] # Utility function coefficients
B = 1
 # Utility function coefficients
model = Model(PATHSolver.Optimizer)
set_silent(model)
@variable(model, x >= 0, start = 1)
 # Installed capacity
@variable(model, Q[\omega = 1:5] >= 0, start = 1) # Consumption
@variable(model, Y[\omega = 1:5] >= 0, start = 1) # Production
@variable(model, P[\omega = 1:5], start = 1) # Electricity price
@variable(model, \mu[\omega = 1:5] \ge 0, start = 1) # Capital scarcity margin
Unit investment cost equals annualized scarcity profit or investment is 0
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : Path 5.0.03
| Termination
| | termination_status : LOCALLY_SOLVED
| | result_count : 1
| L raw_status : The problem was solved
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : NO_SOLUTION
| L objective_value : 0.00000e+00
L Work counters
L solve_time (sec) : 1.72000e-04
```

An equilibrium solution is to build 389 MW:

value(x)

389.31506849315065

The production in each scenario is:

value.(Q)

```
5-element Vector{Float64}:
240.000000000001
289.999999999999
340.0
389.31506849315065
389.31506849315065
```

The price in each scenario is:

value.(P)

```
5-element Vector{Float64}:
59.99999999999886
60.0
59.99999999999999
60.68493150684928
110.68493150684935
```

# 7.9 Example: classification problems

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to show how JuMP can be used to formulate classification problems.

Classification problems deal with constructing functions, called *classifiers*, that can efficiently classify data into two or more distinct sets. A common application is classifying previously unseen data points after training a classifier on known data.

The theory and models in this tutorial come from Section 9.4 of (Ferris et al., 2007).

## **Required packages**

This tutorial uses the following packages:

```
using JuMP
import DelimitedFiles
import Ipopt
import LinearAlgebra
import Plots
import Random
import Test
```

# **Data and visualisation**

To start, let's generate some points to test with. The argument m is the number of 2-dimensional points:

```
function generate_test_points(m; random_seed = 1)
 rng = Random.MersenneTwister(random_seed)
 return 2.0 .* rand(rng, Float64, m, 2)
end
```

generate\_test\_points (generic function with 1 method)

For the sake of the example, let's take m = 100:
The points are represented row-wise in the matrix P. Let's visualise the points using the Plots package:

```
plot = Plots.scatter(
 P[:, 1],
 P[:, 2];
 xlim = (0, 2.02),
 ylim = (0, 2.02),
 color = :white,
 size = (600, 600),
 legend = false,
)
```



We want to split the points into two distinct sets on either side of a dividing line. We'll then label each point depending on which side of the line it happens to fall. Based on the labels of the point, we'll show how to create a classifier using a JuMP model. We can then test how well our classifier reproduces the original labels and the boundary between them.

Let's make a line to divide the points into two sets by defining a gradient and a constant:

w\_0, g\_0 = [5, 3], 8 line(v::AbstractArray; w = w\_0, g = g\_0) = w' \* v - g line(x::Real; w = w\_0, g = g\_0) = -(w[1] \* x - g) / w[2];

Julia's multiple dispatch feature allows us to define the vector and single-variable form of the line function under the same name.

Let's add this to the plot:

Plots.plot!(plot, line; linewidth = 5)



Now we label the points relative to which side of the line they are. It is numerically useful to have the labels +1 and -1 for the upcoming JuMP formulation.

```
labels = ifelse.(line.(eachrow(P)) .>= 0, 1, -1)
Plots.scatter!(
 plot,
 P[:, 1],
 P[:, 2];
 shape = ifelse.(labels .== 1, :cross, :xcross),
 markercolor = ifelse.(labels .== 1, :blue, :crimson),
 markersize = 8,
)
```



Our goal is to show we can reconstruct the line from *just* the points and the labels.

# Formulation: linear support vector machine

A classifier known as the linear support vector machine (SVM) looks for the affine function  $L(p) = w^{\top}p - g$  that satisfies L(p) < 0 for all points p with a label -1 and  $L(p) \ge 0$  for all points p with a label +1.

The linearly constrained quadratic program that implements this is:

$$\min_{w \in \mathbb{R}^n, g \in \mathbb{R}, y \in \mathbb{R}^m} \quad \frac{1}{2} w^\top w + C \sum_{i=1}^m y_i$$
subject to  $D \cdot (Pw - g) + y \ge 1$ 
 $y \ge 0.$ 

where D is a diagonal matrix of the labels.

We need a default value for the positive penalty parameter C:

 $C \ \Theta = 100.0;$ 

#### **JuMP** formulation

Here is the JuMP model:

```
function solve_SVM_classifier(P::Matrix, labels::Vector; C::Float64 = C_0)
 m, n = size(P)
 model = Model(Ipopt.Optimizer)
 set_silent(model)
 @variable(model, w[1:n])
 @variable(model, g)
 @variable(model, y[1:m] >= 0)
 @objective(model, Min, 1 / 2 * w' * w + C * sum(y))
 D = LinearAlgebra.Diagonal(labels)
 @constraint(model, D * (P * w .- g) .+ y .>= 1)
 optimize!(model)
 assert is solved and feasible(model)
 slack = extrema(value.(y))
 println("Minimum slack: ", slack[1], "\nMaximum slack: ", slack[2])
 classifier(x) = line(x; w = value.(w), g = value(g))
 return model, classifier
end
```

solve\_SVM\_classifier (generic function with 1 method)

#### Results

Let's recover the values that define the classifier by solving the model:

```
_, classifier = solve_SVM_classifier(P, labels)
```

(A JuMP Model
 solver: Ipopt
 objective\_sense: MIN\_SENSE
 L objective\_function\_type: QuadExpr
 num\_variables: 103
 num\_constraints: 200
 L AffExpr in MOI.GreaterThan{Float64}: 100
 L VariableRef in MOI.GreaterThan{Float64}: 100
 Names registered in the model
 L :g, :w, :y, Main.classifier)

## CHAPTER 7. NONLINEAR PROGRAMS

With the solution, we can ask: was the value of the penalty constant "sufficiently large" for this data set? This can be judged in part by the range of the slack variables. If the slack is too large, then we need to increase the penalty constant.

Let's plot the solution and check how we did:

Plots.plot!(plot, classifier; linewidth = 5, linestyle = :dashdotdot)



We find that we have recovered the dividing line from just the information of the points and their labels.

## Nonseparable classes of points

Now, what if the point sets are not cleanly separable by a line (or a hyperplane in higher dimensions)? Does this still work? Let's repeat the process, but this time we will simulate nonseparable classes of points by

```
nearby_indices = abs.(line.(eachrow(P))) .< 1.1</pre>
labels_new = ifelse.(nearby_indices, -labels, labels)
model, classifier = solve_SVM_classifier(P, labels_new)
plot = Plots.scatter(
 P[:, 1],
 P[:, 2];
 xlim = (0, 2.02),
 ylim = (0, 2.02),
 color = :white,
 size = (600, 600),
 legend = false,
)
Plots.scatter!(
 plot,
 P[:, 1],
 P[:, 2];
 shape = ifelse.(labels_new .== 1, :cross, :xcross),
 markercolor = ifelse.(labels_new .== 1, :blue, :crimson),
 markersize = 8,
)
Plots.plot!(plot, classifier; linewidth = 5, linestyle = :dashdotdot)
```



So our JuMP formulation still produces a classifier, but it mis-classifies some of the nonseparable points.

We can find out which points are contributing to the shape of the line by looking at the dual values of the affine constraints and comparing them to the penalty constant C:

```
affine_cons = all_constraints(model, AffExpr, MOI.GreaterThan{Float64})
active_cons = findall(isapprox.(dual.(affine_cons), C_0; atol = 0.001))
findall(nearby_indices) ⊆ active_cons
```

```
true
```

The last statement tells us that our nonseparable points are actively contributing to how the classifier is defined. The remaining points are of interest and are highlighted:

```
P_active = P[setdiff(active_cons, findall(nearby_indices)), :]
Plots.scatter!(
 plot,
 P_active[:, 1],
 P_active[:, 2];
 shape = :hexagon,
 markersize = 8,
 markeropacity = 0.5,
)
```



# Advanced: duality and the kernel method

We now consider an alternative formulation for a linear SVM by solving the dual problem.

#### The dual program

The dual of the linear SVM program is also a linearly constrained quadratic program:

$$\begin{split} \min_{u \in \mathbb{R}^m} & \frac{1}{2} u^\top D P P^\top D u - \mathbf{1}^\top u \\ \text{subject to} & \mathbf{1}^\top D u = 0 \\ & 0 \leq u \leq C \mathbf{1}. \end{split}$$

This is the JuMP model:

```
function solve_dual_SVM_classifier(P::Matrix, labels::Vector; C::Float64 = C_0)
m, n = size(P)
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, 0 <= u[1:m] <= C)
D = LinearAlgebra.Diagonal(labels)
@objective(model, Min, 1 / 2 * u' * D * P * P' * D * u - sum(u))
@constraint(model, con, sum(D * u) == 0)
optimize!(model)
assert_is_solved_and_feasible(model)
w = P' * D * value.(u)
g = dual(con)
classifier(x) = line(x; w = w, g = g)
return classifier
end</pre>
```

solve\_dual\_SVM\_classifier (generic function with 1 method)

We recover the line gradient vector w through setting  $w = P^{\top}Du$ , and the line constant g as the dual value of the single affine constraint.

The dual problem has fewer variables and fewer constraints, so in many cases it may be simpler to solve the dual form.

We can check that the dual form has recovered a classifier:

classifier = solve\_dual\_SVM\_classifier(P, labels)
Plots.plot!(plot, classifier; linewidth = 5, linestyle = :dash)



#### The kernel method

Linear SVM techniques are not limited to finding separating hyperplanes in the original space of the dataset. One could first transform the training data under a nonlinear mapping, apply our method, then map the hyperplane back into original space.

The actual data describing the point set is held in a matrix P, but looking at the dual program we see that what actually matters is the Gram matrix  $PP^{\top}$ , expressing a pairwise comparison (an inner-product) between each point vector. It follows that any mapping of the point set only needs to be defined at the level of *pairwise* maps between points. Such maps are known as *kernel functions*:

$$k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}, \qquad (s,t) \mapsto \langle \Phi(s), \Phi(t) \rangle$$

where the right-hand side applies some transformation  $\Phi : \mathbb{R}^n \to \mathbb{R}^{n'}$  followed by an inner-product in that image space.

In practice, we can avoid having  $\Phi$  explicitly given but instead define a kernel function directly between pairs of vectors. This change to using a kernel function without knowing the map is called the *kernel method* (or sometimes, the *kernel trick*).

#### **Classifier using a Gaussian kernel**

We will demonstrate the application of a Gaussian or radial basis function kernel:

$$k(s,t) = \exp\left(-\mu \|s - t\|_{2}^{2}\right)$$

for some positive parameter  $\mu$ .

k\_gauss(s::Vector, t::Vector;  $\mu = 0.5$ ) = exp(- $\mu$  \* LinearAlgebra.norm(s - t)^2)

k\_gauss (generic function with 1 method)

Given a matrix of points expressed row-wise and a kernel, the next function returns the transformed matrix K that replaces  $PP^{\top}$ :

```
function pairwise_transform(kernel::Function, P::Matrix{T}) where {T}
 m, n = size(P)
 K = zeros(T, m, m)
 for j in 1:m, i in 1:j
 K[i, j] = K[j, i] = kernel(P[i, :], P[j, :])
 end
 return LinearAlgebra.Symmetric(K)
end
```

pairwise\_transform (generic function with 1 method)

Now we're ready to define our optimization problem. We need to provide the kernel function to be used in the problem. Note that any extra keyword arguments here (like parameter values) are passed through to the kernel.

```
function solve_kernel_SVM_classifier(
 kernel::Function,
 P::Matrix,
 labels::Vector;
 C::Float64 = C_0,
 kwargs...,
)
 m, n = size(P)
 K = pairwise_transform(kernel, P)
```

```
model = Model(Ipopt.Optimizer)
 set_silent(model)
 @variable(model, 0 <= u[1:m] <= C)</pre>
 D = LinearAlgebra.Diagonal(labels)
 con = @constraint(model, sum(D * u) == 0)
 @objective(model, Min, 1 / 2 * u' * D * K * D * u - sum(u))
 optimize!(model)
 assert_is_solved_and_feasible(model)
 u_sol, g_sol = value.(u), dual(con)
 function classifier(v::Vector)
 return sum(
 D[i, i] * u_sol[i] * kernel(P[i, :], v; kwargs...) for i in 1:m
) - g_sol
 end
 return classifier
end
```

solve\_kernel\_SVM\_classifier (generic function with 1 method)

This time, we don't recover the line gradient vector w directly. Instead, we compute the classifier f using the function:

$$f(v) = \sum_{i=1}^{m} D_{ii} u_i \ k(p_i, v) - g$$

where  $p_i$  is row vector i of P.

#### **Checkerboard dataset**

To demonstrate this nonlinear technique, we'll use the checkerboard dataset.

```
filename = joinpath(@_DIR_, "data", "checker", "checker.txt")
checkerboard = DelimitedFiles.readdlm(filename, ' ', Int)
labels = ifelse.(iszero.(checkerboard[:, 1]), -1, 1)
B = checkerboard[:, 2:3] ./ 100.0 # rescale to [0,2] x [0,2] square.
plot = Plots.scatter(
 B[:, 1],
 B[:, 2];
 color = ifelse.(labels .== 1, :white, :black),
 markersize = ifelse.(labels .== 1, 4, 2),
 size = (600, 600),
 legend = false,
)
```



Is the technique capable of generating a distinctly nonlinear surface? Let's solve the Gaussian kernel based quadratic problem with these parameters:

classifier = solve\_kernel\_SVM\_classifier(k\_gauss, B, labels; C = 1e5,  $\mu$  = 10.0) grid = [[x, y] for x in 0:0.01:2, y in 0:0.01:2] grid\_pos = [Tuple(g) for g in grid if classifier(g) >= 0] Plots.scatter!(plot, grid\_pos; markersize = 0.2)



We find that the kernel method can perform well as a nonlinear classifier.

The result has a fairly strong dependence on the choice of parameters, with larger values of  $\mu$  allowing for a more complex boundary while smaller values lead to a smoother boundary for the classifier. Determining a better performing kernel function and choice of parameters is covered by the process of *cross-validation* with respect to the dataset, where different testing, training and tuning sets are used to validate the best choice of parameters against a statistical measure of error.

## 7.10 Example: portfolio optimization

This tutorial was generated using Literate.jl. Download the source as a .jl file.

## This tutorial was originally contributed by Arpit Bhatia.

Optimization models play an increasingly important role in financial decisions. Many computational finance problems can be solved efficiently using modern optimization techniques.

This tutorial solves the famous Markowitz Portfolio Optimization problem with data from lecture notes from a course taught at Georgia Tech by Shabbir Ahmed.

## **Required packages**

This tutorial uses the following packages:

```
using JuMP
import DataFrames
import Ipopt
import MultiObjectiveAlgorithms as MOA
import Plots
import Statistics
import StatisPlots
```

#### Formulation

Suppose we are considering investing 1000 dollars in three non-dividend paying stocks, IBM (IBM), Walmart (WMT), and Southern Electric (SEHI), for a one-month period.

We will use the initial money to buy shares of the three stocks at the current market prices, hold these for one month, and sell the shares off at the prevailing market prices at the end of the month.

As a rational investor, we hope to make some profit out of this endeavor, that is, the return on our investment should be positive.

Suppose we bought a stock at p dollars per share in the beginning of the month, and sold it off at s dollars per share at the end of the month. Then the one-month return on a share of the stock is  $\frac{s-p}{p}$ .

Since the stock prices are quite uncertain, so is the end-of-month return on our investment. Our goal is to invest in such a way that the expected end-of-month return is at least \$50 or 5%. Furthermore, we want to make sure that the "risk" of not achieving our desired return is minimum.

Note that we are solving the problem under the following assumptions:

- 1. We can trade any continuum of shares.
- 2. No short-selling is allowed.
- 3. There are no transaction costs.

We model this problem by taking decision variables  $x_i$ , i = 1, 2, 3, denoting the dollars invested in each of the 3 stocks.

Let us denote by  $\tilde{r}_i$  the random variable corresponding to the monthly return (increase in the stock price) per dollar for stock i.

Then, the return (or profit) on  $x_i$  dollars invested in stock i is  $\tilde{r}_i x_i$ , and the total (random) return on our investment is  $\sum_{i=1}^{3} \tilde{r}_i x_i$ . The expected return on our investment is then  $\mathbb{E}\left[\sum_{i=1}^{3} \tilde{r}_i x_i\right] = \sum_{i=1}^{3} \overline{r}_i x_i$ , where  $\overline{r}_i$  is the expected value of the  $\tilde{r}_i$ .

Now we need to quantify the notion of "risk" in our investment.

Markowitz, in his Nobel prize winning work, showed that a rational investor's notion of minimizing risk can be closely approximated by minimizing the variance of the return of the investment portfolio. This variance is given by:

$$\operatorname{Var}\left[\sum_{i=1}^{3} \tilde{r}_{i} x_{i}\right] = \sum_{i=1}^{3} \sum_{j=1}^{3} x_{i} x_{j} \sigma_{ij}$$

where  $\sigma_{ij}$  is the covariance of the return of stock i with stock j.

Note that the right hand side of the equation is the most reduced form of the expression and we have not shown the intermediate steps involved in getting to this form. We can also write this equation as:

$$\operatorname{Var}\left[\sum_{i=1}^{3}\tilde{r}_{i}x_{i}\right]=x^{\top}Qx$$

Where Q is the covariance matrix for the random vector  $\tilde{r}$ .

Finally, we can write the model as:

$$\min x^{\top}Qx$$
  
s.t. 
$$\sum_{i=1}^{3} x_i \le 1000$$
$$\overline{r}^{\top}x \ge 50$$
$$x \ge 0$$

#### Data

For the data in our problem, we use the stock prices given below, in monthly values from November 2000, through November 2001.

```
df = DataFrames.DataFrame(
 [
 93.043 51.826 1.063
 84.585 52.823 0.938
 111.453 56.477 1.000
 99.525 49.805 0.938
 95.819 50.287 1.438
 114.708 51.521 1.700
 111.515 51.531 2.540
 113.211 48.664 2.390
 104.942 55.744 3.120
 99.827 47.916 2.980
 91.607 49.438 1.900
 107.937 51.336 1.750
 115.590 55.081 1.800
],
 [:IBM, :WMT, :SEHI],
)
```

Next, we compute the percentage return for the stock in each month:

	IBM	WMT	SEHI
	Float64	Float64	Float64
1	93.043	51.826	1.063
2	84.585	52.823	0.938
3	111.453	56.477	1.0
4	99.525	49.805	0.938
5	95.819	50.287	1.438
6	114.708	51.521	1.7
7	111.515	51.531	2.54
8	113.211	48.664	2.39
9	104.942	55.744	3.12
10	99.827	47.916	2.98
11	91.607	49.438	1.9
12	107.937	51.336	1.75
13	115.59	55.081	1.8

returns = diff(Matrix(df); dims = 1) ./ Matrix(df[1:end-1, :])

12×3 Matrix{	Float64}:	
-0.0909042	0.0192374	-0.117592
0.317645	0.0691744	0.0660981
-0.107023	-0.118137	-0.062
-0.0372369	0.00967774	0.533049
0.197132	0.0245391	0.182197
-0.0278359	0.000194096	0.494118
0.0152087	-0.0556364	-0.0590551
-0.0730406	0.145487	0.305439
-0.0487412	-0.140428	-0.0448718
-0.0823425	0.0317639	-0.362416
0.178261	0.0383915	-0.0789474
0.0709025	0.0729508	0.0285714

The expected monthly return is:

r = vec(Statistics.mean(returns; dims = 1))

```
3-element Vector{Float64}:
0.026002150277777348
0.008101316405671459
0.07371590949198982
```

and the covariance matrix is:

Q = Statistics.cov(returns)

3×3 Matrix{Float64}: 0.018641 0.00359853 0.00130976 0.00359853 0.00643694 0.00488727 0.00130976 0.00488727 0.0686828

## **JuMP** formulation

```
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, x[1:3] >= 0)
@objective(model, Min, x' * Q * x)
@constraint(model, sum(x) <= 1000)
@constraint(model, r' * x >= 50)
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : Ipopt
| Termination
| | termination_status : LOCALLY_SOLVED
| | result_count : 1
| L raw_status : Solve_Succeeded
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | dual_objective_value : 4.52688e+04
| L dual_objective_value : 4.52688e+04
| Work counters
| solve_time (sec) : 2.99191e-03
| barrier_iterations : 11
```

The optimal allocation of our assets is:

value.(x)

```
3-element Vector{Float64}:
497.045529849864
-9.670578501816873e-9
502.9544801594809
```

So we spend \$497 on IBM, and \$503 on SEHI. This results in a variance of:

scalar\_variance = value(x' \* Q \* x)

22634.417849884147

and an expected return of:

scalar\_return = value(r' \* x)

49.999999500002374

### Multi-objective portfolio optimization

The previous model returned a single solution that minimized the variance, ensuring that our expected return was at least \$50. In practice, we might be willing to accept a slightly higher variance if it meant a much increased expected return. To explore this problem space, we can instead formulate our portfolio optimization problem with two objectives:

- 1. to minimize the variance
- 2. to maximize the expected return

The solution to this bi-objective problem is the efficient frontier of modern portfolio theory, and each point in the solution is a point with the best return for a fixed level of risk.

```
model = Model(() -> MOA.Optimizer(Ipopt.Optimizer))
set_silent(model)
```

We also need to choose a solution algorithm for MOA. For our problem, the efficient frontier will have an infinite number of solutions. Since we cannot find all of the solutions, we choose an approximation algorithm and limit the number of solution points that are returned:

```
set_optimizer_attribute(model, MOA.Algorithm(), MOA.EpsilonConstraint())
set_optimizer_attribute(model, MOA.SolutionLimit(), 50)
```

Now we can define the rest of the model:

```
@variable(model, x[1:3] >= 0)
@constraint(model, sum(x) <= 1000)
@expression(model, variance, x' * Q * x)
@expression(model, expected_return, r' * x)
We want to minimize variance and maximize expected return, but we must pick
a single objective sense `Min`, and negate any `Max` objectives:
@objective(model, Min, [variance, -expected_return])
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)</pre>
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : MOA[algorithm=MultiObjectiveAlgorithms.EpsilonConstraint, optimizer=Ipopt]
| Termination
| | termination_status : OPTIMAL
| | result_count : 50
| | raw_status : Solve complete. Found 50 solution(s)
| | objective_bound : [1.57816e-08,-7.37159e+01]
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : NO_SOLUTION
| | objective_value : [1.57816e-08,-4.19663e-05]
| Work counters
| solve_time (sec) : 2.26958e-01
```

The algorithm found 50 different solutions. Let's plot them to see how they differ:

```
objective_space = Plots.hline(
 [scalar_return];
 label = "Single-objective solution",
 linecolor = :red,
)
Plots.vline!(objective space, [scalar variance]; label = "", linecolor = :red)
Plots.scatter!(
 objective_space,
 [value(variance; result = i) for i in 1:result_count(model)],
 [value(expected_return; result = i) for i in 1:result_count(model)];
 xlabel = "Variance",
 ylabel = "Expected Return",
 label = "",
 title = "Objective space",
 markercolor = "white",
 markersize = 5,
 legend = :bottomright,
)
for i in 1:result_count(model)
 y = objective value(model; result = i)
 Plots.annotate!(objective_space, y[1], -y[2], (i, 3))
end
decision_space = StatsPlots.groupedbar(
 vcat([value.(x; result = i)' for i in 1:result_count(model)]...);
 bar_position = :stack,
 label = ["IBM" "WMT" "SEHI"],
 xlabel = "Solution #",
 ylabel = "Investment (\$)",
 title = "Decision space",
)
Plots.plot(objective_space, decision_space; layout = (2, 1), size = (600, 600))
```



Perhaps our trade-off wasn't so bad after all. Our original solution corresponded to picking a solution #17. If we buy more SEHI, we can increase the return, but the variance also increases. If we buy less SEHI, such as a solution like #5 or #6, then we can achieve the corresponding return without deploying all of our capital. We should also note that at no point should we buy WMT.

## 7.11 Example: nonlinear optimal control of a rocket

This tutorial was generated using Literate.jl. Download the source as a .jl file.

### This tutorial was originally contributed by lain Dunning.

The purpose of this tutorial is to demonstrate how to setup and solve a nonlinear optimization problem.

The example is an optimal control problem of a nonlinear rocket.

#### Info

The JuMP extension InfiniteOpt.jl can also be used to model and solve optimal control problems.

## **Required packages**

This tutorial uses the following packages:

using JuMP import Ipopt import Plots

## Overview

Our goal is to maximize the final altitude of a vertically launched rocket.

We can control the thrust of the rocket, and must take account of the rocket mass, fuel consumption rate, gravity, and aerodynamic drag.

Let us consider the basic description of the model (for the full description, including parameters for the rocket, see COPS3).

There are three state variables in our model:

- Velocity:  $x_v(t)$
- Altitude:  $x_h(t)$
- Mass of rocket and remaining fuel,  $x_m(t)$

and a single control variable:

• Thrust:  $u_t(t)$ .

There are three equations that control the dynamics of the rocket:

- Rate of ascent:  $\frac{dx_h}{dt} = x_v$
- Acceleration:  $\frac{dx_v}{dt} = \frac{u_t D(x_h, x_v)}{x_m} g(x_h)$
- Rate of mass loss:  $\frac{dx_m}{dt} = -\frac{u_t}{c}$

where drag  $D(x_h, x_v)$  is a function of altitude and velocity, gravity  $g(x_h)$  is a function of altitude, and c is a constant.

These forces are defined as:

$$D(x_h, x_v) = D_c \cdot x_v^2 \cdot e^{-h_c \left(\frac{x_h - x_h(0)}{x_h(0)}\right)}$$

and  $g(x_h) = g_0 \cdot \left(\frac{x_h(0)}{x_h}\right)^2$ 

We use a discretized model of time, with a fixed number of time steps, T. Our goal is thus to maximize  $x_h(T)$ .

## Data

All parameters in this model have been normalized to be dimensionless, and they are taken from COPS3.

```
h 0 = 1
 # Initial height
v_0 = 0
 # Initial velocity
m_0 = 1.0
 # Initial mass
m_T = 0.6
 # Final mass
g 0 = 1
 # Gravity at the surface
h c = 500
 # Used for drag
c = 0.5 * sqrt(g_0 * h_0) # Thrust-to-fuel mass
D_c = 0.5 * 620 * m_0 / g_0 # Drag scaling
u_t_max = 3.5 * g_0 * m_0 # Maximum thrust
 # Number of seconds
T max = 0.2
T = 1_{000}
 # Number of time steps
\Delta t = 0.2 / T;
 # Time per discretized step
```

#### **JuMP formulation**

First, we create a model and choose an optimizer. Since this is a nonlinear program, we need to use a nonlinear solver like Ipopt. We cannot use a linear solver like HiGHS.

```
model = Model(Ipopt.Optimizer)
set_silent(model)
```

Next, we create our state and control variables, which are each indexed by t. It is good practice for nonlinear programs to always provide a starting solution for each variable.

We implement boundary conditions by fixing variables to values.

```
fix(x_v[1], v_0; force = true)
fix(x_h[1], h_0; force = true)
fix(x_m[1], m_0; force = true)
fix(u_t[T], 0.0; force = true)
```

The objective is to maximize altitude at end of time of flight.

@objective(model, Max, x\_h[T])

### $x_{h_{1000}}$

Forces are defined as functions:

g (generic function with 1 method)

The dynamical equations are implemented as constraints.

```
ddt(x::Vector, t::Int) = (x[t] - x[t-1]) / Δt
@constraint(model, [t in 2:T], ddt(x_h, t) == x_v[t-1])
@constraint(
 model,
 [t in 2:T],
 ddt(x_v, t) == (u_t[t-1] - D(x_h[t-1], x_v[t-1])) / x_m[t-1] - g(x_h[t-1]),
)
@constraint(model, [t in 2:T], ddt(x_m, t) == -u_t[t-1] / c);
```

Now we optimize the model and check that we found a solution:

```
optimize!(model)
assert_is_solved_and_feasible(model)
solution summary(model)
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : Ipopt
| Termination
| | termination_status : LOCALLY_SOLVED
| | result_count : 1
| L raw_status : Solve_Succeeded
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | objective_value : 1.01278e+00
| L dual_objective_value : 4.66547e+00
| Work counters
| solve_time (sec) : 2.45561e-01
| barrier_iterations : 24
```

Finally, we plot the solution:

```
function plot_trajectory(y; kwargs...)
 return Plots.plot(
 (1:T) * Δt,
 value.(y);
 xlabel = "Time (s)",
 legend = false,
 kwargs...,
)
```

end

```
Plots.plot(
 plot_trajectory(x_h; ylabel = "Altitude"),
 plot_trajectory(x_m; ylabel = "Mass"),
 plot_trajectory(x_v; ylabel = "Velocity"),
 plot_trajectory(u_t; ylabel = "Thrust");
 layout = (2, 2),
)
```



### Next steps

- Experiment with different values for the constants. How does the solution change? In particular, what happens if you change T\_max?
- The dynamical equations use rectangular integration for the right-hand side terms. Modify the equations to use the Trapezoidal rule instead. (As an example,  $x_v[t-1]$  would become 0.5 \* ( $x_v[t-1] + x_v[t]$ ).) Is there a difference?

## 7.12 Example: optimal control for a Space Shuttle reentry trajectory

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by Henrique Ferrolho.

This tutorial demonstrates how to compute a reentry trajectory for the Space Shuttle, by formulating and solving a nonlinear programming problem. The problem was drawn from Chapter 6 of (Betts, 2010).

## Info

The JuMP extension InfiniteOpt.jl can also be used to model and solve optimal control problems.

#### Тір

This tutorial is a more-complicated version of the Example: nonlinear optimal control of a rocket tutorial. If you are new to solving nonlinear programs in JuMP, you may want to start there instead.

### **Required packages**

This tutorial uses the following packages:

using JuMP
import Interpolations
import Ipopt

### Formulation

The motion of the vehicle is defined by the following set of DAEs:

$$\begin{split} \dot{h} &= v \sin \gamma, \\ \dot{\phi} &= \frac{v}{r} \cos \gamma \sin \psi / \cos \theta, \\ \dot{\theta} &= \frac{v}{r} \cos \gamma \cos \psi, \\ \dot{v} &= -\frac{D}{m} - g \sin \gamma, \\ \dot{\gamma} &= \frac{L}{mv} \cos(\beta) + \cos \gamma \left(\frac{v}{r} - \frac{g}{v}\right), \\ \dot{\psi} &= \frac{1}{mv \cos \gamma} L \sin(\beta) + \frac{v}{r \cos \theta} \cos \gamma \sin \psi \sin \theta, \\ g &\leq q_U, \end{split}$$

where the aerodynamic heating on the vehicle wing leading edge is  $q = q_a q_r$  and the dynamic variables are

h	altitude (ft),	$\gamma$	flight path angle (rad),
$\phi$	longitude (rad),	$\psi$	azimuth (rad),
$\theta$	latitude (rad),	$\alpha$	angle of attack (rad),
v	velocity (ft/sec),	$\beta$	bank angle (rad).

The aerodynamic and atmospheric forces on the vehicle are specified by the following quantities (English units):

$$\begin{split} D &= \frac{1}{2} c_D S \rho v^2, \qquad a_0 = -0.20704, \\ L &= \frac{1}{2} c_L S \rho v^2, \qquad a_1 = 0.029244, \\ g &= \mu/r^2, \qquad \mu = 0.14076539 \times 10^{17}, \\ r &= R_e + h, \qquad b_0 = 0.07854, \\ \rho &= \rho_0 \exp[-h/h_r], \qquad b_1 = -0.61592 \times 10^{-2}, \\ \rho_0 &= 0.002378, \qquad b_2 = 0.621408 \times 10^{-3}, \\ h_r &= 23800, \qquad q_r = 17700 \sqrt{\rho} (0.0001v)^{3.07}, \\ c_L &= a_0 + a_1 \hat{\alpha}, \qquad q_a = c_0 + c_1 \hat{\alpha} + c_2 \hat{\alpha}^2 + c_3 \hat{\alpha}^3, \\ c_D &= b_0 + b_1 \hat{\alpha} + b_2 \hat{\alpha}^2, \qquad c_0 = 1.0672181, \\ \hat{\alpha} &= 180\alpha/\pi, \qquad c_1 = -0.19213774 \times 10^{-1}, \\ R_e &= 20902900, \qquad c_2 = 0.21286289 \times 10^{-3}, \\ S &= 2690, \qquad c_3 = -0.10117249 \times 10^{-5}. \end{split}$$

The reentry trajectory begins at an altitude where the aerodynamic forces are quite small with a weight of w = 203000 (lb) and mass  $m = w/g_0$  (slug), where  $g_0 = 32.174$  (ft/sec<sup>2</sup>). The initial conditions are as follows:

$h=260000 \; {\rm ft},$	$v=25600 \; {\rm ft/sec},$
$\phi=0 \; \mathrm{deg},$	$\gamma = -1 \deg,$
$\theta=0 ~{\rm deg},$	$\psi=90~{\rm deg.}$

The final point on the reentry trajectory occurs at the unknown (free) time  $t_F$ , at the so-called terminal area energy management (TAEM) interface, which is defined by the conditions

h=80000 ft, v=2500 ft/sec,  $\gamma=-5$  deg.

As explained in the book, our goal is to maximize the final cross-range, which is equivalent to maximizing the final latitude of the vehicle, that is,  $J = \theta(t_F)$ .

#### Approach

We will use a discretized model of time, with a fixed number of discretized points, n. The decision variables at each point are going to be the state of the vehicle and the controls commanded to it. In addition, we will also make each time step size  $\Delta t$  a decision variable; that way, we can either fix the time step size easily, or allow the solver to fine-tune the duration between each adjacent pair of points. Finally, in order to approximate the derivatives of the problem dynamics, we will use either rectangular or trapezoidal integration.

#### Warning

Do not try to actually land a Space Shuttle using this notebook. There's no mesh refinement going on, which can lead to unrealistic trajectories having position and velocity errors with orders of magnitude  $10^4$  ft and  $10^2$  ft/sec, respectively.



Figure 7.1: Max cross-range shuttle reentry

```
Global variables
const w = 203000.0 # weight (lb)
const g₀ = 32.174 # acceleration (ft/sec^2)
const m = w / g_0 # mass (slug)
Aerodynamic and atmospheric forces on the vehicle
const \rho_{0} = 0.002378
const h_r = 23800.0
const R_{e} = 20902900.0
const \mu = 0.14076539e17
const S = 2690.0
const a_0 = -0.20704
const a1 = 0.029244
const b_0 = 0.07854
const b_1 = -0.61592e-2
const b_2 = 0.621408e-3
const c_{0} = 1.0672181
const C1 = -0.19213774e-1
const c_2 = 0.21286289e-3
const c₃ = -0.10117249e-5
Initial conditions
const h_s = 2.6
 # altitude (ft) / 1e5
const \oplus_s = deg2rad(0) # longitude (rad)
const \theta_s = deg2rad(0) # latitude (rad)
const v_s = 2.56
 # velocity (ft/sec) / le4
const \gamma_s = \text{deg2rad(-1)} \# \text{flight path angle (rad)}
const \psi_s = \text{deg2rad(90)} \# azimuth (rad)
const \alpha_s = deg2rad(0) # angle of attack (rad)
const \beta_s = deg2rad(0) # bank angle (rad)
const t_s = 1.00
 # time step (sec)
```

```
const h_t = 0.8 # altitude (ft) / le5
const v_t = 0.25 # velocity (ft/sec) / le4
const v_t = deg2rad(-5) # flight path angle (rad)
Number of mesh points (knots) to be used
const n = 503
Integration scheme to be used for the dynamics
const integration_rule = "rectangular"
```

### Choose a good linear solver

Picking a good linear solver is **extremely important** to maximize the performance of nonlinear solvers. For the best results, it is advised to experiment different linear solvers.

For example, the linear solver MA27 is outdated and can be quite slow. MA57 is a much better alternative, especially for highly sparse problems (such as trajectory optimization problems).

```
Uncomment the lines below to pass user options to the solver
user options = (
"mu_strategy" => "monotone",
"linear_solver" => "ma27",
)
Create JuMP model, using Ipopt as the solver
model = Model(optimizer_with_attributes(Ipopt.Optimizer, user_options...))
@variables(model, begin
 0 \leq \text{scaled}[1:n]
 # altitude (ft) / 1e5
 # longitude (rad)
 φ[1:n]
 deg2rad(-89) \le \theta[1:n] \le deg2rad(89) \# latitude (rad)
 1e-4 \leq scaled_v[1:n] # velocity (ft/sec) / 1e4
 deg2rad(-89) \le \gamma[1:n] \le deg2rad(89) \# flight path angle (rad)
 # azimuth (rad)
 ψ[1:n]
 deg2rad(-90) \le \alpha[1:n] \le deg2rad(90) \# angle of attack (rad)
 deg2rad(-89) \le \beta[1:n] \le deg2rad(1) # bank angle (rad)
```

```
end);
```

#### Info

Above you can find two alternatives for the  $\Delta t$  variables.

The first one,  $3.5 \le \Delta t[1:n] \le 4.5$  (currently commented), allows some wiggle room for the solver to adjust the time step size between pairs of mesh points. This is neat because it allows the solver to figure out which parts of the flight require more dense discretization than others. (Remember, the number of discretized points is fixed, and this example does not implement mesh refinement.) However, this makes the problem more complex to solve, and therefore leads to a longer computation time.

The second line,  $\Delta t[1:n] == 4.0$ , fixes the duration of every time step to exactly 4.0 seconds. This allows the problem to be solved faster. However, to do this we need to know beforehand that the close-to-optimal total duration of the flight is ~2009 seconds. Therefore, if we split the total duration in slices of 4.0 seconds, we know that we require n = 503 knots to discretize the whole trajectory.

```
Fix initial conditions
fix(scaled_h[1], h_s; force = true)
fix($\phi[1], $\phi_s; force = true)
fix($\phi[1], $\phi_s; force = true)
fix(scaled_v[1], $\nu_s; force = true)
fix($\phi[1], $\nu_s; force = true)
fix($\phi[1], $\nu_s; force = true)
```

```
Fix final conditions
fix(scaled_h[n], h_t; force = true)
fix(scaled_v[n], v_t; force = true)
fix(y[n], y_t; force = true)
```

```
Initial guess: linear interpolation between boundary conditions

x_s = [h_s, \phi_s, \theta_s, v_s, \gamma_s, \psi_s, \alpha_s, \beta_s, t_s]

x_t = [h_t, \phi_s, \theta_s, v_t, \gamma_t, \psi_s, \alpha_s, \beta_s, t_s]

interp_linear = Interpolations.LinearInterpolation([1, n], [x_s, x_t])

initial_guess = mapreduce(transpose, vcat, interp_linear.(1:n))

set_start_value.(all_variables(model), vec(initial_guess))
```

```
Functions to restore `h` and `v` to their true scale
@expression(model, h[j = 1:n], scaled_h[j] * 1e5)
@expression(model, v[j = 1:n], scaled_v[j] * 1e4)
```

```
Helper functions
@expression(model, c_L[j = 1:n], a₀ + a₁ * rad2deg(α[j]))
@expression(model, c_D[j = 1:n], b₀ + b₁ * rad2deg(α[j]) + b₂ * rad2deg(α[j])^2)
@expression(model, p[j = 1:n], p₀ * exp(-h[j] / h_r))
@expression(model, D[j = 1:n], 0.5 * c_D[j] * S * p[j] * v[j]^2)
@expression(model, L[j = 1:n], 0.5 * c_L[j] * S * p[j] * v[j]^2)
@expression(model, r[j = 1:n], R_e + h[j])
@expression(model, g[j = 1:n], µ / r[j]^2)
```

```
(expression(model, \delta\theta[j = 1:n], (v[j] / r[j]) * cos(y[j]) * cos(\psi[j])))
<code>@expression(model, \delta v[j = 1:n], -(D[j] / m) - g[j] * sin(\gamma[j]))</code>
@expression(
 model,
 \delta\gamma[j = 1:n],
 (L[j] / (m * v[j])) * cos(\beta[j]) +
 cos(\gamma[j]) * ((v[j] / r[j]) - (g[j] / v[j]))
)
@expression(
 model,
 \delta \psi[j = 1:n],
 (1 / (m * v[j] * cos(\gamma[j]))) * L[j] * sin(\beta[j]) +
 (v[j] / (r[j] * cos(\theta[j]))) * cos(\gamma[j]) * sin(\psi[j]) * sin(\theta[j])
)
System dynamics
for j in 2:n
 i = j - 1 # index of previous knot
 if integration_rule == "rectangular"
 # Rectangular integration
 @constraint(model, h[j] == h[i] + \Delta t[i] * \delta h[i])
 \texttt{@constraint(model, } \phi[j] == \phi[i] + \Delta t[i] * \delta \phi[i])
 \texttt{@constraint(model, } \theta[j] == \theta[i] + \Delta t[i] * \delta \theta[i])
 (constraint(model, v[j] == v[i] + \Delta t[i] * \delta v[i])
 @constraint(model, \gamma[j] == \gamma[i] + \Delta t[i] * \delta \gamma[i])
 \texttt{@constraint(model, } \psi[j] == \psi[i] + \Delta t[i] * \delta \psi[i])
 elseif integration_rule == "trapezoidal"
 # Trapezoidal integration
 \texttt{@constraint(model, h[j] == h[i] + 0.5 * \Delta t[i] * (\delta h[j] + \delta h[i]))}
 \texttt{@constraint(model, } \phi[j] == \phi[i] + 0.5 * \Delta t[i] * (\delta \phi[j] + \delta \phi[i]))
 \texttt{@constraint(model, } \theta[j] == \theta[i] + 0.5 * \Delta t[i] * (\delta \theta[j] + \delta \theta[i]))
 @constraint(model, v[j] == v[i] + 0.5 * \Delta t[i] * (\delta v[j] + \delta v[i]))
 \texttt{@constraint(model, } \gamma[j] == \gamma[i] + 0.5 * \Delta t[i] * (\delta \gamma[j] + \delta \gamma[i]))
 \texttt{@constraint(model, \psi[j] == \psi[i] + 0.5 * \Delta t[i] * (\delta \psi[j] + \delta \psi[i]))}
 else
 @error "Unexpected integration rule '$(integration_rule)'"
 end
end
Objective: Maximize cross-range
<u>@objective(model, Max, \theta[n])</u></u>
set_silent(model) # Hide solver's verbose output
optimize!(model) # Solve for the control and state
assert_is_solved_and_feasible(model)
Show final cross-range of the solution
println(
 "Final latitude \theta = ",
 round(objective value(model) |> rad2deg; digits = 2),

)
```

Final latitude  $\theta$  = 34.18°

### **Plotting the results**

Let's plot the results to visualize the optimal trajectory:

```
using Plots
ts = cumsum([0; value.(\Deltat)])[1:end-1]
plt_altitude = plot(
 ts,
 value.(scaled_h);
 legend = nothing,
 title = "Altitude (100,000 ft)",
)
plt_longitude =
 plot(ts, rad2deg.(value.($\phi)); legend = nothing, title = "Longitude (deg)")
plt_latitude =
 plot(ts, rad2deg.(value.(0)); legend = nothing, title = "Latitude (deg)")
plt_velocity = plot(
 ts,
 value.(scaled_v);
 legend = nothing,
 title = "Velocity (1000 ft/sec)",
)
plt_flight_path =
 plot(ts, rad2deg.(value.(\gamma)); legend = nothing, title = "Flight Path (deg)")
plt_azimuth =
 plot(ts, rad2deg.(value.(\u00c0)); legend = nothing, title = "Azimuth (deg)")
plot(
 plt altitude,
 plt_velocity,
 plt_longitude,
 plt_flight_path,
 plt_latitude,
 plt_azimuth;
 layout = grid(3, 2),
 linewidth = 2,
 size = (700, 700),
)
```



```
plt_attack_angle = plot(
 ts[1:end-1],
 rad2deg.(value.(a)[1:end-1]);
 legend = nothing,
 title = "Angle of Attack (deg)",
)
```

```
plt_bank_angle = plot(
 \texttt{ts[1:end-1],}
 rad2deg.(value.(\beta)[1:end-1]);
 legend = nothing,
 title = "Bank Angle (deg)",
)
plt_heating = plot(
 ts,
 q.(value.(scaled_h) * 1e5, value.(scaled_v) * 1e4, value.(\alpha));
 legend = nothing,
 title = "Heating (BTU/ft/ft/sec)",
)
plot(
 plt_attack_angle,
 plt_bank_angle,
 plt_heating;
 layout = grid(3, 1),
 linewidth = 2,
 size = (700, 700),
)
```




# Space Shuttle Reentry Trajectory

## **Chapter 8**

## **Conic programs**

## 8.1 Introduction

Conic programs are a class of convex nonlinear optimization problems which use cones to represent the nonlinearities. They have the form:

$$\min_{x \in \mathbb{R}^n} \qquad \qquad f_0(x) \tag{8.1}$$

$$f_j(x) \in \mathcal{S}_j \ j = 1 \dots m$$
 (8.2)

Mixed-integer conic programs (MICPs) are extensions of conic programs in which some (or all) of the decision variables take discrete values.

#### How to choose a solver

s.t.

JuMP supports a range of conic solvers, although support differs on what types of cones each solver supports. In the list of Supported solvers, "SOCP" denotes solvers supporting second-order cones and "SDP" denotes solvers supporting semidefinite cones. In addition, solvers such as SCS and Mosek have support for the exponential cone. Moreover, due to the bridging system in MathOptInterface, many of these solvers support a much wider range of exotic cones than they natively support. Solvers supporting discrete variables start with "(MI)" in the list of Supported solvers.

#### Tip

Duality plays a large role in solving conic optimization models. Depending on the solver, it can be more efficient to solve the dual instead of the primal. If performance is an issue, see the Dualization tutorial for more details.

## How these tutorials are structured

Having a high-level overview of how this part of the documentation is structured will help you know where to look for certain things.

The following tutorials are worked examples that present a problem in words, then formulate it in mathematics, and then solve it in JuMP. This usually involves some sort of visualization of the solution. Start here if you are new to JuMP.

- Example: experiment design
- Example: logistic regression
- The Modeling with cones tutorial contains a number of helpful reformulations and tricks you can use when modeling conic programs. Look here if you are stuck trying to formulate a problem as a conic program.
- The remaining tutorials are less verbose and styled in the form of short code examples. These tutorials have less explanation, but may contain useful code snippets, particularly if they are similar to a problem you are trying to solve.

## 8.2 Modeling with cones

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by Arpit Bhatia.

The purpose of this tutorial is to show how you can model various common problems using conic optimization.

#### Тір

A good resource for learning more about functions which can be modeled using cones is the MOSEK Modeling Cookbook.

#### **Required packages**

This tutorial uses the following packages:

using JuMP import Clarabel import LinearAlgebra import MathOptInterface as MOI

## **Background theory**

A subset C of a vector space V is a cone if  $\forall x \in C$  and positive scalars  $\lambda > 0$ , the product  $\lambda x \in C$ .

A cone C is a convex cone if  $\lambda x + (1 - \lambda)y \in C$ , for any  $\lambda \in [0, 1]$ , and any  $x, y \in C$ .

Conic programming problems are convex optimization problems in which a convex function is minimized over the intersection of an affine subspace and a convex cone. An example of a conic-form minimization problems, in the primal form is:

$$\min_{x \in \mathbb{R}^n} \quad a_0^T x + b_0$$
s.t.  $A_i x + b_i \in \mathcal{C}_i \quad i = 1 \dots m$ 

The corresponding dual problem is:

$$\begin{aligned} \max_{y_1,\ldots,y_m} & -\sum_{i=1}^m b_i^T y_i + b_0 \\ \text{s.t.} & a_0 - \sum_{i=1}^m A_i^T y_i = 0 \\ & y_i \in \mathcal{C}_i^* \quad i = 1 \dots m \end{aligned}$$

where each  $C_i$  is a closed convex cone and  $C_i^*$  is its dual cone.

## Second-Order Cone

The SecondOrderCone (or Lorentz Cone) of dimension n is a cone of the form:

$$K_{soc} = \{(t, x) \in \mathbb{R}^n : t \ge ||x||_2\}$$

It is most commonly used to represent the L2-norm of the vector x:

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, x[1:3])
@variable(model, t)
@constraint(model, sum(x) == 1)
@constraint(model, [t; x] in SecondOrderCone())
@objective(model, Min, t)
optimize!(model)
assert_is_solved_and_feasible(model)
value(t), value.(x)
```

(0.5773502650695171, [0.3333333333333368, 0.33333333333333333, 0.33333333333333])

## **Rotated Second-Order Cone**

A Second-Order Cone rotated by  $\pi/4$  in the  $(x_1, x_2)$  plane is called a RotatedSecondOrderCone. It is a cone of the form:

$$K_{rsoc} = \{(t, u, x) \in \mathbb{R}^n : 2tu \ge ||x||_2^2, t, u \ge 0\}$$

When u = 0.5, it represents the sum of squares of a vector x:

```
data = [1.0, 2.0, 3.0, 4.0]
target = [0.45, 1.04, 1.51, 1.97]
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, θ)
```

```
@variable(model, t)
@expression(model, residuals, 0 * data .- target)
@constraint(model, [t; 0.5; residuals] in RotatedSecondOrderCone())
@objective(model, Min, t)
optimize!(model)
assert_is_solved_and_feasible(model)
value(0), value(t)
```

(0.49799999999999994, 0.004979997069489959)

#### **Exponential Cone**

The MOI.ExponentialCone is a set of the form:

$$K_{exp} = \{(x, y, z) \in \mathbb{R}^3 : y \exp(x/y) \le z, y > 0\}$$

It can be used to model problems involving log and exp.

## Exponential

To model  $\exp(x) \leq z$ , use (x, 1, z):

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, x == 1.5)
@variable(model, z)
@objective(model, Min, z)
@constraint(model, [x, 1, z] in MOI.ExponentialCone())
optimize!(model)
assert_is_solved_and_feasible(model)
value(z), exp(1.5)
```

(4.481689066110043, 4.4816890703380645)

#### Logarithm

To model  $x \leq \log(z)$ , use (x, 1, z):

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, x)
@variable(model, z == 1.5)
@objective(model, Max, x)
@constraint(model, [x, 1, z] in MOI.ExponentialCone())
optimize!(model)
assert_is_solved_and_feasible(model)
value(x), log(1.5)
```

(0.4054651080062003, 0.4054651081081644)

## Log-sum-exp

To model  $t \ge \log(\sum e^{x_i})$ , use:

```
N = 3
x0 = rand(N)
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, x[i = 1:N] == x0[i])
@variable(model, t)
@objective(model, Min, t)
@variable(model, u[1:N])
@constraint(model, sum(u) <= 1)
@constraint(model, [i = 1:N], [x[i] - t, 1, u[i]] in MOI.ExponentialCone())
optimize!(model)
value(t), log(sum(exp.(x0)))</pre>
```

(1.4727722754754033, 1.472772274699489)

#### Entropy

The entropy maximization problem consists of maximizing the entropy function,  $H(x) = -x \log x$  subject to linear inequality constraints.

$$\max \quad -\sum_{i=1}^{n} x_i \log x_i$$
  
s.t. 
$$\mathbf{1}^{\top} x = 1$$
$$Ax \le b$$

We can model this problem using an exponential cone by using the following transformation:

$$t \leq -x \log x \iff t \leq x \log(1/x) \iff (t, x, 1) \in K_{exp}$$

Thus, our problem becomes,

$$\begin{array}{ll} \max & 1^T t \\ \text{s.t.} & Ax \leq b \\ & 1^T x = 1 \\ & (t_i, x_i, 1) \in K_{exp} \quad \forall i = 1 \dots n \end{array}$$

```
m, n = 10, 15
A, b = randn(m, n), rand(m, 1)
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, t[1:n])
@variable(model, x[1:n])
@objective(model, Max, sum(t))
@constraint(model, Max, sum(t))
@constraint(model, sum(x) == 1)
@constraint(model, sum(x) == 1)
@constraint(model, A * x .<= b)
@constraint(model, A * x .<= b)
@constraint(model, [i = 1:n], [t[i], x[i], 1] in MOI.ExponentialCone())
optimize!(model)
assert_is_solved_and_feasible(model)
objective_value(model)
```

2.6631503839586976

The MOI.ExponentialCone has a dual, the MOI.DualExponentialCone, that offers an alternative formulation that can be more efficient for some formulations.

There is also the MOI.RelativeEntropyCone for explicitly encoding the relative entropy function

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, t)
@variable(model, x[1:n])
@objective(model, Max, -t)
@constraint(model, sum(x) == 1)
@constraint(model, A * x .<= b)
@constraint(model, [t; ones(n); x] in MOI.RelativeEntropyCone(2n + 1))
optimize!(model)
assert_is_solved_and_feasible(model)
objective_value(model)</pre>
```

2.6631503843096125

#### **PowerCone**

The MOI.PowerCone is a three-dimensional set parameterized by a scalar value  $\alpha$ . It has the form:

$$K_n = \{(x, y, z) \in \mathbb{R}^3 : x^{\alpha} y^{1-\alpha} > |z|, x > 0, y > 0\}$$

The power cone permits a number of reformulations. For example, when p > 1, we can model  $t \ge x^p$  using the power cone (t, 1, x) with  $\alpha = 1/p$ . Thus, to model  $t \ge x^3$  with  $x \ge 0$ 

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, t)
@variable(model, x >= 1.5)
@constraint(model, [t, 1, x] in MOI.PowerCone(1 / 3))
@objective(model, Min, t)
optimize!(model)
assert_is_solved_and_feasible(model)
value(t), value(x)
```

(3.3749999041156307, 1.499999996575596)

The MOI.PowerCone has a dual, the MOI.DualPowerCone, that offers an alternative formulation that can be more efficient for some formulations.

#### **P-Norm**

The p-norm  $||x||_p = \left(\sum_i |x_i|^p\right)^{rac{1}{p}}$  can be modeled using MOI. PowerCones. See the Mosek Modeling Cookbook for the derivation.

```
function p_norm(x::Vector, p)
 N = length(x)
 model = Model(Clarabel.Optimizer)
 set_silent(model)
 @variable(model, r[1:N])
 @variable(model, t)
 @constraint(model, [i = 1:N], [r[i], t, x[i]] in MOI.PowerCone(1 / p))
 @constraint(model, sum(r) == t)
 @objective(model, Min, t)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 return value(t)
end
x = rand(5);
LinearAlgebra.norm(x, 4), p_norm(x, 4)
```

(0.9316922467512209, 0.9316922539087699)

#### **Positive Semidefinite Cone**

The set of positive semidefinite matrices (PSD) of dimension n form a cone in  $\mathbb{R}^n$ . We write this set mathematically as:

$$\mathcal{S}^n_+ = \{ X \in \mathcal{S}^n \mid z^T X z \ge 0, \, \forall z \in \mathbb{R}^n \}.$$

A PSD cone is represented in JuMP using the MOI sets PositiveSemidefiniteConeTriangle (for upper triangle of a PSD matrix) and PositiveSemidefiniteConeSquare (for a complete PSD matrix). However, it is preferable to use the PSDCone shortcut as illustrated below.

## Example: largest eigenvalue of a symmetric matrix

Suppose A has eigenvalues  $\lambda_1 \geq \lambda_2 \ldots \geq \lambda_n$ . Then the matrix tI - A has eigenvalues  $t - \lambda_1, t - \lambda_2, \ldots, t - \lambda_n$ . Note that tI - A is PSD exactly when all these eigenvalues are non-negative, and this happens for values  $t \geq \lambda_1$ . Thus, we can model the problem of finding the largest eigenvalue of a symmetric matrix as:

```
\lambda_1 = \min t
s.t. tI - A \succeq 0
```

```
A = [3 2 4; 2 0 2; 4 2 3]
I = Matrix{Float64}(LinearAlgebra.I, 3, 3)
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, t)
@objective(model, Min, t)
@constraint(model, t .* I - A in PSDCone())
optimize!(model)
assert_is_solved_and_feasible(model)
objective_value(model)
```

7.999999998730902

#### GeometricMeanCone

The MOI.GeometricMeanCone is a cone of the form:

$$K_{geo} = \{(t, x) \in \mathbb{R}^n : x \ge 0, t \le \sqrt[n-1]{x_1 x_2 \cdots x_{n-1}}\}$$

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, x[1:4])
@variable(model, t)
@constraint(model, sum(x) == 1)
@constraint(model, [t; x] in MOI.GeometricMeanCone(5))
optimize!(model)
value(t), value.(x)
```

 $(0.0, \ [0.15913987260919943, \ 0.2007683986913607, \ 0.320045863413361, \ 0.3200458634133662])$ 

## RootDetCone

The MOI.RootDetConeSquare is a cone of the form:

$$K = \{ (t, X) \in \mathbb{R}^{1+d^2} : t \le \det(X)^{\frac{1}{d}} \}$$

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, t)
@variable(model, X[1:2, 1:2])
@objective(model, Max, t)
@constraint(model, [t; vec(X)] in MOI.RootDetConeSquare(2))
@constraint(model, X .== [2 1; 1 3])
optimize!(model)
assert_is_solved_and_feasible(model)
value(t), sqrt(LinearAlgebra.det(value.(X)))
```

(2.236067936397333, 2.236067965376734)

If X is symmetric, then you can use MOI.RootDetConeTriangle instead. This can be more efficient because the solver does not need to add additional constraints to ensure X is symmetric.

When forming the function, use triangle\_vec to obtain the column-wise upper triangle of the matrix as a vector in the order that JuMP requires.

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, t)
@variable(model, X[1:2, 1:2], Symmetric)
@objective(model, Max, t)
@constraint(model, [t; triangle_vec(X)] in MOI.RootDetConeTriangle(2))
@constraint(model, X .== [2 1; 1 3])
optimize!(model)
value(t), sqrt(LinearAlgebra.det(value.(X)))
```

(2.2360679358702282, 2.236067965376757)

#### LogDetCone

The MOI.LogDetConeSquare is a cone of the form:

$$K = \{(t, u, X) \in \mathbb{R}^{2+d^2} : t \le u \log(\det(X/u))\}$$

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, t)
@variable(model, u)
@variable(model, X[1:2, 1:2])
@objective(model, Max, t)
@constraint(model, [t; u; vec(X)] in MOI.LogDetConeSquare(2))
@constraint(model, X .== [2 1; 1 3])
@constraint(model, u == 0.5)
optimize!(model)
assert_is_solved_and_feasible(model)
value(t), 0.5 * log(LinearAlgebra.det(value.(X) ./ 0.5))
```

(1.4978661147944674, 1.4978661335845644)

If X is symmetric, then you can use MOI.LogDetConeTriangle instead. This can be more efficient because the solver does not need to add additional constraints to ensure X is symmetric.

When forming the function, use triangle\_vec to obtain the column-wise upper triangle of the matrix as a vector in the order that JuMP requires.

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, t)
@variable(model, u)
@variable(model, X[1:2, 1:2], Symmetric)
@objective(model, Max, t)
@constraint(model, [t; u; triangle_vec(X)] in MOI.LogDetConeTriangle(2))
@constraint(model, X .== [2 1; 1 3])
@constraint(model, u == 0.5)
optimize!(model)
assert_is_solved_and_feasible(model)
value(t), 0.5 * log(LinearAlgebra.det(value.(X) ./ 0.5))
```

(1.4978661136972233, 1.497866133584564)

#### NormNuclearCone

The MOI.NormNuclearCone is a cone of the form:

$$K = \{(t, X) \in \mathbb{R}^{1+m \cdot n} : t \ge \sum_i \sigma_i(X)\}$$

where  $\sigma_i$  is the *i* singular value of X.

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, t)
@variable(model, X[1:2, 1:3])
@objective(model, Min, t)
@constraint(model, [t; vec(X)] in MOI.NormNuclearCone(2, 3))
@constraint(model, X .== [1 2 3; 4 5 6])
optimize!(model)
assert_is_solved_and_feasible(model)
value(t), sum(LinearAlgebra.svdvals(value.(X)))
```

(10.280901634062255, 10.280901636369206)

## NormSpectralCone

The MOI.NormSpectralCone is a cone of the form:

$$K = \{(t, X) \in \mathbb{R}^{1+m \cdot n} : t \ge \max_i \sigma_i(X)\}$$

where  $\sigma_i$  is the i singular value of X.

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, t)
@variable(model, X[1:2, 1:3])
@objective(model, Min, t)
@constraint(model, [t; vec(X)] in MOI.NormSpectralCone(2, 3))
@constraint(model, X .== [1 2 3; 4 5 6])
optimize!(model)
assert_is_solved_and_feasible(model)
value(t), maximum(LinearAlgebra.svdvals(value.(X)))
```

(9.508031944583905, 9.508032000695724)

#### **Other Cones and Functions**

For other cones supported by JuMP, check out the MathOptInterface Manual.

## 8.3 Dualization

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to explain how to use Dualization.jl to improve the performance of some conic optimization models.

There are two important takeaways:

- 1. JuMP reformulates problems to meet the input requirements of the solver, potentially increasing the problem size by adding slack variables and constraints.
- 2. Solving the dual of a conic model can be more efficient than solving the primal.

Dualization.jl is a package which fixes these problems, allowing you to solve the dual instead of the primal with a one-line change to your code.

## **Required packages**

This tutorial uses the following packages:

using JuMP import Dualization import LinearAlgebra import SCS

#### Background

Conic optimization solvers typically accept one of two input formulations.

The first is the *standard* conic form:

$$\min_{x \in \mathbb{R}^n} c^\top x \tag{8.3}$$

s.t. 
$$Ax = b$$
 (8.4)

$$x \in \mathcal{K}$$
 (8.5)

in which we have a set of linear equality constraints Ax = b and the variables belong to a cone  $\mathcal{K}$ .

The second is the *geometric* conic form:

$$\min_{x \in \mathbb{R}^n} c^\top x \tag{8.6}$$

s.t. 
$$Ax - b \in \mathcal{K}$$
 (8.7)

in which an affine function Ax - b belongs to a cone  $\mathcal{K}$  and the variables are free.

It is trivial to convert between these two representations, for example, to go from the geometric conic form to the standard conic form we introduce slack variables y:

$$\min_{x \in \mathbb{R}^n} c^\top x \tag{8.8}$$

s.t. 
$$\begin{bmatrix} A & -I \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = b$$
 (8.9)

$$\begin{bmatrix} x \\ y \end{bmatrix} \in \mathbb{R}^n \times \mathcal{K} \tag{8.10}$$

and to go from the standard conic form to the geometric conic form, we can rewrite the equality constraint as a function belonging to the  $\{0\}$  cone:

$$\min_{x \in \mathbb{R}^n} c^\top x \tag{8.11}$$

s.t. 
$$\begin{bmatrix} A \\ I \end{bmatrix} x - \begin{bmatrix} b \\ 0 \end{bmatrix} \in \{0\} \times \mathcal{K}$$
 (8.12)

From a theoretical perspective, the two formulations are equivalent, and if you implement a model in the standard conic form and pass it to a geometric conic form solver (or vice versa), then JuMP will automatically reformulate the problem into the correct formulation.

From a practical perspective though, the reformulations are problematic because the additional slack variables and constraints can make the problem much larger and therefore harder to solve.

You should also note many problems contain a mix of conic constraints and variables, and so they do not neatly fall into one of the two formulations. In these cases, JuMP reformulates only the variables and constraints as necessary to convert the problem into the desired form.

#### Primal and dual formulations

Duality plays a large role in conic optimization. For a detailed description of conic duality, see Duality.

A useful observation is that if the primal problem is in standard conic form, then the dual problem is in geometric conic form, and vice versa. Moreover, the primal and dual may have a different number of variables and constraints, although which one is smaller depends on the problem. Therefore, instead of reformulating the problem from one form to the other, it can be more efficient to solve the dual instead of the primal.

To demonstrate, we use a variation of the Maximum cut via SDP example.

The primal formulation (in standard conic form) is:

```
model_primal = Model()
@variable(model_primal, X[1:2, 1:2], PSD)
@objective(model_primal, Max, sum([1 -1; -1 1] .* X))
@constraint(model_primal, primal_c[i = 1:2], 1 - X[i, i] == 0)
print(model_primal)
```

```
Max X[1,1] - 2 X[1,2] + X[2,2]
Subject to
primal_c[1] : -X[1,1] = -1
primal_c[2] : -X[2,2] = -1
[X[1,1] X[1,2]
... X[2,2]] \epsilon PSDCone()
```

This problem has three scalar decision variables (the matrix X is symmetric), two scalar equality constraints, and a constraint that X is positive semidefinite.

The dual of model\_primal is:

```
model_dual = Model()
@variable(model_dual, y[1:2])
@objective(model_dual, Min, sum(y))
@constraint(
 model_dual,
 dual_c,
 LinearAlgebra.Symmetric([y[1]-1 1; 1 y[2]-1]) in PSDCone(),
)
print(model_dual)
```

Min y[1] + y[2]
Subject to
dual\_c : [y[1] - 1 1
... y[2] - 1] ∈ PSDCone()

This problem has two scalar decision variables, and a 2x2 positive semidefinite matrix constraint.

## Тір

If you haven't seen conic duality before, try deriving the dual problem based on the description in Duality. You'll need to know that the dual cone of PSDCone is the PSDCone.

When we solve model\_primal with SCS.Optimizer, SCS reports three variables (variables n: 3), five rows in the constraint matrix (constraints m: 5), and five non-zeros in the matrix (nnz(A): 5):

```
set_optimizer(model_primal, SCS.Optimizer)
optimize!(model_primal)
assert_is_solved_and_feasible(model_primal; dual = true)
```

```

^^I
 SCS v3.2.7 - Splitting Conic Solver
^^I(c) Brendan O'Donoghue, Stanford University, 2012

problem: variables n: 3, constraints m: 5
cones: ^^I z: primal zero / dual free vars: 2
^^I s: psd vars: 3, ssize: 1
settings: eps_abs: 1.0e-04, eps_rel: 1.0e-04, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive_scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration lookback: 10, acceleration interval: 10
^^I compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-qdldl
^^I nnz(A): 5, nnz(P): 0

iter | pri res | dua res | gap | obj | scale | time (s)

 0| 1.65e+01 1.60e-01 5.09e+01 -2.91e+01 1.00e-01 1.02e-04
 50| 1.74e-08 2.70e-10 4.88e-08 -4.00e+00 1.00e-01 1.78e-04

```

```
status: solved
timings: total: 1.79e-04s = setup: 4.51e-05s + solve: 1.34e-04s
^^I lin-sys: 1.13e-05s, cones: 5.77e-05s, accel: 2.60e-06s
objective = -4.000000
```

(There are five rows in the constraint matrix because SCS expects problems in geometric conic form, and so JuMP has reformulated the X, PSD variable constraint into the affine constraint X .+ 0 in PSDCone().)

The solution we obtain is:

value(X)

```
2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
 1.0 -1.0
 -1.0 1.0
```

dual(VariableInSetRef(X))

```
2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
1.0 1.0
1.0 1.0
```

dual.(primal\_c)

2-element Vector{Float64}:
1.9999999997299085
1.9999999997299085

objective\_value(model\_primal)

3.9999999506359716

When we solve model\_dual with SCS.Optimizer, SCS reports two variables (variables n: 2), three rows in the constraint matrix (constraints m: 3), and two non-zeros in the matrix (nz(A): 2):

```
set_optimizer(model_dual, SCS.Optimizer)
optimize!(model_dual)
assert_is_solved_and_feasible(model_dual; dual = true)
```

```

^^I
 SCS v3.2.7 - Splitting Conic Solver
^^I(c) Brendan O'Donoghue, Stanford University, 2012
.....
problem: variables n: 2, constraints m: 3
cones: ^^I s: psd vars: 3, ssize: 1
settings: eps_abs: 1.0e-04, eps_rel: 1.0e-04, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive_scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration_lookback: 10, acceleration_interval: 10
^^I compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-qdldl
^^I nnz(A): 2, nnz(P): 0

iter | pri res | dua res | gap | obj | scale | time (s)

 0| 1.23e+01 1.00e+00 2.73e+01 -9.03e+00 1.00e-01 6.45e-05
 50| 1.13e-07 1.05e-09 3.23e-07 4.00e+00 1.00e-01 1.39e-04

status: solved
timings: total: 1.40e-04s = setup: 2.89e-05s + solve: 1.11e-04s
^^I lin-sys: 8.93e-06s, cones: 4.83e-05s, accel: 2.62e-06s

objective = 4.000000
```

and the solution we obtain is:

dual(dual\_c)

2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
 1.0 -1.0
 -1.0 1.0

value(dual\_c)

2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
1.0 1.0
1.0 1.0

value.(y)

2-element Vector{Float64}:
 2.000000159272681
 2.000000159272681

objective\_value(model\_dual)

4.00000318545362

This particular problem is small enough that it isn't meaningful to compare the solve times, but in general, we should expect model\_dual to solve faster than model\_primal because it contains fewer variables and constraints. The difference is particularly noticeable on large-scale optimization problems.

## dual\_optimizer

Manually deriving the conic dual is difficult and error-prone. The package Dualization.jl provides the Dualization.dual\_optimize meta-solver, which wraps any MathOptInterface-compatible solver in an interface that automatically formulates and solves the dual of an input problem.

To demonstrate, we use Dualization.dual\_optimizer to solve model\_primal:

```
set_optimizer(model_primal, Dualization.dual_optimizer(SCS.Optimizer))
optimize!(model_primal)
assert_is_solved_and_feasible(model_primal; dual = true)
```

```

^^I
 SCS v3.2.7 - Splitting Conic Solver
^^I(c) Brendan O'Donoghue, Stanford University, 2012

problem: variables n: 2, constraints m: 3
cones: ^^I s: psd vars: 3, ssize: 1
settings: eps_abs: 1.0e-04, eps_rel: 1.0e-04, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive_scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration_lookback: 10, acceleration_interval: 10
^^I compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-gdldl
^^I nnz(A): 2, nnz(P): 0
.....
iter | pri res | dua res | gap | obj | scale | time (s)

 0| 1.23e+01 1.00e+00 2.73e+01 -9.03e+00 1.00e-01 6.57e-05
 50| 1.13e-07 1.05e-09 3.23e-07 4.00e+00 1.00e-01 1.40e-04
status: solved
```

```
timings: total: 1.40e-04s = setup: 2.93e-05s + solve: 1.11e-04s
^^I lin-sys: 8.91e-06s, cones: 4.83e-05s, accel: 2.59e-06s
objective = 4.000000
```

The performance is the same as if we solved model\_dual, and the correct solution is returned to X:

value(X)

```
2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
 1.0 -1.0
 -1.0 1.0
```

dual(VariableInSetRef(X))

```
2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
1.0 1.0
1.0 1.0
```

dual.(primal\_c)

```
2-element Vector{Float64}:
 2.000000159272681
 2.000000159272681
```

Moreover, if we use dual\_optimizer on model\_dual, then we get the same performance as if we had solved model\_primal:

```
set_optimizer(model_dual, Dualization.dual_optimizer(SCS.Optimizer))
optimize!(model_dual)
assert_is_solved_and_feasible(model_dual; dual = true)
```

^^I SCS v3.2.7 - Splitting Conic Solver ^^I(c) Brendan O'Donoghue, Stanford University, 2012 problem: variables n: 3, constraints m: 5 cones: ^^I z: primal zero / dual free vars: 2 ^^I s: psd vars: 3, ssize: 1

```
settings: eps_abs: 1.0e-04, eps_rel: 1.0e-04, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive_scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration lookback: 10, acceleration interval: 10
^^I compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-qdldl
^^I nnz(A): 5, nnz(P): 0

iter | pri res | dua res | gap | obj | scale | time (s)

 0| 1.65e+01 1.60e-01 5.09e+01 -2.91e+01 1.00e-01 6.87e-05
 50| 1.74e-08 2.70e-10 4.88e-08 -4.00e+00 1.00e-01 1.43e-04
.....
status: solved
timings: total: 1.44e-04s = setup: 2.98e-05s + solve: 1.15e-04s
^^I lin-sys: 1.04e-05s, cones: 4.86e-05s, accel: 2.49e-06s
.....
objective = -4.000000

```

dual(dual\_c)

```
2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
 1.0 -1.0
 -1.0 1.0
```

value(dual\_c)

```
2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
1.0 1.0
1.0 1.0
```

value.(y)

```
2-element Vector{Float64}:
1.9999999997299085
1.9999999997299085
```

## A mixed example

The Maximum cut via SDP example is nicely defined because the primal is in standard conic form and the dual is in geometric conic form. However, many practical models contain a mix of the two formulations. One example is The minimum distortion problem:

```
D = [0 1 1 1; 1 0 2 2; 1 2 0 2; 1 2 2 0]
model = Model()
@variable(model, c²)
@variable(model, Q[1:4, 1:4], PSD)
@objective(model, Min, c²)
for i in 1:4, j in (i+1):4
 @constraint(model, D[i, j]^2 <= Q[i, i] + Q[j, j] - 2 * Q[i, j])
 @constraint(model, Q[i, i] + Q[j, j] - 2 * Q[i, j] <= c² * D[i, j]^2)
end
@constraint(model, Q[1, 1] == 0)
@constraint(model, c² >= 1)
```

```
c^2 \geq 1
```

In this formulation, the Q variable is of the form  $x \in \mathcal{K}$ , but there is also a free variable,  $c^2$ , a linear equality constraint, Q[1, 1] == 0, and some linear inequality constraints. Rather than attempting to derive the formulation that JuMP would pass to SCS and its dual, the simplest solution is to try solving the problem with and without dual\_optimizer to see which formulation is most efficient.

set\_optimizer(model, SCS.Optimizer)
optimize!(model)

```
^^I
 SCS v3.2.7 - Splitting Conic Solver
^^I(c) Brendan O'Donoghue, Stanford University, 2012

problem: variables n: 11, constraints m: 24
cones: ^^I z: primal zero / dual free vars: 1
^^I l: linear vars: 13
^^I s: psd vars: 10, ssize: 1
settings: eps_abs: 1.0e-04, eps_rel: 1.0e-04, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive_scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration_lookback: 10, acceleration_interval: 10
^^I compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-qdldl
^^I nnz(A): 54, nnz(P): 0

iter | pri res | dua res | gap | obj | scale | time (s)
.....
 0| 4.73e+00 1.00e+00 2.92e+00 1.23e+00 1.00e-01 1.28e-04
 150| 1.01e-04 3.07e-05 6.08e-05 1.33e+00 1.00e-01 7.00e-04

status: solved
timings: total: 7.01e-04s = setup: 6.30e-05s + solve: 6.38e-04s
^^I lin-sys: 1.03e-04s, cones: 3.88e-04s, accel: 3.54e-05s

objective = 1.333363

```

set\_optimizer(model, Dualization.dual\_optimizer(SCS.Optimizer))
optimize!(model)

```

^^I
 SCS v3.2.7 - Splitting Conic Solver
^^I(c) Brendan O'Donoghue, Stanford University, 2012

problem: variables n: 14, constraints m: 24
cones: ^^I z: primal zero / dual free vars: 1
^^I l: linear vars: 13
^^I s: psd vars: 10, ssize: 1
settings: eps_abs: 1.0e-04, eps_rel: 1.0e-04, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration_lookback: 10, acceleration_interval: 10
^^I compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-qdldl
^^I nnz(A): 57, nnz(P): 0
iter | pri res | dua res | gap | obj | scale | time (s)

 0| 3.71e+01 1.48e+00 2.23e+02 -1.13e+02 1.00e-01 9.77e-05
 150| 1.57e-04 2.28e-05 2.08e-04 -1.33e+00 1.00e-01 6.98e-04

status: solved
timings: total: 6.99e-04s = setup: 7.46e-05s + solve: 6.24e-04s
^^I lin-sys: 1.21e-04s, cones: 4.01e-04s, accel: 2.04e-05s

objective = -1.333460

```

For this problem, SCS reports that the primal has variables n: 11, constraints m: 24 and that the dual has variables n: 14, constraints m: 24. Therefore, we should probably use the primal formulation because it has fewer variables and the same number of constraints.

## When to use dual\_optimizer

Because it can make the problem larger or smaller, depending on the problem and the choice of solver, there is no definitive rule on when you should use dual\_optimizer. However, you should try dual\_optimizer if your conic optimization problem takes a long time to solve, or if you need to repeatedly solve similarly structured problems with different data. In some cases solving the dual instead of the primal can make a large difference.

#### 8.4 Arbitrary precision arithmetic

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to explain how to use a solver which supports arithmetic using a number type other than Float64.

#### **Required packages**

This tutorial uses the following packages:

using JuMP
import CDDLib
import Clarabel

## **Higher-precision arithmetic**

To create a model with a number type other than Float64, use GenericModel with an optimizer which supports the same number type:

model = GenericModel{BigFloat}(Clarabel.Optimizer{BigFloat})

```
A JuMP Model
| value_type: BigFloat
| solver: Clarabel
| objective_sense: FEASIBILITY_SENSE
| num_variables: 0
| num_constraints: 0
L Names registered in the model: none
```

The syntax for adding decision variables is the same as a normal JuMP model, except that values are converted to BigFloat:

@variable(model, -1 <= x[1:2] <= sqrt(big"2"))</pre>

```
2-element Vector{GenericVariableRef{BigFloat}}:
 x[1]
 x[2]
```

Note that each x is now a GenericVariableRef{BigFloat}, which means that the value of x in a solution will be a BigFloat.

The lower and upper bounds of the decision variables are also BigFloat:

lower\_bound(x[1])

-1.0

typeof(lower\_bound(x[1]))

BigFloat

upper\_bound(x[2])

1.414213562373095048801688724209698078569671875376948073176679737990732478462102

typeof(upper\_bound(x[2]))

BigFloat

The syntax for adding constraints is the same as a normal JuMP model, except that coefficients are converted to BigFloat:

@constraint(model, c, x[1] == big"2" \* x[2])

 $x_1 - 2.0x_2 = 0.0$ 

The function is a GenericAffExpr with BigFloat for the coefficient and variable types;

constraint = constraint\_object(c)
typeof(constraint.func)

GenericAffExpr{BigFloat, GenericVariableRef{BigFloat}}

and the set is a MOI.EqualTo{BigFloat}:

typeof(constraint.set)

MathOptInterface.EqualTo{BigFloat}

The syntax for adding and objective is the same as a normal JuMP model, except that coefficients are converted to BigFloat:

@objective(model, Min, 3x[1]^2 + 2x[2]^2 - x[1] - big"4" \* x[2])

 $3.0x_1^2 + 2.0x_2^2 - x_1 - 4.0x_2$ 

typeof(objective\_function(model))

GenericQuadExpr{BigFloat, GenericVariableRef{BigFloat}}

Here's the model we have built:

print(model)

```
Min 3.0 x[1]^2 + 2.0 x[2]^2 - x[1] - 4.0 x[2]

Subject to

c : x[1] - 2.0 x[2] = 0.0

x[1] \ge -1.0

x[2] \ge -1.0

x[1] \le 1.414213562373095048801688724209698078569671875376948073176679737990732478462102

x[2] \le 1.414213562373095048801688724209698078569671875376948073176679737990732478462102
```

Let's solve and inspect the solution:

```
optimize!(model)
assert_is_solved_and_feasible(model; dual = true)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : Clarabel
| Termination
| | termination_status : OPTIMAL
| | result_count : 1
| L raw_status : SOLVED
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | dual_objective_value : -6.42857e-01
| L dual_objective_value : -6.42857e-01
| Work counters
| solve_time (sec) : 1.47647e-03
| barrier_iterations : 5
```

The value of each decision variable is a BigFloat:

value.(x)

2-element Vector{BigFloat}: 0.4285714246558161076147072906813123533593766450416896337912086518811186790735189 0.2142857123279078924828007272730108809297577877991360649674411645247653239673801 as well as other solution attributes like the objective value:

objective\_value(model)

#### -0.6428571428571422964607590389935242587959291815638830868454759876473734138856053

and dual solution:

dual(c)

1.571428571977140845343978069015092190548250919787945065022059071052557047888032

This problem has an analytic solution of x = [3//7, 3//14]. Currently, our solution has an error of approximately 1e-9:

value.(x) .- [3 // 7, 3 // 14]

```
2-element Vector{BigFloat}:
-3.915612463813864137890116218069194783529738937637362776690309892355053792476207e-09
-1.957806393231484987012703404784527926486578220746844549760948961746906215408591e-09
```

But by reducing the tolerances, we can obtain a more accurate solution:

```
set_attribute(model, "tol_gap_abs", 1e-32)
set_attribute(model, "tol_gap_rel", 1e-32)
optimize!(model)
assert_is_solved_and_feasible(model)
value.(x) .- [3 // 7, 3 // 14]
```

2-element Vector{BigFloat}:
 -4.120732596246374574619292889406407106157605546563218305172773512099467866195165e-32
 -7.146646610782677659152301436088423235900252780211057986251367981130623553333357e-32

## **Rational arithmetic**

In addition to higher-precision floating point number types like BigFloat, JuMP also supports solvers with exact rational arithmetic. One example is CDDLib.jl, which supports the Rational{BigInt} number type:

model = GenericModel{Rational{BigInt}}(CDDLib.Optimizer{Rational{BigInt}})

```
A JuMP Model
| value_type: Rational{BigInt}
| solver: CDD
| objective_sense: FEASIBILITY_SENSE
| num_variables: 0
| num_constraints: 0
| Names registered in the model: none
```

As before, we can create variables using rational bounds:

@variable(model, 1 // 7 <= x[1:2] <= 2 // 3)</pre>

2-element Vector{GenericVariableRef{Rational{BigInt}}:
 x[1]
 x[2]

 $lower_bound(x[1])$ 

1//7

typeof(lower\_bound(x[1]))

Rational{BigInt}

As well as constraints:

@constraint(model, c1, (2 // 1) \* x[1] + x[2] <= 1)</pre>

 $2//1x_1 + x_2 \le 1//1$ 

@constraint(model, c2, x[1] + 3x[2] <= 9 // 4)</pre>

$$x_1 + 3//1x_2 \le 9//4$$

and objective functions:

@objective(model, Max, sum(x))

 $x_1 + x_2$ 

Here's the model we have built:

print(model)

```
Max x[1] + x[2]

Subject to

c1 : 2//1 x[1] + x[2] \le 1//1

c2 : x[1] + 3//1 x[2] \le 9//4

x[1] \ge 1//7

x[2] \ge 1//7

x[1] \le 2//3

x[2] \le 2//3
```

Let's solve and inspect the solution:

optimize!(model)
assert\_is\_solved\_and\_feasible(model)
solution\_summary(model)

```
solution_summary(; result = 1, verbose = false)
| solver_name : CDD
| Termination
| | termination_status : OPTIMAL
| | result_count : 1
| L raw_status : Optimal
L Solution (result = 1)
| primal_status : FEASIBLE_POINT
| dual_status : NO_SOLUTION
L objective_value : 5//6
```

The optimal values are given in exact rational arithmetic:

value.(x)

2-element Vector{Rational{BigInt}}:
1//6
2//3

objective_value(model)	
5//6	
value(c2)	
13//6	

## 8.5 Primal and dual warm-starts

This tutorial was generated using Literate.jl. Download the source as a .jl file.

Some conic solvers have the ability to set warm-starts for the primal and dual solution. This can improve performance, particularly if you are repeatedly solving a sequence of related problems.

The purpose of this tutorial is to demonstrate how to write a function that sets the primal and dual starts as the optimal solution stored in a model. It is intended to be a starting point for which you can modify if you want to do something similar in your own code.



See set\_start\_values for a generic implementation of this function that was added to JuMP after this tutorial was written.

#### **Required packages**

This tutorial uses the following packages:

using JuMP import SCS

## A basic function

The main component of this tutorial is the following function. The most important observation is that we cache all of the solution values first, and then we modify the model second. (Alternating between querying a value and modifying the model is not allowed in JuMP.)

```
function set_optimal_start_values(model::Model)
Store a mapping of the variable primal solution
variable_primal = Dict(x => value(x) for x in all_variables(model))
In the following, we loop through every constraint and store a mapping
from the constraint index to a tuple containing the primal and dual
solutions.
constraint_solution = Dict()
```

```
for (F, S) in list_of_constraint_types(model)
 # We add a try-catch here because some constraint types might not
 # support getting the primal or dual solution.
 try
 for ci in all constraints(model, F, S)
 constraint_solution[ci] = (value(ci), dual(ci))
 end
 catch
 @info("Something went wrong getting $F-in-$S. Skipping")
 end
 end
 # Now we can loop through our cached solutions and set the starting values.
 for (x, primal_start) in variable_primal
 set_start_value(x, primal_start)
 end
 for (ci, (primal_start, dual_start)) in constraint_solution
 set start_value(ci, primal start)
 set_dual_start_value(ci, dual_start)
 end
 return
end
```

```
set_optimal_start_values (generic function with 1 method)
```

#### **Testing the function**

To test our function, we use the following linear program:

```
model = Model(SCS.Optimizer)
@variable(model, x[1:3] >= 0)
@constraint(model, sum(x) <= 1)
@objective(model, Max, sum(i * x[i] for i in 1:3))
optimize!(model)
assert_is_solved_and_feasible(model)</pre>
```

```

^^I
 SCS v3.2.7 - Splitting Conic Solver
^^I(c) Brendan O'Donoghue, Stanford University, 2012
.....
problem: variables n: 3, constraints m: 4
cones: ^^I l: linear vars: 4
settings: eps_abs: 1.0e-04, eps_rel: 1.0e-04, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive_scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration_lookback: 10, acceleration_interval: 10
\ensuremath{^{\mbox{n}}}\xspace compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-qdldl
^^I nnz(A): 6, nnz(P): 0
.
 iter | pri res | dua res | gap | obj | scale | time (s)
```

0| 4.42e+01 1.00e+00 1.28e+02 -6.64e+01 1.00e-01 8.97e-05 75| 5.30e-07 2.63e-06 3.15e-07 -3.00e+00 1.00e-01 1.68e-04 status: solved timings: total: 1.69e-04s = setup: 4.18e-05s + solve: 1.27e-04s ^^I lin-sys: 1.47e-05s, cones: 7.31e-06s, accel: 3.68e-06s objective = -2.999998

By looking at the log, we can see that SCS took 75 iterations to find the optimal solution. Now we set the optimal solution as our starting point:

```
set_optimal_start_values(model)
```

and we re-optimize:

optimize!(model)

```

^^T
 SCS v3.2.7 - Splitting Conic Solver
^^I(c) Brendan O'Donoghue, Stanford University, 2012

problem: variables n: 3, constraints m: 4
cones: ^^I l: linear vars: 4
settings: eps_abs: 1.0e-04, eps_rel: 1.0e-04, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive_scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration_lookback: 10, acceleration_interval: 10
^{I} compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-qdldl
^^I nnz(A): 6, nnz(P): 0

iter | pri res | dua res | gap | obj | scale | time (s)

 0| 1.90e-05 1.56e-06 9.14e-05 -3.00e+00 1.00e-01 8.30e-05

status: solved
timings: total: 8.39e-05s = setup: 4.01e-05s + solve: 4.37e-05s
^^I lin-sys: 7.92e-07s, cones: 1.83e-06s, accel: 3.00e-08s

objective = -3.000044

```

Now the optimization terminates after 0 iterations because our starting point is already optimal.

#### Caveats

Some solvers do not support setting some parts of the starting solution, for example, they may support only set\_start\_value for variables.

If you encounter an UnsupportedSupported attribute error for MOI.VariablePrimalStart, MOI.ConstraintPrimalStart, or MOI.ConstraintDualStart, comment out the corresponding part of the set\_optimal\_start\_values function.

## 8.6 Simple semidefinite programming examples

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to provide a collection of examples of small conic programs from the field of semidefinite programming (SDP).

## **Required packages**

This tutorial uses the following packages:

using JuMP import Clarabel import LinearAlgebra import Plots import Random import Test

## Maximum cut via SDP

The maximum cut problem is a classical example in graph theory, where we seek to partition a graph into two complementary sets, such that the weight of edges between the two sets is maximized. This problem is NP-hard, but it is possible to obtain an approximate solution using the semidefinite programming relaxation:

```
\begin{array}{ll} \max & 0.25L \bullet X \\ \text{s.t.} & \mathrm{diag}(X) = e \\ & X \succeq 0 \end{array}
```

where L is the weighted graph Laplacian and e is a vector of ones. For more details, see (Goemans and Williamson, 1995).

```
"""
 svd_cholesky(X::AbstractMatrix, rtol)
Return the matrix `U` of the Cholesky decomposition of `X` as `U' * U`.
Note that we do not use the `LinearAlgebra.cholesky` function because it
requires the matrix to be positive definite while `X` may be only
positive *semi*definite.
We use the convention `U' * U` instead of `U * U'` to be consistent with
`LinearAlgebra.cholesky`.
"""
function svd_cholesky(X::AbstractMatrix)
 F = LinearAlgebra.svd(X)
 # We now have `X ≈ `F.U * D² * F.U'` where:
 D = LinearAlgebra.Diagonal(sqrt.(F.S))
 # So `X ≈ U' * U` where `U` is:
```

```
return (F.U * D)'
end
function solve_max_cut_sdp(weights)
 N = size(weights, 1)
 # Calculate the (weighted) Laplacian of the graph: L = D - W.
 L = LinearAlgebra.Diagonal(weights * ones(N)) - weights
 model = Model(Clarabel.Optimizer)
 set_silent(model)
 @variable(model, X[1:N, 1:N], PSD)
 for i in 1:N
 set_start_value(X[i, i], 1.0)
 end
 @objective(model, Max, 0.25 * LinearAlgebra.dot(L, X))
 @constraint(model, LinearAlgebra.diag(X) .== 1)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 V = svd_cholesky(value(X))
 Random.seed!(N)
 r = rand(N)
 r /= LinearAlgebra.norm(r)
 cut = [LinearAlgebra.dot(r, V[:, i]) > 0 for i in 1:N]
 S = findall(cut)
 T = findall(.!cut)
 println("Solution:")
 println(" (S, T) = ({", join(S, ", "), "}, {", join(T, ", "), "})")
 return S, T
end
```

solve\_max\_cut\_sdp (generic function with 1 method)

Given the graph

[1] --- 5 --- [2]

The solution is  $(S, T) = (\{1\}, \{2\})$ 

S, T = solve\_max\_cut\_sdp([0 5; 5 0])

([2], [1])

Given the graph

| \ | | \ | [3] --- [4]

The solution is  $(S, T) = (\{1\}, \{2, 3, 4\})$ 

S, T = solve\_max\_cut\_sdp([0 5 7 6; 5 0 0 1; 7 0 0 1; 6 1 1 0])

([1], [2, 3, 4])

Given the graph

[1] --- 1 --- [2] | | | 5 9 | | 1 | 3] --- 2 --- [4]

The solution is  $(S, T) = (\{1, 4\}, \{2, 3\})$ 

S, T = solve\_max\_cut\_sdp([0 1 5 0; 1 0 0 9; 5 0 0 2; 0 9 2 0])

([1, 4], [2, 3])

#### Low-rank matrix completion

The matrix completion problem seeks to find the missing entries of a matrix with a given (possibly random) subset of fixed entries, such that the completed matrix has the lowest attainable rank.

For more details, see (Recht et al., 2010).

```
function example_matrix_completion(; svdtol = 1e-6)
 rng = Random.MersenneTwister(1234)
 n = 20
 mask = rand(rng, 1:25, n, n) .== 1
 B = randn(rng, n, n)
 model = Model(Clarabel.Optimizer)
 @variable(model, X[1:n, 1:n])
 @constraint(model, X[mask] .== B[mask])
 @variable(model, t)
 @constraint(model, [t; vec(X)] in MOI.NormNuclearCone(n, n))
 @objective(model, Min, t)
 optimize!(model)
```

```
assert_is_solved_and_feasible(model)
Return the approximate rank of the completed matrix to a given tolerance:
 return sum(LinearAlgebra.svdvals(value.(X)) .> svdtol)
end
```

```
example_matrix_completion()
```

9

#### K-means clustering via SDP

Given a set of points  $a_1,\ldots,a_m$  in  $\mathbb{R}^n$ , allocate them to k clusters.

```
For more details, see (Peng and Wei, 2007).
```

```
function example k means clustering()
 a = [[2.0, 2.0], [2.5, 2.1], [7.0, 7.0], [2.2, 2.3], [6.8, 7.0], [7.2, 7.5]]
 m = length(a)
 num_clusters = 2
 W = zeros(m, m)
 for i in 1:m, j in i+1:m
 W[i, j] = W[j, i] = exp(-LinearAlgebra.norm(a[i] - a[j]) / 1.0)
 end
 model = Model(Clarabel.Optimizer)
 set_silent(model)
 @variable(model, Z[1:m, 1:m] >= 0, PSD)
 @objective(model, Min, LinearAlgebra.tr(W * (LinearAlgebra.I - Z)))
 @constraint(model, [i = 1:m], sum(Z[i, :]) .== 1)
 @constraint(model, LinearAlgebra.tr(Z) == num_clusters)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 Z_val = value.(Z)
 current_cluster, visited = 0, Set{Int}()
 for i in 1:m
 if !(i in visited)
 current cluster += 1
 println("Cluster $current_cluster")
 for j in i:m
 if isapprox(Z_val[i, i], Z_val[i, j]; atol = 1e-3)
 println(a[j])
 push!(visited, j)
 end
 end
 end
 end
 return
end
```

Cluster 1 [2.0, 2.0] [2.5, 2.1] [2.2, 2.3] Cluster 2 [7.0, 7.0] [6.8, 7.0] [7.2, 7.5]

## The correlation problem

Given three random variables A, B, and C, and given bounds on two of the three correlation coefficients:

$$-0.2 \le {}_{AB} \le -0.1$$
$$0.4 \le {}_{BC} \le 0.5$$

our problem is to determine upper and lower bounds on other correlation coefficient  $_{AC}$ .

We solve an SDP to make use of the following positive semidefinite property of the correlation matrix:

$$\begin{bmatrix} 1 & AB & AC \\ AB & 1 & BC \\ AC & BC & 1 \end{bmatrix} \succeq 0$$

```
function example_correlation_problem()
 model = Model(Clarabel.Optimizer)
 set_silent(model)
 @variable(model, X[1:3, 1:3], PSD)
 S = ["A", "B", "C"]
 \rho = Containers.DenseAxisArray(X, S, S)
 (constraint(model, [i in S], \rho[i, i] == 1))
 @constraint(model, -0.2 <= \rho["A", "B"] <= -0.1)
 @constraint(model, 0.4 <= \rho["B", "C"] <= 0.5)
 @objective(model, Max, p["A", "C"])
 optimize!(model)
 assert_is_solved_and_feasible(model)
 println("An upper bound for p_AC is $(value(p["A", "C"]))")
 <code>@objective(model, Min, \rho["A", "C"])</code></code>
 optimize!(model)
 assert_is_solved_and_feasible(model)
 println("A lower bound for \rho_AC is (value(\rho["A", "C"]))))
 return
end
```

example\_correlation\_problem()
An upper bound for  $\rho\_AC$  is 0.8719210444233569 A lower bound for  $\rho$  AC is -0.9779977554370655

#### The minimum distortion problem

This example arises from computational geometry, in particular the problem of embedding a general finite metric space into a Euclidean space.

It is known that the 4-point metric space defined by the star graph

[1] \ 1 (0] -- 1 -- [2] / 1 / [3]

cannot be exactly embedded into a Euclidean space of any dimension, where distances are computed by length of the shortest path between vertices. A distance-preserving embedding would require the three leaf nodes to form an equilateral triangle of side length 2, with the centre node (0) mapped to an equidistant point at distance 1; this is impossible since the triangle inequality in Euclidean space implies all points would need to be simultaneously collinear.

Here we will formulate and solve an SDP to compute the best possible embedding, that is, the embedding f assigning each vertex v to a vector f(v) that minimizes the distortion c such that

$$D[a,b] \le ||f(a) - f(b)|| \le c D[a,b]$$

for all edges (a, b) in the graph, where D[a, b] is the distance in the graph metric space.

Any embedding f can be characterized by a Gram matrix Q, which is PSD and such that

$$||f(a) - f(b)||^2 = Q[a, a] + Q[b, b] - 2Q[a, b]$$

The matrix entry Q[a, b] represents the inner product of f(a) with f(b).

We therefore impose the constraint

$$D[a,b]^2 \le Q[a,a] + Q[b,b] - 2Q[a,b] \le c^2 D[a,b]^2$$

for all edges (a, b) in the graph and minimize  $c^2$ , which gives us the SDP formulation below. Since we may choose any point to be the origin, we fix the first vertex at 0.

For more details, see (Matoušek, 2013; Linial, 2002).

```
function example_minimum_distortion()
 model = Model(Clarabel.Optimizer)
 set_silent(model)
 D = [
 0.0 1.0 1.0 1.0
 1.0 0.0 2.0 2.0
 1.0 2.0 0.0 2.0
 1.0 2.0 2.0 0.0
]
 @variable(model, c^2 >= 1.0)
 @variable(model, Q[1:4, 1:4], PSD)
 for i in 1:4, j in (i+1):4
 @constraint(model, D[i, j]^2 <= Q[i, i] + Q[j, j] - 2 * Q[i, j])</pre>
 @constraint(model, Q[i, i] + Q[j, j] - 2 * Q[i, j] <= c² * D[i, j]²)
 end
 fix(Q[1, 1], 0)
 @objective(model, Min, c²)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 Test.@test objective_value(model) \approx 4 / 3 atol = 1e-4
 # Recover the minimal distorted embedding:
 X = [zeros(3) sqrt(value.(Q)[2:end, 2:end])]
 return Plots.plot(
 X[1, :],
 X[2, :],
 X[3, :];
 seriestype = :mesh3d,
 connections = ([0, 0, 0, 1], [1, 2, 3, 2], [2, 3, 1, 3]),
 legend = false,
 fillalpha = 0.1,
 lw = 3,
 ratio = :equal,
 xlim = (-1.1, 1.1),
 ylim = (-1.1, 1.1),
 zlim = (-1.5, 1.0),
 zticks = -1:1,
 camera = (60, 30),
)
end
```

```
example_minimum_distortion()
```



#### Lovász numbers

The Lovász number of a graph, also known as Lovász's theta-function, is a number that lies between two important and related numbers that are computationally hard to determine, namely the chromatic and clique numbers of the graph. It is possible however to efficient compute the Lovász number as the optimal value of a semidefinite program.

Consider the pentagon graph:

[5] / \ [1] [4] | | [2] --- [3]

with five vertices and edges. Its Lovász number is known to be precisely  $\sqrt{5} \approx 2.236$ , lying between 2 (the largest clique size) and 3 (the smallest number needed for a vertex coloring).

Let i, j be integers such that  $1 \le i < j \le 5$ . We define  $A^{ij}$  to be the  $5 \times 5$  symmetric matrix with entries (i, j) and (j, i) equal to 1, with all other entries 0. Let E be the graph's edge set; in this example, E contains (1,2), (2,3), (3,4), (4,5), (5,1) and their transposes. The Lovász number can be computed from the program

$$\max J \bullet X \tag{8.13}$$

s.t. 
$$A^{ij} \bullet X = 0$$
 for all  $(i, j) \notin E$  (8.14)

$$I \bullet X = 1 \tag{8.15}$$

$$X \succeq 0 \tag{8.16}$$

where J is the matrix filled with ones, and I is the identity matrix.

For more details, see (Barvinok, 2002; Knuth, 1994).

```
function example_theta_problem()
 model = Model(Clarabel.Optimizer)
 set_silent(model)
 \mathsf{E} = [(1, 2), (2, 3), (3, 4), (4, 5), (5, 1)]
 @variable(model, X[1:5, 1:5], PSD)
 for i in 1:5
 for j in (i+1):5
 if !((i, j) in E || (j, i) in E)
 A = zeros(Int, 5, 5)
 A[i, j] = 1
 A[j, i] = 1
 @constraint(model, LinearAlgebra.dot(A, X) == 0)
 end
 end
 end
 @constraint(model, LinearAlgebra.tr(LinearAlgebra.I * X) == 1)
 J = ones(Int, 5, 5)
 @objective(model, Max, LinearAlgebra.dot(J, X))
 optimize!(model)
 assert_is_solved_and_feasible(model)
 Test.@test objective_value(model) \approx sqrt(5) rtol = 1e-4
 println("The Lovász number is: $(objective_value(model))")
 return
end
```

example\_theta\_problem()

The Lovász number is: 2.2360679790987037

#### **Robust uncertainty sets**

This example computes the Value at Risk for a data-driven uncertainty set. Closed-form expressions for the optimal value are available. For more details, see (Bertsimas *et al.*, 2018).

```
function example_robust_uncertainty_sets()
R, d, [], ε = 1, 3, 0.05, 0.05
N = ceil((2 + 2 * log(2 / []))^2) + 1
c, µhat, M = randn(d), rand(d), rand(d, d)
Ehat = 1 / (d - 1) * (M - ones(d) * µhat')' * (M - ones(d) * µhat')
F1([], N) = R / sqrt(N) * (2 + sqrt(2 * log(1 / [])))
```

```
\Gamma^{2}([, N) = 2 * R^{2} / sqrt(N) * (2 + sqrt(2 * log(2 / [])))
 model = Model(Clarabel.Optimizer)
 set_silent(model)
 @variable(model, Σ[1:d, 1:d], PSD)
 @variable(model, u[1:d])
 @variable(model, µ[1:d])
 @constraint(model, [\Gamma1(] / 2, N); \mu - \muhat] in SecondOrderCone())
 @constraint(model, [Γ2(] / 2, N); vec(Σ - Σhat)] in SecondOrderCone())
 \texttt{@constraint(model, [((1-\epsilon)/\epsilon) (u - \mu)'; (u-\mu) \Sigma] >= 0, PSDCone())}
 @objective(model, Max, c' * u)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 exact =
 \muhat' * c +
 \Gamma1(\Box / 2, N) * LinearAlgebra.norm(c) +
 sqrt((1 - \epsilon) / \epsilon) *
 sqrt(c' * (\Sigmahat + \Gamma2(\Box / 2, N) * LinearAlgebra.I) * c)
 Test.@test objective_value(model) \approx exact atol = 1e-2
 return
end
```

example\_robust\_uncertainty\_sets()

## 8.7 Chordal decomposition

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to show how to use MathOptChordalDecomposition.jl to improve the performance of models with PSD constraints.

### **Required packages**

This tutorial uses the following packages:

```
using JuMP
import MathOptChordalDecomposition
import SCS
import SparseArrays
```

#### Background

Chordal decomposition is a technique for decomposing a large PSD constraint into a set of smaller PSD constraints and some linear equality constraints.

If the original PSD constraint is sparse, the decomposed problem can be faster to solve than the original.

For more information on chordal decomposition, watch Michael Garstka's talk at JuMP-dev 2019.

Some solvers, such as Clarabel.jl and COSMO.jl implement chordal decomposition internally. Others, such as SCS.jl do not implement chordal decomposition.

The Julia package MathOptChordalDecomposition.jl is a MathOptInterface layer that implements chordal decomposition of sparse semidefinite constraints. It can be used to wrap any solver which supports PSD constraints and does not implement chordal decomposition internally.

# JuMP Model

To demonstrate the benefits of chordal decomposition, we use the mcp124-1 model from SDPLIB.

```
model = read_from_file(joinpath(@__DIR__, "mcp124-1.dat-s"))
```

```
A JuMP Model
 solver: none
 objective_sense: MIN_SENSE
 L objective_function_type: AffExpr
 num_variables: 124
 num_constraints: 1
 L Vector{AffExpr} in MOI.PositiveSemidefiniteConeTriangle: 1
 Names registered in the model: none
```

This model has 124 decision variables and one PSD constraint. This PSD constraint is sparse, which means that many elements of the matrix are zero.

To view the matrix, use all\_constraints to get a list of the constraints, then use constraint\_object to get the function and set form of the constraint:

```
S = MOI.PositiveSemidefiniteConeTriangle
constraints = all_constraints(model, Vector{AffExpr}, S)
con = constraint_object(constraints[1]);
con.set
```

MathOptInterface.PositiveSemidefiniteConeTriangle(124)

con.func

0

```
7750-element Vector{AffExpr}:
_[1] - 0.25
0
_[2] - 0.5
0
0
_[3] - 0.25
0
0
0
_[4]
0
0
0
0
```

0 0 \_[124] - 1

The constraint function is given in vectorized form. Use reshape\_vector to convert it into a matrix:

F = reshape\_vector(con.func, SymmetricMatrixShape(con.set.side\_dimension))

124×124 Linea	rAlgebra.Sym	metric{AffExp	r, Matrix	{AffExpr}}:	
_[1] - 0.25	Θ	Θ	0	Θ	0
0	_[2] - 0.5	Θ	0	Θ	0
0	Θ	_[3] - 0.25	0	Θ	0
0	Θ	Θ	_[4]	Θ	0
0	Θ	Θ	0	Θ	0
0	Θ	Θ	0	Θ	0
0	Θ	Θ	0	Θ	0
0	Θ	0.25	0	Θ	0
Θ	Θ	Θ	Θ	Θ	0
0	Θ	Θ	0	Θ	0
Θ	Θ	Θ	0	Θ	Θ
Θ	Θ	Θ	Θ	Θ	Θ
0	Θ	Θ	0	Θ	0
Θ	Θ	Θ	Θ	Θ	0
Θ	Θ	Θ	0	Θ	0
Θ	Θ	Θ	0	Θ	0
Θ	Θ	Θ	0	Θ	0
Θ	Θ	Θ	Θ	_[123] - 1	0
Θ	Θ	Θ	Θ	Θ	_[124] - 1

The F matrix is dense, but many elements are zero. Use SparseArrays.sparse to turn it into a sparse matrix:

A = SparseArrays.sparse(F)

124×124 SparseArrays.SparseMatrixCSC{AffExpr,	Int64}	with	422	stored	entries:
[					

The sparse matrix has 422 non-zeros, which is a density of 2.7%:

SparseArrays.nnz(A) / size(A, 1)^2

0.027445369406867846

#### Solution speed

SCS.jl is a first-order solver that does not exploit the sparsity of PSD constraints. Let's solve it and see how long it took:

set\_optimizer(model, SCS.Optimizer)
@time optimize!(model)

```

^^I
 SCS v3.2.7 - Splitting Conic Solver
^^I(c) Brendan O'Donoghue, Stanford University, 2012

problem: variables n: 124, constraints m: 7750
cones: ^^I s: psd vars: 7750, ssize: 1
settings: eps_abs: 1.0e-04, eps_rel: 1.0e-04, eps_infeas: 1.0e-07
^^I alpha: 1.50, scale: 1.00e-01, adaptive_scale: 1
^^I max_iters: 100000, normalize: 1, rho_x: 1.00e-06
^^I acceleration lookback: 10, acceleration interval: 10
^^I compiled with openmp parallelization enabled
lin-sys: sparse-direct-amd-qdldl
^^I nnz(A): 124, nnz(P): 0

iter | pri res | dua res | gap | obj | scale | time (s)
.....
 0| 9.71e-01 3.87e+00 3.12e+02 2.65e+02 1.00e-01 4.27e-03
 250| 1.20e-03 3.27e-04 2.64e-01 1.42e+02 2.92e-02 8.11e-01
 500| 2.97e-04 7.52e-05 4.58e-02 1.42e+02 2.92e-02 1.62e+00
 750 2.40e-04 6.14e-05 3.89e-02 1.42e+02 2.92e-02 2.43e+00
 1000| 1.91e-04 5.10e-05 3.30e-02 1.42e+02 2.92e-02 3.26e+00
 1250 | 1.55e-04 4.34e-05 2.78e-02 1.42e+02 2.92e-02 4.06e+00
 1500| 1.34e-04 3.65e-05 2.34e-02 1.42e+02 2.92e-02 4.86e+00
 1750 2.19e-01 2.84e-01 3.74e+00 1.44e+02 2.92e-02 5.66e+00
 2000 | 1.01e-04 2.57e-05 1.67e-02 1.42e+02 2.92e-02 6.46e+00
```

```
2250| 9.41e-05 2.15e-05 1.42e-02 1.42e+02 2.92e-02 7.26e+00

status: solved

timings: total: 7.26e+00s = setup: 1.86e-03s + solve: 7.25e+00s

^I lin-sys: 1.66e-01s, cones: 7.02e+00s, accel: 1.04e-02s

objective = 141.972713

7.262444 seconds (4.24 k allocations: 1.035 MiB)
```

In comparison, if we wrap SCS.Optimizer in MathOptChordalDecomposition.Optimizer, then the problem takes less than 1 second to solve:

set\_optimizer(model, () -> MathOptChordalDecomposition.Optimizer(SCS.Optimizer))
@time optimize!(model)

\_\_\_\_\_ ^^I SCS v3.2.7 - Splitting Conic Solver ^^I(c) Brendan O'Donoghue, Stanford University, 2012 \_\_\_\_\_ problem: variables n: 1155, constraints m: 8781 cones: ^^I z: primal zero / dual free vars: 7750 ^^I s: psd vars: 1031, ssize: 115 settings: eps\_abs: 1.0e-04, eps\_rel: 1.0e-04, eps\_infeas: 1.0e-07 ^^I alpha: 1.50, scale: 1.00e-01, adaptive\_scale: 1 ^^I max\_iters: 100000, normalize: 1, rho\_x: 1.00e-06 ^^I acceleration\_lookback: 10, acceleration\_interval: 10  $^{I}$  compiled with openmp parallelization enabled lin-sys: sparse-direct-amd-qdldl ^^I nnz(A): 2186, nnz(P): 0 \_\_\_\_\_ iter | pri res | dua res | gap | obj | scale | time (s) \_\_\_\_\_ 0| 2.75e+01 1.00e+00 9.93e+03 -4.81e+03 1.00e-01 2.77e-03 250| 3.65e-03 9.48e-04 1.32e-03 1.42e+02 8.07e-01 1.06e-01 500| 2.63e-04 7.03e-05 1.77e-04 1.42e+02 2.57e+00 2.10e-01 \_\_\_\_\_ status: solved timings: total: 2.10e-01s = setup: 2.38e-03s + solve: 2.08e-01s ^^I lin-sys: 5.95e-02s, cones: 1.29e-01s, accel: 2.43e-03s ..... objective = 141.988826 -----0.215524 seconds (15.72 k allocations: 3.361 MiB)

The difference in performance is because of the chordal decomposition. The decomposed problem introduced new variables (there are now 1,155 variables instead of 124) and constraints (there are now 115 PSD constraints instead of one), but each PSD constraint is smaller than the original.

decom = unsafe\_backend(model)

With a bit of effort, we can compute the number of PSD constraints of each size:

```
count_by_size = Dict{Int,Int}()
for ci in MOI.get(decom, MOI.ListOfConstraintIndices{MOI.VectorOfVariables,S}())
 set = MOI.get(decom, MOI.ConstraintSet(), ci)
 n = set.side_dimension
 count_by_size[n] = get(count_by_size, n, 0) + 1
end
count_by_size
```

```
Dict{Int64, Int64} with 10 entries:

5 => 7

4 => 15

6 => 3

7 => 3

2 => 33

10 => 2

9 => 2

8 => 3

3 => 35

1 => 12
```

The largest PSD constraint is now of size 10, which is much smaller than the original 124-by-124 matrix.

### 8.8 Example: logistic regression

This tutorial was generated using Literate.jl. Download the source as a .jl file.

### This tutorial was originally contributed by François Pacaud.

This tutorial shows how to solve a logistic regression problem with JuMP. Logistic regression is a well known method in machine learning, useful when we want to classify binary variables with the help of a given set of features. To this goal, we find the optimal combination of features maximizing the (log)-likelihood onto a training set.

## **Required packages**

This tutorial uses the following packages:

using JuMP
import Clarabel
import MathOptInterface as MOI
import Random

Random.seed!(2713);

## Formulating the logistic regression problem

Suppose we have a set of training data-point  $i = 1, \dots, n$ , where for each i we have a vector of features  $x_i \in \mathbb{R}^p$  and a categorical observation  $y_i \in \{-1, 1\}$ .

The log-likelihood is given by

$$l(\theta) = \sum_{i=1}^{n} \log(\frac{1}{1 + \exp(-y_i \theta^\top x_i)})$$

and the optimal  $\boldsymbol{\theta}$  minimizes the logistic loss function:

$$\min_{\theta} \sum_{i=1}^{n} \log(1 + \exp(-y_i \theta^\top x_i)).$$

Most of the time, instead of solving directly the previous optimization problem, we prefer to add a regularization term:

$$\min_{\theta} \sum_{i=1}^{n} \log(1 + \exp(-y_i \theta^\top x_i)) + \lambda \|\theta\|$$

with  $\lambda \in \mathbb{R}_+$  a penalty and  $\|.\|$  a norm function. By adding such a regularization term, we avoid overfitting on the training set and usually achieve a greater score in cross-validation.

#### Reformulation as a conic optimization problem

By introducing auxiliary variables  $t_1,\cdots,t_n$  and r, the optimization problem is equivalent to

$$\begin{split} \min_{t,r,\theta} \; \sum_{i=1}^n t_i + \lambda r \\ \text{subject to} \quad t_i \geq \log(1 + \exp(-y_i \theta^\top x_i)) \\ \quad r \geq \|\theta\| \end{split}$$

Now, the trick is to reformulate the constraints  $t_i \ge \log(1 + \exp(-y_i \theta^\top x_i))$  with the help of the *exponential* cone

$$K_{exp} = \{(x, y, z) \in \mathbb{R}^3 : y \exp(x/y) \le z\}.$$

Indeed, by passing to the exponential, we see that for all  $i = 1, \dots, n$ , the constraint  $t_i \ge \log(1 + \exp(-y_i \theta^\top x_i))$  is equivalent to

$$\exp(-t_i) + \exp(u_i - t_i) \le 1$$

with  $u_i = -y_i \theta^\top x_i$ . Then, by adding two auxiliary variables  $z_{i1}$  and  $z_{i2}$  such that  $z_{i1} \ge \exp(u_i - t_i)$  and  $z_{i2} \ge \exp(-t_i)$ , we get the equivalent formulation

$$\begin{cases} (u_i - t_i, 1, z_{i1}) \in K_{exp} \\ (-t_i, 1, z_{i2}) \in K_{exp} \\ z_{i1} + z_{i2} \le 1 \end{cases}$$

In this setting, the conic version of the logistic regression problems writes out

$$\begin{split} \min_{t,z,r,\theta} \; \sum_{i=1}^n t_i + \lambda r \\ \text{subject to} \; & (u_i - t_i, 1, z_{i1}) \in K_{exp} \\ & (-t_i, 1, z_{i2}) \in K_{exp} \\ & z_{i1} + z_{i2} \leq 1 \\ & u_i = -y_i x_i^\top \theta \\ & r \geq \|\theta\| \end{split}$$

and thus encompasses 3n + p + 1 variables and 3n + 1 constraints ( $u_i = -y_i \theta^\top x_i$  is only a virtual constraint used to clarify the notation). Thus, if  $n \gg 1$ , we get a large number of variables and constraints.

#### Fitting logistic regression with a conic solver

We start by implementing a function to generate a fake dataset, and where we could tune the correlation between the feature variables. The function is a direct transcription of the one used in this blog post.

```
function generate_dataset(n_samples = 100, n_features = 10; shift = 0.0)
X = randn(n_samples, n_features)
w = randn(n_features)
y = sign.(X * w)
X .+= 0.8 * randn(n_samples, n_features) # add noise
X .+= shift # shift the points in the feature space
X = hcat(X, ones(n_samples, 1))
return X, y
end
```

generate\_dataset (generic function with 3 methods)

We write a softplus function to formulate each constraint  $t \ge \log(1 + \exp(u))$  with two exponential cones.

```
function softplus(model, t, u)
z = @variable(model, [1:2], lower_bound = 0.0)
@constraint(model, sum(z) <= 1.0)
@constraint(model, [u - t, 1, z[1]] in MOI.ExponentialCone())
@constraint(model, [-t, 1, z[2]] in MOI.ExponentialCone())
end</pre>
```

softplus (generic function with 1 method)

#### $\ell_2$ regularized logistic regression

Then, with the help of the softplus function, we could write our optimization model. In the  $\ell_2$  regularization case, the constraint  $r \ge \|\theta\|_2$  rewrites as a second order cone constraint.

```
function build_logit_model(X, y, \lambda)
 n, p = size(X)
 model = Model()
 @variable(model, 0[1:p])
 @variable(model, t[1:n])
 for i in 1:n
 u = -(X[i, :]' * θ) * y[i]
 softplus(model, t[i], u)
 end
 # Add []2 regularization
 @variable(model, 0.0 <= reg)</pre>
 @constraint(model, [reg; \theta] in SecondOrderCone())
 # Define objective
 @objective(model, Min, sum(t) + \lambda * reg)
 return model
end
```

build\_logit\_model (generic function with 1 method)

We generate the dataset.

Warning

Be careful here, for large n and p Clarabel could fail to converge.

n, p = 200, 10
X, y = generate\_dataset(n, p; shift = 10.0);

```
We could now solve the logistic regression problem

λ = 10.0

model = build_logit_model(X, y, λ)

set_optimizer(model, Clarabel.Optimizer)

set_silent(model)

optimize!(model)

assert_is_solved_and_feasible(model)
```

 $\theta \ddagger$  = value.(model[: $\theta$ ])

```
11-element Vector{Float64}:

0.020412982291617057

0.16139910603563973

0.3570079277158122

-0.3078808169026966

-0.3939185273466955

-0.0591404576296592

0.34717361925866896

-0.8812125871219857

0.20125673132663888

0.5409401386231345

0.0809042115958132
```

It appears that the speed of convergence is not that impacted by the correlation of the dataset, nor by the penalty  $\lambda$ .

## $\ell_1$ regularized logistic regression

We now formulate the logistic problem with a  $\ell_1$  regularization term. The  $\ell_1$  regularization ensures sparsity in the optimal solution of the resulting optimization problem. Luckily, the  $\ell_1$  norm is implemented as a set in MathOptInterface. Thus, we could formulate the sparse logistic regression problem with the help of a MOI.NormOneCone set.

```
function build_sparse_logit_model(X, y, λ)
n, p = size(X)
model = Model()
@variable(model, θ[1:p])
@variable(model, t[1:n])
for i in 1:n
 u = -(X[i, :]' * θ) * y[i]
 softplus(model, t[i], u)
end
Add [] regularization
@variable(model, 0.0 <= reg)
@constraint(model, [reg; θ] in MOI.NormOneCone(p + 1))
Define objective
@objective(model, Min, sum(t) + λ * reg)</pre>
```

return model

end

```
Auxiliary function to count non-null components:
count_nonzero(v::Vector; tol = 1e-6) = sum(abs.(v) .>= tol)
We solve the sparse logistic regression problem on the same dataset as before.
\lambda = 10.0
sparse_model = build_sparse_logit_model(X, y, \lambda)
set_optimizer(sparse_model, Clarabel.Optimizer)
set_silent(sparse_model)
optimize!(sparse_model)
assert_is_solved_and_feasible(sparse_model)
```

```
θ# = value.(sparse_model[:0])
println(
 "Number of non-zero components: ",
 count_nonzero(0#),
 " (out of ",
 p,
 " features)",
)
```

Number of non-zero components: 8 (out of 10 features)

### Extensions

A direct extension would be to consider the sparse logistic regression with *hard* thresholding, which, on contrary to the *soft* version using a  $\ell_1$  regularization, adds an explicit cardinality constraint in its formulation:

$$\begin{split} \min_{\boldsymbol{\theta}} \; \sum_{i=1}^n \log(1 + \exp(-y_i \boldsymbol{\theta}^\top x_i)) + \lambda \|\boldsymbol{\theta}\|_2^2 \\ \text{subject to} \; \; \|\boldsymbol{\theta}\|_0 <= k \end{split}$$

where k is the maximum number of non-zero components in the vector  $\theta$ , and  $\|.\|_0$  is the  $\ell_0$  pseudo-norm:

$$||x||_0 = \#\{i: x_i \neq 0\}$$

The cardinality constraint  $\|\theta\|_0 \le k$  could be reformulated with binary variables. Thus the hard sparse regression problem could be solved by any solver supporting mixed integer conic problems.

# 8.9 Example: experiment design

This tutorial was generated using Literate.jl. Download the source as a .jl file.

#### This tutorial was originally contributed by Arpit Bhatia and Chris Coey.

This tutorial covers experiment design examples (D-optimal, A-optimal, and E-optimal) from section 7.5 of (Boyd and Vandenberghe, 2004).

## **Required packages**

This tutorial uses the following packages:

```
using JuMP
import Clarabel
import LinearAlgebra
import MathOptInterface as MOI
import Random
```

We set a seed so the random numbers are repeatable:

```
Random.seed!(1234)
```

Random.TaskLocalRNG()

#### The relaxed experiment design problem

The basic experiment design problem is as follows.

Given the menu of possible choices for experiments,  $v_1, \ldots, v_p$ , and the total number m of experiments to be carried out, choose the numbers of each type of experiment, that is,  $m_1, \ldots, m_p$  to make the error covariance E small (in some sense).

The variables  $m_1, \ldots, m_p$  must, of course, be integers and sum to m the given total number of experiments. This leads to the optimization problem:

$$\min\left(\text{w.r.t.}\mathbf{S}^{n}_{+}\right)E = \left(\sum_{j=1}^{p} m_{j}v_{j}v_{j}^{T}\right)^{-1}$$
  
subject to $m_{i} \ge 0$   
$$\sum_{i=1}^{p} m_{i} = m$$
$$m_{i} \in \mathbb{Z}, \quad i = 1, \dots, p$$

The basic experiment design problem can be a hard combinatorial problem when m, the total number of experiments, is comparable to n, since in this case the  $m_i$  are all small integers.

In the case when m is large compared to n, however, a good approximate solution can be found by ignoring, or relaxing, the constraint that the  $m_i$  are integers.

Let  $\lambda_i = m_i/m$ , which is the fraction of the total number of experiments for which  $a_j = v_i$ , or the relative frequency of experiment *i*. We can express the error covariance in terms of  $\lambda_i$  as:

$$E = \frac{1}{m} \left( \sum_{i=1}^{p} \lambda_i v_i v_i^T \right)^{-1}$$

The vector  $\lambda \in \mathbf{R}^p$  satisfies  $\lambda \succeq 0, \mathbf{1}^T \lambda = 1$ , and also, each  $\lambda_i$  is an integer multiple of 1/m. By ignoring this last constraint, we arrive at the problem:

$$\min\left(\text{w.r.t.}\mathbf{S}^{n}_{+}\right)E = (1/m)\left(\sum_{i=1}^{p}\lambda_{i}v_{i}v_{i}^{T}\right)^{-1}$$
  
subject to:  $\lambda \succeq 0$   
 $\mathbf{1}^{T}\lambda = 1$ 

Several scalarizations have been proposed for the experiment design problem, which is a vector optimization problem over the positive semidefinite cone.

```
q = 4 # dimension of estimate space
p = 8 # number of experimental vectors
n_max = 3 # upper bound on lambda
n = 12
V = randn(q, p)
eye = Matrix{Float64}(LinearAlgebra.I, q, q);
```

## A-optimal design

In A-optimal experiment design, we minimize tr E, the trace of the covariance matrix. This objective is simply the mean of the norm of the error squared:

$$\mathbf{E} \|e\|_2^2 = \mathbf{E} \operatorname{tr} \left( e e^T \right) = \operatorname{tr} E$$

The A-optimal experiment design problem in SDP form is

$$\begin{split} \min \mathbf{1}^T u \\ \text{subject to} \left[ \sum_{i=1}^p \lambda_i v_i v_i^T e_k \\ e_k^T u_k \right] \succeq 0, \quad k = 1, \dots, n \\ \lambda \succeq 0 \\ \mathbf{1}^T \lambda = 1 \end{split}$$

```
aOpt = Model(Clarabel.Optimizer)
set_silent(a0pt)
@variable(a0pt, np[1:p], lower_bound = 0, upper_bound = n_max)
@variable(a0pt, u[1:q], lower_bound = 0)
@constraint(aOpt, sum(np) <= n)</pre>
for i in 1:q
 matrix = [
 V*LinearAlgebra.Diagonal(np ./ n)*V' eye[:, i]
 eye[i, :]' u[i]
]
 @constraint(aOpt, matrix >= 0, PSDCone())
end
@objective(a0pt, Min, sum(u))
optimize!(a0pt)
assert_is_solved_and_feasible(a0pt)
objective_value(a0pt)
```

5.103199058617686

value.(np)

```
8-element Vector{Float64}:
2.949598980152687
1.7791514505035955
7.276305887925417e-10
1.4509996590179904e-9
2.108204914610362
1.6982760520528668
1.263425849963127
2.201342749918528
```

### **E-optimal design**

In  ${\cal E}$  -optimal design, we minimize the norm of the error covariance matrix, that is, the maximum eigenvalue of  ${\cal E}.$ 

Since the diameter (twice the longest semi-axis) of the confidence ellipsoid  $\mathcal{E}$  is proportional to  $||E||_2^{1/2}$ , minimizing  $||E||_2$  can be interpreted geometrically as minimizing the diameter of the confidence ellipsoid.

E-optimal design can also be interpreted as minimizing the maximum variance of  $q^T e$ , over all q with  $||q||_2 = 1$ . The E-optimal experiment design problem in SDP form is:

$$\begin{aligned} \min t \\ \text{subject to} \sum_{i=1}^{p} \lambda_i v_i v_i^T \succeq t I \\ \lambda \succeq 0 \\ \mathbf{1}^T \lambda = 1 \end{aligned}$$

```
e0pt = Model(Clarabel.Optimizer)
set_silent(e0pt)
@variable(e0pt, 0 <= np[1:p] <= n_max)
@variable(e0pt, t)
@constraint(
 e0pt,
 V * LinearAlgebra.Diagonal(np ./ n) * V' - (t .* eye) >= 0,
 PSDCone(),
)
@constraint(e0pt, sum(np) <= n)
@objective(e0pt, Max, t)
optimize!(e0pt)
assert_is_solved_and_feasible(e0pt)
objective_value(e0pt)</pre>
```

0.4353846161234582

value.(np)

8-element Vector{Float64}: 2.999999998431658 2.752797006101644 4.10051457567522e-9 4.8207598279432515e-9 2.181840806067763 2.3253295524575566 0.21895619171783068 1.5210764321965744

### **D-optimal design**

The most widely used scalarization is called D-optimal design, in which we minimize the determinant of the error covariance matrix E. This corresponds to designing the experiment to minimize the volume of the resulting confidence ellipsoid (for a fixed confidence level). Ignoring the constant factor 1/m in E, and taking the logarithm of the objective, we can pose this problem as convex optimization problem:

$$\begin{split} \min\log\det\left(\sum_{i=1}^p\lambda_iv_iv_i^T\right)^{-1}\\ \text{subject to}\lambda\succeq 0\\ \mathbf{1}^T\lambda=1 \end{split}$$

d0pt = Model(Clarabel.Optimizer)
set\_silent(d0pt)

```
@variable(d0pt, np[1:p], lower_bound = 0, upper_bound = n_max)
@variable(d0pt, t)
@objective(d0pt, Max, t)
@constraint(d0pt, sum(np) <= n)
E = V * LinearAlgebra.Diagonal(np ./ n) * V'
@constraint(d0pt, [t; 1; triangle_vec(E)] in MOI.LogDetConeTriangle(q))
optimize!(d0pt)
assert_is_solved_and_feasible(d0pt)
objective_value(d0pt)</pre>
```

0.3084755071033654

value.(np)

```
8-element Vector{Float64}:
0.42764306607681873
2.9100111142529657
6.411004669047624e-10
8.680825215182773e-10
2.9158597562579103
2.6732765281643593
2.7353858758724106
0.33782365595975183
```

# 8.10 Example: minimal ellipses

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This example comes from section 8.4.1 of the book Convex Optimization by Boyd and Vandenberghe (2004).

### Formulation

Given a set of m ellipses of the form:

$$E(A, b, c) = \{x : x^{\top}Ax + 2b^{\top}x + c \le 0\},\$$

the minimal ellipse problem finds an ellipse with the minimum area that encloses the given ellipses.

It is convenient to parameterize the minimal enclosing ellipse as

$$\{x: ||Px+q|| \le 1\}.$$

Then the optimal P and q are given by the convex semidefinite program;

$$\begin{array}{ll} \mbox{maximize} & \log(\det(P)) \\ \mbox{subject to} & \tau_i \geq 0, & i = 1, \dots, m \\ & \begin{bmatrix} P^2 - \tau_i A_i & Pq - \tau_i b_i & 0 \\ (Pq - \tau_i b_i)^\top & -1 - \tau_i c_i & (Pq)^\top \\ 0 & (Pq) & -P^2 \end{bmatrix} \preceq 0 \ \mbox{(PSD)} \quad i = 1, \dots, m \end{array}$$

with helper variables au.

### **Required packages**

This tutorial uses the following packages:

```
using JuMP
import Clarabel
import LinearAlgebra
import Plots
import Test
```

# Data

First, define the m input ellipses (here m = 6), parameterized as  $x^T A_i x + 2b_i^T x + c \le 0$ :

```
struct Ellipse
 A::Matrix{Float64}
 b::Vector{Float64}
 c::Float64
 function Ellipse(A::Matrix{Float64}, b::Vector{Float64}, c::Float64)
 @assert isreal(A) && LinearAlgebra.issymmetric(A)
 return new(A, b, c)
 end
end
ellipses = [
 Ellipse([1.2576 -0.3873; -0.3873 0.3467], [0.2722, 0.1969], 0.1831),
 Ellipse([1.4125 -2.1777; -2.1777 6.7775], [-1.228, -0.0521], 0.3295),
 Ellipse([1.7018 0.8141; 0.8141 1.7538], [-0.4049, 1.5713], 0.2077),
 Ellipse([0.9742 -0.7202; -0.7202 1.5444], [0.0265, 0.5623], 0.2362),
 Ellipse([0.6798 -0.1424; -0.1424 0.6871], [-0.4301, -1.0157], 0.3284),
 Ellipse([0.1796 -0.1423; -0.1423 2.6181], [-0.3286, 0.557], 0.4931),
];
```

We visualise the ellipses using the Plots package:

```
function plot_ellipse(plot, ellipse::Ellipse)
A, b, c = ellipse.A, ellipse.b, ellipse.c
\theta = range(0, 2pi + 0.05; step = 0.05)
Some linear algebra to convert \theta into (x,y) coordinates.
x_y = \sqrt{A} \setminus (\sqrt{b' * (A \setminus b)} - c) .* hcat(cos.(\theta), sin.(\theta)) .- (\sqrt{A \setminus b})')'
Plots.plot!(plot, x_y[1, :], x_y[2, :]; label = nothing, c = :navy)
return
```

### end

```
plot = Plots.plot(; size = (600, 600))
for ellipse in ellipses
 plot_ellipse(plot, ellipse)
end
plot
```



## **Build the model**

Now let's build the model, using the change-of-variables  $P^2 = P^2$  and  $P_q = Pq$ . We'll recover the true value of P and q after the solve.

```
model = Model(Clarabel.Optimizer)
set_silent(model)
m, n = length(ellipses), size(first(ellipses).A, 1)
@variable(model, \tau[1:m] >= 0)
@variable(model, P²[1:n, 1:n], PSD)
@variable(model, P_q[1:n])
for (i, ellipse) in enumerate(ellipses)
 A, b, c = ellipse.A, ellipse.b, ellipse.c
 X = [
 #! format: off
 (P^2 - \tau[i] * A) (P_q - \tau[i] * b) zeros(n, n)
 (P_q - \tau[i] * b)' (-1 - \tau[i] * c) P_q'
 zeros(n, n) P_q
 - P²
 #! format: on
]
 @constraint(model, LinearAlgebra.Symmetric(X) <= 0, PSDCone())</pre>
end
```

We cannot directly represent the objective log(det(P)), so we introduce the conic reformulation:

```
@variable(model, log_det_P)
@constraint(model, [log_det_P; 1; vec(P²)] in MOI.LogDetConeSquare(n))
@objective(model, Max, log_det_P)
```

 $log\_det\_P$ 

Now, solve the program:

optimize!(model)
assert\_is\_solved\_and\_feasible(model)
solution summary(model)

```
solution_summary(; result = 1, verbose = false)
| solver_name : Clarabel
| Termination
| | termination_status : OPTIMAL
| | result_count : 1
| L raw_status : SOLVED
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | dual_objective_value : -4.04369e+00
| L dual_objective_value : -4.04369e+00
L Work counters
| solve_time (sec) : 5.11439e-03
L barrier_iterations : 15
```

# Results

After solving the model to optimality we can recover the solution in terms of  ${\cal P}$  and  $q{:}$ 

 $P = sqrt(value.(P^2))$  $q = P \setminus value.(P_q)$ 

```
2-element Vector{Float64}:
-0.3964824351357496
-0.02122031038251278
```

Finally, overlaying the solution in the plot we see the minimal area enclosing ellipsoid:

```
Plots.plot!(
 plot,
 [tuple(P \ [cos(θ) - q[1], sin(θ) - q[2]]...) for θ in 0:0.05:(2pi+0.05)];
 c = :crimson,
 label = nothing,
)
```



# 8.11 Example: ellipsoid approximation

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial considers the problem of computing *extremal ellipsoids*: finding ellipsoids that best approximate a given set. As an extension, we show how to use JuMP to inspect the bridges that were used, and how to explore alternative formulations.

The model comes from Section 4.9 of (Ben-Tal and Nemirovski, 2001).

For a related example, see also the Example: minimal ellipses tutorial.

## **Required packages**

This tutorial uses the following packages:

using JuMP import Clarabel import LinearAlgebra import Plots import Random import Test

## **Problem formulation**

Suppose that we are given a set  ${\mathcal S}$  consisting of m points in n-dimensional space:

$$\mathcal{S} = \{x_1, \dots, x_m\} \subset \mathbb{R}^n$$

Our goal is to determine an optimal vector  $c \in \mathbb{R}^n$  and an optimal  $n \times n$  real symmetric matrix D such that the ellipse:

$$E(D,c) = \{x : (x-c)^{\top} D(x-c) \le 1\},\$$

contains  ${\mathcal S}$  and has the smallest possible volume.

The optimal D and c are given by the optimization problem:

$$\begin{array}{ll} \max & t \\ \text{s.t.} & Z \succeq 0 \\ & \begin{bmatrix} s & z^\top \\ z & Z \end{bmatrix} \succeq 0 \\ & x_i^\top Z x_i - 2 x_i^\top z + s \leq 1 \qquad i = 1, \dots, m \\ & t \leq \sqrt[n]{\det(Z)}, \end{array}$$

where  $D = Z_*$  and  $c = Z_*^{-1} z_*$ .

## Data

We first need to generate some points to work with.

generate\_point\_cloud (generic function with 1 method)

For the sake of this example, let's take m=100:

```
S = generate_point_cloud(100);
```

We will visualise the points (and ellipse) using the Plots package:

```
r = 1.1 * maximum(abs.(S))
plot = Plots.scatter(
 S[:, 1],
 S[:, 2];
 xlim = (-r, r),
 ylim = (-r, r),
 label = nothing,
 c = :green,
 shape = :x,
 size = (600, 600),
)
```



### **JuMP** formulation

Now let's build and the JuMP model. We'll compute D and c after the solve.

```
model = Model(Clarabel.Optimizer)
set_silent(model)
m, n = size(S)
@variable(model, z[1:n])
@variable(model, Z[1:n, 1:n], PSD)
@variable(model, s)
@variable(model, t)
@constraint(model, [s z'; z Z] >= 0, PSDCone())
@constraint(
 model,
 [i in 1:m],
```

```
S[i, :]' * Z * S[i, :] - 2 * S[i, :]' * z + s <= 1,
)
@constraint(model, [t; vec(Z)] in MOI.RootDetConeSquare(n))
@objective(model, Max, t)
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)</pre>
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : Clarabel
| Termination
| | termination_status : OPTIMAL
| | result_count : 1
| L raw_status : SOLVED
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | objective_value : 7.92350e-03
| L dual_objective_value : 7.92350e-03
| Work counters
| solve_time (sec) : 2.45823e-03
| barrier_iterations : 16
```

## Results

After solving the model to optimality we can recover the solution in terms of D and c:

D = value.(Z)

```
2×2 Matrix{Float64}:
0.012616 -0.02132
-0.02132 0.0410053
```

 $c = D \setminus value.(z)$ 

```
2-element Vector{Float64}:
-1.6318778797029514
-0.670458237995941
```

We can check that each point lies inside the ellipsoid, by checking if the largest normalized radius is less than 1:

largest\_radius = maximum(map(x -> (x - c) \* D \* (x - c), eachrow(S)))

#### 0.9999999991403553

Finally, overlaying the solution in the plot we see the minimal volume approximating ellipsoid:

```
P = sqrt(D)
q = -P * c
data = [tuple(P \ [cos(θ) - q[1], sin(θ) - q[2]]...) for θ in 0:0.05:(2pi+0.05)]
Plots.plot!(plot, data; c = :crimson, label = nothing)
```



## Alternative formulations

The formulation of model uses MOI.RootDetConeSquare. However, because Clarabel does not natively support this cone, JuMP automatically reformulates the problem into an equivalent problem that Clarabel *does* support. You can see the reformulation that JuMP chose using print\_active\_bridges:

print\_active\_bridges(model)

* Unsupported objective: MOI.VariableIndex
Dridged by: MOTE_Objective_EurotionConversionBridge/Eleat64MOT_ScalarAffineEurotion/Eleat64}
$ MOI.VariableIndex} $
may introduce:
<pre>* Supported objective: MOI.ScalarAffineFunction{Float64}</pre>
* Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.LessThan{Float64}
bridged by:
<pre>MOIB.Constraint.LessToGreaterBridge{Float64, MOI.ScalarAffineFunction{Float64},</pre>
<pre>→ MOI.ScalarAffineFunction{Float64}}</pre>
may introduce:
* Unsupported constraint: MULSCalarATTINEFUNCTIOn{Float64}-in-MULGreaterInan{Float64}
<pre>MOIB Constraint VectorizeBridge{Eloat64 MOI VectorAffineEunction{Eloat64}</pre>
$\rightarrow$ MOI.Nonnegatives. MOI.ScalarAffineFunction{Float64}}
may introduce:
<pre>* Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives</pre>
* Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeSquare
bridged by:
<pre>MOIB.Constraint.SquareBridge{Float64, MOI.VectorAffineFunction{Float64},</pre>
→ MOI.ScalarAffineFunction{Float64}, MOI.PositiveSemidefiniteConeTriangle,
→ MUL.POSITIVeSemideTiniteLoneSquare}
may introduce:
<pre>→ MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle</pre>
bridged by:
MOIB.Constraint.SetDotScalingBridge{Float64, MOI.PositiveSemidefiniteConeTriangle,
→ MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
may introduce:
* Supported constraint:
→ MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle}
<pre>* Unsupported constraint: MUI.ScalarAffineFunction{Float64}-in-MUI.Equallo{Float64}</pre>
Driaged by:   MOIR Constraint VectorizeBridge/Eleat64_MOI VectorAffineEunction/Eleat64}_MOI Zeros
$ = MOI.ScalarAffineEunction{Eloat64}} $
may introduce:
<pre>\$ Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Zeros</pre>
* Unsupported constraint: MOI.VectorOfVariables-in-MOI.PositiveSemidefiniteConeTriangle
bridged by:
<pre>MOIB.Constraint.SetDotScalingBridge{Float64, MOI.PositiveSemidefiniteConeTriangle,</pre>
→ MOI.VectorAffineFunction{Float64}, MOI.VectorOfVariables}
may introduce:
$\sim$ Supported CONSTRAINT: $\sim$ MOT VectorAffineFunction/Float64} in MOT Scaled/MOT PositiveSemidefiniteConsTriangle}
* Unsupported constraint: MOI.VectorOfVariables-in-MOI.RootDetConeSquare
bridged by:

MOIB.Constraint.SquareBridge{Float64, MOI.VectorOfVariables, 1 → MOI.ScalarAffineFunction{Float64}, MOI.RootDetConeTriangle, MOI.RootDetConeSquare} | may introduce: \* Unsupported constraint: MOI.VectorOfVariables-in-MOI.RootDetConeTriangle | bridged by: MOIB.Constraint.RootDetBridge{Float64, MOI.VectorAffineFunction{Float64}, → MOI.VectorOfVariables, MOI.VectorOfVariables} 1 | may introduce: \* Unsupported constraint: → MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle | | bridged by: MOIB.Constraint.SetDotScalingBridge{Float64, MOI.PositiveSemidefiniteConeTriangle, → MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}} | may introduce: \* Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle}  $\rightarrow$ \* Unsupported constraint: MOI.VectorOfVariables-in-MOI.GeometricMeanCone L | bridged by: MOIB.Constraint.GeoMeanToPowerBridge{Float64, MOI.VectorOfVariables} | may introduce: \* Supported constraint: MOI.VectorOfVariables-in-MOI.PowerCone{Float64} 1 \* Supported variable: MOI.Nonnegatives \* Supported variable: MOI.Reals \* Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.EqualTo{Float64} | bridged by: MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64}, MOI.Zeros, → MOI.ScalarAffineFunction{Float64}} | may introduce: \* Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Zeros 

There's a lot going on here, but the first bullet is:

\* Unsupported objective: MOI.VariableIndex

```
| bridged by:
```

MOIB.Objective.FunctionConversionBridge{Float64}

```
| may introduce:
```

\* Supported objective: MOI.ScalarAffineFunction{Float64}

This says that Clarabel does not support a MOI.VariableIndex objective function, and that JuMP used a MOI.Bridges.Objective.FunctionConversionBridge to convert it into a MOI.ScalarAffineFunction{Float64} objective function.

We can leave JuMP to do the reformulation, or we can rewrite our model to have an objective function that Clarabel natively supports:

@objective(model, Max, 1.0 \* t + 0.0);

Re-printing the active bridges:

print\_active\_bridges(model)

\* Supported objective: MOI.ScalarAffineFunction{Float64} \* Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.LessThan{Float64} | bridged by: MOIB.Constraint.LessToGreaterBridge{Float64, MOI.ScalarAffineFunction{Float64}, → MOI.ScalarAffineFunction{Float64}} | may introduce: \* Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.GreaterThan{Float64} | bridged by: MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64}, → MOI.Nonnegatives, MOI.ScalarAffineFunction{Float64}} | | may introduce: \* Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives \* Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeSquare | bridged by: MOIB.Constraint.SquareBridge{Float64, MOI.VectorAffineFunction{Float64}, → MOI.ScalarAffineFunction{Float64}, MOI.PositiveSemidefiniteConeTriangle, → MOI.PositiveSemidefiniteConeSquare} | may introduce: \* Unsupported constraint: → MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle | bridged by: MOIB.Constraint.SetDotScalingBridge{Float64, MOI.PositiveSemidefiniteConeTriangle, → MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}} | may introduce: \* Supported constraint: → MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle} \* Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.EqualTo{Float64} | | bridged by: | MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64}, MOI.Zeros, → MOI.ScalarAffineFunction{Float64}} may introduce: \* Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Zeros \* Unsupported constraint: MOI.VectorOfVariables-in-MOI.PositiveSemidefiniteConeTriangle | bridged by: MOIB.Constraint.SetDotScalingBridge{Float64, MOI.PositiveSemidefiniteConeTriangle, → MOI.VectorAffineFunction{Float64}, MOI.VectorOfVariables} | may introduce: \* Supported constraint:  $\rightarrow \quad \texttt{MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle}}$ \* Unsupported constraint: MOI.VectorOfVariables-in-MOI.RootDetConeSquare | bridged by: MOIB.Constraint.SquareBridge{Float64, MOI.VectorOfVariables, → MOI.ScalarAffineFunction{Float64}, MOI.RootDetConeTriangle, MOI.RootDetConeSquare} | may introduce: \* Unsupported constraint: MOI.VectorOfVariables-in-MOI.RootDetConeTriangle | bridged by: MOIB.Constraint.RootDetBridge{Float64, MOI.VectorAffineFunction{Float64}, → MOI.VectorOfVariables, MOI.VectorOfVariables} | may introduce: \* Unsupported constraint:  $\hookrightarrow \quad \texttt{MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle}$ | | | bridged by: MOIB.Constraint.SetDotScalingBridge{Float64, MOI.PositiveSemidefiniteConeTriangle, → MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}

| | | may introduce:

	* Supported constraint:
$\hookrightarrow$	MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle}
	* Unsupported constraint: MOI.VectorOfVariables-in-MOI.GeometricMeanCone
	bridged by:
	<pre>  MOIB.Constraint.GeoMeanToPowerBridge{Float64, MOI.VectorOfVariables}</pre>
	may introduce:
	<pre>  * Supported constraint: MOI.VectorOfVariables-in-MOI.PowerCone{Float64}</pre>
I	* Supported variable: MOI.Nonnegatives
I	* Supported variable: MOI.Reals
	* Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.EqualTo{Float64}
	bridged by:
	<pre>MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64}, MOI.Zeros,</pre>
$\hookrightarrow$	MOI.ScalarAffineFunction{Float64}}
	may introduce:
I	* Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Zeros

we get \* Supported objective: MOI.ScalarAffineFunction{Float64}.

We can manually implement some other reformulations to change our model to something that Clarabel more closely supports by:

- Replacing the MOI.VectorOfVariables in MOI.PositiveSemidefiniteConeTriangle constraint@variable(model, Z[1:n, 1:n], PSD) with the MOI.VectorAffineFunction in MOI.PositiveSemidefiniteConeTriangle @constraint(model, Z >= 0, PSDCone()).
- Replacing the MOI.VectorOfVariables in MOI.PositiveSemidefiniteConeSquare constraint [s z'; z Z] >= 0, PSDCone() with the MOI.VectorAffineFunction in MOI.PositiveSemidefiniteConeTriangle @constraint(model, LinearAlgebra.Symmetric([s z'; z Z]) >= 0, PSDCone()).
- Replacing the MOI.ScalarAffineFunction in MOI.GreaterThan constraints with the vectorized equivalent of MOI.VectorAffineFunction in MOI.Nonnegatives
- Replacing the MOI.VectorOfVariables in MOI.RootDetConeSquare constraint with MOI.VectorAffineFunction in MOI.RootDetConeTriangle.

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, z[1:n])
@variable(model, s)
@variable(model, t)
The former @variable(model, Z[1:n, 1:n], PSD)
@variable(model, Z[1:n, 1:n], Symmetric)
@constraint(model, Z >= 0, PSDCone())
The former [s z'; z Z] >= 0, PSDCone()
@constraint(model, LinearAlgebra.Symmetric([s z'; z Z]) >= 0, PSDCone())
The former constraint S[i, :]' * Z * S[i, :] - 2 * S[i, :]' * z + s <= 1</pre>
f = [1 - S[i, :]' * Z * S[i, :] + 2 * S[i, :]' * z - s for i in 1:m]
@constraint(model, f in MOI.Nonnegatives(m))
The former constraint [t; vec(Z)] in MOI.RootDetConeSquare(n)
@constraint(model, 1 * [t; triangle_vec(Z)] .+ 0 in MOI.RootDetConeTriangle(n))
The former @objective(model, Max, t)
@objective(model, Max, 1 * t + 0)
optimize!(model)
```

assert\_is\_solved\_and\_feasible(model)
solve\_time\_1 = solve\_time(model)

0.002134378

This formulation gives the much smaller graph:

print\_active\_bridges(model)

* Supported objective: MOI.ScalarAffineFunction{Float64}
* Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
* Unsupported constraint:
$\leftrightarrow$ MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
bridged by:
<pre>MOIB.Constraint.SetDotScalingBridge{Float64, MOI.PositiveSemidefiniteConeTriangle,</pre>
→ MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
may introduce:
* Supported constraint:
$\leftrightarrow$ MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle}
* Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.RootDetConeTriangle
bridged by:
<pre>MOIB.Constraint.RootDetBridge{Float64, MOI.VectorAffineFunction{Float64},</pre>
$\hookrightarrow$ MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
may introduce:
* Unsupported constraint:
$\hookrightarrow$ MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
bridged by:
<pre>  MOIB.Constraint.SetDotScalingBridge{Float64, MOI.PositiveSemidefiniteConeTriangle,</pre>
$\rightarrow$ MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
may introduce:
* Supported constraint:
$\rightarrow$ MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle}
* Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.GeometricMeanCone
bridged by:
<pre>  MOIB.Constraint.GeoMeanToPowerBridge{Float64, MOI.VectorAffineFunction{Float64}}</pre>
may introduce:
<pre>  * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.PowerCone{Float64}</pre>
* Supported variable: MOI.Nonnegatives
* Supported variable: MOI.Reals

Note that we still need to bridge MOI.PositiveSemidefiniteConeTriangle constraints because Clarabel uses the MOI.Scaled PSD cone.

```
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, z[1:n])
@variable(model, s)
@variable(model, t)
```

```
@variable(model, Z[1:n, 1:n], Symmetric)
The former @constraint(model, Z in PSDCone())
f = triangle_vec(Z)
scale_f = [1.0, sqrt(2), 1.0]
@constraint(
 model,
 scale_f .* f in MOI.Scaled(MOI.PositiveSemidefiniteConeTriangle(n)),
)
The former LinearAlgebra.Symmetric([s z'; z Z]) >= 0, PSDCone()
g = triangle_vec(LinearAlgebra.Symmetric([s z'; z Z]))
scale_g = [1.0, sqrt(2), 1.0, sqrt(2), sqrt(2), 1.0]
@constraint(
 model.
 scale_g .* g in MOI.Scaled(MOI.PositiveSemidefiniteConeTriangle(1 + n)),
)
f = [1 - S[i, :]' * Z * S[i, :] + 2 * S[i, :]' * z - s for i in 1:m]
@constraint(model, f in MOI.Nonnegatives(m))
@constraint(model, 1 * [t; triangle_vec(Z)] .+ 0 in MOI.RootDetConeTriangle(n))
@objective(model, Max, 1 * t + 0)
optimize!(model)
assert_is_solved_and_feasible(model)
solve_time_2 = solve_time(model)
```

```
0.002105204000000003
```

This formulation gives the much smaller graph:

print\_active\_bridges(model)

```
* Supported objective: MOI.ScalarAffineFunction{Float64}
```

- \* Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
- \* Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.RootDetConeTriangle | bridged by:
- MOIB.Constraint.RootDetBridge{Float64, MOI.VectorAffineFunction{Float64},
- $\hookrightarrow \quad \texttt{MOI.VectorAffineFunction{Float64}, \ \texttt{MOI.VectorAffineFunction{Float64}}}$
- | may introduce:
- \* Unsupported constraint:
- $\hookrightarrow \quad \texttt{MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle}$
- | | bridged by:
- | MOIB.Constraint.SetDotScalingBridge{Float64, MOI.PositiveSemidefiniteConeTriangle,
- $\hookrightarrow$  MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
- | | may introduce:
- | | \* Supported constraint:
- $\rightarrow \quad \texttt{MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle} }$
- | | bridged by:

- | MOIB.Constraint.GeoMeanToPowerBridge{Float64, MOI.VectorAffineFunction{Float64}}
- | | may introduce:
  - \* Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.PowerCone{Float64}
  - \* Supported variable: MOI.Nonnegatives
```
| * Supported variable: MOI.Reals
* Supported constraint:
→ MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle}
```

Now there is only a single Unsupported constraint bullet, showing how JuMP reformulated the MOI.RootDetConeTriangle constraint by adding a mix of MOI.PositiveSemidefiniteConeTriangle and MOI.GeometricMeanCone constraints.

Because Clarabel doesn't natively support the MOI.GeometricMeanCone, these constraints were further bridged using a MOI.Bridges.Constraint.GeoMeanToPowerBridge to a series of MOI.PowerCone constraints.

However, there are many other ways that a MOI. GeometricMeanCone can be reformulated into something that Clarabel supports. Let's see what happens if we use remove\_bridge to remove the MOI.Bridges.Constraint.GeoMeanToPowerBr

```
remove_bridge(model, MOI.Bridges.Constraint.GeoMeanToPowerBridge)
optimize!(model)
assert_is_solved_and_feasible(model)
```

This time, the solve took:

solve\_time\_3 = solve\_time(model)

0.001948692000000001

where previously it took

solve\_time\_2

0.002105204000000003

Why was the solve time different?

print\_active\_bridges(model)

- \* Supported objective: MOI.ScalarAffineFunction{Float64}
- \* Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
- \* Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.RootDetConeTriangle

- MOIB.Constraint.RootDetBridge{Float64, MOI.VectorAffineFunction{Float64},
- → MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}

| may introduce:

- \* Unsupported constraint:
- → MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
- | | bridged by:

<sup>|</sup> bridged by:

```
MOIB.Constraint.SetDotScalingBridge{Float64, MOI.PositiveSemidefiniteConeTriangle,
1
 → MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
 | may introduce:
1
 * Supported constraint:
→ MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle}
 * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.GeometricMeanCone
| bridged by:
MOIB.Constraint.GeoMeantoRelEntrBridge{Float64, MOI.VectorOfVariables,
MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
 l mav introduce:
* Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.RelativeEntropyCone
 | bridged by:
 MOIB.Constraint.RelativeEntropyBridge{Float64, MOI.ScalarAffineFunction{Float64},
1
→ MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
 | may introduce:
* Unsupported constraint:
 1
 MOI.ScalarAffineFunction{Float64}-in-MOI.GreaterThan{Float64}
\hookrightarrow
 | | bridged by:
 MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64},
\hookrightarrow \quad \texttt{MOI.Nonnegatives, MOI.ScalarAffineFunction{Float64}} \}
| | | may introduce:
 * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
1
 * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.ExponentialCone
1
 1
 * Supported variable: MOI.Reals
1
 * Supported variable: MOI.Nonnegatives
* Supported variable: MOI.Reals
1
* Supported constraint:
→ MOI.VectorAffineFunction{Float64}-in-MOI.Scaled{MOI.PositiveSemidefiniteConeTriangle}
```

This time, JuMP used a MOI.Bridges.Constraint.GeoMeantoRelEntrBridge to reformulate the constraint into a set of MOI.RelativeEntropyCone constraints, which were further reformulated into a set of supported MOI.ExponentialCone constraints.

Since the two models are equivalent, we can conclude that for this particular model, the formulations have similar performance.

In general though, the performance of a particular reformulation is problem- and solver-specific. Therefore, JuMP chooses to minimize the number of bridges in the default reformulation, leaving you to explore alternative formulations using the tools and techniques shown in this tutorial.

### 8.12 Example: fitting of circles and ellipses

This tutorial was generated using Literate.jl. Download the source as a .jl file.

Ellipse fitting is a common task in data analysis and computer vision and is of key importance in many application areas. In this tutorial we show how to fit an ellipse to a set of points using a conic optimization approach.

### **Required packages**

This tutorial uses the following packages:

```
using JuMP
import Clarabel
import Clustering
import DSP
import Images
import LinearAlgebra
import LinearOperatorCollection as LOC
import Plots
import RegularizedLeastSquares as RLS
import Wavelets
```

### Parametrization of an ellipse

An ellipse is a set of the form:

$$\mathcal{E} = \{\xi : (\xi - c)^{\top} D(\xi - c) = r^2\}$$

where  $c \in \mathbb{R}^2$  is the center of the ellipse,  $D \in \mathbb{R}^{2 \times 2} \succ 0$  is a symmetric positive definite matrix and r > 0.

We can setup a coordinate system  $(x, y) \in \mathbb{X} \times \mathbb{Y}$  with  $x, y \ge 0$ . We use definition (1) to write an ellipse as the root of a quadratic form in homogeneous coordinates:

$$\begin{bmatrix} \xi \\ 1 \end{bmatrix}^T \begin{bmatrix} Q & d \\ d^T & e \end{bmatrix} \begin{bmatrix} \xi \\ 1 \end{bmatrix} = 0$$

where:

$$Q = D \tag{8.17}$$

$$d = -Dc \tag{8.18}$$

$$e = c^T D c - r^2 \tag{8.19}$$

The residual distance  $r_0$  of a random point  $\xi_0 = (x_0, y_0)$  to the ellipse is then given by:

$$r_0 \triangleq \begin{bmatrix} \xi_0 \\ 1 \end{bmatrix}^T \begin{bmatrix} Q & d \\ d^T & e \end{bmatrix} \begin{bmatrix} \xi_0 \\ 1 \end{bmatrix}$$

The value of  $r_0$  is positive if the point is outside the ellipse, zero if it is on the ellipse and negative if it is inside the ellipse. We also see we only need six parameters to uniquely define an ellipse.

### **Helper functions**

We define some helper functions to help us visualize the results.

```
function plot_dwt(x, sz = (500, 500))
 return Plots.heatmap(
 x;
 color = :grays,
 aspect_ratio = 1,
 cbar = false,
 xlims = (0, size(x, 2)),
 ylims = (0, size(x, 1)),
 size = sz,
 dpi = 300,
)
end
function normalize(x::AbstractArray)
 l, u = extrema(x)
 return (x .- l) ./ (u - l)
end
```

normalize (generic function with 1 method)

### Reading the test image

To test our ellipse-fitting algorithm we need a test image with elliptical features. For our test image we will use an image of the cartwheel galaxy, captured by the James Webb Space Telescope. Galaxies come in many shapes and sizes, elliptical being one of them.

This is just a toy problem with little scientific value, but you can imagine how the rotation and position of elliptical galaxies can be useful information to astronomers.

```
filename = joinpath(@_DIR_, "..", "..", "assets", "cartwheel_galaxy.png")
img = Images.load(filename);
```

We convert the image to gray scale so that we can work with a single channel.

```
img_gray = Images.Gray.(img)
Images.mosaicview(img, img_gray; nrow = 1)
```



Instead of operating on the entire image, we select a region of interest (ROI) which is a subset of  $\mathbb{X} \times \mathbb{Y}$ .

```
sz = 256
X_c = 600
Y_c = 140
X = X_c:X_c+sz-1
Y = Y_c:Y_c+sz-1
roi = (X, Y)
img_roi = img[roi...]
img_gray_roi = img_gray[roi...]
Images.mosaicview(img_roi, img_gray_roi; nrow = 1)
```



## **Extracting image features**

We cannot directly fit ellipses to the image, so we need to extract features that enable us to find the elliptical galaxies.

The first step is to find a sparse representation of the image. We will use the discrete wavelet transform (DWT) in combination with the Iterative Shrinking and Thresholding (ISTA) algorithm to denoise the image and find a sparse representation. This will remove redundant information and make it much easier to detect the edges of galaxies.

Finding a sparse representation amounts to solving the following optimization problem:

$$\min_{x} \frac{1}{2} \|y - \Phi x\|_{2}^{2} + \lambda \|x\|_{1}$$

where y is the noisy image,  $\Phi$  is the sparsifying basis, x is the sparse representation of our image, and  $\lambda$  is the regularization parameter which we set to 0.1.

To work with our image we must first convert it to Float64.

x = convert(Array{Float64}, img\_gray\_roi)

```
256×256 Matrix{Float64}:
0.0823529 0.0588235 0.054902 ... 0.0431373 0.06666667 0.054902
0.0784314 0.0588235 0.054902 0.054902 0.06666667 0.0431373
0.0705882 0.0666667 0.0627451 0.0588235 0.0705882 0.0588235
0.0627451 0.06666667 0.0705882 0.054902 0.0627451 0.0627451
0.0509804 0.06666667 0.0745098 0.0509804 0.0509804 0.0588235

 0.0509804
 0.0627451
 0.054902
 ...
 0.054902
 0.0588235
 0.0588235
 0.0588235
 0.0588235

 0.0588235
 0.0784314
 0.054902
 ...
 0.0784314
 0.0627451
 0.0431373

0.0627451 \quad 0.0509804 \quad 0.0509804 \qquad 0.06666667 \quad 0.0588235 \quad 0.06666667
0.0627451 \quad 0.0470588 \quad 0.054902 \qquad \quad 0.0470588 \quad 0.054902 \quad \quad 0.0901961
0.0588235 0.0509804 0.0470588 0.0470588 0.0470588 0.054902
 Π
 П
0.054902 0.054902 0.0392157 0.054902 0.06666667 0.0745098
 0.0470588 0.054902 0.0509804 0.054902 0.054902 0.0666667
 0.0470588 0.0431373 0.0470588 0.0823529 0.0588235 0.0588235
0.0470588 0.0509804 0.0431373 ... 0.0784314 0.0588235 0.054902
0.0509804 0.0509804 0.0431373 0.054902 0.0627451 0.054902
0.0431373 0.0470588 0.0431373 0.054902 0.0705882 0.0666667
0.0509804 0.0470588 0.0509804 0.06666667 0.0705882 0.0588235
0.0509804 0.0431373 0.0470588
 0.0627451 0.0627451 0.0509804
 0.054902 0.0470588 0.0470588 ... 0.0823529 0.0705882 0.0666667
```

We then use ISTA in combination with our wavelet sparsifying basis  $\Psi$  obtained from the family of Daubechies wavelets. We use the db4 wavelet which has 4 vanishing moments. We set the number of iterations to 15.

```
reg = RLS.L1Regularization(0.1);

Φ = L0C.Wavelet0p(

 Float64;

 shape = size(x),

 wt = Wavelets.wavelet(Wavelets.WT.db4),
```

);

solver = RLS.createLinearSolver(RLS.OptISTA, \$\overline{4}; reg = reg, iterations = 15);

The sampled image in wavelet domain is given by:

 $b = \Phi * vec(x);$ 

We can now solve the optimization problem to find the sparse representation of the image.

```
x_approx = RLS.solve!(solver, b)
x_approx = reshape(x_approx, size(x));
x_final = normalize(x_approx)
Images.mosaicview(x, Images.Gray.(x_final); nrow = 1)
```



We then use a binarization algorithm to map each grayscale pixel  $(x_i, y_i)$  to a binary value so  $x_i, y_i \rightarrow \{0, 1\}$ .

```
x_bin = Images.binarize(x_final, Images.Otsu(); nbins = 128)
x_bin = convert(Array{Bool}, x_bin)
plt = plot_dwt(img_roi)
Plots.heatmap!(x_bin; color = :grays, alpha = 0.45)
```



## Edge detection and clustering

Now that we have our binary image, we can use edge detection to find the edges of the galaxies. We will use the Sobel operator for this task.

```
function edge_detector(
 f_smooth::Matrix{Float64},
 dl::Float64 = 0.1,
 d2::Float64 = 0.1,
)
 rows, cols = size(f_smooth)
 gradient_magnitude = zeros(Float64, rows, cols)
 laplacian_magnitude = zeros(Float64, rows, cols)
 sobel_x = [-1 0 1; -2 0 2; -1 0 1]
 sobel_y = [-1 -2 -1; 0 0 0; 1 2 1]
```

```
sobel_xx = [-1 2 -1; 2 -4 2; -1 2 -1]
sobel_yy = [-1 2 -1; 2 -4 2; -1 2 -1]
gradient_x = DSP.conv(f_smooth, sobel_x)
gradient_magnitude = sqrt.(gradient_x .^ 2 + gradient_y .^ 2)
gradient_xx = DSP.conv(f_smooth, sobel_xx)
gradient_yy = DSP.conv(f_smooth, sobel_yy)
laplacian_magnitude = sqrt.(gradient_xx .^ 2 + gradient_yy .^ 2)
return (gradient_magnitude .> d1) .& (laplacian_magnitude .< d2)</pre>
```

end

edge\_detector (generic function with 3 methods)

We apply the Sobel operator to the binary image:

```
edges = edge_detector(convert(Matrix{Float64}, x_bin), le-1, le2)
edges = Images.thinning(edges; algo = Images.GuoAlgo())
```

```
258×258 BitMatrix:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0
 0
 0
 0
 0 0 0
 0 0 0 0 0 0 0
 0 0
 00
 0 0
 0
 0
 0
0
 0
 0
 0
 0
 0
 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0
 0
 0
 0
0
 0
 0
 0
 0
 0
0 0 0 0 0
 0 0 0 0
 0 0
 0
0 0 0 0 0
 000
 0
 0 0
 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0
 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0
 0 0 0
0 0 0 0 0
 0 0 0 0
 0
 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0
 0
 0 0
0
 000
 0
 0 0 0 0 0 0
 0 0
 000
 0
 0
 0
 0
Π
 Π
 Π
 00
 0
 0 0 0 0 0 0 0
 0 0
 0 0 0 0
 0
 0
 0
 0
 0
 0
 0
0
 0
 0
 0
 0
 0 0 0 0
 0
 0
 0 0
 0
0
 0
 0
 0
 0
 0 ...
 0
 0
 0
 0
 0
 0
 0
0
 0
 0
 0
 0 0 0 0
 0 0 0
 0 0
 00
 0 0
 0
 0
 0
 0
 0
 0
 0
 0
 00
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0
 0
 0 0
 0
0
 0
 0
 0
 0
 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0
0
 0 0 0 0
 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

And finally we cluster the edges using dbscan so we can fit ellipses to individual galaxies. We can control the minimum size of galaxies by changing the minimum cluster size.

```
points = findall(edges)
points = getfield.(points, :I)
points = hcat([p[1] for p in points], [p[2] for p in points])'
result = Clustering.dbscan(
 convert(Matrix{Float64}, points),
```

3.0; min\_neighbors = 2, min\_cluster\_size = 15,

)

Clustering.DbscanResult(Clustering.DbscanCluster[Clustering.DbscanCluster(41, [12, 13, 14, 15, 16,
→ 17, 18, 19, 20, 21 ... 47, 48, 49, 51, 52, 53, 54, 55, 56, 57], Int64[]),
→ Clustering.DbscanCluster(28, [83, 84, 85, 86, 87, 88, 89, 90, 91, 92 ... 102, 105, 106, 107,
→ 108, 109, 110, 111, 112, 113], Int64[]), Clustering.DbscanCluster(17, [121, 122, 125, 126, 127,
→ 130, 131, 134, 135, 138, 139, 144, 145, 146, 147, 148, 149], Int64[]),
→ Clustering.DbscanCluster(26, [152, 153, 154, 155, 156, 157, 158, 159, 160, 161 ... 168, 169,
→ 170, 171, 172, 173, 174, 175, 176, 177], Int64[]), Clustering.DbscanCluster(27, [178, 179, 180,
→ 181, 182, 183, 184, 185, 186, 187 ... 195, 196, 197, 198, 199, 200, 201, 202, 203, 204],
→ Int64[]), Clustering.DbscanCluster(53, [220, 221, 222, 223, 224, 226, 227, 228, 229, 230 ...
→ 297, 298, 301, 302, 303, 304, 305, 308, 309, 310], Int64[]), Clustering.DbscanCluster(62, [237,
→ 238, 239, 245, 246, 247, 248, 249, 256, 257 ... 336, 337, 338, 341, 342, 343, 344, 345, 346,
→ 347], Int64[])], [12, 83, 121, 152, 178, 220, 237], [41, 28, 17, 26, 27, 53, 62], [0, 0, 0, 0,
→ 0, 0, 0, 0, 0, 0 ... 0, 0, 0, 0, 0, 0, 0, 0])

The result of the clustering is a list of clusters to which we will assign a unique color. Each cluster is a list of points that belong to the same galaxy.

```
clusters = result.clusters
N_clusters = length(clusters)
colors = Plots.distinguishable_colors(N_clusters + 1)[2:end]
plt = plot_dwt(x_final)
for (i, cluster) in enumerate(clusters)
 p_cluster = points[:, cluster.core_indices]
 Plots.scatter!(
 plt,
 p_cluster[2, :],
 p_cluster[1, :];
 color = colors[i],
 label = false,
 markerstrokewidth = 0,
 markersize = 1.5,
)
end
Plots.plot!(
 plt:
 axis = false,
 legend = :topleft,
 legendcolumns = 1,
 legendfontsize = 12,
)
```



### **Fitting ellipses**

Now that we have all the ingredients we can finally start fitting ellipses. We will use a conic optimization approach to do so since it is a very natural way to represent ellipses.

First, we define the residual distance definition (6) of a point to an ellipse in JuMP:

```
function create_ellipse_model(E::Array{Tuple{Int,Int},1}, e = 1e-5)
N = length(E)
model = Model(Clarabel.Optimizer)
set_silent(model)
@variable(model, 0[1:2, 1:2], PSD)
@variable(model, d[1:2])
@variable(model, e)
@expression(
 model,
```

```
r[i in 1:N],
 [Ξ[i][1], Ξ[i][2], 1]' * [Q d; d' e] * [Ξ[i][1], Ξ[i][2], 1]
)
return model
end
```

create\_ellipse\_model (generic function with 2 methods)

### **Objective 1: Minimize the total squared distance**

For our first objective we will minimize the total squared distance of all points to the ellipse. Hence we will use the sum of the squared distances as our objective function, also known as the  $L^2$  norm:

$$\min_{Q,d,e} P_{\mathsf{res}}(\mathcal{E}) = \min_{Q,d,e} \sum_{i \in N} d_{\mathsf{res}}^2(\xi_i, \mathcal{E}) = \min_{Q,d,e} ||d_{\mathsf{res}}||_2^2$$

This problem is equivalent to:

$$\min_{Q,d,e,} \tag{8.20}$$

s.t. 
$$\geq d_{\text{res}}^2(\xi_i, \mathcal{E}) \quad \forall i \in N$$
 (8.21)

And hence can be modelled as a second-order cone program (SOCP) using MOI.RotatedSecondOrderCone as follows:

```
ellipses_C1 = Dict{Symbol, Any}[]
for (i, cluster) in enumerate(clusters)
 p_cluster = points[:, cluster.core_indices]
 \Xi = [(point[1], point[2]) \text{ for point in eachcol}(p_cluster)]
 model = create_ellipse_model(E)
 @variable(model, \zeta \ge 0)
 @constraint(
 model,
 [1 / 2; ζ; model[:r]] in
 MOI.RotatedSecondOrderCone(2 + length(model[:r]))
)
 (dobjective(model, Min, \zeta)
 optimize!(model)
 assert is solved and feasible(model)
 Q, d, e = value.(model[:Q]), value.(model[:d]), value.(model[:e])
 push!(ellipses_C1, Dict(:Q => Q, :d => d, :e => e))
end
W, H = size(img_roi)
x_range = 0:1:W
y_range = 0:1:H
X, Y = [x for x in x_range], [y for y in y_range]
function ellipse_eq(x, y, Q, d, e)
 Z = zeros(length(x), length(y))
```

```
for i in eachindex(x), j in eachindex(y)
 \xi = [x[i], y[j]]
 \label{eq:zi} \mathsf{Z[i, j]} \ = \ [\xi; \ 1.0]' \ * \ [Q \ d; \ d' \ e] \ * \ [\xi; \ 1.0]
 end
 return Z
end
for ellipse in ellipses_C1
 Q, d, e = ellipse[:0], ellipse[:d], ellipse[:e]
 Z_sq = ellipse_eq(X, Y, Q, d, e)
 Plots.contour!(
 plt,
 x_range,
 y_range,
 Z_sq;
 levels = [0.0],
 linewidth = 2,
 color = :red,
 cbar = false,
)
end
plt
```



### **Objective 2: Minimize the maximum residual distance**

For our second objective we will minimize the maximum residual distance of all points to the ellipse:

$$\min_{Q,d,e} \max_{\xi_i \in \mathcal{F}} d_{\mathsf{res}}(\xi_i, \mathcal{E}) = \min_{Q,d,e} ||d_{\mathsf{res}}||_{\infty}$$

This objective can be implemented in JuMP using MOI.NormInfinityCone as follows:

```
ellipses_C2 = Dict{Symbol,Any}[]
for (i, cluster) in enumerate(clusters)
 p_cluster = points[:, cluster.core_indices]
 Ξ = [(point[1], point[2]) for point in eachcol(p_cluster)]
 model = create_ellipse_model(Ξ)
```

```
N = length(\Xi)
 @variable(model, \zeta)
 @constraint(
 model,
 [ζ; model[:r]] in MOI.NormInfinityCone(1 + length(model[:r]))
)
 <code>@objective(model, Min, \zeta)</code>
 optimize!(model)
 assert_is_solved_and_feasible(model; allow_almost = true)
 Q, d, e = value.(model[:Q]), value.(model[:d]), value.(model[:e])
 push!(ellipses_C2, Dict(:Q => Q, :d => d, :e => e))
end
for ellipse in ellipses_C2
 Q, d, e = ellipse[:0], ellipse[:d], ellipse[:e]
 Z_sq = ellipse_eq(X, Y, Q, d, e)
 Plots.contour!(
 plt,
 x_range,
 y_range,
 Z_sq;
 levels = [0.0],
 linewidth = 2,
 color = :green,
 cbar = false,
)
end
Plots.scatter!([0], [0]; color = :red, label = "Squared (Obj. 1)")
Plots.scatter!([0], [0]; color = :green, label = "Min-Max (Obj. 2)")
```



### 8.13 Example: quantum state discrimination

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial solves the problem of quantum state discrimination.

The purpose of this tutorial to demonstrate how to solve problems involving complex-valued decision variables and the HermitianPSDCone. See Complex number support for more details.

## **Required packages**

This tutorial uses the following packages:

using JuMP import Clarabel import LinearAlgebra

### Formulation

A d-dimensional quantum state,  $\rho$ , can be defined by a complex-valued Hermitian matrix with a trace of 1. Assume we have N d-dimensional quantum states,  $\{\rho_i\}_{i=1}^N$ , each of which is equally likely.

The goal of the quantum state discrimination problem is to choose a positive operator-valued measure (POVM),  $\{E_i\}_{i=1}^N$ , such that if we observe  $E_i$  then the most probable state that we are in is  $\rho_i$ .

Each POVM element,  $E_i$ , is a complex-valued Hermitian matrix, and there is a requirement that  $\sum_{i=1}^{N} E_i = \mathbf{I}$ .

To choose a POVM, we want to maximize the probability that we guess the quantum state correctly. This can be formulated as the following optimization problem:

$$\max_{E} \quad \frac{1}{N} \sum_{i=1}^{N} \operatorname{tr}(\rho_{i} E_{i})$$
  
s.t. 
$$\sum_{i=1}^{N} E_{i} = \mathbf{I}$$
$$E_{i} \succeq 0 \; \forall i = 1, \dots, N.$$

### Data

To setup our problem, we need N d-dimensional quantum states. To keep the problem simple, we use N = 2 and d = 2.

N, d = 2, 2

```
(2, 2)
```

We then generated N random d-dimensional quantum states:

```
function random_state(d)
 x = randn(ComplexF64, (d, d))
 y = x * x'
 return LinearAlgebra.Hermitian(y / LinearAlgebra.tr(y))
end
 p = [random_state(d) for i in 1:N]
```

```
2-element Vector{LinearAlgebra.Hermitian{ComplexF64, Matrix{ComplexF64}}}:
[0.9049983143615443 + 0.0im -0.11984167294191471 + 0.2224268161763913im; -0.11984167294191471 -

→ 0.2224268161763913im 0.09500168563845571 + 0.0im]
[0.2503060704439256 + 0.0im -0.16997258971668044 - 0.09226624832975384im; -0.16997258971668044 +

→ 0.09226624832975384im 0.7496939295560743 + 0.0im]
```

### **JuMP** formulation

To model the problem in JuMP, we need a solver that supports positive semidefinite matrices:

```
model = Model(Clarabel.Optimizer)
set silent(model)
```

Then, we construct our set of E variables:

```
E = [@variable(model, [1:d, 1:d] in HermitianPSDCone()) for i in 1:N]
```

```
2-element Vector{LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},

→ Matrix{GenericAffExpr{ComplexF64, VariableRef}}}:

[_[1] _[2] + _[4] im; _[2] - _[4] im _[3]]

[_[5] _[6] + _[8] im; _[6] - _[8] im _[7]]
```

Here we have created a vector of matrices. This is different to other modeling languages such as YALMIP, which allow you to create a multi-dimensional array in which 2-dimensional slices of the array are Hermitian matrices.

We also need to enforce the constraint that  $\sum\limits_{i=1}^N E_i = \mathbf{I}$ :

```
@constraint(model, sum(E) == LinearAlgebra.I)
```

$$\begin{bmatrix} -1 + -5 - 1 & -2 + -6 + -4im + -8im \\ -2 + -6 - -4im - -8im & -3 + -7 - 1 \end{bmatrix} \in \operatorname{Zeros}()$$

This constraint is a complex-valued equality constraint. In the solver, it will be decomposed onto two types of equality constraints: one to enforce equality of the real components, and one to enforce equality of the imaginary components.

Our objective is to maximize the expected probability of guessing correctly:

```
@objective(
 model,
 Max,
 sum(real(LinearAlgebra.tr(ρ[i] * E[i])) for i in 1:N) / N,
)
```

Now we optimize:

```
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : Clarabel
| Termination
| | termination_status : OPTIMAL
| | result_count : 1
| L raw_status : SOLVED
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | dual_objective_value : 8.64063e-01
| L dual_objective_value : 8.64063e-01
| Work counters
| solve_time (sec) : 1.24622e-03
| barrier_iterations : 8
```

The probability of guessing correctly is:

objective\_value(model)

0.8640627425181733

When N = 2, there is a known analytical solution of:

 $0.5 + 0.25 * sum(LinearAlgebra.svdvals(\rho[1] - \rho[2]))$ 

```
0.8640627582954737
```

proving that we found the optimal solution.

Finally, the optimal POVM is:

solution = [value.(e) for e in E]

```
2-element Vector{Matrix{ComplexF64}}:
[0.9496066849311735 + 0.0im 0.034413986810280475 + 0.21603121721998986im; 0.034413986810280475 -

→ 0.21603121721998986im 0.050393307139617316 + 0.0im]
[0.05039331506882663 + 0.0im -0.034413986810280475 - 0.21603121721998986im; -0.034413986810280475

→ + 0.21603121721998986im 0.9496066928603828 + 0.0im]
```

## Тір

Duality plays a large role in solving conic optimization models. Depending on the solver, it can be more efficient to solve the dual of this problem instead of the primal. If performance is an issue, see the Dualization tutorial for more details.

### **Alternative formulation**

The formulation above includes N Hermitian matrices and a set of linear equality constraints. We can simplify the problem by replacing  $E_N$  with  $E_N = I - \sum_{i=1}^{N-1} E_i$ . This results in:

```
model = Model(Clarabel.Optimizer)
set_silent(model)
E = [@variable(model, [1:d, 1:d] in HermitianPSDCone()) for i in 1:N-1]
E_N = LinearAlgebra.Hermitian(LinearAlgebra.I - sum(E))
@constraint(model, E_N in HermitianPSDCone())
push!(E, E_N)
```

```
2-element Vector{LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},

→ Matrix{GenericAffExpr{ComplexF64, VariableRef}}}:

[_[1] _[2] + _[4] im; _[2] - _[4] im _[3]]

[-_[1] + 1 -_[2] - _[4] im; -_[2] + _[4] im -_[3] + 1]
```

The objective can also be simplified, by observing that it is equivalent to:

```
@objective(model, Max, real(LinearAlgebra.dot(p, E)) / N)
```

Then we can check that we get the same solution:

optimize!(model)
assert\_is\_solved\_and\_feasible(model)
solution\_summary(model)

```
solution_summary(; result = 1, verbose = false)
| solver_name : Clarabel
| Termination
| | termination_status : OPTIMAL
| | result_count : 1
| L raw_status : SOLVED
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | dual_objective_value : 8.64063e-01
| L dual_objective_value : 8.64063e-01
| Work counters
| solve_time (sec) : 1.31532e-03
| barrier iterations : 9
```

objective\_value(model)

0.8640627545817933

## **Chapter 9**

# Algorithms

## 9.1 Benders decomposition

This tutorial was generated using Literate.jl. Download the source as a .jl file.

### This tutorial was originally contributed by Shuvomoy Das Gupta.

This tutorial describes how to implement Benders decomposition in JuMP. It uses the following packages:

using JuMP import Gurobi import HiGHS import Printf

### Theory

Benders decomposition is a useful algorithm for solving convex optimization problems with a large number of variables. It works best when a larger problem can be decomposed into two (or more) smaller problems that are individually much easier to solve.

This tutorial demonstrates Benders decomposition on the following mixed-integer linear program:

min 
$$c_1(x) + c_2(y)$$
  
subject to  $f_1(x) \in S_1$   
 $f_2(y) \in S_2$   
 $f_3(x, y) \in S_3$   
 $x \in \mathbb{Z}^m$   
 $y \in \mathbb{R}^n$ 

where the functions f and c are linear, and the sets S are inequality sets like  $\geq l, \leq u$ , or = b.

Any mixed integer programming problem can be written in the form above.

If there are relatively few integer variables, and many more continuous variables, then it may be beneficial to decompose the problem into a small problem containing only integer variables and a linear program containing only continuous variables. Hopefully, the linear program will be much easier to solve in isolation than in the full mixed-integer linear program.

For example, if we knew a feasible solution for  $\bar{x}$ , we could obtain a solution for y by solving:

$$\begin{array}{ll} V_2(\bar{x}) = \min & c_2(y) \\ \text{subject to} & f_2(y) \in S_2 \\ & f_3(x,y) \in S_3 \\ & x = \bar{x} \ [\pi] \\ & y \in \mathbb{R}^n \end{array}$$

Note that we have included a "copy" of the x variable to simplify computing  $\pi$ , which is the dual of  $V_2$  with respect to  $\bar{x}$ .

Because this model is a linear program, it is easy to solve.

Replacing the  $c_2(y)$  component of the objective in our original problem with  $V_2$  yields:

$$\begin{array}{ll} V_1 = \min & c_1(x) + V_2(x) \\ \text{subject to} & f_1(x) \in S_1 \\ & x \in \mathbb{Z}^m. \end{array}$$

This problem looks a lot simpler to solve because it involves only x and a subset of the constraints, but we need to do something else with  $V_2$  first.

Because  $\bar{x}$  is a constant that appears on the right-hand side of the constraints,  $V_2$  is a convex function with respect to  $\bar{x}$ , and the dual variable  $\pi$  is a subgradient of  $V_2(x)$  with respect to x. Therefore, if we have a candidate solution  $x_k$ , then we can solve  $V_2(x_k)$  and obtain a feasible dual vector  $\pi_k$ . Using these values, we can construct a first-order Taylor-series approximation of  $V_2$  about the point  $x_k$ :

$$V_2(x) \ge V_2(x_k) + \pi_k^+ (x - x_k).$$

By convexity, we know that this inequality holds for all x, and we call these inequalities *cuts*.

Benders decomposition is an iterative technique that replaces  $V_2(x)$  with a new decision variable  $\theta$ , and approximates it from below using cuts:

$$egin{aligned} V_1^K = \min & c_1(x) + \theta \ & & & & subject ext{to} & f_1(x) \in S_1 \ & & & & x \in \mathbb{Z}^m \ & & & & ext{$\theta \geq M$} \ & & & & & ext{$\theta \geq V_2(x_k) + \pi_k^ op (x-x_k)$} & orall k = 1, \dots, K. \end{aligned}$$

This integer program is called the *first-stage* subproblem.

To generate cuts, we solve  $V_1^K$  to obtain a candidate first-stage solution  $x_k$ , then we use that solution to solve  $V_2(x_k)$ . Then, using the optimal objective value and dual solution from  $V_2$ , we add a new cut to form  $V_1^{K+1}$  and repeat.

### Bounds

Due to convexity, we know that  $V_2(x) \ge \theta$  for all x. Therefore, the optimal objective value of  $V_1^K$  provides a valid *lower* bound on the objective value of the full problem. In addition, if we take a feasible solution for x from the first-stage problem, then  $c_1(x) + V_2(x)$  is a valid *upper* bound on the objective value of the full problem.

Benders decomposition uses the lower and upper bounds to determine when it has found the global optimal solution.

### Monolithic problem

As an example problem, we consider the following variant of The max-flow problem, in which there is a binary variable to decide whether to open each arc for a cost of 0.1 unit, and we can open at most 11 arcs:

```
G = [
 0 3 2 2 0 0 0 0
 00005100
 0 0 0 0 1 3 1 0
 0 0 0 0 0 1 0 0
 0 0 0 0 0 0 0 4
 0 0 0 0 0 0 0 0 2
 0 0 0 0 0 0 0 4
 0 0 0 0 0 0 0 0 0
]
n = size(G, 1)
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:n, 1:n], Bin)
@variable(model, y[1:n, 1:n] >= 0)
@constraint(model, sum(x) <= 11)</pre>
@constraint(model, [i = 1:n, j = 1:n], y[i, j] <= G[i, j] * x[i, j])</pre>
@constraint(model, [i = 2:n-1], sum(y[i, :]) == sum(y[:, i]))
@objective(model, Min, 0.1 * sum(x) - sum(y[1, :]))
optimize!(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
 : HiGHS
- solver name
- Termination
| | termination_status : OPTIMAL
| | result_count : 1
: kHighsModelStatusOptimal
| L objective_bound : -5.10000e+00
Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : NO_SOLUTION
| | objective_value : -5.10000e+00
| ^L relative_gap : 0.00000e+00
^L Work counters
 - solve time (sec) : 2.27332e-03
 - simplex iterations : 15
```

The optimal objective value is -5.1:

objective\_value(model)

-5.1

and the optimal flows are:

```
\label{eq:function_optimal_flows(x)} \mbox{return [(i, j) => x[i, j] for i in 1:n for j in 1:n if x[i, j] > 0]} \mbox{end} end
```

monolithic\_solution = optimal\_flows(value.(y))

```
9-element Vector{Pair{Tuple{Int64, Int64}, Float64}}:
(1, 2) => 3.0
(1, 3) => 2.0
(1, 4) => 1.0
(2, 5) => 3.0
(3, 5) => 1.0
(3, 6) => 1.0
(4, 6) => 1.0
(5, 8) => 4.0
(6, 8) => 2.0
```

### **Iterative method**

#### Warning

This is a basic implementation for pedagogical purposes. We haven't discussed any of the computational tricks that are required to build a performant implementation for large-scale problems. See In-place iterative method for one improvement that helps computation time.

We start by formulating the first-stage subproblem. It includes the x variables, and the constraints involving only x, and the terms in the objective containing only x. We also need an initial lower bound on the cost-to-go variable  $\theta$ . One valid lower bound is to assume that we do not pay for opening arcs, and there is flow all the arcs.

```
M = -sum(G)
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:n, 1:n], Bin)
```

```
@variable(model, θ >= M)
@constraint(model, sum(x) <= 11)
@objective(model, Min, 0.1 * sum(x) + θ)
model</pre>
```

A JuMP Model
 solver: HiGHS
 objective\_sense: MIN\_SENSE
 L objective\_function\_type: AffExpr
 num\_variables: 65
 num\_constraints: 66
 AffExpr in MOI.LessThan{Float64}: 1
 VariableRef in MOI.GreaterThan{Float64}: 1
 VariableRef in MOI.ZeroOne: 64
 Names registered in the model
 L :x, :0

For the next step, we need a function that takes a first-stage candidate solution x and returns the optimal solution from the second-stage subproblem:

```
function solve_subproblem(x_bar)
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[i in 1:n, j in 1:n] == x_bar[i, j])
@variable(model, y[1:n, 1:n] >= 0)
@constraint(model, [i = 1:n, j = 1:n], y[i, j] <= G[i, j] * x[i, j])
@constraint(model, [i = 2:n-1], sum(y[i, :]) == sum(y[:, i]))
@objective(model, Min, -sum(y[1, :]))
optimize!(model)
assert_is_solved_and_feasible(model; dual = true)
return (obj = objective_value(model), y = value.(y), π = reduced_cost.(x))
end</pre>
```

solve\_subproblem (generic function with 1 method)

Note that solve\_subproblem returns a NamedTuple of the objective value, the optimal primal solution for y, and the optimal dual solution for  $\pi$ , which we obtained from the reduced\_cost of the x variables.

We're almost ready for our optimization loop, but first, here's a helpful function for logging:

```
function print_iteration(k, args...)
 f(x) = Printf.@sprintf("%12.4e", x)
 println(lpad(k, 9), " ", join(f.(args), " "))
 return
end
```

### CHAPTER 9. ALGORITHMS

### print\_iteration (generic function with 1 method)

We also need to put a limit on the number of iterations before termination:

MAXIMUM\_ITERATIONS = 100

100

And a way to check if the lower and upper bounds are close-enough to terminate:

ABSOLUTE\_OPTIMALITY\_GAP = 1e-6

1.0e-6

Now we're ready to iterate Benders decomposition:

```
println("Iteration Lower Bound Upper Bound
 Gap")
for k in 1:MAXIMUM_ITERATIONS
 optimize!(model)
 assert_is_solved_and_feasible(model)
 lower_bound = objective_value(model)
 x_k = value.(x)
 ret = solve_subproblem(x_k)
 upper_bound = (objective_value(model) - value(\theta)) + ret.obj
 gap = abs(upper_bound - lower_bound) / abs(upper_bound)
 print_iteration(k, lower_bound, upper_bound, gap)
 if gap < ABSOLUTE_OPTIMALITY_GAP</pre>
 println("Terminating with the optimal solution")
 break
 end
 cut = @constraint(model, \theta \ge ret.obj + sum(ret.\pi .* (x .- x k)))
 @info "Adding the cut $(cut)"
end
```

```
Iteration Lower Bound Upper Bound Gap

1 -2.9000e+01 0.0000e+00 Inf

[Info: Adding the cut 3 x[1,2] + 2 x[1,3] + 2 x[1,4] + \theta \ge 0

2 -6.7000e+00 3.0000e-01 2.3333e+01

[Info: Adding the cut 5 x[2,5] + x[3,5] + x[2,6] + 3 x[3,6] + x[4,6] + x[3,7] + \theta \ge 0

3 -6.5000e+00 5.0000e-01 1.4000e+01

[Info: Adding the cut x[3,7] + 4 x[5,8] + 2 x[6,8] + \theta \ge 0

4 -6.2000e+00 -4.2000e+00 4.7619e-01

[Info: Adding the cut 3 x[1,2] + x[3,5] + 2 x[6,8] + 4 x[7,8] + \theta \ge 0

5 -6.1000e+00 -4.1000e+00 4.8780e-01
```

```
[Info: Adding the cut 3 x[1,2] + x[3,5] + 3 x[3,6] + x[4,6] + x[3,7] + \theta \ge 0
6 -6.1000e+00 -4.1000e+00 4.8780e-01
[Info: Adding the cut 3 x[1,2] + 2 x[1,3] + x[4,6] + \theta \ge 0
7 -5.1000e+00 -5.1000e+00 0.0000e+00
Terminating with the optimal solution
```

Finally, we can obtain the optimal solution:

```
optimize!(model)
assert_is_solved_and_feasible(model)
x_optimal = value.(x)
optimal_ret = solve_subproblem(x_optimal)
iterative_solution = optimal_flows(optimal_ret.y)
```

```
9-element Vector{Pair{Tuple{Int64, Int64}, Float64}}:
(1, 2) => 3.0
(1, 3) => 2.0
(1, 4) => 1.0
(2, 5) => 3.0
(3, 5) => 1.0
(3, 6) => 1.0
(4, 6) => 1.0
(5, 8) => 4.0
(6, 8) => 2.0
```

which is the same as the monolithic solution:

```
iterative_solution == monolithic_solution
```

true

and it has the same objective value:

objective\_value(model)

-5.1

### **Callback method**

The Iterative method section implemented Benders decomposition using a loop. In each iteration, we re-solved the first-stage subproblem to generate a candidate solution. However, modern MILP solvers such as CPLEX, Gurobi, and GLPK provide lazy constraint callbacks which allow us to add new cuts *while the solver is running*.

### CHAPTER 9. ALGORITHMS

This can be more efficient than an iterative method because we can avoid repeating work such as solving the root node of the first-stage MILP at each iteration.

Tip We use Gurobi for this model because HiGHS does not support lazy constraints. For more information on callbacks, read the page Solver-independent callbacks.

As before, we construct the same first-stage subproblem:

```
optimizer = Gurobi.Optimizer
lazy_model = Model(optimizer)
set_silent(lazy_model)
@variable(lazy_model, x[1:n, 1:n], Bin)
@variable(lazy_model, \theta \ge M)
@constraint(lazy_model, sum(x) <= 11)
@objective(lazy_model, Min, 0.1 * sum(x) + \theta)
lazy_model
```

```
A JuMP Model
 solver: Gurobi
 objective_sense: MIN_SENSE
 L objective_function_type: AffExpr
 num_variables: 65
 num_constraints: 66
 AffExpr in MOI.LessThan{Float64}: 1
 VariableRef in MOI.GreaterThan{Float64}: 1
 L VariableRef in MOI.ZeroOne: 64
 Names registered in the model
 L :x, :0
```

What differs is that we write a callback function instead of a loop:

```
number_of_subproblem_solves = 0
function my_callback(cb_data)
 status = callback node status(cb_data, lazy_model)
 if status != MOI.CALLBACK_NODE_STATUS_INTEGER
 # Only add the constraint if `x` is an integer feasible solution
 return
 end
 x_k = callback_value.(cb_data, x)
 \theta_k = callback_value(cb_data, \theta)
 global number_of_subproblem_solves += 1
 ret = solve_subproblem(x_k)
 if \theta_k < (ret.obj - 1e-6)
 # Only add the constraint if \theta_k violates the constraint
 cut = @build_constraint(\theta \ge ret.obj + sum(ret.\pi .* (x .- x_k)))
 MOI.submit(lazy_model, MOI.LazyConstraint(cb_data), cut)
 end
 return
end
```

### CHAPTER 9. ALGORITHMS

set\_attribute(lazy\_model, MOI.LazyConstraintCallback(), my\_callback)

Now when we optimize!, our callback is run:

```
optimize!(lazy_model)
assert_is_solved_and_feasible(lazy_model)
```

For this model, the callback algorithm required more solves of the subproblem:

number\_of\_subproblem\_solves

### 17

But for larger problems, you can expect the callback algorithm to be more efficient than the iterative algorithm.

Finally, we can obtain the optimal solution:

```
x_optimal = value.(x)
optimal_ret = solve_subproblem(x_optimal)
callback_solution = optimal_flows(optimal_ret.y)
```

```
9-element Vector{Pair{Tuple{Int64, Int64}, Float64}}:
(1, 2) => 3.0
(1, 3) => 2.0
(1, 4) => 1.0
(2, 5) => 3.0
(3, 5) => 1.0
(3, 6) => 1.0
(4, 6) => 1.0
(5, 8) => 4.0
(6, 8) => 2.0
```

which is the same as the monolithic solution:

callback\_solution == monolithic\_solution



### In-place iterative method

Our implementation of the iterative method has a problem: every time we need to solve the subproblem, we must rebuild it from scratch. This is expensive, and it can be the bottleneck in the solution process. We can improve our implementation by using re-using the subproblem between solves.

First, we create our first-stage problem as usual:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:n, 1:n], Bin)
@variable(model, θ >= M)
@constraint(model, sum(x) <= 11)
@objective(model, Min, 0.1 * sum(x) + θ)
model</pre>
```

```
A JuMP Model
 solver: HiGHS
 objective_sense: MIN_SENSE
 L objective_function_type: AffExpr
 num_variables: 65
 num_constraints: 66
 AffExpr in MOI.LessThan{Float64}: 1
 VariableRef in MOI.GreaterThan{Float64}: 1
 L VariableRef in MOI.ZeroOne: 64
 Names registered in the model
 L :x, :0
```

Then, instead of building the subproblem in a function, we build it once here:

```
subproblem = Model(HiGHS.Optimizer)
set_silent(subproblem)
@variable(subproblem, x_copy[i in 1:n, j in 1:n])
@variable(subproblem, y[1:n, 1:n] >= 0)
@constraint(subproblem, [i = 1:n, j = 1:n], y[i, j] <= G[i, j] * x_copy[i, j])
@constraint(subproblem, [i = 2:n-1], sum(y[i, :]) == sum(y[:, i]))
@objective(subproblem, Min, -sum(y[1, :]))
subproblem</pre>
```

A JuMP Model
 solver: HiGHS
 objective\_sense: MIN\_SENSE
 L objective\_function\_type: AffExpr
 num\_variables: 128
 num\_constraints: 134
 L AffExpr in MOI.EqualTo{Float64}: 6
 L AffExpr in MOI.LessThan{Float64}: 64
 L VariableRef in MOI.GreaterThan{Float64}: 64
 L Names registered in the model
 L :x\_copy, :y

Our function to solve the subproblem is also slightly different because we need to fix the value of the  $x_copy$  variables to the value of x from the first-stage problem:

```
function solve_subproblem(model, x)
 fix.(model[:x_copy], x)
 optimize!(model)
 assert_is_solved_and_feasible(model; dual = true)
 return (
 obj = objective_value(model),
 y = value.(model[:y]),
 π = reduced_cost.(model[:x_copy]),
)
end
```

solve\_subproblem (generic function with 2 methods)

Now we're ready to iterate our in-place Benders decomposition:

```
println("Iteration Lower Bound Upper Bound
 Gap")
for k in 1:MAXIMUM_ITERATIONS
 optimize!(model)
 assert_is_solved_and_feasible(model)
 lower_bound = objective_value(model)
 x_k = value.(x)
 ret = solve_subproblem(subproblem, x_k)
 upper_bound = (objective_value(model) - value(\theta)) + ret.obj
 gap = abs(upper_bound - lower_bound) / abs(upper_bound)
 print_iteration(k, lower_bound, upper_bound, gap)
 if gap < ABSOLUTE_OPTIMALITY_GAP</pre>
 println("Terminating with the optimal solution")
 break
 end
 cut = @constraint(model, \theta \ge ret.obj + sum(ret.\pi .* (x .- x_k)))
 @info "Adding the cut $(cut)"
end
```

```
Iteration Lower Bound Upper Bound
 Gap
 1 -2.9000e+01 0.0000e+00
 Inf
[Info: Adding the cut 3 x[1,2] + 2 x[1,3] + 2 x[1,4] + \theta \ge 0
 2 -6.7000e+00 3.0000e-01 2.3333e+01
[Info: Adding the cut 5 x[2,5] + x[3,5] + x[2,6] + 3 x[3,6] + x[4,6] + x[3,7] + \theta \ge 0
 3 -6.5000e+00 5.0000e-01 1.4000e+01
[Info: Adding the cut x[3,7] + 4 x[5,8] + 2 x[6,8] + \theta \ge 0
 4 -6.2000e+00 -4.2000e+00 4.7619e-01
[Info: Adding the cut 3 x[1,2] + x[3,5] + 2 x[6,8] + 4 x[7,8] + \theta \ge 0
 5 -6.1000e+00 -4.1000e+00 4.8780e-01
[Info: Adding the cut 3 x[1,2] + x[3,5] + 3 x[3,6] + x[4,6] + x[3,7] + \theta \ge 0
 6 -6.1000e+00 -4.1000e+00 4.8780e-01
[Info: Adding the cut 3 x[1,2] + 2 x[1,3] + x[4,6] + \theta \ge 0
 7 -5.1000e+00 -5.1000e+00 0.0000e+00
Terminating with the optimal solution
```

Finally, we can obtain the optimal solution:

optimize!(model)
assert\_is\_solved\_and\_feasible(model)
x\_optimal = value.(x)
optimal\_ret = solve\_subproblem(subproblem, x\_optimal)
inplace\_solution = optimal\_flows(optimal\_ret.y)

```
9-element Vector{Pair{Tuple{Int64, Int64}, Float64}}:
(1, 2) => 3.0
(1, 3) => 2.0
(1, 4) => 1.0
(2, 5) => 3.0
(3, 5) => 1.0
(3, 6) => 1.0
(4, 6) => 1.0
(5, 8) => 4.0
(6, 8) => 2.0
```

which is the same as the monolithic solution:

```
inplace_solution == monolithic_solution
```

true

### **Feasibility cuts**

So far, we have discussed only Benders optimality cuts. However, for some first-stage values of x, the subproblem might be infeasible. The solution is to add a Benders feasibility cut:

$$v_k + u_k^\top (x - x_k) \le 0$$

where  $u_k$  is a dual unbounded ray of the subproblem and  $v_k$  is the intercept of the unbounded ray.

As a variation of our example which leads to infeasibilities, we add a constraint that  $sum(y) \ge 1$ . This means we need a choice of first-stage x for which at least one unit can flow.

The first-stage problem remains the same:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:n, 1:n], Bin)
@variable(model, θ >= M)
@constraint(model, sum(x) <= 11)
@objective(model, Min, 0.1 * sum(x) + θ)
model</pre>
```

```
A JuMP Model
 solver: HiGHS
 objective_sense: MIN_SENSE
 L objective_function_type: AffExpr
 num_variables: 65
 num_constraints: 66
 AffExpr in MOI.LessThan{Float64}: 1
 VariableRef in MOI.GreaterThan{Float64}: 1
 L VariableRef in MOI.ZeroOne: 64
 Names registered in the model
 L :x, :0
```

But the subproblem has a new constraint that  $sum(y) \ge 1$ :

```
subproblem = Model(HiGHS.Optimizer)
set_silent(subproblem)
We need to turn presolve off so that HiGHS will return an infeasibility
certificate.
set_attribute(subproblem, "presolve", "off")
@variable(subproblem, x_copy[i in 1:n, j in 1:n])
@variable(subproblem, y[1:n, 1:n] >= 0)
@constraint(subproblem, sum(y) >= 1) # <--- THIS IS NEW
@constraint(subproblem, [i = 1:n, j = 1:n], y[i, j] <= G[i, j] * x_copy[i, j])
@constraint(subproblem, [i = 2:n-1], sum(y[i, :]) == sum(y[:, i]))
@objective(subproblem, Min, -sum(y[1, :]))
subproblem</pre>
```

A JuMP Model
| solver: HiGHS
| objective\_sense: MIN\_SENSE
| L objective\_function\_type: AffExpr
| num\_variables: 128
| num\_constraints: 135
| L AffExpr in MOI.EqualTo{Float64}: 6
| L AffExpr in MOI.GreaterThan{Float64}: 1
| L AffExpr in MOI.LessThan{Float64}: 64
L VariableRef in MOI.GreaterThan{Float64}: 64
L Names registered in the model
L :x\_copy, :y

The function to solve the subproblem now checks for feasibility, and returns the dual objective value and an dual unbounded ray if the subproblem is infeasible:

```
solve_subproblem_with_feasibility (generic function with 1 method)
```

Now we're ready to iterate our in-place Benders decomposition:

```
println("Iteration Lower Bound Upper Bound
 Gap")
for k in 1:MAXIMUM_ITERATIONS
 optimize!(model)
 assert_is_solved_and_feasible(model)
 lower_bound = objective_value(model)
 x k = value.(x)
 ret = solve_subproblem_with_feasibility(subproblem, x_k)
 if ret.is feasible
 # Benders Optimality Cuts
 upper_bound = (objective_value(model) - value(\theta)) + ret.obj
 gap = abs(upper_bound - lower_bound) / abs(upper_bound)
 print_iteration(k, lower_bound, upper_bound, gap)
 if gap < ABSOLUTE_OPTIMALITY_GAP</pre>
 println("Terminating with the optimal solution")
 break
 end
 \operatorname{@constraint(model, \theta \ge ret.obj + sum(ret.\pi .* (x .- x_k)))}
 else
 # Benders Feasibility Cuts
 cut = @constraint(model, ret.v + sum(ret.u .* (x .- x_k)) <= 0)</pre>
 @info "Adding the feasibility cut $(cut)"
 end
end
```

```
[Info: Adding the feasibility cut -3 x[1,2] - 4 x[1,3] - 2 x[1,4] - 10 x[2,5] - x[3,5] - 4 x[6,8]
\hookrightarrow - 4 x[7,8] \leq -1
[Info: Adding the feasibility cut -6 x[1,2] - 6 x[1,3] - 4 x[1,4] - 4 x[5,8] - 2 x[6,8] \leq -1
[Info: Adding the feasibility cut -6 x[1,2] - 6 x[1,3] - 2 x[4,6] - 4 x[5,8] - 2 x[6,8] \leq -1
[Info: Adding the feasibility cut -6 x[1,2] - 2 x[1,4] - 2 x[3,5] - 6 x[3,6] - x[4,6] - 2 x[3,7] -
\rightarrow 4 x[5,8] - 2 x[6,8] - 4 x[7,8] \leq -1
[Info: Adding the feasibility cut -6 x[1,2] - 2 x[3,5] - 6 x[3,6] - 2 x[4,6] - 2 x[3,7] - 4 x[5,8]
\hookrightarrow - 2 x[6,8] - 4 x[7,8] \leq -1
[Info: Adding the feasibility cut -5 x[2,5] - x[3,5] - x[2,6] - 3 x[3,6] - x[4,6] - x[3,7] - 8
\rightarrow x[5,8] - 4 x[6,8] - 8 x[7,8] \leq -1
[Info: Adding the feasibility cut -2 x[1,3] - 6 x[1,4] - 9.99999999999998 x[2,5] - x[3,5] - 3
\rightarrow x[7,8] \leq -0.999999999999998
[Info: Adding the feasibility cut -1.999999999999987 x[1,3] - 5.9999999999999997 x[1,4] -

ightarrow 5.99999999999999964 x[3,6] - x[3,7] - 3.9999999999999 x[7,8] ≤ -0.99999999999999999998
[Info: Adding the feasibility cut -3 x[1,2] - 5.9999999999997 x[1,4] - 9.99999999999999 x[2,5]
\rightarrow - 2.9999999999999999 x[3,5] - 1.999999999999 x[2,6] - 8.9999999999999 x[3,6] -
\rightarrow 1.99999999999999999 x[3,7] - 3.999999999999 x[7,8] \leq -0.99999999999999999999
[Info: Adding the feasibility cut -3 x[1,2] - 6 x[1,4] - 2 x[2,6] - 9 x[3,6] - 2 x[3,7] - 12
\hookrightarrow x[5,8] - 4 x[7,8] \leq -1
[Info: Adding the feasibility cut -3 x[1,2] - 6 x[1,4] - 2 x[2,6] - 9 x[3,6] - 3 x[3,7] - 12
\hookrightarrow x[5,8] \leq -1
[Info: Adding the feasibility cut -2 \times [1,4] - 3 \times [3,7] - 12 \times [5,8] - 6 \times [6,8] \le -1
[Info: Adding the feasibility cut -3 \times [3,7] - 12 \times [5,8] - 6 \times [6,8] \le -1
[Info: Adding the feasibility cut -15.00000000000007 x[2,5] - 3.00000000000013 x[3,5] - x[3,7]
\hookrightarrow - 6 x[6,8] - 8 x[7,8] \leq -1
[Info: Adding the feasibility cut -9 x[1,2] - 2 x[1,3] - 2 x[3,5] - 6 x[3,6] - 3 x[4,6] - x[3,7] -
\hookrightarrow 4 x[7,8] \leq -1
[Info: Adding the feasibility cut -8.99999999999998 x[1,2] - 3.00000000000001 x[3,5] -
→ 8.9999999999999999 x[3,6] - 2.9999999999999999 x[4,6] - 12.00000000000004 x[7,8] ≤
\hookrightarrow -1.00000000000000000
[Info: Adding the feasibility cut -15 x[2,5] - 3 x[3,5] - 3 x[2,6] - 9 x[3,6] - 3 x[4,6] - 12
\rightarrow x[7,8] \leq -1
[Info: Adding the feasibility cut -15 x[2,5] - 3 x[3,5] - 3 x[2,6] - 9 x[3,6] - 3 x[4,6] - 3
\rightarrow x[3.7] \leq -1
[Info: Adding the feasibility cut -9.00000000000004 x[1,2] - 6 x[1,4] - 3 x[3,5] - 9 x[3,6] -
\hookrightarrow 2.99999999999999 x[3,7] \leq -1
[Info: Adding the feasibility cut -6.0000000000003 x[1,2] - 4.0000000000002 x[1,3] -
\hookrightarrow x[3,6] - 4 x[7,8] \leq -1
[Info: Adding the feasibility cut -2 x[1,4] - 5 x[2,5] - x[3,5] - x[2,6] - 3 x[3,6] - 8 x[5,8] - 4
\hookrightarrow x[6,8] - 12 x[7,8] \leq -1
[Info: Adding the feasibility cut -5 \times [2,5] - \times [3,5] - 8 \times [5,8] - 6 \times [6,8] - 12 \times [7,8] \le -1
[Info: Adding the feasibility cut -2.000000000000004 x[1,3] - 2 x[1,4] - 2.999999999999999987
\hookrightarrow \ x[2,6] \ - \ 5.999999999999998 \ x[3,6] \ - \ 1.99999999999999 \ x[4,6] \ - \ 12 \ x[5,8] \ - \ 8 \ x[7,8] \ \le \ -1
[Info: Adding the feasibility cut -6 x[1,3] - 2 x[1,4] - 3 x[2,6] - 2 x[4,6] - 12 x[5,8] - 8
\hookrightarrow x[7,8] \leq -1
 29 -2.8700e+01 -7.0000e-01 4.0000e+01
[Info: Adding the feasibility cut -2 x[1,4] - 15 x[2,5] - 3 x[3,5] - 3 x[3,7] - 6 x[6,8] ≤ -1
 31 -1.1400e+01 -4.0000e-01 2.7500e+01
 32 -8.2000e+00 -1.2000e+00 5.8333e+00
 33 -8.0000e+00 -4.0000e+00 1.0000e+00
 34 -5.3000e+00 -3.3000e+00 6.0606e-01
 35 -5.2000e+00 -1.2000e+00 3.3333e+00
```
```
 36
 -5.2000e+00
 -4.2000e+00
 2.3810e-01

 37
 -5.2000e+00
 -4.2000e+00
 2.3810e-01

 38
 -5.1000e+00
 -5.1000e+00
 0.0000e+00

 Terminating with the optimal solution
 -5.1000e+00
 -5.1000e+00
```

Finally, we can obtain the optimal solution:

```
optimize!(model)
assert_is_solved_and_feasible(model)
x_optimal = value.(x)
optimal_ret = solve_subproblem(subproblem, x_optimal)
feasible_inplace_solution = optimal_flows(optimal_ret.y)
```

```
9-element Vector{Pair{Tuple{Int64, Int64}, Float64}}:
(1, 2) => 3.0
(1, 3) => 2.0
(1, 4) => 1.0
(2, 5) => 3.0
(3, 5) => 1.0
(3, 6) => 1.0
(4, 6) => 1.0
(5, 8) => 4.0
(6, 8) => 2.0
```

which is the same as the monolithic solution (because  $sum(y) \ge 1$  in the monolithic solution):

feasible\_inplace\_solution == monolithic\_solution

true

## 9.2 Column generation

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to demonstrate the column generation algorithm. As an example, it solves the Cutting stock problem.

This tutorial uses the following packages:

using JuMP import DataFrames import HiGHS import Plots import SparseArrays

# Background

The cutting stock problem is about cutting large rolls of paper into smaller pieces.

We denote the set of possible sized pieces that a roll can be cut into by  $i \in 1, ..., I$ . Each piece i has a width,  $w_i$ , and a demand,  $d_i$ . The width of the large roll is W.

Our objective is to minimize the number of rolls needed to meet all demand.

Here's the data that we are going to use in this tutorial:

```
struct Piece
 w::Float64
 d::Int
end
struct Data
 pieces::Vector{Piece}
 W::Float64
end
function Base.show(io::IO, d::Data)
 println(io, "Data for the cutting stock problem:")
 println(io, " W = (d.W)")
 println(io, "with pieces:")
 println(io, " i w_i d_i")
 println(io, " -----")
 for (i, p) in enumerate(d.pieces)
 println(io, lpad(i, 4), " ", lpad(p.w, 5), " ", lpad(p.d, 3))
 end
 return
end
function get_data()
 data = [
 75.0 38
 75.0 44
 75.0 30
 75.0 41
 75.0 36
 53.8 33
 53.0 36
 51.0 41
 50.2 35
 32.2 37
 30.8 44
 29.8 49
 20.1 37
 16.2 36
 14.5 42
 11.0 33
 8.6 47
 8.2 35
 6.6 49
 5.1 42
]
 return Data([Piece(data[i, 1], data[i, 2]) for i in axes(data, 1)], 100.0)
end
```

data = get\_data()

```
Data for the cutting stock problem:
 W = 100.0
with pieces:
 i w_i d_i

 1 75.0 38
 2 75.0 44
 3 75.0 30
 4 75.0 41
 5 75.0
 36
 6 53.8 33
 7 53.0 36
 8 51.0 41
 9 50.2 35
 10 32.2 37
 11 30.8 44
 12 29.8 49
 13 20.1 37
 14 16.2 36
 15 14.5 42
 16 11.0 33
 17
 8.6 47
 18
 8.2 35
 19
 6.6 49
 20
 5.1 42
```

# **Mathematical formulation**

To formulate the cutting stock problem as a mixed-integer linear program, we assume that there is a set of large rolls  $j = 1, \ldots, J$  to use. Then, we introduce two classes of decision variables:

- $x_{ij} \ge 0$ , integer,  $\forall i = 1, ..., I, \ j = 1, ..., J$
- $y_j \in \{0, 1\}, \forall j = 1, \dots, J.$

 $y_j$  is a binary variable that indicates if we use roll j, and  $x_{ij}$  counts how many pieces of size i that we cut from roll j.

Our mixed-integer linear program is therefore:

$$\min\sum_{j=1}^{J} y_j \tag{9.1}$$

s.t. 
$$\sum_{i=1}^{I} w_i x_{ij} \le W y_j \qquad \qquad \forall j = 1, \dots, J \qquad (9.2)$$

$$\sum_{i=1}^{J} x_{ij} \ge d_i \qquad \qquad \forall i = 1, \dots, I$$
(9.3)

$$\begin{aligned} x_{ij} &\geq 0 & \forall i = 1, \dots, I, j = 1, \dots, J & (9.4) \\ x_{ij} &\in \mathbb{Z} & \forall i = 1, \dots, I, j = 1, \dots, J & (9.5) \end{aligned}$$

$$\forall j = 1, \dots, J \tag{9.6}$$

The objective is to minimize the number of rolls that we use, and the two constraints ensure that we respect the total width of each large roll and that we satisfy demand exactly.

The JuMP formulation of this model is:

 $y_j \in \{0, 1\}$ 

```
I = length(data.pieces)
J = 1_000 # Some large number
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:I, 1:J] >= 0, Int)
@variable(model, y[1:J], Bin)
@objective(model, Min, sum(y))
@constraint(model, [i in 1:I], sum(x[i, :]) >= data.pieces[i].d)
@constraint(
 model,
 [j in 1:J],
 sum(data.pieces[i].w * x[i, j] for i in 1:I) <= data.W * y[j],
);</pre>
```

Unfortunately, we can't solve this formulation for realistic instances because it takes a very long time to solve. (Try removing the time limit.)

set\_time\_limit\_sec(model, 5.0)
optimize!(model)
solution\_summary(model)

```
solution_summary(; result = 1, verbose = false)
+ solver_name : HiGHS
+ Termination
| + termination_status : TIME_LIMIT
| + result_count : 1
| + raw_status : kHighsModelStatusTimeLimit
| L objective_bound : 2.93000e+02
+ Solution (result = 1)
| + primal_status : FEASIBLE_POINT
```

```
| | dual_status: N0_SOLUTION| | objective_value: 4.11000e+02
| | dual_objective_value : NaN
| ^L relative_gap : 2.87105e-01
L Work counters
 - solve_time (sec) : 5.05008e+00
 simplex_iterations : 21016
 barrier_iterations : -1
 L node_count
 : 0
```

However, there is a formulation that solves much faster, and that is to use a column generation scheme.

## **Column generation theory**

The key insight for column generation is to recognize that feasible columns in the x matrix of variables encode cutting patterns.

For example, if we look only at the roll j = 1, then a feasible solution is:

- $x_{1,1} = 1$  (1 unit of piece #1)
- $x_{13,1} = 1$  (1 unit of piece #13)
- All other  $x_{i,1} = 0$

Another solution is

- $x_{20,1} = 19$  (19 unit of piece #20)
- All other  $x_{i,1} = 0$

Cutting patterns like  $x_{1,1} = 1$  and  $x_{2,1} = 1$  are infeasible because the combined length is greater than W.

Since there are a finite number of ways that we could cut a roll into a valid cutting pattern, we could create a set of all possible cutting patterns  $p=1,\ldots,P$ , with data  $a_{i,p}$  indicating how many units of piece i we cut in pattern p. Then, we can formulate our mixed-integer linear program as:

$$\min\sum_{p=1}^{P} x_p \tag{9.8}$$

s.t. 
$$\sum_{p=1}^{P} a_{ip} x_p \ge d_i \qquad \qquad \forall i = 1, \dots, I$$
(9.9)

$$\begin{aligned} x_p \geq 0 & \forall p = 1, \dots, P \\ x_p \in \mathbb{Z} & \forall p = 1, \dots, P \end{aligned}$$
 (9.10)

Unfortunately, there will be a very large number of these patterns, so it is often intractable to enumerate all columns  $p = 1, \ldots, P$ .

Column generation is an iterative algorithm that starts with a small set of initial patterns, and then cleverly chooses new columns to add to the main MILP so that we find the optimal solution without having to enumerate every column.

(9.11)

## Choosing the initial set of patterns

For the initial set of patterns, we create a trivial cutting pattern which cuts as many units of piece i as will fit.

```
patterns = map(1:I) do i
 n_pieces = floor(Int, data.W / data.pieces[i].w)
 return SparseArrays.sparsevec([i], [n_pieces], I)
end
```

```
20-element Vector{SparseArrays.SparseVector{Int64, Int64}}:
sparsevec([1], [1], 20)
 sparsevec([2], [1], 20)
 sparsevec([3], [1], 20)
 sparsevec([4], [1], 20)
 sparsevec([5], [1], 20)
 sparsevec([6], [1], 20)
 sparsevec([7], [1], 20)
 sparsevec([8], [1], 20)
 sparsevec([9], [1], 20)
 sparsevec([10], [3], 20)
 sparsevec([11], [3], 20)
 sparsevec([12], [3], 20)
 sparsevec([13], [4], 20)
 sparsevec([14], [6], 20)
 sparsevec([15], [6], 20)
 sparsevec([16], [9], 20)
 sparsevec([17], [11], 20)
 sparsevec([18], [12], 20)
 sparsevec([19], [15], 20)
 sparsevec([20], [19], 20)
```

We can visualize the patterns as follows:

```
.....
 cutting_locations(data::Data, pattern::SparseArrays.SparseVector)
A function which returns a vector of the locations along the roll at which to
cut in order to produce pattern `pattern`.
.....
function cutting_locations(data::Data, pattern::SparseArrays.SparseVector)
 locations = Float64[]
 offset = 0.0
 for (i, c) in zip(SparseArrays.findnz(pattern)...)
 for in 1:c
 offset += data.pieces[i].w
 push!(locations, offset)
 end
 end
 return locations
end
function plot_patterns(data::Data, patterns)
 plot = Plots.bar(;
```

```
xlims = (0, length(patterns) + 1),
 ylims = (0, data.W),
 xlabel = "Pattern",
 ylabel = "Roll length",
)
 for (i, p) in enumerate(patterns)
 locations = cutting_locations(data, p)
 Plots.bar!(
 plot,
 fill(i, length(locations)),
 reverse(locations);
 bar_width = 0.6,
 label = false,
 color = "#90caf9",
)
 end
 return plot
end
```

```
plot_patterns(data, patterns)
```



# The base problem

Using the initial set of patterns, we can create and optimize our base model:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:length(patterns)] >= 0, Int)
@objective(model, Min, sum(x))
@constraint(model, demand[i in 1:I], patterns[i]' * x >= data.pieces[i].d)
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
- solver_name
 : HiGHS
- Termination
| + result_count : 1
| + raw_status : kHighsModelStatusOptimal
| L objective_bound : 4.21000e+02
Solution (result = 1)
| + primal_status : FEASIBLE_POINT
| + dual_status : NO_SOLUTION
| + objective_value : 4.21000e+02
| | dual_objective_value : NaN
| ^L relative_gap : 0.00000e+00
^L Work counters
 solve_time (sec) : 1.74284e-04
 - simplex_iterations : 0
 barrier_iterations : -1
 L node_count
 : 0
```

This solution requires 421 rolls. This solution is sub-optimal because the model does not contain the full set of possible patterns.

How do we find a new column that leads to an improved solution?

## Choosing new columns

Column generation chooses a new column by relaxing the integrality constraint on x and looking at the dual variable  $\pi_i$  associated with demand constraint i.

For example, the dual of demand [13] is:

```
unset_integer.(x)
optimize!(model)
assert_is_solved_and_feasible(model; dual = true)
\pi_13 = dual(demand[13])
```

#### 0.25

Using the economic interpretation of the dual variable, we can say that a one unit increase in demand for piece i will cost an extra  $\pi_i$  rolls. Alternatively, we can say that a one unit increase in the left-hand side (for

example, due to a new cutting pattern) will *save* us  $\pi_i$  rolls. Therefore, we want a new column that maximizes the savings associated with the dual variables, while respecting the total width of the roll:

$$\max\sum_{i=1}^{I} \pi_i y_i \tag{9.12}$$

s.t. 
$$\sum_{i=1}^{I} w_i y_i \le W$$
(9.13)

$$y_i \ge 0 \qquad \qquad \forall i = 1, \dots, I \tag{9.14}$$

 $\forall i = 1, \dots, I$  (9.15)

(9.16)

If this problem, called the *pricing problem*, has an objective value greater than 1, then we estimate than adding y as the coefficients of a new column will decrease the objective by more than the cost of an extra roll.

Here is code to solve the pricing problem:

 $y_i \in \mathbb{Z}$ 

```
function solve_pricing(data::Data, π::Vector{Float64})
 I = length(\pi)
 model = Model(HiGHS.Optimizer)
 set silent(model)
 @variable(model, y[1:I] >= 0, Int)
 @constraint(model, sum(data.pieces[i].w * y[i] for i in 1:I) <= data.W)</pre>
 <code>@objective(model, Max, sum(\pi[i] * y[i] for i in 1:I))</code>
 optimize!(model)
 assert_is_solved_and_feasible(model)
 number_of_rolls_saved = objective_value(model)
 if number_of_rolls_saved > 1 + 1e-8
 # Benefit of pattern is more than the cost of a new roll plus some
 # tolerance
 return SparseArrays.sparse(round.(Int, value.(y)))
 end
 return nothing
end
```

solve\_pricing (generic function with 1 method)

If we solve the pricing problem with an artificial dual vector:

solve\_pricing(data, [1.0 / i for i in 1:I])

```
20-element SparseArrays.SparseVector{Int64, Int64} with 3 stored entries:

[1] = 1

[17] = 1

[20] = 3
```

the solution is a roll with 1 unit of piece #1, 1 unit of piece #17, and 3 units of piece #20.

If we solve the pricing problem with a dual vector of zeros, then the benefit of the new pattern is less than the cost of a roll, and so the function returns nothing:

```
solve_pricing(data, zeros(I))
```

## Iterative algorithm

Now we can combine our base model with the pricing subproblem in an iterative column generation scheme:

```
while true
 # Solve the linear relaxation
 optimize!(model)
 assert is solved and feasible(model; dual = true)
 # Obtain a new dual vector
 \pi = dual.(demand)
 # Solve the pricing problem
 new_pattern = solve_pricing(data, \pi)
 # Stop iterating if there is no new pattern
 if new_pattern === nothing
 @info "No new patterns, terminating the algorithm."
 break
 end
 push!(patterns, new_pattern)
 # Create a new column
 push!(x, @variable(model, lower_bound = 0))
 # Update the objective coefficient of the new column
 set_objective_coefficient(model, x[end], 1.0)
 # Update the non-zeros in the coefficient matrix
 for (i, count) in zip(SparseArrays.findnz(new_pattern)...)
 set_normalized_coefficient(demand[i], x[end], count)
 end
 println("Found new pattern. Total patterns = $(length(patterns))")
end
```

```
Found new pattern. Total patterns = 21
Found new pattern. Total patterns = 22
Found new pattern. Total patterns = 23
Found new pattern. Total patterns = 24
Found new pattern. Total patterns = 25
Found new pattern. Total patterns = 26
Found new pattern. Total patterns = 27
Found new pattern. Total patterns = 28
Found new pattern. Total patterns = 29
Found new pattern. Total patterns = 30
Found new pattern. Total patterns = 31
Found new pattern. Total patterns = 32
Found new pattern. Total patterns = 33
Found new pattern. Total patterns = 34
Found new pattern. Total patterns = 35
[Info: No new patterns, terminating the algorithm.
```

We found lots of new patterns. Here's pattern 21:

patterns[21]

```
20-element SparseArrays.SparseVector{Int64, Int64} with 3 stored entries:
[9] = 1
[13] = 2
[17] = 1
```

### Let's have a look at the patterns now:

```
plot_patterns(data, patterns)
```



# Looking at the solution

Let's see how many of each column we need:

```
solution = DataFrames.DataFrame([
 (pattern = p, rolls = value(x_p)) for (p, x_p) in enumerate(x)
])
filter!(row -> row.rolls > 0, solution)
```

	pattern rolls		
	Int64	Float64	
1	1	38.0	
2	2	44.0	
3	3	30.0	
4	21	0.5	
5	22	10.2	
6	23	14.65	
7	24	23.1	
8	25	11.25	
9	26	21.35	
10	28	4.3	
11	29	19.55	
12	30	11.25	
13	31	17.45	
14	33	36.0	
15	34	11.4	
16	35	41.0	

Since we solved a linear program, some of our columns have fractional solutions. We can create a integer feasible solution by rounding up the orders. This requires 341 rolls:

```
sum(ceil.(Int, solution.rolls))
```

```
341
```

Alternatively, we can re-introduce the integrality constraints and resolve the problem:

```
set_integer.(x)
optimize!(model)
assert_is_solved_and_feasible(model)
solution = DataFrames.DataFrame([
 (pattern = p, rolls = value(x_p)) for (p, x_p) in enumerate(x)
])
filter!(row -> row.rolls > 0, solution)
```

This now requires 334 rolls:

sum(solution.rolls)

```
333.999999999999994
```

Note that this may not be the global minimum because we are not adding new columns during the solution of the mixed-integer problem model (an algorithm known as branch and price). Nevertheless, the column generation algorithm typically finds good integer feasible solutions to an otherwise intractable optimization problem.

	pattern rolls		
	Int64	Float64	
1	1	38.0	
2	2	44.0	
3	3	30.0	
4	21	1.0	
5	22	9.0	
6	23	19.0	
7	24	19.0	
8	25	13.0	
9	26	17.0	
10	28	2.0	
11	29	19.0	
12	30	13.0	
13	31	18.0	
14	33	36.0	
15	34	15.0	
16	35	41.0	

## Next steps

- Our objective function is to minimize the total number of rolls. What is the total length of waste? How
  does that compare to the total demand?
- Writing the optimization algorithm is only part of the challenge. Can you develop a better way to communicate the solution to stakeholders?

# 9.3 Traveling Salesperson Problem

This tutorial was generated using Literate.jl. Download the source as a .jl file.

### This tutorial was originally contributed by Daniel Schermer.

This tutorial describes how to implement the Traveling Salesperson Problem in JuMP using solver-independent lazy constraints that dynamically separate subtours. To be more precise, we use lazy constraints to cut off infeasible subtours only when necessary and not before needed.

It uses the following packages:

using JuMP import Gurobi import Plots import Random import Test

## **Mathematical Formulation**

Assume that we are given a complete graph  $\mathcal{G}(V, E)$  where V is the set of vertices (or cities) and E is the set of edges (or roads). For each pair of vertices  $i, j \in V, i \neq j$  the edge  $(i, j) \in E$  is associated with a weight (or distance)  $d_{ij} \in \mathbb{R}^+$ .

For this tutorial, we assume the problem to be symmetric, that is,  $d_{ij} = d_{ji} \, \forall i,j \in V.$ 

In the Traveling Salesperson Problem, we are tasked with finding a tour with minimal length that visits every vertex exactly once and then returns to the point of origin, that is, a *Hamiltonian cycle* with minimal weight.

To model the problem, we introduce a binary variable,  $x_{ij} \in \{0,1\} \forall i, j \in V$ , that indicates if edge (i, j) is part of the tour or not. Using these variables, the Traveling Salesperson Problem can be modeled as the following integer linear program.

### **Objective Function**

The objective is to minimize the length of the tour (due to the assumed symmetry, the second sum only contains i < j):

$$\min \ \sum_{i \in V} \sum_{j \in V, i < j} d_{ij} x_{ij}.$$

Note that it is also possible to use the following objective function instead:

$$\min \sum_{i \in V} \sum_{j \in V} \frac{d_{ij} x_{ij}}{2}.$$

### Constraints

There are four classes of constraints in our formulation.

First, due to the presumed symmetry, the following constraints must hold:

$$x_{ij} = x_{ji} \quad \forall i, j \in V.$$

Second, for each vertex i, exactly two edges must be selected that connect it to other vertices j in the graph G:

$$\sum_{j \in V} x_{ij} = 2 \quad \forall i \in V.$$

Third, we do not permit loops to occur:

$$x_{ii} = 0 \quad \forall i \in V.$$

The fourth constraint is more complicated. A major difficulty of the Traveling Salesperson Problem arises from the fact that we need to prevent *subtours*, that is, several distinct Hamiltonian cycles existing on subgraphs of G.

Note that the previous constraints *do not* guarantee that the solution will be free of subtours. To this end, by S we label a subset of vertices. Then, for each proper subset  $S \subset V$ , the following constraints guarantee that no subtour may occur:

$$\sum_{i \in S} \sum_{j \in S, i < j} x_{ij} \le |S| - 1 \quad \forall S \subset V.$$

Problematically, we require exponentially many of these constraints as |V| increases. Therefore, we will add these constraints only when necessary.

### Implementation

There are two ways we can eliminate subtours in JuMP, both of which will be shown in what follows:

- iteratively solving a new model that incorporates previously identified subtours,
- or adding violated subtours as *lazy constraints*.

# Data

The vertices are assumed to be randomly distributed in the Euclidean space; thus, the weight (distance) of each edge is defined as follows.

```
function generate_distance_matrix(n; random_seed = 1)
 rng = Random.MersenneTwister(random_seed)
 X = 100 * rand(rng, n)
 Y = 100 * rand(rng, n)
 d = [sqrt((X[i] - X[j])^2 + (Y[i] - Y[j])^2) for i in 1:n, j in 1:n]
 return X, Y, d
end
n = 100
X, Y, d = generate distance matrix(n)
```

```
([9.913970137863682, 70.19797138879542, 50.3261785841856, 87.58412053070398, 95.34654118744876,

→ 50.7810571056071, 78.97511635624403, 7.125413261100788, 13.837807897217225, 39.31891799217675

→ ... 84.87369607977678, 61.680928138712, 5.665730912653899, 15.622563304879634,

→ 36.90767228785501, 70.07597765092129, 79.43901471209098, 46.482254570311675, 68.59072330642508,

→ 86.69884288310024], [96.78179466896867, 56.23453714649542, 67.44638756669107,

→ 7.1115103002265645, 92.78034391338332, 34.57366887562756, 76.53412034001651,

→ 33.078576899782796, 62.27235533684083, 31.31072581673351 ... 1.4463814325218927,

→ 42.253985947804495, 53.81635009641501, 76.50117040708963, 27.74238915740479, 60.20183753580153,

→ 19.81346291572821, 90.60507365183767, 31.10234142135033, 21.085230265206945], [0.0

→ 72.65150307747324 ... 88.07242440917794 107.82340444001487; 72.65150307747324 0.0 ...

→ 25.183536454701592 38.829789264256306; ...; 88.07242440917794 25.183536454701592 ... 0.0

→ 20.69411777577625; 107.82340444001487 38.829789264256306 ... 20.69411777577625 0.0])
```

For the JuMP model, we first initialize the model object. Then, we create the binary decision variables and add the objective function and constraints. By defining the x matrix as Symmetric, we do not need to add explicit constraints that x[i, j] == x[j, i].

```
function build_tsp_model(d, n, optimizer)
model = Model(optimizer)
set_silent(model)
@variable(model, x[1:n, 1:n], Bin, Symmetric)
@objective(model, Min, sum(d .* x) / 2)
@constraint(model, [i in 1:n], sum(x[i, :]) == 2)
@constraint(model, [i in 1:n], x[i, i] == 0)
return model
end
```

build\_tsp\_model (generic function with 1 method)

To search for violated constraints, based on the edges that are currently in the solution (that is, those that have value  $x_{ij} = 1$ ), we identify the shortest cycle through the function subtour(). Whenever a subtour has been identified, a constraint corresponding to the form above can be added to the model.

```
function subtour(edges::Vector{Tuple{Int,Int}}, n)
 shortest_subtour, unvisited = collect(1:n), Set(collect(1:n))
 while !isempty(unvisited)
 this_cycle, neighbors = Int[], unvisited
 while !isempty(neighbors)
 current = pop!(neighbors)
 push!(this cycle, current)
 if length(this cycle) > 1
 pop!(unvisited, current)
 end
 neighbors =
 [j for (i, j) in edges if i == current && j in unvisited]
 end
 if length(this cycle) < length(shortest subtour)</pre>
 shortest_subtour = this_cycle
 end
 end
 return shortest_subtour
end
```

subtour (generic function with 1 method)

Let us declare a helper function selected\_edges() that will be repeatedly used in what follows.

```
function selected_edges(x::Matrix{Float64}, n)
 return Tuple{Int,Int}[(i, j) for i in 1:n, j in 1:n if x[i, j] > 0.5]
end
```

## CHAPTER 9. ALGORITHMS

Other helper functions for computing subtours:

```
subtour(x::Matrix{Float64}) = subtour(selected_edges(x, size(x, 1)), size(x, 1))
subtour(x::AbstractMatrix{VariableRef}) = subtour(value.(x))
```

subtour (generic function with 3 methods)

### **Iterative method**

An iterative way of eliminating subtours is the following.

However, it is reasonable to assume that this is not the most efficient way: whenever a new subtour elimination constraint is added to the model, the optimization has to start from the very beginning.

That way, the solver will repeatedly discard useful information encountered during previous solves (for example, all cuts, the incumbent solution, or lower bounds).

## Info

Note that, in principle, it would also be feasible to add all subtours (instead of just the shortest one) to the model. However, preventing just the shortest cycle is often sufficient for breaking other subtours and will keep the model size smaller.

```
optimizer = Gurobi.Optimizer
iterative_model = build_tsp_model(d, n, optimizer)
optimize!(iterative model)
assert_is_solved_and_feasible(iterative_model)
time_iterated = solve_time(iterative_model)
cycle = subtour(iterative_model[:x])
while 1 < length(cycle) < n</pre>
 println("Found cycle of length $(length(cycle))")
 S = [(i, j) for (i, j) in Iterators.product(cycle, cycle) if i < j]
 @constraint(
 iterative model,
 sum(iterative_model[:x][i, j] for (i, j) in S) <= length(cycle) - 1,</pre>
)
 optimize!(iterative model)
 assert_is_solved_and_feasible(iterative_model)
 global time_iterated += solve_time(iterative_model)
 global cycle = subtour(iterative_model[:x])
```

```
end
```

Set parameter WLSAccessID Set parameter WLSSecret Set parameter LicenseID to value 722777 WLS license 722777 - registered to JuMP Development Found cycle of length 3 Found cycle of length 4 Found cycle of length 3 Found cycle of length 3 Found cycle of length 3 Found cycle of length 4 Found cycle of length 5 Found cycle of length 6 Found cycle of length 5 Found cycle of length 3 Found cycle of length 4 Found cycle of length 6 Found cycle of length 4 Found cycle of length 8 Found cycle of length 5 Found cycle of length 10 Found cycle of length 10 Found cycle of length 15 Found cycle of length 11 Found cycle of length 4 Found cycle of length 22 Found cycle of length 3 Found cycle of length 5 Found cycle of length 21

objective\_value(iterative\_model)

744.6016576596794

time\_iterated

3.9510159492492676

As a quick sanity check, we visualize the optimal tour to verify that no subtour is present:

```
function plot_tour(X, Y, x)
plot = Plots.plot()
for (i, j) in selected_edges(x, size(x, 1))
 Plots.plot!([X[i], X[j]], [Y[i], Y[j]]; legend = false)
end
return plot
```

end

### plot\_tour(X, Y, value.(iterative\_model[:x]))



## Lazy constraint method

A more sophisticated approach makes use of *lazy constraints*. To be more precise, we do this through the subtour\_elimination\_callback() below, which is only run whenever we encounter a new integer-feasible solution.

### Tip

We use Gurobi for this model because HiGHS does not support lazy constraints. For more information on callbacks, read the page Solver-independent callbacks.

As before, we construct the same first-stage subproblem:

```
lazy_model = build_tsp_model(d, n, optimizer)
function subtour_elimination_callback(cb_data)
status = callback_node_status(cb_data, lazy_model)
if status != MOI.CALLBACK_NODE_STATUS_INTEGER
 return # Only run at integer solutions
end
cycle = subtour(callback_value.(cb_data, lazy_model[:x]))
if !(1 < length(cycle) < n)
 return # Only add a constraint if there is a cycle</pre>
```

```
end
 S = [(i, j) for (i, j) in Iterators.product(cycle, cycle) if i < j]
 con = @build_constraint(
 \label{eq:sum} sum(lazy_model[:x][i, j] \mbox{ for (i, j) in S}) \mbox{ <= length(cycle) - 1,}
)
 MOI.submit(lazy_model, MOI.LazyConstraint(cb_data), con)
 return
end
set_attribute(
 lazy_model,
 MOI.LazyConstraintCallback(),
 subtour_elimination_callback,
)
optimize!(lazy_model)
assert_is_solved_and_feasible(lazy_model)
objective_value(lazy_model)
```

744.6016576596794

time\_lazy = solve\_time(lazy\_model)

1.6928150653839111

This finds the same optimal tour:

plot\_tour(X, Y, value.(lazy\_model[:x]))



The solution time is faster than the iterative approach:

Test.@test time\_lazy < time\_iterated</pre>

Test Passed

# 9.4 Rolling horizon problems

This tutorial was generated using Literate.jl. Download the source as a .jl file.

## This tutorial was originally contributed by Diego Tejada.

The purpose of this tutorial is to demonstrate how to use ParametricOptInterface.jl to solve a rolling horizon optimization problem.

The term "rolling horizon" refers to solving a time-dependent model repeatedly, where the planning interval is shifted forward in time during each solution step.

As a motivating example, this tutorial models the operations of a power system with solar generation and a battery.

## **Required packages**

This tutorial uses the following packages

## CHAPTER 9. ALGORITHMS

using JuMP import CSV import DataFrames import HiGHS import ParametricOptInterface as POI import Plots

# The optimization model

The model is a simplified model of a power system's operations with battery storage.

We model the system of a set of time-steps  $t \in 1, \dots, T$  , where each time step is a period of one hour.

There are five types of decision variables in the model:

- Renewable production:  $r_t \ge 0$
- Thermal production:  $0 \le p_t \le \overline{P}$
- Storage level:  $0 \le s_t \le \overline{S}$
- Storage charging:  $0 \le c_t \le \overline{C}$
- Storage discharging:  $0 \leq d_t \leq \overline{D}$

For the purpose of this tutorial, there are three parameters of interest:

- Demand at time  $t: D_t$
- Renewable availability at time  $t: A_t$
- Initial storage:  $S_0$

The objective function to minimize is the total cost of thermal generation:

$$\min \sum_t O \cdot p_t$$

For the constraints, we must balance power generation and consumption in all time periods:

$$p_t + r_t + d_t = D_t + c_t, \forall t$$

We need to account for the dynamics of the battery storage:

$$s_t = s_{t-1} + \eta^c \cdot c_t - \frac{d_t}{\eta^d}, \forall t$$

with the boundary condition that  $s_0 = S_0$ .

Finally, the level of renewable energy production is limited by the quantity of potential solar generation A:

$$r_t \leq A_t, \quad \forall t$$

Solving this problem with a large number of time steps is computationally challenging. A common practice is to use the rolling horizon idea to solve multiple identical problems of a smaller size. These problems differ only in parameters such as demand, renewable availability, and initial storage. By combining the solution of many smaller problems, we can recover a feasible solution to the full problem. However, because we don't optimize the full set of decisions in a single optimization problem, the recovered solution might be suboptimal.

## Parameter definition and input data

There are two main parameters for a rolling horizon implementation: the optimization window and the move forward.

**Optimization Window**: this value defines how many periods (for example, hours) we will optimize each time. For this example, we set the default value to 48 hours, meaning we will optimize two days each time.

optimization\_window = 48;

**Move Forward**: this value defines how many periods (for example, hours) we will move forward to optimize the next optimization window. For this example, we set the default value in 24 hours, meaning we will move one day ahead each time.

move\_forward = 24;

Note that the move forward parameter must be lower or equal to the optimization window parameter to work correctly.

@assert optimization\_window >= move\_forward

Let's explore the input data in file rolling\_horizon.csv. We have a total time horizon of a week (that is, 168 hours), an electricity demand, and a solar production profile.

```
filename = joinpath(@__DIR__, "rolling_horizon.csv")
time_series = CSV.read(filename, DataFrames.DataFrame)
time_series[1:21:end, :]
```

We define the solar investment (for example, 150 MW) to determine the solar production during the operation optimization step.

solar\_investment = 150;

We multiply the level of solar investment by the time series of availability to get actual MW generated.

day	hour	demand_MW	solar_pu
Int64	Int64	Float64	Float64
1	0	51.6	0.0
1	21	59.0	0.0
2	18	80.7	0.0
3	15	69.5	0.00966184
4	12	65.9	0.78744
5	9	83.8	0.628019
6	6	67.4	0.0
7	3	57.5	0.0
	day Int64 1 2 3 4 5 6 7	day         hour           Int64         Int64           1         0           1         21           2         18           3         15           4         12           5         9           6         6           7         3	dayhourdemand_MWInt64Int64Float641051.612159.021880.731569.541265.95983.86667.47357.5

time\_series.solar\_MW = solar\_investment \* time\_series.solar\_pu;

In addition, we can determine some basic information about the rolling horizon, such as the number of data points we have:

total\_time\_length = size(time\_series, 1)

168

and the number of windows that we are going to optimize given the problem's time horizon:

(total\_time\_length + move\_forward - optimization\_window) / move\_forward

6.0

Finally, we can see a plot representing the first two optimization windows and the move forward parameter to have a better idea of how the rolling horizon works.

```
x_series = 1:total_time_length
y_series = [time_series.demand_MW, time_series.solar_MW]
plot_1 = Plots.plot(x_series, y_series; label = ["demand" "solar"])
plot_2 = Plots.plot(x_series, y_series; label = false)
window = [0, optimization_window]
Plots.vspan!(plot_1, window; alpha = 0.25, label = false)
Plots.vspan!(plot_2, move_forward .+ window; alpha = 0.25, label = false)
text_1 = Plots.text("optimization\n window 1", :top, :left, 8)
Plots.annotate!(plot_1, 18, time_series.solar_MW[12], text_1)
text_2 = Plots.text("optimization\n window 2", :top, :left, 8)
Plots.annotate!(plot_2, 42, time_series.solar_MW[12], text_2)
Plots.plot(
 plot_1,
 plot 2;
 layout = (2, 1),
 linewidth = 3,
```

)

```
xticks = 0:12:total_time_length,
xlabel = "Hours",
ylabel = "MW",
```



## JuMP model

We have all the information we need to create a JuMP model to solve a single window of our rolling horizon problem.

As the optimizer, we use POI.Optimizer, which is part of ParametricOptInterface.jl. POI.Optimizer converts the Parameter decision variables into constants in the underlying optimization model, and it efficiently updates the solver in-place when we call set\_parameter\_value which avoids having to rebuild the problem each time we call optimize!.

```
model = Model(() -> POI.Optimizer(HiGHS.Optimizer()))
set_silent(model)
@variables(model, begin
 0 <= r[1:optimization_window]
 0 <= p[1:optimization_window] <= 150
 0 <= s[1:optimization_window] <= 40
 0 <= c[1:optimization_window] <= 10
 0 <= d[1:optimization_window] <= 10
 # Initialize empty parameters. These values will get updated later
 D[t in 1:optimization_window] in Parameter(0)
 A[t in 1:optimization_window] in Parameter(0)</pre>
```

```
$_0 in Parameter(0)
end)
@objective(model, Min, 50 * sum(p))
@constraints(
 model,
 begin
 p .+ r .+ d .== D .+ c
 s[1] == $_0 + 0.9 * c[1] - d[1] / 0.9
 [t in 2:optimization_window], s[t] == s[t-1] + 0.9 * c[t] - d[t] / 0.9
 r .<= A
 end
)
model</pre>
```

```
A JuMP Model
| solver: Parametric Optimizer with HiGHS attached
| objective_sense: MIN_SENSE
| L objective_function_type: AffExpr
| num_variables: 337
| num_constraints: 673
| AffExpr in MOI.EqualTo{Float64}: 96
| AffExpr in MOI.LessThan{Float64}: 48
| VariableRef in MOI.LessThan{Float64}: 192
| L VariableRef in MOI.Parameter{Float64}: 97
L Names registered in the model
L :A, :D, :S_0, :c, :d, :p, :r, :s
```

After the optimization, we can store the results in vectors. It's important to note that despite optimizing for 48 hours (the default value), we only store the values for the "move forward" parameter (for example, 24 hours or one day using the default value). This approach ensures that there is a buffer of additional periods or hours beyond the "move forward" parameter to prevent the storage from depleting entirely at the end of the specified hours.

```
sol_complete = Dict(
 :r => zeros(total_time_length),
 :p => zeros(total_time_length),
 :c => zeros(total_time_length),
 :d => zeros(total_time_length),
 # The storage level is initialized with an initial value
 :s => zeros(total_time_length + 1),
)
sol_windows = Pair{Int,Dict{Symbol,Vector{Float64}}[]
```

Pair{Int64, Dict{Symbol, Vector{Float64}}]

Now we can iterate across the windows of our rolling horizon problem, and at each window, we:

1. update the parameters in the models

- 2. solve the model for that window
- 3. store the results for later analysis

```
offsets = 0:move_forward:total_time_length-optimization_window
for offset in offsets
 # Step 1: update the parameter values over the optimization window
 for t in 1:optimization_window
 set_parameter_value(model[:D][t], time_series[offset+t, :demand_MW])
 set_parameter_value(model[:A][t], time_series[offset+t, :solar_MW])
 end
 # Set the starting storage level as the value from the end of the previous
 # solve. The `+1` accounts for the initial storage value in time step "t=0"
 set_parameter_value(model[:S_0], sol_complete[:s][offset+1])
 # Step 2: solve the model
 optimize!(model)
 # Step 3: store the results of the move_forward values, except in the last
 horizon where we store the full `optimization_window`.
 for t in 1:(offset == last(offsets) ? optimization window : move_forward)
 for key in (:r, :p, :c, :d)
 sol_complete[key][offset+t] = value(model[key][t])
 end
 sol_complete[:s][offset+t+1] = value(model[:s][t])
 end
 sol_window = Dict(key => value.(model[key]) for key in (:r, :p, :s, :c, :d))
 push!(sol_windows, offset => sol_window)
end
```

## Solution

Here is a function to plot the solution at each of the time-steps to help visualize the rolling horizon scheme:

```
function plot_solution(sol; offset = 0, kwargs...)
 plot = Plots.plot(;
 ylabel = "MW",
 xlims = (0, total_time_length),
 xticks = 0:12:total_time_length,
 kwargs...,
)
 y = hcat(sol[:p], sol[:r], sol[:d])
 x = offset .+ (1:size(y, 1))
 if offset == 0
 Plots.areaplot!(x, y; label = ["thermal" "solar" "discharge"])
 Plots.areaplot!(x, -sol[:c]; label = "charge")
 else
 Plots.areaplot!(x, y; label = false)
 Plots.areaplot!(x, -sol[:c]; label = false)
 end
 return plot
end
Plots.plot(
 [plot_solution(sol; offset) for (offset, sol) in sol_windows]...;
```

)

```
layout = (length(sol_windows), 1),
size = (600, 800),
margin = 3Plots.mm,
```



We can re-use the function to plot the recovered solution of the full problem:



plot\_solution(sol\_complete; offset = 0, xlabel = "Hour")

# **Final remark**

ParametricOptInterface.jl offers an easy way to update the parameters of an optimization problem that will be solved several times, as in the rolling horizon implementation. It has the benefit of avoiding rebuilding the model each time we want to solve it with new information in a new window.

# 9.5 Parallelism

The purpose of this tutorial is to give a brief overview of parallelism in Julia as it pertains to JuMP, and to explain some of the things to be aware of when writing parallel algorithms involving JuMP models.

# Overview

There are two main types of parallelism in Julia:

- 1. Multi-threading
- 2. Distributed computing

In multi-threading, multiple tasks are run in a single Julia process and share the same memory. In distributed computing, tasks are run in multiple Julia processes with independent memory spaces. This can include processes across multiple physical machines, such as in a high-performance computing cluster.

Choosing and understanding the type of parallelism you are using is important because the code you write for each type is different, and there are different limitations and benefits to each approach. However, the best choice is highly problem dependent, so you may want to experiment with both approaches to determine what works for your situation.

### Multi-threading

To use multi-threading with Julia, you must either start Julia with the command line flag --threads=N, or you must set the JULIA\_NUM\_THREADS environment variable before launching Julia. For this documentation, we set the environment variable to:

```
julia> ENV["JULIA_NUM_THREADS"]
"4"
```

You can check how many threads are available using:

```
julia> Threads.nthreads()
4
```

The easiest way to use multi-threading in Julia is by placing the Threads .@threads macro in front of a for-loop:

```
julia> @time begin
 ids = Int[]
 my_lock = Threads.ReentrantLock()
 Threads.@threads for i in 1:Threads.nthreads()
 global ids, my_lock
 Threads.lock(my_lock) do
 push!(ids, Threads.threadid())
 end
 sleep(1.0)
 end
end
1.037087 seconds (31.32 k allocations: 1.836 MiB, 2.02% compilation time)
```

This for-loop sleeps for 1 second on each iteration. Thus, if it had executed sequentially, it should have taken the same number of seconds as there are threads available. Instead, it took only 1 second, showing that the iterations were executed simultaneously. We can verify this by checking the Threads.threadid() of the thread that executed each iteration:

julia> ids
4-element Vector{Int64}:
2
4
1
3

### Danger

The Threads.threadid() that a task runs on may change during execution. Therefore, it is not safe to use Threads.threadid() to index into, say, a vector of buffer or stateful objects. As an example, do not do:

```
x = rand(Threads.nthreads())
Threads.@threads for i in 1:Threads.nthreads()
 x[Threads.threadid()] *= 2 # Danger! This use of threadid is not safe
end
```

For more information, read PSA: Thread-local state is no longer recommended.

### Thread safety

When working with threads, you must avoid race conditions, in which two threads attempt to write to the same variable at the same time. In the above example we avoided a race condition by using ReentrantLock. See the Multi-threading section of the Julia documentation for more details.

JuMP models are not thread-safe. Code that uses multi-threading to simultaneously modify or optimize a single JuMP model across threads may error, crash Julia, or silently produce incorrect results.

For example, the following incorrect use of multi-threading crashes Julia:

```
zsh: abort julia -t 4
```

To avoid issues with thread safety, create a new instance of a JuMP model in each iteration of the for-loop. In addition, you must avoid race conditions in the rest of your Julia code, for example, by using a lock when pushing elements to a shared vector.

### Example: parameter search with multi-threading

Here is an example of how to use multi-threading to solve a collection of JuMP models in parallel.

```
julia> using JuMP, HiGHS
```

```
julia> function a_good_way_to_use_threading()
 solutions = Pair{Int,Float64}[]
 my_lock = Threads.ReentrantLock();
 Threads.@threads for i in 1:10
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 set_attribute(model, MOI.NumberOfThreads(), 1)
 @variable(model, x >= i)
 @objective(model, Min, x)
 optimize!(model)
 assert is solved and feasible(model)
 Threads.lock(my_lock) do
 push!(solutions, i => objective_value(model))
 end
 end
 return solutions
 end
```

a\_good\_way\_to\_use\_threading (generic function with 1 method)

```
julia> a_good_way_to_use_threading()
10-element Vector{Pair{Int64, Float64}}:
```

7 => 7.0 9 => 9.0 4 => 4.0 1 => 1.0 5 => 5.0 2 => 2.0 8 => 8.0 10 => 10.0 3 => 3.06 => 6.0

#### Warning

For some solvers, it may be necessary to limit the number of threads used internally by the solver to 1 by setting the MOI.NumberOfThreads attribute.

## Example: building data structures in parallel

For large problems, building the model in JuMP can be a bottleneck, and you may consider trying to write code that builds the model in parallel, for example, by wrapping a for-loop that adds constraints with Threads.@threads. Here's an example:

```
julia> using JuMP
julia> function an_incorrect_way_to_build_with_multithreading()
 model = Model()
 @variable(model, x[1:10])
 Threads.@threads for i in 1:10
 @constraint(model, x[i] <= i)
 end
 return model
 end
```

```
julia> an_incorrect_way_to_build_with_multithreading()
A JuMP Model
| solver: none
| objective_sense: FEASIBILITY_SENSE
| num_variables: 10
| num_constraints: 7
| L AffExpr in MOI.LessThan{Float64}: 7
L Names registered in the model
L :x
```

Unfortunately, this model is wrong. It has only seven constraints instead of the expected ten. This happens because JuMP models are not thread-safe. Code that uses multi-threading to simultaneously modify or optimize a single JuMP model across threads may error, crash Julia, or silently produce incorrect results.

The correct way to build a JuMP model with multi-threading is to build the data structures in parallel, but add them to the JuMP model in a thread-safe way:

```
julia> using JuMP
julia> function a_correct_way_to_build_with_multithreading()
 model = Model()
 @variable(model, x[1:10])
 my_lock = Threads.ReentrantLock()
 Threads.@threads for i in 1:10
 con = @build_constraint(x[i] <= i)
 Threads.lock(my_lock) do
 add_constraint(model, con)
 end
 end
 return model
 end
julia> a_correct_way_to_build_with_multithreading()
A JuMP Model
```

```
> solver: none
> objective_sense: FEASIBILITY_SENSE
> num_variables: 10
> num_constraints: 10
> L AffExpr in MOI.LessThan{Float64}: 10
L Names registered in the model
L :x
```

#### Warning

**Do not use multi-threading to build a JuMP model just because your original code is slow.** In most cases, we find that the reason for the bottleneck is not JuMP, but in how you are constructing the problem data, and that with changes, it is possible to build a model in a way that is not the bottleneck in the solution process. If you need help to make your code run faster, ask for help on the community forum. Make sure to include a reproducible example of your code.

## **Distributed computing**

To use distributed computing with Julia, use the Distributed package:

julia> import Distributed

Like multi-threading, we need to tell Julia how many processes to add. We can do this either by starting Julia with the -p N command line argument, or by using Distributed.addprocs:

```
julia> import Pkg
julia> project = Pkg.project();
julia> workers = Distributed.addprocs(4; exeflags = "--project=$(project.path)")
4-element Vector{Int64}:
2
3
4
5
```

#### Warning

Not loading the parent environment with --project is a common mistake.

The added processes are "worker" processes that we can use to do computation with. They are orchestrated by the process with the id 1. You can check what process the code is currently running on using Distributed.myid()

```
julia> Distributed.myid()
1
```

As a general rule, to get maximum performance you should add as many processes as you have logical cores available.

Unlike the for-loop approach of multi-threading, distributed computing extends the Julia map function to a "parallel-map" function Distributed.pmap. For each element in the list of arguments to map over, Julia will copy the element to an idle worker process and evaluate the function, passing the element as an input argument.

CHAPTER 9. ALGORITHMS

Unfortunately, if you try this code directly, you will get an error message that says 0n worker 2: UndefVarError: hard\_work not defined. The error is thrown because, although process 1 knows what the hard\_work function is, the worker processes do not.

To fix the error, we need to use Distributed.@everywhere, which evaluates the code on every process:

```
julia> Distributed.@everywhere begin
 function hard_work(i::Int)
 sleep(1.0)
 return Distributed.myid()
 end
 end
```

Now if we run pmap, we see that it took only 1 second instead of 4, and that it executed on each of the worker processes:

```
julia> @time ids = Distributed.pmap(hard_work, 1:4)
 1.202006 seconds (216.39 k allocations: 13.301 MiB, 4.07% compilation time)
4-element Vector{Int64}:
 2
 3
 5
 4
```

### Tip

For more information, read the Julia documentation Distributed Computing.

## Example: parameter search with distributed computing

With distributed computing, remember to evaluate all of the code on all of the processes using Distributed.@everywhere, and then write a function which creates a new instance of the model on every evaluation:

```
julia> Distributed.@everywhere begin
 using JuMP
 import HiGHS
 end
julia> Distributed.@everywhere begin
 function solve_model_with_right_hand_side(i)
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, x)
 @objective(model, Min, x)
 set_lower_bound(x, i)
 optimize!(model)
 assert_is_solved_and_feasible(sudoku)
 return objective_value(model)
 end
 end
```

julia> solutions = Distributed.pmap(solve\_model\_with\_right\_hand\_side, 1:10)
```
10-element Vector{Float64}:

1.0

2.0

3.0

4.0

5.0

6.0

7.0

8.0

9.0

10.0
```

# Parallelism within the solver

Many solvers use parallelism internally. For example, commercial solvers like Gurobi.jl and CPLEX.jl both parallelize the search in branch-and-bound.

Solvers supporting internal parallelism will typically support the MOI.NumberOfThreads attribute, which you can set using set\_attribute:

```
using JuMP, Gurobi
model = Model(Gurobi.Optimizer)
set_attribute(model, MOI.NumberOfThreads(), 4)
```

## **GPU** parallelism

JuMP does not support GPU programming, but some solvers support execution on a GPU.

One example is SCS.jl, which supports using a GPU to internally solve a system of linear equations. If you are on x86\_64 Linux machine, do:

```
using JuMP, SCS, SCS_GPU_jll
model = Model(SCS.0ptimizer)
set_attribute(model, "linear_solver", SCS.GpuIndirectSolver)
```

# 9.6 Writing a solver interface

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to demonstrate how to implement a basic solver interface to MathOptInterface. As a motivating example, we implement the Primal Dual Hybrid Gradient (PDHG) method. PDHG is a first-order method that can solve convex optimization problems.

Google has a good introduction to the math behind PDLP, which is a variant of PDHG specialized for linear programs.

# **Required packages**

This tutorial requires the following packages:

CHAPTER 9. ALGORITHMS

using JuMP import LinearAlgebra import MathOptInterface as MOI import Printf import SparseArrays

# **Primal Dual Hybrid Gradient**

The following function is a pedagogical implementation of PDHG that solves the linear program:

$$\min c^{\top} x \\ \text{subject to } Ax = b \\ x \ge 0.$$

Note that this implementation is intentionally kept simple. It is not robust nor efficient, and it does not incorporate the theoretical improvements in the PDLP paper. It does use two workspace vectors so that the body of the iteration loop is non-allocating.

```
function solve_pdhg(
 A::SparseArrays.SparseMatrixCSC{Float64,Int},
 b::Vector{Float64},
 c::Vector{Float64};
 maximum iterations::Int = 100 000,
 tol::Float64 = 1e-4,
 verbose::Bool = true,
 log_frequency::Int = 1_000,
)
 printf(x::Float64) = Printf.@sprintf("% 1.6e", x)
 printf(x::Int) = Printf.@sprintf("%6d", x)
 m, n = size(A)
 \eta = \tau = 1 / LinearAlgebra.norm(A) - 1e-6
 x, x_next, y, k, status = zeros(n), zeros(n), zeros(m), \theta, MOI.OTHER_ERROR
 m_workspace, n_workspace = zeros(m), zeros(n)
 if verbose
 println(
 " iter pobj dobj pfeas dfeas objfeas",
)
 end
 while status == MOI.OTHER_ERROR
 k += 1
 # _____
 # This block computes x next = max.(0.0, x - \eta * (A' * y + c))
 LinearAlgebra.mul!(x next, A', y)
 LinearAlgebra.axpby!(-η, c, -η, x_next)
 x_next .+= x
 x_next = max.(0.0, x_next)
 # This block computes y += \tau * (A * (2 * x_{next} - x) - b)
 copy!(n workspace, x next)
 LinearAlgebra.axpby!(-1.0, x, 2.0, n_workspace)
 LinearAlgebra.mul!(y, A, n_workspace, \tau, 1.0)
```

```
LinearAlgebra.axpy!(-τ, b, y)
 # _____
 copy!(x, x_next)
 # _____
 # This block computes p feas = LinearAlgebra.norm(A * x - b)
 LinearAlgebra.mul!(m_workspace, A, x)
 m_workspace .-= b
 pfeas = LinearAlgebra.norm(m_workspace)
 # _____
 # This block computes dfeas = LinearAlgebra.norm(min.(0.0, A' * y + c))
 LinearAlgebra.mul!(n_workspace, A', y)
 n_workspace .+= c
 n_workspace .= min.(0.0, n_workspace)
 dfeas = LinearAlgebra.norm(n_workspace)
 objfeas = abs(LinearAlgebra.dot(c, x) + LinearAlgebra.dot(b, y))
 if pfeas <= tol && dfeas <= tol && objfeas <= tol
 status = MOI.OPTIMAL
 elseif k == maximum_iterations
 status = MOI.ITERATION LIMIT
 end
 if verbose \&\& (mod(k, log_frequency) == 0 || status != MOI.OTHER_ERROR)
 logs = printf.((k, c' * x, -b' * y, pfeas, dfeas, objfeas))
 println(join(logs, " "))
 end
end
return status, k, x, y
```

solve\_pdhg (generic function with 1 method)

#### Here's an example:

end

```
A = [0.0 -1.0 -1.0 0.0 0.0; 6.0 8.0 0.0 -1.0 0.0; 7.0 12.0 0.0 0.0 -1.0]
b = [-3.0, 100.0, 120.0]
c = [12.0, 20.0, 0.0, 0.0, 0.0]
status, k, x, y = solve_pdhg(SparseArrays.sparse(A), b, c);
```

```
 iter
 pobj
 dobj
 pfeas
 dfeas
 objfeas

 1000
 2.050187e+02
 2.044002e+02
 2.006420e+01
 2.674295e+02
 6.185366e+01

 2000
 2.049895e+02
 2.051241e+02
 1.705136e+02
 2.746207e+02
 1.346232e+01

 3000
 2.050050e+02
 2.050805e+02
 8.907061e+03
 8.405470e+03
 7.550812e+02

 4000
 2.050024e+02
 2.049755e+02
 4.046623e+03
 9.374982e+04
 2.689079e+02

 5000
 2.049995e+02
 2.049831e+02
 8.635908e+04
 6.483234e+04
 1.633722e+02

 6000
 2.049995e+02
 2.050016e+02
 7.833794e+04
 1.676266e+04
 2.095135e+03

 7000
 2.050001e+02
 2.05003e+02
 2.811341e+05
 3.065459e+04
 2.964863e+03

 8000
 2.050001e+02
 2.050002e+02
 1.316682e+04
 1.674879e+05
 8.453982e+05

 8365
 2.049999e+02
 2.05000e+02
 9.473351e+05
 2.381905e+06
 7.955573e+05
```

The termination status is:

status

OPTIMAL::TerminationStatusCode = 1

The solve took the following number of iterations:

k

8365

The primal solution is:

х

```
5-element Vector{Float64}:
15.000070735249105
1.2499547393201815
1.750098059200116
0.0
0.0
```

The dual multipliers are:

у

```
3-element Vector{Float64}:
1.931930909998547e-6
-0.2500022801722057
-1.4999982446009117
```

# The MOI interface

Converting a linear program from the modeler's form into the A, b, and c matrices of the standard form required by our implementation of PDHG is tedious and error-prone. This section walks through how to implement a basic interface to MathOptInterface, so that we can use our algorithm from JuMP.

For a more comprehensive guide, see Implementing a solver interface.

#### The Optimizer type

Create an optimizer by subtyping MOI.AbstractOptimizer. By convention, the name of this type is Optimizer, and most optimizers are available as PackageName.Optimizer.

The fields inside the optimizer are arbitrary. Store whatever is useful.

```
.....
 Optimizer()
Create a new optimizer for PDHG.
.....
mutable struct Optimizer <: MOI.AbstractOptimizer</pre>
 # A mapping from variable to column
 x_to_col::Dict{MOI.VariableIndex,Int}
 # A mapping from constraint to rows
 ci_to_rows::Dict{
 MOI.ConstraintIndex{MOI.VectorAffineFunction{Float64},MOI.Zeros},
 Vector{Int},
 }
 # Information from solve_pdhg
 status::MOI.TerminationStatusCode
 iterations::Int
 x::Vector{Float64}
 y::Vector{Float64}
 # Other useful quantities
 solve_time::Float64
 obj_value::Float64
 function Optimizer()
 F = MOI.VectorAffineFunction{Float64}
 return new(
 Dict{MOI.VariableIndex,Int}(),
 Dict{MOI.ConstraintIndex{F,MOI.Zeros},Vector{Int}}(),
 MOI.OPTIMIZE_NOT_CALLED,
 Θ,
 Float64[],
 Float64[],
 0.0,
 0.0,
)
 end
end
```

```
Main.Optimizer
```

Now that we have an Optimizer, we need to implement two methods: MOI.is\_empty and MOI.empty!. These are called whenever MOI needs to ensure that the optimizer is in a clean state.

```
function MOI.is_empty(model::Optimizer)
 # You might want to check every field, not just a few
 return isempty(model.x_to_col) && model.status == MOI.OPTIMIZE_NOT_CALLED
end
```

```
function MOI.empty!(model::Optimizer)
 empty!(model.x_to_col)
 empty!(model.ci_to_rows)
 model.status = MOI.OPTIMIZE_NOT_CALLED
 model.iterations = 0
 model.solve_time = 0.0
 model.obj_value = 0.0
 empty!(model.x)
 empty!(model.y)
 return
end
```

Next, we need to define what constraints the optimizer supports. Since our standard form was Ax = b, we support only  $Ax + b \in \{0\}$ , which is a MOI.VectorAffineFunction in MOI.Zeros constraint. Note that you might have expected  $Ax - b \in \{0\}$ . We'll address the difference in the sign of b in a few places later on.

```
function MOI.supports_constraint(
 ::Optimizer,
 ::Type{MOI.VectorAffineFunction{Float64}},
 ::Type{MOI.Zeros},
)
 return true
end
```

By default, MOI assumes that it can add free variables. This isn't true for our standard form, because we support only  $x \ge 0$ . Let's tell MOI that:

```
MOI.supports_add_constrained_variables(::Optimizer, ::Type{MOI.Reals}) = false
function MOI.supports_add_constrained_variables(
 ::Optimizer,
 ::Type{MOI.Nonnegatives},
)
 return true
end
```

The objective function that we support is MOI.ScalarAffineFunction:

```
function MOI.supports(
 ::Optimizer,
 ::MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Float64}},
)
 return true
end
```

Finally, we'll implement MOI.SolverName so that MOI knows how to print the name of our optimizer:

MOI.get(::Optimizer, ::MOI.SolverName) = "PDHG"

# GenericModel

The simplest way to solve a problem with your optimizer is to implement the method MOI.optimize!(dest::Optimizer, src::MOI.ModelLike), where src is an input model and dest is your empty optimizer.

To implement this method you would need to query the variables and constraints in src and the convert these into the matrix data expected by solve\_pdhg. Since matrix input is a common requirement of solvers, MOI includes utilities to simplify the process.

The downside of the utilities is that they involve a highly parameterized type with a large number of possible configurations. The upside of the utilities is that, once setup, they requires few lines of code to extract the problem matrices.

First, we need to define the set of sets that our standard form supports. For PDHG, we support only Ax + b in  $\{0\}$ :

```
MOI.Utilities.@product_of_sets(SetOfZeros, MOI.Zeros)
```

Then, we define a MOI.Utilities.GenericModel. This is the highly parameterized type that can be customized.

```
const CacheModel = MOI.Utilities.GenericModel{
 # The coefficient type is Float64
 Float64,
 # We use the default objective container
 MOI.Utilities.ObjectiveContainer{Float64},
 # We use the default variable container
 MOI.Utilities.VariablesContainer{Float64},
 # We use a Matrix of Constraints to represent `A * x + b in K`
 MOI.Utilities.MatrixOfConstraints{
 # The number type is Float64
 Float64,
 # The matrix type `A` is a sparse matrix
 MOI.Utilities.MutableSparseMatrixCSC{
 # ... with Float64 coefficients
 Float64,
 # ... Int64 row and column indices
 Int.
 # ... and it uses one-based indexing
 MOI.Utilities.OneBasedIndexing,
 },
 # The vector type `b` is a Julia `Vector`
 Vector{Float64},
 # The set type `K` is the SetOfZeros type we defined above
 SetOfZeros{Float64},
 },
```

```
}
```

MathOptInterface.Utilities.GenericModel{Float64,

→ MathOptInterface.Utilities.ObjectiveContainer{Float64},

 $\hookrightarrow$  MathOptInterface.Utilities.VariablesContainer{Float64},

→ MathOptInterface.Utilities.MatrixOfConstraints{Float64,

 $\hookrightarrow \quad \texttt{MathOptInterface.Utilities.MutableSparseMatrixCSC{Float64, Int64,} \\$ 

→ MathOptInterface.Utilities.OneBasedIndexing}, Vector{Float64}, Main.SetOfZeros{Float64}}}

As one example of possible alternate configuration, if you were interfacing with a solver written in C that expected zero-based indices, you might use instead:

```
MOI.Utilities.MutableSparseMatrixCSC{
 Cdouble,
 Cint,
 MOI.Utilities.ZeroBasedIndexing,
}
```

MathOptInterface.Utilities.MutableSparseMatrixCSC{Float64, Int32, → MathOptInterface.Utilities.ZeroBasedIndexing}

# Тір

The best place to look at how to configure GenericModel is to find an existing solver with the same input standard form that you require.

We need to make one modification to CacheModel to tell MOI that  $x \in \mathbb{R}_+$  is equivalent to adding variables in MOI.GreaterThan:

```
function MOI.add_constrained_variables(model::CacheModel, set::MOI.Nonnegatives)
 x = MOI.add_variables(model, MOI.dimension(set))
 MOI.add_constraint.(model, x, MOI.GreaterThan(0.0))
 ci = MOI.ConstraintIndex{MOI.VectorOfVariables,MOI.Nonnegatives}(x[1].value)
 return x, ci
end
```

#### The optimize method

Now we define the most important method for our optimizer.

```
function MOI.optimize!(dest::Optimizer, src::MOI.ModelLike)
 # In addition to the values returned by `solve pdhg`, it may be useful to
 # record other attributes, such as the solve time.
 start time = time()
 # Construct a cache to store our problem data:
 cache = CacheModel()
 # MOI includes a utility to copy an arbitrary `src` model into `cache`. The
 # return, `index map`, is a mapping from indices in `src` to indices in
 # `dest`.
 index_map = MOI.copy_to(cache, src)
 # Now we can access the `A` matrix:
 A = convert(
 SparseArrays.SparseMatrixCSC{Float64,Int},
 cache.constraints.coefficients,
)
 # and the b vector (note that MOI models Ax = b as Ax + b in \{0\}, so b
 # differs by -):
 b = -cache.constraints.constants
```

```
The `c` vector is more involved, because we need to account for the
 # objective sense:
 sense = ifelse(cache.objective.sense == MOI.MAX_SENSE, -1, 1)
 F = MOI.ScalarAffineFunction{Float64}
 obj = MOI.get(src, MOI.ObjectiveFunction{F}())
 c = zeros(size(A, 2))
 for term in obj.terms
 c[term.variable.value] += sense * term.coefficient
 end
 # Now we can solve the problem with PDHG and record the solution:
 dest.status, dest.iterations, dest.x, dest.y = solve_pdhg(A, b, c)
 # To help assign the values of the x and y vectors to the appropriate
 # variables and constrats, we need a map of the constraint indices to their
 # row in the `dest` matrix and a map of the variable indices to their
 # column in the `dest` matrix:
 F, S = MOI.VectorAffineFunction{Float64}, MOI.Zeros
 for src_ci in MOI.get(src, MOI.ListOfConstraintIndices{F,S}())
 dest.ci_to_rows[index_map[src_ci]] =
 MOI.Utilities.rows(cache.constraints.sets, index_map[src_ci])
 end
 for (i, src_x) in enumerate(MOI.get(src, MOI.ListOfVariableIndices()))
 dest.x_to_col[index_map[src_x]] = i
 end
 # We can now record two derived quantities: the primal objective value and
 # the solve time.
 dest.obj_value = obj.constant + sense * c' * dest.x
 dest.solve_time = time() - start_time
 # We need to return the index map, and `false`, to indicate to MOI that we
 # do not support incremental modification of the model.
 return index map, false
end
```

# Solutions

Now that we know how to solve a model, let's implement the required solution attributes.

First, we need to tell MOI how many solutions we found via MOI.ResultCount:

```
function MOI.get(model::Optimizer, ::MOI.ResultCount)
 return model.status == MOI.OPTIMAL ? 1 : 0
end
```

and implement MOI.RawStatusString to provide a user-readable string that describes what happened:

```
function MOI.get(model::Optimizer, ::MOI.RawStatusString)
 if model.status == MOI.OPTIMAL
 return "found a primal-dual optimal solution (subject to tolerances)"
 end
 return "failed to solve"
end
```

Then, we need to implement the three types of problem status: MOI.TerminationStatus, MOI.PrimalStatus and MOI.DualStatus:

```
MOI.get(model::Optimizer, ::MOI.TerminationStatus) = model.status
function MOI.get(model::Optimizer, attr::Union{MOI.PrimalStatus,MOI.DualStatus})
 if attr.result_index == 1 && model.status == MOI.OPTIMAL
 return MOI.FEASIBLE_POINT
 end
 return MOI.NO_SOLUTION
end
```

Now we can implement MOI.ObjectiveValue, MOI.VariablePrimal, and MOI.ConstraintDual:

```
function MOI.get(model::Optimizer, attr::MOI.ObjectiveValue)
 MOI.check result index bounds(model, attr)
 return model.obj_value
end
function MOI.get(
 model::Optimizer,
 attr::MOI.VariablePrimal,
 x::MOI.VariableIndex,
)
 MOI.check_result_index_bounds(model, attr)
 return model.x[model.x_to_col[x]]
end
function MOI.get(
 model::Optimizer,
 attr::MOI.ConstraintDual,
 ci::MOI.ConstraintIndex{MOI.VectorAffineFunction{Float64},MOI.Zeros},
)
 MOI.check_result_index_bounds(model, attr)
 # MOI models Ax = b as Ax + b in \{0\}, so the dual differs by -
 return -model.y[model.ci to rows[ci]]
end
```

Some other useful result quantities are MOI.SolveTimeSec and MOI.BarrierIterations:

```
MOI.get(model::Optimizer, ::MOI.SolveTimeSec) = model.solve_time
MOI.get(model::Optimizer, ::MOI.BarrierIterations) = model.iterations
```

#### A JuMP example

Now we can solve an arbitrary linear program with JuMP. Here's the same standard form as before:

```
model = Model(Optimizer)
@variable(model, x[1:5] >= 0)
@objective(model, Min, c' * x)
@constraint(model, c3, A * x == b)
optimize!(model)
```

iterpobjdobjpfeasdfeasobjfeas10002.050187e+022.044002e+022.006420e-012.674295e-026.185366e-0120002.049895e+022.051241e+021.705136e-022.746207e-021.346232e-0130002.050050e+022.050805e+028.907061e-038.405470e-037.550812e-0240002.050024e+022.049755e+024.046623e-039.374982e-042.689079e-0250002.049995e+022.049831e+028.635908e-046.483234e-041.633722e-0260002.049995e+022.050016e+027.833794e-041.676266e-042.095135e-0370002.05000e+022.050030e+022.811341e-053.065459e-042.964863e-0380002.05001e+022.050002e+021.316682e-041.674879e-058.453982e-0583652.049999e+022.05000e+029.473351e-052.381905e-067.955573e-05

solution\_summary(model; verbose = true)

```
solution_summary(; result = 1, verbose = true)
- solver_name
 : PDHG
- Termination
| | termination_status : OPTIMAL
| | result_count : 1
L raw_status : found a primal-dual optimal solution (subject to tolerances)
Solution (result = 1)
| + dual_status : FEASIBLE_POINT
| + objective_value : 2.05000e+02
| | dual_objective_value : 2.05000e+02
| | value
| | | ×[1] : 1.50001e+01
| | | ×[2] : 1.24995e+00
| | + x[3] : 1.75010e+00
| | + x[4] : 0.00000e+00
| | ^L x[5] : 0.00000e+00
| ^L dual
L c3 : [-1.93193e-06,2.50002e-01,1.50000e+00]
^{\rm L} Work counters
 - solve_time (sec) : 2.14339e-01
 L barrier_iterations : 8365
```

But we could also have written:

```
model = Model(Optimizer)
@variable(model, x >= 0)
@variable(model, 0 <= y <= 3)
@objective(model, Min, 12x + 20y)
@constraint(model, c1, 6x + 8y >= 100)
@constraint(model, c2, 7x + 12y >= 120)
optimize!(model)
```

iter pobj dobj pfeas dfeas objfeas 1000 2.050187e+02 2.044002e+02 2.006420e-01 2.674295e-02 6.185366e-01

```
20002.049895e+022.051241e+021.705136e-022.746207e-021.346232e-0130002.050050e+022.050805e+028.907061e-038.405470e-037.550812e-0240002.050024e+022.049755e+024.046623e-039.374982e-042.689079e-0250002.049995e+022.049831e+028.635908e-046.483234e-041.633722e-0260002.049995e+022.05001e+027.833794e-041.676266e-042.095135e-0370002.05000e+022.050030e+022.811341e-053.065459e-042.964863e-0380002.050001e+022.050002e+021.316682e-041.674879e-058.453982e-0583652.049999e+022.05000e+029.473351e-052.381905e-067.95573e-05
```

solution\_summary(model; verbose = true)

```
solution_summary(; result = 1, verbose = true)
- solver name
 : PDHG
- Termination
| | result_count : 1
| ^L raw_status : found a primal-dual optimal solution (subject to tolerances)
Solution (result = 1)

 | + primal_status
 : FEASIBLE_POINT

 | + dual_status
 : FEASIBLE_POINT

 | + objective_value
 : 2.05000e+02

| + dual_objective_value : 2.05000e+02
| | value
| | | × : 1.50001e+01
| | ^L y : 1.24995e+00
| ^L dual
| | c1 : 2.50002e-01
 ^L c2 : 1.50000e+00
L Work counters
 - solve_time (sec) : 1.69992e-03
 L barrier_iterations : 8365
```

Other variations are also possible:

model = Model(Optimizer)
@variable(model, x[1:5] >= 0)
@objective(model, Max, -c' \* x)
@constraint(model, c4, A \* x .== b)
optimize!(model)

iter	pobj	dobj	pfeas	dfeas	objfeas
1000	2.050187e+02	2.044002e+02	2.006420e-01	2.674295e-02	6.185366e-01
2000	2.049895e+02	2.051241e+02	1.705136e-02	2.746207e-02	1.346232e-01
3000	2.050050e+02	2.050805e+02	8.907061e-03	8.405470e-03	7.550812e-02
4000	2.050024e+02	2.049755e+02	4.046623e-03	9.374982e-04	2.689079e-02
5000	2.049995e+02	2.049831e+02	8.635908e-04	6.483234e-04	1.633722e-02
6000	2.049995e+02	2.050016e+02	7.833794e-04	1.676266e-04	2.095135e-03
7000	2.050000e+02	2.050030e+02	2.811341e-05	3.065459e-04	2.964863e-03

```
 8000
 2.050001e+02
 2.050002e+02
 1.316682e-04
 1.674879e-05
 8.453982e-05

 8365
 2.049999e+02
 2.050000e+02
 9.473351e-05
 2.381905e-06
 7.955573e-05
```

solution\_summary(model; verbose = true)

```
solution_summary(; result = 1, verbose = true)
 : PDHG
- solver_name
- Termination
| + termination_status : OPTIMAL
| F result_count : 1
| L raw_status : found a primal-dual optimal solution (subject to tolerances)
Solution (result = 1)
| + primal_status : FEASIBLE_POINT
| + dual_status : FEASIBLE_POINT
| + objective_value : -2.05000e+02
| | dual_objective_value : -2.05000e+02
| | value
| | + x[1] : 1.50001e+01
| | + x[2] : 1.24995e+00
| | + x[3] : 1.75010e+00
| | | ×[4] : 0.00000e+00
| | ^L x[5] : 0.00000e+00
| ^L dual
 ^{\mbox{ L}} c4 : multiple constraints with the same name
L Work counters
 solve_time (sec) : 1.71018e-03
 L barrier_iterations : 8365
```

Behind the scenes, JuMP and MathOptInterface reformulate the problem from the modeller's form into the standard form defined by our Optimizer.

# Chapter 10

# Applications

# 10.1 Power Systems

This tutorial was generated using Literate.jl. Download the source as a .jl file.

# This tutorial was originally contributed by Yury Dvorkin and Miles Lubin.

This tutorial demonstrates how to formulate basic power systems engineering models in JuMP.

We will consider basic "economic dispatch" and "unit commitment" models without taking into account transmission constraints.

For this tutorial, we use the following packages:

using JuMP import DataFrames import HiGHS import Plots import StatsPlots

#### **Economic dispatch**

Economic dispatch (ED) is an optimization problem that minimizes the cost of supplying energy demand subject to operational constraints on power system assets. In its simplest modification, ED is an LP problem solved for an aggregated load and wind forecast and for a single infinitesimal moment.

Mathematically, the ED problem can be written as follows:

$$\min\sum_{i\in I}c_i^g\cdot g_i+c^w\cdot w,$$

where  $c_i$  and  $g_i$  are the incremental cost (\$/MWh) and power output (MW) of the  $i^{th}$  generator, respectively, and  $c^w$  and w are the incremental cost (\$/MWh) and wind power injection (MW), respectively.

Subject to the constraints:

- Minimum  $(g^{\min})$  and maximum  $(g^{\max})$  limits on power outputs of generators:  $g_i^{\min} \le g_i \le g_i^{\max}$ .
- Constraint on the wind power injection:  $0 \le w \le w^f$ , where w and  $w^f$  are the wind power injection and wind power forecast, respectively.

• Power balance constraint:  $\sum_{i \in I} g_i + w = d^f$ , where  $d^f$  is the demand forecast.

Further reading on ED models can be found in A. J. Wood, B. F. Wollenberg, and G. B. Sheblé, "Power Generation, Operation and Control," Wiley, 2013.

Define some input data about the test system.

We define some thermal generators:

```
function ThermalGenerator(
 min::Float64,
 max::Float64,
 fixed_cost::Float64,
 variable_cost::Float64,
)
 return (
 min = min,
 max = max,
 fixed_cost = fixed_cost,
 variable_cost = variable_cost,
)
end
generators = [
 ThermalGenerator(0.0, 1000.0, 1000.0, 50.0),
 ThermalGenerator(300.0, 1000.0, 0.0, 100.0),
]
```

2-element Vector{@NamedTuple{min::Float64, max::Float64, fixed\_cost::Float64, → variable\_cost::Float64}}: (min = 0.0, max = 1000.0, fixed\_cost = 1000.0, variable\_cost = 50.0) (min = 300.0, max = 1000.0, fixed\_cost = 0.0, variable\_cost = 100.0)

A wind generator

```
WindGenerator(variable_cost::Float64) = (variable_cost = variable_cost,)
```

wind\_generator = WindGenerator(50.0)

scenario = Scenario(1500.0, 200.0)

(variable\_cost = 50.0,)

And a scenario

```
function Scenario(demand::Float64, wind::Float64)
 return (demand = demand, wind = wind)
end
```

(demand = 1500.0, wind = 200.0)

Create a function solve\_economic\_dispatch, which solves the economic dispatch problem for a given set of input parameters.

```
function solve_economic_dispatch(generators::Vector, wind, scenario)
 # Define the economic dispatch (ED) model
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 # Define decision variables
 # power output of generators
 N = length(generators)
 @variable(model, generators[i].min <= g[i = 1:N] <= generators[i].max)</pre>
 # wind power injection
 @variable(model, 0 <= w <= scenario.wind)</pre>
 # Define the objective function
 @objective(
 model,
 Min,
 sum(generators[i].variable_cost * g[i] for i in 1:N) +
 wind.variable_cost * w,
)
 # Define the power balance constraint
 @constraint(model, sum(g[i] for i in 1:N) + w == scenario.demand)
 # Solve statement
 optimize!(model)
 assert_is_solved_and_feasible(model)
 # return the optimal value of the objective function and its minimizers
 return (
 g = value.(g),
 w = value(w),
 wind_spill = scenario.wind - value(w),
 total_cost = objective_value(model),
)
end
```

solve\_economic\_dispatch (generic function with 1 method)

Solve the economic dispatch problem

```
solution = solve_economic_dispatch(generators, wind_generator, scenario);
println("Dispatch of Generators: ", solution.g, " MW")
println("Dispatch of Wind: ", solution.w, " MW")
println("Wind spillage: ", solution.wind_spill, " MW")
println("Total cost: \$", solution.total_cost)
```

```
Dispatch of Generators: [1000.0, 300.0] MW
Dispatch of Wind: 200.0 MW
Wind spillage: 0.0 MW
Total cost: $90000.0
```

#### Economic dispatch with adjustable incremental costs

In the following exercise we adjust the incremental cost of generator G1 and observe its impact on the total cost.

```
function scale_generator_cost(g, scale)
 return ThermalGenerator(g.min, g.max, g.fixed_cost, scale * g.variable_cost)
end
start = time()
c_g_scale_df = DataFrames.DataFrame(;
 # Scale factor
 scale = Float64[],
 # Dispatch of Generator 1 [MW]
 dispatch_G1 = Float64[],
 # Dispatch of Generator 2 [MW]
 dispatch_G2 = Float64[],
 # Dispatch of Wind [MW]
 dispatch_wind = Float64[],
 # Spillage of Wind [MW]
 spillage_wind = Float64[],
 # Total cost [$]
 total_cost = Float64[],
)
for c_g1_scale in 0.5:0.1:3.0
 # Update the incremental cost of the first generator at every iteration.
 new_generators = scale_generator_cost.(generators, [c_g1_scale, 1.0])
 # Solve the economic-dispatch problem with the updated incremental cost
 sol = solve_economic_dispatch(new_generators, wind_generator, scenario)
 push!(
 c_g_scale_df,
 (c_g1_scale, sol.g[1], sol.g[2], sol.w, sol.wind_spill, sol.total_cost),
)
end
print(string("elapsed time: ", time() - start, " seconds"))
```

elapsed time: 0.13479399681091309 seconds

c\_g\_scale\_df

	scale	dispatch_G1	spatch_G1 dispatch_G2 dispatch_wind spillage_wind		spillage_wind	total_cost
	Float64	Float64	Float64	Float64	Float64	Float64
1	0.5	1000.0	300.0	200.0	0.0	65000.0
2	0.6	1000.0	300.0	200.0	0.0	70000.0
3	0.7	1000.0	300.0	200.0	0.0	75000.0
4	0.8	1000.0	300.0	200.0	0.0	80000.0
5	0.9	1000.0	300.0	200.0	0.0	85000.0
6	1.0	1000.0	300.0	200.0	0.0	90000.0
7	1.1	1000.0	300.0	200.0	0.0	95000.0
8	1.2	1000.0	300.0	200.0	0.0	100000.0
9	1.3	1000.0	300.0	200.0	0.0	105000.0
10	1.4	1000.0	300.0	200.0	0.0	110000.0
11	1.5	1000.0	300.0	200.0	0.0	115000.0
12	1.6	1000.0	300.0	200.0	0.0	120000.0
13	1.7	1000.0	300.0	200.0	0.0	125000.0
14	1.8	1000.0	300.0	200.0	0.0	130000.0
15	1.9	1000.0	300.0	200.0	0.0	135000.0
16	2.0	300.0	1000.0	200.0	0.0	140000.0
17	2.1	300.0	1000.0	200.0	0.0	141500.0
18	2.2	300.0	1000.0	200.0	0.0	143000.0
19	2.3	300.0	1000.0	200.0	0.0	144500.0
20	2.4	300.0	1000.0	200.0	0.0	146000.0
21	2.5	300.0	1000.0	200.0	0.0	147500.0
22	2.6	300.0	1000.0	200.0	0.0	149000.0
23	2.7	300.0	1000.0	200.0	0.0	150500.0
24	2.8	300.0	1000.0	200.0	0.0	152000.0
25	2.9	300.0	1000.0	200.0	0.0	153500.0
26	3.0	300.0	1000.0	200.0	0.0	155000.0

#### Modifying the JuMP model in-place

Note that in the previous exercise we entirely rebuilt the optimization model at every iteration of the internal loop, which incurs an additional computational burden. This burden can be alleviated if instead of re-building the entire model, we modify the constraints or objective function, as it shown in the example below.

Compare the computing time in case of the above and below models.

```
function solve_economic_dispatch_inplace(
 generators::Vector,
 wind,
 scenario,
 scale::AbstractVector{Float64},
)
 obj_out = Float64[]
 w_out = Float64[]
 g1_out = Float64[]
 g2_out = Float64[]
 # This function only works for two generators
 @assert length(generators) == 2
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 N = length(generators)
 @variable(model, generators[i].min <= g[i = 1:N] <= generators[i].max)</pre>
```

```
@variable(model, 0 <= w <= scenario.wind)</pre>
 @objective(
 model,
 Min,
 sum(generators[i].variable_cost * g[i] for i in 1:N) +
 wind.variable_cost * w,
)
 @constraint(model, sum(g[i] for i in 1:N) + w == scenario.demand)
 for c_g1_scale in scale
 @objective(
 model,
 Min,
 c_g1_scale * generators[1].variable_cost * g[1] +
 generators[2].variable_cost * g[2] +
 wind.variable_cost * w,
)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 push!(obj_out, objective_value(model))
 push!(w_out, value(w))
 push!(g1_out, value(g[1]))
 push!(g2_out, value(g[2]))
 end
 df = DataFrames.DataFrame(;
 scale = scale,
 dispatch_G1 = g1_out,
 dispatch_G2 = g2_out,
 dispatch_wind = w_out,
 spillage_wind = scenario.wind .- w_out,
 total_cost = obj_out,
)
 return df
end
start = time()
inplace_df = solve_economic_dispatch_inplace(
 generators,
 wind generator,
 scenario,
 0.5:0.1:3.0,
print(string("elapsed time: ", time() - start, " seconds"))
```

elapsed time: 0.1675879955291748 seconds

For small models, adjusting specific constraints or the objective function is sometimes faster and sometimes slower than re-building the entire model. However, as the problem size increases, updating the model in-place is usually faster.

inplace\_df

)

	scale	dispatch_G1	dispatch_G2	dispatch_wind	spillage_wind	total_cost
	Float64	Float64	Float64	Float64	Float64	Float64
1	0.5	1000.0	300.0	200.0	0.0	65000.0
2	0.6	1000.0	300.0	200.0	0.0	70000.0
3	0.7	1000.0	300.0	200.0	0.0	75000.0
4	0.8	1000.0	300.0	200.0	0.0	80000.0
5	0.9	1000.0	300.0	200.0	0.0	85000.0
6	1.0	1000.0	300.0	200.0	0.0	90000.0
7	1.1	1000.0	300.0	200.0	0.0	95000.0
8	1.2	1000.0	300.0	200.0	0.0	100000.0
9	1.3	1000.0	300.0	200.0	0.0	105000.0
10	1.4	1000.0	300.0	200.0	0.0	110000.0
11	1.5	1000.0	300.0	200.0	0.0	115000.0
12	1.6	1000.0	300.0	200.0	0.0	120000.0
13	1.7	1000.0	300.0	200.0	0.0	125000.0
14	1.8	1000.0	300.0	200.0	0.0	130000.0
15	1.9	1000.0	300.0	200.0	0.0	135000.0
16	2.0	1000.0	300.0	200.0	0.0	140000.0
17	2.1	300.0	1000.0	200.0	0.0	141500.0
18	2.2	300.0	1000.0	200.0	0.0	143000.0
19	2.3	300.0	1000.0	200.0	0.0	144500.0
20	2.4	300.0	1000.0	200.0	0.0	146000.0
21	2.5	300.0	1000.0	200.0	0.0	147500.0
22	2.6	300.0	1000.0	200.0	0.0	149000.0
23	2.7	300.0	1000.0	200.0	0.0	150500.0
24	2.8	300.0	1000.0	200.0	0.0	152000.0
25	2.9	300.0	1000.0	200.0	0.0	153500.0
26	3.0	300.0	1000.0	200.0	0.0	155000.0

#### Inefficient usage of wind generators

The economic dispatch problem does not perform commitment decisions and, thus, assumes that all generators must be dispatched at least at their minimum power output limit. This approach is not cost efficient and may lead to absurd decisions. For example, if  $d = \sum_{i \in I} g_i^{\min}$ , the wind power injection must be zero, that is, all available wind generation is spilled, to meet the minimum power output constraints on generators.

In the following example, we adjust the total demand and observed how it affects wind spillage.

```
demand_scale_df = DataFrames.DataFrame(;
 demand = Float64[],
 dispatch_G1 = Float64[],
 dispatch_wind = Float64[],
 spillage_wind = Float64[],
 total_cost = Float64[],
)
function scale_demand(scenario, scale)
 return Scenario(scale * scenario.demand, scenario.wind)
end
for demand_scale in 0.2:0.1:1.4
 new_scenario = scale_demand(scenario, demand_scale)
```

```
sol = solve_economic_dispatch(generators, wind_generator, new_scenario)
push!(
 demand_scale_df,
 (
 new_scenario.demand,
 sol.g[1],
 sol.g[2],
 sol.w,
 sol.wind_spill,
 sol.total_cost,
),
)
end
```

```
demand_scale_df
```

	demand	nd dispatch_G1 dispatch_G2 d		dispatch_wind	spillage_wind	total_cost
	Float64	Float64	Float64	Float64	Float64	Float64
1	300.0	0.0	300.0	0.0	200.0	30000.0
2	450.0	150.0	300.0	0.0	200.0	37500.0
3	600.0	300.0	300.0	0.0	200.0	45000.0
4	750.0	450.0	300.0	0.0	200.0	52500.0
5	900.0	600.0	300.0	0.0	200.0	60000.0
6	1050.0	750.0	300.0	0.0	200.0	67500.0
7	1200.0	900.0	300.0	0.0	200.0	75000.0
8	1350.0	850.0	300.0	200.0	0.0	82500.0
9	1500.0	1000.0	300.0	200.0	0.0	90000.0
10	1650.0	1000.0	450.0	200.0	0.0	105000.0
11	1800.0	1000.0	600.0	200.0	0.0	120000.0
12	1950.0	1000.0	750.0	200.0	0.0	135000.0
13	2100.0	1000.0	900.0	200.0	0.0	150000.0

```
dispatch_plot = StatsPlots.@df(
 demand_scale_df,
 Plots.plot(
 :demand,
 [:dispatch_G1, :dispatch_G2],
 labels = ["G1" "G2"],
 title = "Thermal Dispatch",
 legend = :bottomright,
 linewidth = 3,
 xlabel = "Demand",
 ylabel = "Dispatch [MW]",
),
)
wind_plot = StatsPlots.@df(
 demand_scale_df,
 Plots.plot(
 :demand,
 [:dispatch_wind, :spillage_wind],
 labels = ["Dispatch" "Spillage"],
```

)

title = "Wind", legend = :bottomright, linewidth = 3, xlabel = "Demand [MW]", ylabel = "Energy [MW]", ),

```
Plots.plot(dispatch_plot, wind_plot)
```



This particular drawback can be overcome by introducing binary decisions on the "on/off" status of generators. This model is called unit commitment and considered later in these notes.

For further reading on the interplay between wind generation and the minimum power output constraints of generators, we refer interested readers to R. Baldick, "Wind and energy markets: a case study of Texas," IEEE Systems Journal, vol. 6, pp. 27-34, 2012.

# Unit commitment

The Unit Commitment (UC) model can be obtained from ED model by introducing binary variable associated with each generator. This binary variable can attain two values: if it is "1," the generator is synchronized and, thus, can be dispatched, otherwise, that is, if the binary variable is "0," that generator is not synchronized and its power output is set to 0.

To obtain the mathematical formulation of the UC model, we will modify the constraints of the ED model as follows:

$$g_i^{\min} \cdot u_{t,i} \le g_i \le g_i^{\max} \cdot u_{t,i},$$

where  $u_i \in \{0,1\}$ . In this constraint, if  $u_i = 0$ , then  $g_i = 0$ . On the other hand, if  $u_i = 1$ , then  $g_i^{min} \le g_i \le g_i^{max}$ .

For further reading on the UC problem we refer interested readers to G. Morales-Espana, J. M. Latorre, and A. Ramos, "Tight and Compact MILP Formulation for the Thermal Unit Commitment Problem," IEEE Transactions on Power Systems, vol. 28, pp. 4897-4908, 2013.

In the following example we convert the ED model explained above to the UC model.

```
function solve_unit_commitment(generators::Vector, wind, scenario)
 model = Model(HiGHS.Optimizer)
 set silent(model)
 N = length(generators)
 @variable(model, 0 <= g[i = 1:N] <= generators[i].max)</pre>
 @variable(model, 0 <= w <= scenario.wind)</pre>
 @constraint(model, sum(g[i] for i in 1:N) + w == scenario.demand)
 # !!! New: add binary on-off variables for each generator
 @variable(model, u[i = 1:N], Bin)
 @constraint(model, [i = 1:N], g[i] <= generators[i].max * u[i])</pre>
 @constraint(model, [i = 1:N], g[i] >= generators[i].min * u[i])
 @objective(
 model,
 Min,
 sum(generators[i].variable_cost * g[i] for i in 1:N) +
 wind.variable_cost * w +
 # !!! new
 sum(generators[i].fixed_cost * u[i] for i in 1:N)
)
 optimize!(model)
 status = termination_status(model)
 if status != OPTIMAL
 return (status = status,)
 end
 @assert primal status(model) == FEASIBLE POINT
 return (
 status = status,
 g = value.(g),
 w = value(w),
 wind_spill = scenario.wind - value(w),
 u = value.(u),
 total_cost = objective_value(model),
)
end
```

solve\_unit\_commitment (generic function with 1 method)

Solve the unit commitment problem

```
solution = solve_unit_commitment(generators, wind_generator, scenario)
println("Dispatch of Generators: ", solution.g, " MW")
println("Commitments of Generators: ", solution.u)
println("Dispatch of Wind: ", solution.w, " MW")
println("Wind spillage: ", solution.wind_spill, " MW")
println("Total cost: \$", solution.total_cost)
```

```
Dispatch of Generators: [1000.0, 300.0] MW
Commitments of Generators: [1.0, 1.0]
Dispatch of Wind: 200.0 MW
Wind spillage: 0.0 MW
Total cost: $91000.0
```

# Unit commitment as a function of demand

After implementing the unit commitment model, we can now assess the interplay between the minimum power output constraints on generators and wind generation.

```
uc_df = DataFrames.DataFrame(;
 demand = Float64[],
 commitment_G1 = Float64[],
 commitment_G2 = Float64[],
 dispatch_G1 = Float64[],
 dispatch_G2 = Float64[],
 dispatch_wind = Float64[],
 spillage_wind = Float64[],
 total_cost = Float64[],
)
for demand_scale in 0.2:0.1:1.4
 new_scenario = scale_demand(scenario, demand_scale)
 sol = solve_unit_commitment(generators, wind_generator, new_scenario)
 if sol.status == OPTIMAL
 push!(
 uc_df,
 (
 new_scenario.demand,
 sol.u[1],
 sol.u[2],
 sol.g[1],
 sol.g[2],
 sol.w,
 sol.wind_spill,
 sol.total_cost,
),
)
 end
 println("Status: $(sol.status) for demand_scale = $(demand_scale)")
end
```

```
Status: OPTIMAL for demand_scale = 0.2
Status: OPTIMAL for demand_scale = 0.3
Status: OPTIMAL for demand_scale = 0.4
Status: OPTIMAL for demand_scale = 0.5
Status: OPTIMAL for demand_scale = 0.6
Status: OPTIMAL for demand_scale = 0.7
Status: OPTIMAL for demand_scale = 0.8
Status: OPTIMAL for demand_scale = 0.9
Status: OPTIMAL for demand_scale = 1.0
Status: OPTIMAL for demand_scale = 1.1
Status: OPTIMAL for demand_scale = 1.2
Status: OPTIMAL for demand_scale = 1.3
Status: OPTIMAL for demand_scale = 1.3
```

# uc\_df

	demand	commitment_G1	commitment_G2	dispatch_G1	dispatch_G2	dispatch_wind	spillage_wind	total_cost
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	300.0	1.0	0.0	100.0	0.0	200.0	0.0	16000.0
2	450.0	1.0	0.0	250.0	0.0	200.0	0.0	23500.0
3	600.0	1.0	0.0	400.0	0.0	200.0	0.0	31000.0
4	750.0	1.0	0.0	550.0	0.0	200.0	0.0	38500.0
5	900.0	1.0	0.0	700.0	0.0	200.0	0.0	46000.0
6	1050.0	1.0	0.0	850.0	0.0	200.0	0.0	53500.0
7	1200.0	1.0	0.0	1000.0	0.0	200.0	0.0	61000.0
8	1350.0	1.0	1.0	850.0	300.0	200.0	0.0	83500.0
9	1500.0	1.0	1.0	1000.0	300.0	200.0	0.0	91000.0
10	1650.0	1.0	1.0	1000.0	450.0	200.0	0.0	106000.0
11	1800.0	1.0	1.0	1000.0	600.0	200.0	0.0	121000.0
12	1950.0	1.0	1.0	1000.0	750.0	200.0	0.0	136000.0
13	2100.0	1.0	1.0	1000.0	900.0	200.0	0.0	151000.0

```
commitment_plot = StatsPlots.@df(
 uc_df,
 Plots.plot(
 :demand,
 [:commitment_G1, :commitment_G2],
 labels = ["G1" "G2"],
 title = "Commitment",
 legend = :bottomright,
 linewidth = 3,
 xlabel = "Demand [MW]",
 ylabel = "Commitment decision {0, 1}",
),
)
dispatch_plot = StatsPlots.@df(
 uc_df,
 Plots.plot(
 :demand,
 [:dispatch_G1, :dispatch_G2, :dispatch_wind],
 labels = ["G1" "G2" "Wind"],
```

title = "Dispatch [MW]",

```
legend = :bottomright,
linewidth = 3,
xlabel = "Demand",
ylabel = "Dispatch [MW]",
),
```

```
Plots.plot(commitment_plot, dispatch_plot)
```



# Nonlinear economic dispatch

As a final example, we modify our economic dispatch problem in two ways:

- · The thermal cost function is user-defined
- The output of the wind is only the square-root of the dispatch

#### import Ipopt

. . . .

thermal\_cost\_function(g)

A user-defined thermal cost function in pure-Julia! You can include nonlinearities, and even things like control flow.

```
!!! warning
 It's still up to you to make sure that the function has a meaningful
 derivative.
.....
function thermal_cost_function(g)
 if g <= 500
 return g
 else
 return g + 1e-2 * (g - 500)^2
 end
end
function solve_nonlinear_economic_dispatch(
 generators::Vector,
 wind,
 scenario;
 silent::Bool = false,
)
 model = Model(Ipopt.Optimizer)
 if silent
 set_silent(model)
 end
 @operator(model, op_tcf, 1, thermal_cost_function)
 N = length(generators)
 @variable(model, generators[i].min <= g[i = 1:N] <= generators[i].max)</pre>
 @variable(model, 0 <= w <= scenario.wind)</pre>
 @objective(
 model,
 Min,
 sum(generators[i].variable_cost * op_tcf(g[i]) for i in 1:N) +
 wind.variable_cost * w,
)
 @constraint(model, sum(g[i] for i in 1:N) + sqrt(w) == scenario.demand)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 return (
 g = value.(g),
 w = value(w),
 wind_spill = scenario.wind - value(w),
 total_cost = objective_value(model),
)
end
solution =
 solve_nonlinear_economic_dispatch(generators, wind_generator, scenario)
```

```
(g = [847.3509933774712, 648.6754966887423], w = 15.788781193899027, wind_spill = 184.211218806101,

→ total_cost = 190455.298013245)
```

Now let's see how the wind is dispatched as a function of the cost:

)

```
wind_cost = 0.0:1:100
wind_dispatch = Float64[]
for c in wind_cost
 sol = solve_nonlinear_economic_dispatch(
 generators,
 WindGenerator(c),
 scenario;
 silent = true,
)
 push!(wind_dispatch, sol.w)
end
Plots.plot(
 wind_cost,
 wind_dispatch;
 xlabel = "Cost",
 ylabel = "Dispatch [MW]",
 label = false,
```



# 10.2 Optimal power flow

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial formulates and solves an alternating current optimal power flow (AC-OPF) problem, a much-studied nonlinear problem from the field of electrical engineering.

Once we've formulated and solved the nonlinear problem, we will turn our focus to obtaining a good estimate of the objective value at the global optimum through the use of semidefinite programming.

One main purpose of this tutorial is to highlight JuMP's ability to directly formulate problems involving complexvalued decision variables and complex matrix cones such as the HermitianPSDCone object.

For another example of modeling with complex decision variables, see the Example: quantum state discrimination tutorial, and see the Complex number support section of the manual for more details.

#### Info

This tutorial takes a matrix-oriented approach focused on network nodes that simplifies the construction of semidefinite programs. Another approach is to formulate the problem focusing on network lines (known as a *branch model*) where it is easier to work with flow constraints. A general approach is provided by PowerModels.jl, an open-source framework to a broad range of power flow model formulations along with utilities for working with detailed network data.

#### Required packages

This tutorial requires the following packages:

using JuMP import Clarabel import DataFrames import Ipopt import LinearAlgebra import SparseArrays import Test

#### Initial formulation

Optimal power flow problems for electrical transmission typically pose the following question: what is the most cost-effective operation of electricity generators while meeting constraints on the safe limits of network components?

We'll use the 9-node network test case case9mod to explore this problem.

The graph of the network, shown here, has three nodes (or *buses*) each for the different purposes of generation G (nodes 1, 2, and 3), trans-shipment (nodes 4, 6, and 8), and demand D (nodes 5, 7, and 9).

This example is a modified version of the MATPOWER ((Zimmerman *et al.*, 2011)) test case case9 (archive) created by (Bukhsh *et al.*, 2013) for their test case archive of optimal power flow problems with local optima. This test case is also extensively evaluated in (Krasko and Rebennack, 2017).

Here bus and network node are taken as analogous terms, as are branch and transmission line.

For future reference, let's name the number of nodes in the network:

N = 9;

The network data can be summarised using a small number of arrays. Using the sparsevec function from the SparseArrays standard library package, we can give the indices and values of the non-zero data points:



Figure 10.1: Nine Nodes

```
Real generation: lower (`lb`) and upper (`ub`) bounds
P_Gen_lb = SparseArrays.sparsevec([1, 2, 3], [10, 10, 10], N)
P_Gen_ub = SparseArrays.sparsevec([1, 2, 3], [250, 300, 270], N)
Reactive generation: lower (`lb`) and upper (`ub`) bounds
Q_Gen_lb = SparseArrays.sparsevec([1, 2, 3], [-5, -5, -5], N)
Q_Gen_ub = SparseArrays.sparsevec([1, 2, 3], [300, 300, 300], N)
Power demand levels (real, reactive, and complex form)
P_Demand = SparseArrays.sparsevec([5, 7, 9], [54, 60, 75], N)
Q_Demand = SparseArrays.sparsevec([5, 7, 9], [18, 21, 30], N)
S_Demand = P_Demand + im * Q_Demand
```

```
9-element SparseArrays.SparseVector{Complex{Int64}, Int64} with 3 stored entries:
[5] = 54+18im
[7] = 60+21im
[9] = 75+30im
```

The key decision variables are the real power injections  $P^G$  and reactive power injections  $Q^G$  over the allowed range of the generators. All other buses must restrict their generation variables to 0. On the other hand, these non-generator nodes have a fixed real and reactive power demand, denoted  $P^D$  and  $Q^D$  respectively (these are fixed at 0 in the case of trans-shipment and generator nodes).

The cost of operating each generator is modeled as a quadratic function of its real power output; in our specific test case, the objective function to minimize is:

min0.11 
$$(P_1^G)^2 + 5P_1^G + 150$$
 (10.1)

- $+0.085 (P_2^G)^2 + 1.2P_2^G + 600$  (10.2)
- $+0.1225 (P_3^G)^2 + P_3^G + 335$  (10.3)
  - (10.4)

Let's create an initial JuMP model with some of this data:

```
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, P_Gen_lb[i] <= P_G[i in 1:N] <= P_Gen_ub[i])
@objective(
 model,
 Min,
 (0.11 * P_G[1]^2 + 5 * P_G[1] + 150) +
 (0.085 * P_G[2]^2 + 1.2 * P_G[2] + 600) +
 (0.1225 * P_G[3]^2 + P_G[3] + 335),
);</pre>
```

Even before solving an optimization problem, we can estimate a lower bound on the best objective value by substituting the lower bound on each generator's real power range (all 10, as it turns out in this case):

```
basic_lower_bound = value(lower_bound, objective_function(model));
println("Objective value (basic lower bound) : $basic_lower_bound")
```

Objective value (basic lower bound) : 1188.75

to see that we can do no better than an objective cost of 1188.75.

(Direct substitution works because a quadratic function of a single variable x with positive coefficients is strictly increasing for all  $x \ge 0$ .)

In fact, we can get a quick but even better estimate from the direct observation that the real power generated must meet or exceed the real power demand:

```
@constraint(model, sum(P_G) >= sum(P_Demand))
optimize!(model)
assert_is_solved_and_feasible(model)
better_lower_bound = round(objective_value(model); digits = 2)
println("Objective value (better lower bound): $better_lower_bound")
```

Objective value (better lower bound): 2733.55

However, there are additional power flow constraints that must be satisfied.

Power must flow from one or more generation nodes through the transmission lines and end up at a demand node. The state variables of our steady-state alternating current (AC) electrical network are *complex-valued* voltage variables  $V_1, \ldots, V_N$ . Voltages capture both a magnitude and phase of the node's electrical state in relation to the rest of the system. An AC power system also extends the notion of resistance in wires found in a direct current (DC) circuit to a complex quantity, known as the *impedance*, of each transmission line. The reciprocal of impedance is known as *admittance*. Together, these complex quantities are used to express a complex version of *Ohm's law*: current flow through a line is proportional to the difference in voltages on each end of the line, multiplied by the admittance.

#### **Network data**

Let's assemble the data we need for writing the complex power flow constraints. The data for the problem consists of a list of the real and imaginary parts of the line impedance. We obtain the following data table from the branch data section of the case9mod MATPOWER format file:

```
branch_data = DataFrames.DataFrame([
 (1, 4, 0.0, 0.0576, 0.0),
 (4, 5, 0.017, 0.092, 0.158),
 (6, 5, 0.039, 0.17, 0.358),
 (3, 6, 0.0, 0.0586, 0.0),
 (6, 7, 0.0119, 0.1008, 0.209),
 (8, 7, 0.0085, 0.072, 0.149),
 (2, 8, 0.0, 0.0625, 0.0),
 (8, 9, 0.032, 0.161, 0.306),
 (4, 9, 0.01, 0.085, 0.176),
]);
DataFrames.rename!(branch_data, [:F_BUS, :T_BUS, :BR_R, :BR_X, :BR_Bc])
```

The first two columns describe the network, supplying the *from* and *to* connection points of the lines. The last three columns give the branch resistance, branch reactance and *line-charging susceptance*.

# CHAPTER 10. APPLICATIONS

	F_BUS	T_BUS	BR_R	BR_X	BR_Bc
	Int64	Int64	Float64	Float64	Float64
1	1	4	0.0	0.0576	0.0
2	4	5	0.017	0.092	0.158
3	6	5	0.039	0.17	0.358
4	3	6	0.0	0.0586	0.0
5	6	7	0.0119	0.1008	0.209
6	8	7	0.0085	0.072	0.149
7	2	8	0.0	0.0625	0.0
8	8	9	0.032	0.161	0.306
9	4	9	0.01	0.085	0.176

We will also need to reference the base\_MVA number (used for re-scaling):

base\_MVA = 100;

and the number of lines:

M = size(branch\_data, 1)

9

From the first two columns of the branch data table, we can create a sparse incidence matrix that simplifies handling of the network layout:

A =
SparseArrays.sparse(branch\_data.F\_BUS, 1:M, 1, N, M) +
SparseArrays.sparse(branch\_data.T\_BUS, 1:M, -1, N, M)

9×9 SparseArrays.SparseMatrixCSC{Int64, Int64} with 18 stored entries:

1	•	•	•	•	•	•	•	•
·	•	•	•	•	•	1	•	•
·	•	•	1	•	•		•	•
-1	1	•	•	•	•		•	1
·	-1	- 1	•	•	•	·	•	•
·	•	1	-1	1	•		•	•
·	•	•	•	-1	-1		•	•
·	•	•	•	•	1	- 1	1	•
·	•	•	•	•	•	·	-1	-1

We form the network impedance vector from the next two columns

z = (branch\_data.BR\_R .+ im \* branch\_data.BR\_X) / base\_MVA;

and calculate it's corresponding bus admittance matrix as

```
Y_0 = A * SparseArrays.spdiagm(1 ./ z) * A';
```

while the last column gives the branch line-charging susceptance

y\_sh = 1 / 2 \* (im \* branch\_data.BR\_Bc) \* base\_MVA;

and leads to the shunt admittance matrix

```
Y_sh = SparseArrays.spdiagm(
 LinearAlgebra.diag(A * SparseArrays.spdiagm(y_sh) * A'),
);
```

(The construction of the shunt admittance matrix Y\_sh looks somewhat more complicated than Y\_0 because we only want to add the diagonal elements in the calculation; the line-charging is used only in the nodal voltage terms and not the line voltage terms.)

The full  $\mathit{bus} \ \mathit{admittance} \ matrix \ Y$  is then defined as

 $Y = Y_0 + Y_sh;$ 

#### JuMP model

Now we're ready to write the complex power flow constraints we need to more accurately model the electricity system.

We'll introduce a number of constraints that model both the physics and operational requirements.

Let's start by initializing a new model:

```
model = Model(Ipopt.Optimizer)
set_silent(model)
```

Then we'll create the nodal power generation variables:

```
@variable(
 model,
 S_G[i in 1:N] in ComplexPlane(),
 lower_bound = P_Gen_lb[i] + Q_Gen_lb[i] * im,
 upper_bound = P_Gen_ub[i] + Q_Gen_ub[i] * im,
)
```

```
9-element Vector{GenericAffExpr{ComplexF64, VariableRef}}:
 real(S_G[1]) + imag(S_G[1]) im
 real(S_G[2]) + imag(S_G[2]) im
 real(S_G[3]) + imag(S_G[3]) im
 real(S_G[4]) + imag(S_G[4]) im
 real(S_G[5]) + imag(S_G[5]) im
```

```
real(S_G[6]) + imag(S_G[6]) im
real(S_G[7]) + imag(S_G[7]) im
real(S_G[8]) + imag(S_G[8]) im
real(S_G[9]) + imag(S_G[9]) im
```

We need complex nodal voltages (the system state variables):

@variable(model, V[1:N] in ComplexPlane(), start = 1.0 + 0.0im)

```
9-element Vector{GenericAffExpr{ComplexF64, VariableRef}}:
 real(V[1]) + imag(V[1]) im
 real(V[2]) + imag(V[2]) im
 real(V[3]) + imag(V[3]) im
 real(V[4]) + imag(V[4]) im
 real(V[5]) + imag(V[5]) im
 real(V[6]) + imag(V[6]) im
 real(V[7]) + imag(V[7]) im
 real(V[8]) + imag(V[8]) im
 real(V[9]) + imag(V[9]) im
```

and operational constraints for maintaining voltage magnitude levels:

@constraint(model, [i in 1:N], 0.9<sup>2</sup> <= real(V[i])<sup>2</sup> + imag(V[i])<sup>2</sup> <= 1.1<sup>2</sup>)

We also need to fix an origin or *reference angle* from which all other complex voltage angles (arguments) are determined. Here we will use node 1 as the nominated *reference bus*. Fixing the imaginary component of a reference bus to zero sets its complex voltage angle to 0, while constraining the real part to be non-negative disallows equivalent solutions that are just a reflection by 180 degrees:

```
@constraint(model, imag(V[1]) == 0);
@constraint(model, real(V[1]) >= 0);
```

The power flow equations express a conservation of energy (power) principle, where power generated less the power consumed must balance the power exchanged with the network:

@constraint(model, S\_G - S\_Demand .== V .\* conj(Y \* V))

```
9-element Vector{ConstraintRef{Model,
\hookrightarrow \quad \texttt{MathOptInterface.ConstraintIndex{MathOptInterface.ScalarQuadraticFunction{ComplexF64}, and a second second
→ MathOptInterface.EqualTo{ComplexF64}}, ScalarShape}}:
 -1736.111111111111 real(V[1])² + 1736.1111111111111 real(V[1])*real(V[4]) + 1736.11111111111
 \rightarrow real(V[1])*imag(V[4]) - 1736.111111111111 im imag(V[1])² - 1736.11111111111
 \rightarrow imag(V[1])*real(V[4]) + 1736.111111111111 im imag(V[1])*imag(V[4]) + real(S_G[1]) +
 \rightarrow imag(S_G[1]) im = 0
 -1600im real(V[2])² + 1600im real(V[2])*real(V[8]) + 1600 real(V[2])*imag(V[8]) - 1600im
 \rightarrow imag(V[2])² - 1600 imag(V[2])*real(V[8]) + 1600im imag(V[2])*imag(V[8]) + real(S_G[2]) +
 \rightarrow imag(S_G[2]) im = 0
 -1706.484641638225im real(V[3])² + 1706.484641638225im real(V[3])*real(V[6]) + 1706.484641638225
 \hookrightarrow real(V[3])*imag(V[6]) - 1706.484641638225im imag(V[3])² - 1706.484641638225
 imag(V[3])*real(V[6]) + 1706.484641638225im imag(V[3])*imag(V[6]) + real(S_G[3]) +
 \rightarrow imag(S_G[3]) im = 0
 1736.111111111111 imag(V[4])*real(V[1]) - 1736.11111111111 imag(V[4])*real(V[1]) +
 1736.11111111111 real(V[4])*imag(V[1]) + 1736.111111111111 im imag(V[4])*imag(V[1]) +
 (-330.7378962025307 - 3930.8888726118976im) real(V[4])² + (194.21912487147264 +
 1051.0682051867932im) real(V[4])*real(V[5]) + (1051.0682051867932 - 194.21912487147264im)
 real(V[4])*imag(V[5]) + (136.51877133105802 + 1160.409556313993im) real(V[4])*real(V[9]) +
 3930.8888726118976im) imag(V[4])² + (-1051.0682051867932 + 194.21912487147264im)
 → imag(V[4])*real(V[5]) + (194.21912487147264 + 1051.0682051867932im) imag(V[4])*imag(V[5]) +
 (-1160.409556313993 + 136.51877133105802im) imag(V[4])*real(V[9]) + (136.51877133105802 +
 \hookrightarrow
 \rightarrow 1160.409556313993im) imag(V[4])*imag(V[9]) + real(S_G[4]) + imag(S_G[4]) im = 0
 \rightarrow \quad 194.21912487147264 \text{im}) \quad \text{imag}(\text{V[5]})*\text{real}(\text{V[4]}) \ + \ (1051.0682051867932 \ - \ 194.21912487147264 \text{im})
 \rightarrow real(V[5])*imag(V[4]) + (194.21912487147264 + 1051.0682051867932im) imag(V[5])*imag(V[4]) +
 (-322.4200387138841 - 1584.0927014229458im) real(V[5])² + (128.20091384241147 +
 → 558.8244962361526im) real(V[5])*real(V[6]) + (558.8244962361526 - 128.20091384241147im)
 → real(V[5])*imag(V[6]) + (-322.4200387138841 - 1584.0927014229458im) imag(V[5])² +
 (-558.8244962361526 + 128.20091384241147im) imag(V[5])*real(V[6]) + (128.20091384241147 +
 \rightarrow 558.8244962361526im) imag(V[5])*imag(V[6]) + real(S_G[5]) + imag(S_G[5]) im = (54 + 18im)
 1706.484641638225im real(V[6])*real(V[3]) - 1706.484641638225 imag(V[6])*real(V[3]) +
 → 1706.484641638225 real(V[6])*imag(V[3]) + 1706.484641638225im imag(V[6])*imag(V[3]) +
 (128.20091384241147 + 558.8244962361526im) real(V[6])*real(V[5]) + (-558.8244962361526 +
 \rightarrow
 128.20091384241147im) imag(V[6])*real(V[5]) + (558.8244962361526 - 128.20091384241147im)
 \hookrightarrow
 real(V[6])*imag(V[5]) + (128.20091384241147 + 558.8244962361526im) imag(V[6])*imag(V[5]) +
 \hookrightarrow
 (-243.70966193142118 - 3215.386180510695im) real(V[6])² + (115.5087480890097 +
 \hookrightarrow
 978.4270426363173im) real(V[6])*real(V[7]) + (978.4270426363173 - 115.5087480890097im)
 \hookrightarrow
 real(V[6])*imag(V[7]) + (-243.70966193142118 - 3215.386180510695im) imag(V[6])² +
 (-978.4270426363173 + 115.5087480890097im) imag(V[6])*real(V[7]) + (115.5087480890097 +
 978.4270426363173im) imag(V[6])*imag(V[7]) + real(S_G[6]) + imag(S_G[6]) im = 0
 (115.5087480890097 + 978.4270426363173im) real(V[7])*real(V[6]) + (-978.4270426363173 +
 115.5087480890097im) imag(V[7])*real(V[6]) + (978.4270426363173 - 115.5087480890097im)
 real(V[7])*imag(V[6]) + (115.5087480890097 + 978.4270426363173im) imag(V[7])*imag(V[6]) +
 (-277.22099541362326 - 2330.3249023271615im) real(V[7])^2 + (161.71224732461357 + 161.7122473246137 + 161.7122473246137 + 161.712247387 + 161.712247387 + 161.712247387 + 161.71237 + 161.712277 + 161.71277 + 161.71277 + 161.71277 + 161.71277 + 161.71277 + 161.71277 + 161.7127 + 161.71277 + 161.7127 + 161.71277 + 161.7127 + 161.717 + 160.717 + 160.717 + 160.717 + 160.717 + 160.777 + 160.777 + 160.777 + 160.777 + 160.777 + 160.777 + 160.777
 1369.7978596908442im) real(V[7])*real(V[8]) + (1369.7978596908442 - 161.71224732461357im)
 real(V[7])*imag(V[8]) + (-277.22099541362326 - 2330.3249023271615im) imag(V[7])² +
 (-1369.7978596908442 + 161.71224732461357im) imag(V[7])*real(V[8]) + (161.71224732461357 +
 \hookrightarrow
 → 1369.7978596908442im) imag(V[7])*imag(V[8]) + real(S_G[7]) + imag(S_G[7]) im = (60 + 21im)
 1600 \text{ im real}(V[8]) * \text{real}(V[2]) - 1600 \text{ imag}(V[8]) * \text{real}(V[2]) + 1600 \text{ real}(V[8]) * \text{imag}(V[2]) + 1600 \text{ imag}(V[2]) + 1600 \text{ imag}
 → imag(V[8])*imag(V[2]) + (161.71224732461357 + 1369.7978596908442im) real(V[8])*real(V[7]) +
 → (-1369.7978596908442 + 161.71224732461357im) imag(V[8])*real(V[7]) + (1369.7978596908442 -
 → 161.71224732461357im) real(V[8])*imag(V[7]) + (161.71224732461357 + 1369.7978596908442im)
 → imag(V[8])*imag(V[7]) + (-280.4726852537284 - 3544.5613130217034im) real(V[8])² +
 → (118.76043792911486 + 597.5134533308592im) real(V[8])*real(V[9]) + (597.5134533308592 -
 → 118.76043792911486im) real(V[8])*imag(V[9]) + (-280.4726852537284 - 3544.5613130217034im)
 imag(V[8])² + (-597.5134533308592 + 118.76043792911486im) imag(V[8])*real(V[9]) +
 \rightarrow
```
```
(136.51877133105802 + 1160.409556313993im) real(V[9])*real(V[4]) + (-1160.409556313993 +

→ 136.51877133105802im) imag(V[9])*real(V[4]) + (1160.409556313993 - 136.51877133105802im)

→ real(V[9])*imag(V[4]) + (136.51877133105802 + 1160.409556313993im) imag(V[9])*imag(V[4]) +

→ (118.76043792911486 + 597.5134533308592im) real(V[9])*real(V[8]) + (-597.5134533308592 +

118.76043792911486im) imag(V[9])*real(V[8]) + (597.5134533308592 - 118.76043792911486im)

→ real(V[9])*imag(V[8]) + (118.76043792911486 + 597.5134533308592 - 118.76043792911486im)

→ real(V[9])*imag(V[8]) + (118.76043792911486 + 597.5134533308592im) imag(V[9])*imag(V[8]) +

→ (-255.27920926017288 - 1733.8230096448524im) real(V[9])² + (-255.27920926017288 -

→ 1733.8230096448524im) imag(V[9])² + real(S_G[9]) + imag(S_G[9]) im = (75 + 30im)
```

As above, the objective function is a quadratic cost of real power:

```
P_G = real(S_G)
@objective(
 model,
 Min,
 (0.11 * P_G[1]^2 + 5 * P_G[1] + 150) +
 (0.085 * P_G[2]^2 + 1.2 * P_G[2] + 600) +
 (0.1225 * P_G[3]^2 + P_G[3] + 335),
);
```

We're finally ready to solve our nonlinear AC-OPF problem:

```
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : Ipopt
| Termination
| | termination_status : LOCALLY_SOLVED
| | result_count : 1
| L raw_status : Solve_Succeeded
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| L objective_value : 3.08784e+03
L Work counters
| solve_time (sec) : 5.45287e-03
L barrier_iterations : 16
```

```
objval_solution = round(objective_value(model); digits = 2)
println("Objective value (feasible solution) : $(objval_solution)")
```

Objective value (feasible solution) : 3087.84

The solution's power generation (in rectangular form) and complex voltage values (in polar form using degrees) are:

```
DataFrames.DataFrame(;
 Bus = 1:N,
 ComplexPowerGen = round.(value.(S_G); digits = 2),
 VoltageMagnitude = round.(abs.(value.(V)); digits = 2),
 VoltageAngle_Deg = round.(rad2deg.(angle.(value.(V))); digits = 2),
)
```

	Bus	ComplexPowerGen	VoltageMagnitude	VoltageAngle_Deg
	Int64	Complex	Float64	Float64
1	1	10.0-5.0im	0.91	-0.0
2	2	125.37-5.0im	0.92	12.37
3	3	57.03-5.0im	0.94	7.01
4	4	0.0+0.0im	0.91	-0.4
5	5	0.0+0.0im	0.92	-0.73
6	6	0.0+0.0im	0.94	4.84
7	7	0.0+0.0im	0.93	4.52
8	8	0.0+0.0im	0.93	7.12
9	9	0.0+0.0im	0.9	-0.63

## **Relaxations and better objective bounds**

The lpopt solver uses an interior-point algorithm. It has local optimality guarantees, but is unable to certify whether the solution is globally optimal. The solution we found is indeed globally optimal. The work to verify this has been done in (Bukhsh *et al.*, 2013) and (Krasko and Rebennack, 2017), and different solvers (such as Gurobi, SCIP and GLOMIQO) are also able to verify this.

The techniques of convex relaxations can also be used to improve on our current best lower bound:

better\_lower\_bound

2733.55

To this end, observe that the nonlinear constraints in the AC-OPF formulation are quadratic equalities for power flow along with quadratic voltage inequalities.

Let's linearize these constraints by first making the substitution  $W = VV^*$ , where:

$$W = VV^* \quad \Longleftrightarrow \quad W_{ii} = |V_i|^2, \quad W_{ik} = V_i \ \overline{V_k}, \quad \forall i, k \in \{1, \dots, N\}$$

and where  $V^*$  is the conjugate transpose of V.

On the face of it, this turns a quadratic voltage bound constraint like:

$$v_L \le |V_i|^2 \le v_U, \quad i \in \{1, \dots, N\}$$

for some real  $v_L$  and  $v_U$  into a simple two-sided bound:

$$v_L \le W_{ii} \le v_U,$$

while each quadratic expression for the nodal power term:

$$S_i^{Node} = V_i \overline{(YV)_i}$$

becomes the linear combination:

$$S_i^{Node} = (E_{ii}Y^T) \bullet W.$$

Here  $A \bullet B = tr(A^*B)$  is the Frobenius inner product of two complex matrices, while  $E_{kn}$  denotes the matrix unit with a single nonzero entry of 1 in row k and column n.

E(k, n) = SparseArrays.sparse([k], [n], 1, N, N);

Of course, we've shifted the nonlinearity into the equality constraint  $W = VV^*$ : it is this constraint we will now relax using a semidefinite programming approach.

We will make use of complex voltages and relax  $W = VV^*$  to;

$$W \succeq VV^*,$$

where the relation  $\succeq$  is the ordering in the Hermitian positive semidefinite cone.

The above constraint is equivalent to:

$$\begin{bmatrix} 1 & V^* \\ V & W \end{bmatrix} \succeq 0$$

by the theory of the Schur complement. This matrix inequality implies a number of second-order cone constraints by taking certain  $2 \times 2$  minors of the matrix for each  $i \in \{1, ..., N\}$ :

$$\begin{bmatrix} 1 & V_i^* \\ V_i & W_{ii} \end{bmatrix} \succeq 0,$$

which is equivalent to the real second-order cone inequality:

$$\operatorname{real}(W_{ii}) \ge \operatorname{real}(V_i)^2 + \operatorname{imag}(V_i)^2.$$

We include these implied constraints as well for demonstration purposes.

Putting it all together we get the following semidefinite relaxation of the AC-OPF problem:

```
model = Model(Clarabel.Optimizer)
set_attribute(model, "tol_gap_rel", 1e-3)
set_attribute(model, "tol_feas", 1e-3)
set_attribute(model, "tol_ktratio", le-3)
@variable(
 model,
 S_G[i in 1:N] in ComplexPlane(),
 lower_bound = P_Gen_lb[i] + Q_Gen_lb[i] * im,
 upper_bound = P_Gen_ub[i] + Q_Gen_ub[i] * im,
)
@variable(model, W[1:N, 1:N] in HermitianPSDCone())
@variable(model, V[1:N] in ComplexPlane(), start = 1.0 + 0.0im)
@constraint(model, [i in 1:N], 0.9^2 <= real(W[i, i]) <= 1.1^2)</pre>
@constraint(model, real(V[1]) >= 0)
@constraint(model, imag(V[1]) == 0)
@constraint(model, 0.9 <= real(V[1]) <= 1.1)</pre>
@constraint(model, LinearAlgebra.Hermitian([1 V'; V W]) in HermitianPSDCone())
2 x 2 minor inequalities:
@constraint(
 model,
 [i in 1:N],
 [0.5, real(W[i, i]), real(V[i]), imag(V[i])] in RotatedSecondOrderCone()
)
@constraint(
 model,
 [i in 1:N],
 S_G[i] - S_Demand[i] == LinearAlgebra.tr((conj(Y) * E(i, i)) * W),
)
P_G = real(S_G)
@objective(
 model,
 Min,
 (0.11 * P_G[1]^2 + 5 * P_G[1] + 150) +
 (0.085 * P_G[2]^2 + 1.2 * P_G[2] + 600) +
 (0.1225 * P_G[3]^2 + P_G[3] + 335),
)
```

```
optimize!(model)
```

Clarabel.jl v0.10.0 - Clever Acronym (c) Paul Goulart University of Oxford, 2022 problem: variables = 117 constraints = 493 = 3 nnz(P) = 547 nnz(A) cones (total) = 14= 1, numel = 19 : Zero : Nonnegative = 2, numel = (18,39) : SecondOrder = 9, numel = (4, 4, 4, 4, ..., 4): PSDTriangle = 2, numel = (171,210)

```
settings:
 linear algebra: direct / qdldl, precision: Float64
 max iter = 200, time limit = Inf, max step = 0.990
 tol_feas = 1.0e-03, tol_gap_abs = 1.0e-08, tol_gap_rel = 1.0e-03,
 static reg : on, \epsilon 1 = 1.0e-08, \epsilon 2 = 4.9e-32
 dynamic reg: on, \varepsilon = 1.0e-13, \delta = 2.0e-07
 iter refine: on, reltol = 1.0e-13, abstol = 1.0e-12,
 max iter = 10, stop ratio = 5.0
 equilibrate: on, min_scale = 1.0e-04, max_scale = 1.0e+04
 max iter = 10
iter pcost
 dcost
 dres
 k/t
 step
 gap
 pres
 μ

 0 6.6673e+03 -3.0980e+05 4.75e+01 9.32e-02 8.43e-01 1.00e+00 3.34e+03 -----
 3.3968e+03 -6.4534e+04 2.00e+01 1.95e-02 3.85e-01 2.41e+02 9.42e+02 8.13e-01
 1
 2
 1.9798e+03 -1.7012e+04 9.59e+00 5.49e-03 2.64e-01 1.51e+02 3.06e+02 9.90e-01
 3
 2.1414e+03 -2.3620e+03 2.10e+00 1.22e-03 8.46e-02 3.29e+01 7.63e+01 8.60e-01
 4
 2.0597e+03 1.1123e+03 8.52e-01 2.41e-04 5.67e-02 5.52e+00 1.93e+01 8.55e-01
 5 1.6312e+03 1.4437e+03 1.30e-01 3.59e-05 1.04e-02 9.18e-01 3.20e+00 9.12e-01
 6 1.5768e+03 1.5579e+03 1.21e-02 3.46e-06 1.05e-03 9.28e-02 3.09e-01 9.19e-01
 7 1.5691e+03 1.5654e+03 2.35e-03 6.96e-07 2.05e-04 1.96e-02 5.73e-02 8.55e-01
 8 1.5657e+03 1.5640e+03 1.08e-03 4.01e-07 9.61e-05 8.54e-03 2.47e-02 7.64e-01
 9 1.5653e+03 1.5641e+03 7.88e-04 2.96e-07 6.55e-05 6.14e-03 1.79e-02 4.07e-01

Terminated with status = solved
solve time = 56.2ms
```

```
assert_is_solved_and_feasible(model; allow_almost = true)
sdp_relaxation_lower_bound = round(objective_value(model); digits = 2)
println(
 "Objective value (W & V relax. lower bound): $sdp_relaxation_lower_bound",
)
```

Objective value (W & V relax. lower bound): 2754.1

We can more easily see solution values by rounding out noisy data:

W\_1 = SparseArrays.sparse(round.(value.(W); digits = 2))

```
9×9 SparseArrays.SparseMatrixCSC{ComplexF64, Int64} with 81 stored entries:
1.19+0.0im 1.14-0.07im 1.13-0.05im ... 1.15-0.02im 1.14+0.05im
1.14+0.07im 1.2+0.0im 1.15+0.02im 1.17+0.05im 1.14+0.12im
1.13+0.05im 1.15-0.02im 1.19+0.0im 1.16+0.03im 1.14+0.1im
1.14-0.03im 1.15-0.1im 1.15-0.08im 1.17-0.05im 1.15+0.03im
1.15-0.05im 1.15-0.12im 1.15-0.1im 1.17-0.07im 1.15+0.01im
1.15+0.01im 1.17-0.06im 1.16-0.03im ... 1.18-0.0im 1.16+0.07im
1.15-0.01im 1.16-0.08im 1.16-0.06im 1.18-0.03im 1.16+0.05im
```

1.15+0.02im	1.17-0.05im	1.16-0.03im	1.18+0.0im	1.16+0.07im
1.14-0.05im	1.14-0.12im	1.14-0.1im	1.16-0.07im	1.15+0.0im

and recover an approximation to the voltage variables as:

```
DataFrames.DataFrame(;
 Bus = 1:N,
 Magnitude = round.(abs.(value.(V)); digits = 2),
 AngleDeg = round.(rad2deg.(angle.(value.(V))); digits = 2),
)
```

	Bus	Magnitude	AngleDeg
	Int64	Float64	Float64
1	1	0.95	-0.0
2	2	0.84	3.78
3	3	0.83	2.67
4	4	0.85	-1.21
5	5	0.85	-2.11
6	6	0.86	0.94
7	7	0.86	-0.17
8	8	0.86	1.15
9	9	0.85	-2.46

For further information on exploiting sparsity see (Jabr, 2012).

This relaxation has the advantage that we can work directly with complex voltages to extend the formulation, strengthen the relaxation and gain additional approximate information about the voltage variables.

## **10.3** Serving web apps

This tutorial was generated using Literate.jl. Download the source as a .jl file.

This tutorial demonstrates how to setup and serve JuMP models via a REST API.

In the example app we are building, we solve a trivial mixed-integer program, which is parameterized by the lower bound of a variable. To call the service, users send an HTTP POST request with JSON contents indicating the lower bound. The returned value is the solution of the mixed-integer program as JSON.

First, we need JuMP and a solver:

using JuMP import HiGHS

We also need HTTP.jl to act as our REST server, and JSON.jl to marshal data.

import HTTP
import JSON

## The server side

The core components of our REST server are *endpoints*. These are functions which accept a Dict{String,Any} of input parameters, and return a Dict{String,Any} as output. The types are Dict{String,Any} because we're going to read these to and from JSON.

Here's a very simple endpoint: it accepts params as input, formulates and solves a trivial mixed-integer program, and then returns a dictionary with the result.

```
function endpoint_solve(params::Dict{String,Any})
 if !haskey(params, "lower_bound")
 return Dict{String,Any}(
 "status" => "failure",
 "reason" => "missing lower_bound param",
)
 elseif !(params["lower_bound"] isa Real)
 return Dict{String,Any}(
 "status" => "failure",
 "reason" => "lower_bound is not a number",
)
 end
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, x >= params["lower_bound"], Int)
 optimize!(model)
 ret = Dict{String,Any}(
 "status" => "okay",
 "terminaton_status" => termination_status(model),
 "primal_status" => primal_status(model),
)
 # Only include the `x` key if it has a value.
 if primal_status(model) == FEASIBLE_POINT
 ret["x"] = value(x)
 end
 return ret
end
```

endpoint\_solve (generic function with 1 method)

When we call this, we get:

endpoint\_solve(Dict{String,Any}("lower\_bound" => 1.2))

Dict{String, Any} with 4 entries: "status" => "okay" "x" => 2.0 "primal\_status" => FEASIBLE\_POINT "terminaton\_status" => OPTIMAL

```
endpoint_solve(Dict{String,Any}())
```

```
Dict{String, Any} with 2 entries:
 "status" => "failure"
 "reason" => "missing lower_bound param"
```

For a second function, we need a function that accepts an HTTP.Request object and returns an HTTP.Response object.

```
function serve_solve(request::HTTP.Request)
 data = JSON.parse(String(request.body))
 solution = endpoint_solve(data)
 return HTTP.Response(200, JSON.json(solution))
end
```

serve\_solve (generic function with 1 method)

Finally, we need an HTTP server. There are a variety of ways you can do this in HTTP.jl. We use an explicit Sockets.listen so we have manual control of when we shutdown the server.

```
function setup_server(host, port)
 server = HTTP.Sockets.listen(host, port)
 HTTP.serve!(host, port; server = server) do request
 try
 # Extend the server by adding other endpoints here.
 if request.target == "/api/solve"
 return serve_solve(request)
 else
 return HTTP.Response(404, "target $(request.target) not found")
 end
 catch err
 # Log details about the exception server-side
 @info "Unhandled exception: $err"
 # Return a response to the client
 return HTTP.Response(500, "internal error")
 end
 end
 return server
end
```

setup\_server (generic function with 1 method)

Warning

HTTP.jl does not serve requests on a separate thread. Therefore, a long-running job will block the main thread, preventing concurrent users from submitting requests. To work-around this, read HTTP.jl issue 798 or watch Building Microservices and Applications in Julia from JuliaCon 2020.

server = setup\_server(HTTP.ip"127.0.0.1", 8080)

Sockets.TCPServer(RawFD(37) active)

#### The client side

Now that we have a server, we can send it requests via this function:

```
function send_request(data::Dict; endpoint::String = "solve")
 ret = HTTP.request(
 "POST",
 # This should match the URL and endpoint we defined for our server.
 "http://127.0.0.1:8080/api/$endpoint",
 ["Content-Type" => "application/json"],
 JSON.json(data),
)
 if ret.status != 200
 # This could happen if there are time-outs, network errors, etc.
 return Dict(
 "status" => "failure",
 "code" => ret.status,
 "body" => String(ret.body),
)
 end
 return JSON.parse(String(ret.body))
end
```

send\_request (generic function with 1 method)

Let's see what happens:

send\_request(Dict("lower\_bound" => 0))

```
Dict{String, Any} with 4 entries:
 "status" => "okay"
 "x" => 0.0
 "primal_status" => "FEASIBLE_POINT"
 "terminaton_status" => "OPTIMAL"
```

send\_request(Dict("lower\_bound" => 1.2))

```
Dict{String, Any} with 4 entries:
 "status" => "okay"
 "x" => 2.0
 "primal_status" => "FEASIBLE_POINT"
 "terminaton_status" => "OPTIMAL"
```

If we don't send a lower\_bound, we get:

send\_request(Dict("invalid\_param" => 1.2))

```
Dict{String, Any} with 2 entries:
 "status" => "failure"
 "reason" => "missing lower_bound param"
```

If we don't send a lower\_bound that is a number, we get:

send\_request(Dict("lower\_bound" => "1.2"))

```
Dict{String, Any} with 2 entries:
 "status" => "failure"
 "reason" => "lower_bound is not a number"
```

Finally, we can shutdown our HTTP server:

close(server)

[ Info: Server on 127.0.0.1:8080 closing

#### Next steps

For more complicated examples relating to HTTP servers, consult the HTTP.jl documentation.

To see how you can integrate this with a larger JuMP model, read Design patterns for larger models.

## 10.4 Two-stage stochastic programs

This tutorial was generated using Literate.jl. Download the source as a .jl file.

The purpose of this tutorial is to demonstrate how to model and solve a two-stage stochastic program.

Info

The JuMP extension InfiniteOpt.jl can also be used to model and solve two-stage stochastic programs. The JuMP extension SDDP.jl can be used to model and solve multi-stage stochastic programs.

This tutorial uses the following packages

```
using JuMP
import Distributions
import HiGHS
import Plots
import StatsPlots
import Statistics
```

#### Background

During the week, you are a busy practitioner of Operations Research. To escape the drudgery of mathematics, you decide to open a side business selling creamy mushroom pies with puff pastry. After a few weeks, it quickly becomes apparent that operating a food business is not so easy.

The pies must be prepared in the morning, *before* you open for the day and can gauge the level of demand. If you bake too many, the unsold pies at the end of the day must be discarded and you have wasted time and money on their production. But if you bake too few, then there may be un-served customers and you could have made more money by baking more pies.

After a few weeks of poor decision making, you decide to put your knowledge of Operations Research to good use, starting with some data collection.

Each pie costs you \$2 to make, and you sell them at \$5 each. Disposal of an unsold pie costs \$0.10. Based on three weeks of data collected, in which you made 200 pies each week, you sold 150, 190, and 200 pies. Thus, as a guess, you assume a triangular distribution of demand with a minimum of 150, a median of 200, and a maximum of 250.

We can model this problem by a two-stage stochastic program. In the first stage, we decide a quantity of pies to make x. We make this decision before we observe the demand  $d_{\omega}$ . In the second stage, we sell  $y_{\omega}$  pies, and incur any costs for unsold pies.

We can formulate this problem as follows:

$$\max_{x,y_{\omega}} -2x + \mathbb{E}_{\omega}[5y_{\omega} - 0.1(x - y_{\omega})]$$
$$y_{\omega} \le x \qquad \qquad \forall \omega \in \Omega$$
$$0 \le y_{\omega} \le d_{\omega} \qquad \qquad \forall \omega \in \Omega$$
$$x > 0.$$

#### Sample Average approximation

If the distribution of demand is continuous, then our problem has an infinite number of variables and constraints. To form a computationally tractable problem, we instead use a finite set of samples drawn from the distribution. This is called sample average approximation (SAA).

```
D = Distributions.TriangularDist(150.0, 250.0, 200.0)
N = 100
d = sort!(rand(D, N));
Ω = 1:N
P = fill(1 / N, N);
StatsPlots.histogram(d; bins = 20, label = "", xlabel = "Demand")
```



#### JuMP model

The implementation of our two-stage stochastic program in JuMP is:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x >= 0)
@variable(model, 0 <= y[ω in Ω] <= d[ω])
@constraint(model, [ω in Ω], y[ω] <= x)
@expression(model, z[ω in Ω], 5y[ω] - 0.1 * (x - y[ω]))
@objective(model, Max, -2x + sum(P[ω] * z[ω] for ω in Ω))
optimize!(model)
assert_is_solved_and_feasible(model)
solution_summary(model)</pre>
```

```
solution_summary(; result = 1, verbose = false)
| solver_name : HiGHS
| Termination
| | termination_status : OPTIMAL
| | result_count : 1
| | raw_status : kHighsModelStatusOptimal
| L objective_bound : 5.64271e+02
| Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | dual_objective_value : 5.64271e+02
| L relative_gap : 0.00000e+00
L Work counters
| solve_time (sec) : 4.22955e-04
| simplex_iterations : 42
| barrier_iterations : 0
L node_count : -1
```

The optimal number of pies to make is:

value(x)

```
206.8364169803885
```

The distribution of total profit is:

total\_profit = [-2 \* value(x) + value(z[ $\omega$ ]) for  $\omega$  in  $\Omega$ ]

```
100-element Vector{Float64}:
399.9151768229468
405.7454219177082
417.61135488562354
420.5047206624374
424.1692725508499
426.7062730398078
434.2880791737741
434.96479364495735
 444.5071682942105
449.5828663580197
 П
620.5092509411653
620.5092509411653
620.5092509411653
620.5092509411653
620.5092509411653
 620.5092509411653
620.5092509411653
```

620.5092509411653 620.5092509411653

Let's plot it:

```
....
 bin_distribution(x::Vector{Float64}, N::Int)
A helper function that discretizes `x` into bins of width `N`.
.....
bin_distribution(x, N) = N * (floor(minimum(x) / N):ceil(maximum(x) / N))
plot = StatsPlots.histogram(
 total_profit;
 bins = bin_distribution(total_profit, 25),
 label = "",
 xlabel = "Profit [\$]",
 ylabel = "Number of outcomes",
)
\mu = Statistics.mean(total_profit)
Plots.vline!(
 plot,
 [μ];
 label = "Expected profit (\$$(round(Int, µ)))",
 linewidth = 3,
)
plot
```



#### **Risk measures**

A risk measure is a function which maps a random variable to a real number. Common risk measures include the mean (expectation), median, mode, and maximum. We need a risk measure to convert the distribution of second stage costs into a single number that can be optimized.

Our model currently uses the expectation risk measure, but others are possible too. One popular risk measure is the conditional value at risk (CVaR).

CVaR has a parameter  $\gamma$ , and it computes the expectation of the worst  $\gamma$  fraction of outcomes.

If we are maximizing, so that small outcomes are bad, the definition of CVaR is:

$$CVaR_{\gamma}[Z] = \max_{\xi} \ \xi - \frac{1}{\gamma} \mathbb{E}_{\omega} \left[ (\xi - Z)_{+} \right]$$

which can be formulated as the linear program:

$$CVaR_{\gamma}[Z] = \max_{\xi, z_{\omega}} \xi - \frac{1}{\gamma} \sum P_{\omega} z_{\omega}$$
$$z_{\omega} \ge \xi - Z_{\omega} \qquad \forall \omega$$
$$z_{\omega} \ge 0 \qquad \forall \omega.$$

```
function CVaR(Z::Vector{Float64}, P::Vector{Float64}; Y::Float64)
@assert 0 < Y <= 1
N = length(Z)
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, ξ)
@variable(model, z[1:N] >= 0)
@constraint(model, [i in 1:N], z[i] >= ξ - Z[i])
@objective(model, Max, ξ - 1 / Y * sum(P[i] * z[i] for i in 1:N))
optimize!(model)
assert_is_solved_and_feasible(model)
return objective_value(model)
end
```

CVaR (generic function with 1 method)

When  $\gamma$  is 1.0, we compute the mean of the profit:

cvar\_10 = CVaR(total\_profit, P;  $\gamma = 1.0$ )

564.2707958834865

Statistics.mean(total\_profit)

564.2707958834864

As  $\gamma$  approaches 0.0, we compute the worst-case (minimum) profit:

```
cvar_00 = CVaR(total_profit, P; \gamma = 0.0001)
```

399.9151768229468

minimum(total\_profit)

399.9151768229468

By varying  $\gamma$  between 0 and 1 we can compute some trade-off of these two extremes:

cvar\_05 = CVaR(total\_profit, P;  $\gamma$  = 0.5)

511.14442019068

Let's plot these outcomes on our distribution:

```
plot = StatsPlots.histogram(
 total_profit;
 bins = bin_distribution(total_profit, 25),
 label = "",
 xlabel = "Profit [\$]",
 ylabel = "Number of outcomes",
)
Plots.vline!(
 plot,
 [cvar_10 cvar_05 cvar_00];
 label = ["Y = 1.0" "Y = 0.5" "Y = 0.0"],
 linewidth = 3,
)
plot
```



#### Risk averse sample average approximation

Because CVaR can be formulated as a linear program, we can form a risk averse sample average approximation model by combining the two formulations:

```
\begin{split} &\gamma=0.4\\ &\text{model}=\text{Model}(\text{HiGHS.Optimizer})\\ &\text{set_silent}(\text{model})\\ &\text{@variable}(\text{model}, x >= 0)\\ &\text{@variable}(\text{model}, 0 <= y[\omega \text{ in }\Omega] <= d[\omega])\\ &\text{@constraint}(\text{model}, [\omega \text{ in }\Omega], y[\omega] <= x)\\ &\text{@expression}(\text{model}, Z[\omega \text{ in }\Omega], 5 * y[\omega] - 0.1(x - y[\omega]))\\ &\text{@variable}(\text{model}, \xi)\\ &\text{@variable}(\text{model}, z[\omega \text{ in }\Omega] >= 0)\\ &\text{@constraint}(\text{model}, [\omega \text{ in }\Omega], z[\omega] >= \xi - Z[\omega])\\ &\text{@objective}(\text{model}, Max, -2x + \xi - 1 / \gamma * sum(P[\omega] * z[\omega] \text{ for } \omega \text{ in }\Omega))\\ &\text{optimize!}(\text{model})\\ &\text{assert_is_solved_and_feasible}(\text{model}) \end{split}
```

When  $\gamma=0.4$ , the optimal number of pies to bake is:

value(x)

187.3712615331823

The distribution of total profit is:

```
risk_averse_total_profit = [value(-2x + Z[ω]) for ω in Ω]
bins = bin_distribution([total_profit; risk_averse_total_profit], 25)
plot = StatsPlots.histogram(total_profit; label = "Expectation", bins = bins)
StatsPlots.histogram!(
 plot,
 risk_averse_total_profit;
 label = "CV@R",
 bins = bins,
 alpha = 0.5,
)
plot
```



## Next steps

- Try solving this problem for different numbers of samples and different distributions.
- Refactor the example to avoid hard-coding the costs. What happens to the solution if the cost of disposing unsold pies increases?
- Plot the optimal number of pies to make for different values of the risk aversion parameter  $\gamma.$  What is the relationship?

Part III

Manual

# Chapter 11

# Models

JuMP models are the fundamental building block that we use to construct optimization problems. They hold things like the variables and constraints, as well as which solver to use and even solution information.

## Info

JuMP uses "optimizer" as a synonym for "solver." Our convention is to use "solver" to refer to the underlying software, and use "optimizer" to refer to the Julia object that wraps the solver. For example, HiGHS is a solver, and HiGHS.Optimizer is an optimizer.

Tip

See Supported solvers for a list of available solvers.

## 11.1 Create a model

Create a model by passing an optimizer to Model:

```
julia> model = Model(HiGHS.Optimizer)
A JuMP Model
b solver: HiGHS
b objective_sense: FEASIBILITY_SENSE
num_variables: 0
num_constraints: 0
L Names registered in the model: none
```

If you don't know which optimizer you will be using at creation time, create a model without an optimizer, and then call set\_optimizer at any time prior to optimize!:

```
julia> model = Model()
A JuMP Model
| solver: none
| objective_sense: FEASIBILITY_SENSE
| num_variables: 0
| num_constraints: 0
L Names registered in the model: none
```

julia> set\_optimizer(model, HiGHS.Optimizer)

#### Tip

Don't know what the fields Model mode and CachingOptimizer state mean? Read the Backends section.

## What is the difference?

For most models, there is no difference between passing the optimizer to Model, and calling set\_optimizer.

However, if an optimizer does not support a constraint in the model, the timing of when an error will be thrown can differ:

- If you pass an optimizer, an error will be thrown when you try to add the constraint.
- If you call set\_optimizer, an error will be thrown when you try to solve the model via optimize!.

Therefore, most users should pass an optimizer to Model because it provides the earliest warning that your solver is not suitable for the model you are trying to build. However, if you are modifying a problem by adding and deleting different constraint types, you may need to use set\_optimizer. See Switching optimizer for the relaxed problem for an example of when this is useful.

#### Solvers which expect environments

Some solvers accept (or require) positional arguments such as a license environment or a path to a binary executable. For these solvers, you can pass a function to Model which takes zero arguments and returns an instance of the optimizer.

A common use-case for this is passing an environment or sub-solver to the optimizer:

```
julia> import HiGHS
julia> import MultiObjectiveAlgorithms as MOA
julia> model = Model(() -> MOA.Optimizer(HiGHS.Optimizer))
A JuMP Model
+ solver: MOA[algorithm=MultiObjectiveAlgorithms.Lexicographic, optimizer=HiGHS]
+ objective_sense: FEASIBILITY_SENSE
+ num_variables: 0
+ num_constraints: 0
- Names registered in the model: none
```

## 11.2 Solver options

JuMP uses "attribute" as a synonym for "option." Use optimizer\_with\_attributes to create an optimizer with some attributes initialized:

Alternatively, use set\_attribute to set an attribute after the model has been created:

```
julia> model = Model(HiGHS.Optimizer);
julia> set_attribute(model, "output_flag", false)
julia> get_attribute(model, "output_flag")
false
```

You can also modify attributes within an optimizer\_with\_attributes object:

```
julia> solver = optimizer_with_attributes(HiGHS.Optimizer, "output_flag" => true);
julia> get_attribute(solver, "output_flag")
true
julia> set_attribute(solver, "output_flag", false)
julia> get_attribute(solver, "output_flag")
false
julia> model = Model(solver);
```

## 11.3 Changing the number types

By default, the coefficients of affine and quadratic expressions are numbers of type either Float64 or Complex{Float64} (see Complex number support).

The type Float64 can be changed using the GenericModel constructor:

```
julia> model = GenericModel{Rational{BigInt}}();
julia> @variable(model, x)
x
julia> @expression(model, expr, 1 // 3 * x)
1//3 x
julia> typeof(expr)
GenericAffExpr{Rational{BigInt}, GenericVariableRef{Rational{BigInt}}}
```

555

## CHAPTER 11. MODELS

Using a value\_type other than Float64 is an advanced operation and should be used only if the underlying solver actually solves the problem using the provided value type.

Warning

Nonlinear Modeling is currently restricted to the Float64 number type.

## 11.4 Print the model

By default, show(model) will print a summary of the problem:

```
julia> model = Model(); @variable(model, x >= 0); @objective(model, Max, x);
```

#### julia> model

```
A JuMP Model
 solver: none
 objective_sense: MAX_SENSE
 L objective_function_type: VariableRef
 num_variables: 1
 num_constraints: 1
 L VariableRef in MOI.GreaterThan{Float64}: 1
 Names registered in the model
 L :x
```

Use print to print the formulation of the model (in IJulia, this will render as LaTeX.

## julia> print(model) Max x Subject to $x \ge 0$

#### Warning

This format is specific to JuMP and may change in any future release. It is not intended to be an instance format. To write the model to a file, use write\_to\_file instead.

Use latex\_formulation to display the model in LaTeX form.

```
julia> latex_formulation(model)
$$ \begin{aligned}
\max\quad & x\\
\text{Subject to} \quad & x \geq 0\\
\end{aligned} $$
```

In IJulia (and Documenter), ending a cell in with latex\_formulation will render the model in LaTeX:

latex\_formulation(model)

 $\begin{array}{ll} \max & x \\ \text{Subject to} & x \geq 0 \end{array}$ 

### 11.5 Turn off output

Use set\_silent and unset\_silent to disable or enable printing output from the solver.

```
julia> model = Model(HiGHS.Optimizer);
```

julia> set\_silent(model)

julia> unset\_silent(model)

## Тір

Most solvers will also have a solver-specific option to provide finer-grained control over the output. Consult their README's for details.

## 11.6 Set a time limit

Use set\_time\_limit\_sec, unset\_time\_limit\_sec, and time\_limit\_sec to manage time limits.

```
julia> model = Model(HiGHS.Optimizer);
julia> set_time_limit_sec(model, 60.0)
julia> time_limit_sec(model)
60.0
julia> unset_time_limit_sec(model)
julia> limit = time_limit_sec(model)
julia> limit === nothing
true
```

If your time limit is encoded as a Dates.Period object, use the following code to convert it to Float64 for set\_time\_limit\_sec:

```
julia> import Dates
julia> seconds(x::Dates.Period) = 1e-3 * Dates.value(round(x, Dates.Millisecond))
seconds (generic function with 1 method)
```

julia> set\_time\_limit\_sec(model, seconds(Dates.Hour(1)))

```
julia> time_limit_sec(model)
3600.0
```

#### Info

Some solvers do not support time limits. In these cases, an error will be thrown.

## 11.7 Write a model to file

JuMP can write models to a variety of file-formats using write\_to\_file and Base.write.

For most common file formats, the file type will be detected from the extension.

For example, here is how to write an MPS file:

```
julia> model = Model();
```

julia> write\_to\_file(model, "model.mps")

#### Other supported file formats include:

- .cbf for the Conic Benchmark Format
- . lp for the LP file format
- .mof.json for the MathOptFormat
- .nl for AMPL's NL file format
- . rew for the REW file format
- .sdpa and ".dat-s" for the SDPA file format

To write to a specific io:::I0, use Base.write. Specify the file type by passing a MOI.FileFormats.FileFormat enum.

```
julia> model = Model();
```

```
julia> io = IOBuffer();
```

julia> write(io, model; format = MOI.FileFormats.FORMAT\_MPS)

## 11.8 Read a model from file

JuMP models can be created from file formats using read\_from\_file and Base.read.

```
julia> model = read_from_file("model.mps")
A JuMP Model
+ solver: none
+ objective_sense: MIN_SENSE
| L objective_function_type: AffExpr
+ num_variables: 0
+ num_constraints: 0
L Names registered in the model: none
```

#### julia> seekstart(io);

julia> model2 = read(io, Model; format = MOI.FileFormats.FORMAT\_MPS)
A JuMP Model
+ solver: none
+ objective\_sense: MIN\_SENSE
| L objective\_function\_type: AffExpr
+ num\_variables: 0
+ num\_constraints: 0
L Names registered in the model: none

## Note

Because file formats do not serialize the containers of JuMP variables and constraints, the names in the model will *not* be registered. Therefore, you cannot access named variables and constraints via model[:x]. Instead, use variable\_by\_name or constraint\_by\_name to access specific variables or constraints.

## **Nonlinear file formats**

To maintain backwards compatibility, nonlinear models in .mof.json and .nl files are parsed into a MOI.NLPBlock. To parse as MOI.ScalarNonlinearFunction, pass the keyword use\_nlp\_block = false:

```
julia> model = Model();
julia> @variable(model, x >= 0);
julia> @objective(model, Min, log(x));
julia> filename = joinpath(mktempdir(), "model.mof.json");
julia> write_to_file(model, filename)
julia> new_model = read_from_file(filename; use_nlp_block = false)
A JuMP Model
+ solver: none
+ objective_sense: MIN_SENSE
| L objective_function_type: NonlinearExpr
+ num_variables: 1
+ num_constraints: 1
| L VariableRef in MOI.GreaterThan{Float64}: 1
- Names registered in the model: none
```

julia> print(new\_model) Min log(x) Subject to  $x \ge 0$  CHAPTER 11. MODELS

## 11.9 Relax integrality

Use relax\_integrality to remove any integrality constraints from the model, such as integer and binary restrictions on variables. relax\_integrality returns a function that can be later called with zero arguments to re-add the removed constraints:

```
julia> model = Model();
julia> @variable(model, x, Int)
x
julia> num_constraints(model, VariableRef, MOI.Integer)
1
julia> undo = relax_integrality(model);
julia> num_constraints(model, VariableRef, MOI.Integer)
0
julia> undo()
julia> num_constraints(model, VariableRef, MOI.Integer)
1
```

#### Switching optimizer for the relaxed problem

A common reason for relaxing integrality is to compute dual variables of the relaxed problem. However, some mixed-integer linear solvers (for example, Cbc) do not return dual solutions, even if the problem does not have integrality restrictions.

Therefore, after relax\_integrality you should call set\_optimizer with a solver that does support dual solutions, such as Clp.

For example, instead of:

```
using JuMP, Cbc
model = Model(Cbc.Optimizer)
@variable(model, x, Int)
undo = relax_integrality(model)
optimize!(model)
reduced_cost(x) # Errors
```

do:

```
using JuMP, Cbc, Clp
model = Model(Cbc.Optimizer)
@variable(model, x, Int)
undo = relax_integrality(model)
set_optimizer(model, Clp.Optimizer)
optimize!(model)
reduced_cost(x) # Works
```

## 11.10 Get the matrix representation

Use lp\_matrix\_data to return a data structure that represents the matrix form of a linear program.

```
julia> begin
 model = Model()
 @variable(model, x >= 1, Bin)
 @variable(model, 2 <= y)
 @variable(model, 3 <= z <= 4, Int)
 @constraint(model, x == 5)
 @constraint(model, 2x + 3y <= 6)
 @constraint(model, -4y >= 5z + 7)
 @constraint(model, -1 <= x + y <= 2)
 @objective(model, Max, 1 + 2x)
end;</pre>
```

julia> data = lp\_matrix\_data(model);

#### julia> data.A

 $4\times3$  SparseArrays.SparseMatrixCSC{Float64, Int64} with 7 stored entries:

- 1.0 · · · -4.0 -5.0 2.0 3.0 ·
- 1.0 1.0 .

#### julia> data.b\_lower

4-element Vector{Float64}:

5.0 7.0 -Inf -1.0

#### julia> data.b\_upper

4-element Vector{Float64}: 5.0 Inf 6.0 2.0

#### julia> data.x\_lower

3-element Vector{Float64}: 1.0 2.0 3.0

## julia> data.x\_upper

3-element Vector{Float64}: Inf 4.0

#### julia> data.c

3-element Vector{Float64}: 2.0 0.0 0.0

```
julia> data.c_offset
1.0
```

julia> data.sense
MAX SENSE::OptimizationSense = 1

#### julia> data.integers

```
1-element Vector{Int64}:
3
```

# julia> data.binaries 1-element Vector{Int64}:

1

#### Warning

lp\_matrix\_data is intentionally limited in the types of problems that it supports and the structure of the matrices it outputs. It is mainly intended as a pedagogical and debugging tool. It should not be used to interface solvers, see Implementing a solver interface instead.

## 11.11 Backends

#### Info

This section discusses advanced features of JuMP. For new users, you may want to skip this section. You don't need to know how JuMP manages problems behind the scenes to create and solve JuMP models.

A JuMP Model is a thin layer around a *backend* of type MOI.ModelLike that stores the optimization problem and acts as the optimization solver.

However, if you construct a model like Model (HiGHS.Optimizer), the backend is not a HiGHS.Optimizer, but a more complicated object.

From JuMP, the MOI backend can be accessed using the backend function. Let's see what the backend of a JuMP Model is:

julia> model = Model(HiGHS.Optimizer);

#### julia> b = backend(model)

```
MOIU.CachingOptimizer
```

- state: EMPTY\_OPTIMIZER
- mode: AUTOMATIC
- h model\_cache: MOIU.UniversalFallback{MOIU.Model{Float64}}

- | | NumberOfVariables: 0
- | L NumberOfConstraints: 0
- L optimizer: MOIB.LazyBridgeOptimizer{HiGHS.Optimizer}
- + Variable bridges: none

```
+ Objective bridges: none
L model: A HiGHS model with 0 columns and 0 rows.
```

Uh oh. Even though we passed a HiGHS.Optimizer, the backend is a much more complicated object.

#### CachingOptimizer

A MOIU.CachingOptimizer is a layer that abstracts the difference between solvers that support incremental modification (for example, they support adding variables one-by-one), and solvers that require the entire problem in a single API call (for example, they only accept the A, b and c matrices of a linear program).

It has two parts:

1. A cache, where the model can be built and modified incrementally

2. An optimizer, which is used to solve the problem

#### julia> b.optimizer

MOIB.LazyBridgeOptimizer{HiGHS.Optimizer}

- Variable bridges: none
- Constraint bridges: none
- Objective bridges: none
- L model: A HiGHS model with 0 columns and 0 rows.

#### Info

The LazyBridgeOptimizer section explains what a LazyBridgeOptimizer is.

The CachingOptimizer has logic to decide when to copy the problem from the cache to the optimizer, and when it can efficiently update the optimizer in-place.

A CachingOptimizer may be in one of three possible states:

- N0\_0PTIMIZER: The CachingOptimizer does not have any optimizer.
- EMPTY\_OPTIMIZER: The CachingOptimizer has an empty optimizer, and it is not synchronized with the cached model.
- ATTACHED\_OPTIMIZER: The CachingOptimizer has an optimizer, and it is synchronized with the cached model.

A CachingOptimizer has two modes of operation:

- AUTOMATIC: The CachingOptimizer changes its state when necessary. For example, optimize! will
  automatically call attach\_optimizer (an optimizer must have been previously set). Attempting to add a
  constraint or perform a modification not supported by the optimizer results in a drop to EMPTY\_OPTIMIZER
  mode.
- MANUAL: The user must change the state of the CachingOptimizer using MOIU.reset\_optimizer(::JuMP.Model), MOIU.drop\_optimizer(::JuMP.Model), and MOIU.attach\_optimizer(::JuMP.Model). Attempting to perform an operation in the incorrect state results in an error.

By default Model will create a CachingOptimizer in AUTOMATIC mode.

#### LazyBridgeOptimizer

The second layer that JuMP applies automatically is a MOI.Bridges.LazyBridgeOptimizer. A MOI.Bridges.LazyBridgeOptimizer is an MOI layer that attempts to transform the problem from the formulation provided by the user into an equivalent problem supported by the solver. This may involve adding new variables and constraints to the optimizer. The transformations are selected from a set of known recipes called *bridges*.

A common example of a bridge is one that splits an interval constraint like @constraint(model,  $1 \le x + y \le 2$ ) into two constraints, @constraint(model,  $x + y \ge 1$ ) and @constraint(model,  $x + y \le 2$ ).

Use the add\_bridges = false keyword to remove the bridging layer:

```
julia> model = Model(HiGHS.Optimizer; add_bridges = false)
A JuMP Model
- solver: HiGHS
bjective_sense: FEASIBILITY_SENSE
- num_variables: 0
- num_constraints: 0
^L Names registered in the model: none
julia> backend(model)
MOIU.CachingOptimizer
- state: EMPTY OPTIMIZER
- mode: AUTOMATIC
+ model cache: MOIU.UniversalFallback{MOIU.Model{Float64}}
| - ObjectiveSense: FEASIBILITY SENSE
- NumberOfVariables: 0
 L NumberOfConstraints: 0
```

<sup>L</sup> optimizer: A HiGHS model with 0 columns and 0 rows.

Bridges can be added and removed from a MOI.Bridges.LazyBridgeOptimizer using add\_bridge and remove\_bridge. Use print\_active\_bridges to see which bridges are used to reformulate the model. Read the Example: ellipsoid approximation tutorial for more details.

#### **Unsafe backend**

In some advanced use-cases, it is necessary to work with the inner optimization model directly. To access this model, use unsafe backend:

julia> backend(model)
MOIU.CachingOptimizer
 state: EMPTY\_OPTIMIZER

## CHAPTER 11. MODELS

- mode: AUTOMATIC

- head = model\_cache: MOIU.UniversalFallback{MOIU.Model{Float64}}
- | | ObjectiveSense: FEASIBILITY\_SENSE
- | ObjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
- | | NumberOfVariables: 0
- L NumberOfConstraints: 0
- L optimizer: MOIB.LazyBridgeOptimizer{HiGHS.Optimizer}
- Variable bridges: none
- Constraint bridges: none
- $^{\rm L}$  model: A HiGHS model with 0 columns and 0 rows.

#### julia> unsafe\_backend(model)

```
A HiGHS model with 0 columns and 0 rows.
```

#### Warning

backend and unsafe\_backend are advanced routines. Read their docstrings to understand the caveats of their usage, and only call them if you wish to access low-level solver-specific functions.

## 11.12 Direct mode

Using a CachingOptimizer results in an additional copy of the model being stored by JuMP in the .model\_cache field. To avoid this overhead, create a JuMP model using direct\_model:

```
julia> model = direct_model(HiGHS.Optimizer())
A JuMP Model
+ mode: DIRECT
+ solver: HiGHS
+ objective_sense: FEASIBILITY_SENSE
+ num_variables: 0
+ num_constraints: 0
L Names registered in the model: none
```

#### Warning

Solvers that do not support incremental modification do not support direct\_model. An error will be thrown, telling you to use a CachingOptimizer instead.

The benefit of using direct\_model is that there are no extra layers (for example, Cachingoptimizer or LazyBridgeOptimizer) between model and the provided optimizer:

```
julia> backend(model)
A HiGHS model with 0 columns and 0 rows.
```

A downside of direct mode is that there is no bridging layer. Therefore, only constraints which are natively supported by the solver are supported. For example, HiGHS.jl does not implement quadratic constraints:

```
julia> model = direct_model(HiGHS.Optimizer());
```

julia> set\_silent(model)

julia> @variable(model, x[1:2]);

julia> @constraint(model, x[1]^2 + x[2]^2 <= 2)
ERROR: Constraints of type</pre>

→ MathOptInterface.ScalarQuadraticFunction{Float64}-in-MathOptInterface.LessThan{Float64} are not
 → supported by the solver.

If you expected the solver to support your problem, you may have an error in your formulation.  $\hookrightarrow$  Otherwise, consider using a different solver.

```
The list of available solvers, along with the problem types they support, is available at

→ https://jump.dev/JuMP.jl/stable/installation/#Supported-solvers.

Stacktrace:
```

#### Warning

Another downside of direct mode is that the behavior of querying solution information after modifying the problem is solver-specific. This can lead to errors, or the solver silently returning an incorrect value. See OptimizeNotCalled errors for more information.

## **Chapter 12**

# Variables

The term *variable* in mathematical optimization has many meanings. For example, *optimization* variables (also called decision variables) are the unknowns x that we are solving for in the problem:

$\min_{x \in \mathbb{R}^n}$	$f_0(x)$		(12.1)
s.t.	$f_i(x) \in \mathcal{S}_i$	$i = 1 \dots m$	(12.2)

To complicate things, Julia uses *variable* to mean a binding between a name and a value. For example, in the statement:

**julia>** x = 1 1

1

x is a variable that stores the value 1.

JuMP uses *variable* in a third way, to mean an instance of the VariableRef struct. JuMP variables are the link between Julia and the optimization variables inside a JuMP model.

This page explains how to create and manage JuMP variables in a variety of contexts.

## 12.1 Create a variable

Create variables using the @variable macro:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> typeof(x)
VariableRef (alias for GenericVariableRef{Float64})
julia> num_variables(model)
```

Here x is a Julia variable that is bound to a VariableRef object, and we have added 1 decision variable to our model.

To make the binding more explicit, we could have written:

```
julia> model = Model();
julia> x = @variable(model, x)
x
```

but there is no need to in general; the macro does it for us.

When creating a variable, you can also specify variable bounds:

```
julia> model = Model();
```

julia> @variable(model, x\_free)
x\_free

julia> @variable(model, x\_lower >= 0)
x\_lower

julia> @variable(model, x\_upper <= 1)
x\_upper</pre>

julia> @variable(model, 2 <= x\_interval <= 3)
x\_interval</pre>

julia> @variable(model, x\_fixed == 4)
x\_fixed

#### julia> print(model)

Feasibility
Subject to
x\_fixed = 4
x\_lower ≥ 0
x\_interval ≥ 2
x\_upper ≤ 1
x\_interval ≤ 3

#### Warning

When creating a variable with a single lower- or upper-bound, and the value of the bound is not a numeric literal (for example, 1 or 1.0), the name of the variable *must* appear on the left-hand side. Putting the name on the right-hand side is an error. For example, to create a variable x:

```
a = 1
@variable(model, x >= 1) # [Okay
@variable(model, 1.0 <= x) # [Okay
@variable(model, x >= a) # [Okay
@variable(model, a <= x) # × Not okay
@variable(model, x >= 1 / 2) # [Okay
@variable(model, 1 / 2 <= x) # × Not okay</pre>
```
CHAPTER 12. VARIABLES

## **Containers of variables**

The @variable macro also supports creating collections of JuMP variables. We'll cover some brief syntax here; read the Variable containers section for more details.

You can create arrays of JuMP variables:

```
julia> model = Model();
julia> @variable(model, x[1:2, 1:2])
2×2 Matrix{VariableRef}:
 x[1,1] x[1,2]
 x[2,1] x[2,2]
julia> x[1, 2]
 x[1,2]
```

Index sets can be named, and bounds can depend on those names:

Sets can be any Julia type that supports iteration:

```
julia> model = Model();
```

```
julia> @variable(model, x[i = 2:3, j = 1:2:3, ["red", "blue"]] >= 0)
3-dimensional DenseAxisArray{VariableRef,3,...} with index sets:
 Dimension 1, 2:3
 Dimension 2, 1:2:3
 Dimension 3, ["red", "blue"]
And data, a 2×2×2 Array{VariableRef, 3}:
[:, :, "red"] =
 x[2,1,red] x[2,3,red]
 x[3,1,red] x[2,3,blue]
[:, :, "blue"] =
 x[2,1,blue] x[2,3,blue]
julia> x[2, 1, "red"]
 x[2,1,red]
```

Sets can depend upon previous indices:

```
julia> model = Model();
```

```
julia> @variable(model, u[i = 1:2, j = i:3])
JuMP.Containers.SparseAxisArray{VariableRef, 2, Tuple{Int64, Int64}} with 5 entries:
 [1, 1] = u[1,1]
 [1, 2] = u[1,2]
 [1, 3] = u[1,3]
 [2, 2] = u[2,2]
 [2, 3] = u[2,3]
```

and we can filter elements in the sets using the ; syntax:

```
julia> model = Model();
```

```
julia> @variable(model, v[i = 1:9; mod(i, 3) == 0])
JuMP.Containers.SparseAxisArray{VariableRef, 1, Tuple{Int64}} with 3 entries:
[3] = v[3]
[6] = v[6]
[9] = v[9]
```

# 12.2 Registered variables

When you create variables, JuMP registers them inside the model using their corresponding symbol. Get a registered name using model[:key]:

```
julia> model = Model()
A JuMP Model
- solver: none
bjective_sense: FEASIBILITY_SENSE
- num variables: 0
- num constraints: 0
L Names registered in the model: none
julia> @variable(model, x)
Х
julia> model
A JuMP Model
- solver: none
bjective_sense: FEASIBILITY_SENSE
- num variables: 1
- num_constraints: 0
^L Names registered in the model
 L :X
julia> model[:x] === x
true
```

Registered names are most useful when you start to write larger models and want to break up the model construction into functions:

```
julia> function set_objective(model::Model)
 @objective(model, Min, 2 * model[:my_x] + 1)
 return
 end
set_objective (generic function with 1 method)
julia> model = Model();
julia> @variable(model, my_x);
julia> set_objective(model)
julia> print(model)
Min 2 my_x + 1
Subject to
```

## 12.3 Anonymous variables

To reduce the likelihood of accidental bugs, and because JuMP registers variables inside a model, creating two variables with the same name is an error:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, x)
ERROR: An object of name x is already attached to this model. If this
 is intended, consider using the anonymous construction syntax, for example,
 `x = @variable(model, [1:N], ...)` where the name of the object does
 not appear inside the macro.
 Alternatively, use `unregister(model, :x)` to first unregister
 the existing name from the model. Note that this will not delete the
 object; it will just remove the reference at `model[:x]`.
[...]
```

A common reason for encountering this error is adding variables in a loop.

As a work-around, JuMP provides *anonymous* variables. Create a scalar valued anonymous variable by omitting the name argument:

```
julia> model = Model();
julia> x = @variable(model)
_[1]
```

Anonymous variables get printed as an underscore followed by a unique index of the variable.

## CHAPTER 12. VARIABLES

#### Warning

The index of the variable may not correspond to the column of the variable in the solver.

Create a container of anonymous JuMP variables by dropping the name in front of the [:

```
julia> model = Model();
julia> y = @variable(model, [1:2])
2-element Vector{VariableRef}:
_[1]
_[2]
```

The <= and >= short-hand cannot be used to set bounds on scalar-valued anonymous JuMP variables. Instead, use the lower\_bound and upper\_bound keywords:

```
julia> model = Model();
julia> x_lower = @variable(model, lower_bound = 1.0)
_[1]
julia> x_upper = @variable(model, upper_bound = 2.0)
_[2]
julia> x_interval = @variable(model, lower_bound = 3.0, upper_bound = 4.0)
_[3]
```

## 12.4 Variable names

In addition to the symbol that variables are registered with, JuMP variables have a String name that is used for printing and writing to file formats.

Get and set the name of a variable using name and set\_name:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> name(x)
"x"
julia> set_name(x, "my_x_name")
julia> x
my_x_name
```

Override the default choice of name using the base\_name keyword:

```
julia> model = Model();
```

```
julia> @variable(model, x[i=1:2], base_name = "my_var")
2-element Vector{VariableRef}:
 my_var[1]
 my_var[2]
```

Note that names apply to each element of the container, not to the container of variables:

```
julia> name(x[1])
"my_var[1]"
```

julia> set\_name(x[1], "my\_x")

**julia>** ×

```
2-element Vector{VariableRef}:
 my_x
 my_var[2]
```

# Тір

For some models, setting the string name of each variable can take a non-trivial portion of the total time required to build the model. Turn off String names by passing set\_string\_name = false to @variable:

julia> model = Model();

```
julia> @variable(model, x, set_string_name = false)
_[1]
```

See Disable string names for more information.

#### Retrieve a variable by name

Retrieve a variable from a model using variable\_by\_name:

```
julia> variable_by_name(model, "my_x")
my_x
```

If the name is not present, nothing will be returned:

julia> variable\_by\_name(model, "bad\_name")

You can only look up individual variables using variable\_by\_name. Something like this will not work:

julia> model = Model();

```
julia> @variable(model, [i = 1:2], base_name = "my_var")
2-element Vector{VariableRef}:
 my_var[1]
 my_var[2]
julia> variable_by_name(model, "my_var")
```

To look up a collection of variables, do not use variable\_by\_name. Instead, register them using the model[:key] = value syntax:

```
julia> model = Model();
julia> model[:x] = @variable(model, [i = 1:2], base_name = "my_var")
2-element Vector{VariableRef}:
 my_var[1]
 my_var[2]
julia> model[:x]
2-element Vector{VariableRef}:
 my_var[1]
 my_var[2]
```

## 12.5 String names, symbolic names, and bindings

It's common for new users to experience confusion relating to JuMP variables. Part of the problem is the overloaded use of "variable" in mathematical optimization, along with the difference between the name that a variable is registered under and the String name used for printing.

Here's a summary of the differences:

- JuMP variables are created using @variable.
- JuMP variables can be named or anonymous.
- Named JuMP variables have the form @variable(model, x). For named variables:
  - The String name of the variable is set to "x".
  - A Julia variable x is created that binds x to the JuMP variable.
  - The name : x is registered as a key in the model with the value x.
- Anonymous JuMP variables have the form x = @variable(model). For anonymous variables:
  - The String name of the variable is set to "". When printed, this is replaced with "\_[i]" where i is the index of the variable.
  - You control the name of the Julia variable used as the binding.
  - No name is registered as a key in the model.
- The base\_name keyword can override the String name of the variable.
- You can manually register names in the model via model[:key] = value

```
Here's an example that should make things clearer:
```

```
julia> model = Model();
julia> x_binding = @variable(model, base_name = "x")
Х
julia> model
A JuMP Model
- solver: none
bjective_sense: FEASIBILITY_SENSE
- num_variables: 1
hum_constraints: 0
^{\rm L} Names registered in the model: none
julia> x
ERROR: UndefVarError: `x` not defined
julia> x_binding
Х
julia> name(x_binding)
"X"
julia> model[:x_register] = x_binding
Х
julia> model
A JuMP Model
- solver: none
bjective_sense: FEASIBILITY_SENSE
- num_variables: 1
- num_constraints: 0
^{\rm L} Names registered in the model
 L :x_register
julia> model[:x_register]
julia> model[:x_register] === x_binding
true
julia> x
ERROR: UndefVarError: `x` not defined
```

## 12.6 Create, delete, and modify variable bounds

Query whether a variable has a bound using has\_lower\_bound, has\_upper\_bound, and is\_fixed:

```
julia> has_lower_bound(x_free)
false
julia> has_upper_bound(x_upper)
true
```

```
julia> is_fixed(x_fixed)
true
```

If a variable has a particular bound, query the value of it using lower\_bound, upper\_bound, and fix\_value:

```
julia> lower_bound(x_interval)
2.0
julia> upper_bound(x_interval)
3.0
julia> fix_value(x_fixed)
4.0
```

Querying the value of a bound that does not exist will result in an error.

Delete variable bounds using delete\_lower\_bound, delete\_upper\_bound, and unfix:

```
julia> delete_lower_bound(x_lower)
julia> has_lower_bound(x_lower)
false
julia> delete_upper_bound(x_upper)
julia> has_upper_bound(x_upper)
false
julia> unfix(x_fixed)
julia> is_fixed(x_fixed)
```

false

Set or update variable bounds using set\_lower\_bound, set\_upper\_bound, and fix:

```
julia> set_lower_bound(x_lower, 1.1)
julia> set_upper_bound(x_upper, 2.1)
julia> fix(x_fixed, 4.1)
```

Fixing a variable with existing bounds will throw an error. To delete the bounds prior to fixing, use fix(variable, value; force = true).

```
julia> model = Model();
julia> @variable(model, x >= 1)
x
julia> fix(x, 2)
```

```
julia> fix_value(x)
2.0
```

## Тір

```
Use fix instead of @constraint(model, x == 2). The former modifies variable bounds, while the latter adds a new linear constraint to the problem.
```

#### 12.7 Binary variables

Binary variables are constrained to the set  $x \in \{0, 1\}$ .

Create a binary variable by passing Bin as an optional positional argument:

```
julia> model = Model();
```

```
julia> @variable(model, x, Bin)
```

#### Warning

Solvers use tolerances to decide whether a variable satisfies the binary constraint. Thus, the true feasible region is  $[-\varepsilon, \varepsilon] \cup [1-\varepsilon, 1+\varepsilon]$ , where  $\varepsilon$  is solver-specific, but typically 1e-6. As a result, you should expect the value(x) of a Bin variable to sometimes take a value like -0.0, 1e-8, or 0.9999999.

Check if a variable is binary using is\_binary:

```
julia> is_binary(x)
true
```

Delete a binary constraint using unset\_binary:

```
julia> unset_binary(x)
```

```
julia> is_binary(x)
false
```

Binary variables can also be created by setting the binary keyword to true:

julia> model = Model();

julia> @variable(model, x, binary=true)
x

or by using set\_binary:

julia> model = Model();

```
julia> @variable(model, x)
x
```

julia> set\_binary(x)

## 12.8 Integer variables

Integer variables are constrained to the set  $x \in \mathbb{Z}$ .

Create an integer variable by passing Int as an optional positional argument:

```
julia> model = Model();
julia> @variable(model, x, Int)
```

#### Warning

Solvers use tolerances to decide whether a variable satisfies the integer constraint. Thus, the true feasible region is  $\bigcup_{z \in \mathbb{Z}} [z - \varepsilon, z + \varepsilon]$ , where  $\varepsilon$  is solver-specific, but typically 1e-6. As a result, you should expect the value(x) of an Int variable to sometimes take a value like 1e-8, or 2.999999.

Check if a variable is integer using is\_integer:

```
julia> is_integer(x)
true
```

Delete an integer constraint using unset\_integer.

```
julia> unset_integer(x)
```

```
julia> is_integer(x)
false
```

Integer variables can also be created by setting the integer keyword to true:

```
julia> model = Model();
julia> @variable(model, x, integer=true)
x
```

```
or by using set_integer:
```

julia> model = Model();

julia> @variable(model, x)

Х

julia> set\_integer(x)

# Тір

The relax\_integrality function relaxes all integrality constraints in the model, returning a function that can be called to undo the operation later on.

#### 12.9 Semi-integer and semi-continuous variables

Semi-continuous variables are constrained to the set  $x \in \{0\} \cup [l, u]$ .

Create a semi-continuous variable using the Semicontinuous set:

```
julia> model = Model();
```

```
julia> @variable(model, x in Semicontinuous(1.5, 3.5))
x
```

Semi-integer variables are constrained to the set  $x \in \{0\} \cup \{l, l+1, \dots, u\}$ .

Create a semi-integer variable using the Semiinteger set:

```
julia> model = Model();
julia> @variable(model, x in Semiinteger(1.0, 3.0))
x
```

# 12.10 Start values

There are two ways to provide a primal starting solution (also called MIP-start or a warmstart) for each variable:

- using the start keyword in the @variable macro
- using set\_start\_value

The starting value of a variable can be queried using start\_value. If no start value has been set, start\_value will return nothing.

```
julia> model = Model();
julia> @variable(model, x)
x
```

```
julia> start_value(x)
```

```
julia> @variable(model, y, start = 1)
y
julia> start_value(y)
1.0
julia> set_start_value(y, 2)
julia> start_value(y)
2.0
```

The start keyword argument can depend on the indices of a variable container:

```
julia> model = Model();
```

```
julia> @variable(model, z[i = 1:2], start = i^2)
2-element Vector{VariableRef}:
 z[1]
 z[2]
julia> start_value.(z)
2-element Vector{Float64}:
 1.0
 4.0
```

#### Warning

Some solvers do not support start values. If a solver does not support start values, an MathOptInterface.UnsupportedAttribute{MathOptInterface.VariablePrimalStart} error will be thrown.

## Тір

To set the optimal solution from a previous solve as a new starting value, use all\_variables to get a vector of all the variables in the model, then run:

x = all\_variables(model)
x\_solution = value.(x)
set\_start\_value.(x, x\_solution)

Alternatively, use set\_start\_values.

## 12.11 Delete a variable

Use delete to delete a variable from a model. Use is\_valid to check if a variable belongs to a model and has not been deleted.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> is_valid(model, x)
true
julia> delete(model, x)
julia> is_valid(model, x)
false
```

Deleting a variable does not unregister the corresponding name from the model. Therefore, creating a new variable of the same name will throw an error:

```
julia> @variable(model, x)
ERROR: An object of name x is already attached to this model. If this
 is intended, consider using the anonymous construction syntax, for example,
 `x = @variable(model, [1:N], ...)` where the name of the object does
 not appear inside the macro.
 Alternatively, use `unregister(model, :x)` to first unregister
 the existing name from the model. Note that this will not delete the
 object; it will just remove the reference at `model[:x]`.
[...]
```

After calling delete, call unregister to remove the symbolic reference:

```
julia> unregister(model, :x)
julia> @variable(model, x)
x
```

## Info

delete does not automatically unregister because we do not distinguish between names that are automatically registered by JuMP macros and names that are manually registered by the user by setting values in object\_dictionary. In addition, deleting a variable and then adding a new variable of the same name is an easy way to introduce bugs into your code.

# 12.12 Variable containers

JuMP provides a mechanism for creating collections of variables in three types of data structures, which we refer to as *containers*.

The three types are Arrays, DenseAxisArrays, and SparseAxisArrays. We explain each of these in the following. Тір

You can read more about containers in the Containers section.

#### Arrays

We have already seen the creation of an array of JuMP variables with the x[1:2] syntax. This can be extended to create multi-dimensional arrays of JuMP variables. For example:

```
julia> model = Model();
julia> @variable(model, x[1:2, 1:2])
2×2 Matrix{VariableRef}:
 x[1,1] x[1,2]
 x[2,1] x[2,2]
```

Arrays of JuMP variables can be indexed and sliced as follows:

```
julia> x[1, 2]
x[1,2]
julia> x[2, :]
```

2-element Vector{VariableRef}:
 x[2,1]
 x[2,2]

Variable bounds can depend upon the indices:

```
julia> model = Model();
julia> @variable(model, x[i=1:2, j=1:2] >= 2i + j)
2×2 Matrix{VariableRef}:
 x[1,1] x[1,2]
 x[2,1] x[2,2]
julia> lower_bound.(x)
2×2 Matrix{Float64}:
 3.0 4.0
 5.0 6.0
```

JuMP will form an Array of JuMP variables when it can determine at compile time that the indices are one-based integer ranges. Therefore x[1:b] will create an Array of JuMP variables, but x[a:b] will not. If JuMP cannot determine that the indices are one-based integer ranges (for example, in the case of x[a:b]), JuMP will create a DenseAxisArray instead.

#### DenseAxisArrays

We often want to create arrays where the indices are not one-based integer ranges. For example, we may want to create a variable indexed by the name of a product or a location. The syntax is the same as that above, except with an arbitrary vector as an index as opposed to a one-based range. The biggest difference is that instead of returning an Array of JuMP variables, JuMP will return a DenseAxisArray. For example:

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2, [:A,:B]])
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
 Dimension 1, Base.OneTo(2)
 Dimension 2, [:A, :B]
And data, a 2×2 Matrix{VariableRef}:
 x[1,A] x[1,B]
 x[2,A] x[2,B]
```

DenseAxisArrays can be indexed and sliced as follows:

```
julia> x[1, :A]
x[1,A]
```

```
julia> x[2, :]
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
 Dimension 1, [:A, :B]
And data, a 2-element Vector{VariableRef}:
 x[2,A]
 x[2,B]
```

Bounds can depend upon indices:

```
julia> model = Model();
julia> @variable(model, x[i=2:3, j=1:2:3] >= 0.5i + j)
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
 Dimension 1, 2:3
 Dimension 2, 1:2:3
And data, a 2×2 Matrix{VariableRef}:
 x[2,1] x[2,3]
 x[3,1] x[3,3]
julia> lower_bound.(x)
```

```
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
 Dimension 1, 2:3
 Dimension 2, 1:2:3
And data, a 2×2 Matrix{Float64}:
 2.0 4.0
 2.5 4.5
```

#### SparseAxisArrays

The third container type that JuMP natively supports is SparseAxisArray. These arrays are created when the indices do not form a rectangular set. For example, this applies when indices have a dependence upon previous indices (called *triangular indexing*). JuMP supports this as follows:

julia> model = Model();

julia> @variable(model, x[i=1:2, j=i:2])

```
JuMP.Containers.SparseAxisArray{VariableRef, 2, Tuple{Int64, Int64}} with 3 entries:
 [1, 1] = x[1,1]
 [1, 2] = x[1,2]
 [2, 2] = x[2,2]
```

We can also conditionally create variables via a JuMP-specific syntax. This syntax appends a comparison check that depends upon the named indices and is separated from the indices by a semi-colon (;). For example:

```
julia> model = Model();
```

```
julia> @variable(model, x[i=1:4; mod(i, 2)==0])
JuMP.Containers.SparseAxisArray{VariableRef, 1, Tuple{Int64}} with 2 entries:
 [2] = x[2]
 [4] = x[4]
```

#### Performance considerations

When using the semi-colon as a filter, JuMP iterates over *all* indices and evaluates the conditional for each combination. If there are many index dimensions and a large amount of sparsity, this can be inefficient.

For example:

```
julia> model = Model();
julia> N = 10
10
julia> S = [(1, 1, 1), (N, N, N)]
2-element Vector{Tuple{Int64, Int64, Int64}}:
(1, 1, 1)
(10, 10, 10)
julia> @time @variable(model, x1[i=1:N, j=1:N, k=1:N; (i, j, k) in S])
 0.203861 seconds (392.22 k allocations: 23.977 MiB, 99.10% compilation time)
JuMP.Containers.SparseAxisArray{VariableRef, 3, Tuple{Int64, Int64, Int64}} with 2 entries:
 [1, 1, 1] = x1[1,1,1]
 [10, 10, 10] = \times 1[10, 10, 10]
julia> @time @variable(model, x2[S])
 0.045407 seconds (65.24 k allocations: 3.771 MiB, 99.15% compilation time)
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
 Dimension 1, [(1, 1, 1), (10, 10, 10)]
And data, a 2-element Vector{VariableRef}:
```

```
x2[(1, 1, 1)]
x2[(10, 10, 10)]
```

The first option is slower because it is equivalent to:

julia> model = Model();

```
julia> x1 = Dict{NTuple{3,Int},VariableRef}()
Dict{Tuple{Int64, Int64, Int64}, VariableRef}()
```

If performance is a concern, explicitly construct the set of indices instead of using the filtering syntax.

## Forcing the container type

When creating a container of JuMP variables, JuMP will attempt to choose the tightest container type that can store the JuMP variables. Thus, it will prefer to create an Array before a DenseAxisArray and a DenseAxisArray before a SparseAxisArray. However, because this happens at compile time, JuMP does not always make the best choice. To illustrate this, consider the following example:

```
julia> model = Model();
julia> A = 1:2
1:2
julia> @variable(model, x[A])
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
 Dimension 1, 1:2
And data, a 2-element Vector{VariableRef}:
 x[1]
 x[2]
```

Since the value (and type) of A is unknown at parsing time, JuMP is unable to infer that A is a one-based integer range. Therefore, JuMP creates a DenseAxisArray, even though it could store these two variables in a standard one-dimensional Array.

We can share our knowledge that it is possible to store these JuMP variables as an array by setting the container keyword:

```
julia> @variable(model, y[A], container=Array)
2-element Vector{VariableRef}:
 y[1]
 y[2]
```

JuMP now creates a vector of JuMP variables instead of a DenseAxisArray. Choosing an invalid container type will throw an error.

## **User-defined containers**

In addition to the built-in container types, you can create your own collections of JuMP variables.

Tip This is a point that users often overlook: you are not restricted to the built-in container types in JuMP.

For example, the following code creates a dictionary with symmetric matrices as the values:

Another common scenario is a request to add variables to existing containers, for example:

```
using JuMP
model = Model()
@variable(model, x[1:2] >= 0)
Later I want to add
@variable(model, x[3:4] >= 0)
```

This is not possible with the built-in JuMP container types. However, you can use regular Julia types instead:

```
julia> model = Model();
julia> x = model[:x] = @variable(model, [1:2], lower_bound = 0, base_name = "x")
2-element Vector{VariableRef}:
 x[1]
 x[2]
julia> append!(x, @variable(model, [1:2], lower_bound = 0, base_name = "y"));
julia> model[:x]
4-element Vector{VariableRef}:
 x[1]
 x[2]
 y[1]
 y[2]
```

## **Sparse arrays**

To construct a SparseArrays.SparseMatrixCSC of JuMP variables, create a vector of JuMP variables for the non-zero elements, and then use the SparseArrays.sparse constructor:

julia> import SparseArrays

julia> model = Model();

julia> x = @variable(model, [1:5]);

# 12.13 Semidefinite variables

Declare a square matrix of JuMP variables to be positive semidefinite by passing PSD as a positional argument:

```
julia> model = Model();
julia> @variable(model, x[1:2, 1:2], PSD)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
x[1,1] x[1,2]
x[1,2] x[2,2]
```

This will ensure that x is symmetric, and that all of its eigenvalues are nonnegative.

#### Note

x must be a square 2-dimensional Array of JuMP variables; it cannot be a DenseAxisArray or a SparseAxisArray.

Use VariableInSetRef to obtain the associated constraint reference:

```
julia> model = Model();
```

The PSD argument must be provided explicitly to the macro. Passing it via a variable throws an error:

```
julia> model = Model();
julia> type = :PSD
:PSD
```

Instead, pass PSDCone via the x in Set syntax:

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2, 1:2] in PSDCone())
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 x[1,1] x[1,2]
 x[1,2] x[2,2]
```

```
julia> set = PSDCone();
```

```
julia> @variable(model, y[1:2, 1:2] in set)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
y[1,1] y[1,2]
y[1,2] y[2,2]
```

```
or via the set keyword:
```

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2, 1:2], set = PSDCone())
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
x[1,1] x[1,2]
x[1,2] x[2,2]
```

```
julia> set = PSDCone();
```

```
julia> @variable(model, y[1:2, 1:2], set = set)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
y[1,1] y[1,2]
y[1,2] y[2,2]
```

# 12.14 Symmetric variables

Declare a square matrix of JuMP variables to be symmetric (but not necessarily positive semidefinite) by passing Symmetric as an optional positional argument:

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2, 1:2], Symmetric)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
```

x[1,1] x[1,2] x[1,2] x[2,2]

The Symmetric argument must provided explicitly to the macro. Passing it via a variable throws an error:

```
Instead, pass SymmetricMatrixSpace via the x in Set syntax:
```

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2, 1:2] in SymmetricMatrixSpace())
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 x[1,1] x[1,2]
 x[1,2] x[2,2]
```

```
julia> set = SymmetricMatrixSpace();
```

```
julia> @variable(model, y[1:2, 1:2] in set)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
y[1,1] y[1,2]
y[1,2] y[2,2]
```

or via the set keyword:

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2, 1:2], set = SymmetricMatrixSpace())
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 x[1,1] x[1,2]
 x[1,2] x[2,2]
```

julia> set = SymmetricMatrixSpace();

```
julia> @variable(model, y[1:2, 1:2], set = set)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
y[1,1] y[1,2]
y[1,2] y[2,2]
```

## 12.15 The @variables macro

If you have many @variable calls, JuMP provides the macro @variables that can improve readability:

The @variables macro returns a tuple of the variables that were defined.

Note

 $Y_2[2] \ge 2$ z binary

Keyword arguments must be contained within parentheses.

## 12.16 Variables constrained on creation

All uses of the *@variable* macro documented so far translate into separate calls for variable creation and the adding of any bound or integrality constraints.

For example,  $@variable(model, x \ge 0, Int)$ , is equivalent to:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> set_lower_bound(x, 0.0)
julia> set_integer(x)
```

Importantly, the bound and integrality constraints are added after the variable has been created.

However, some solvers require a set specifying the variable domain to be given when the variable is first created. We say that these variables are *constrained on creation*.

Use in within @variable to access the special syntax for constraining variables on creation.

For example, the following creates a vector of variables that belong to the SecondOrderCone:

julia> model = Model();

```
julia> @variable(model, y[1:3] in SecondOrderCone())
3-element Vector{VariableRef}:
 y[1]
 y[2]
 y[3]
```

For contrast, the standard syntax is as follows:

```
julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
```

```
julia> @constraint(model, x in SecondOrderCone())
[x[1], x[2], x[3]] ∈ MathOptInterface.SecondOrderCone(3)
```

An alternate syntax to x in Set is to use the set keyword of @variable. This is most useful when creating anonymous variables:

```
julia> model = Model();
```

```
julia> x = @variable(model, [1:3], set = SecondOrderCone())
3-element Vector{VariableRef}:
 _[1]
 _[2]
 _[3]
```

#### Note

You cannot delete the constraint associated with a variable constrained on creation.

To check if a variable was constrained on creation, use is\_variable\_in\_set, and use VariableInSetRef to obtain the associated constraint reference:

```
julia> is_variable_in_set(y)
false
julia> @variable(model, z in Semicontinuous(1, 2))
z
julia> is_variable_in_set(z)
true
julia> c_z = VariableInSetRef(z)
z ∈ MathOptInterface.Semicontinuous{Int64}(1, 2)
```

#### Example: positive semidefinite variables

An alternative to the syntax in Semidefinite variables, declare a matrix of JuMP variables to be positive semidefinite using PSDCone:

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2, 1:2] in PSDCone())
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
x[1,1] x[1,2]
x[1,2] x[2,2]
julia> @variable(model, y[1:2, 1:2], set = PSDCone())
2x2 LinearAlgebra Symmetric{VariableRef Matrix{VariableRef}}:
```

```
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 y[1,1] y[1,2]
 y[1,2] y[2,2]
```

## **Example: symmetric variables**

As an alternative to the syntax in Symmetric variables, declare a matrix of JuMP variables to be symmetric using SymmetricMatrixSpace:

```
julia> model = Model();
julia> @variable(model, x[1:2, 1:2] in SymmetricMatrixSpace())
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
x[1,1] x[1,2]
```

```
x[1,2] x[2,2]
```

```
julia> @variable(model, y[1:2, 1:2], set = SymmetricMatrixSpace())
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
y[1,1] y[1,2]
y[1,2] y[2,2]
```

## Example: skew-symmetric variables

Declare a matrix of JuMP variables to be skew-symmetric using SkewSymmetricMatrixSpace:

```
julia> model = Model();
```

#### Note

Even though x is a 2 by 2 matrix, only one decision variable is added to model; the remaining elements in x are linear transformations of the single variable.

#### Example: Hermitian positive semidefinite variables

Declare a matrix of JuMP variables to be Hermitian positive semidefinite using HermitianPSDCone:

```
julia> model = Model();
```

This adds 4 real variables in the MOI.HermitianPositiveSemidefiniteConeTriangle:

```
julia> c = VariableInSetRef(H)
```

[real(H[1,1]) real(H[1,2]) + imag(H[1,2]) im real(H[1,2]) - imag(H[1,2]) im real(H[2,2])] \infty HermitianPSDCone()

#### julia> o = constraint\_object(c);

```
julia> o.func
```

4-element Vector{VariableRef}:
 real(H[1,1])
 real(H[1,2])
 real(H[2,2])
imag(H[1,2])

#### julia> o.set

MathOptInterface.HermitianPositiveSemidefiniteConeTriangle(2)

#### **Example: Hermitian variables**

Declare a matrix of JuMP variables to be Hermitian using the Hermitian tag:

```
julia> model = Model();
```

This is equivalent to declaring the variable in HermitianMatrixSpace:

```
julia> model = Model();
```

#### Why use variables constrained on creation?

For most users, it does not matter if you use the constrained on creation syntax. Therefore, use whatever syntax you find most convenient.

However, if you use direct\_model, you may be forced to use the constrained on creation syntax.

The technical difference between variables constrained on creation and the standard JuMP syntax is that variables constrained on creation calls MOI.add\_constrained\_variables, while the standard JuMP syntax calls MOI.add\_variables and then MOI.add\_constraint.

Consult the implementation of solver package you are using to see if your solver requires MOI.add\_constrained\_variables.

#### **12.17** Parameters

Some solvers have explicit support for parameters, which are constants in the model that can be efficiently updated between solves.

JuMP implements parameters by a decision variable constrained on creation to a value of the Parameter set. For example, the following creates two parameters, p[1] and p[2], with parameter values 2.0 and 4.0:

```
julia> model = Model();
julia> @variable(model, x);
julia> @variable(model, p[i in 1:2] in Parameter(2.0 * i))
2-element Vector{VariableRef}:
p[1]
p[2]
```

Use parameter\_value and set\_parameter\_value to query or update the value of a parameter.

```
julia> parameter_value.(p)
2-element Vector{Float64}:
 2.0
 4.0
julia> set_parameter_value(p[2], 3.0)
julia> parameter_value.(p)
2-element Vector{Float64}:
 2.0
 3.0
```

Use is\_parameter and ParameterRef to check if the variable is a parameter and to get the constraint that makes the variable a parameter.

```
julia> is_parameter(p[1])
true
julia> is_parameter(x)
false
julia> ParameterRef(p[2])
p[2] ∈ MathOptInterface.Parameter{Float64}(3.0)
```

Create anonymous parameters using the set keyword:

```
julia> anon_parameter = @variable(model, set = Parameter(1.0))
_[4]
```

## Limitations

Parameters are implemented as decision variables belonging to the Parameter set. If the solver supports the MOI.Parameter set, it may decide to replace all instances of the parameter variable by the associated constant. If the solver does not support parameters, it will add the parameter as a decision variable with fixed bounds.

The most important implication of this design is that JuMP treats a parameter multiplied by a decision variable as a quadratic expression, even though it is equivalent to a linear expression.

```
julia> begin
```

```
model = Model()
@variable(model, x >= 3)
@variable(model, p in Parameter(2))
@objective(model, Min, p * x)
objective_function_type(model)
end
QuadExpr (alias for GenericQuadExpr{Float64, GenericVariableRef{Float64}})
```

As a consequence, solving a "linear" program with a solver like HiGHS fails:

```
julia> using HiGHS
```

```
julia> begin
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, x >= 3)
 @variable(model, p in Parameter(2))
 @objective(model, Min, p * x)
 optimize!(model)
 is_solved_and_feasible(model)
 end
false
```

because the model is equivalent to a quadratic program with an indefinite objective:

#### Note

The quadratic limitation affects only models with multiplicative parameters such as p \* x. Functions that depend additively on parameters such as p + x do not have the limitation because the resulting function is still affine.

#### ParametricOptInterface

To avoid the problem of p \* x being an indefinite quadratic, use ParametricOptInterface.jl. ParametricOptInterface.jl. ParametricOptInterface provides a POI.Optimizer layer that will substitute each parameter with its numeric value prior to solving. Thus, in the following example, HiGHS will successfully solve a linear program instead of failing to solve a quadratic program.

```
julia> using HiGHS
julia> import ParametricOptInterface as POI
julia> begin
 model = Model(() -> POI.Optimizer(HiGHS.Optimizer()))
 set_silent(model)
 @variable(model, x >= 3)
 @variable(model, p in Parameter(2))
 @objective(model, Min, p * x)
```

```
optimize!(model)
end
julia> is_solved_and_feasible(model)
true
julia> objective_value(model)
6.0
```

If you use Parameter, then in most cases you should also use ParametricOptInterface.jl. There are two main exceptions:

- 1. your solver natively supports the MOI. Parameter set (for example, lpopt.jl)
- 2. you have only additive parameters (for example, x + p), and your solver supports some sort of presolve that can remove fixed variables.

#### When to use a parameter

Parameters are most useful when solving models in a sequence. For example:

```
julia> using JuMP, Ipopt
julia> model = Model(Ipopt.Optimizer);
julia> set_silent(model)
julia> @variable(model, x)
Х
julia> @variable(model, p in Parameter(1.0))
р
julia> @objective(model, Min, (x - p)^2)
x^2 - 2 p^*x + p^2
julia> solution = Dict{Int,Float64}();
julia> for p_value in 1:5
 set_parameter_value(p, p_value)
 optimize!(model)
 assert_is_solved_and_feasible(model)
 solution[p_value] = value(x)
 end
julia> solution
Dict{Int64, Float64} with 5 entries:
 5 => 5.0
 4 => 4.0
 2 => 2.0
 3 => 3.0
 1 => 1.0
```

```
597
```

# CHAPTER 12. VARIABLES

Using parameters can be faster than creating a new model from scratch with updated data because JuMP is able to avoid repeating a number of steps in processing the model before handing it off to the solver.

# **Chapter 13**

# Constraints

JuMP is based on the MathOptInterface (MOI) API. Because of this, JuMP uses the following standard form to represent problems:

$\min_{x \in \mathbb{R}^n}$	$f_0(x)$		(13.1)
s.t.	$f_i(x) \in \mathcal{S}_i$	$i = 1 \dots m$	(13.2)

Each constraint,  $f_i(x) \in S_i$ , is composed of a function and a set. For example, instead of calling  $a^{\top}x \leq b$  a *less-than-or-equal-to* constraint, we say that it is a *scalar-affine-in-less-than* constraint, where the function  $a^{\top}x$  belongs to the *less-than* set  $(-\infty, b]$ . We use the shorthand *function-in-set* to refer to constraints composed of different types of functions and sets.

This page explains how to write various types of constraints in JuMP. For nonlinear constraints, see Nonlinear Modeling instead.

# **13.1 Add a constraint**

Add a constraint to a JuMP model using the @constraint macro. The syntax to use depends on the type of constraint you wish to add.

#### Add a linear constraint

Create linear constraints using the @constraint macro:

```
julia> model = Model();
julia> @variable(model, x[1:3]);
julia> @constraint(model, c1, sum(x) <= 1)
c1 : x[1] + x[2] + x[3] ≤ 1
julia> @constraint(model, c2, x[1] + 2 * x[3] >= 2)
c2 : x[1] + 2 x[3] ≥ 2
julia> @constraint(model, c3, sum(i * x[i] for i in 1:3) == 3)
c3 : x[1] + 2 x[2] + 3 x[3] = 3
```

```
julia> @constraint(model, c4, 4 <= 2 * x[2] <= 5)
c4 : 2 x[2] ∈ [4, 5]
```

## Normalization

JuMP normalizes constraints by moving all of the terms containing variables to the left-hand side and all of the constant terms to the right-hand side. Thus, we get:

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, c, 2x + 1 <= 4x + 4)
c : -2 x ≤ 3</pre>
```

#### Add a quadratic constraint

In addition to affine functions, JuMP also supports constraints with quadratic terms. For example:

```
julia> model = Model();
julia> @variable(model, x[i=1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
julia> @variable(model, t >= 0)
t
julia> @constraint(model, my_q, x[1]^2 + x[2]^2 <= t^2)
my_q : x[1]² + x[2]² - t² ≤ 0
```

## Тір

Because solvers can take advantage of the knowledge that a constraint is quadratic, prefer adding quadratic constraints using @constraint, rather than @NLconstraint.

# 13.2 Vectorized constraints

You can also add constraints to JuMP using vectorized linear algebra. For example:

```
julia> @variable(model, x[i=1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

julia> model = Model();

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
1 2
3 4
julia> b = [5, 6]
2-element Vector{Int64}:
5
6
julia> @constraint(model, con_vector, A * x == b)
con_vector : [x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] \in Zeros()
julia> @constraint(model, con_scalar, A * x .== b)
2-element Vector{ConstraintRef{Model,
← MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
→ MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
con_scalar : x[1] + 2 x[2] = 5
con_scalar : 3 \times [1] + 4 \times [2] = 6
```

The two constraints, == and .== are similar, but subtly different. The first creates a single constraint that is a MOI.VectorAffineFunction in MOI.Zeros constraint. The second creates a vector of MOI.ScalarAffineFunction in MOI.EqualTo constraints.

Which formulation to choose depends on the solver, and what you want to do with the constraint object con\_vector or con\_scalar.

- If you are using a conic solver, expect the dual of con\_vector to be a Vector{Float64}, and do not
  intend to delete a row in the constraint, choose the == formulation.
- If you are using a solver that expects a list of scalar constraints, for example HiGHS, or you wish to delete
  part of the constraint or access a single row of the constraint, for example, dual(con\_scalar[2]), then
  use the broadcast .==.

JuMP reformulates both constraints into the other form if needed by the solver, but choosing the right format for a particular solver is more efficient.

You can also use <=, .<= , >=, and .>= as comparison operators in the constraint.

```
julia> @constraint(model, A * x <= b)
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ Nonpositives()</pre>
```

```
julia> @constraint(model, A * x .<= b)</pre>
```

```
2-element Vector{ConstraintRef{Model,

→ MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

→ MathOptInterface.LessThan{Float64}}, ScalarShape}}:

x[1] + 2 x[2] ≤ 5

3 x[1] + 4 x[2] ≤ 6
```

```
julia> @constraint(model, A * x >= b)
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ Nonnegatives()
```

#### julia> @constraint(model, A \* x .>= b)

2-element Vector{ConstraintRef{Model,

← MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

 $\hookrightarrow \quad \texttt{MathOptInterface.GreaterThan{Float64}}, \ \texttt{ScalarShape}\}:$ 

```
x[1] + 2 x[2] \ge 5
3 x[1] + 4 x[2] ≥ 6
```

## 13.3 Matrix inequalities

Inequalities between matrices are not supported, due to the common ambiguity between elementwise inequalities and a PSDCone constraint.

```
julia> model = Model();
julia> @variable(model, x[1:2, 1:2], Symmetric);
julia> @variable(model, y[1:2, 1:2], Symmetric);
julia> @constraint(model, x >= y)
ERROR: At none:1: `@constraint(model, x >= y)`:
The syntax x \ge y is ambiguous for matrices because we cannot tell if
you intend a positive semidefinite constraint or an elementwise
inequality.
To create a positive semidefinite constraint, pass `PSDCone()` or
`HermitianPSDCone()`:
```julia
@constraint(model, x >= y, PSDCone())
To create an element-wise inequality, pass `Nonnegatives()`, or use
broadcasting:
```julia
@constraint(model, x >= y, Nonnegatives())
or
@constraint(model, x .>= y)
Stacktrace:
[...]
```

Instead, use the Set inequality syntax to specify a set like PSDCone or Nonnegatives:

x[2,2] - y[2,2]] ∈ Nonpositives()

```
602
```

```
julia> @constraint(model, x >= y, Zeros())
[x[1,1] - y[1,1] x[1,2] - y[1,2]
...
x[2,2] - y[2,2]] ∈ Zeros()
```

#### Special cases

There are two exceptions: if the result of the left-hand side minus the right-hand side is a LinearAlgebra.Symmetric matrix or a LinearAlgebra.Hermitian matrix, you may use the non-broadcasting equality syntax:

```
julia> using LinearAlgebra
julia> model = Model();
julia> @variable(model, X[1:2, 1:2], Symmetric)
2×2 Symmetric{VariableRef, Matrix{VariableRef}}:
X[1,1] X[1,2]
X[1,2] X[2,2]
julia> @constraint(model, X == LinearAlgebra.I)
[X[1,1] - 1 X[1,2]
... X[2,2] - 1] ∈ Zeros()
```

This will add only three rows to the constraint matrix because the symmetric constraints are redundant. In contrast, the broadcasting syntax adds four linear constraints:

#### The same holds for LinearAlgebra.Hermitian matrices:

```
julia> using LinearAlgebra
```

```
julia> model = Model();
```

#### julia> @constraint(model, X .== LinearAlgebra.I)

2×2 Matrix{ConstraintRef{Model, → MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{ComplexF64}, → MathOptInterface.EqualTo{ComplexF64}}, ScalarShape}}:

## 13.4 Containers of constraints

The @constraint macro supports creating collections of constraints. We'll cover some brief syntax here; read the Constraint containers section for more details:

Create arrays of constraints:

Sets can be any Julia type that supports iteration:

 $c[2, red] : x[2] \le 4$ 

Sets can depend upon previous indices:

julia> model = Model();

julia> @variable(model, x[1:3]);
and you can filter elements in the sets using the ; syntax:

### 13.5 Registered constraints

When you create constraints, JuMP registers them inside the model using their corresponding symbol. Get a registered name using model[:key]:

```
julia> model = Model()
A JuMP Model
- solver: none
bjective_sense: FEASIBILITY_SENSE
- num_variables: 0
- num constraints: 0
L Names registered in the model: none
julia> @variable(model, x)
Х
julia> @constraint(model, my_c, 2x <= 1)</pre>
my_c : 2 \times \le 1
iulia> model
A JuMP Model
- solver: none
bjective_sense: FEASIBILITY_SENSE
- num_variables: 1
- num constraints: 1
L AffExpr in MOI.LessThan{Float64}: 1
```

```
L Names registered in the model
L :my_c, :x
julia> model[:my_c] === my_c
true
```

## 13.6 Anonymous constraints

To reduce the likelihood of accidental bugs, and because JuMP registers constraints inside a model, creating two constraints with the same name is an error:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @constraint(model, c, 2x <= 1)
c : 2 x ≤ 1
julia> @constraint(model, c, 2x <= 1)
ERROR: An object of name c is already attached to this model. If this
 is intended, consider using the anonymous construction syntax, for example,
 `x = @variable(model, [1:N], ...)` where the name of the object does
 not appear inside the macro.
Alternatively, use `unregister(model, :c)` to first unregister
 the existing name from the model. Note that this will not delete the
 object; it will just remove the reference at `model[:c]`.
[...]
```

A common reason for encountering this error is adding constraints in a loop.

As a work-around, JuMP provides *anonymous* constraints. Create an anonymous constraint by omitting the name argument:

```
julia> model = Model();
julia> @variable(model, x);
julia> c = @constraint(model, 2x <= 1)
2 x ≤ 1
```

Create a container of anonymous constraints by dropping the name in front of the [:

```
julia> model = Model();
```

julia> @variable(model, x[1:3]);

julia> c = @constraint(model, [i = 1:3], x[i] <= i)</pre>

3-element Vector{ConstraintRef{Model,

 $\hookrightarrow \quad \texttt{MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},}$ 

 $\hookrightarrow$  MathOptInterface.LessThan{Float64}}, ScalarShape}}:

 $x[1] \le 1$  $x[2] \le 2$  $x[3] \le 3$ 

## 13.7 Constraint names

In addition to the symbol that constraints are registered with, constraints have a String name that is used for printing and writing to file formats.

Get and set the name of a constraint using name(::JuMP.ConstraintRef) and set\_name(::JuMP.ConstraintRef, ::String):

```
julia> model = Model(); @variable(model, x);
julia> @constraint(model, con, x <= 1)
con : x ≤ 1
julia> name(con)
"con"
julia> set_name(con, "my_con_name")
julia> con
my_con_name : x ≤ 1
```

Override the default choice of name using the base\_name keyword:

Note that names apply to each element of the container, not to the container of constraints:

### Тір

For some models, setting the string name of each constraint can take a non-trivial portion of the total time required to build the model. Turn off String names by passing set\_string\_name = false to @constraint:

julia> model = Model();

```
julia> @variable(model, x);
```

```
julia> @constraint(model, con, x <= 2, set_string_name = false)
x ≤ 2</pre>
```

See Disable string names for more information.

#### Retrieve a constraint by name

Retrieve a constraint from a model using constraint\_by\_name:

```
julia> constraint_by_name(model, "c")
c : x ≤ 1
```

If the name is not present, nothing will be returned:

```
julia> constraint_by_name(model, "bad_name")
```

You can only look up individual constraints using constraint\_by\_name. Something like this will not work:

julia> constraint\_by\_name(model, "my\_con")

To look up a collection of constraints, do not use constraint\_by\_name. Instead, register them using the model[:key] = value syntax:

```
my_con[2] : x \le 2
```

### 13.8 String names, symbolic names, and bindings

It's common for new users to experience confusion relating to constraints. Part of the problem is the difference between the name that a constraint is registered under and the String name used for printing.

Here's a summary of the differences:

- Constraints are created using @constraint.
- Constraints can be named or anonymous.
- Named constraints have the form @constraint(model, c, expr). For named constraints:
  - The String name of the constraint is set to "c".
  - A Julia variable c is created that binds c to the JuMP constraint.
  - The name : c is registered as a key in the model with the value c.
- Anonymous constraints have the form c = @constraint(model, expr). For anonymous constraints:
  - The String name of the constraint is set to "".
  - You control the name of the Julia variable used as the binding.
  - No name is registered as a key in the model.
- The base\_name keyword can override the String name of the constraint.
- You can manually register names in the model via model[:key] = value.

Here's an example of the differences:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> c_binding = @constraint(model, 2x <= 1, base_name = "c")
c : 2 x ≤ 1
julia> model
```

```
A JuMP Model

+ solver: none

+ objective_sense: FEASIBILITY_SENSE

+ num_variables: 1
```

```
hum_constraints: 1
 L AffExpr in MOI.LessThan{Float64}: 1
 Names registered in the model
 L :x
```

julia> c ERROR: UndefVarError: `c` not defined

**julia**> c\_binding c :  $2 \times \le 1$ 

julia> name(c\_binding)
"c"

julia> model[:c\_register] = c\_binding
c : 2 x ≤ 1

#### julia> model

```
A JuMP Model
 solver: none
 objective_sense: FEASIBILITY_SENSE
 num_variables: 1
 num_constraints: 1
 L AffExpr in MOI.LessThan{Float64}: 1
 Names registered in the model
 L :c_register, :x
```

```
julia> model[:c_register]
c : 2 x ≤ 1
```

julia> model[:c\_register] === c\_binding
true

julia> c
ERROR: UndefVarError: `c` not defined

# 13.9 The @constraints macro

If you have many @constraint calls, use the @constraints macro to improve readability:

c : x ≥ -1 2 x ≤ 1

The @constraints macro returns a tuple of the constraints that were defined.

### 13.10 Duality

JuMP adopts the notion of conic duality from MathOptInterface. For linear programs, a feasible dual on a >= constraint is nonnegative and a feasible dual on a <= constraint is nonpositive. If the constraint is an equality constraint, it depends on which direction is binding.

#### Warning

JuMP's definition of duality is independent of the objective sense. That is, the sign of feasible duals associated with a constraint depends on the direction of the constraint and not whether the problem is maximization or minimization. **This is a different convention from linear programming duality in some common textbooks.** If you have a linear program, and you want the textbook definition, you probably want to use shadow\_price and reduced\_cost instead.

The dual value associated with a constraint in the most recent solution can be accessed using the dual function. For example:

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x)
X
julia> @constraint(model, con, x <= 1)</pre>
con : x \le 1
julia> @objective(model, Min, -2x)
-2 x
julia> dual_status(model)
NO_SOLUTION::ResultStatusCode = 0
julia> optimize!(model)
julia> dual_status(model)
FEASIBLE_POINT::ResultStatusCode = 1
julia> dual(con)
-2.0
julia> @objective(model, Max, 2x)
2 x
julia> optimize!(model)
julia> dual_status(model)
```

```
FEASIBLE_POINT::ResultStatusCode = 1
julia> dual(con)
-2.0
```

To help users who may be less familiar with conic duality, JuMP provides shadow\_price, which returns a value that can be interpreted as the improvement in the objective in response to an infinitesimal relaxation (on the scale of one unit) in the right-hand side of the constraint. shadow\_price can be used only on linear constraints with a <=, >=, or == comparison operator.

In the example above, dual(con) returned -2.0 regardless of the optimization sense. However, in the second case when the optimization sense is Max, shadow\_price returns:

```
julia> shadow_price(con)
2.0
```

#### **Duals of variable bounds**

To query the dual variables associated with a variable bound, first obtain a constraint reference using one of UpperBoundRef, LowerBoundRef, or FixRef, and then call dual on the returned constraint reference. The reduced\_cost function may simplify this process as it returns the shadow price of an active bound of a variable (or zero, if no active bound exists).

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x <= 1)
x
julia> @objective(model, Min, -2x)
-2 x
julia> optimize!(model)
julia> dual(UpperBoundRef(x))
-2.0
julia> reduced_cost(x)
-2.0
```

### 13.11 Modify a constant term

This section explains how to modify the constant term in a constraint. There are multiple ways to achieve this goal; we explain three options.

### Option 1: change the right-hand side

Use set\_normalized\_rhs to modify the right-hand side (constant) term of a linear or quadratic constraint. Use normalized\_rhs to query the right-hand side term.

```
julia> model = Model();
```

```
julia> @variable(model, x);
```

julia> @constraint(model, con, 2x <= 1)
con : 2 x ≤ 1</pre>

julia> set\_normalized\_rhs(con, 3)

julia> con con :  $2 \times \le 3$ 

julia> normalized\_rhs(con)
3.0

#### Warning

set\_normalized\_rhs sets the right-hand side term of the normalized constraint. See Normalization for more details.

### **Option 2: use fixed variables**

If constraints are complicated, for example, they are composed of a number of components, each of which has a constant term, then it may be difficult to calculate what the right-hand side term is in the standard form.

For this situation, JuMP includes the ability to *fix* variables to a value using the fix function. Fixing a variable sets its lower and upper bound to the same value. Thus, changes in a constant term can be simulated by adding a new variable and fixing it to different values. Here is an example:

```
julia> model = Model();
julia> @variable(model, x);
julia> @variable(model, const_term)
const_term
julia> @constraint(model, con, 2x <= const_term + 1)
con : 2 x - const_term ≤ 1
julia> fix(const_term, 1.0)
```

The constraint con is now equivalent to  $2x \le 2$ .

#### Warning

Fixed variables are not replaced with constants when communicating the problem to a solver. Therefore, even though const\_term is fixed, it is still a decision variable, and so const\_term \* x is bilinear. CHAPTER 13. CONSTRAINTS

### Option 3: modify the function's constant term

The third option is to use add\_to\_function\_constant. The constant given is added to the function of a func-inset constraint. In the following example, adding 2 to the function has the effect of removing 2 to the right-hand side:

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, con, 2x <= 1)
con : 2 x ≤ 1
julia> add_to_function_constant(con, 2)
julia> con
con : 2 x ≤ -1
julia> normalized_rhs(con)
-1.0
```

In the case of interval constraints, the constant is removed from each bound:

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, con, 0 <= 2x + 1 <= 2)
con : 2 x ∈ [-1, 1]
julia> add_to_function_constant(con, 3)
julia> con
con : 2 x ∈ [-4, -2]
```

### 13.12 Modify a variable coefficient

#### **Scalar constraints**

To modify the coefficients for a linear term in a constraint, use set\_normalized\_coefficient. To query the current coefficient, use normalized\_coefficient.

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @constraint(model, con, 2x[1] + x[2] <= 1)
con : 2 x[1] + x[2] ≤ 1
julia> set_normalized_coefficient(con, x[2], 0)
julia> con
con : 2 x[1] ≤ 1
```

```
julia> normalized_coefficient(con, x[2])
0.0
```

To modify quadratic terms, pass two variables:

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @constraint(model, con, x[1]^2 + x[1] * x[2] <= 1)
con : x[1]² + x[1]*x[2] ≤ 1
julia> set_normalized_coefficient(con, x[1], x[1], 2)
julia> set_normalized_coefficient(con, x[1], x[2], 3)
julia> con
con : 2 x[1]² + 3 x[1]*x[2] ≤ 1
julia> normalized_coefficient(con, x[1], x[1])
2.0
julia> normalized_coefficient(con, x[1], x[2])
3.0
```

#### Warning

set\_normalized\_coefficient sets the coefficient of the normalized constraint. See Normalization for more details.

### Vector constraints

To modify the coefficients of a vector-valued constraint, use set\_normalized\_coefficient.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @constraint(model, con, [2x + 3x, 4x] in MOI.Nonnegatives(2))
con : [5 x, 4 x] ∈ MathOptInterface.Nonnegatives(2)
julia> set_normalized_coefficient(con, x, [(1, 3.0)])
julia> con
con : [3 x, 4 x] ∈ MathOptInterface.Nonnegatives(2)
julia> set_normalized_coefficient(con, x, [(1, 2.0), (2, 5.0)])
```

julia> con con : [2 x, 5 x] ∈ MathOptInterface.Nonnegatives(2)

### 13.13 Delete a constraint

Use delete to delete a constraint from a model. Use is\_valid to check if a constraint belongs to a model and has not been deleted.

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, con, 2x <= 1)
con : 2 x ≤ 1
julia> is_valid(model, con)
true
julia> delete(model, con)
julia> is_valid(model, con)
false
```

Deleting a constraint does not unregister the symbolic reference from the model. Therefore, creating a new constraint of the same name will throw an error:

```
julia> @constraint(model, con, 2x <= 1)
ERROR: An object of name con is already attached to this model. If this
 is intended, consider using the anonymous construction syntax, for example,
 `x = @variable(model, [1:N], ...)` where the name of the object does
 not appear inside the macro.
 Alternatively, use `unregister(model, :con)` to first unregister
 the existing name from the model. Note that this will not delete the
 object; it will just remove the reference at `model[:con]`.
[...]
```

After calling delete, call unregister to remove the symbolic reference:

```
julia> unregister(model, :con)
```

```
julia> @constraint(model, con, 2x <= 1)
con : 2 x ≤ 1</pre>
```

#### Info

delete does not automatically unregister because we do not distinguish between names that are automatically registered by JuMP macros, and names that are manually registered by the user by setting values in object\_dictionary. In addition, deleting a constraint and then adding a new constraint of the same name is an easy way to introduce bugs into your code.

## 13.14 Start values

Provide a starting value (also called warmstart) for a constraint's primal and dual solutions using set\_start\_value and set\_dual\_start\_value.

Query the starting value for a constraint's primal and dual solution using start\_value and dual\_start\_value. If no start value has been set, the methods will return nothing.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @constraint(model, con, x >= 10)
con : x ≥ 10
julia> start_value(con)
julia> start_value(con, 10.0)
julia> start_value(con, 10.0)
julia> dual_start_value(con)
julia> set_dual_start_value(con, 2)
julia> dual_start_value(con)
2.0
```

Vector-valued constraints require a vector:

```
julia> model = Model();
julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
```

julia> @constraint(model, con, x in SecondOrderCone())
con : [x[1], x[2], x[3]] in MathOptInterface.SecondOrderCone(3)

```
julia> dual_start_value(con)
```

julia> set\_dual\_start\_value(con, [1.0, 2.0, 3.0])

julia> dual\_start\_value(con)

3-element Vector{Float64}: 1.0 2.0 3.0

### Тір

To simplify setting start values for all variables and constraints in a model, see set\_start\_values. The Primal and dual warm-starts tutorial also gives a detailed description of how to iterate over constraints in the model to set custom start values.

# 13.15 Constraint containers

Like Variable containers, JuMP provides a mechanism for building groups of constraints compactly. References to these groups of constraints are returned in *containers*. Three types of constraint containers are supported: Arrays, DenseAxisArrays, and SparseAxisArrays. We explain each of these in the following.

#### Тір

You can read more about containers in the Containers section.

#### Arrays

One way of adding a group of constraints compactly is the following:

JuMP returns references to the three constraints in an Array that is bound to the Julia variable con. This array can be accessed and sliced as you would with any Julia array:

```
julia> con[1]
con[1] : x ≤ 2
julia> con[2:3]
2-element Vector{ConstraintRef{Model,

→ MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

→ MathOptInterface.LessThan{Float64}}, ScalarShape}}:

con[2] : 2 x ≤ 3
con[3] : 3 x ≤ 4
```

Anonymous containers can also be constructed by dropping the name (for example, con) before the square brackets:

Just like @variable, JuMP will form an Array of constraints when it can determine at parse time that the indices are one-based integer ranges. Therefore con[1:b] will create an Array, but con[a:b] will not. A special case is con[Base.OneTo(n)] which will produce an Array. If JuMP cannot determine that the indices are one-based integer ranges (for example, in the case of con[a:b]), JuMP will create a DenseAxisArray instead.

#### DenseAxisArrays

The syntax for constructing a DenseAxisArray of constraints is very similar to the syntax for constructing a DenseAxisArray of variables.

#### SparseAxisArrays

The syntax for constructing a SparseAxisArray of constraints is very similar to the syntax for constructing a SparseAxisArray of variables.

#### Warning

If you have many index dimensions and a large amount of sparsity, read Performance considerations.

#### Forcing the container type

When creating a container of constraints, JuMP will attempt to choose the tightest container type that can store the constraints. However, because this happens at parse time, it does not always make the best choice. Just like in <code>@variable</code>, you can force the type of container using the container keyword. For syntax and the reason behind this, take a look at the variable docs.

#### **Constraints with similar indices**

Containers are often used to create constraints over a set of indices. However, you'll often have cases in which you are repeating the indices:

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @variable(model, y[1:2]);
julia> @constraints(model, begin
 [i=1:2, j=1:2, k=1:2], i * x[j] <= k
 [i=1:2, j=1:2, k=1:2], i * y[j] <= k
 end);
```

This is hard to read and leads to a lot of copy-paste. A more readable way is to use a for-loop:

```
julia> for i=1:2, j=1:2, k=1:2
 @constraints(model, begin
 i * x[j] <= k
 i * y[j] <= k
 end)
 end</pre>
```

### 13.16 Accessing constraints from a model

Query the types of function-in-set constraints in a model using list\_of\_constraint\_types:

```
julia> model = Model();
julia> @variable(model, x[i=1:2] >= i, Int);
julia> @constraint(model, x[1] + x[2] <= 1);
julia> list_of_constraint_types(model)
3-element Vector{Tuple{Type, Type}}:
(AffExpr, MathOptInterface.LessThan{Float64})
(VariableRef, MathOptInterface.GreaterThan{Float64})
```

For a given combination of function and set type, use num\_constraints to access the number of constraints and all\_constraints to access a list of their references:

You can also count the total number of constraints in the model, but you must explicitly choose whether to count VariableRef constraints such as bound and integrality constraints:

```
julia> num_constraints(model; count_variable_in_set_constraints = true)
5
julia> num_constraints(model; count_variable_in_set_constraints = false)
1
```

The same also applies for all\_constraints:

```
julia> all_constraints(model; include_variable_in_set_constraints = true)
5-element Vector{ConstraintRef}:
 x[1] + x[2] ≤ 1
 x[1] ≥ 1
 x[2] ≥ 2
 x[1] integer
 x[2] integer
julia> all_constraints(model; include_variable_in_set_constraints = false)
1-element Vector{ConstraintRef}:
```

```
x[1] + x[2] \le 1
```

If you need finer-grained control on which constraints to include, use a variant of:

Use constraint\_object to get an instance of an AbstractConstraint object that stores the constraint data:

```
julia> con = constraint_object(cons[1])
ScalarConstraint{VariableRef, MathOptInterface.Integer}(x[1], MathOptInterface.Integer())
```

```
julia> con.func
x[1]
```

julia> con.set
MathOptInterface.Integer()

### **13.17** MathOptInterface constraints

Because JuMP is based on MathOptInterface, you can add any constraints supported by MathOptInterface using the function-in-set syntax. For a list of supported functions and sets, read Standard form problem.

Note

We use MOI as an alias for the MathOptInterface module. This alias is defined by using JuMP. You may also define it in your code as follows:

import MathOptInterface as MOI

For example, the following two constraints are equivalent:

```
julia> model = Model();
julia> @variable(model, x[1:3]);
julia> @constraint(model, 2 * x[1] <= 1)
2 x[1] ≤ 1
julia> @constraint(model, 2 * x[1] in MOI.LessThan(1.0))
2 x[1] ≤ 1
```

You can also use any set defined by MathOptInterface:

```
julia> @constraint(model, x - [1; 2; 3] in MOI.Nonnegatives(3))
[x[1] - 1, x[2] - 2, x[3] - 3] ∈ MathOptInterface.Nonnegatives(3)
```

```
julia> @constraint(model, x in MOI.ExponentialCone())
[x[1], x[2], x[3]] ∈ MathOptInterface.ExponentialCone()
```

### Info

Similar to how JuMP defines the <= and >= syntax as a convenience way to specify MOI.LessThan and MOI.GreaterThan constraints, the remaining sections in this page describe functions and syntax that have been added for the convenience of common modeling situations.

### 13.18 Set inequality syntax

For modeling convenience, the syntax @constraint(model, x >= y, Set()) is short-hand for @constraint(model, x - y in Set()). Therefore, the following calls are equivalent:

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> y = [0.5, 0.75];
julia> @constraint(model, x >= y, MOI.Nonnegatives(2))
[x[1] - 0.5, x[2] - 0.75] ∈ MathOptInterface.Nonnegatives(2)
julia> @constraint(model, x - y in MOI.Nonnegatives(2))
```

[x[1] - 0.5, x[2] - 0.75] Ext{ MathOptInterface.Nonnegatives(2)

Non-zero constants are not supported in this syntax:

Use instead:

```
julia> @constraint(model, x .- 1 >= 0, MOI.Nonnegatives(2))
[x[1] - 1, x[2] - 1] ∈ MathOptInterface.Nonnegatives(2)
```

#### Warning

The syntax @constraint(model,  $y \le x$ , Set()) is supported, but it is not recommended because the value of the primal and dual solutions associated with the constraint may be the negative of what you expect.

### 13.19 Second-order cone constraints

A SecondOrderCone constrains the variables t and x to the set:

```
||x||_2 \le t,
```

and  $t \ge 0$ . It can be added as follows:

```
julia> model = Model();
julia> @variable(model, t)
t
```

```
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

```
julia> @constraint(model, [t; x] in SecondOrderCone())
[t, x[1], x[2]] ∈ MathOptInterface.SecondOrderCone(3)
```

### 13.20 Rotated second-order cone constraints

A RotatedSecondOrderCone constrains the variables t, u, and x to the set:

$$||x||_2^2 \le 2t \cdot u$$

and  $t, u \geq 0$ . It can be added as follows:

```
[t, u, x[1], x[2]] ∈ MathOptInterface.RotatedSecondOrderCone(4)
```

# 13.21 Special Ordered Sets of Type 1

In a Special Ordered Set of Type 1 (often denoted SOS-I or SOS1), at most one element can take a non-zero value.

Construct SOS-I constraints using the SOS1 set:

 Although not required for feasibility, solvers can benefit from an ordering of the variables (for example, the variables represent different factories to build, at most one factory can be built, and the factories can be ordered according to cost). To induce an ordering, a vector of weights can be provided, and the variables are ordered according to their corresponding weight.

For example, in the constraint:

```
julia> @constraint(model, x in SOS1([3.1, 1.2, 2.3]))
[x[1], x[2], x[3]] in MathOptInterface.SOS1{Float64}([3.1, 1.2, 2.3])
```

the variables x have precedence x[2], x[3], x[1].

### 13.22 Special Ordered Sets of Type 2

In a Special Ordered Set of Type 2 (SOS-II), at most two elements can be non-zero, and if there are two non-zeros, they must be consecutive according to the ordering induced by a weight vector.

Construct SOS-II constraints using the SOS2 set:

```
julia> @constraint(model, x in SOS2([3.0, 1.0, 2.0]))
[x[1], x[2], x[3]] in MathOptInterface.SOS2{Float64}([3.0, 1.0, 2.0])
```

The possible non-zero pairs are (x[1], x[3]) and (x[2], x[3]):

If the weight vector is omitted, JuMP induces an ordering from 1:length(x):

```
julia> @constraint(model, x in SOS2())
[x[1], x[2], x[3]] in MathOptInterface.SOS2{Float64}([1.0, 2.0, 3.0])
```

### 13.23 Indicator constraints

Indicator constraints consist of a binary variable and a linear constraint. The constraint holds when the binary variable takes the value 1. The constraint may or may not hold when the binary variable takes the value 0.

To enforce the constraint  $x + y \le 1$  when the binary variable a is 1, use:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> @variable(model, a, Bin)
a
julia> @constraint(model, a --> {x + y <= 1})
a --> {x + y < 1}</pre>
```

If the constraint must hold when a is zero, add ! or ¬ before the binary variable;

```
julia> @constraint(model, !a --> {x + y <= 1})
!a --> {x + y ≤ 1}
```

#### Warning

You cannot use an expression for the left-hand side of an indicator constraint.

# 13.24 Semidefinite constraints

To constrain a matrix to be positive semidefinite (PSD), use PSDCone:

### Тір

Where possible, prefer constructing a matrix of Semidefinite variables using the @variable macro, rather than adding a constraint like @constraint(model,  $X \ge 0$ , PSDCone()). In some solvers, adding the constraint via @constraint is less efficient, and can result in additional intermediate variables and constraints being added to the model.

The inequality  $X \ge Y$  between two square matrices X and Y is understood as constraining X - Y to be positive semidefinite.

```
julia> Y = [1 2; 2 1]
2×2 Matrix{Int64}:
 1 2
 2 1

julia> @constraint(model, X >= Y, PSDCone())
[X[1,1] - 1 X[1,2] - 2
X[2,1] - 2 X[2,2] - 1] ∈ PSDCone()
```

#### Warning

The syntax @constraint(model, Y <= X, PSDCone()) is supported, but it is not recommended because the value of the primal and dual solutions associated with the constraint may be the negative of what you expect.

#### Symmetry

Solvers supporting PSD constraints usually expect to be given a matrix that is *symbolically* symmetric, that is, for which the expression in corresponding off-diagonal entries are the same. In our example, the expressions of entries (1, 2) and (2, 1) are respectively X[1,2] - 2 and X[2,1] - 2 which are different.

To bridge the gap between the constraint modeled and what the solver expects, solvers may add an equality constraint X[1,2] - 2 = X[2,1] - 2 to force symmetry. Use LinearAlgebra.Symmetric to explicitly tell the solver that the matrix is symmetric:

julia> import LinearAlgebra

Note that the lower triangular entries are ignored even if they are different so use it with caution:

```
julia> @constraint(model, LinearAlgebra.Symmetric(X) >= 0, PSDCone())
[X[1,1] X[1,2]
... X[2,2]] ∈ PSDCone()
```

(Note that no error is thrown, even though X is not symmetric.)

### 13.25 Complementarity constraints

A mixed complementarity constraint  $F(x) \perp x$  consists of finding x in the interval [lb, ub], such that the following holds:

- F(x) == 0 if lb < x < ub
- F(x) >= 0 if lb == x
- F(x) <= 0 if x == ub

JuMP supports mixed complementarity constraints via complements (F(x), x) or  $F(x) \perp x$  in the @constraint macro. The interval set [lb, ub] is obtained from the variable bounds on x.

For example, to define the problem  $2x - 1 \perp x$  with  $x \in [0, \infty)$ , do:

```
julia> model = Model();
julia> @variable(model, x >= 0)
x
julia> @constraint(model, 2x - 1 ⊥ x)
[2 x - 1, x] ∈ MathOptInterface.Complements(2)
```

### CHAPTER 13. CONSTRAINTS

This problem has a unique solution at x = 0.5.

The perp operator  $\perp$  can be entered in most editors (and the Julia REPL) by typing \perp<tab>.

An alternative approach that does not require the  $\perp$  symbol uses the complements function as follows:

```
julia> @constraint(model, complements(2x - 1, x))
[2 x - 1, x] ∈ MathOptInterface.Complements(2)
```

In both cases, the mapping F(x) is supplied as the first argument, and the matching variable x is supplied as the second.

Vector-valued complementarity constraints are also supported:

```
julia> @variable(model, -2 <= y[1:2] <= 2)
2-element Vector{VariableRef}:
 y[1]
 y[2]

julia> M = [1 2; 3 4]
2×2 Matrix{Int64}:
 1 2
 3 4

julia> q = [5, 6]
2-element Vector{Int64}:
 5
 6

julia> @constraint(model, M * y + q ± y)
[y[1] + 2 y[2] + 5, 3 y[1] + 4 y[2] + 6, y[1], y[2]] ∈ MathOptInterface.Complements(4)
```

## 13.26 Boolean constraints

Add a Boolean constraint (a MOI.EqualTo{Bool} set) using the := operator with a Bool right-hand side term:

```
julia> model = GenericModel{Bool}();
julia> @variable(model, x[1:2]);
julia> @constraint(model, x[1] || x[2] := true)
x[1] || x[2] = true
julia> @constraint(model, x[1] && x[2] := false)
x[1] && x[2] = false
julia> model
A JuMP Model
| value_type: Bool
| solver: none
| objective_sense: FEASIBILITY_SENSE
| num_variables: 2
| num_constraints: 2
```

```
| L GenericNonlinearExpr{GenericVariableRef{Bool}} in MOI.EqualTo{Bool}: 2
L Names registered in the model
L :x
```

Boolean constraints should not be added using the == operator because JuMP will rewrite the constraint as lhs - rhs = 0, and because constraints like a == b == c require parentheses to disambiguate between (a == b) == c and a == (b == c). In contrast, a == b := c is equivalent to (a == b) := c:

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> rhs = false
false
julia> @constraint(model, (x[1] == x[2]) == rhs)
(x[1] == x[2]) - 0.0 = 0
julia> @constraint(model, x[1] == x[2] := rhs)
x[1] == x[2] = false
```

# **Chapter 14**

# **Expressions**

JuMP has three types of expressions: affine, quadratic, and nonlinear. These expressions can be inserted into constraints or into the objective. This is particularly useful if an expression is used in multiple places in the model.

# 14.1 Affine expressions

There are four ways of constructing an affine expression in JuMP: with the @expression macro, with operator overloading, with the AffExpr constructor, and with add\_to\_expression!.

#### Macros

The recommended way to create an affine expression is via the @expression macro.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> ex = @expression(model, 2x + y - 1)
2 x + y - 1
```

This expression can be used in the objective or added to a constraint. For example:

```
julia> @objective(model, Min, 2 * ex - 1)
4 x + 2 y - 3
julia> objective_function(model)
4 x + 2 y - 3
```

Just like variables and constraints, named expressions can also be created. For example

```
julia> model = Model();
```

```
julia> @variable(model, x[i = 1:3]);
```

julia> @expression(model, expr[i = 1:3], i \* sum(x[j] for j in i:3));

#### julia> expr

```
3-element Vector{AffExpr}:
x[1] + x[2] + x[3]
2 x[2] + 2 x[3]
3 x[3]
```

### Тір

You can read more about containers in the Containers section.

### **Operator overloading**

Expressions can also be created without macros. However, note that in some cases, this can be much slower that constructing an expression using macros.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> ex = 2x + y - 1
2 x + y - 1
```

#### Constructors

A third way to create an affine expression is by the AffExpr constructor. The first argument is the constant term, and the remaining arguments are variable-coefficient pairs.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> ex = AffExpr(-1.0, x => 2.0, y => 1.0)
2 x + y - 1
```

### add\_to\_expression!

The fourth way to create an affine expression is by using add\_to\_expression!. Compared to the operator overloading method, this approach is faster because it avoids constructing temporary objects. The @expression macro uses add\_to\_expression! behind-the-scenes.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> ex = AffExpr(-1.0)
-1
julia> add_to_expression!(ex, 2.0, x)
2 x - 1
julia> add_to_expression!(ex, 1.0, y)
2 x + y - 1
```

```
add_to_expression! can also be used to sum expressions in-place:
```

```
julia> model = Model();
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
julia> @expression(model, ex1, sum(x))
 x[1] + x[2]
julia> @expression(model, ex2, 2 * sum(x))
2 x[1] + 2 x[2]
julia> add_to_expression!(ex1, ex2)
3 x[1] + 3 x[2]
julia> ex1
3 x[1] + 3 x[2]
julia> ex2
```

#### Warning

 $2 \times [1] + 2 \times [2]$ 

Read the section Initializing arrays for some cases to be careful about when using add\_to\_expression!.

### **Removing zero terms**

Use drop\_zeros! to remove terms from an affine expression with a 0 coefficient.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @expression(model, ex, x + 1 - x)
0 x + 1
julia> drop_zeros!(ex)
julia> ex
```

#### Coefficients

1

Use coefficient to return the coefficient associated with a variable in an affine expression.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> @expression(model, ex, 2x + 1)
2 x + 1
julia> coefficient(ex, x)
2.0
julia> coefficient(ex, y)
0.0
```

### 14.2 Quadratic expressions

Like affine expressions, there are four ways of constructing a quadratic expression in JuMP: macros, operator overloading, constructors, and add\_to\_expression!.

#### Macros

The @expression macro can be used to create quadratic expressions by including quadratic terms.

```
julia> model = Model();
julia> @variable(model, x)
x
```

```
julia> @variable(model, y)
y
julia> ex = @expression(model, x^2 + 2 * x * y + y^2 + x + y - 1)
x² + 2 x*y + y² + x + y - 1
```

### **Operator overloading**

Operator overloading can also be used to create quadratic expressions. The same performance warning (discussed in the affine expression section) applies.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> ex = x^2 + 2 * x * y + y^2 + x + y - 1
x² + 2 x*y + y² + x + y - 1
```

#### Constructors

Quadratic expressions can also be created using the QuadExpr constructor. The first argument is an affine expression, and the remaining arguments are pairs, where the first term is a JuMP.UnorderedPair and the second term is the coefficient.

### add\_to\_expression!

Finally, add\_to\_expression! can also be used to add quadratic terms.

```
julia> model = Model();
```

```
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> ex = QuadExpr(x + y - 1.0)
x + y - 1
julia> add_to_expression!(ex, 1.0, x, x)
x² + x + y - 1
julia> add_to_expression!(ex, 2.0, x, y)
x² + 2 x*y + x + y - 1
```

julia> add\_to\_expression!(ex, 1.0, y, y)
x<sup>2</sup> + 2 x\*y + y<sup>2</sup> + x + y - 1

#### Warning

Read the section Initializing arrays for some cases to be careful about when using add\_to\_expression!.

### **Removing zero terms**

Use drop\_zeros! to remove terms from a quadratic expression with a 0 coefficient.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @expression(model, ex, x^2 + x + 1 - x^2)
0 x² + x + 1
julia> drop_zeros!(ex)
julia> ex
```

x + 1

### Coefficients

Use coefficient to return the coefficient associated with a pair of variables in a quadratic expression.

```
julia> model = Model();
julia> @variable(model, x)
x
```

```
julia> @variable(model, y)
y
julia> @expression(model, ex, 2*x*y + 3*x)
2 x*y + 3 x
julia> coefficient(ex, x, y)
2.0
julia> coefficient(ex, x, x)
0.0
julia> coefficient(ex, y, x)
2.0
julia> coefficient(ex, x, x)
3.0
```

### 14.3 Nonlinear expressions

Nonlinear expressions in JuMP are represented by a NonlinearExpr object. See Nonlinear expressions in detail for more details.

#### 14.4 Initializing arrays

JuMP implements zero(AffExpr) and one(AffExpr) to support various functions in LinearAlgebra (for example, accessing the off-diagonal of a Diagonal matrix).

```
julia> zero(AffExpr)
0
julia> one(AffExpr)
1
```

However, this can result in a subtle bug if you call add\_to\_expression! or the MutableArithmetics API on an element created by zeros or ones:

```
julia> x = zeros(AffExpr, 2)
2-element Vector{AffExpr}:
0
0
julia> add_to_expression!(x[1], 1.1)
1.1
julia> x
2-element Vector{AffExpr}:
1.1
1.1
```

Notice how we modified x[1], but we also changed x[2]!

This happened because zeros(AffExpr, 2) calls zero(AffExpr) once to obtain a zero element, and then creates an appropriately sized array filled with the same element.

This also happens with broadcasting calls containing a conversion of 0 or 1:

```
julia> x = Vector{AffExpr}(undef, 2)
2-element Vector{AffExpr}:
 #undef
julia> x .= 0
2-element Vector{AffExpr}:
 0
 0
julia> add_to_expression!(x[1], 1.1)
1.1
julia> x
2-element Vector{AffExpr}:
 1.1
 1.1
```

The recommended way to create an array of empty expressions is as follows:

Alternatively, use non-mutating operation to avoid updating x[1] in-place:

```
julia> x = zeros(AffExpr, 2)
2-element Vector{AffExpr}:
0
0
julia> x[1] += 1.1
1.1
julia> x
2-element Vector{AffExpr}:
```

# CHAPTER 14. EXPRESSIONS

1.1 0

Note that for large expressions this will be slower due to the allocation of additional temporary objects.

# **Chapter 15**

# **Objectives**

This page describes macros and functions related to linear and quadratic objective functions only, unless otherwise indicated. For nonlinear objective functions, see Nonlinear Modeling.

# 15.1 Set a linear objective

Use the <code>@objective</code> macro to set a linear objective function.

Use Min to create a minimization objective:

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2x + 1)
2 x + 1
```

Use Max to create a maximization objective:

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Max, 2x + 1)
2 x + 1
```

### **15.2 Set a quadratic objective**

Use the @objective macro to set a quadratic objective function.

Use ^2 to have a variable squared:

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, x^2 + 2x + 1)
x² + 2 x + 1
```

You can also have bilinear terms between variables:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> @objective(model, Max, x * y + x + y)
x*y + x + y
```

## 15.3 Set a nonlinear objective

Use the @objective macro to set a nonlinear objective function:

```
julia> model = Model();
julia> @variable(model, x <= 1);
julia> @objective(model, Max, log(x))
log(x)
```

## 15.4 Query the objective function

Use objective\_function to return the current objective function.

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2x + 1)
2 x + 1
julia> objective_function(model)
2 x + 1
```

### 15.5 Evaluate the objective function at a point

Use value to evaluate an objective function at a point specifying values for variables.

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @objective(model, Min, 2x[1]^2 + x[1] + 0.5*x[2])
2 x[1]² + x[1] + 0.5 x[2]
```
```
julia> f = objective_function(model)
2 x[1]² + x[1] + 0.5 x[2]
julia> point = Dict(x[1] => 2.0, x[2] => 1.0);
julia> value(z -> point[z], f)
10.5
```

## 15.6 Query the objective sense

Use objective\_sense to return the current objective sense.

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2x + 1)
2 x + 1
julia> objective_sense(model)
```

MIN\_SENSE::OptimizationSense = 0

## 15.7 Modify an objective

To modify an objective, call <a>[@objective</a> with the new objective function.

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2x)
2 x
julia> @objective(model, Max, -2x)
-2 x
```

Alternatively, use set\_objective\_function.

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2x)
2 x
julia> new_objective = @expression(model, -2 * x)
-2 x
julia> set_objective_function(model, new_objective)
```

## 15.8 Modify an objective coefficient

Use set\_objective\_coefficient to modify an objective coefficient.

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2x)
2 x
julia> set_objective_coefficient(model, x, 3)
julia> objective_function(model)
3 x
```

Use set\_objective\_coefficient with two variables to modify a quadratic objective coefficient:

```
julia> model = Model();
julia> @variable(model, x);
julia> @variable(model, y);
julia> @objective(model, Min, x^2 + x * y)
x² + x*y
julia> set_objective_coefficient(model, x, x, 2)
julia> set_objective_coefficient(model, x, y, 3)
julia> objective_function(model)
2 x² + 3 x*y
```

## 15.9 Modify the objective sense

Use set objective sense to modify the objective sense.

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2x)
2 x
julia> objective_sense(model)
MIN_SENSE::OptimizationSense = 0
julia> set_objective_sense(model, MAX_SENSE);
```

```
julia> objective_sense(model)
MAX_SENSE::OptimizationSense = 1
```

Alternatively, call <a>@objective</a> and pass the existing objective function.

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2x)
2 x
julia> @objective(model, Max, objective_function(model))
2 x
```

#### 15.10 Remove an objective

To remove an objective function use set\_objective\_sense to set the sense to FEASIBILITY\_SENSE:

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2x);
julia> objective_function(model)
2 x
julia> set_objective_sense(model, FEASIBILITY_SENSE)
julia> objective_sense(model)
FEASIBILITY_SENSE::OptimizationSense = 2
julia> objective_function(model)
0
```

#### 15.11 Set a vector-valued objective

Define a multi-objective optimization problem by passing a vector of objectives:

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @objective(model, Min, [1 + x[1], 2 * x[2]])
2-element Vector{AffExpr}:
x[1] + 1
2 x[2]
julia> f = objective_function(model)
2-element Vector{AffExpr}:
x[1] + 1
2 x[2]
```

## Тір

The Multi-objective knapsack tutorial provides an example of solving a multi-objective integer program.

In most cases, multi-objective optimization solvers will return multiple solutions, corresponding to points on the Pareto frontier. See <u>Multiple solutions</u> for information on how to guery and work with multiple solutions.

Note that you must set a single objective sense, that is, you cannot have both minimization and maximization objectives. Work around this limitation by choosing Min and negating any objectives you want to maximize:

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @expression(model, obj1, 1 + x[1])
x[1] + 1
julia> @expression(model, obj2, 2 * x[1])
2 x[1]
julia> @objective(model, Min, [obj1, -obj2])
2-element Vector{AffExpr}:
x[1] + 1
-2 x[1]
```

2-element Vector{AffExpr}:

Defining your objectives as expressions allows flexibility in how you can solve variations of the same problem, with some objectives removed and constrained to be no worse that a fixed value.

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @expression(model, obj1, 1 + x[1])
\times [1] + 1
julia> @expression(model, obj2, 2 * x[1])
2 x[1]
julia> @expression(model, obj3, x[1] + x[2])
x[1] + x[2]
julia> @objective(model, Min, [obj1, obj2, obj3]) # Three-objective problem
3-element Vector{AffExpr}:
x[1] + 1
2 x[1]
x[1] + x[2]
julia> # optimize!(model), look at the solution, talk to stakeholders, then
 # decide you want to solve a new problem where the third objective is
 # removed and constrained to be better than 2.0.
 nothing
julia> @objective(model, Min, [obj1, obj2]) # Two-objective problem
```

## CHAPTER 15. OBJECTIVES

x[1] + 1 2 x[1]

julia> @constraint(model, obj3 <= 2.0)
x[1] + x[2] ≤ 2</pre>

## **Chapter 16**

# Containers

JuMP provides specialized containers similar to AxisArrays that enable multi-dimensional arrays with noninteger indices.

These containers are created automatically by JuMP's macros. Each macro has the same basic syntax:

@macroname(model, name[key1=index1, index2; optional\_condition], other stuff)

The containers are generated by the name[key1=index1, index2; optional\_condition] syntax. Everything else is specific to the particular macro.

Containers can be named, for example, name[key=index], or unnamed, for example, [key=index]. We call unnamed containers *anonymous*.

We call the bits inside the square brackets and before the ; the *index sets*. The index sets can be named, for example, [i = 1:4], or they can be unnamed, for example, [1:4].

We call the bit inside the square brackets and after the ; the condition. Conditions are optional.

In addition to the standard JuMP macros like <code>@variable</code> and <code>@constraint</code>, which construct containers of variables and constraints respectively, you can use <code>Containers.@container</code> to construct containers with arbitrary elements.

We will use this macro to explain the three types of containers that are natively supported by JuMP: Array, Containers.DenseAxisArray, and Containers.SparseAxisArray.

## 16.1 Array

An Array is created when the index sets are rectangular and the index sets are of the form 1:n.

```
julia> Containers.@container(x[i = 1:2, j = 1:3], (i, j))
2×3 Matrix{Tuple{Int64, Int64}}:
 (1, 1) (1, 2) (1, 3)
 (2, 1) (2, 2) (2, 3)
```

The result is a normal Julia Array, so you can do all the usual things.

#### Slicing

Arrays can be sliced

julia> x[:, 1]
2-element Vector{Tuple{Int64, Int64}}:
 (1, 1)
 (2, 1)
julia> x[2, :]
3-element Vector{Tuple{Int64, Int64}}:
 (2, 1)
 (2, 2)
 (2, 3)

## Looping

Use eachindex to loop over the elements:

#### Get the index sets

Use axes to obtain the index sets:

julia> axes(x)
(Base.OneTo(2), Base.OneTo(3))

#### Broadcasting

Broadcasting over an Array returns an Array

```
julia> swap(x::Tuple) = (last(x), first(x))
swap (generic function with 1 method)
julia> swap.(x)
2×3 Matrix{Tuple{Int64, Int64}}:
 (1, 1) (2, 1) (3, 1)
 (1, 2) (2, 2) (3, 2)
```

#### Tables

Use Containers.rowtable to convert the Array into a Tables.jl compatible Vector{<:NamedTuple}:

```
julia> table = Containers.rowtable(x; header = [:I, :J, :value])
6-element Vector{@NamedTuple{I::Int64, J::Int64, value::Tuple{Int64, Int64}}}:
(I = 1, J = 1, value = (1, 1))
(I = 2, J = 1, value = (2, 1))
(I = 1, J = 2, value = (1, 2))
(I = 2, J = 2, value = (2, 2))
(I = 1, J = 3, value = (1, 3))
(I = 2, J = 3, value = (2, 3))
```

Because it supports the Tables.jl interface, you can pass it to any function which accepts a table as input:

```
julia> import DataFrames;
```

#### julia> DataFrames.DataFrame(table)

6×3 DataFrame						
Row	I	J	value			
	Int64	Int64	Tuple…			
1	1	1	(1, 1)			
2	2	1	(2, 1)			
3	1	2	(1, 2)			
4	2	2	(2, 2)			
5	1	3	(1, 3)			
6	2	3	(2, 3)			

#### 16.2 DenseAxisArray

A Containers.DenseAxisArray is created when the index sets are rectangular, but not of the form 1:n. The index sets can be of any type.

```
julia> x = Containers.@container([i = 1:2, j = [:A, :B]], (i, j))
2-dimensional DenseAxisArray{Tuple{Int64, Symbol},2,...} with index sets:
 Dimension 1, Base.OneTo(2)
 Dimension 2, [:A, :B]
And data, a 2×2 Matrix{Tuple{Int64, Symbol}}:
 (1, :A) (1, :B)
 (2, :A) (2, :B)
```

#### Slicing

DenseAxisArrays can be sliced

```
julia> x[:, :A]
1-dimensional DenseAxisArray{Tuple{Int64, Symbol},1,...} with index sets:
 Dimension 1, Base.OneTo(2)
And data, a 2-element Vector{Tuple{Int64, Symbol}}:
 (1, :A)
 (2, :A)
```

julia> x[1, :]

```
1-dimensional DenseAxisArray{Tuple{Int64, Symbol},1,...} with index sets:
 Dimension 1, [:A, :B]
And data, a 2-element Vector{Tuple{Int64, Symbol}}:
 (1, :A)
 (1, :B)
```

## Looping

Use eachindex to loop over the elements:

#### Get the index sets

Use axes to obtain the index sets:

julia> axes(x)
(Base.OneTo(2), [:A, :B])

#### Broadcasting

Broadcasting over a DenseAxisArray returns a DenseAxisArray

```
julia> swap(x::Tuple) = (last(x), first(x))
swap (generic function with 1 method)
julia> swap.(x)
2-dimensional DenseAxisArray{Tuple{Symbol, Int64},2,...} with index sets:
 Dimension 1, Base.OneTo(2)
 Dimension 2, [:A, :B]
And data, a 2×2 Matrix{Tuple{Symbol, Int64}}:
 (:A, 1) (:B, 1)
 (:A, 2) (:B, 2)
```

#### Access internal data

Use Array(x) to copy the internal data array into a new Array:

```
julia> Array(x)
2×2 Matrix{Tuple{Int64, Symbol}}:
 (1, :A) (1, :B)
 (2, :A) (2, :B)
```

To access the internal data without a copy, use x.data.

```
julia> x.data
2x2 Matrix{Tuple{Int64, Symbol}}:
 (1, :A) (1, :B)
 (2, :A) (2, :B)
```

#### Tables

Use Containers.rowtable to convert the DenseAxisArray into a Tables.jl compatible Vector{<:NamedTuple}:

```
julia> table = Containers.rowtable(x; header = [:I, :J, :value])
4-element Vector{@NamedTuple{I::Int64, J::Symbol, value::Tuple{Int64, Symbol}}:
 (I = 1, J = :A, value = (1, :A))
 (I = 2, J = :A, value = (2, :A))
 (I = 1, J = :B, value = (1, :B))
 (I = 2, J = :B, value = (2, :B))
```

Because it supports the Tables.jl interface, you can pass it to any function which accepts a table as input:

julia> import DataFrames;

```
julia> DataFrames.DataFrame(table)
```

4×3 DataFrame

Row	I	J	valı	le
	Int64	Symbol	Tup	le
1	1	A	(1,	:A)
2	2	A	(2,	:A)
3	1	В	(1,	:B)
4	2	В	(2,	:B)

#### **Keyword indexing**

If all axes are named, you can use keyword indexing:

```
julia> x[i = 2, j = :A]
(2, :A)
julia> x[i = :, j = :B]
1-dimensional DenseAxisArray{Tuple{Int64, Symbol},1,...} with index sets:
 Dimension 1, Base.OneTo(2)
And data, a 2-element Vector{Tuple{Int64, Symbol}}:
 (1, :B)
 (2, :B)
```

#### 16.3 SparseAxisArray

A Containers.SparseAxisArray is created when the index sets are non-rectangular. This occurs in two circumstances:

An index depends on a prior index:

```
julia> Containers.@container([i = 1:2, j = i:2], (i, j))
JuMP.Containers.SparseAxisArray{Tuple{Int64, Int64}, 2, Tuple{Int64, Int64}} with 3 entries:
 [1, 1] = (1, 1)
 [1, 2] = (1, 2)
 [2, 2] = (2, 2)
```

The [indices; condition] syntax is used:

```
julia> x = Containers.@container([i = 1:3, j = [:A, :B]; i > 1], (i, j))
JuMP.Containers.SparseAxisArray{Tuple{Int64, Symbol}, 2, Tuple{Int64, Symbol}} with 4 entries:
 [2, A] = (2, :A)
 [2, B] = (2, :B)
 [3, A] = (3, :A)
 [3, B] = (3, :B)
```

Here we have the index sets i = 1:3, j = [:A, :B], followed by ;, and then a condition, which evaluates to true or false: i > 1.

#### Slicing

Slicing is supported:

```
julia> y = x[:, :B]
JuMP.Containers.SparseAxisArray{Tuple{Int64, Symbol}, 1, Tuple{Int64}} with 2 entries:
[2] = (2, :B)
[3] = (3, :B)
```

#### Looping

Use eachindex to loop over the elements:

#### Warning

If you use a macro to construct a SparseAxisArray, then the iteration order is row-major, that is, indices are varied from right to left. As an example, when iterating over x above, the j index is iterated, keeping i constant. This order is in contrast to Base.Arrays, which iterate in column-major order, that is, by varying indices from left to right.

#### Broadcasting

Broadcasting over a SparseAxisArray returns a SparseAxisArray

```
julia> swap(x::Tuple) = (last(x), first(x))
swap (generic function with 1 method)
```

```
julia> swap.(y)
```

JuMP.Containers.SparseAxisArray{Tuple{Symbol, Int64}, 1, Tuple{Int64}} with 2 entries:
[2] = (:B, 2)
[3] = (:B, 3)

#### Tables

Use Containers.rowtable to convert the SparseAxisArray into a Tables.jl compatible Vector{<: NamedTuple}:

```
julia> table = Containers.rowtable(x; header = [:I, :J, :value])
4-element Vector{@NamedTuple{I::Int64, J::Symbol, value::Tuple{Int64, Symbol}}:
 (I = 2, J = :A, value = (2, :A))
 (I = 2, J = :B, value = (2, :B))
 (I = 3, J = :A, value = (3, :A))
 (I = 3, J = :B, value = (3, :B))
```

Because it supports the Tables.jl interface, you can pass it to any function which accepts a table as input:

```
julia> import DataFrames;
```

#### julia> DataFrames.DataFrame(table)

4×3 Da	ataFrame		
Row	I	J	value
	Int64	Symbol	Tuple…
1	2	A	(2, :A)
2	2	В	(2, :B)
3	3	A	(3, :A)
4	3	В	(3. :B)

#### **Keyword indexing**

If all axes are named, you can use keyword indexing:

```
julia> x[i = 2, j = :A]
(2, :A)
julia> x[i = :, j = :B]
JuMP.Containers.SparseAxisArray{Tuple{Int64, Symbol}, 1, Tuple{Int64}} with 2 entries:
 [2] = (2, :B)
 [3] = (3, :B)
```

## 16.4 Forcing the container type

```
Pass container = T to use T as the container. For example:
```

```
julia> Containers.@container([i = 1:2, j = 1:2], i + j, container = Array)
2×2 Matrix{Int64}:
2 3
3 4

julia> Containers.@container([i = 1:2, j = 1:2], i + j, container = Dict)
Dict{Tuple{Int64, Int64}, Int64} with 4 entries:
 (1, 2) => 3
 (1, 1) => 2
 (2, 2) => 4
 (2, 1) => 3
```

You can also pass DenseAxisArray or SparseAxisArray.

## 16.5 How different container types are chosen

If the compiler can prove at compile time that the index sets are rectangular, and indexed by a compact set of integers that start at 1, Containers.@container will return an array. This is the case if your index sets are visible to the macro as 1:n:

```
julia> Containers.@container([i=1:3, j=1:5], i + j)
3×5 Matrix{Int64}:
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

or an instance of Base.OneTo:

```
julia> set = Base.OneTo(3)
Base.OneTo(3)
julia> Containers.@container([i=set, j=1:5], i + j)
3×5 Matrix{Int64}:
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

If the compiler can prove that the index set is rectangular, but not necessarily of the form 1:n at compile time, then a Containers.DenseAxisArray will be constructed instead:

```
julia> set = 1:3
1:3
```

```
julia> Containers.@container([i=set, j=1:5], i + j)
```

```
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
 Dimension 1, 1:3
 Dimension 2, Base.OneTo(5)
And data, a 3×5 Matrix{Int64}:
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

#### Info

What happened here? Although we know that set contains 1:3, at compile time the typeof(set) is a UnitRange{Int}. Therefore, Julia can't prove that the range starts at 1 (it only finds this out at runtime), and it defaults to a DenseAxisArray. The case where we explicitly wrote i = 1:3 worked because the macro can "see" the 1 at compile time.

However, if you know that the indices do form an Array, you can force the container type with container = Array:

```
julia> set = 1:3
1:3
julia> Containers.@container([i=set, j=1:5], i + j, container = Array)
3×5 Matrix{Int64}:
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

Here's another example with something similar:

```
julia> a = 1
1
julia> Containers.@container([i=a:3, j=1:5], i + j)
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
 Dimension 1, 1:3
 Dimension 2, Base.OneTo(5)
And data, a 3×5 Matrix{Int64}:
 2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
julia> Containers.@container([i=1:a, j=1:5], i + j)
1×5 Matrix{Int64}:
 2 3 4 5 6
```

CHAPTER 16. CONTAINERS

Finally, if the compiler cannot prove that the index set is rectangular, a Containers.SparseAxisArray will be created.

This occurs when some indices depend on a previous one:

```
julia> Containers.@container([i=1:3, j=1:i], i + j)
JuMP.Containers.SparseAxisArray{Int64, 2, Tuple{Int64, Int64}} with 6 entries:
 [1, 1] = 2
 [2, 1] = 3
 [2, 2] = 4
 [3, 1] = 4
 [3, 2] = 5
 [3, 3] = 6
```

or if there is a condition on the index sets:

```
julia> Containers.@container([i = 1:5; isodd(i)], i^2)
JuMP.Containers.SparseAxisArray{Int64, 1, Tuple{Int64}} with 3 entries:
[1] = 1
[3] = 9
[5] = 25
```

The condition can depend on multiple indices, the only requirement is that it is an expression that returns true or false:

```
julia> condition(i, j) = isodd(i) && iseven(j)
condition (generic function with 1 method)
julia> Containers.@container([i = 1:2, j = 1:4; condition(i, j)], i + j)
JuMP.Containers.SparseAxisArray{Int64, 2, Tuple{Int64, Int64}} with 2 entries:
[1, 2] = 3
[1, 4] = 5
```

## **Chapter 17**

## Solutions

This section of the manual describes how to access a solved solution to a problem. It uses the following model as an example:

```
julia> begin
```

```
model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, x >= 0)
 @variable(model, y[[:a, :b]] <= 1)</pre>
 @objective(model, Max, -12x - 20y[:a])
 @expression(model, my_expr, 6x + 8y[:a])
 @constraint(model, my_expr >= 100)
 @constraint(model, c1, 7x + 12y[:a] >= 120)
 optimize!(model)
 print(model)
 end
Max -12 x - 20 y[a]
Subject to
6 \times + 8 y[a] \ge 100
c1 : 7 \times + 12 y[a] \ge 120
X \ge 0
y[a] ≤ 1
y[b] \leq 1
```

#### 17.1 Check if an optimal solution exists

Use is\_solved\_and\_feasible to check if the solver found an optimal solution:

```
julia> is_solved_and_feasible(model)
true
```

By default, is\_solved\_and\_feasible returns true for both global and local optima. Pass allow\_local = false to check if the solver found a globally optimal solution:

```
julia> is_solved_and_feasible(model; allow_local = false)
true
```

Pass dual = true to check if the solver found an optimal dual solution in addition to an optimal primal solution:

```
julia> is_solved_and_feasible(model; dual = true)
true
```

If this function returns false, use the functions mentioned below like solution\_summary, termination\_status, primal\_status, and dual\_status to understand what solution (if any) the solver found.

## 17.2 Solutions summary

solution\_summary can be used for checking the summary of the optimization solutions.

```
julia> solution summary(model)
solution_summary(; result = 1, verbose = false)
- solver_name : HiGHS
- Termination
| | termination_status : OPTIMAL
| | result_count : 1
| | raw_status : kHighsModelStatusOptimal
 L objective bound : -2.05143e+02
- Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | objective_value : -2.05143e+02
| | dual_objective_value : -2.05143e+02
L relative_gap : 1.38546e-16
L Work counters
 - solve_time (sec) : 5.93345e-04
 - simplex_iterations : 2
 barrier_iterations : 0
 L node_count : -1
julia> solution summary(model; verbose = true)
solution_summary(; result = 1, verbose = true)
├ solver_name : HiGHS
- Termination
: kHighsModelStatusOptimal
L objective_bound : -2.05143e+02
- Solution (result = 1)
| | primal_status : FEASIBLE_POINT
| | dual_status : FEASIBLE_POINT
| | objective_value : -2.05143e+02
| - dual objective value : -2.05143e+02
| | value
| | - x : 1.54286e+01
| | | y[a] : 1.00000e+00
| | ^L y[b] : 1.00000e+00
| ^L dual
 ^L c1 : 1.71429e+00
^L Work counters
 - solve_time (sec) : 5.93345e-04
```

```
| simplex_iterations : 2
| barrier_iterations : 0
L node_count : -1
```

## 17.3 Why did the solver stop?

Usetermination\_status to understand why the solver stopped.

```
julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1
```

The MOI.TerminationStatusCode enum describes the full list of statuses that could be returned.

Common return values include OPTIMAL, LOCALLY\_SOLVED, INFEASIBLE, DUAL\_INFEASIBLE, and TIME\_LIMIT.

Info

A return status of OPTIMAL means the solver found (and proved) a globally optimal solution. A return status of LOCALLY\_SOLVED means the solver found a locally optimal solution (which may also be globally optimal, but it could not prove so).

#### Warning

A return status of DUAL\_INFEASIBLE does not guarantee that the primal is unbounded. When the dual is infeasible, the primal is unbounded if there exists a feasible primal solution.

Use raw status to get a solver-specific string explaining why the optimization stopped:

```
julia> raw_status(model)
"kHighsModelStatusOptimal"
```

## 17.4 Primal solutions

#### **Primal solution status**

Use primal\_status to return an MOI.ResultStatusCode enum describing the status of the primal solution.

```
julia> primal_status(model)
FEASIBLE POINT::ResultStatusCode = 1
```

Other common returns are NO\_SOLUTION, and INFEASIBILITY\_CERTIFICATE. The first means that the solver doesn't have a solution to return, and the second means that the primal solution is a certificate of dual infeasibility (a primal unbounded ray).

## **Objective values**

The objective value of a solved problem can be obtained via objective\_value. The best known bound on the optimal objective value can be obtained via objective\_bound. If the solver supports it, the value of the dual objective can be obtained via dual\_objective\_value.

julia> objective\_value(model)
-205.14285714285714

julia> objective\_bound(model)
-205.1428571428571

julia> dual\_objective\_value(model)
-205.1428571428571

## **Primal solution values**

If the solver has a primal solution to return, use value to access it:

julia> value(x)
15.428571428571429

#### Broadcast value over containers:

```
julia> value.(y)
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
 Dimension 1, [:a, :b]
And data, a 2-element Vector{Float64}:
 1.0
 1.0
```

value also works on expressions:

```
julia> value(my_expr)
100.57142857142857
```

#### and constraints:

julia> value(c1)
120.0

#### Info

Calling value on a constraint returns the constraint function evaluated at the solution.

#### **Unbounded rays**

If the termination\_status is DUAL\_INFEASIBLE and primal\_status is INFEASIBILITY\_CERTIFICATE, then the value is a certificate of dual infeasibility. If a feasible primal solution exists, the certificate is an unbounded ray of the primal problem. The objective\_value is the value of the objective, evaluated using the ray, excluding any constant term. For more details, see Infeasibility certificates.

```
julia> using HiGHS
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> set_attribute(model, "presolve", "off")
julia> @variable(model, x[1:2]);
julia> @objective(model, Max, sum(x) + 2.0);
julia> @constraint(model, 2 * x[1] + x[2] <= 0);</pre>
julia> optimize!(model)
julia> termination_status(model)
DUAL INFEASIBLE::TerminationStatusCode = 3
julia> primal_status(model)
INFEASIBILITY_CERTIFICATE::ResultStatusCode = 4
julia> d = value.(x)
2-element Vector{Float64}:
-0.5
 1.0
```

julia> objective\_value(model)
0.5

#### 17.5 Dual solutions

#### **Dual solution status**

Use dual\_status to return an MOI.ResultStatusCode enum describing the status of the dual solution.

```
julia> dual_status(model)
FEASIBLE_POINT::ResultStatusCode = 1
```

Other common returns are NO\_SOLUTION, and INFEASIBILITY\_CERTIFICATE. The first means that the solver doesn't have a solution to return, and the second means that the dual solution is a certificate of primal infeasibility (a dual unbounded ray).

#### **Dual solution values**

If the solver has a dual solution to return, use dual to access it:

#### CHAPTER 17. SOLUTIONS

# julia> dual(c1) 1.7142857142857142

Query the duals of variable bounds using LowerBoundRef, UpperBoundRef, and FixRef:

```
julia> dual(LowerBoundRef(x))
0.0
julia> dual.(UpperBoundRef.(y))
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
 Dimension 1, [:a, :b]
And data, a 2-element Vector{Float64}:
 -0.5714285714285694
 0.0
```

#### Warning

JuMP's definition of duality is independent of the objective sense. That is, the sign of feasible duals associated with a constraint depends on the direction of the constraint and not whether the problem is maximization or minimization. **This is a different convention from linear programming duality in some common textbooks.** If you have a linear program, and you want the textbook definition, you probably want to use shadow\_price and reduced\_cost instead.

```
julia> shadow_price(c1)
1.7142857142857142
```

```
julia> reduced_cost(x)
-0.0
```

```
julia> reduced_cost.(y)
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
 Dimension 1, [:a, :b]
And data, a 2-element Vector{Float64}:
 0.5714285714285694
 -0.0
```

#### **Unbounded rays**

If the termination\_status is INFEASIBLE and dual\_status is INFEASIBILITY\_CERTIFICATE, then the dual is a certificate of primal infeasibility. If a feasible dual solution exists, the certificate is an unbounded ray of the dual problem. The dual\_objective\_value is the value of the dual objective, evaluated using the ray, excluding any constant term. For more details, see Infeasibility certificates.

```
julia> using HiGHS
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
```

```
julia> set_attribute(model, "presolve", "off")
julia> @variable(model, x[1:2] >= 0);
julia> @objective(model, Max, sum(x) + 0.5);
julia> @constraint(model, c, 2 * x[1] + x[2] <= -1);</pre>
julia> optimize!(model)
julia> termination_status(model)
INFEASIBLE::TerminationStatusCode = 2
julia> dual_status(model)
INFEASIBILITY_CERTIFICATE::ResultStatusCode = 4
julia> d_c = dual(c)
-1.0
julia> d_x = dual.(LowerBoundRef.(x))
2-element Vector{Float64}:
2.0
1.0
julia> dual_objective_value(model)
```

```
-1.0
```

#### 17.6 Recommended workflow

You should always check whether the solver found a solution before calling solution functions like value or objective\_value.

A simple approach for small scripts and notebooks is to use is\_solved\_and\_feasible:

```
julia> function solve_and_print_solution(model)
 optimize!(model)
 if !is_solved_and_feasible(model; dual = true)
 error(

 The model was not solved correctly:
 termination_status : $(termination_status(model))
 primal_status : $(primal_status(model))
 dual_status : $(dual_status(model))
raw_status : $(raw_status(model))
 ····,
)
 end
 println("Solution is optimal")
 println(" objective value = ", objective_value(model))
 println(" primal solution: x = ", value(x))
 println(" dual solution: c1 = ", dual(c1))
 return
 end
solve_and_print_solution (generic function with 1 method)
```

```
julia> solve_and_print_solution(model)
Solution is optimal
 objective value = -205.14285714285714
 primal solution: x = 15.428571428571429
 dual solution: c1 = 1.7142857142857142
```

For code like libraries that should be more robust to the range of possible termination and result statuses, do some variation of the following:

```
julia> function solve and print solution(model)
 optimize!(model)
 status = termination_status(model)
 if status in (OPTIMAL, LOCALLY_SOLVED)
 println("Solution is optimal")
 elseif status in (ALMOST_OPTIMAL, ALMOST_LOCALLY_SOLVED)
 println("Solution is optimal to a relaxed tolerance")
 elseif status == TIME LIMIT
 println(
 "Solver stopped due to a time limit. If a solution is available, " \ast
 "it may be suboptimal."
)
 elseif status in (
 ITERATION LIMIT, NODE LIMIT, SOLUTION LIMIT, MEMORY LIMIT,
 OBJECTIVE LIMIT, NORM LIMIT, OTHER LIMIT,
)
 println(
 "Solver stopped due to a limit. If a solution is available, it " *
 "may be suboptimal."
)
 elseif status in (INFEASIBLE, LOCALLY INFEASIBLE)
 println("The problem is primal infeasible")
 elseif status == DUAL_INFEASIBLE
 println(
 "The problem is dual infeasible. If a primal feasible solution " *
 "exists, the problem is unbounded. To check, set the objective " *
 "to `@objective(model, Min, 0)` and re-solve. If the problem is " *
 "feasible, the primal is unbounded. If the problem is " *
 "infeasible, both the primal and dual are infeasible.",
)
 elseif status == INFEASIBLE OR UNBOUNDED
 println(
 "The model is either infeasible or unbounded. Set the objective " *
 "to `@objective(model, Min, 0)` and re-solve to disambiguate. If " *
 "the problem was infeasible, it will still be infeasible. If the " st
 "problem was unbounded, it will now have a finite optimal solution.",
)
 else
 println(
 "The model was not solved correctly. The termination status is $status",
)
 end
 if primal_status(model) in (FEASIBLE_POINT, NEARLY_FEASIBLE_POINT)
 println(" objective value = ", objective_value(model))
```

```
println(" primal solution: x = ", value(x))
elseif primal_status(model) == INFEASIBILITY_CERTIFICATE
 println(" primal certificate: x = ", value(x))
end
if dual_status(model) in (FEASIBLE_POINT, NEARLY_FEASIBLE_POINT)
 println(" dual solution: cl = ", dual(cl))
elseif dual_status(model) == INFEASIBILITY_CERTIFICATE
 println(" dual certificate: cl = ", dual(cl))
end
return
end
```

solve\_and\_print\_solution (generic function with 1 method)

```
julia> solve_and_print_solution(model)
```

```
Solution is optimal
 objective value = -205.14285714285714
 primal solution: x = 15.428571428571429
 dual solution: c1 = 1.7142857142857142
```

## 17.7 OptimizeNotCalled errors

Due to differences in how solvers cache solutions internally, modifying a model after calling optimize! will reset the model into the OPTIMIZE\_NOT\_CALLED state. If you then attempt to query solution information, an OptimizeNotCalled error will be thrown.

If you are iteratively querying solution information and modifying a model, query all the results first, then modify the problem.

For example, instead of:

```
julia> model = Model(HiGHS.Optimizer);
```

julia> set\_silent(model)

julia> @variable(model, x >= 0);

julia> optimize!(model)

julia> termination\_status(model)
OPTIMAL::TerminationStatusCode = 1

```
julia> set_upper_bound(x, 1)
```

```
julia> x_val = value(x)
```

Warning: The model has been modified since the last call to `optimize!` (or `optimize!` has not
 been called yet). If you are iteratively querying solution information and modifying a model,
 query all the results first, then modify the model.
 @ JuMP ~/.julia/dev/JuMP/src/optimizer\_interface.jl:1085
ERROR: OptimizeNotCalled()
Stacktrace:
[...]

julia> termination\_status(model)
OPTIMIZE\_NOT\_CALLED::TerminationStatusCode = 0

do

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x >= 0);
julia> optimize!(model);
julia> x_val = value(x)
0.0
julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1
julia> set_upper_bound(x, 1)
julia> set_lower_bound(x, x_val)
julia> termination_status(model)
OPTIMIZE_NOT_CALLED::TerminationStatusCode = 0
```

If you know that your particular solver supports querying solution information after modifications, you can use direct\_model to bypass the OPTIMIZE\_NOT\_CALLED state:

```
julia> model = direct_model(HiGHS.Optimizer());
julia> set_silent(model)
julia> @variable(model, x >= 0);
julia> optimize!(model)
julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1
julia> set_upper_bound(x, 1)
julia> x_val = value(x)
0.0
julia> set_lower_bound(x, x_val)
julia> termination_status(model)
```

#### Warning

OPTIMAL::TerminationStatusCode = 1

Be careful doing this. If your particular solver does not support querying solution information after modification, it may silently return incorrect solutions or throw an error.

#### 17.8 Accessing attributes

MathOptInterface defines many model attributes that can be queried. Some attributes can be directly accessed by getter functions. These include:

- solve\_time
- relative\_gap
- simplex\_iterations
- barrier\_iterations
- node\_count

## 17.9 Sensitivity analysis for LP

Given an LP problem and an optimal solution corresponding to a basis, we can question how much an objective coefficient or standard form right-hand side coefficient (c.f., normalized\_rhs) can change without violating primal or dual feasibility of the basic solution.

Note that not all solvers compute the basis, and for sensitivity analysis, the solver interface must implement MOI.ConstraintBasisStatus.

Tip Read the Sensitivity analysis of a linear program for more information on sensitivity analysis.

To give a simple example, we could analyze the sensitivity of the optimal solution to the following (nondegenerate) LP problem:

```
julia> begin
```

```
model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, x[1:2])
 set_lower_bound(x[2], -0.5)
 set_upper_bound(x[2], 0.5)
 @constraint(model, c1, x[1] + x[2] <= 1)</pre>
 @constraint(model, c2, x[1] - x[2] <= 1)</pre>
 @objective(model, Max, x[1])
 print(model)
 end
Max x[1]
Subject to
c1 : x[1] + x[2] \le 1
c2 : x[1] - x[2] \le 1
x[2] \ge -0.5
x[2] \le 0.5
```

To analyze the sensitivity of the problem we could check the allowed perturbation ranges of, for example, the cost coefficients and the right-hand side coefficient of the constraint c1 as follows:

julia> optimize!(model)

```
julia> value.(x)
2-element Vector{Float64}:
 1.0
 -0.0
```

julia> report = lp\_sensitivity\_report(model);

julia> x1\_lo, x1\_hi = report[x[1]]
(-1.0, Inf)

julia> println("The objective coefficient of x[1] could decrease by  $(x1_lo)$  or increase by  $\Rightarrow (x1_hi)$ .")

The objective coefficient of x[1] could decrease by -1.0 or increase by Inf.

julia> x2\_lo, x2\_hi = report[x[2]]
(-1.0, 1.0)

```
julia> println("The objective coefficient of x[2] could decrease by $(x2_lo) or increase by

\Rightarrow $(x2_hi).")

The objective coefficient of x[2] could decrease by -1.0 or increase by 1.0.
```

julia> c\_lo, c\_hi = report[c1]
(-1.0, 1.0)

```
julia> println("The RHS of cl could decrease by $(c_lo) or increase by $(c_hi).")
The RHS of cl could decrease by -1.0 or increase by 1.0.
```

The range associated with a variable is the range of the allowed perturbation of the corresponding objective coefficient. Note that the current primal solution remains optimal within this range; however the corresponding dual solution might change since a cost coefficient is perturbed. Similarly, the range associated with a constraint is the range of the allowed perturbation of the corresponding right-hand side coefficient. In this range the current dual solution remains optimal, but the optimal primal solution might change.

If the problem is degenerate, there are multiple optimal bases and hence these ranges might not be as intuitive and seem too narrow, for example, a larger cost coefficient perturbation might not invalidate the optimality of the current primal solution. Moreover, if a problem is degenerate, due to finite precision, it can happen that, for example, a perturbation seems to invalidate a basis even though it doesn't (again providing too narrow ranges). To prevent this, increase the atol keyword argument to lp\_sensitivity\_report. Note that this might make the ranges too wide for numerically challenging instances. Thus, do not blindly trust these ranges, especially not for highly degenerate or numerically unstable instances.

## 17.10 Conflicts

When the model you input is infeasible, some solvers can help you find the cause of this infeasibility by offering a conflict, that is, a subset of the constraints that create this infeasibility. Depending on the solver, this can also be called an IIS (irreducible inconsistent subsystem).

If supported by the solver, use compute\_conflict! to trigger the computation of a conflict. Once this process is finished, query the MOI.ConflictStatus attribute to check if a conflict was found.

If found, copy the IIS to a new model using copy\_conflict, which you can then print or write to a file for easier debugging:

```
julia> using JuMP
```

```
julia> import Gurobi
julia> model = Model(Gurobi.Optimizer);
julia> set_silent(model)
julia> @variable(model, x >= 0)
х
julia> @constraint(model, c1, x >= 2)
c1 : x \ge 2.0
julia> @constraint(model, c2, x <= 1)</pre>
c2 : x \leq 1.0
julia> optimize!(model)
julia> compute_conflict!(model)
julia> if get_attribute(model, MOI.ConflictStatus()) == MOI.CONFLICT_FOUND
 iis_model, _ = copy_conflict(model)
 print(iis_model)
 end
Feasibility
Subject to
c1 : x \ge 2.0
c2 \ : \ x \ \le \ 1.0
```

If you need more control over the list of constraints that appear in the conflict, iterate over the list of constraints and query the MOI.ConstraintConflictStatus attribute:

#### 17.11 Multiple solutions

Some solvers support returning multiple solutions. You can check how many solutions are available to query using result\_count.

Functions for querying the solutions, for example, primal\_status, dual\_status, value, dual, and solution\_summary all take an additional keyword argument result which can be used to specify which result to return.

```
Warning
```

Even if termination\_status is OPTIMAL, some of the returned solutions may be suboptimal. However, if the solver found at least one optimal solution, then result = 1 will always return an optimal solution. Use objective\_value to assess the quality of the remaining solutions.

```
julia> using JuMP
```

julia> import MultiObjectiveAlgorithms as MOA

```
julia> import HiGHS
```

```
julia> model = Model(() -> MOA.Optimizer(HiGHS.Optimizer));
```

```
julia> set_attribute(model, MOA.Algorithm(), MOA.Dichotomy())
```

```
julia> set_silent(model)
```

```
julia> @variable(model, x1 >= 0)
x1
```

```
julia> @variable(model, 0 <= x2 <= 3)
x2</pre>
```

```
julia> @objective(model, Min, [3x1 + x2, -x1 - 2x2])
2-element Vector{AffExpr}:
 3 x1 + x2
```

```
-x1 - 2 x2
```

```
julia> @constraint(model, 3x1 - x2 <= 6)
3 x1 - x2 ≤ 6</pre>
```

```
julia> optimize!(model)
```

```
julia> solution_summary(model; result = 1)
```

```
solution_summary(; result = 1, verbose = false)
+ solver_name : MOA[algorithm=MultiObjectiveAlgorithms.Dichotomy, optimizer=HiGHS]
+ Termination
| + termination_status : OPTIMAL
| + result_count : 3
| + raw_status : Solve complete. Found 3 solution(s)
| L objective_bound : [0.00000e+00,-9.00000e+00]
+ Solution (result = 1)
| + primal_status : FEASIBLE_POINT
| + dual_status : NO_SOLUTION
| L objective_value : [0.00000e+00,0.00000e+00]
L Work counters
L solve_time (sec) : 1.88589e-03
```

```
println(" obj = ", objective_value(model; result = i))
end
Solution 1
x = [0.0, 0.0]
obj = [0.0, 0.0]
Solution 2
x = [0.0, 3.0]
obj = [3.0, -6.0]
Solution 3
x = [3.0, 3.0]
obj = [12.0, -9.0]
```

#### Тір

The Multi-objective knapsack tutorial provides more examples of querying multiple solutions.

### 17.12 Checking feasibility of solutions

To check the feasibility of a primal solution, use primal\_feasibility\_report, which takes a model, a dictionary mapping each variable to a primal solution value (defaults to the last solved solution), and a tolerance atol (defaults to 0.0).

The function returns a dictionary which maps the infeasible constraint references to the distance between the primal value of the constraint and the nearest point in the corresponding set. A point is classed as infeasible if the distance is greater than the supplied tolerance atol.

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x >= 1, Int);
julia> @variable(model, y);
julia> @constraint(model, c1, x + y <= 1.95);
julia> point = Dict(x => 1.9, y => 0.06);
julia> primal_feasibility_report(model, point)
Dict{Any, Float64} with 2 entries:
 x integer => 0.1
 cl : x + y ≤ 1.95 => 0.01
julia> primal_feasibility_report(model, point; atol = 0.02)
Dict{Any, Float64} with 1 entry:
 x integer => 0.1
```

If the point is feasible, an empty dictionary is returned:

```
julia> primal_feasibility_report(model, Dict(x => 1.0, y => 0.0))
Dict{Any, Float64}()
```

#### CHAPTER 17. SOLUTIONS

To use the primal solution from a solve, omit the point argument:

```
julia> optimize!(model)
```

```
julia> primal_feasibility_report(model; atol = 0.0)
Dict{Any, Float64}()
```

Calling primal\_feasibility\_report without the point argument is useful when primal\_status is FEASIBLE\_POINT or NEARLY\_FEASIBLE\_POINT, and you want to assess the solution quality.

Warning

To apply primal\_feasibility\_report to infeasible models, you must also provide a candidate point (solvers generally do not provide one). To diagnose the source of infeasibility, see Conflicts.

Pass skip\_mising = true to skip constraints which contain variables that are not in point:

```
julia> primal_feasibility_report(model, Dict(x => 2.1); skip_missing = true)
Dict{Any, Float64} with 1 entry:
 x integer => 0.1
```

You can also use the functional form, where the first argument is a function that maps variables to their primal values:

#### julia> optimize!(model)

```
julia> primal_feasibility_report(v -> value(v), model)
Dict{Any, Float64}()
```

## **Chapter 18**

# Solver-independent Callbacks

Some mixed-integer (linear, conic, and nonlinear) programming solvers offer the ability to modify the solve process. Examples include changing branching decisions in branch-and-bound, adding custom cutting planes, providing custom heuristics to find feasible solutions, or implementing on-demand separators to add new constraints only when they are violated by the current solution (also known as lazy constraints).

While historically this functionality has been limited to solver-specific interfaces, JuMP provides solver-independent support for three types of callbacks:

- 1. lazy constraints
- 2. user-cuts
- 3. heuristic solutions

Even though JuMP provides a solver-independent way of writing these three callbacks, you should not assume that you will see identical behavior when running the same code on different solvers. For example, some solvers may ignore user-cuts for various reasons, while other solvers may add every user-cut. Read the underlying solver's callback documentation to understand details specific to each solver.

## 18.1 Available solvers

Solver-independent callback support is limited to a few solvers, including:

- CPLEX
- GLPK
- Gurobi
- SCIP (SCIP does not support lazy constraints).
- Xpress

Each solver listed above also provides a solver-*dependent* callback to provide access to the full range of solverspecific features. Consult the solver's README for an example of how to use the solver-dependent callback. This will require you to understand the C interface of the solver.

#### 18.2 Things you can and cannot do during solver-independent callbacks

There is a limited range of things you can do during a callback. Only use the functions and macros explicitly stated in this page of the documentation, or in the Callbacks tutorial.

Using any other part of the JuMP API (for example, adding a constraint with @constraint or modifying a variable bound with set\_lower\_bound) is undefined behavior, and your solver may throw an error, return an incorrect solution, or result in a segfault that aborts Julia.

In each of the three solver-independent callbacks, there are two things you may query:

- callback\_node\_status returns an MOI.CallbackNodeStatusCode enum indicating if the current primal solution is integer feasible.
- callback\_value returns the current primal solution of a variable.

If you need to query any other information, use a solver-dependent callback instead. Each solver supporting a solver-dependent callback has information on how to use it in the README of their GitHub repository.

You can only set each callback once. Calling set\_attribute twice will over-write the earlier callback. In addition, if you use a solver-independent callback, you cannot set a solver-dependent callback.

#### 18.3 Lazy constraints

Lazy constraints are useful when the full set of constraints is too large to explicitly include in the initial formulation. When a MIP solver reaches a new solution, for example with a heuristic or by solving a problem at a node in the branch-and-bound tree, it will give the user the chance to provide constraints that would make the current solution infeasible. For some more information about lazy constraints, see this blog post by Paul Rubin.

A lazy constraint callback can be set using the following syntax:

```
julia> model = Model();
julia> @variable(model, x <= 10, Int)</pre>
Х
julia> @objective(model, Max, x)
julia> function my_callback_function(cb_data)
 status = callback_node_status(cb_data, model)
 if status == MOI.CALLBACK_NODE_STATUS_FRACTIONAL
 # `callback_value(cb_data, x)` is not integer (to some tolerance).
 # If, for example, your lazy constraint generator requires an
 # integer-feasible primal solution, you can add a `return` here.
 return
 elseif status == MOI.CALLBACK NODE STATUS INTEGER
 # `callback_value(cb_data, x)` is integer (to some tolerance).
 else
 @assert status == MOI.CALLBACK_NODE_STATUS_UNKNOWN
 # `callback_value(cb_data, x)` might be fractional or integer.
 end
 x_val = callback_value(cb_data, x)
 if x_val > 2 + 1e-6
```

```
con = @build_constraint(x <= 2)
MOI.submit(model, MOI.LazyConstraint(cb_data), con)
end
return
end
my_callback_function (generic function with 1 method)</pre>
```

julia> set\_attribute(model, MOI.LazyConstraintCallback(), my\_callback\_function)

#### Notes

- The lazy constraint callback may be called at fractional or integer nodes in the branch-and-bound tree. Use callback\_node\_status to distinguish.
- There is no guarantee that the callback is called at *every* primal solution.
- The solver may visit a point that was cut off by a previous lazy constraint, for example, because the earlier lazy constraint was removed during presolve. If this happens, you must re-add the lazy constraint.
- Only add a lazy constraint if the primal solution violates the constraint. Adding the lazy constraint irrespective of feasibility may result in the solver returning an incorrect solution, or lead to many constraints being added, slowing down the solution process. For example, instead of:

```
model = Model()
@variable(model, x <= 10, Int)</pre>
@objective(model, Max, x)
function bad_callback_function(cb_data)
 con = @build constraint(x <= 2)</pre>
 MOI.submit(model, MOI.LazyConstraint(cb_data), con)
 return
end
do
function good_callback_function(cb_data)
 if callback_value(cb_data, x) > 2
 con = @build_constraint(x <= 2)</pre>
 MOI.submit(model, MOI.LazyConstraint(cb_data), con)
 end
 return
end
set_attribute(model, MOI.LazyConstraintCallback(), good_callback_function)
```

#### 18.4 User cuts

User cuts, or simply cuts, provide a way for the user to tighten the LP relaxation using problem-specific knowledge that the solver cannot or is unable to infer from the model. Just like with lazy constraints, when a MIP solver reaches a new node in the branch-and-bound tree, it will give the user the chance to provide cuts to make the current relaxed (fractional) solution infeasible in the hopes of obtaining an integer solution. For more details about the difference between user cuts and lazy constraints see the aforementioned blog post.

A user-cut callback can be added using the following syntax:

julia> set attribute(model, MOI.UserCutCallback(), my callback function)

#### Notes

- User cuts **must not** change the set of integer feasible solutions. Equivalently, user cuts can only remove fractional solutions. If you add a cut that removes an integer solution (even one that is not optimal), the solver may return an incorrect solution.
- The user-cut callback *may* be called at fractional nodes in the branch-and-bound tree. There is no guarantee that the callback is called at *every* fractional primal solution.

#### 18.5 Heuristic solutions

Integer programming solvers frequently include heuristics that run at the nodes of the branch-and-bound tree. They aim to find integer solutions quicker than plain branch-and-bound would to tighten the bound, allowing us to fathom nodes quicker and to tighten the integrality gap.

Some heuristics take integer solutions and explore their "local neighborhood" (for example, flipping binary variables, fix some variables and solve a smaller MILP) and others take fractional solutions and attempt to round them in an intelligent way.

You may want to add a heuristic of your own if you have some special insight into the problem structure that the solver is not aware of, for example, you can consistently take fractional solutions and intelligently guess integer solutions from them.

A heuristic solution callback can be added using the following syntax:

```
Heuristic solution by rounding down to nearest integer
Float64[floor(Int, callback_value(cb_data, x))],
)
println(
 "I submitted a heuristic solution, and the status was: ",
 status,
)
return
end
my_callback_function (generic function with 1 method)
```

```
julia> set_attribute(model, MOI.HeuristicCallback(), my_callback_function)
```

MOI.submit returns a MOI.HeuristicSolutionStatus enum that indicates whether the solver accepted the solution. The possible return codes are:

- MOI.HEURISTIC\_SOLUTION\_ACCEPTED
- MOI.HEURISTIC\_SOLUTION\_REJECTED
- MOI.HEURISTIC\_SOLUTION\_UNKNOWN

#### Notes

- Some solvers may accept partial solutions. Others require a feasible integer solution for every variable. If in doubt, provide a complete solution.
- The heuristic solution callback *may* be called at fractional nodes in the branch-and-bound tree. There is no guarantee that the callback is called at *every* fractional primal solution.
# **Chapter 19**

# **Complex number support**

This page explains the complex-valued variables and constraints that JuMP supports. For a worked-example using these features, read the Example: quantum state discrimination tutorial.

# 19.1 Complex-valued variables

Create a complex-valued variable using ComplexPlane:

```
julia> model = Model();
```

```
julia> @variable(model, x in ComplexPlane())
real(x) + imag(x) im
```

Note that x is not a VariableRef; instead, it is an affine expression with Complex{Float64}-valued coefficients:

```
julia> typeof(x)
GenericAffExpr{ComplexF64, VariableRef}
```

Behind the scenes, JuMP has created two real-valued variables, with names "real(x)" and "imag(x)":

```
julia> all_variables(model)
2-element Vector{VariableRef}:
 real(x)
 imag(x)
```

```
julia> name.(all_variables(model))
2-element Vector{String}:
 "real(x)"
 "imag(x)"
```

Use the real and imag functions on x to return a real-valued affine expression representing each variable:

```
julia> typeof(real(x))
AffExpr (alias for GenericAffExpr{Float64, GenericVariableRef{Float64}})
```

#### julia> typeof(imag(x))

AffExpr (alias for GenericAffExpr{Float64, GenericVariableRef{Float64}})

To create an anonymous variable, use the set keyword argument:

```
julia> model = Model();
julia> x = @variable(model, set = ComplexPlane())
_[1] + _[2] im
```

# 19.2 Complex-valued variable and start values bounds

Because complex-valued variables lack a total ordering, the definition of a variable bound for a complex-valued variable is ambiguous. If you pass a real- or complex-valued argument to keywords such as lower\_bound, upper\_bound, and start\_value, JuMP will apply the real and imaginary parts to the associated real-valued variables.

```
julia> model = Model();
julia> @variable(
 model,
 x in ComplexPlane(),
 lower_bound = 1.0,
 upper_bound = 2.0 + 3.0im,
 start = 4im,
)
real(x) + imag(x) im
julia> vars = all_variables(model)
2-element Vector{VariableRef}:
real(x)
imag(x)
julia> lower_bound.(vars)
2-element Vector{Float64}:
1.0
0.0
julia> upper_bound.(vars)
2-element Vector{Float64}:
2.0
3.0
julia> start_value.(vars)
2-element Vector{Float64}:
0.0
4.0
```

You can modify the bounds and start values by passing imag(x) or real(x) to the appropriate function:

```
julia> set_lower_bound(imag(x), 2)
julia> lower_bound(imag(x))
2.0
```

```
julia> delete_upper_bound(real(x))
julia> has_upper_bound(real(x))
false
julia> set_start_value(imag(x), 3)
julia> start_value(imag(x))
3.0
```

# 19.3 Complex-valued equality constraints

JuMP reformulates complex-valued equality constraints into two real-valued constraints: one representing the real part, and one representing the imaginary part. Thus, complex-valued equality constraints can be solved any solver that supports the real-valued constraint type.

For example:

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x[1:2]);
julia> @constraint(model, (1 + 2im) * x[1] + 3 * x[2] == 4 + 5im)
(1 + 2im) x[1] + 3 x[2] = (4 + 5im)
julia> optimize!(model)
julia> value.(x)
2-element Vector{Float64}:
2.5
0.5
```

is equivalent to

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x[1:2]);
julia> @constraint(model, 1 * x[1] + 3 * x[2] == 4) # real component
x[1] + 3 x[2] = 4
julia> @constraint(model, 2 * x[1] == 5) # imag component
2 x[1] = 5
julia> optimize!(model)
julia> value.(x)
2-element Vector{Float64}:
```

2.5 0.5

This also applies if the variables are complex-valued:

```
julia> model = Model(HiGHS.Optimizer);
```

julia> set\_silent(model)

```
julia> @variable(model, x in ComplexPlane());
```

```
julia> @constraint(model, (1 + 2im) * x + 3 * x == 4 + 5im)
(4 + 2im) real(x) + (-2 + 4im) imag(x) = (4 + 5im)
```

julia> optimize!(model)

julia> value(x)
1.3 + 0.6000000000000001im

which is equivalent to

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x_real);
julia> @variable(model, x_imag);
julia> @constraint(model, x_real - 2 * x_imag + 3 * x_real == 4)
4 x_real - 2 x_imag = 4
julia> @constraint(model, x_imag + 2 * x_real + 3 * x_imag == 5)
2 x_real + 4 x_imag = 5
julia> optimize!(model)
julia> value(x_real) + value(x_imag) * im
```

```
1.3 + 0.600000000000001im
```

# 19.4 Hermitian PSD Cones

JuMP supports creating matrices where are Hermitian.

julia> model = Model();

```
 \begin{array}{ll} \mbox{real}(H[1,2]) &- \mbox{imag}(H[1,2]) \mbox{ im} \\ \mbox{real}(H[1,3]) &- \mbox{imag}(H[1,3]) \mbox{ im} \\ \mbox{real}(H[3,3]) \end{array} \\ \label{eq:harden} \begin{array}{ll} \mbox{real}(H[2,3]) &+ \mbox{imag}(H[2,3]) \mbox{ im} \\ \mbox{real}(H[3,3]) \end{array} \\ \label{eq:harden} \end{array}
```

Behind the scenes, JuMP has created nine real-valued decision variables:

```
julia> all_variables(model)
9-element Vector{VariableRef}:
 real(H[1,1])
 real(H[1,2])
 real(H[2,2])
 real(H[2,3])
 real(H[2,3])
 imag(H[1,2])
 imag(H[1,3])
 imag(H[2,3])
```

and a Vector{VariableRef}-in-MOI.HermitianPositiveSemidefiniteConeTriangle constraint:

julia> num\_constraints(model, Vector{VariableRef}, MOI.HermitianPositiveSemidefiniteConeTriangle)
1

The MOI.HermitianPositiveSemidefiniteConeTriangle set can be efficiently bridged to MOI.PositiveSemidefiniteConeTria so it can be solved by any solver that supports PSD constraints.

Each element of H is an affine expression with Complex{Float64}-valued coefficients:

```
julia> typeof(H[1, 1])
GenericAffExpr{ComplexF64, VariableRef}
```

julia> typeof(H[2, 1])
GenericAffExpr{ComplexF64, VariableRef}

## **Start values**

When setting the start value, you must be careful to set only the upper triangle of real variables, and the upper triangle excluding the diagonal of imaginary variables:

```
julia> import LinearAlgebra
julia> function set_hermitian_start(
 H::LinearAlgebra.Hermitian,
 start::LinearAlgebra.Hermitian,
)
 for j in 1:size(H, 2), i in 1:j
 set_start_value(real(H[i, j]), real(start[i, j]))
 if i < j
 set_start_value(imag(H[i, j]), imag(start[i, j]))
 end
 end</pre>
```

# return

end

set\_hermitian\_start (generic function with 1 method)

julia> set\_hermitian\_start(H, H0)

#### julia> start\_value.(all\_variables(model))

9-element Vector{Float64}: 1.0 2.0 4.0 5.0 7.0 9.0 3.0 6.0 8.0

# **19.5 Hermitian PSD constraints**

The HermitianPSDCone can also be used in the @constraint macro:

# Note

The matrix H in H in HermitianPSDCone() must be a LinearAlgebra.Hermitian matrix type. A build\_constraint error will be thrown if the matrix is a different matrix type.

# **Chapter 20**

# **Nonlinear Modeling**

JuMP has support for nonlinear (convex and nonconvex) optimization problems. JuMP is able to automatically provide exact, sparse second-order derivatives to solvers. This information can improve solver accuracy and performance.

# 20.1 Set a nonlinear objective

Use @objective to set a nonlinear objective.

```
julia> model = Model();
```

julia> @variable(model, x[1:2]);

```
julia> @objective(model, Min, exp(x[1]) - sqrt(x[2]))
exp(x[1]) - sqrt(x[2])
```

To modify a nonlinear objective, call @objective again.

# 20.2 Add a nonlinear constraint

Use @constraint to add a nonlinear constraint.

Delete a nonlinear constraint using delete:

```
julia> delete(model, con[1])
```

# 20.3 Add a parameter

Some solvers have explicit support for parameters, which are constants in the model that can be efficiently updated between solves.

JuMP implements parameters by a decision variable constrained on creation to the Parameter set.

```
julia> model = Model();
julia> @variable(model, x);
julia> @variable(model, p[i = 1:2] in Parameter(i))
2-element Vector{VariableRef}:
 p[1]
 p[2]
julia> parameter_value(p[1])
1.0
julia> set_parameter_value(p[1], 3.5)
julia> @objective(model, Max, log(p[1] * x + p[2]))
log(p[1]*x + p[2])
```

See Parameters for more information on how to create and manage parameters.

Parameters are most useful when solving nonlinear models in a sequence:

```
julia> using JuMP, Ipopt
julia> model = Model(Ipopt.Optimizer);
julia> set_silent(model)
julia> @variable(model, x)
x
julia> @variable(model, p in Parameter(1.0))
p
julia> @objective(model, Min, (x - p)^2)
x² - 2 p*x + p²
julia> optimize!(model)
julia> value(x)
1.0
julia> set_parameter_value(p, 5.0)
julia> optimize!(model)
```

```
julia> value(x)
5.0
```

Using parameters can be faster than creating a new model from scratch with updated data because JuMP is able to avoid repeating a number of steps in processing the model before handing it off to the solver.

## 20.4 Create a nonlinear expression

Use @expression to create nonlinear expression objects:

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> expr = @expression(model, exp(x[1]) + sqrt(x[2]))
exp(x[1]) + sqrt(x[2])
julia> my_anon_expr = @expression(model, [i = 1:2], sin(x[i]))
2-element Vector{NonlinearExpr}:
sin(x[1])
sin(x[2])
julia> @expression(model, my_expr[i = 1:2], sin(x[i]))
2-element Vector{NonlinearExpr}:
sin(x[1])
sin(x[2])
```

A NonlinearExpr can be used in @objective, @constraint, and even nested in other @expressions.

Use value to query the value of a nonlinear expression:

julia> set\_start\_value(x[1], 1.0)

```
julia> value(start_value, nested[1])
0.7456241416655579
```

```
julia> sin(sin(1.0))
0.7456241416655579
```

# 20.5 Common subexpressions

JuMP does not perform common subexpression elimination. Instead, if you re-use an expression in multiple places, JuMP will insert a copy of the expression.

JuMP's lack of common subexpression elimination is a common cause of performance problems, particularly in nonlinear models with a pattern like sum(t / common\_term for t in terms). One example is the logistic loss:

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @expression(model, expr, sum(exp.(x)))
0.0 + exp(x[2]) + exp(x[1])
julia> @objective(model, Min, sum(exp(x[i]) / expr for i in 1:2))
(exp(x[1]) / (0.0 + exp(x[2]) + exp(x[1]))) + (exp(x[2]) / (0.0 + exp(x[2]) + exp(x[1])))
```

In this model, JuMP will compute the value (and derivatives) of the denominator twice, without realizing that it is the same expression.

As a work-around, create a new @variable and use an == @constraint to constrain the value of the variable to the subexpression.

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @variable(model, expr);
julia> @constraint(model, expr == sum(exp.(x)))
expr - (0.0 + exp(x[2]) + exp(x[1])) = 0
julia> @objective(model, Min, sum(exp(x[i]) / expr for i in 1:2))
(exp(x[1]) / expr) + (exp(x[2]) / expr)
```

The reason JuMP does not perform common subexpression elimination automatically is for simplicity, and because there is a trade-off: for simple expressions, the extra complexity of detecting and merging common subexpressions may outweigh the cost of computing them independently. Instead, we leave it to the user to decide which expressions to extract as common subexpressions.

# 20.6 Automatic differentiation

JuMP computes first- and second-order derivatives using sparse reverse-mode automatic differentiation. For details, see ReverseAD.

For a tutorial on how to construct and query the derivatives, see Computing Hessians

# 20.7 Nonlinear expressions in detail

Nonlinear expressions in JuMP are represented by a NonlinearExpr object.

# Constructors

Nonlinear expressions can be created using the NonlinearExpr constructors:

```
julia> model = Model();
julia> @variable(model, x);
julia> expr = NonlinearExpr(:sin, Any[x])
sin(x)
```

or via operator overloading:

```
julia> model = Model();
```

julia> @variable(model, x);

julia> expr = sin(x)
sin(x)

#### Supported arguments

Nonlinear expressions can contain a mix of numbers, AffExpr, QuadExpr, and other NonlinearExpr:

```
julia> model = Model();
julia> @variable(model, x);
julia> aff = x + 1;
julia> quad = x^2 + x;
julia> expr = cos(x) * sin(quad) + aff
(cos(x) * sin(x² + x)) + (x + 1)
```

# Supported operators

The list of supported operators may vary between solvers. Given an optimizer, query the list of supported operators using MOI.ListOfSupportedNonlinearOperators:

julia> import Ipopt

julia> import MathOptInterface as MOI

```
julia> MOI.get(Ipopt.Optimizer(), MOI.ListOfSupportedNonlinearOperators())
85-element Vector{Symbol}:
 :+
```

:-
:abs
:sqrt
:cbrt
:abs2
:inv
:log
:log10
:log2
:min
:max
: &&
:
:<=
:(==)
:>=
' <

In some univariate cases, the operator is defined in SpecialFunctions.jl. To use these functions, you must explicitly import SpecialFunctions.jl

```
julia> import Ipopt
julia> op = MOI.get(Ipopt.Optimizer(), MOI.ListOfSupportedNonlinearOperators());
julia> :erfcx in op
true
julia> :dawson in op
true
julia> import SpecialFunctions
julia> model = Model();
julia> @variable(model, x)
x
julia> @expression(model, SpecialFunctions.erfcx(x))
erfcx(x)
julia> @expression(model, SpecialFunctions.dawson(x))
dawson(x)
```

# Limitations

Some nonlinear expressions cannot be created via operator overloading. For example, to minimize the likelihood of bugs in user-code, we have not overloaded comparisons such as < and >= between JuMP objects:

```
julia> model = Model();
julia> @variable(model, x);
julia> x < 1
ERROR: Cannot evaluate `<` between a variable and a number.
[...]
```

Instead, wrap the expression in the @expression macro:

```
julia> model = Model();
julia> @variable(model, x);
julia> expr = @expression(model, x < 1)
x < 1</pre>
```

For technical reasons, other operators that are not overloaded include ||, &&, and ifelse.

```
julia> model = Model();
julia> @variable(model, x);
julia> expr = @expression(model, ifelse(x < -1 || x >= 1, x^2, 0.0))
ifelse((x < -1) || (x >= 1), x², 0.0)
```

As an alternative, use the JuMP.op\_functions, which fallback to the various comparison and logical operators:

The available functions are:

# Fields

Each NonlinearExpr has two fields.

The .head field is a Symbol that represents the operator being called:

```
julia> expr.head
:sin
```

JuMP function	Julia function
op_ifelse	ifelse
op_and	&&
op_or	11
op_greater_than_or_equal_to	>=
op_less_than_or_equal_to	<=
op_equal_to	==
op_strictly_greater_than	>
op_strictly_less_than	<

The .args field is a Vector{Any} containing the arguments to the operator:

```
julia> expr.args
1-element Vector{Any}:
 x
```

# Forcing nonlinear expressions

The JuMP macros and operator overloading will preferentially build affine (GenericAffExpr) and quadratic (GenericQuadExpr) expressions instead of GenericNonlinearExpr. For example:

julia> model = Model();

julia> @variable(model, x);

**julia> f = (x** - 0.1)^2 x<sup>2</sup> - 0.2 x + 0.01000000000000002

```
julia> typeof(f)
```

QuadExpr (alias for GenericQuadExpr{Float64, GenericVariableRef{Float64}})

To override this behavior, use the @force nonlinear macro:

julia> g = @force\_nonlinear((x - 0.1)^2) (x - 0.1) ^ 2

julia> typeof(g) NonlinearExpr (alias for GenericNonlinearExpr{GenericVariableRef{Float64}})

#### Warning

Use this macro only if necessary. See the docstring of @force\_nonlinear for more details on when you should use it.

# 20.8 Function tracing

Nonlinear expressions can be constructed using *function tracing*. Function tracing is when you call a regular Julia function with JuMP variables as arguments and the function builds a nonlinear expression via operator overloading. For example:

```
julia> using JuMP
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> f(x::Vector{VariableRef}) = 2 * sin(x[1]^2) + sqrt(x[2])
f (generic function with 1 method)
julia> y = f(x)
(2.0 * sin(x[1]²)) + sqrt(x[2])
julia> typeof(y)
NonlinearExpr (alias for GenericNonlinearExpr{GenericVariableRef{Float64}})
```

```
julia> @objective(model, Max, f(x))
(2.0 * sin(x[1]²)) + sqrt(x[2])
```

Function tracing supports functions which return vectors or arrays of NonlinearExpr:

```
julia> using JuMP
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> f(x::Vector{VariableRef}) = sqrt.(x)
f (generic function with 1 method)
julia> y = f(x)
2-element Vector{NonlinearExpr}:
sqrt(x[1])
sqrt(x[2])
julia> typeof(y)
Vector{NonlinearExpr} (alias for Array{GenericNonlinearExpr{GenericVariableRef{Float64}}, 1})
julia> @constraint(model, f(x) .<= 2)</pre>
2-element Vector{ConstraintRef{Model,
\hookrightarrow \mbox{MathOptInterface.ConstraintIndex{MathOptInterface.ScalarNonlinearFunction,} \label{eq:MathOptInterface}
\hookrightarrow MathOptInterface.LessThan{Float64}}, ScalarShape}}:
sqrt(x[1]) - 2.0 \le 0
```

```
julia> @objective(model, Max, sum(f(x)))
0.0 + sqrt(x[2]) + sqrt(x[1])
```

 $sqrt(x[2]) - 2.0 \le 0$ 

Because function tracing uses operator overloading, there are many functions for which it will not work. For

example:

```
julia> using JuMP
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> f(x::Vector{VariableRef}) = x[1] > 1 ? 0 : x[2]
f (generic function with 1 method)
julia> f(x)
ERROR: Cannot evaluate `>` between a variable and a number.
[...]
```

In these cases, you should define a User-defined operator using the @operator macro.

# 20.9 User-defined operators

In addition to a standard list of univariate and multivariate operators recognized by the MOI.Nonlinear submodule, JuMP supports user-defined operators, which let you represent nonlinear functions that cannot (or should not) be traced, for example, because they rely on non-Julia subroutines.

#### Warning

User-defined operators must return a scalar output. For a work-around, see User-defined operators with vector outputs.

# Add an operator

Add a user-defined operator using the <a>@operator</a> macro:

```
julia> using JuMP
```

```
julia> square(x) = x^2
square (generic function with 1 method)
```

```
julia> f(x, y) = (x - 1)^2 + (y - 2)^2
f (generic function with 1 method)
```

```
julia> model = Model();
```

julia> @operator(model, op\_square, 1, square) NonlinearOperator(square, :op\_square)

```
julia> @operator(model, op_f, 2, f)
NonlinearOperator(f, :op_f)
```

```
julia> @variable(model, x[1:2]);
```

```
julia> @objective(model, Min, op_f(x[1], op_square(x[2])))
op_f(x[1], op_square(x[2]))
```

The arguments to **@operator** are:

- 1. The model to which the operator is added.
- 2. A Julia symbol object which serves as the name of the user-defined operator in JuMP expressions. This name must not be the same as that of the function.
- 3. The number of scalar input arguments that the function takes.
- 4. A Julia method which computes the function.

#### Warning

User-defined operators cannot be deleted.

You can obtain a reference to the operator using the model[:key] syntax:

```
julia> using JuMP
julia> square(x) = x^2
square (generic function with 1 method)
julia> model = Model();
```

julia> @operator(model, op\_square, 1, square) NonlinearOperator(square, :op\_square)

julia> op\_square\_2 = model[:op\_square] NonlinearOperator(square, :op\_square)

# **Automatic differentiation**

JuMP computes first- and second-order derivatives of expressions using ReverseAD, which implements sparse reverse-mode automatic differentiation. However, because ReverseAD requires the algebraic expression as input, JuMP cannot use ReverseAD to differentiate user-defined operators.

Instead, unless Gradients and Hessians are explicitly provided, user-defined operators must support automatic differentiation by ForwardDiff.jl.

The use of FowardDiff.jl has two important implications:

- 1. ForwardDiff.jl supports only a limited subset of Julia. If you encounter an error adding the operator, see Common mistakes when writing a user-defined operator.
- 2. Differentiating operators with many arguments is slow. In general, you should try to keep the number of arguments to less than 100, and ideally, to less than 10.

Because of the use of ForwardDiff, in most cases, you should prefer to use function tracing instead of defining a user-defined operator.

# Add an operator without macros

The <u>@operator</u> macro is syntactic sugar for <u>add\_nonlinear\_operator</u>. Thus, the non-macro version of the preceding example is:

#### julia> using JuMP

julia> square(x) = x<sup>2</sup> square (generic function with 1 method)

julia> f(x, y) = (x - 1)^2 + (y - 2)^2
f (generic function with 1 method)

julia> model = Model();

julia> op\_square = add\_nonlinear\_operator(model, 1, square; name = :op\_square)
NonlinearOperator(square, :op square)

julia> model[:op\_square] = op\_square NonlinearOperator(square, :op\_square)

julia> op\_f = add\_nonlinear\_operator(model, 2, f; name = :op\_f) NonlinearOperator(f, :op\_f)

julia> model[:op\_f] = op\_f NonlinearOperator(f, :op\_f)

```
julia> @variable(model, x[1:2]);
```

```
julia> @objective(model, Min, op_f(x[1], op_square(x[2])))
op_f(x[1], op_square(x[2]))
```

#### Operators with the same name as an existing function

A common error encountered is the following:

```
julia> using JuMP
```

julia> model = Model();

julia> f(x) = x^2
f (generic function with 1 method)

```
julia> @operator(model, f, 1, f)
ERROR: Unable to add the nonlinear operator `:f` with the same name as
an existing function.
[...]
```

This error occurs because @operator(model, f, 1, f) is equivalent to:

julia> f = add\_nonlinear\_operator(model, 1, f; name = :f)

but f already exists as a Julia function.

If you evaluate the function without adding it as an operator, JuMP will trace the function using operator overloading:

```
julia> @variable(model, x);
julia> f(x)
x²
```

To force JuMP to treat f as a user-defined operator and not trace it, add the operator using add\_nonlinear\_operator and define a new method which manually creates a NonlinearExpr:

```
julia> _ = add_nonlinear_operator(model, 1, f; name = :f)
NonlinearOperator(f, :f)
julia> f(x::AbstractJuMPScalar) = NonlinearExpr(:f, Any[x])
f (generic function with 2 methods)
julia> @expression(model, log(f(x)))
log(f(x))
```

# **Gradients and Hessians**

By default, JuMP will use automatic differentiation to compute the gradient and Hessian of user-defined operators. If your function is not amenable to the default automatic differentiation, or you can compute analytic derivatives, you may pass additional arguments to <u>@operator</u> to compute the first- and second-derivatives.

# Тір

The tutorial Automatic differentiation of user-defined operators has examples of how to use third-party Julia packages to compute automatic derivatives.

# **Univariate functions**

For univariate functions, a gradient function  $\nabla f$  returns a number that represents the first-order derivative. You may, in addition, pass a third function which returns a number representing the second-order derivative:

```
julia> using JuMP
```

```
julia> f(x) = x²
f (generic function with 1 method)
```

julia> ∇f(x) = 2x ∇f (generic function with 1 method)

**julia**>  $\nabla^2 f(x) = 2$  $\nabla^2 f$  (generic function with 1 method)

```
julia> model = Model();
```

**julia>** @operator(model, op\_f, 1, f,  $\nabla f$ ,  $\nabla^2 f$ ) # Providing  $\nabla^2 f$  is optional NonlinearOperator(f, :op\_f)

```
julia> @variable(model, x)
x
julia> @objective(model, Min, op_f(x))
op_f(x)
```

# **Multivariate functions**

For multivariate functions, the gradient function  $\nabla f$  must take an AbstractVector as the first argument that is filled in-place. The Hessian function,  $\nabla^2 f$ , must take an AbstractMatrix as the first argument, the lower-triangular of which is filled in-place:

```
julia> using JuMP
```

```
julia> f(x...) = (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2
f (generic function with 1 method)
julia> function ∇f(g::AbstractVector{T}, x::T...) where {T}
 g[1] = 400 * x[1]^3 - 400 * x[1] * x[2] + 2 * x[1] - 2
 g[2] = 200 * (x[2] - x[1]^2)
 return
 end
\nabla f (generic function with 1 method)
julia> function \nabla^2 f(H::AbstractMatrix{T}, x::T...) where {T}
 H[1, 1] = 1200 * x[1]^2 - 400 * x[2] + 2
 # H[1, 2] = -400 * x[1] <-- Not needed. Fill the lower-triangular only.</pre>
 H[2, 1] = -400 * x[1]
 H[2, 2] = 200.0
 return
 end
\nabla^2 f (generic function with 1 method)
julia> model = Model();
julia> @operator(model, rosenbrock, 2, f, \nabla f, \nabla^2 f) # Providing \nabla^2 f is optional
NonlinearOperator(f, :rosenbrock)
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
x[1]
x[2]
julia> @objective(model, Min, rosenbrock(x[1], x[2]))
```

rosenbrock(x[1], x[2])

You may assume the Hessian matrix H is initialized with zeros, and because H is symmetric, you need only to fill in the non-zero lower-triangular terms. The matrix type passed in as H depends on the automatic differentiation system, so make sure the first argument to the Hessian function supports an AbstractMatrix (it may be something other than Matrix{Float64}). Moreover, you may assume only that H supports size(H) and setindex!. Finally, the matrix is treated as dense, so the performance will be poor on functions with high-dimensional input.

# User-defined operators with vector inputs

User-defined operators which take vectors as input arguments (for example, f(x::Vector)) are *not* supported. Instead, use Julia's splatting syntax to create a function with scalar arguments. For example, instead of:

```
f(x::Vector) = sum(x[i]^i for i in 1:length(x))
```

define:

f(x...) = sum(x[i]^i for i in 1:length(x))

Another approach is to define the splatted function as an anonymous function:

op\_f(x[1], x[2], x[3], x[4], x[5])

If the operator takes several vector inputs, write a function that takes the splatted arguments and reconstructs the required vector inputs:

```
julia> using JuMP
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @variable(model, y[1:2]);
julia> @variable(model, z);
julia> f(x::Vector, y::Vector, z) = sum((x[i] * y[i])^z for i in 1:2)
f (generic function with 1 method)
julia> f(x, y, z)
```

 $((x[1]*y[1]) ^ z) + ((x[2]*y[2]) ^ z)$ 

```
julia> f_splat(args...) = f(collect(args[1:2]), collect(args[3:4]), args[5])
f_splat (generic function with 1 method)
```

julia> f\_splat(x..., y..., z)
((x[1]\*y[1]) ^ z) + ((x[2]\*y[2]) ^ z)

```
julia> @operator(model, op_f, 5, f_splat)
NonlinearOperator(f_splat, :op_f)
```

```
julia> @objective(model, Min, op_f(x..., y..., z))
op_f(x[1], x[2], y[1], y[2], z)
```

### Common mistakes when writing a user-defined operator

JuMP uses ForwardDiff.jl to compute the first-order derivatives of user-defined operators. ForwardDiff has a number of limitations that you should be aware of when writing user-defined operators.

The rest of this section provides debugging advice and explains some common mistakes.

Warning

Get an error like No method matching Float64(::ForwardDiff.Dual)? Read this section.

# Debugging

If you add an operator that does not support ForwardDiff, a long error message will be printed. You can review the stacktrace for more information, but it can often be hard to understand why and where your function is failing.

It may be helpful to debug the operator outside of JuMP as follows.

If the operator is univariate, do:

```
julia> import ForwardDiff
```

```
julia> my_operator(a) = a^2
my_operator (generic function with 1 method)
```

```
julia> ForwardDiff.derivative(my_operator, 1.0)
2.0
```

If the operator is multivariate, do:

julia> import ForwardDiff

```
julia> my_operator(a, b) = a² + b²
my_operator (generic function with 1 method)
```

```
julia> ForwardDiff.gradient(x -> my_operator(x...), [1.0, 2.0])
2-element Vector{Float64}:
 2.0
 4.0
```

Note that even though the operator takes the splatted arguments, ForwardDiff.gradient requires a vector as input.

#### Operator calls something unsupported by ForwardDiff

ForwardDiff works by overloading many Julia functions for a special type ForwardDiff.Dual <: Real. If your operator attempts to call a function for which an overload has not been defined, a MethodError will be thrown.

For example, your operator cannot call external C functions, or be the optimal objective value of a JuMP model.

```
julia> import ForwardDiff
julia> my_operator_bad(x) = @ccall sqrt(x::Cdouble)::Cdouble
my_operator_bad (generic function with 1 method)
julia> ForwardDiff.derivative(my_operator_bad, 1.0)
ERROR: MethodError: no method matching
↔ Float64(::ForwardDiff.Dual{ForwardDiff.Tag{typeof(my_operator_bad), Float64}, Float64, 1})
[...]
```

Unfortunately, the list of calls supported by ForwardDiff is too large to enumerate what is an isn't allowed, so the best advice is to try and see if it works.

#### Operator does not accept splatted input

The operator takes f(x::Vector) as input, instead of the splatted f(x...).

```
julia> import ForwardDiff
julia> my_operator_bad(x::Vector) = sum(x[i]^2 for i in eachindex(x))
my_operator_bad (generic function with 1 method)
julia> my_operator_good(x...) = sum(x[i]^2 for i in eachindex(x))
my_operator_good (generic function with 1 method)
julia> ForwardDiff.gradient(x -> my_operator_bad(x...), [1.0, 2.0])
ERROR: MethodError: no method matching

→ my_operator_bad(::ForwardDiff.Dual{ForwardDiff.Tag{var"#5#6", Float64}, Float64, 2},

→ ::ForwardDiff.Dual{ForwardDiff.Tag{var"#5#6", Float64}, Float64, 2},

i...]
julia> ForwardDiff.gradient(x -> my_operator_good(x...), [1.0, 2.0])
2-element Vector{Float64}:

2.0

4.0
```

#### **Operator assumes Float64 as input**

The operator assumes Float64 will be passed as input, but it must work for any generic Real type.

```
julia> import ForwardDiff
```

julia> my\_operator\_bad(x::Float64...) = sum(x[i]^2 for i in eachindex(x))

my\_operator\_bad (generic function with 1 method)

```
julia> my_operator_good(x::Real...) = sum(x[i]^2 for i in eachindex(x))
my_operator_good (generic function with 1 method)
julia> ForwardDiff.gradient(x -> my_operator_bad(x...), [1.0, 2.0])
ERROR: MethodError: no method matching

 my_operator_bad(::ForwardDiff.Dual{ForwardDiff.Tag{var"#5#6", Float64}, Float64, 2},

 ::ForwardDiff.Dual{ForwardDiff.Tag{var"#5#6", Float64}, Float64, 2})
```

```
julia> ForwardDiff.gradient(x -> my_operator_good(x...), [1.0, 2.0])
2-element Vector{Float64}:
 2.0
 4.0
```

#### **Operator allocates Float64 storage**

[...]

The operator allocates temporary storage using zeros (3) or similar. This defaults to Float64, so use zeros (T, 3) instead.

```
julia> import ForwardDiff
julia> function my_operator_bad(x::Real...)
 # This line is problematic. zeros(n) is short for zeros(Float64, n)
 y = zeros(length(x))
 for i in eachindex(x)
 y[i] = x[i]^2
 end
 return sum(y)
 end
my_operator_bad (generic function with 1 method)
julia> function my_operator_good(x::T...) where {T<:Real}</pre>
 y = zeros(T, length(x))
 for i in eachindex(x)
 y[i] = x[i]^2
 end
 return sum(y)
 end
my_operator_good (generic function with 1 method)
julia> ForwardDiff.gradient(x -> my_operator_bad(x...), [1.0, 2.0])
ERROR: MethodError: no method matching Float64(::ForwardDiff.Dual{ForwardDiff.Tag{var"#1#2",
\hookrightarrow Float64}, Float64, 2})
[...]
julia> ForwardDiff.gradient(x -> my_operator_good(x...), [1.0, 2.0])
2-element Vector{Float64}:
2.0
4.0
```

# **Chapter 21**

# Nonlinear Modeling (Legacy)

#### Warning

This page describes the legacy nonlinear interface to JuMP. It has a number of quirks and limitations that prompted the development of a new nonlinear interface. The new interface is documented at Nonlinear Modeling. This legacy interface will remain for all future v1.X releases of JuMP. The two nonlinear interfaces cannot be combined.

JuMP has support for general smooth nonlinear (convex and nonconvex) optimization problems. JuMP is able to provide exact, sparse second-order derivatives to solvers. This information can improve solver accuracy and performance.

There are three main changes to solve nonlinear programs in JuMP.

- Use @NLobjective instead of @objective
- Use @NLconstraint instead of @constraint
- Use @NLexpression instead of @expression

#### Info

There are some restrictions on what syntax you can use in the @NLxxx macros. Make sure to read the Syntax notes.

# 21.1 Set a nonlinear objective

Use @NLobjective to set a nonlinear objective.

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2]);
```

```
julia> @NLobjective(model, Min, exp(x[1]) - sqrt(x[2]))
```

To modify a nonlinear objective, call @NLobjective again.

# 21.2 Add a nonlinear constraint

Use @NLconstraint to add a nonlinear constraint.

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @NLconstraint(model, exp(x[1]) <= 1)
exp(x[1]) - 1.0 ≤ 0
julia> @NLconstraint(model, [i = 1:2], x[i]^i >= i)
2-element Vector{NonlinearConstraintRef{ScalarShape}}:
x[1] ^ 1.0 - 1.0 ≥ 0
x[2] ^ 2.0 - 2.0 ≥ 0
julia> @NLconstraint(model, con[i = 1:2], prod(x[j] for j = 1:i) == i)
2-element Vector{NonlinearConstraintRef{ScalarShape}}:
```

```
2-element Vector{NonlinearConstraintRef{ScalarShape}}:
 (*)(x[1]) - 1.0 = 0
 x[1] * x[2] - 2.0 = 0
```

# Info

You can only create nonlinear constraints with <=, >=, and ==. More general Nonlinear-in-Set constraints are not supported.

Delete a nonlinear constraint using delete:

```
julia> delete(model, con[1])
```

# 21.3 Create a nonlinear expression

Use <u>@NLexpression</u> to create nonlinear expression objects. The syntax is identical to <u>@expression</u>, except that the expression can contain nonlinear terms.

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> expr = @NLexpression(model, exp(x[1]) + sqrt(x[2]))
subexpression[1]: exp(x[1]) + sqrt(x[2])
julia> my_anon_expr = @NLexpression(model, [i = 1:2], sin(x[i]))
2-element Vector{NonlinearExpression}:
subexpression[2]: sin(x[1])
subexpression[3]: sin(x[2])
julia> @NLexpression(model, my_expr[i = 1:2], sin(x[i]))
2-element Vector{NonlinearExpression}:
subexpression[4]: sin(x[1])
subexpression[5]: sin(x[2])
```

Nonlinear expression can be used in @NLobjective, @NLconstraint, and even nested in other @NLexpressions.

```
julia> @NLobjective(model, Min, expr^2 + 1)
julia> @NLconstraint(model, [i = 1:2], my_expr[i] <= i)
2-element Vector{NonlinearConstraintRef{ScalarShape}}:
subexpression[4] - 1.0 ≤ 0
subexpression[5] - 2.0 ≤ 0
julia> @NLexpression(model, nested[i = 1:2], sin(my_expr[i]))
2-element Vector{NonlinearExpression}:
subexpression[6]: sin(subexpression[4])
subexpression[7]: sin(subexpression[5])
```

Use value to query the value of a nonlinear expression:

```
julia> set_start_value(x[1], 1.0)
julia> value(start_value, nested[1])
0.7456241416655579
```

julia> sin(sin(1.0))
0.7456241416655579

# 21.4 Create a nonlinear parameter

For nonlinear models only, JuMP offers a syntax for explicit "parameter" objects, which are constants in the model that can be efficiently updated between solves.

Nonlinear parameters are declared by using the @NLparameter macro and may be indexed by arbitrary sets analogously to JuMP variables and expressions.

The initial value of the parameter must be provided on the right-hand side of the == sign.

```
julia> model = Model();
julia> @variable(model, x);
julia> @NLparameter(model, p[i = 1:2] == i)
2-element Vector{NonlinearParameter}:
parameter[1] == 1.0
parameter[2] == 2.0
```

Create anonymous parameters using the value keyword:

```
julia> anon_parameter = @NLparameter(model, value = 1)
parameter[3] == 1.0
```

#### Info

A parameter is not an optimization variable. It must be fixed to a value with ==. If you want a parameter that is <= or >=, create a variable instead using @variable.

Use value and set\_value to query or update the value of a parameter.

```
julia> value.(p)
2-element Vector{Float64}:
 1.0
 2.0
julia> set_value(p[2], 3.0)
3.0
julia> value.(p)
2-element Vector{Float64}:
 1.0
 3.0
```

Nonlinear parameters must be used within nonlinear macros only.

# When to use a parameter

Nonlinear parameters are useful when solving nonlinear models in a sequence:

```
using JuMP, Ipopt
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, z)
@NLparameter(model, x == 1.0)
@NLobjective(model, Min, (z - x)^2)
optimize!(model)
@show value(z) # Equals 1.0.
Now, update the value of x to solve a different problem.
set_value(x, 5.0)
optimize!(model)
@show value(z) # Equals 5.0
```

value(z) = 1.0value(z) = 5.0

# Info

Using nonlinear parameters can be faster than creating a new model from scratch with updated data because JuMP is able to avoid repeating a number of steps in processing the model before handing it off to the solver.

# 21.5 Syntax notes

The syntax accepted in nonlinear macros is more restricted than the syntax for linear and quadratic macros. We note some important points below.

# Scalar operations only

Except for the splatting syntax discussed below, all expressions must be simple scalar operations. You cannot use dot, matrix-vector products, vector slices, etc.

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @variable(model, y);
julia> c = [1, 2];
julia> @NLobjective(model, Min, c' * x + 3y)
ERROR: Unexpected array [1 2] in nonlinear expression. Nonlinear expressions may contain only
↔ scalar expressions.
[...]
```

Translate vector operations into explicit sum() operations:

julia> @NLobjective(model, Min, sum(c[i] \* x[i] for i = 1:2) + 3y)

Or use an @expression:

```
julia> @expression(model, expr, c' * x)
x[1] + 2 x[2]
```

julia> @NLobjective(model, Min, expr + 3y)

# Splatting

The splatting operator ... is recognized in a very restricted setting for expanding function arguments. The expression splatted can be *only* a symbol. More complex expressions are not recognized.

```
julia> model = Model();
julia> @variable(model, x[1:3]);
julia> @NLconstraint(model, *(x...) <= 1.0)
x[1] * x[2] * x[3] - 1.0 ≤ 0
julia> @NLconstraint(model, *((x / 2)...) <= 0.0)
ERROR: Unsupported use of the splatting operator. JuMP supports splatting only symbols. For
↔ example, `x...` is ok, but `(x + 1)...`, `[x; y]...` and `g(f(y)...)` are not.
```

# 21.6 User-defined Functions

JuMP natively supports the set of univariate and multivariate functions recognized by the MOI.Nonlinear submodule. In addition to this list of functions, it is possible to register custom *user-defined* nonlinear functions. User-defined functions can be used anywhere in <code>@NLobjective</code>, <code>@NLconstraint</code>, and <code>@NLexpression</code>.

JuMP will attempt to automatically register functions it detects in your nonlinear expressions, which usually means manually registering a function is not needed. Two exceptions are if you want to provide custom derivatives, or if the function is not available in the scope of the nonlinear expression.

#### Warning

User-defined functions must return a scalar output. For a work-around, see User-defined operators with vector outputs.

#### Automatic differentiation

JuMP does not support black-box optimization, so all user-defined functions must provide derivatives in some form. Fortunately, JuMP supports **automatic differentiation of user-defined functions**, a feature to our knowledge not available in any comparable modeling systems.

#### Info

Automatic differentiation is *not* finite differencing. JuMP's automatically computed derivatives are not subject to approximation error.

JuMP uses ForwardDiff.jl to perform automatic differentiation; see the ForwardDiff.jl documentation for a description of how to write a function suitable for automatic differentiation.

# Common mistakes when writing a user-defined function

#### Warning

Get an error like No method matching Float64(::ForwardDiff.Dual)? Read this section, and see the guidelines at ForwardDiff.jl.

The most common error is that your user-defined function is not generic with respect to the number type, that is, don't assume that the input to the function is Float64.

f(x::Float64) = 2 \* x # This will not work.
f(x::Real) = 2 \* x # This is good.
f(x) = 2 \* x # This is also good.

Another reason you may encounter this error is if you create arrays inside your function which are Float64.

```
function bad_f(x...)
y = zeros(length(x)) # This constructs an array of `Float64`!
for i = 1:length(x)
y[i] = x[i]^i
end
return sum(y)
```

# end function good\_f(x::T...) where {T<:Real} y = zeros(T, length(x)) # Construct an array of type `T` instead! for i = 1:length(x) y[i] = x[i]^i end return sum(y) end</pre>

# **Register a function**

To register a user-defined function with derivatives computed by automatic differentiation, use the register method as in the following example:

```
square(x) = x^2
f(x, y) = (x - 1)^2 + (y - 2)^2
model = Model()
register(model, :square, 1, square; autodiff = true)
register(model, :my_f, 2, f; autodiff = true)
@variable(model, x[1:2] >= 0.5)
@NLobjective(model, Min, my_f(x[1], square(x[2])))
```

The above code creates a JuMP model with the objective function  $(x[1] - 1)^2 + (x[2]^2 - 2)^2$ . The arguments to register are:

- 1. The model for which the functions are registered.
- 2. A Julia symbol object which serves as the name of the user-defined function in JuMP expressions.
- 3. The number of input arguments that the function takes.
- 4. The Julia method which computes the function
- 5. A flag to instruct JuMP to compute exact gradients automatically.

## Тір

The symbol :my\_f doesn't have to match the name of the function f. However, it's more readable if it does. Make sure you use my\_f and not f in the macros.

#### Warning

User-defined functions cannot be re-registered and will not update if you modify the underlying Julia function. If you want to change a user-defined function between solves, rebuild the model or use a different name. To use a different name programmatically, see Raw expression input.

## **Register a function and gradient**

Forward-mode automatic differentiation as implemented by ForwardDiff.jl has a computational cost that scales linearly with the number of input dimensions. As such, it is not the most efficient way to compute gradients of user-defined functions if the number of input arguments is large. In this case, users may want to provide their own routines for evaluating gradients.

#### **Univariate functions**

For univariate functions, the gradient function ∇f returns a number that represents the first-order derivative:

```
f(x) = x^2
∇f(x) = 2x
model = Model()
register(model, :my_square, 1, f, ∇f; autodiff = true)
@variable(model, x >= 0)
@NLobjective(model, Min, my_square(x))
```

If autodiff = true, JuMP will use automatic differentiation to compute the hessian.

# **Multivariate functions**

For multivariate functions, the gradient function  $\nabla f$  must take a gradient vector as the first argument that is filled in-place:

```
f(x, y) = (x - 1)^2 + (y - 2)^2
function ∇f(g::AbstractVector{T}, x::T, y::T) where {T}
 g[1] = 2 * (x - 1)
 g[2] = 2 * (y - 2)
 return
end
model = Model()
register(model, :my_square, 2, f, ∇f)
@variable(model, x[1:2] >= 0)
```

#### Warning

Make sure the first argument to  $\nabla f$  supports an AbstractVector, and do not assume the input is Float64.

# Register a function, gradient, and hessian

@NLobjective(model, Min, my\_square(x[1], x[2]))

You can also register a function with the second-order derivative information, which is a scalar for univariate functions, and a symmetric matrix for multivariate functions.

#### **Univariate functions**

Pass a function which returns a number representing the second-order derivative:

```
f(x) = x^2

\nablaf(x) = 2x

V²f(x) = 2
model = Model()
register(model, :my_square, 1, f, \nable(f, \nable(''))
@variable(model, x >= 0)
@NLobjective(model, Min, my_square(x))
```

# **Multivariate functions**

For multivariate functions, the hessian function  $\nabla^2 f$  must take an AbstractMatrix as the first argument, the lower-triangular of which is filled in-place:

```
f(x...) = (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2
function ∇f(g, x...)
 g[1] = 400 * x[1]^3 - 400 * x[1] * x[2] + 2 * x[1] - 2
 g[2] = 200 * (x[2] - x[1]^2)
 return
end
function \nabla^2 f(H, x...)
 H[1, 1] = 1200 * x[1]^2 - 400 * x[2] + 2
 # H[1, 2] = -400 * x[1] <-- Not needed. Fill the lower-triangular only.
 H[2, 1] = -400 * x[1]
 H[2, 2] = 200.0
 return
end
model = Model()
register(model, :rosenbrock, 2, f, \nabla f, \nabla^2 f)
@variable(model, x[1:2])
@NLobjective(model, Min, rosenbrock(x[1], x[2]))
```

#### Warning

You may assume the Hessian matrix H is initialized with zeros, and because H is symmetric, you need only to fill in the non-zero of the lower-triangular terms. The matrix type passed in as H depends on the automatic differentiation system, so make sure the first argument to the Hessian function supports an AbstractMatrix (it may be something other than Matrix{Float64}). However, you may assume only that H supports size(H) and setindex!. Finally, the matrix is treated as dense, so the performance will be poor on functions with high-dimensional input.

### User-defined functions with vector inputs

User-defined functions which take vectors as input arguments (for example, f(x::Vector)) are *not* supported. Instead, use Julia's splatting syntax to create a function with scalar arguments. For example, instead of

```
f(x::Vector) = sum(x[i]^i for i in 1:length(x))
```

```
f(x...) = sum(x[i]^i for i in 1:length(x))
```

This function f can be used in a JuMP model as follows:

model = Model()
@variable(model, x[1:5] >= 0)
f(x...) = sum(x[i]^i for i in 1:length(x))
register(model, :f, 5, f; autodiff = true)
@NLobjective(model, Min, f(x...))

# Тір

Make sure to read the syntax restrictions of Splatting.

# 21.7 Factors affecting solution time

The execution time when solving a nonlinear programming problem can be divided into two parts, the time spent in the optimization algorithm (the solver) and the time spent evaluating the nonlinear functions and corresponding derivatives. Ipopt explicitly displays these two timings in its output, for example:

Total	CPU secs	in 1	IPOPT	(w/o	function	evaluations)	=	7.412
Total	CPU secs	in M	NLP fu	inctio	n evaluat	ions	=	2.083

For Ipopt in particular, one can improve the performance by installing advanced sparse linear algebra packages, see Installation Guide. For other solvers, see their respective documentation for performance tips.

The function evaluation time, on the other hand, is the responsibility of the modeling language. JuMP computes derivatives by using reverse-mode automatic differentiation with graph coloring methods for exploiting sparsity of the Hessian matrix. As a conservative bound, JuMP's performance here currently may be expected to be within a factor of 5 of AMPL's. Our paper in SIAM Review has more details.

# 21.8 Querying derivatives from a JuMP model

For some advanced use cases, one may want to directly query the derivatives of a JuMP model instead of handing the problem off to a solver. Internally, JuMP implements the MOI.AbstractNLPEvaluator interface. To obtain an NLP evaluator object from a JuMP model, use NLPEvaluator. index returns the MOI.VariableIndex corresponding to a JuMP variable. MOI.VariableIndex itself is a type-safe wrapper for Int64 (stored in the .value field.)

For example:

```
julia> raw_index(v::MOI.VariableIndex) = v.value
raw_index (generic function with 1 method)
julia> model = Model();
julia> @variable(model, x)
x
```

```
julia> @variable(model, y)
у
julia> @NLobjective(model, Min, sin(x) + sin(y))
julia> values = zeros(2)
2-element Vector{Float64}:
0.0
0.0
julia> x_index = raw_index(JuMP.index(x))
1
julia> y_index = raw_index(JuMP.index(y))
2
julia> values[x_index] = 2.0
2.0
julia> values[y_index] = 3.0
3.0
julia> d = NLPEvaluator(model)
Nonlinear. Evaluator with available features:
 * :Grad
 * :Jac
 * :JacVec
 * :Hess
 * :HessVec
 * :ExprGraph
julia> MOI.initialize(d, [:Grad])
julia> MOI.eval_objective(d, values)
1.0504174348855488
julia> sin(2.0) + sin(3.0)
1.0504174348855488
julia> \nabla f = zeros(2)
2-element Vector{Float64}:
0.0
0.0
julia> MOI.eval_objective_gradient(d, ∇f, values)
julia> \forall f[x_index], \forall f[y_index]
(-0.4161468365471424, -0.9899924966004454)
julia> cos(2.0), cos(3.0)
(-0.4161468365471424, -0.9899924966004454)
```

Only nonlinear constraints (those added with @NLconstraint), and nonlinear objectives (added with @NLobjective) exist in the scope of the NLPEvaluator.
The NLPEvaluator does not evaluate derivatives of linear or quadratic constraints or objectives.

The index method applied to a nonlinear constraint reference object returns its index as a MOI.Nonlinear.ConstraintIndex. For example:

```
julia> model = Model();
julia> @variable(model, x);
julia> @NLconstraint(model, cons1, sin(x) <= 1);
julia> @NLconstraint(model, cons2, x + 5 == 10);
julia> typeof(cons1)
NonlinearConstraintRef{ScalarShape} (alias for ConstraintRef{GenericModel{Float64},

→ MathOptInterface.Nonlinear.ConstraintIndex, ScalarShape})
julia> index(cons1)
MathOptInterface.Nonlinear.ConstraintIndex(1)
julia> index(cons2)
MathOptInterface.Nonlinear.ConstraintIndex(2)
```

Note that for one-sided nonlinear constraints, JuMP subtracts any values on the right-hand side when computing expressions. In other words, one-sided nonlinear constraints are always transformed to have a right-hand side of zero.

This method of querying derivatives directly from a JuMP model is convenient for interacting with the model in a structured way, for example, for accessing derivatives of specific variables. For example, in statistical maximum likelihood estimation problems, one is often interested in the Hessian matrix at the optimal solution, which can be queried using the NLPEvaluator.

#### 21.9 Raw expression input

#### Warning

This section requires advanced knowledge of Julia's Expr. You should read the Expressions and evaluation section of the Julia documentation first.

In addition to the @NLexpression, @NLobjective and @NLconstraint macros, it is also possible to provide Julia Expr objects directly by using add\_nonlinear\_expression, set\_nonlinear\_objective and add\_nonlinear\_constraint.

This input form may be useful if the expressions are generated programmatically, or if you experience compilation issues with the macro input (see Known performance issues for more information).

#### Add a nonlinear expression

Use add\_nonlinear\_expression to add a nonlinear expression to the model.

```
julia> model = Model();
julia> @variable(model, x)
x
```

```
julia> expr = :($(x) + sin($(x)^2))
:(x + sin(x ^ 2))
```

julia> expr\_ref = add\_nonlinear\_expression(model, expr)
subexpression[1]: x + sin(x ^ 2.0)

#### This is equivalent to

julia> model = Model();

julia> @variable(model, x);

```
julia> expr_ref = @NLexpression(model, x + sin(x^2))
subexpression[1]: x + sin(x ^ 2.0)
```

#### Note

You must interpolate the variables directly into the expression expr.

#### Set the objective function

Use set\_nonlinear\_objective to set a nonlinear objective.

```
julia> model = Model();
julia> @variable(model, x);
julia> expr = :($(x) + $(x)^2)
:(x + x ^ 2)
```

julia> set\_nonlinear\_objective(model, MIN\_SENSE, expr)

This is equivalent to

julia> model = Model();

julia> @variable(model, x);

julia> @NLobjective(model, Min, x + x<sup>2</sup>)

# Note

You must use MIN\_SENSE or MAX\_SENSE instead of Min and Max.

## Add a constraint

Use add\_nonlinear\_constraint to add a nonlinear constraint.

```
julia> model = Model();
```

julia> @variable(model, x);

julia> expr = :(\$(x) + \$(x)^2) :(x + x ^ 2)

julia> add\_nonlinear\_constraint(model, :(\$(expr) <= 1)) (x + x ^ 2.0) - 1.0  $\leq$  0

This is equivalent to

julia> model = Model(); julia> @variable(model, x); julia> @NLconstraint(model, Min, x +  $x^2 \le 1$ ) (x + x ^ 2.0) - 1.0  $\le 0$ 

#### More complicated examples

Raw expression input is most useful when the expressions are generated programmatically, often in conjunction with user-defined functions.

As an example, we construct a model with the nonlinear constraints  $f(x) \le 1$ , where  $f(x) = x^2$  and  $f(x) = sin(x)^2$ :

```
julia> function main(functions::Vector{Function})
 model = Model()
 @variable(model, x)
 for (i, f) in enumerate(functions)
 f_sym = Symbol("f_$(i)")
 register(model, f_sym, 1, f; autodiff = true)
 add_nonlinear_constraint(model, :($(f_sym)($(x)) <= 1))</pre>
 end
 print(model)
 return
 end
main (generic function with 1 method)
julia> main([x -> x², x -> sin(x)²])
Feasibility
Subject to
f_1(x) - 1.0 \le 0
f_2(x) - 1.0 \le 0
```

As another example, we construct a model with the constraint  $x^2 + sin(x)^2 <= 1$ :

```
julia> function main(functions::Vector{Function})
 model = Model()
 @variable(model, x)
```

```
expr = Expr(:call, :+)
for (i, f) in enumerate(functions)
 f_sym = Symbol("f_$(i)")
 register(model, f_sym, 1, f; autodiff = true)
 push!(expr.args, :($(f_sym)($(x))))
 end
 add_nonlinear_constraint(model, :($(expr) <= 1))
 print(model)
 return
 end
main (generic function with 1 method)</pre>
```

```
julia> main([x -> x^2, x -> sin(x)^2])
Feasibility
Subject to
 (f_1(x) + f_2(x)) - 1.0 ≤ 0
```

## Registered functions with a variable number of arguments

User defined functions require a fixed number of input arguments. However, sometimes you will want to use a registered function like:

```
julia> f(x...) = sum(exp(x[i]^2) for i in 1:length(x));
```

with different numbers of arguments.

The solution is to register the same function f for each unique number of input arguments, making sure to use a unique name each time. For example:

```
julia> A = [[1], [1, 2], [2, 3, 4], [1, 3, 4, 5]];
julia> model = Model();
julia> @variable(model, x[1:5]);
julia> funcs = Set{Symbol}();
julia> for a in A
 key = Symbol("f$(length(a))")
 if !(key in funcs)
 push!(funcs, key)
 register(model, key, length(a), f; autodiff = true)
 end
 add_nonlinear_constraint(model, :($key($(x[a]...)) <= 1))</pre>
 end
julia> print(model)
Feasibility
Subject to
 f1(x[1]) - 1.0 \le 0
f_2(x[1], x[2]) - 1.0 \le 0
 f_3(x[2], x[3], x[4]) - 1.0 \le 0
 f4(x[1], x[3], x[4], x[5]) - 1.0 \le 0
```

# 21.10 Known performance issues

The macro-based input to JuMP's nonlinear interface can cause a performance issue if you:

- 1. write a macro with a large number (hundreds) of terms
- 2. call that macro from within a function instead of from the top-level in global scope.

The first issue does not depend on the number of resulting terms in the mathematical expression, but rather the number of terms in the Julia Expr representation of that expression. For example, the expression  $sum(x[i] for i in 1:1_000_000)$  contains one million mathematical terms, but the Expr representation is just a single sum.

The most common cause, other than a lot of tedious typing, is if you write a program that automatically writes a JuMP model as a text file, which you later execute. One example is MINLPlib.jl which automatically transpiled models in the GAMS scalar format into JuMP examples.

As a rule of thumb, if you are writing programs to automatically generate expressions for the JuMP macros, you should target the Raw expression input instead. For more information, read MathOptInterface Issue#1997.

Part IV

# **API Reference**

# **Chapter 22**

# Docstrings

# 22.1 JuMP

## JuMP

This page lists the public API of JuMP.

#### Info

This page is an unstructured list of the JuMP API. For a more structured overview, read the Manual or Tutorial parts of this documentation.

Load all of the public the API into the current scope with:

using JuMP

Alternatively, load only the module with:

import JuMP

and then prefix all calls with JuMP. to create JuMP.<NAME>.

# @build\_constraint

JuMP.@build\_constraint - Macro.

@build\_constraint(constraint\_expr)

Constructs a ScalarConstraint or VectorConstraint using the same machinery as @constraint but without adding the constraint to a model.

Constraints using broadcast operators like x .<= 1 are also supported and will create arrays of ScalarConstraint or VectorConstraint.

julia> model = Model();

julia> @variable(model, x);

#### julia> @build\_constraint(2x >= 1)

ScalarConstraint{AffExpr, MathOptInterface.GreaterThan{Float64}}(2 x, → MathOptInterface.GreaterThan{Float64}(1.0))

```
julia> model = Model();
```

julia> @variable(model, x[1:2]);

#### julia> @build\_constraint(x .>= 0)

2-element Vector{ScalarConstraint{AffExpr, MathOptInterface.GreaterThan{Float64}}: ScalarConstraint{AffExpr, MathOptInterface.GreaterThan{Float64}}(x[1], → MathOptInterface.GreaterThan{Float64}(-0.0)) ScalarConstraint{AffExpr, MathOptInterface.GreaterThan{Float64}}(x[2], → MathOptInterface.GreaterThan{Float64}(-0.0))

#### source

#### @constraint

JuMP.@constraint - Macro.

```
@constraint(model, expr, args...; kwargs...)
@constraint(model, [index_sets...], expr, args...; kwargs...)
@constraint(model, name, expr, args...; kwargs...)
@constraint(model, name[index_sets...], expr, args...; kwargs...)
```

Add a constraint described by the expression expr.

The name argument is optional. If index sets are passed, a container is built and the constraint may depend on the indices of the index sets.

The expression expr may be one of following forms:

- func in set, constraining the function func to belong to the set set, which is either a MOI. AbstractSet
  or one of the JuMP shortcuts like SecondOrderCone or PSDCone
- a <op> b, where <op> is one of ==,  $\geq$ , >=,  $\leq$ , <=
- l <= f <= u or u >= f >= l, constraining the expression f to lie between l and u
- $f(x) \perp x$ , which defines a complementarity constraint
- z --> {expr}, which defines an indicator constraint that activates when z is 1
- !z --> {expr}, which defines an indicator constraint that activates when z is 0
- z <--> {expr}, which defines a reified constraint
- expr := rhs, which defines a Boolean equality constraint

Broadcasted comparison operators like .== are also supported for the case when the left- and right-hand sides of the comparison operator are arrays.

JuMP extensions may additionally provide support for constraint expressions which are not listed here.

#### **Keyword arguments**

- base\_name: sets the name prefix used to generate constraint names. It corresponds to the constraint name for scalar constraints, otherwise, the constraint names are set to base\_name[...] for each index ....
- container = :Auto: force the container type by passing container = Array,

container = DenseAxisArray, container = SparseAxisArray, or any another container type which is supported by a JuMP extension.

 set\_string\_name::Bool = true: control whether to set the MOI.ConstraintName attribute. Passing set\_string\_name = false can improve performance.

Other keyword arguments may be supported by JuMP extensions.

```
julia> model = Model();
julia> @variable(model, x[1:3]);
julia> @variable(model, z, Bin);
julia> @constraint(model, x in SecondOrderCone())
[x[1], x[2], x[3]] ∈ MathOptInterface.SecondOrderCone(3)
julia> @constraint(model, [i in 1:3], x[i] == i)
3-element Vector{ConstraintRef{Model,
← MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↔ MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
x[1] = 1
x[2] = 2
 x[3] = 3
julia> @constraint(model, x .== [1, 2, 3])
3-element Vector{ConstraintRef{Model,
← MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
→ MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
x[1] = 1
x[2] = 2
 x[3] = 3
julia> @constraint(model, con name, 1 <= x[1] + x[2] <= 3)</pre>
con_name : x[1] + x[2] \in [1, 3]
julia> @constraint(model, con_perp[i in 1:3], x[i] - 1 \propto x[i])
3-element Vector{ConstraintRef{Model,
```

```
→ MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},

→ MathOptInterface.Complements}, VectorShape}}:

con_perp[1] : [x[1] - 1, x[1]] ∈ MathOptInterface.Complements(2)

con_perp[2] : [x[2] - 1, x[2]] ∈ MathOptInterface.Complements(2)
```

```
con_perp[3] : [x[3] - 1, x[3]] ∈ MathOptInterface.Complements(2)
julia> @constraint(model, z --> {x[1] >= 0})
z --> {x[1] ≥ 0}
julia> @constraint(model, !z --> {2 * x[2] <= 3})
!z --> {2 x[2] ≤ 3}
```

source

#### @constraints

JuMP.@constraints - Macro.

@constraints(model, args...)

Adds groups of constraints at once, in the same fashion as the @constraint macro.

The model must be the first argument, and multiple constraints can be added on multiple lines wrapped in a begin ... end block.

The macro returns a tuple containing the constraints that were defined.

#### Example

Subject to
sum\_to\_one[1] : y + z[1] = 1
sum\_to\_one[2] : y + z[2] = 1
sum\_to\_one[3] : y + z[3] = 1
x ≥ 1
-w + y ≤ 2

#### @expression

JuMP.@expression - Macro.

```
@expression(model::GenericModel, expression)
@expression(model::GenericModel, [index_sets...], expression)
@expression(model::GenericModel, name, expression)
@expression(model::GenericModel, name[index_sets...], expression)
```

Efficiently builds and returns an expression.

The name argument is optional. If index sets are passed, a container is built and the expression may depend on the indices of the index sets.

#### **Keyword arguments**

 container = :Auto: force the container type by passing container = Array, container = DenseAxisArray, container = SparseAxisArray, or any another container type which is supported by a JuMP extension.

## Example

```
julia> model = Model();
julia> @variable(model, x[1:5]);
julia> @expression(model, shared, sum(i * x[i] for i in 1:5))
x[1] + 2 x[2] + 3 x[3] + 4 x[4] + 5 x[5]
julia> shared
x[1] + 2 x[2] + 3 x[3] + 4 x[4] + 5 x[5]
```

In the same way as @variable, the second argument may define index sets, and those indices can be used in the construction of the expressions:

```
julia> model = Model();
```

julia> @variable(model, x[1:3]);

```
julia> @expression(model, expr[i = 1:3], i * sum(x[j] for j in 1:3))
3-element Vector{AffExpr}:
 x[1] + x[2] + x[3]
 2 x[1] + 2 x[2] + 2 x[3]
 3 x[1] + 3 x[2] + 3 x[3]
```

Anonymous syntax is also supported:

```
julia> model = Model();
julia> @variable(model, x[1:3]);
julia> expr = @expression(model, [i in 1:3], i * sum(x[j] for j in 1:3))
```

```
3-element Vector{AffExpr}:
x[1] + x[2] + x[3]
2 x[1] + 2 x[2] + 2 x[3]
3 x[1] + 3 x[2] + 3 x[3]
```

#### source

#### @expressions

JuMP.@expressions - Macro.

@expressions(model, args...)

Adds multiple expressions to model at once, in the same fashion as the dexpression macro.

The model must be the first argument, and multiple expressions can be added on multiple lines wrapped in a begin ... end block.

The macro returns a tuple containing the expressions that were defined.

#### Example

#### source

#### @force\_nonlinear

JuMP.@force\_nonlinear - Macro.

@force\_nonlinear(expr)

Change the parsing of expr to construct GenericNonlinearExpr instead of GenericAffExpr or GenericQuadExpr.

This macro works by walking expr and substituting all calls to +, -, \*, /, and ^ in favor of ones that construct GenericNonlinearExpr.

This macro will error if the resulting expression does not produce a GenericNonlinearExpr because, for example, it is used on an expression that does not use the basic arithmetic operators.

#### When to use this macro

In most cases, you should not use this macro.

Use this macro only if the intended output type is a GenericNonlinearExpr and the regular macro calls destroy problem structure, or in rare cases, if the regular macro calls introduce a large amount of intermediate variables, for example, because they promote types to a common quadratic expression.

#### Example

Use-case one: preserve problem structure.

```
julia> model = Model();
julia> @variable(model, x);
julia> @expression(model, (x - 0.1)^2)
x² - 0.2 x + 0.0100000000000002
julia> @expression(model, @force_nonlinear((x - 0.1)^2))
(x - 0.1) ^ 2
julia> (x - 0.1)^2
x² - 0.2 x + 0.01000000000000002
```

```
julia> @force_nonlinear((x - 0.1)^2)
(x - 0.1) ^ 2
```

#### Use-case two: reduce allocations

In this example, we know that x \* 2.0 \* (1 + x) \* x is going to construct a nonlinear expression. However, the default parsing first constructs:

- the GenericAffExpr a = x \* 2.0,
- another GenericAffExpr b = 1 + x
- the GenericQuadExpr c = a \* b
- a GenericNonlinearExpr\*(c, x)

In contrast, the modified parsing constructs:

- the GenericNonlinearExpr a = GenericNonlinearExpr(:+, 1, x)
- the GenericNonlinearExpr GenericNonlinearExpr(:\*, x, 2.0, a, x)

This results in significantly fewer allocations.

```
julia> model = Model();
julia> @variable(model, x);
julia> @expression(model, x * 2.0 * (1 + x) * x)
```

 $(2 x^{2} + 2 x) * x$ 

julia> @expression(model, @force\_nonlinear(x \* 2.0 \* (1 + x) \* x))
x \* 2.0 \* (1 + x) \* x

julia> @allocated @expression(model, x \* 2.0 \* (1 + x) \* x)
3680

```
julia> @allocated @expression(model, @force_nonlinear(x * 2.0 * (1 + x) * x))
768
```

source

## **@objective**

JuMP.@objective - Macro.

@objective(model::GenericModel, sense, func)

Set the objective sense to sense and objective function to func.

#### sense

The objective sense must be either be the literals Min or Max, or one of the three MOI.OptimizationSense enum values (MIN\_SENSE, MAX\_SENSE, or FEASIBILITY\_SENSE).

In order to set the sense programmatically, that is, when sense is a Julia variable whose value is the sense, you must use a MOI.OptimizationSense.

# FEASIBILITY\_SENSE

FEASIBILITY\_SENSE implies that there is no objective function. Therefore, you should not set sense to FEASIBILITY\_SENSE with a non-zero func.

To reset the model to FEASIBILITY\_SENSE, do @objective(model, FEASIBILITY\_SENSE, 0), or use set\_objective\_sense: set\_objective\_sense(model, FEASIBILITY\_SENSE).

#### Example

Minimize the value of the variable x, do:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @objective(model, Min, x)
x
```

Maximize the value of the affine expression 2x - 1:

```
julia> model = Model();
julia> @variable(model, x)
```

Х

julia> @objective(model, Max, 2x - 1)
2 x - 1

Set the objective sense programmatically:

```
julia> model = Model();
julia> @variable(model, x)
x
julia> sense = MIN_SENSE
MIN_SENSE::OptimizationSense = 0
julia> @objective(model, sense, x^2 - 2x + 1)
x² - 2 x + 1
```

Remove an objective function:

```
julia> model = Model();
julia> @variable(model, x >= 0);
julia> @objective(model, Min, 2x + 1)
2 x + 1
julia> print(model)
Min 2 x + 1
Subject to
 x ≥ 0
julia> @objective(model, FEASIBILITY_SENSE, 0)
0
julia> print(model)
Feasibility
Subject to
 x ≥ 0
```

#### source

## @operator

JuMP.@operator - Macro.

```
@operator(model, operator, dim, f[, \nablaf[, \nabla²f]])
```

Add the nonlinear operator operator in model with dim arguments, and create a new NonlinearOperator object called operator in the current scope.

The function f evaluates the operator and must return a scalar.

The optional function  $\nabla f$  evaluates the first derivative, and the optional function  $\nabla^2 f$  evaluates the second derivative.

 $\nabla^2 f$  may be provided only if  $\nabla f$  is also provided.

#### **Univariate syntax**

If dim == 1, then the method signatures of each function must be:

- f(::T)::T where {T<:Real}
- ∇f(::T)::T where {T<:Real}
- $\nabla^2 f(::T)::T$  where {T<:Real}

## Multivariate syntax

If dim > 1, then the method signatures of each function must be:

- f(x::T...)::T where {T<:Real}
- ∇f(g::AbstractVector{T}, x::T...)::Nothing where {T<:Real}
- $\nabla^2 f(H::AbstractMatrix{T}, x::T...)::Nothing where {T<:Real}$

Where the gradient vector g and Hessian matrix H are filled in-place. For the Hessian, you must fill in the non-zero lower-triangular entries only. Setting an off-diagonal upper-triangular element may error.

```
julia> model = Model();
julia> @variable(model, x)
Х
julia> f(x::Float64) = x^2
f (generic function with 1 method)
julia> \[\nbox{f(x::Float64)} = 2 * x
\nabla f (generic function with 1 method)
julia> \nabla^2 f(x::Float64) = 2.0
\nabla^2 f (generic function with 1 method)
julia> @operator(model, op_f, 1, f, \nabla f, \nabla^2 f)
NonlinearOperator(f, :op_f)
julia> @objective(model, Min, op_f(x))
op_f(x)
julia> op_f(2.0)
4.0
julia> model[:op_f]
NonlinearOperator(f, :op_f)
julia> model[:op_f](x)
op_f(x)
```

#### **Non-macro version**

This macro is provided as helpful syntax that matches the style of the rest of the JuMP macros. However, you may also add operators outside the macro using add\_nonlinear\_operator. For example:

julia> model = Model(); julia> f(x) = x^2 f (generic function with 1 method)

julia> @operator(model, op\_f, 1, f) NonlinearOperator(f, :op\_f)

#### is equivalent to

```
julia> model = Model();
```

julia> f(x) = x^2
f (generic function with 1 method)

```
julia> op_f = model[:op_f] = add_nonlinear_operator(model, 1, f; name = :op_f)
NonlinearOperator(f, :op_f)
```

#### source

#### @variable

JuMP.@variable - Macro.

@variable(model, expr, args..., kw\_args...)

Add a variable to the model model described by the expression expr, the positional arguments args and the keyword arguments kw\_args.

#### Anonymous and named variables

expr must be one of the forms:

- Omitted, like @variable(model), which creates an anonymous variable
- A single symbol like @variable(model, x)
- A container expression like @variable(model, x[i=1:3])
- An anonymous container expression like @variable(model, [i=1:3])

#### Bounds

In addition, the expression can have bounds, such as:

- @variable(model, x >= 0)
- @variable(model, x <= 0)
- @variable(model, x == 0)

• @variable(model, 0 <= x <= 1)

and bounds can depend on the indices of the container expressions:

• @variable(model, -i <= x[i=1:3] <= i)

#### Sets

You can explicitly specify the set to which the variable belongs:

• @variable(model, x in MOI.Interval(0.0, 1.0))

For more information on this syntax, read Variables constrained on creation.

#### **Positional arguments**

The recognized positional arguments in args are the following:

- Bin: restricts the variable to the MOI.ZeroOne set, that is, {0, 1}. For example, @variable(model, x, Bin). Note: you cannot use @variable(model, Bin), use the binary keyword instead.
- Int: restricts the variable to the set of integers, that is, ..., -2, -1, 0, 1, 2, ... For example, @variable(model, x, Int). Note: you cannot use @variable(model, Int), use the integer keyword instead.
- Symmetric: Only available when creating a square matrix of variables, that is when expr is of the form varname[1:n,1:n] or varname[i=1:n,j=1:n], it creates a symmetric matrix of variables.
- PSD: A restrictive extension to Symmetric which constraints a square matrix of variables to Symmetric and constrains to be positive semidefinite.

#### **Keyword arguments**

Four keyword arguments are useful in all cases:

- base\_name: Sets the name prefix used to generate variable names. It corresponds to the variable name for scalar variable, otherwise, the variable names are set to base\_name[...] for each index
   ... of the axes axes.
- start::Float64: specify the value passed to set\_start\_value for each variable
- container: specify the container type. See Forcing the container type for more information.
- set\_string\_name::Bool = true: control whether to set the MOI.VariableName attribute. Passing set\_string\_name = false can improve performance.

Other keyword arguments are needed to disambiguate sitations with anonymous variables:

- lower\_bound::Float64: an alternative to x >= lb, sets the value of the variable lower bound.
- upper\_bound::Float64: an alternative to x <= ub, sets the value of the variable upper bound.
- binary::Bool: an alternative to passing Bin, sets whether the variable is binary or not.
- integer::Bool: an alternative to passing Int, sets whether the variable is integer or not.
- set::MOI.AbstractSet: an alternative to using x in set
- variable type: used by JuMP extensions. See Extend @variable for more information.

#### Example

The following are equivalent ways of creating a variable x of name x with lower bound 0:

```
julia> model = Model();
julia> @variable(model, x >= 0)
x
```

```
julia> model = Model();
```

```
julia> @variable(model, x, lower_bound = \theta) x
```

```
julia> model = Model();
julia> x = @variable(model, base_name = "x", lower_bound = 0)
x
```

Other examples:

```
julia> model = Model();
julia> @variable(model, x[i=1:3] <= i, Int, start = sqrt(i), lower_bound = -i)</pre>
3-element Vector{VariableRef}:
x[1]
 x[2]
 x[3]
julia> @variable(model, y[i=1:3], container = DenseAxisArray, set = MOI.ZeroOne())
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
 Dimension 1, Base.OneTo(3)
And data, a 3-element Vector{VariableRef}:
y[1]
y[2]
y[3]
julia> @variable(model, z[i=1:3], set_string_name = false)
3-element Vector{VariableRef}:
[7]
[8]
_[9]
```

#### source

## @variables

JuMP.@variables - Macro.

@variables(model, args...)

Adds multiple variables to model at once, in the same fashion as the @variable macro.

The model must be the first argument, and multiple variables can be added on multiple lines wrapped in a begin ... end block.

The macro returns a tuple containing the variables that were defined.

#### Example

#### Note

Keyword arguments must be contained within parentheses (refer to the example above).

#### source

#### add\_bridge

JuMP.add\_bridge - Function.

```
add_bridge(
 model::GenericModel{T},
 BT::Type{<:MOI.Bridges.AbstractBridge};
 coefficient_type::Type{S} = T,
) where {T,S}</pre>
```

Add BT{T} to the list of bridges that can be used to transform unsupported constraints into an equivalent formulation using only constraints supported by the optimizer.

See also: remove\_bridge.

#### add\_constraint

JuMP.add\_constraint - Function.

```
add_constraint(
 model::GenericModel,
 con::AbstractConstraint,
 name::String= "",
)
```

This method should only be implemented by developers creating JuMP extensions. It should never be called by users of JuMP.

source

#### add\_nonlinear\_operator

JuMP.add\_nonlinear\_operator - Function.

```
add_nonlinear_operator(
 model::Model,
 dim::Int,
 f::Function,
 [∇f::Function,]
 [∇²f::Function];
 [name::Symbol = Symbol(f),]
)
```

Add a new nonlinear operator with dim input arguments to model and associate it with the name name.

The function f evaluates the operator and must return a scalar.

The optional function  $\nabla f$  evaluates the first derivative, and the optional function  $\nabla^2 f$  evaluates the second derivative.

 $\nabla^2 f$  may be provided only if  $\nabla f$  is also provided.

#### Univariate syntax

If dim == 1, then the method signatures of each function must be:

- f(::T)::T where {T<:Real}
- ∇f(::T)::T where {T<:Real}
- $\nabla^2 f(::T)::T$  where {T<:Real}

#### Multivariate syntax

If dim > 1, then the method signatures of each function must be:

- f(x::T...)::T where {T<:Real}
- $\nabla f(g::AbstractVector{T}, x::T...)::Nothing where {T<:Real}$
- ∇<sup>2</sup>f(H::AbstractMatrix{T}, x::T...)::Nothing where {T<:Real}

Where the gradient vector g and Hessian matrix H are filled in-place. For the Hessian, you must fill in the non-zero lower-triangular entries only. Setting an off-diagonal upper-triangular element may error.

#### Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> f(x::Float64) = x^2
f (generic function with 1 method)
julia> \nabla f(x::Float64) = 2 * x
\nabla f (generic function with 1 method)
julia> \nabla^2 f(x::Float64) = 2.0
\nabla^2 f (generic function with 1 method)
julia> op_f = add_nonlinear_operator(model, 1, f, \nabla f, \nabla^2 f)
NonlinearOperator(f, :f)
julia> @objective(model, Min, op_f(x))
```

**julia**> op\_f(2.0)

#### source

f(x)

4.0

#### add\_to\_expression!

JuMP.add\_to\_expression! - Function.

add\_to\_expression!(expression, terms...)

Updates expression in-place to expression + (\*)(terms...).

This is typically much more efficient than expression += (\*)(terms...) because it avoids the temorary allocation of the right-hand side term.

For example, add\_to\_expression! (expression, a, b) produces the same result as expression += a\*b, and add\_to\_expression! (expression, a) produces the same result as expression += a.

#### When to implement

Only a few methods are defined, mostly for internal use, and only for the cases when:

- 1. they can be implemented efficiently
- expression is capable of storing the result. For example, add\_to\_expression!(::AffExpr, ::GenericVariableRef, ::GenericVariableRef) is not defined because a GenericAffExpr cannot store the product of two variables.

#### Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> expr = 2 + x
x + 2
```

```
julia> add_to_expression!(expr, 3, x)
4 x + 2
```

julia> expr
4 x + 2

```
julia> model = Model();
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
julia> @expression(model, ex1, sum(x))
 x[1] + x[2]
julia> @expression(model, ex2, 2 * sum(x))
2 x[1] + 2 x[2]
julia> add_to_expression!(ex1, ex2)
3 x[1] + 3 x[2]
julia> ex1
3 x[1] + 3 x[2]
```

julia> ex2
2 x[1] + 2 x[2]

source

#### add\_to\_function\_constant

JuMP.add\_to\_function\_constant - Function.

add\_to\_function\_constant(constraint::ConstraintRef, value)

Add value to the function constant term of constraint.

Note that for scalar constraints, JuMP will aggregate all constant terms onto the right-hand side of the constraint so instead of modifying the function, the set will be translated by -value. For example, given a constraint  $2x \le 3$ , add\_to\_function\_constant(c, 4) will modify it to  $2x \le -1$ .

#### Example

For scalar constraints, the set is translated by -value:

```
julia> model = Model();
```

julia> @variable(model, x);

```
julia> @constraint(model, con, θ <= 2x - 1 <= 2)
con : 2 x ∈ [1, 3]
```

julia> add\_to\_function\_constant(con, 4)

**julia> con** con : 2 x ∈ [-3, -1]

For vector constraints, the constant is added to the function:

julia> model = Model();

julia> @variable(model, x);

julia> @variable(model, y);

julia> @constraint(model, con, [x + y, x, y] in SecondOrderCone())
con : [x + y, x, y] ∈ MathOptInterface.SecondOrderCone(3)

julia> add\_to\_function\_constant(con, [1, 2, 2])

julia> con con : [x + y + 1, x + 2, y + 2] ∈ MathOptInterface.SecondOrderCone(3)

#### source

## add\_variable

JuMP.add variable - Function.

add\_variable(m::GenericModel, v::AbstractVariable, name::String = "")

This method should only be implemented by developers creating JuMP extensions. It should never be called by users of JuMP.

source

## all\_constraints

JuMP.all\_constraints - Function.

all\_constraints(model::GenericModel, function\_type, set\_type)::Vector{<:ConstraintRef}</pre>

Return a list of all constraints currently in the model where the function has type function\_type and the set has type set\_type. The constraints are ordered by creation time.

See also list\_of\_constraint\_types and num\_constraints.

#### Example

```
julia> model = Model();
julia> @variable(model, x >= 0, Bin);
julia> @constraint(model, 2x <= 1);</pre>
julia> all_constraints(model, VariableRef, MOI.GreaterThan{Float64})
1-element Vector{ConstraintRef{Model,
→ MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
→ MathOptInterface.GreaterThan{Float64}}, ScalarShape}}:
 X \ge 0
julia> all_constraints(model, VariableRef, MOI.ZeroOne)
1-element Vector{ConstraintRef{Model,
← MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne},
\hookrightarrow ScalarShape}}:
x binary
julia> all_constraints(model, AffExpr, MOI.LessThan{Float64})
1-element Vector{ConstraintRef{Model,
← MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↔ MathOptInterface.LessThan{Float64}}, ScalarShape}}:
2 X ≤ 1
```

#### source

```
all_constraints(
 model::GenericModel;
 include_variable_in_set_constraints::Bool,
)::Vector{ConstraintRef}
```

Return a list of all constraints in model.

If include\_variable\_in\_set\_constraints == true, then VariableRef constraints such as VariableRefin-Integer are included. To return only the structural constraints (for example, the rows in the constraint matrix of a linear program), pass include\_variable\_in\_set\_constraints = false.

```
julia> model = Model();
julia> @variable(model, x >= 0, Int);
julia> @constraint(model, 2x <= 1);
julia> @NLconstraint(model, x^2 <= 1);
julia> all_constraints(model; include_variable_in_set_constraints = true)
```

```
4-element Vector{ConstraintRef}:
2 x ≤ 1
x ≥ 0
x integer
x ^ 2.0 - 1.0 ≤ 0
julia> all_constraints(model; include_variable_in_set_constraints = false)
2-element Vector{ConstraintRef}:
2 x ≤ 1
x ^ 2.0 - 1.0 ≤ 0
```

#### **Performance considerations**

Note that this function is type-unstable because it returns an abstractly typed vector. If performance is a problem, consider using list\_of\_constraint\_types and a function barrier. See the Performance tips for extensions section of the documentation for more details.

source

#### all\_variables

JuMP.all\_variables - Function.

all\_variables(model::GenericModel{T}):::Vector{GenericVariableRef{T}} where {T}

Returns a list of all variables currently in the model. The variables are ordered by creation time.

#### Example

#### source

anonymous\_name

JuMP.anonymous\_name - Function.

```
anonymous_name(::MIME, x::AbstractVariableRef)
```

The name to use for an anonymous variable x when printing.

```
julia> model = Model();
julia> x = @variable(model);
julia> anonymous_name(MIME("text/plain"), x)
"_[1]"
```

source

#### assert\_is\_solved\_and\_feasible

JuMP.assert\_is\_solved\_and\_feasible - Function.

assert\_is\_solved\_and\_feasible(model::GenericModel; kwargs...)

A function calls is\_solved\_and\_feasible and, if the return is false, errors with an informative error message describing the state of the solver.

#### **Keyword arguments**

See is\_solved\_and\_feasible for a description of all keyword arguments.

#### Example

```
julia> import Ipopt
julia> model = Model(Ipopt.Optimizer);
julia> is_solved_and_feasible(model)
false
julia> assert_is_solved_and_feasible(model)
ERROR: The model was not solved correctly. Here is the output of `solution_summary` to help
\hookrightarrow debug why this happened:
solution_summary(; result = 1, verbose = false)
- solver_name
 : Ipopt
- Termination
| | termination_status : OPTIMIZE_NOT_CALLED
| + result_count : 0
| L raw_status : optimize not called
L Solution (result = 1)

 primal_status : NO_SOLUTION
 dual_status : NO_SOLUTION
Stacktrace:
[...]
```

#### source

#### backend

JuMP.backend - Function.

#### backend(model::GenericModel)

Return the lower-level MathOptInterface model that sits underneath JuMP. This model depends on which operating mode JuMP is in (see mode).

- If JuMP is in DIRECT mode (that is, the model was created using direct\_model), the backend will be the optimizer passed to direct\_model.
- If JuMP is in MANUAL or AUTOMATIC mode, the backend is a MOI.Utilities.CachingOptimizer.

Use index to get the index of a variable or constraint in the backend model.

#### Warning

This function should only be used by advanced users looking to access low-level MathOptInterface or solver-specific functionality.

#### Notes

If JuMP is not in DIRECT mode, the type returned by backend may change between any JuMP releases. Therefore, only use the public API exposed by MathOptInterface, and do not access internal fields. If you require access to the innermost optimizer, see unsafe\_backend. Alternatively, use direct\_model to create a JuMP model in DIRECT mode.

See also: unsafe\_backend.

#### Example

julia> import HiGHS

julia> model = direct\_model(HiGHS.Optimizer());

julia> set\_silent(model)

julia> @variable(model, x >= 0)
x

julia> highs = backend(model)
A HiGHS model with 1 columns and 0 rows.

```
julia> index(x)
MOI.VariableIndex(1)
```

## source

#### barrier\_iterations

JuMP.barrier\_iterations - Function.

barrier\_iterations(model::GenericModel)

If available, returns the cumulative number of barrier iterations during the most-recent optimization (the MOI.BarrierIterations attribute).

Throws a MOI.GetAttributeNotAllowed error if the attribute is not implemented by the solver.

#### Example

```
julia> import HiGHS
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> optimize!(model)
julia> barrier_iterations(model)
0
```

#### source

# bridge\_constraints

JuMP.bridge\_constraints - Function.

bridge\_constraints(model::GenericModel)

When in direct mode, return false.

When in manual or automatic mode, return a Bool indicating whether the optimizer is set and unsupported constraints are automatically bridged to equivalent supported constraints when an appropriate transformation is available.

#### Example

```
julia> import Ipopt
julia> model = Model(Ipopt.Optimizer);
julia> bridge_constraints(model)
true
julia> model = Model(Ipopt.Optimizer; add_bridges = false);
julia> bridge_constraints(model)
false
```

#### source

## build\_constraint

JuMP.build\_constraint - Function.

```
build_constraint(error_fn::Function, func, set, args...; kwargs...)
```

This method should only be implemented by developers creating JuMP extensions. It should never be called by users of JuMP.

source

## build\_variable

JuMP.build\_variable - Function.

```
build_variable(
 error_fn::Function,
 info::VariableInfo,
 args...;
 kwargs...,
)
```

Return a new AbstractVariable object.

This method should only be implemented by developers creating JuMP extensions. It should never be called by users of JuMP.

#### Arguments

- error\_fn: a function to call instead of error. error\_fn annotates the error message with additional information for the user.
- info: an instance of VariableInfo. This has a variety of fields relating to the variable such as info.lower\_bound and info.binary.
- args: optional additional positional arguments for extending the @variable macro.
- kwargs: optional keyword arguments for extending the @variable macro.

#### See also: @variable

#### Warning

Extensions should define a method with ONE positional argument to dispatch the call to a different method. Creating an extension that relies on multiple positional arguments leads to MethodErrors if the user passes the arguments in the wrong order.

#### Example

@variable(model, x, Foo)

#### will call

build\_variable(error\_fn::Function, info::VariableInfo, ::Type{Foo})

Passing special-case positional arguments such as Bin, Int, and PSD is okay, along with keyword arguments:

```
@variable(model, x, Int, Foo(), mykwarg = true)
or
@variable(model, x, Foo(), Int, mykwarg = true)
```

will call

build\_variable(error\_fn::Function, info::VariableInfo, ::Foo; mykwarg)

and info.integer will be true.

Note that the order of the positional arguments does not matter.

source

#### callback\_node\_status

JuMP.callback\_node\_status - Function.

```
callback_node_status(cb_data, model::GenericModel)
```

Return an MOI.CallbackNodeStatusCode enum, indicating if the current primal solution available from callback\_value is integer feasible.

```
julia> import Gurobi
julia> model = Model(Gurobi.Optimizer);
julia> set_silent(model)
julia> @variable(model, x <= 10, Int);</pre>
julia> @objective(model, Max, x);
julia> function my_callback_function(cb_data, cb_where)
 status = callback_node_status(cb_data, model)
 if status == MOI.CALLBACK_NODE_STATUS_INTEGER
 println("Status is: ", status)
 end
 return
 end
my_callback_function (generic function with 1 method)
julia> set_attribute(model, Gurobi.CallbackFunction(), my_callback_function)
julia> optimize!(model)
Status is: CALLBACK_NODE_STATUS_INTEGER
```

#### callback\_value

JuMP.callback\_value - Function.

```
callback_value(cb_data, x::GenericVariableRef)
callback_value(cb_data, x::Union{GenericAffExpr,GenericQuadExpr})
```

Return the primal solution of x inside a callback.

cb\_data is the argument to the callback function, and the type is dependent on the solver.

Use callback\_node\_status to check whether a solution is available.

#### Example

```
julia> import Gurobi
julia> model = Model(Gurobi.Optimizer);
julia> set_silent(model)
julia> @variable(model, x <= 10, Int);</pre>
julia> @objective(model, Max, x);
julia> function my_callback_function(cb_data, cb_where)
 status = callback_node_status(cb_data, model)
 if status == MOI.CALLBACK NODE STATUS INTEGER
 Gurobi.load_callback_variable_primal(cb_data, cb_where)
 println("Solution is: ", callback_value(cb_data, x))
 end
 return
 end
my_callback_function (generic function with 1 method)
julia> set_attribute(model, Gurobi.CallbackFunction(), my_callback_function)
julia> optimize!(model)
Solution is: 10.0
```

## source

#### check\_belongs\_to\_model

JuMP.check\_belongs\_to\_model - Function.

check\_belongs\_to\_model(x::AbstractJuMPScalar, model::AbstractModel)
check\_belongs\_to\_model(x::AbstractConstraint, model::AbstractModel)

Throw VariableNotOwned if the owner\_model of x is not model.

```
julia> model = Model();
julia> @variable(model, x);
julia> check_belongs_to_model(x, model)
julia> model_2 = Model();
julia> check_belongs_to_model(x, model_2)
ERROR: VariableNotOwned{VariableRef}(x): the variable x cannot be used in this model because
it belongs to a different model.
[...]
```

source

#### coefficient

JuMP.coefficient - Function.

coefficient(v1::GenericVariableRef{T}, v2::GenericVariableRef{T}) where {T}

Return one(T) if v1 = v2 and zero(T) otherwise.

This is a fallback for other coefficient methods to simplify code in which the expression may be a single variable.

#### Example

julia> model = Model();

julia> @variable(model, x[1:2]);

julia> coefficient(x[1], x[1])
1.0
julia> coefficient(x[1], x[2])

```
0.0
```

#### source

coefficient(a::GenericAffExpr{C,V}, v::V) where {C,V}

Return the coefficient associated with variable v in the affine expression a.

```
julia> model = Model();
julia> @variable(model, x);
julia> expr = 2.0 * x + 1.0;
```

```
julia> coefficient(expr, x)
2.0
```

#### source

```
coefficient(a::GenericQuadExpr{C,V}, v1::V, v2::V) where {C,V}
```

Return the coefficient associated with the term v1 \* v2 in the quadratic expression a.

```
Note that coefficient(a, v1, v2) is the same as coefficient(a, v2, v1).
```

#### Example

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> expr = 2.0 * x[1] * x[2];
julia> coefficient(expr, x[1], x[2])
2.0
julia> coefficient(expr, x[2], x[1])
2.0
julia> coefficient(expr, x[1], x[1])
0.0
```

#### source

coefficient(a::GenericQuadExpr{C,V}, v::V) where {C,V}

Return the coefficient associated with variable v in the affine component of a.

#### Example

```
julia> model = Model();
julia> @variable(model, x);
julia> expr = 2.0 * x² + 3.0 * x;
julia> coefficient(expr, x)
3.0
```

#### source

## compute\_conflict!

JuMP.compute\_conflict! - Function.

compute\_conflict!(model::GenericModel)

Compute a conflict if the model is infeasible.

The conflict is also called the Irreducible Infeasible Subsystem (IIS).

If an optimizer has not been set yet (see set\_optimizer), a NoOptimizer error is thrown.

The status of the conflict can be checked with the MOI.ConflictStatus model attribute. Then, the status for each constraint can be queried with the MOI.ConstraintConflictStatus attribute.

See also: copy\_conflict

#### Example

julia> using JuMP julia> model = Model(Gurobi.Optimizer); julia> set\_silent(model) julia> @variable(model, x >= 0); julia> @constraint(model, c1, x >= 2); julia> @constraint(model, c2, x <= 1); julia> optimize!(model) julia> compute\_conflict!(model) julia> get\_attribute(model, MOI.ConflictStatus()) CONFLICT\_FOUND::ConflictStatusCode = 3

#### source

#### constant

JuMP.constant - Function.

constant(aff::GenericAffExpr{C,V})::C

Return the constant of the affine expression.

#### Example

julia> model = Model();

julia> @variable(model, x);

julia> aff = 2.0 \* x + 3.0; julia> constant(aff) 3.0

#### source

constant(quad::GenericQuadExpr{C,V})::C

Return the constant of the quadratic expression.

#### Example

```
julia> model = Model();
julia> @variable(model, x);
julia> quad = 2.0 * x^2 + 3.0;
julia> constant(quad)
3.0
```

#### source

#### constraint\_by\_name

JuMP.constraint\_by\_name - Function.

constraint\_by\_name(model::AbstractModel, name::String, [F, S])::Union{ConstraintRef,Nothing}

Return the reference of the constraint with name attribute name or Nothing if no constraint has this name attribute.

Throws an error if several constraints have name as their name attribute.

If F and S are provided, this method addititionally throws an error if the constraint is not an F-in-S contraint where F is either the JuMP or MOI type of the function and S is the MOI type of the set.

Providing F and S is recommended if you know the type of the function and set since its returned type can be inferred while for the method above (that is, without F and S), the exact return type of the constraint index cannot be inferred.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @constraint(model, con, x^2 == 1)
```
# CHAPTER 22. DOCSTRINGS

 $con : x^2 = 1$ 

julia> constraint\_by\_name(model, "kon")
julia> constraint\_by\_name(model, "con")
con : x<sup>2</sup> = 1
julia> constraint\_by\_name(model, "con", AffExpr, MOI.EqualTo{Float64})
julia> constraint\_by\_name(model, "con", QuadExpr, MOI.EqualTo{Float64})

source

#### constraint\_object

 $con : x^2 = 1$ 

JuMP.constraint\_object - Function.

```
constraint_object(con_ref::ConstraintRef)
```

Return the underlying constraint data for the constraint referenced by con\_ref.

#### Example

A scalar constraint:

julia> model = Model();

julia> @variable(model, x);

julia> @constraint(model, c, 2x <= 1)
c : 2 x ≤ 1</pre>

```
julia> object = constraint_object(c)
ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(2 x,
→ MathOptInterface.LessThan{Float64}(1.0))
```

```
julia> typeof(object)
```

```
ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}
```

```
julia> object.func
2 x
```

ΖX

```
julia> object.set
MathOptInterface.LessThan{Float64}(1.0)
```

A vector constraint:

julia> model = Model();

julia> @variable(model, x[1:3]);

```
julia> @constraint(model, c, x in SecondOrderCone())
```

c : [x[1], x[2], x[3]] ∈ MathOptInterface.SecondOrderCone(3)

#### julia> object = constraint\_object(c)

```
VectorConstraint{VariableRef, MathOptInterface.SecondOrderCone, VectorShape}(VariableRef[x[1],

→ x[2], x[3]], MathOptInterface.SecondOrderCone(3), VectorShape())
```

#### julia> typeof(object)

VectorConstraint{VariableRef, MathOptInterface.SecondOrderCone, VectorShape}

#### julia> object.func

3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

# julia> object.set

MathOptInterface.SecondOrderCone(3)

#### source

# constraint\_ref\_with\_index

JuMP.constraint\_ref\_with\_index - Function.

```
constraint_ref_with_index(model::AbstractModel, index::MOI.ConstraintIndex)
```

Return a ConstraintRef of model corresponding to index.

This function is a helper function used internally by JuMP and some JuMP extensions. It should not need to be called in user-code.

#### source

# constraint\_string

JuMP.constraint\_string - Function.

```
constraint_string(
 mode::MIME,
 ref::ConstraintRef;
 in_math_mode::Bool = false,
)
```

Return a string representation of the constraint ref, given the mode.

```
julia> model = Model();
julia> @variable(model, x);
```

```
julia> @constraint(model, c, 2 * x <= 1);
julia> constraint_string(MIME("text/plain"), c)
"c : 2 x ≤ 1"
```

#### constraints\_string

JuMP.constraints\_string - Function.

constraints\_string(mode, model::AbstractModel)::Vector{String}

Return a list of Strings describing each constraint of the model.

# Example

```
julia> model = Model();
julia> @variable(model, x >= 0);
julia> @constraint(model, c, 2 * x <= 1);
julia> constraints_string(MIME("text/plain"), model)
2-element Vector{String}:
 "c : 2 x ≤ 1"
 "x ≥ 0"
```

#### source

# copy\_conflict

JuMP.copy\_conflict - Function.

copy\_conflict(model::GenericModel)

Return a copy of the current conflict for the model model and a GenericReferenceMap that can be used to obtain the variable and constraint reference of the new model corresponding to a given model's reference.

This is a convenience function that provides a filtering function for copy\_model.

#### Note

Model copy is not supported in DIRECT mode, that is, when a model is constructed using the direct\_model constructor instead of the Model constructor. Moreover, independently on whether an optimizer was provided at model construction, the new model will have no optimizer, that is, an optimizer will have to be provided to the new model in the optimize! call.

In the following example, a model model is constructed with a variable x and two constraints c1 and c2. This model has no solution, as the two constraints are mutually exclusive. The solver is asked to compute a conflict with compute\_conflict!. The parts of model participating in the conflict are then copied into a model is\_model.

```
julia> using JuMP
julia> import Gurobi
julia> model = Model(Gurobi.Optimizer);
julia> set_silent(model)
julia> @variable(model, x >= 0)
х
julia> @constraint(model, c1, x >= 2)
c1 : x \ge 2
julia> @constraint(model, c2, x <= 1)</pre>
c2 : x \le 1
julia> optimize!(model)
julia> compute_conflict!(model)
julia> if get attribute(model, MOI.ConflictStatus()) == MOI.CONFLICT FOUND
 iis_model, reference_map = copy_conflict(model)
 print(iis_model)
 end
Feasibility
Subject to
 c1 : x \ge 2
 c2 \ : \ x \ \le \ 1
```

source

#### copy\_extension\_data

JuMP.copy\_extension\_data - Function.

copy\_extension\_data(data, new\_model::AbstractModel, model::AbstractModel)

Return a copy of the extension data data of the model model to the extension data of the new model new\_model.

A method should be added for any JuMP extension storing data in the ext field.

This method should only be implemented by developers creating JuMP extensions. It should never be called by users of JuMP.

# CHAPTER 22. DOCSTRINGS

#### Warning

Do not engage in type piracy by implementing this method for types of data that you did not define! JuMP extensions should store types that they define in model.ext, rather than regular Julia types.

source

# copy\_model

JuMP.copy\_model - Function.

copy\_model(model::GenericModel; filter\_constraints::Union{Nothing, Function}=nothing)

Return a copy of the model model and a GenericReferenceMap that can be used to obtain the variable and constraint reference of the new model corresponding to a given model's reference. A Base.copy(::AbstractModel) method has also been implemented, it is similar to copy\_model but does not return the reference map.

If the filter\_constraints argument is given, only the constraints for which this function returns true will be copied. This function is given a constraint reference as argument.

#### Note

Model copy is not supported in DIRECT mode, that is, when a model is constructed using the direct\_model constructor instead of the Model constructor. Moreover, independently on whether an optimizer was provided at model construction, the new model will have no optimizer, that is, an optimizer will have to be provided to the new model in the optimize! call.

#### Example

In the following example, a model model is constructed with a variable x and a constraint cref. It is then copied into a model new\_model with the new references assigned to x\_new and cref\_new.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @constraint(model, cref, x == 2)
cref : x = 2
julia> new_model, reference_map = copy_model(model);
julia> x_new = reference_map[x]
x
julia> cref_new = reference_map[cref]
cref : x = 2
```

source

# delete

JuMP.delete - Function.

delete(model::GenericModel, con\_ref::ConstraintRef)

Delete the constraint associated with constraint\_ref from the model model.

Note that delete does not unregister the name from the model, so adding a new constraint of the same name will throw an error. Use unregister to unregister the name after deletion.

#### Example

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, c, 2x <= 1)
c : 2 x ≤ 1
julia> delete(model, c)
julia> unregister(model, :c)
julia> print(model)
Feasibility
Subject to
julia> model[:c]
ERROR: KeyError: key :c not found
```

Stacktrace: [...]

#### source

delete(model::GenericModel, con\_refs::Vector{<:ConstraintRef})</pre>

Delete the constraints associated with con\_refs from the model model.

Solvers may implement specialized methods for deleting multiple constraints of the same concrete type. These methods may be more efficient than repeatedly calling the single constraint delete method.

See also: unregister

#### Example

julia> model = Model();

julia> @variable(model, x[1:3]);

#### julia> @constraint(model, c, 2 \* x .<= 1)</pre>

3-element Vector{ConstraintRef{Model,

- $\hookrightarrow \mbox{MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64}, and a second sec$
- → MathOptInterface.LessThan{Float64}}, ScalarShape}}:

```
c : 2 x[1] ≤ 1
c : 2 x[2] ≤ 1
c : 2 x[3] ≤ 1
julia> delete(model, c)
julia> unregister(model, :c)
julia> print(model)
Feasibility
Subject to
julia> model[:c]
ERROR: KeyError: key :c not found
Stacktrace:
[...]
```

delete(model::GenericModel, variable\_ref::GenericVariableRef)

Delete the variable associated with variable\_ref from the model model.

Note that delete does not unregister the name from the model, so adding a new variable of the same name will throw an error. Use unregister to unregister the name after deletion.

#### Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> delete(model, x)
julia> unregister(model, :x)
julia> print(model)
Feasibility
Subject to
julia> model[:x]
ERROR: KeyError: key :x not found
Stacktrace:
[...]
```

#### source

```
delete(model::GenericModel, variable_refs::Vector{<:GenericVariableRef})</pre>
```

Delete the variables associated with variable\_refs from the model model. Solvers may implement methods for deleting multiple variables that are more efficient than repeatedly calling the single variable delete method. See also: unregister

# Example

julia> model = Model();

julia> @variable(model, x[1:2]);

julia> delete(model, x)

julia> unregister(model, :x)

**julia> print(model)** Feasibility Subject to

# julia> model[:x] ERROR: KeyError: key :x not found Stacktrace: [...]

#### source

#### delete\_lower\_bound

JuMP.delete\_lower\_bound - Function.

```
delete_lower_bound(v::GenericVariableRef)
```

Delete the lower bound constraint of a variable.

See also LowerBoundRef, has\_lower\_bound, lower\_bound, set\_lower\_bound.

#### Example

```
julia> model = Model();
julia> @variable(model, x >= 1.0);
julia> has_lower_bound(x)
true
julia> delete_lower_bound(x)
julia> has_lower_bound(x)
false
```

#### source

# delete\_upper\_bound

JuMP.delete\_upper\_bound - Function.

delete\_upper\_bound(v::GenericVariableRef)

Delete the upper bound constraint of a variable.

Errors if one does not exist.

See also UpperBoundRef, has\_upper\_bound, upper\_bound, set\_upper\_bound.

# Example

```
julia> model = Model();
```

```
julia> @variable(model, x <= 1.0);</pre>
```

julia> has\_upper\_bound(x)
true

julia> delete\_upper\_bound(x)

julia> has\_upper\_bound(x)
false

#### source

#### direct\_generic\_model

JuMP.direct\_generic\_model - Function.

```
direct_generic_model(
 value_type::Type{T},
 backend::MOI.ModelLike;
) where {T<:Real}</pre>
```

Return a new JuMP model using backend to store the model and solve it.

As opposed to the Model constructor, no cache of the model is stored outside of backend and no bridges are automatically applied to backend.

# Notes

The absence of a cache reduces the memory footprint but, it is important to bear in mind the following implications of creating models using this *direct* mode:

- When backend does not support an operation, such as modifying constraints or adding variables/constraints after solving, an error is thrown. For models created using the Model constructor, such situations can be dealt with by storing the modifications in a cache and loading them into the optimizer when optimize! is called.
- No constraint bridging is supported by default.
- The optimizer used cannot be changed the model is constructed.
- The model created cannot be copied.

```
direct_generic_model(::Type{T}, factory::MOI.OptimizerWithAttributes)
```

Create a direct\_generic\_model using factory, a MOI.OptimizerWithAttributes object created by optimizer\_with\_attri

#### Example

#### is equivalent to:

# julia> import HiGHS

```
julia> model = direct_generic_model(Float64, HiGHS.Optimizer())
A JuMP Model
| mode: DIRECT
| solver: HiGHS
| objective_sense: FEASIBILITY_SENSE
| num_variables: 0
| num_constraints: 0
L Names registered in the model: none
```

```
julia> set_attribute(model, "presolve", "off")
```

```
julia> set_attribute(model, MOI.Silent(), true)
```

#### source

# direct\_model

```
JuMP.direct_model - Function.
```

```
direct model(backend::MOI.ModelLike)
```

Return a new JuMP model using backend to store the model and solve it.

As opposed to the Model constructor, no cache of the model is stored outside of backend and no bridges are automatically applied to backend.

#### Notes

The absence of a cache reduces the memory footprint but, it is important to bear in mind the following implications of creating models using this *direct* mode:

- When backend does not support an operation, such as modifying constraints or adding variables/constraints after solving, an error is thrown. For models created using the Model constructor, such situations can be dealt with by storing the modifications in a cache and loading them into the optimizer when optimize! is called.
- No constraint bridging is supported by default.
- The optimizer used cannot be changed the model is constructed.
- The model created cannot be copied.

#### source

direct\_model(factory::MOI.OptimizerWithAttributes)

Create a direct\_model using factory, a MOI.OptimizerWithAttributes object created by optimizer\_with\_attributes.

#### Example

```
julia> import HiGHS
julia> optimizer = optimizer_with_attributes(
```

```
HiGHS.Optimizer,
 "presolve" => "off",
 MOI.Silent() => true,
);
```

```
julia> model = direct_model(optimizer)
```

```
A JuMP Model

+ mode: DIRECT

+ solver: HiGHS

+ objective_sense: FEASIBILITY_SENSE

+ num_variables: 0

+ num_constraints: 0

L Names registered in the model: none
```

#### is equivalent to:

#### julia> import HiGHS

```
julia> model = direct_model(HiGHS.Optimizer())
A JuMP Model
 mode: DIRECT
 solver: HiGHS
 objective_sense: FEASIBILITY_SENSE
 num_variables: 0
 num_constraints: 0
```

 $^{\rm L}$  Names registered in the model: none

julia> set\_attribute(model, "presolve", "off")

```
julia> set_attribute(model, MOI.Silent(), true)
```

#### source

# drop\_zeros!

JuMP.drop\_zeros! - Function.

drop\_zeros!(expr::GenericAffExpr)

Remove terms in the affine expression with 0 coefficients.

# Example

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2]);
```

**julia**> expr = x[1] + x[2];

julia> add\_to\_expression!(expr, -1.0, x[1])
0 x[1] + x[2]

julia> drop\_zeros!(expr)

julia> expr
x[2]

#### source

drop\_zeros!(expr::GenericQuadExpr)

Remove terms in the quadratic expression with 0 coefficients.

#### Example

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> expr = x[1]^2 + x[2]^2;
julia> add_to_expression!(expr, -1.0, x[1], x[1])
0 x[1]² + x[2]²
julia> drop_zeros!(expr)
```

760

julia> expr
x[2]<sup>2</sup>

source

# dual

```
JuMP.dual - Function.
```

```
dual(con_ref::ConstraintRef; result::Int = 1)
```

Return the dual value of constraint con\_ref associated with result index result of the most-recent solution returned by the solver.

Use dual\_status to check if a result exists before asking for values.

```
See also: result_count, shadow_price.
```

#### Example

julia> import HiGHS

julia> model = Model(HiGHS.Optimizer);

julia> set\_silent(model)

julia> @variable(model, x);

julia> @constraint(model, c, x <= 1)
c : x ≤ 1</pre>

julia> @objective(model, Max, 2 \* x + 1);

julia> optimize!(model)

```
julia> dual_status(model)
FEASIBLE_POINT::ResultStatusCode = 1
```

```
julia> dual(c)
-2.0
```

#### source

# dual\_objective\_value

JuMP.dual\_objective\_value - Function.

```
dual_objective_value(model::GenericModel; result::Int = 1)
```

Return the value of the objective of the dual problem associated with result index result of the most-recent solution returned by the solver.

Throws MOI.UnsupportedAttribute{MOI.DualObjectiveValue} if the solver does not support this attribute.

This function is equivalent to querying the MOI.DualObjectiveValue attribute.

See also: result\_count.

#### Example

julia> import HiGHS

julia> model = Model(HiGHS.Optimizer);

julia> set\_silent(model)

```
julia> @variable(model, x >= 1);
```

```
julia> @objective(model, Min, 2 * x + 1);
```

julia> optimize!(model)

julia> dual\_objective\_value(model)
3.0

#### source

# dual\_shape

JuMP.dual\_shape - Function.

dual\_shape(shape::AbstractShape)::AbstractShape

Returns the shape of the dual space of the space of objects of shape shape. By default, the dual\_shape of a shape is itself. See the examples section below for an example for which this is not the case.

#### Example

Consider polynomial constraints for which the dual is moment constraints and moment constraints for which the dual is polynomial constraints. Shapes for polynomials can be defined as follows:

```
struct Polynomial
 coefficients::Vector{Float64}
 monomials::Vector{Monomial}
end
struct PolynomialShape <: AbstractShape
 monomials::Vector{Monomial}</pre>
```

# end

```
JuMP.reshape_vector(x::Vector, shape::PolynomialShape) = Polynomial(x, shape.monomials)
```

and a shape for moments can be defined as follows:

```
struct Moments
 coefficients::Vector{Float64}
 monomials::Vector{Monomial}
end
struct MomentsShape <: AbstractShape
 monomials::Vector{Monomial}
end
JuMP.reshape_vector(x::Vector, shape::MomentsShape) = Moments(x, shape.monomials)</pre>
```

Then dual\_shape allows the definition of the shape of the dual of polynomial and moment constraints:

```
dual_shape(shape::PolynomialShape) = MomentsShape(shape.monomials)
dual_shape(shape::MomentsShape) = PolynomialShape(shape.monomials)
```

#### source

# dual\_start\_value

JuMP.dual\_start\_value - Function.

```
dual_start_value(con_ref::ConstraintRef)
```

Return the dual start value (MOI attribute ConstraintDualStart) of the constraint con\_ref.

If no dual start value has been set, dual\_start\_value will return nothing.

```
See also set_dual_start_value.
```

julia> dual\_start\_value(c)

```
julia> model = Model();
julia> @variable(model, x, start = 2.0);
julia> @constraint(model, c, [2x] in Nonnegatives())
c : [2 x] ∈ Nonnegatives()
julia> set_dual_start_value(c, [0.0])
julia> dual_start_value(c)
1-element Vector{Float64}:
0.0
julia> set_dual_start_value(c, nothing)
```

#### dual\_status

JuMP.dual\_status - Function.

```
dual_status(model::GenericModel; result::Int = 1)
```

Return a MOI.ResultStatusCode describing the status of the most recent dual solution of the solver (that is, the MOI.DualStatus attribute) associated with the result index result.

See also: result\_count.

#### Example

julia> import Ipopt

```
julia> model = Model(Ipopt.Optimizer);
```

```
julia> dual_status(model; result = 2)
N0_SOLUTION::ResultStatusCode = 0
```

#### source

# error\_if\_direct\_mode

JuMP.error\_if\_direct\_mode - Function.

error\_if\_direct\_mode(model::GenericModel, func::Symbol)

Errors if model is in direct mode during a call from the function named func.

Used internally within JuMP, or by JuMP extensions who do not want to support models in direct mode.

# Example

```
julia> import HiGHS
```

```
julia> model = direct_model(HiGHS.Optimizer());
```

```
julia> error_if_direct_mode(model, :foo)
ERROR: The `foo` function is not supported in DIRECT mode.
Stacktrace:
[...]
```

source

# fix

JuMP.fix - Function.

```
fix(v::GenericVariableRef, value::Number; force::Bool = false)
```

Fix a variable to a value. Update the fixing constraint if one exists, otherwise create a new one.

If the variable already has variable bounds and force=false, calling fix will throw an error. If force=true, existing variable bounds will be deleted, and the fixing constraint will be added. Note a variable will have no bounds after a call to unfix.

See also FixRef, is\_fixed, fix\_value, unfix.

#### Example

```
julia> model = Model();
julia> @variable(model, x);
julia> is_fixed(x)
false
julia> fix(x, 1.0)
julia> is_fixed(x)
true
```

```
julia> model = Model();
julia> @variable(model, 0 <= x <= 1);
julia> is_fixed(x)
false
julia> fix(x, 1.0; force = true)
julia> is_fixed(x)
true
```

#### source

#### fix\_discrete\_variables

JuMP.fix\_discrete\_variables - Function.

fix\_discrete\_variables([var\_value::Function = value,] model::GenericModel)

Modifies model to convert all binary and integer variables to continuous variables with fixed bounds of  $var_value(x)$ .

#### Return

Returns a function that can be called without any arguments to restore the original model. The behavior of this function is undefined if additional changes are made to the affected variables in the meantime.

#### Notes

- An error is thrown if semi-continuous or semi-integer constraints are present (support may be added for these in the future).
- All other constraints are ignored (left in place). This includes discrete constraints like SOS and indicator constraints.

#### Example

```
julia> model = Model();
```

julia> @variable(model, x, Bin, start = 1);

julia> @variable(model, 1 <= y <= 10, Int, start = 2);</pre>

julia> @objective(model, Min, x + y);

julia> undo\_relax = fix\_discrete\_variables(start\_value, model);

#### julia> print(model)

Min x + ySubject to x = 1y = 2

#### julia> undo\_relax()

julia> print(model)
Min x + y
Subject to
 y ≥ 1
 y ≤ 10
 y integer
 x binary

# source

#### fix\_value

JuMP.fix\_value - Function.

fix\_value(v::GenericVariableRef)

Return the value to which a variable is fixed.

Error if one does not exist.

See also FixRef, is\_fixed, fix, unfix.

# Example

julia> model = Model();

julia> @variable(model, x == 1);

julia> fix\_value(x)
1.0

source

#### flatten!

JuMP.flatten! - Function.

flatten!(expr::GenericNonlinearExpr)

Flatten a nonlinear expression in-place by lifting nested + and \* nodes into a single n-ary operation.

# Motivation

Nonlinear expressions created using operator overloading can be deeply nested and unbalanced. For example, prod(x for i in 1:4) creates \*(x, \*(x, \*(x, x))) instead of the more preferable \*(x, x, x, x).

#### Example

julia> model = Model(); julia> @variable(model, x) x julia> y = prod(x for i in 1:4) ((x<sup>2</sup>) \* x) \* x julia> flatten!(y)

(x²) \* x \* x

julia> flatten!(sin(prod(x for i in 1:4)))
sin((x<sup>2</sup>) \* x \* x)

#### source

# function\_string

JuMP.function\_string - Function.

```
function_string(
 mode::MIME,
 func::Union{JuMP.AbstractJuMPScalar,Vector{<:JuMP.AbstractJuMPScalar}},
)</pre>
```

Return a String representing the function func using print mode mode.

```
julia> model = Model();
julia> @variable(model, x);
julia> function_string(MIME("text/plain"), 2 * x + 1)
"2 x + 1"
```

#### get\_attribute

JuMP.get\_attribute - Function.

```
get_attribute(model::GenericModel, attr::MOI.AbstractModelAttribute)
get_attribute(x::GenericVariableRef, attr::MOI.AbstractVariableAttribute)
get_attribute(cr::ConstraintRef, attr::MOI.AbstractConstraintAttribute)
```

Get the value of a solver-specifc attribute attr.

This is equivalent to calling MOI.get with the associated MOI model and, for variables and constraints, with the associated MOI.VariableIndex or MOI.ConstraintIndex.

#### Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @constraint(model, c, 2 * x <= 1)
c : 2 x ≤ 1
julia> get_attribute(model, MOI.Name())
""
julia> get_attribute(x, MOI.VariableName())
"x"
julia> get_attribute(c, MOI.ConstraintName())
"c"
```

#### source

```
get_attribute(
 model::Union{GenericModel,MOI.OptimizerWithAttributes},
 attr::Union{AbstractString,MOI.AbstractOptimizerAttribute},
)
```

Get the value of a solver-specifc attribute attr.

This is equivalent to calling MOI.get with the associated MOI model.

If attr is an AbstractString, it is converted to MOI.RawOptimizerAttribute.

#### Example

```
julia> import HiGHS
julia> opt = optimizer_with_attributes(HiGHS.Optimizer, "output_flag" => true);
julia> model = Model(opt);
julia> get_attribute(model, "output_flag")
true
julia> get_attribute(model, MOI.RawOptimizerAttribute("output_flag"))
true
julia> get_attribute(opt, "output_flag")
true
```

# source

#### has duals

JuMP.has duals - Function.

has\_duals(model::GenericModel; result::Int = 1)

Return true if the solver has a dual solution in result index result available to query, otherwise return false.

See also dual, shadow\_price, and result\_count.

```
julia> import HiGHS
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x);
julia> @constraint(model, c, x <= 1)
c : x ≤ 1
julia> @objective(model, Max, 2 * x + 1);
julia> has_duals(model)
false
julia> optimize!(model)
```

julia> has\_duals(model)
true

#### source

# has\_lower\_bound

JuMP.has\_lower\_bound - Function.

has\_lower\_bound(v::GenericVariableRef)

Return true if v has a lower bound. If true, the lower bound can be queried with lower\_bound.

See also LowerBoundRef, lower\_bound, set\_lower\_bound, delete\_lower\_bound.

# Example

```
julia> model = Model();
```

```
julia> @variable(model, x >= 1.0);
```

julia> has\_lower\_bound(x)
true

#### source

# has\_start\_value

JuMP.has\_start\_value - Function.

has\_start\_value(variable::AbstractVariableRef)

Return true if the variable has a start value set, otherwise return false.

See also: start\_value, set\_start\_value.

```
julia> model = Model();
julia> @variable(model, x, start = 1.5);
julia> @variable(model, y);
julia> has_start_value(x)
true
julia> has_start_value(y)
false
```

```
julia> start_value(x)
1.5
julia> set_start_value(y, 2.0)
julia> has_start_value(y)
true
julia> start_value(y)
2.0
```

# has\_upper\_bound

JuMP.has\_upper\_bound - Function.

has\_upper\_bound(v::GenericVariableRef)

Return true if v has a upper bound. If true, the upper bound can be queried with upper\_bound.

See also UpperBoundRef, upper\_bound, set\_upper\_bound, delete\_upper\_bound.

# Example

julia> model = Model(); julia> @variable(model, x <= 1.0); julia> has\_upper\_bound(x) true

# source

# has\_values

JuMP.has\_values - Function.

has\_values(model::GenericModel; result::Int = 1)

Return true if the solver has a primal solution in result index result available to query, otherwise return false.

See also value and result\_count.

julia> import HiGHS

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x);
julia> @constraint(model, c, x <= 1)
c : x ≤ 1
julia> @objective(model, Max, 2 * x + 1);
julia> has_values(model)
false
julia> optimize!(model)
julia> has_values(model)
true
```

# source

# in\_set\_string

JuMP.in\_set\_string - Function.

in\_set\_string(mode::MIME, set)

Return a String representing the membership to the set set using print mode mode.

# Extensions

JuMP extensions may extend this method for new set types to improve the legibility of their printing.

#### Example

```
julia> in_set_string(MIME("text/plain"), MOI.Interval(1.0, 2.0))
"6 [1, 2]"
```

# source

#### index

JuMP.index - Function.

index(cr::ConstraintRef)::MOI.ConstraintIndex

Return the index of the constraint that corresponds to cr in the MOI backend.

julia> model = Model();

julia> @variable(model, x);

julia> @constraint(model, c, x >= 0);

#### julia> index(c)

# source

index(v::GenericVariableRef)::MOI.VariableIndex

Return the index of the variable that corresponds to  $\boldsymbol{v}$  in the MOI backend.

#### Example

```
julia> model = Model();
```

julia> @variable(model, x);

```
julia> index(x)
MOI.VariableIndex(1)
```

# source

# is\_binary

JuMP.is\_binary - Function.

is\_binary(v::GenericVariableRef)

Return true if v is constrained to be binary.

See also BinaryRef, set\_binary, unset\_binary.

# Example

```
julia> model = Model();
```

julia> @variable(model, x, Bin);

julia> is\_binary(x)
true

#### source

# is\_fixed

JuMP.is\_fixed - Function.

is\_fixed(v::GenericVariableRef)

Return true if v is a fixed variable. If true, the fixed value can be queried with fix\_value.

See also FixRef, fix\_value, fix, unfix.

# Example

```
julia> model = Model();
julia> @variable(model, x);
julia> is_fixed(x)
false
julia> fix(x, 1.0)
julia> is_fixed(x)
true
```

## source

# is\_integer

JuMP.is\_integer - Function.

is\_integer(v::GenericVariableRef)

Return true if v is constrained to be integer.

See also IntegerRef, set\_integer, unset\_integer.

# Example

julia> model = Model();

julia> @variable(model, x);

julia> is\_integer(x)
false

julia> set\_integer(x)

julia> is\_integer(x)
true

source

#### is\_parameter

JuMP.is\_parameter - Function.

is\_parameter(x::GenericVariableRef)::Bool

Return true if x is constrained to be a parameter.

See also ParameterRef, set\_parameter\_value, parameter\_value.

# Example

```
julia> model = Model();
julia> @variable(model, p in Parameter(2))
p
julia> is_parameter(p)
true
julia> @variable(model, x)
x
julia> is_parameter(x)
false
```

#### source

# is\_solved\_and\_feasible

JuMP.is\_solved\_and\_feasible - Function.

```
is_solved_and_feasible(
 model::GenericModel;
 allow_local::Bool = true,
 allow_almost::Bool = false,
 dual::Bool = false,
 result::Int = 1,
)
```

Return true if:

- the termination\_status is one of:
  - OPTIMAL (the solver found a global optimum)
  - LOCALLY\_SOLVED (the solver found a local optimum, which may also be the global optimum, but the solver could not prove so).
- the primal\_status of the result index result is FEASIBLE\_POINT.

This function is conservative, in that it returns false for situations like the solver terminating with a feasible solution due to a time limit.

If this function returns false, use termination\_status, result\_count, primal\_status and dual\_status to understand what solutions are available (if any).

See also: assert\_is\_solved\_and\_feasible.

#### **Keyword arguments**

# allow\_local

If allow\_local = false, then this function returns true only if the termination\_status is OPTIMAL.

# allow\_almost

If allow\_almost = true, then the termination\_status may additionally be ALMOST\_OPTIMAL or ALMOST\_LOCALLY\_SOLVED (if allow\_local), and the primal\_status and dual\_status may additionally be NEARLY\_FEASIBLE\_POINT.

# dual

If dual, additionally check that an optimal dual solution is available via dual\_status. The allow\_keywords control both the primal and dual solutions.

# result

The index of the result to query. This value is passed to the result keyword arguments of primal\_status and dual\_status.

#### Example

julia> import Ipopt

```
julia> model = Model(Ipopt.Optimizer);
julia> is_solved_and_feasible(model)
false
julia> is_solved_and_feasible(
 model;
 allow_almost = true,
 dual = true,
 result = 2,
)
false
```

source

#### is\_valid

JuMP.is\_valid - Function.

is\_valid(model::GenericModel, con\_ref::ConstraintRef{<:AbstractModel})</pre>

Return true if con\_ref refers to a valid constraint in model.

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, c, 2 * x <= 1);
julia> is_valid(model, c)
true
julia> model_2 = Model();
julia> is_valid(model_2, c)
false
```

is\_valid(model::GenericModel, variable\_ref::GenericVariableRef)

Return true if variable refers to a valid variable in model.

# Example

```
julia> model = Model();
julia> @variable(model, x);
julia> is_valid(model, x)
true
julia> model_2 = Model();
julia> is_valid(model_2, x)
false
```

#### source

# is\_variable\_in\_set

JuMP.is\_variable\_in\_set - Function.

```
is_variable_in_set(
 model::GenericModel,
 x::Union{AbstractJuMPScalar,AbstractArray{<:AbstractJuMPScalar}},
)::Bool</pre>
```

Return a Bool if VariableInSetRef returns a valid constraint reference without erroring.

# Exceptions

This function does not apply for variable bounds or integrality restrictions of a scalar variable. For example:

```
julia> model = Model();
julia> @variable(model, x >= 0, Int)
x
julia> is_variable_in_set(x)
false
```

Use instead is\_integer, is\_binary, has\_lower\_bound, has\_upper\_bound, and is\_fixed.

```
julia> model = Model();
julia> @variable(model, x >= 0, Int)
x
julia> is_integer(x)
true
julia> has_lower_bound(x)
true
```

# Example

julia> model = Model();

```
julia> @variable(model, x[1:2, 1:2], PSD)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
x[1,1] x[1,2]
x[1,2] x[2,2]
julia> is_variable_in_set(x)
true
julia> c = VariableInSetRef(x)
```

[x[1,1] x[1,2] ... x[2,2]] ∈ PSDCone()

julia> @variable(model, y)
v

julia> is\_variable\_in\_set(y)
false

julia> @variable(model, z in Semicontinuous(1, 2))
z

julia> is\_variable\_in\_set(z)
true

julia> c\_z = VariableInSetRef(z)
z ∈ MathOptInterface.Semicontinuous{Int64}(1, 2)

# isequal\_canonical

JuMP.isequal\_canonical - Function.

```
isequal_canonical(
 x::T,
 y::T
) where {T<:AbstractJuMPScalar,AbstractArray{<:AbstractJuMPScalar}}</pre>
```

Return true if x is equal to y after dropping zeros and disregarding the order.

This method is mainly useful for testing, because fallbacks like x == y do not account for valid mathematical comparisons like x[1] + 0 x[2] + 1 == x[1] + 1.

# Example

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> a = x[1] + 1.0
x[1] + 1
julia> b = x[1] + x[2] + 1.0
x[1] + x[2] + 1
julia> add_to_expression!(b, -1.0, x[2])
x[1] + 0 x[2] + 1
julia> a == b
false
julia> isequal_canonical(a, b)
true
```

#### source

#### jump\_function

JuMP.jump\_function - Function.

jump\_function(model::AbstractModel, x::MOI.AbstractFunction)

Given an MathOptInterface object x, return the JuMP equivalent.

See also: moi\_function.

# Example

julia> model = Model();

julia> @variable(model, x);

```
julia> f = 2.0 * index(x) + 1.0
1.0 + 2.0 MOI.VariableIndex(1)
julia> jump_function(model, f)
2 x + 1
```

#### jump\_function\_type

JuMP.jump\_function\_type - Function.

jump\_function\_type(model::AbstractModel, ::Type{T}) where {T}

Given an MathOptInterface object type T, return the JuMP equivalent.

See also: moi\_function\_type.

# Example

julia> model = Model();

```
julia> jump_function_type(model, MOI.ScalarAffineFunction{Float64})
AffExpr (alias for GenericAffExpr{Float64, GenericVariableRef{Float64}})
```

#### source

#### latex\_formulation

JuMP.latex\_formulation - Function.

latex\_formulation(model::AbstractModel)

Wrap model in a type so that it can be pretty-printed as text/latex in a notebook like IJulia, or in Documenter.

To render the model, end the cell with latex\_formulation(model), or call display(latex\_formulation(model)) in to force the display of the model from inside a function.

source

#### linear\_terms

JuMP.linear\_terms - Function.

```
linear_terms(aff::GenericAffExpr{C,V})
```

Provides an iterator over coefficient-variable tuples (a\_i::C,  $x_i::V$ ) in the linear part of the affine expression.

source

linear\_terms(quad::GenericQuadExpr{C,V})

Provides an iterator over tuples (coefficient::C, variable::V) in the linear part of the quadratic expression.

source

# list\_of\_constraint\_types

JuMP.list\_of\_constraint\_types - Function.

list\_of\_constraint\_types(model::GenericModel)::Vector{Tuple{Type,Type}}

Return a list of tuples of the form (F, S) where F is a JuMP function type and S is an MOI set type such that  $all_constraints(model, F, S)$  returns a nonempty list.

# Example

```
julia> model = Model();
julia> @variable(model, x >= 0, Bin);
julia> @constraint(model, 2x <= 1);
julia> list_of_constraint_types(model)
3-element Vector{Tuple{Type, Type}}:
(AffExpr, MathOptInterface.LessThan{Float64})
(VariableRef, MathOptInterface.GreaterThan{Float64})
(VariableRef, MathOptInterface.ZeroOne)
```

#### Performance considerations

Iterating over the list of function and set types is a type-unstable operation. Consider using a function barrier. See the Performance tips for extensions section of the documentation for more details.

source

#### lower\_bound

JuMP.lower\_bound - Function.

lower\_bound(v::GenericVariableRef)

Return the lower bound of a variable. Error if one does not exist.

See also LowerBoundRef, has\_lower\_bound, set\_lower\_bound, delete\_lower\_bound.

```
julia> model = Model();
julia> @variable(model, x >= 1.0);
julia> lower_bound(x)
1.0
```

# lp\_matrix\_data

JuMP.lp\_matrix\_data - Function.

```
lp_matrix_data(model::GenericModel{T})
```

Given a JuMP model of a linear program, return an LPMatrixData{T} struct storing data for an equivalent linear program in the form:

$$\min c^{\top} x + c_0$$
$$b_l \le Ax \le b_u$$
$$x_l \le x \le x_u$$

where elements in x may be continuous, integer, or binary variables.

#### Fields

The struct returned by lp\_matrix\_data has the fields:

- A::SparseArrays.SparseMatrixCSC{T,Int}: the constraint matrix in sparse matrix form.
- b\_lower::Vector{T}: the dense vector of row lower bounds. If missing, the value of typemin(T) is used.
- b\_upper::Vector{T}: the dense vector of row upper bounds. If missing, the value of typemax(T) is used.
- x\_lower::Vector{T}: the dense vector of variable lower bounds. If missing, the value of typemin(T) is used.
- x\_upper::Vector{T}: the dense vector of variable upper bounds. If missing, the value of typemax(T) is used.
- c::Vector{T}: the dense vector of linear objective coefficients
- c\_offset::T: the constant term in the objective function.
- sense::MOI.OptimizationSense: the objective sense of the model.
- integers::Vector{Int}: the sorted list of column indices that are integer variables.
- binaries::Vector{Int}: the sorted list of column indices that are binary variables.
- variables::Vector{GenericVariableRef{T}}: a vector of GenericVariableRef, corresponding to
  order of the columns in the matrix form.
- affine\_constraints::Vector{ConstraintRef}: a vector of ConstraintRef, corresponding to the order of rows in the matrix form.

# Limitations

The models supported by lp\_matrix\_data are intentionally limited to linear programs.

#### Example

julia> model = Model();

julia> @variable(model, x[1:2] >= 0);

julia> @constraint(model, x[1] + 2 \* x[2] <= 1);</pre>

julia> @objective(model, Max, x[2]);

julia> data = lp\_matrix\_data(model);

#### julia> data.A

 $1{\times}2$  SparseArrays.SparseMatrixCSC{Float64, Int64} with 2 stored entries: 1.0 2.0

#### julia> data.b\_lower

1-element Vector{Float64}:
 -Inf

julia> data.b\_upper
1-element Vector{Float64}:
1.0

# julia> data.x\_lower 2-element Vector{Float64}: 0.0 0.0

#### julia> data.x\_upper

2-element Vector{Float64}:
Inf
Inf

# **julia>** data.c

2-element Vector{Float64}: 0.0 1.0

# julia> data.c\_offset 0.0

julia> data.sense
MAX SENSE::OptimizationSense = 1

# source

# lp\_sensitivity\_report

JuMP.lp\_sensitivity\_report - Function.

Given a linear program model with a current optimal basis, return a SensitivityReport object, which maps:

- Every variable reference to a tuple (d\_lo, d\_hi)::Tuple{T,T}, explaining how much the objective coefficient of the corresponding variable can change by, such that the original basis remains optimal.
- Every constraint reference to a tuple (d\_lo, d\_hi)::Tuple{T,T}, explaining how much the righthand side of the corresponding constraint can change by, such that the basis remains optimal.

Both tuples are relative, rather than absolute. So given a objective coefficient of 1.0 and a tuple (-0.5, 0.5), the objective coefficient can range between 1.0 - 0.5 an 1.0 + 0.5.

atol is the primal/dual optimality tolerance, and should match the tolerance of the solver used to compute the basis.

Note: interval constraints are NOT supported.

```
julia> import HiGHS
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, -1 <= x <= 2)</pre>
Х
julia> @objective(model, Min, x)
julia> optimize!(model)
julia> report = lp_sensitivity_report(model; atol = 1e-7);
julia> dx_lo, dx_hi = report[x]
(-1.0, Inf)
julia> println(
 "The objective coefficient of `x` can decrease by dx_0 or " *
 "increase by $dx_hi."
)
The objective coefficient of `x` can decrease by -1.0 or increase by Inf.
julia> dRHS_lo, dRHS_hi = report[LowerBoundRef(x)]
(-Inf, 3.0)
julia> println(
 "The lower bound of `x` can decrease by $dRHS_lo or increase " *
 "by $dRHS_hi."
)
The lower bound of `x` can decrease by -Inf or increase by 3.0.
```
source

# map\_coefficients

JuMP.map\_coefficients - Function.

map\_coefficients(f::Function, a::GenericAffExpr)

Apply f to the coefficients and constant term of an GenericAffExpr a and return a new expression.

See also: map\_coefficients\_inplace!

Example

```
julia> model = Model();
julia> @variable(model, x);
julia> a = GenericAffExpr(1.0, x => 1.0)
x + 1
julia> map_coefficients(c -> 2 * c, a)
2 x + 2
julia> a
```

x + 1

#### source

map\_coefficients(f::Function, a::GenericQuadExpr)

Apply f to the coefficients and constant term of an GenericQuadExpr a and return a new expression.

```
See also: map_coefficients_inplace!
```

#### Example

```
julia> model = Model();
```

julia> @variable(model, x);

julia> a = @expression(model, x^2 + x + 1) x<sup>2</sup> + x + 1

julia> map\_coefficients(c -> 2 \* c, a)
2 x<sup>2</sup> + 2 x + 2

**julia>** a x<sup>2</sup> + x + 1

# map\_coefficients\_inplace!

JuMP.map\_coefficients\_inplace! - Function.

map\_coefficients\_inplace!(f::Function, a::GenericAffExpr)

Apply f to the coefficients and constant term of an GenericAffExpr a and update them in-place.

```
See also: map_coefficients
```

# Example

```
julia> model = Model();
julia> @variable(model, x);
julia> a = GenericAffExpr(1.0, x => 1.0)
x + 1
julia> map_coefficients_inplace!(c -> 2 * c, a)
2 x + 2
julia> a
```

#### source

2 x + 2

```
map_coefficients_inplace!(f::Function, a::GenericQuadExpr)
```

Apply f to the coefficients and constant term of an GenericQuadExpr a and update them in-place.

```
See also: map_coefficients
```

# Example

```
julia> model = Model();
julia> @variable(model, x);
julia> a = @expression(model, x^2 + x + 1)
x² + x + 1
julia> map_coefficients_inplace!(c -> 2 * c, a)
2 x² + 2 x + 2
julia> a
2 x² + 2 x + 2
```

## mode

JuMP.mode - Function.

mode(model::GenericModel)

Return the ModelMode of model.

## Example

```
julia> model = Model();
```

julia> mode(model)
AUTOMATIC::ModelMode = 0

#### source

# model\_convert

JuMP.model\_convert - Function.

```
model_convert(
 model::AbstractModel,
 rhs::Union{
 AbstractConstraint,
 Number,
 AbstractJuMPScalar,
 MOI.AbstractSet,
 },
)
```

Convert the coefficients and constants of functions and sets in the rhs to the coefficient type value\_type(typeof(model)).

#### Purpose

Creating and adding a constraint is a two-step process. The first step calls build\_constraint, and the result of that is passed to add\_constraint.

However, because build\_constraint does not take the model as an argument, the coefficients and constants of the function or set might be different than value\_type(typeof(model)).

Therefore, the result of build\_constraint is converted in a call to model\_convert before the result is passed to add\_constraint.

source

# model\_string

JuMP.model\_string - Function.

model\_string(mode::MIME, model::AbstractModel)

Return a String representation of model given the mode.

## Example

source

# moi\_function

JuMP.moi\_function - Function.

```
moi_function(x::AbstractJuMPScalar)
moi_function(x::AbstractArray{<:AbstractJuMPScalar})</pre>
```

Given a JuMP object x, return the MathOptInterface equivalent.

See also: jump\_function.

# Example

```
julia> model = Model();
```

julia> @variable(model, x);

**julia> f** = 2.0 \* x + 1.0 2 x + 1

julia> moi\_function(f)
1.0 + 2.0 MOI.VariableIndex(1)

source

# moi\_function\_type

JuMP.moi\_function\_type - Function.

moi\_function\_type(::Type{T}) where {T}

Given a JuMP object type T, return the MathOptInterface equivalent.

See also: jump\_function\_type.

julia> moi\_function\_type(AffExpr)
MathOptInterface.ScalarAffineFunction{Float64}

source

# moi\_set

JuMP.moi set - Function.

moi\_set(constraint::AbstractConstraint)

Return the set of the constraint constraint in the function-in-set form as a MathOptInterface.AbstractSet.

```
moi_set(s::AbstractVectorSet, dim::Int)
```

Returns the MOI set of dimension dim corresponding to the JuMP set s.

```
moi_set(s::AbstractScalarSet)
```

Returns the MOI set corresponding to the JuMP set s.

source

## name

JuMP.name - Function.

name(con\_ref::ConstraintRef)

Get a constraint's name attribute.

# Example

```
julia> model = Model();
```

julia> @variable(model, x);

julia> @constraint(model, c, [2x] in Nonnegatives())
c : [2 x] ∈ Nonnegatives()

julia> name(c)
"c"

name(v::GenericVariableRef)::String

Get a variable's name attribute.

#### Example

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
julia> name(x[1])
"x[1]"
```

#### source

name(model::AbstractModel)

Return the MOI.Name attribute of model's backend, or a default if empty.

# Example

```
julia> model = Model();
```

julia> name(model)
"A JuMP Model"

#### source

# node\_count

JuMP.node\_count - Function.

node\_count(model::GenericModel)

If available, returns the total number of branch-and-bound nodes explored during the most recent optimization in a Mixed Integer Program (the MOI.NodeCount attribute).

Throws a MOI.GetAttributeNotAllowed error if the attribute is not implemented by the solver.

# Example julia> import HiGHS

julia> model = Model(HiGHS.Optimizer);

```
julia> set_silent(model)
```

```
julia> optimize!(model)
julia> node_count(model)
0
```

#### source

## normalized\_coefficient

JuMP.normalized\_coefficient - Function.

```
normalized_coefficient(
 constraint::ConstraintRef,
 variable::GenericVariableRef,
)
```

Return the coefficient associated with variable in constraint after JuMP has normalized the constraint into its standard form.

```
See also set_normalized_coefficient.
```

#### Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @constraint(model, con, 2x + 3x <= 2)
con : 5 x ≤ 2
julia> normalized_coefficient(con, x)
5.0
julia> @constraint(model, con_vec, [x, 2x + 1, 3] >= 0)
con_vec : [x, 2 x + 1, 3] ∈ Nonnegatives()
julia> normalized_coefficient(con_vec, x)
2-element Vector{Tuple{Int64, Float64}}:
(1, 1.0)
(2, 2.0)
```

#### source

```
normalized_coefficient(
 constraint::ConstraintRef,
 variable_1::GenericVariableRef,
 variable_2::GenericVariableRef,
)
```

Return the quadratic coefficient associated with variable\_1 and variable\_2 in constraint after JuMP has normalized the constraint into its standard form.

See also set\_normalized\_coefficient.

## Example

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @constraint(model, con, 2x[1]^2 + 3 * x[1] * x[2] + x[2] <= 2)
con : 2 x[1]² + 3 x[1]*x[2] + x[2] < 2
julia> normalized_coefficient(con, x[1], x[1])
2.0
julia> normalized_coefficient(con, x[1], x[2])
3.0
julia> @constraint(model, con_vec, x.^2 <= [1, 2])
con_vec : [x[1]² - 1, x[2]² - 2] ∈ Nonpositives()
julia> normalized_coefficient(con_vec, x[1], x[1])
1-element Vector{Tuple{Int64, Float64}}:
(1, 1.0)
julia> normalized_coefficient(con_vec, x[1], x[2])
Tuple{Int64, Float64}[]
```

#### source

## normalized\_rhs

JuMP.normalized\_rhs - Function.

normalized\_rhs(constraint::ConstraintRef)

Return the right-hand side term of constraint after JuMP has converted the constraint into its normalized form.

See also set\_normalized\_rhs.

# Example

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, con, 2x + 1 <= 2)
con : 2 x ≤ 1
julia> normalized_rhs(con)
1.0
```

## num\_constraints

JuMP.num\_constraints - Function.

```
num_constraints(model::GenericModel, function_type, set_type)::Int64
```

Return the number of constraints currently in the model where the function has type function\_type and the set has type set\_type.

See also list\_of\_constraint\_types and all\_constraints.

# Example

```
julia> model = Model();
julia> @variable(model, x >= 0, Bin);
julia> @variable(model, y);
julia> @constraint(model, y in MOI.GreaterThan(1.0));
julia> @constraint(model, y <= 1.0);
julia> @constraint(model, 2x <= 1);
julia> num_constraints(model, VariableRef, MOI.GreaterThan{Float64})
2
julia> num_constraints(model, VariableRef, MOI.ZeroOne)
1
julia> num_constraints(model, AffExpr, MOI.LessThan{Float64})
2
```

#### source

num constraints(model::GenericModel; count variable in set constraints::Bool)

Return the number of constraints in model.

If count\_variable\_in\_set\_constraints == true, then VariableRef constraints such as VariableRefin-Integer are included. To count only the number of structural constraints (for example, the rows in the constraint matrix of a linear program), pass count\_variable\_in\_set\_constraints = false.

```
julia> model = Model();
julia> @variable(model, x >= 0, Int);
julia> @constraint(model, 2x <= 1);
julia> num_constraints(model; count_variable_in_set_constraints = true)
```

julia> num\_constraints(model; count\_variable\_in\_set\_constraints = false)
1

#### source

3

#### num\_variables

```
JuMP.num_variables - Function.
```

num\_variables(model::GenericModel)::Int64

Returns number of variables in model.

# Example

```
julia> model = Model();
```

julia> @variable(model, x[1:2]);

julia> num\_variables(model)
2

#### source

# object\_dictionary

JuMP.object\_dictionary - Function.

object\_dictionary(model::GenericModel)

Return the dictionary that maps the symbol name of a variable, constraint, or expression to the corresponding object.

Objects are registered to a specific symbol in the macros. For example, @variable(model, x[1:2, 1:2]) registers the array of variables x to the symbol :x.

This method should be defined for any subtype of AbstractModel.

See also: unregister.

## Example

julia> model = Model();

julia> @variable(model, x[1:2]);

julia> object\_dictionary(model)
Dict{Symbol, Any} with 1 entry:
 :x => VariableRef[x[1], x[2]]

794

#### objective\_bound

JuMP.objective\_bound - Function.

objective\_bound(model::GenericModel)

Return the best known bound on the optimal objective value after a call to optimize! (model).

For scalar-valued objectives, this function returns a Float64. For vector-valued objectives, it returns a Vector{Float64}.

In the case of a vector-valued objective, this returns the *ideal point*, that is, the point obtained if each objective was optimized independently.

This function is equivalent to querying the MOI.ObjectiveBound attribute.

## Example

```
julia> import HiGHS
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x >= 1, Int);
julia> @objective(model, Min, 2 * x + 1);
julia> optimize!(model)
julia> objective_bound(model)
3.0
```

#### source

# objective\_function

JuMP.objective\_function - Function.

```
objective_function(
 model::GenericModel,
 ::Type{F} = objective_function_type(model),
) where {F}
```

Return an object of type F representing the objective function.

Errors if the objective is not convertible to type F.

This function is equivalent to querying the MOI.ObjectiveFunction{F} attribute.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @objective(model, Min, 2x + 1)
2 x + 1
julia> objective_function(model, AffExpr)
2 x + 1
julia> objective_function(model, QuadExpr)
2 x + 1
julia> typeof(objective_function(model, QuadExpr))
QuadExpr (alias for GenericQuadExpr{Float64, GenericVariableRef{Float64}})
```

We see with the last two commands that even if the objective function is affine, as it is convertible to a quadratic function, it can be queried as a quadratic function and the result is quadratic.

However, it is not convertible to a variable:

```
julia> objective_function(model, VariableRef)
ERROR: InexactError: convert(MathOptInterface.VariableIndex, 1.0 + 2.0 MOI.VariableIndex(1))
[...]
```

source

# objective\_function\_string

JuMP.objective\_function\_string - Function.

objective\_function\_string(mode, model::AbstractModel)::String

Return a String describing the objective function of the model.

Example

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2 * x);
julia> objective_function_string(MIME("text/plain"), model)
"2 x"
```

# objective\_function\_type

```
JuMP.objective_function_type - Function.
```

objective\_function\_type(model::GenericModel)::AbstractJuMPScalar

Return the type of the objective function.

This function is equivalent to querying the MOI.ObjectiveFunctionType attribute.

## Example

```
julia> model = Model();
```

julia> @variable(model, x);

julia> @objective(model, Min, 2 \* x + 1);

julia> objective\_function\_type(model)
AffExpr (alias for GenericAffExpr{Float64, GenericVariableRef{Float64}})

#### source

## objective\_sense

JuMP.objective\_sense - Function.

objective\_sense(model::GenericModel)::MOI.OptimizationSense

Return the objective sense.

This function is equivalent to querying the MOI.ObjectiveSense attribute.

#### Example

```
julia> model = Model();
julia> objective_sense(model)
FEASIBILITY_SENSE::OptimizationSense = 2
```

julia> @variable(model, x);

julia> @objective(model, Max, x)
x

```
julia> objective_sense(model)
MAX SENSE::OptimizationSense = 1
```

# objective\_value

JuMP.objective\_value - Function.

```
objective_value(model::GenericModel; result::Int = 1)
```

Return the objective value associated with result index result of the most-recent solution returned by the solver.

For scalar-valued objectives, this function returns a Float64. For vector-valued objectives, it returns a Vector{Float64}.

This function is equivalent to querying the MOI.ObjectiveValue attribute.

See also: result\_count.

# Example

source

# op\_ifelse

JuMP.op\_ifelse - Function.

op\_ifelse(a, x, y)

A function that falls back to ifelse(a, x, y), but when called with a JuMP variables or expression in the first argument, returns a GenericNonlinearExpr.

```
julia> model = Model();
julia> @variable(model, x);
julia> op_ifelse(true, 1.0, 2.0)
1.0
julia> op_ifelse(x, 1.0, 2.0)
ifelse(x, 1.0, 2.0)
julia> op_ifelse(true, x, 2.0)
x
```

source

# op\_string

JuMP.op\_string - Function.

op\_string(mime::MIME, x::GenericNonlinearExpr, ::Val{op}) where {op}

Return the string that should be printed for the operator op when function\_string is called with mime and x.

# Example

```
julia> model = Model();
julia> @variable(model, x[1:2], Bin);
julia> f = @expression(model, x[1] || x[2]);
julia> op_string(MIME("text/plain"), f, Val(:||))
"||"
```

source

# operator\_to\_set

JuMP.operator\_to\_set - Function.

operator\_to\_set(error\_fn::Function, ::Val{sense\_symbol})

Converts a sense symbol to a set set such that @constraint(model, func sense\_symbol 0) is equivalent to @constraint(model, func in set) for any func::AbstractJuMPScalar.

## Example

Once a custom set is defined you can directly create a JuMP constraint with it:

However, there might be an appropriate sign that could be used in order to provide a more convenient syntax:

```
julia> JuMP.operator_to_set(::Function, ::Val{:[]}) = CustomSet(0.0)
julia> MOIU.supports_shift_constant(::Type{<:CustomSet}) = true
julia> MOIU.shift_constant(set::CustomSet, value) = CustomSet(set.value + value)
julia> cref = @constraint(model, x [1)
x € CustomSet{Float64}(1.0)
```

Note that the whole function is first moved to the right-hand side, then the sign is transformed into a set with zero constant and finally the constant is moved to the set with MOIU.shift\_constant.

#### source

## operator\_warn

JuMP.operator\_warn - Function.

```
operator_warn(model::AbstractModel)
operator_warn(model::GenericModel)
```

This function is called on the model whenever two affine expressions are added together without using destructive\_add!, and at least one of the two expressions has more than 50 terms.

For the case of Model, if this function is called more than 20,000 times then a warning is generated once.

This method should only be implemented by developers creating JuMP extensions. It should never be called by users of JuMP.

source

#### optimize!

JuMP.optimize! - Function.

```
optimize!(
 model::GenericModel;
 ignore_optimize_hook = (model.optimize_hook === nothing),
 kwargs...,
)
```

#### Optimize the model.

If an optimizer has not been set yet (see set\_optimizer), a NoOptimizer error is thrown.

If ignore\_optimize\_hook == true, the optimize hook is ignored and the model is solved as if the hook was not set. Keyword arguments kwargs are passed to the optimize\_hook. An error is thrown if optimize\_hook is nothing and keyword arguments are provided.

#### Example

```
julia> import HiGHS
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> function my_optimize_hook(model; foo)
 println("Hook called with foo = ", foo)
 return optimize!(model; ignore_optimize_hook = true)
 end
my_optimize_hook (generic function with 1 method)
julia> set_optimize_hook(model, my_optimize_hook)
my_optimize_hook (generic function with 1 method)
```

```
julia> optimize!(model; foo = 2)
Hook called with foo = 2
```

#### source

#### optimizer\_index

JuMP.optimizer\_index - Function.

```
optimizer_index(x::GenericVariableRef)::MOI.VariableIndex
optimizer_index(x::ConstraintRef{<:GenericModel})::MOI.ConstraintIndex</pre>
```

Return the variable or constraint index that corresponds to x in the associated model  $unsafe_backend(owner_model(x))$ .

This function should be used with unsafe\_backend.

As a safer alternative, use backend and index. See the docstrings of backend and unsafe\_backend for more details.

#### Throws

• Throws NoOptimizer if no optimizer is set.

- Throws an ErrorException if the optimizer is set but is not attached.
- Throws an ErrorException if the index is bridged.

# Example

julia> import HiGHS

julia> model = Model(HiGHS.Optimizer);

julia> set\_silent(model)

julia> @variable(model, x >= 0)

julia> MOI.Utilities.attach\_optimizer(model)

julia> highs = unsafe\_backend(model)
A HiGHS model with 1 columns and 0 rows.

julia> optimizer\_index(x)
MOI.VariableIndex(1)

#### source

#### optimizer\_with\_attributes

JuMP.optimizer\_with\_attributes - Function.

```
optimizer_with_attributes(optimizer_constructor, attrs::Pair...)
```

Groups an optimizer constructor with the list of attributes attrs. Note that it is equivalent to MOI.OptimizerWithAttributes.

When provided to the Model constructor or to set\_optimizer, it creates an optimizer by calling optimizer\_constructor(), and then sets the attributes using set\_attribute.

See also: set\_attribute, get\_attribute.

#### Note

The string names of the attributes are specific to each solver. One should consult the solver's documentation to find the attributes of interest.

## Example

is equivalent to:

julia> import HiGHS

julia> model = Model(HiGHS.Optimizer);

julia> set\_attribute(model, "presolve", "off")

julia> set\_attribute(model, MOI.Silent(), true)

#### source

# owner\_model

JuMP.owner\_model - Function.

owner\_model(s::AbstractJuMPScalar)

Return the model owning the scalar s.

# Example

julia> model = Model();

julia> @variable(model, x);

julia> owner\_model(x) === model
true

#### source

# parameter\_value

JuMP.parameter\_value - Function.

parameter\_value(x::GenericVariableRef)

Return the value of the parameter x.

Errors if x is not a parameter.

See also ParameterRef, is\_parameter, set\_parameter\_value.

#### Example

julia> model = Model(); julia> @variable(model, p in Parameter(2)) p julia> parameter\_value(p) 2.0

```
julia> set_parameter_value(p, 2.5)
julia> parameter_value(p)
2.5
```

source

### parse\_constraint

JuMP.parse\_constraint - Function.

parse\_constraint(error\_fn::Function, expr::Expr)

The entry-point for all constraint-related parsing.

# Arguments

- The error\_fn function is passed everywhere to provide better error messages
- expr comes from the @constraint macro. There are two possibilities:
  - @constraint(model, expr)
  - @constraint(model, name[args], expr)

In both cases, expr is the main component of the constraint.

## Supported syntax

JuMP currently supports the following expr objects:

- lhs <= rhs
- lhs == rhs
- lhs >= rhs
- l <= body <= u
- u >= body >= l
- lhs ⊥ rhs
- lhs in rhs
- lhs E rhs
- z --> {constraint}
- !z --> {constraint}
- z <--> {constraint}
- !z <--> {constraint}
- z => {constraint}
- !z => {constraint}

as well as all broadcasted variants.

#### Extensions

The infrastructure behind parse\_constraint is extendable. See parse\_constraint\_head and parse\_constraint\_call
for details.

# parse\_constraint\_call

JuMP.parse\_constraint\_call - Function.

```
parse_constraint_call(
 error_fn::Function,
 is_vectorized::Bool,
 ::Val{op},
 args...,
)
```

Implement this method to intercept the parsing of a :call expression with operator op.

#### Warning

Extending the constraint macro at parse time is an advanced operation and has the potential to interfere with existing JuMP syntax. Please discuss with the developer chatroom before publishing any code that implements these methods.

## Arguments

- error\_fn: a function that accepts a String and throws the string as an error, along with some descriptive information of the macro from which it was thrown.
- is\_vectorized: a boolean to indicate if op should be broadcast or not
- · op: the first element of the .args field of the Expr to intercept
- args...: the .args field of the Expr.

#### Returns

This function must return:

- parse\_code::Expr: an expression containing any setup or rewriting code that needs to be called before build\_constraint
- build\_code::Expr: an expression that calls build\_constraint( or build\_constraint.( depending on is\_vectorized.

See also: parse\_constraint\_head, build\_constraint

```
source
```

```
parse_constraint_call(
 error_fn::Function,
 vectorized::Bool,
 ::Val{op},
 lhs,
 rhs,
) where {op}
```

Fallback handler for binary operators. These might be infix operators like @constraint(model, lhs op rhs), or normal operators like @constraint(model, op(lhs, rhs)).

In both cases, we rewrite as lhs - rhs in operator\_to\_set(error\_fn, op).

See operator\_to\_set for details.

source

#### parse\_constraint\_head

JuMP.parse\_constraint\_head - Function.

parse\_constraint\_head(error\_fn::Function, ::Val{head}, args...)

Implement this method to intercept the parsing of an expression with head head.

#### Warning

Extending the constraint macro at parse time is an advanced operation and has the potential to interfere with existing JuMP syntax. Please discuss with the developer chatroom before publishing any code that implements these methods.

#### Arguments

- error\_fn: a function that accepts a String and throws the string as an error, along with some descriptive information of the macro from which it was thrown.
- · head: the . head field of the Expr to intercept
- args...: the .args field of the Expr.

#### Returns

This function must return:

- is\_vectorized::Bool: whether the expression represents a broadcasted expression like x .<= 1
- parse\_code::Expr: an expression containing any setup or rewriting code that needs to be called before build\_constraint
- build\_code::Expr: an expression that calls build\_constraint( or build\_constraint.( depending on is\_vectorized.

# **Existing implementations**

JuMP currently implements:

- ::Val{:call}, which forwards calls to parse\_constraint\_call
- ::Val{:comparison}, which handles the special case of l <= body <= u.

See also: parse\_constraint\_call, build\_constraint

source

## parse\_one\_operator\_variable

JuMP.parse\_one\_operator\_variable - Function.

```
parse_one_operator_variable(
 error_fn::Function,
 info_expr::_VariableInfoExpr,
 sense::Val{S},
 value,
) where {S}
```

Update infoexr for a variable expression in the @variable macro of the form variable name S value. source

#### parse\_ternary\_variable

JuMP.parse ternary variable - Function.

parse\_ternary\_variable(error\_fn, info\_expr, lhs\_sense, lhs, rhs\_sense, rhs)

A hook for JuMP extensions to intercept the parsing of a :comparison expression, which has the form lhs lhs\_sense variable rhs\_sense rhs.

```
source
```

# parse\_variable

JuMP.parse\_variable - Function.

parse\_variable(error\_fn::Function, ::\_VariableInfoExpr, args...)

A hook for extensions to intercept the parsing of inequality constraints in the @variable macro.

source

# primal\_feasibility\_report

JuMP.primal\_feasibility\_report - Function.

```
primal_feasibility_report(
 model::GenericModel{T},
 point::AbstractDict{GenericVariableRef{T},T} = _last_primal_solution(model),
 atol::T = zero(T),
 skip_missing::Bool = false,
)::Dict{Any,T}
```

Given a dictionary point, which maps variables to primal values, return a dictionary whose keys are the constraints with an infeasibility greater than the supplied tolerance atol. The value corresponding to each key is the respective infeasibility. Infeasibility is defined as the distance between the primal value of the constraint (see MOI.ConstraintPrimal) and the nearest point by Euclidean distance in the corresponding set.

Notes

- If skip\_missing = true, constraints containing variables that are not in point will be ignored.
- If skip\_missing = false and a partial primal solution is provided, an error will be thrown.
- If no point is provided, the primal solution from the last time the model was solved is used.

# Example

#### source

```
primal_feasibility_report(
 point::Function,
 model::GenericModel{T};
 atol::T = zero(T),
 skip_missing::Bool = false,
) where {T}
```

A form of primal\_feasibility\_report where a function is passed as the first argument instead of a dictionary as the second argument.

#### Example

#### source

# primal\_status

JuMP.primal\_status - Function.

```
primal_status(model::GenericModel; result::Int = 1)
```

Return a MOI.ResultStatusCode describing the status of the most recent primal solution of the solver (that is, the MOI.PrimalStatus attribute) associated with the result index result.

See also: result\_count.

## Example

julia> import Ipopt julia> model = Model(Ipopt.Optimizer); julia> primal\_status(model; result = 2) NO\_SOLUTION::ResultStatusCode = 0

#### source

#### print\_active\_bridges

JuMP.print\_active\_bridges - Function.

```
print_active_bridges([io:::I0 = stdout,] model::GenericModel)
```

Print a list of the variable, constraint, and objective bridges that are currently used in the model.

source

```
print_active_bridges([io::IO = stdout,] model::GenericModel, ::Type{F}) where {F}
```

Print a list of bridges required for an objective function of type F.

source

```
print_active_bridges(
 [io::I0 = stdout,]
 model::GenericModel,
 F::Type,
 S::Type{<:MOI.AbstractSet},
)</pre>
```

Print a list of bridges required for a constraint of type F-in-S.

```
source
```

```
print_active_bridges(
 [io::IO = stdout,]
 model::GenericModel,
 S::Type{<:MOI.AbstractSet},
)</pre>
```

Print a list of bridges required to add a variable constrained to the set S.

## print\_bridge\_graph

JuMP.print\_bridge\_graph - Function.

print\_bridge\_graph([io:::I0,] model::GenericModel)

Print the hyper-graph containing all variable, constraint, and objective types that could be obtained by bridging the variables, constraints, and objectives that are present in the model.

#### Warning

This function is intended for advanced users. If you want to see only the bridges that are currently used, use print\_active\_bridges instead.

#### **Explanation of output**

Each node in the hyper-graph corresponds to a variable, constraint, or objective type.

- Variable nodes are indicated by [ ]
- Constraint nodes are indicated by ( )
- Objective nodes are indicated by | |

The number inside each pair of brackets is an index of the node in the hyper-graph.

Note that this hyper-graph is the full list of possible transformations. When the bridged model is created, we select the shortest hyper-path(s) from this graph, so many nodes may be un-used.

For more information, see Legat, B., Dowson, O., Garcia, J., and Lubin, M. (2020). "MathOptInterface: a data structure for mathematical optimization problems." URL: https://arxiv.org/abs/2002.03447

source

## print\_macro\_timing\_summary

JuMP.print\_macro\_timing\_summary - Function.

print\_macro\_timing\_summary([io::IO = stdout], model::GenericModel)

Print a summary of the runtime of each macro.

Before calling this method, you must have enabled the macro timing feature using set macro timing.

Example

```
julia> begin
 model = Model()
 set_macro_timing(model, true)
 @variable(model, x[1:2])
 @objective(model, Min, sum(x))
 end;
```

julia> print\_macro\_timing\_summary(model)

source

# quad\_terms

JuMP.quad\_terms - Function.

quad\_terms(quad::GenericQuadExpr{C,V})

Provides an iterator over tuples (coefficient::C, var\_1::V, var\_2::V) in the quadratic part of the quadratic expression.

source

## raw\_status

JuMP.raw\_status - Function.

raw\_status(model::GenericModel)

Return the reason why the solver stopped in its own words (that is, the MathOptInterface model attribute MOI.RawStatusString).

#### Example

julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> raw\_status(model)
"optimize not called"

## source

# read\_from\_file

JuMP.read\_from\_file - Function.

```
read_from_file(
 filename::String;
 format::MOI.FileFormats.FileFormat = MOI.FileFormats.FORMAT_AUTOMATIC,
 kwargs...,
)
```

Return a JuMP model read from filename in the format format.

See MOI.FileFormats.FileFormat for a list of supported formats.

## **Keyword arguments**

Other kwargs are passed to the Model constructor of the chosen format.

For details, see the docstring each file format's Model constructor. For example, MOI.FileFormats.MPS.Model.

#### **Nonlinear models**

To maintain backwards compatibility, nonlinear models in .mof.json and .nl files are parsed into a MOI.NLPBlock. To parse as MOI.ScalarNonlinearFunction, pass the keyword use\_nlp\_block = false.

#### Compression

If the filename ends in .gz, the file will be uncompressed using GZip.

If the filename ends in .bz2, the file will be uncompressed using BZip2.

# Example

## julia> print(new\_model)

 $\begin{array}{l} \text{Min log}(x) \\ \text{Subject to} \\ x \ge 0 \end{array}$ 

#### reduced\_cost

JuMP.reduced\_cost - Function.

reduced\_cost(x::GenericVariableRef{T})::T where {T}

Return the reduced cost associated with variable x.

One interpretation of the reduced cost is that it is the change in the objective from an infinitesimal relaxation of the variable bounds.

This method is equivalent to querying the shadow price of the active variable bound (if one exists and is active).

See also: shadow\_price.

## Example

julia> import HiGHS julia> model = Model(HiGHS.Optimizer); julia> set\_silent(model) julia> @variable(model, x <= 1); julia> @objective(model, Max, 2 \* x + 1); julia> optimize!(model) julia> dual\_status(model) FEASIBLE\_POINT::ResultStatusCode = 1 julia> reduced\_cost(x) 2.0

#### source

## relative\_gap

JuMP.relative\_gap - Function.

relative\_gap(model::GenericModel)

Return the final relative optimality gap after a call to optimize! (model).

Exact value depends upon implementation of MOI.RelativeGap by the particular solver used for optimization.

This function is equivalent to querying the MOI.RelativeGap attribute.

julia> import HiGHS

```
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x >= 1, Int);
julia> @objective(model, Min, 2 * x + 1);
julia> optimize!(model)
julia> relative_gap(model)
0.0
```

source

## relax\_integrality

JuMP.relax\_integrality - Function.

```
relax_integrality(model::GenericModel)
```

Modifies model to "relax" all binary and integrality constraints on variables. Specifically,

- Binary constraints are deleted, and variable bounds are tightened if necessary to ensure the variable is constrained to the interval [0, 1].
- Integrality constraints are deleted without modifying variable bounds.
- An error is thrown if semi-continuous or semi-integer constraints are present (support may be added for these in the future).
- All other constraints are ignored (left in place). This includes discrete constraints like SOS and indicator constraints.

Returns a function that can be called without any arguments to restore the original model. The behavior of this function is undefined if additional changes are made to the affected variables in the meantime.

```
julia> model = Model();
julia> @variable(model, x, Bin);
julia> @variable(model, 1 <= y <= 10, Int);
julia> @objective(model, Min, x + y);
julia> undo_relax = relax_integrality(model);
julia> print(model)
Min x + y
Subject to
```

x ≥ 0 y ≥ 1 x ≤ 1 y ≤ 10 julia> undo\_relax() julia> print(model) Min x + y Subject to y ≥ 1 y ≤ 10 y integer x binary

source

## relax\_with\_penalty!

JuMP.relax\_with\_penalty! - Function.

```
relax_with_penalty!(
 model::GenericModel{T},
 [penalties::Dict{ConstraintRef,T}];
 [default::Union{Nothing,Real} = nothing,]
) where {T}
```

Destructively modify the model in-place to create a penalized relaxation of the constraints.

#### Warning

This is a destructive routine that modifies the model in-place. If you don't want to modify the original model, use copy\_model to create a copy before calling relax\_with\_penalty!.

#### Reformulation

See MOI.Utilities.ScalarPenaltyRelaxation for details of the reformulation.

For each constraint ci, the penalty passed to MOI.Utilities.ScalarPenaltyRelaxation is get(penalties, ci, default). If the value is nothing, because ci does not exist in penalties and default = nothing, then the constraint is skipped.

# **Return value**

This function returns a Dict{ConstraintRef,AffExpr} that maps each constraint index to the corresponding y + z as an AffExpr. In an optimal solution, query the value of these functions to compute the violation of each constraint.

#### **Relax a subset of constraints**

To relax a subset of constraints, pass a penalties dictionary and set default = nothing.

```
julia> function new_model()
 model = Model()
 @variable(model, x)
 @objective(model, Max, 2x + 1)
 @constraint(model, c1, 2x - 1 <= -2)</pre>
 (constraint(model, c2, 3x >= 0))
 return model
 end
new_model (generic function with 1 method)
julia> model_1 = new_model();
julia> penalty_map = relax_with_penalty!(model_1; default = 2.0);
julia> penalty_map[model_1[:c1]]
[3]
julia> penalty_map[model_1[:c2]]
[2]
julia> print(model_1)
Max 2 x - 2 [2] - 2 [3] + 1
Subject to
c2 : 3 \times + [2] \ge 0
c1 : 2 x - _[3] ≤ -1
 _[2] ≥ 0
 _[3] ≥ 0
julia> model_2 = new_model();
julia> relax_with_penalty!(model_2, Dict(model_2[:c2] => 3.0))
Dict{ConstraintRef{Model,
← MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
→ MathOptInterface.GreaterThan{Float64}}, ScalarShape}, AffExpr} with 1 entry:
c2 : 3 \times + [2] \ge 0 \Longrightarrow [2]
julia> print(model 2)
Max 2 x - 3 [2] + 1
Subject to
c2 : 3 \times + [2] \ge 0
```

source

c1 : 2 x ≤ -1 \_[2] ≥ 0

#### remove\_bridge

JuMP.remove\_bridge - Function.

```
remove_bridge(
 model::GenericModel{S},
 BT::Type{<:MOI.Bridges.AbstractBridge};
 coefficient_type::Type{T} = S,
) where {S,T}</pre>
```

Remove BT{T} from the list of bridges that can be used to transform unsupported constraints into an equivalent formulation using only constraints supported by the optimizer.

See also: add\_bridge.

#### Example

# source

# reshape\_set

JuMP.reshape\_set - Function.

reshape\_set(vectorized\_set::MOI.AbstractSet, shape::AbstractShape)

Return a set in its original shape shape given its vectorized form vectorized\_form.

# Example

Given a SymmetricMatrixShape of vectorized form [1, 2, 3] in MOI.PositiveSemidefinieConeTriangle(2), the following code returns the set of the original constraint Symmetric(Matrix[1 2; 2 3]) in PSDCone():

julia> reshape\_set(MOI.PositiveSemidefiniteConeTriangle(2), SymmetricMatrixShape(2))
PSDCone()

## source

#### reshape\_vector

JuMP.reshape\_vector - Function.

reshape\_vector(vectorized\_form::Vector, shape::AbstractShape)

# CHAPTER 22. DOCSTRINGS

Return an object in its original shape shape given its vectorized form vectorized\_form.

#### Example

Given a SymmetricMatrixShape of vectorized form [1, 2, 3], the following code returns the matrix Symmetric(Matrix[1 2; 2 3]):

```
julia> reshape_vector([1, 2, 3], SymmetricMatrixShape(2))
2×2 LinearAlgebra.Symmetric{Int64, Matrix{Int64}}:
 1 2
 2 3
```

source

# result\_count

JuMP.result\_count - Function.

result\_count(model::GenericModel)

Return the number of results available to query after a call to optimize!.

# Example

```
julia> import Ipopt
julia> model = Model(Ipopt.Optimizer);
julia> result_count(model)
0
```

## source

#### reverse\_sense

JuMP.reverse\_sense - Function.

reverse\_sense(:::Val{T}) where {T}

Given an (in)equality symbol T, return a new Val object with the opposite (in)equality symbol.

This function is intended for use in JuMP extensions.

# Example

julia> reverse\_sense(Val(:>=))
Val{:<=}()</pre>

# set\_attribute

JuMP.set\_attribute - Function.

```
set_attribute(model::GenericModel, attr::MOI.AbstractModelAttribute, value)
set_attribute(x::GenericVariableRef, attr::MOI.AbstractVariableAttribute, value)
set_attribute(cr::ConstraintRef, attr::MOI.AbstractConstraintAttribute, value)
```

Set the value of a solver-specifc attribute attr to value.

This is equivalent to calling MOI.set with the associated MOI model and, for variables and constraints, with the associated MOI.VariableIndex or MOI.ConstraintIndex.

#### Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @constraint(model, c, 2 * x <= 1)
c : 2 x ≤ 1
julia> set_attribute(model, MOI.Name(), "model_new")
julia> set_attribute(x, MOI.VariableName(), "x_new")
julia> set_attribute(c, MOI.ConstraintName(), "c_new")
```

#### source

```
set_attribute(
 model::Union{GenericModel,MOI.OptimizerWithAttributes},
 attr::Union{AbstractString,MOI.AbstractOptimizerAttribute},
 value,
)
```

Set the value of a solver-specifc attribute attr to value.

This is equivalent to calling MOI.set with the associated MOI model.

If attr is an AbstractString, it is converted to MOI.RawOptimizerAttribute.

```
julia> import HiGHS
julia> opt = optimizer_with_attributes(HiGHS.Optimizer, "output_flag" => false);
julia> model = Model(opt);
julia> set_attribute(model, "output_flag", false)
julia> set_attribute(model, MOI.RawOptimizerAttribute("output_flag"), true)
```

julia> set\_attribute(opt, "output\_flag", true)

julia> set\_attribute(opt, MOI.RawOptimizerAttribute("output\_flag"), false)

#### source

# set\_attributes

JuMP.set\_attributes - Function.

```
set_attributes(
 destination::Union{
 GenericModel,
 MOI.OptimizerWithAttributes,
 GenericVariableRef,
 ConstraintRef,
 },
 pairs::Pair...,
)
```

Given a list of attribute => value pairs, calls set\_attribute(destination, attribute, value) for each pair.

See also: set\_attribute, get\_attribute.

# Example

julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> set\_attributes(model, "tol" => 1e-4, "max\_iter" => 100)

is equivalent to:

julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> set\_attribute(model, "tol", 1e-4)

julia> set\_attribute(model, "max\_iter", 100)

# source

#### set\_binary

JuMP.set\_binary - Function.
```
set_binary(v::GenericVariableRef)
```

Add a constraint on the variable v that it must take values in the set  $\{0, 1\}$ .

See also BinaryRef, is\_binary, unset\_binary.

#### Example

```
julia> model = Model();
```

```
julia> @variable(model, x);
```

julia> is\_binary(x)
false

julia> set\_binary(x)

julia> is\_binary(x)
true

#### source

### set\_dual\_start\_value

JuMP.set\_dual\_start\_value - Function.

```
set_dual_start_value(con_ref::ConstraintRef, value)
```

Set the dual start value (MOI attribute ConstraintDualStart) of the constraint con\_ref to value.

```
To remove a dual start value set it to nothing.
```

```
See also dual_start_value.
```

## Example

julia> model = Model();

julia> @variable(model, x, start = 2.0);

```
julia> @constraint(model, c, [2x] in Nonnegatives())
c : [2 x] ∈ Nonnegatives()
```

julia> set\_dual\_start\_value(c, [0.0])

```
julia> dual_start_value(c)
1-element Vector{Float64}:
0.0
```

julia> set\_dual\_start\_value(c, nothing)

```
julia> dual_start_value(c)
```

source

## set\_integer

JuMP.set\_integer - Function.

set\_integer(variable\_ref::GenericVariableRef)

Add an integrality constraint on the variable variable\_ref.

See also IntegerRef, is\_integer, unset\_integer.

## Example

```
julia> model = Model();
julia> @variable(model, x);
julia> is_integer(x)
false
julia> set_integer(x)
julia> is_integer(x)
true
```

#### source

### set\_lower\_bound

JuMP.set\_lower\_bound - Function.

set\_lower\_bound(v::GenericVariableRef, lower::Number)

Set the lower bound of a variable. If one does not exist, create a new lower bound constraint.

See also LowerBoundRef, has\_lower\_bound, lower\_bound, delete\_lower\_bound.

```
julia> model = Model();
julia> @variable(model, x >= 1.0);
julia> lower_bound(x)
1.0
julia> set_lower_bound(x, 2.0)
julia> lower_bound(x)
2.0
```

### set\_macro\_timing

JuMP.set\_macro\_timing - Function.

```
set_macro_timing(:::GenericModel, value::Bool)
```

Turn on (if value, or off, if !value) JuMP's built-in profiling of model construction macros.

Use print\_macro\_timing\_summary to display a summary.

### Example

```
julia> begin
 model = Model()
 set_macro_timing(model, true)
 @variable(model, x[1:2])
 @objective(model, Min, sum(x))
 end;
```

#### source

# set\_name

JuMP.set\_name - Function.

set\_name(con\_ref::ConstraintRef, s::AbstractString)

Set a constraint's name attribute.

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, c, [2x] in Nonnegatives())
c : [2 x] ∈ Nonnegatives()
julia> set_name(c, "my_constraint")
julia> name(c)
```

## CHAPTER 22. DOCSTRINGS

"my\_constraint"

julia> c
my\_constraint : [2 x] E Nonnegatives()

#### source

set\_name(v::GenericVariableRef, s::AbstractString)

Set a variable's name attribute.

### Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> set_name(x, "x_foo")
```

**julia>** x

x\_foo

julia> name(x)
"x\_foo"

### source

set\_name(model::GenericModel, name::AbstractString)

Set the MOI.Name attribute of model's backend to name.

## Example

julia> model = Model();

julia> set\_name(model, "My Model")

**julia>** name(model) "My Model"

### source

## set\_normalized\_coefficient

JuMP.set\_normalized\_coefficient - Function.

```
set_normalized_coefficient(
 constraint::ConstraintRef,
 variable::GenericVariableRef,
 value::Number,
)
```

Set the coefficient of variable in the constraint constraint to value.

Note that prior to this step, JuMP will aggregate multiple terms containing the same variable. For example, given a constraint  $2x + 3x \le 2$ , set\_normalized\_coefficient(con, x, 4) will create the constraint  $4x \le 2$ .

### Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @constraint(model, con, 2x + 3x <= 2)
con : 5 x ≤ 2
```

julia> set\_normalized\_coefficient(con, x, 4)

**julia> con** con :  $4 \times \le 2$ 

#### source

```
set_normalized_coefficient(
 constraints::AbstractVector{
 <:ConstraintRef{<:AbstractModel,<:MOI.ConstraintIndex{F}},
 },
 variables::AbstractVector{<:AbstractVariableRef},
 coeffs::AbstractVector{<:Number},
) where {
 T,
 F<:Union{MOI.ScalarAffineFunction{T},MOI.ScalarQuadraticFunction{T}},
}</pre>
```

Set multiple coefficient of variables in the constraints constraints to coeffs.

#### **Concrete types**

Note that constraints must be a concrete vector of a single constraint type. You cannot mix, for example, <= and >= constraints in the same vector.

### Normalization

Note that prior to this step, JuMP will aggregate multiple terms containing the same variable. For example, given a constraint  $2x + 3x \le 2$ , set\_normalized\_coefficient(con, [x], [4]) will create the constraint  $4x \le 2$ .

```
julia> @variable(model, x)
x
julia> @variable(model, y)
y
```

julia> model = Model();

julia> @constraint(model, con,  $2x + 3x + 4y \le 2$ ) con : 5 x + 4 y  $\le 2$ 

julia> set\_normalized\_coefficient([con, con], [x, y], [6, 7])

julia> con con :  $6 \times + 7 y \le 2$ 

## source

```
set_normalized_coefficient(
 con_ref::ConstraintRef,
 variable::AbstractVariableRef,
 new_coefficients::Vector{Tuple{Int64,T}},
)
```

Set the coefficients of variable in the constraint con\_ref to new\_coefficients, where each element in new\_coefficients is a tuple which maps the row to a new coefficient.

Note that prior to this step, during constraint creation, JuMP will aggregate multiple terms containing the same variable.

## Example

julia> @variable(model, x)
x

julia> model = Model();

julia> @constraint(model, con, [2x + 3x, 4x] in MOI.Nonnegatives(2))
con : [5 x, 4 x] ∈ MathOptInterface.Nonnegatives(2)

julia> set\_normalized\_coefficient(con, x, [(1, 2.0), (2, 5.0)])

julia> con con : [2 x, 5 x] ∈ MathOptInterface.Nonnegatives(2)

## source

```
set_normalized_coefficient(
 constraint::ConstraintRef,
 variable_1:GenericVariableRef,
 variable_2:GenericVariableRef,
 value::Number,
)
```

Set the quadratic coefficient associated with variable\_1 and variable\_2 in the constraint constraint to value.

Note that prior to this step, JuMP will aggregate multiple terms containing the same variable. For example, given a constraint  $2x^2 + 3x^2 \le 2$ , set\_normalized\_coefficient(con, x, x, 4) will create the constraint  $4x^2 \le 2$ .

### Example

julia> model = Model();

julia> @variable(model, x[1:2]);

julia> @constraint(model, con,  $2x[1]^2 + 3 * x[1] * x[2] + x[2] <= 2$ ) con :  $2 x[1]^2 + 3 x[1]*x[2] + x[2] \le 2$ 

julia> set\_normalized\_coefficient(con, x[1], x[1], 4)

```
julia> set_normalized_coefficient(con, x[1], x[2], 5)
```

julia> con con : 4 x[1]<sup>2</sup> + 5 x[1]\*x[2] + x[2] ≤ 2

### source

```
set_normalized_coefficient(
 constraints::AbstractVector{<:ConstraintRef},
 variables_1:AbstractVector{<:GenericVariableRef},
 variables_2:AbstractVector{<:GenericVariableRef},
 values::AbstractVector{<:Number},
)</pre>
```

Set multiple quadratic coefficients associated with variables\_1 and variables\_2 in the constraints constraints to values.

Note that prior to this step, JuMP will aggregate multiple terms containing the same variable. For example, given a constraint  $2x^2 + 3x^2 \le 2$ , set\_normalized\_coefficient(con, [x], [x], [4]) will create the constraint  $4x^2 \le 2$ .

## Example

```
julia> model = Model();
```

julia> @variable(model, x[1:2]);

julia> @constraint(model, con, 2x[1]^2 + 3 \* x[1] \* x[2] + x[2] <= 2) con : 2 x[1]<sup>2</sup> + 3 x[1]\*x[2] + x[2] ≤ 2

julia> set\_normalized\_coefficient([con, con], [x[1], x[1]], [x[1], x[2]], [4, 5])

julia> con con : 4 x[1]<sup>2</sup> + 5 x[1]\*x[2] + x[2] ≤ 2

### set\_normalized\_rhs

JuMP.set\_normalized\_rhs - Function.

set\_normalized\_rhs(constraint::ConstraintRef, value::Number)

Set the right-hand side term of constraint to value.

Note that prior to this step, JuMP will aggregate all constant terms onto the right-hand side of the constraint. For example, given a constraint  $2x + 1 \le 2$ , set\_normalized\_rhs(con, 4) will create the constraint  $2x \le 4$ , not  $2x + 1 \le 4$ .

### Example

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, con, 2x + 1 <= 2)
con : 2 x ≤ 1
```

julia> set\_normalized\_rhs(con, 4)

**julia>** con con :  $2 \times \le 4$ 

#### source

```
set_normalized_rhs(
 constraints::AbstractVector{<:ConstraintRef},
 values::AbstractVector{<:Number}
)</pre>
```

Set the right-hand side terms of all constraints to values.

Note that prior to this step, JuMP will aggregate all constant terms onto the right-hand side of the constraint. For example, given a constraint  $2x + 1 \le 2$ , set\_normalized\_rhs([con], [4]) will create the constraint  $2x \le 4$ , not  $2x + 1 \le 4$ .

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, con1, 2x + 1 <= 2)
con1 : 2 x ≤ 1
julia> @constraint(model, con2, 3x + 2 <= 4)
con2 : 3 x ≤ 2
julia> set_normalized_rhs([con1, con2], [4, 5])
```

```
julia> con1
con1 : 2 x ≤ 4
julia> con2
con2 : 3 x ≤ 5
```

source

### set\_objective

JuMP.set\_objective - Function.

set\_objective(model::AbstractModel, sense::MOI.OptimizationSense, func)

The functional equivalent of the <a>@objective</a> macro.

This function sets the objective sense and objective function simultaneously, and it is equivalent to calling set\_objective\_sense followed by set\_objective\_function.

This is a low-level function; the recommended way to set the objective function and sense is with the <a href="mailto:dobjective">@objective</a> macro.

#### FEASIBILITY\_SENSE

You should not set sense to FEASIBILITY\_SENSE because FEASIBILITY\_SENSE implies that there is no objective function.

Instead of set\_objective(model, FEASIBILITY\_SENSE, f), do set\_objective\_sense(model, FEASIBILITY\_SENSE).

#### Example

```
julia> model = Model();
julia> @variable(model, x);
julia> set_objective(model, MIN_SENSE, x)
julia> objective_sense(model)
MIN_SENSE::OptimizationSense = 0
julia> objective_function(model)
x
```

#### source

# set\_objective\_coefficient

JuMP.set\_objective\_coefficient - Function.

```
set_objective_coefficient(
 model::GenericModel,
 variable::GenericVariableRef,
 coefficient::Real,
)
```

Set the linear objective coefficient associated with variable to coefficient.

Note: this function will throw an error if a nonlinear objective is set.

#### Example

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, 2x + 1)
2 x + 1
julia> set_objective_coefficient(model, x, 3)
julia> objective_function(model)
3 x + 1
```

### source

```
set_objective_coefficient(
 model::GenericModel,
 variables::Vector{<:GenericVariableRef},
 coefficients::Vector{<:Real},
)</pre>
```

Set multiple linear objective coefficients associated with variables to coefficients, in a single call.

Note: this function will throw an error if a nonlinear objective is set.

## Example

```
julia> model = Model();
julia> @variable(model, x);
julia> @variable(model, y);
julia> @objective(model, Min, 3x + 2y + 1)
3 x + 2 y + 1
julia> set_objective_coefficient(model, [x, y], [5, 4])
julia> objective_function(model)
```

5 x + 4 y + 1

#### source

```
set_objective_coefficient(
 model::GenericModel{T},
 variable_1::GenericVariableRef{T},
 variable_2::GenericVariableRef{T},
 coefficient::Real,
) where {T}
```

Set the quadratic objective coefficient associated with variable\_1 and variable\_2 to coefficient.

Note: this function will throw an error if a nonlinear objective is set.

#### Example

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @objective(model, Min, x[1]^2 + x[1] * x[2])
x[1]² + x[1]*x[2]
julia> set_objective_coefficient(model, x[1], x[1], 2)
julia> set_objective_coefficient(model, x[1], x[2], 3)
julia> objective_function(model)
2 x[1]² + 3 x[1]*x[2]
```

### source

```
set_objective_coefficient(
 model::GenericModel{T},
 variables_1::AbstractVector{<:GenericVariableRef{T}},
 variables_2::AbstractVector{<:GenericVariableRef{T}},
 coefficients::AbstractVector{<:Real},
) where {T}</pre>
```

Set multiple quadratic objective coefficients associated with variables\_1 and variables\_2 to coefficients, in a single call.

Note: this function will throw an error if a nonlinear objective is set.

#### Example

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> @objective(model, Min, x[1]^2 + x[1] * x[2])
x[1]² + x[1]*x[2]
julia> set_objective_coefficient(model, [x[1], x[1]], [x[1], x[2]], [2, 3])
julia> objective_function(model)
2 x[1]² + 3 x[1]*x[2]
```

## source

### set\_objective\_function

JuMP.set\_objective\_function - Function.

```
set_objective_function(model::GenericModel, func::MOI.AbstractFunction)
set_objective_function(model::GenericModel, func::AbstractJuMPScalar)
set_objective_function(model::GenericModel, func::Real)
set_objective_function(model::GenericModel, func::Vector{<:AbstractJuMPScalar})</pre>
```

Set the objective function of model to the given function func.

```
See also: @objective, set_objective_function, set_objective
```

### Example

```
julia> model = Model();
julia> @variable(model, x);
julia> @objective(model, Min, x);
julia> objective_function(model)
x
```

julia> set\_objective\_function(model, 2 \* x + 1)

julia> objective\_function(model)
2 x + 1

#### source

### set\_objective\_sense

JuMP.set\_objective\_sense - Function.

set\_objective\_sense(model::GenericModel, sense::MOI.OptimizationSense)

Sets the objective sense of the model to the given sense.

See also: @objective, set\_objective\_function, set\_objective

## FEASIBILITY\_SENSE

Setting the objective sense to FEASIBILITY\_SENSE will remove any existing objective.

#### Example

julia> model = Model();

```
julia> objective_sense(model)
FEASIBILITY_SENSE::OptimizationSense = 2
```

julia> set\_objective\_sense(model, MOI.MAX\_SENSE)

julia> objective\_sense(model)
MAX\_SENSE::OptimizationSense = 1

source

#### set\_optimize\_hook

JuMP.set\_optimize\_hook - Function.

set\_optimize\_hook(model::GenericModel, f::Union{Function,Nothing})

Set the function f as the optimize hook for model.

f should have a signature f(model::GenericModel; kwargs...), where the kwargs are those passed to
optimize!.

### Notes

- The optimize hook should generally modify the model, or some external state in some way, and then call optimize!(model; ignore\_optimize\_hook = true) to optimize the problem, bypassing the hook.
- Use set\_optimize\_hook(model, nothing) to unset an optimize hook.

#### Example

julia> model = Model(); julia> function my\_hook(model::Model; kwargs...)

```
println(kwargs)
println("Calling with `ignore_optimize_hook = true`")
optimize!(model; ignore_optimize_hook = true)
return
end
```

my\_hook (generic function with 1 method)

```
julia> set_optimize_hook(model, my_hook)
my hook (generic function with 1 method)
```

```
julia> optimize!(model; test_arg = true)
Base.Pairs{Symbol, Bool, Tuple{Symbol}, @NamedTuple{test_arg::Bool}}(:test_arg => 1)
Calling with `ignore_optimize_hook = true`
ERROR: NoOptimizer()
[...]
```

source

### set\_optimizer

JuMP.set\_optimizer - Function.

```
set_optimizer(
 model::GenericModel,
 optimizer_factory;
 add_bridges::Bool = true,
 kwargs...,
)
```

Creates an empty MOI.AbstractOptimizer instance by calling MOI.instantiate on optimizer\_factory and sets it as the optimizer of model.

Specifically, optimizer\_factory must be callable with zero arguments and return an empty MOI.AbstractOptimizer.

If add\_bridges is true, constraints and objectives that are not supported by the optimizer are automatically bridged to equivalent supported formulation. Passing add\_bridges = false can improve performance if the solver natively supports all of the elements in model.

Additional kwargs are passed to MOI.instantiate.

See set\_attribute for setting solver-specific parameters of the optimizer.

#### Example

julia> import HiGHS

julia> model = Model();

julia> set\_optimizer(model, () -> HiGHS.Optimizer())

julia> set\_optimizer(model, HiGHS.Optimizer; add\_bridges = false)

### source

### set\_parameter\_value

JuMP.set\_parameter\_value - Function.

```
set_parameter_value(x::GenericVariableRef, value)
```

Update the parameter constraint on the variable x to value.

Errors if x is not a parameter.

See also ParameterRef, is\_parameter, parameter\_value.

#### Example

```
julia> model = Model();
julia> @variable(model, p in Parameter(2))
p
julia> parameter_value(p)
2.0
julia> set_parameter_value(p, 2.5)
julia> parameter_value(p)
2.5
```

source

### set\_silent

JuMP.set\_silent - Function.

set\_silent(model::GenericModel)

Takes precedence over any other attribute controlling verbosity and requires the solver to produce no output.

See also: unset\_silent.

## Example

julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> set\_silent(model)

julia> get\_attribute(model, MOI.Silent())
true

julia> unset\_silent(model)

julia> get\_attribute(model, MOI.Silent())
false

#### source

## set\_start\_value

JuMP.set\_start\_value - Function.

set\_start\_value(con\_ref::ConstraintRef, value)

Set the primal start value (MOI. ConstraintPrimalStart) of the constraint con\_ref to value.

To remove a primal start value set it to nothing.

See also start\_value.

```
julia> model = Model();
julia> @variable(model, x, start = 2.0);
julia> @constraint(model, c, [2x] in Nonnegatives())
c : [2 x] ∈ Nonnegatives()
julia> set_start_value(c, [4.0])
julia> start_value(c)
```

## CHAPTER 22. DOCSTRINGS

1-element Vector{Float64}:
 4.0

julia> set\_start\_value(c, nothing)

julia> start\_value(c)

#### source

set\_start\_value(variable::GenericVariableRef, value::Union{Real,Nothing})

Set the start value (MOI.VariablePrimalStart) of the variable to value.

Pass nothing to unset the start value.

Note: VariablePrimalStarts are sometimes called "MIP-starts" or "warmstarts".

See also: has\_start\_value, start\_value.

### Example

julia> model = Model();

julia> @variable(model, x, start = 1.5);

julia> @variable(model, y);

julia> has\_start\_value(x)
true

julia> has\_start\_value(y)
false

julia> start\_value(x)
1.5

julia> set\_start\_value(x, nothing)

julia> has\_start\_value(x)
false

julia> set\_start\_value(y, 2.0)

julia> has\_start\_value(y)
true

julia> start\_value(y)
2.0

#### source

### set\_start\_values

JuMP.set\_start\_values - Function.

```
set_start_values(
 model::GenericModel;
 variable_primal_start::Union{Nothing,Function} = value,
 constraint_primal_start::Union{Nothing,Function} = value,
 constraint_dual_start::Union{Nothing,Function} = dual,
 nonlinear_dual_start::Union{Nothing,Function} = nonlinear_dual_start_value,
)
```

Set the primal and dual starting values in model using the functions provided.

If any keyword argument is nothing, the corresponding start value is skipped.

If the optimizer does not support setting the starting value, the value will be skipped.

#### variable\_primal\_start

This function controls the primal starting solution for the variables. It is equivalent to calling set\_start\_value for each variable, or setting the MOI.VariablePrimalStart attribute.

If it is a function, it must have the form variable\_primal\_start(x::VariableRef) that maps each variable x to the starting primal value.

The default is value.

#### constraint\_primal\_start

This function controls the primal starting solution for the constraints. It is equivalent to calling set\_start\_value for each constraint, or setting the MOI.ConstraintPrimalStart attribute.

If it is a function, it must have the form constraint\_primal\_start(ci::ConstraintRef) that maps each constraint ci to the starting primal value.

The default is value.

#### constraint\_dual\_start

This function controls the dual starting solution for the constraints. It is equivalent to calling set\_dual\_start\_value for each constraint, or setting the MOI.ConstraintDualStart attribute.

If it is a function, it must have the form constraint\_dual\_start(ci::ConstraintRef) that maps each constraint ci to the starting dual value.

The default is dual.

#### nonlinear\_dual\_start

This function controls the dual starting solution for the nonlinear constraints It is equivalent to calling set\_nonlinear\_dual\_start\_value.

If it is a function, it must have the form nonlinear\_dual\_start(model::GenericModel) that returns a vector corresponding to the dual start of the constraints.

The default is nonlinear\_dual\_start\_value.

source

#### set\_string\_names\_on\_creation

JuMP.set\_string\_names\_on\_creation - Function.

set\_string\_names\_on\_creation(model::GenericModel, value::Bool)

Set the default argument of the set\_string\_name keyword in the @variable and @constraint macros to value.

The set\_string\_name keyword is used to determine whether to assign String names to all variables and constraints in model.

By default, value is true. However, for larger models calling set\_string\_names\_on\_creation(model, false) can improve performance at the cost of reducing the readability of printing and solver log messages.

#### Example

```
julia> import HiGHS
julia> model = Model(HiGHS.Optimizer);
julia> set_string_names_on_creation(model)
true
julia> set_string_names_on_creation(model, false)
julia> set_string_names_on_creation(model)
```

source

false

#### set\_time\_limit\_sec

JuMP.set\_time\_limit\_sec - Function.

set\_time\_limit\_sec(model::GenericModel, limit::Float64)

Set the time limit (in seconds) of the solver.

Can be unset using unset\_time\_limit\_sec or with limit set to nothing.

See also: unset\_time\_limit\_sec, time\_limit\_sec.

```
julia> import Ipopt
julia> model = Model(Ipopt.Optimizer);
julia> time_limit_sec(model)
julia> set_time_limit_sec(model, 60.0)
julia> time_limit_sec(model)
60.0
```

julia> unset\_time\_limit\_sec(model)

julia> time\_limit\_sec(model)

#### source

### set\_upper\_bound

JuMP.set\_upper\_bound - Function.

set\_upper\_bound(v::GenericVariableRef, upper::Number)

Set the upper bound of a variable. If one does not exist, create an upper bound constraint.

See also UpperBoundRef, has\_upper\_bound, upper\_bound, delete\_upper\_bound.

## Example

```
julia> model = Model();
julia> @variable(model, x <= 1.0);
julia> upper_bound(x)
1.0
julia> set_upper_bound(x, 2.0)
julia> upper_bound(x)
2.0
```

## source

## shadow\_price

JuMP.shadow\_price - Function.

shadow\_price(con\_ref::ConstraintRef)

Return the change in the objective from an infinitesimal relaxation of the constraint.

The shadow price is computed from dual and can be queried only when the objective\_sense is MIN\_SENSE or MAX\_SENSE (not FEASIBILITY\_SENSE).

See also reduced\_cost.

### Comparison to dual

The shadow prices differ at most in sign from the dual value depending on the objective\_sense. The differences are summarized in the table:

Notes

	MIN_SENSE	MAX_SENSE
f(x) <= b	dual(con_ref)	-dual(con_ref)
f(x) >= b	-dual(con_ref)	dual(con_ref)

This function simply translates signs from dual and does not validate the conditions needed to guarantee the sensitivity interpretation of the shadow price. The caller is responsible, for example, for checking whether the solver converged to an optimal primal-dual pair or a proof of infeasibility.

The relaxation of equality constraints (and hence the shadow price) is defined based on which sense of the equality constraint is active.

### Example

```
julia> import HiGHS
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x);
julia> @constraint(model, c, x <= 1)
c : x ≤ 1
julia> @objective(model, Max, 2 * x + 1);
julia> optimize!(model)
julia> dual_status(model)
FEASIBLE_POINT::ResultStatusCode = 1
julia> shadow_price(c)
2.0
julia> dual(c)
-2.0
```

### source

#### shape

JuMP.shape – Function.

shape(c::AbstractConstraint)::AbstractShape

Return the shape of the constraint c.

#### Example

julia> model = Model();

julia> @variable(model, x[1:2]);

julia> c = @constraint(model, x[2] <= 1);</pre>

julia> shape(constraint\_object(c))
ScalarShape()

```
julia> d = @constraint(model, x in SOS1());
```

julia> shape(constraint\_object(d))
VectorShape()

source

### show\_backend\_summary

JuMP.show\_backend\_summary - Function.

show\_backend\_summary(io:::I0, model::GenericModel)

Print a summary of the optimizer backing model.

#### Extensions

AbstractModels should implement this method.

#### Example

julia> model = Model();

```
julia> show_backend_summary(stdout, model)
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.
```

### source

#### show\_constraints\_summary

JuMP.show\_constraints\_summary - Function.

show\_constraints\_summary(io:::I0, model::AbstractModel)

Write to io a summary of the number of constraints.

#### Extensions

AbstractModels should implement this method.

julia> model = Model();

julia> @variable(model, x >= 0);

```
julia> show_constraints_summary(stdout, model)
`VariableRef`-in-`MathOptInterface.GreaterThan{Float64}`: 1 constraint
```

### source

### show\_objective\_function\_summary

```
JuMP.show_objective_function_summary - Function.
```

```
show_objective_function_summary(io:::I0, model::AbstractModel)
```

Write to io a summary of the objective function type.

#### Extensions

AbstractModels should implement this method.

#### Example

julia> model = Model();

```
julia> show_objective_function_summary(stdout, model)
Objective function type: AffExpr
```

### source

### simplex\_iterations

Example

JuMP.simplex\_iterations - Function.

simplex\_iterations(model::GenericModel)

If available, returns the cumulative number of simplex iterations during the most-recent optimization (the MOI.SimplexIterations attribute).

Throws a MOI.GetAttributeNotAllowed error if the attribute is not implemented by the solver.

julia> import HiGHS julia> model = Model(HiGHS.Optimizer); julia> set\_silent(model) julia> optimize!(model)

```
julia> simplex_iterations(model)
0
```

source

#### solution\_summary

JuMP.solution\_summary - Function.

```
solution summary(model::GenericModel; result::Int = 1, verbose::Bool = false)
```

Return a struct that can be used print a summary of the solution in result result.

If verbose=true, write out the primal solution for every variable and the dual solution for every constraint, excluding those with empty names.

#### Example

When called at the REPL, the summary is automatically printed:

```
julia> model = Model();
```

#### julia> solution\_summary(model)

```
solution_summary(; result = 1, verbose = false)
| solver_name : No optimizer attached.
| Termination
| | termination_status : OPTIMIZE_NOT_CALLED
| | result_count : 0
| L raw_status : optimize not called
L Solution (result = 1)
| primal_status : NO_SOLUTION
L dual_status : NO_SOLUTION
```

Use print to force the printing of the summary from inside a function:

```
julia> model = Model();
```

foo (generic function with 1 method)

### julia> foo(model)

```
solution_summary(; result = 1, verbose = false)
| solver_name : No optimizer attached.
| Termination
| | termination_status : OPTIMIZE_NOT_CALLED
| | result_count : 0
| L raw_status : optimize not called
L Solution (result = 1)
| primal_status : NO_SOLUTION
L dual_status : NO_SOLUTION
```

source

## solve\_time

JuMP.solve\_time - Function.

solve\_time(model::GenericModel)

If available, returns the solve time in wall-clock seconds reported by the solver (the MOI.SolveTimeSec attribute).

Throws a MOI.GetAttributeNotAllowed error if the attribute is not implemented by the solver.

#### Example

```
julia> import HiGHS
```

julia> model = Model(HiGHS.Optimizer);

julia> set\_silent(model)

julia> optimize!(model)

julia> solve\_time(model)
1.0488089174032211e-5

#### source

## solver\_name

JuMP.solver\_name - Function.

solver\_name(model::GenericModel)

If available, returns the MOI.SolverName property of the underlying optimizer.

Returns "No optimizer attached." in AUTOMATIC or MANUAL modes when no optimizer is attached.

Returns "SolverName() attribute not implemented by the optimizer." if the attribute is not implemented.

#### Example

```
julia> import Ipopt
julia> model = Model(Ipopt.Optimizer);
```

julia> solver\_name(model)
"Ipopt"

julia> model = Model();

## CHAPTER 22. DOCSTRINGS

```
julia> solver_name(model)
"No optimizer attached."
```

julia> model = Model(MOI.FileFormats.MPS.Model);

#### julia> solver\_name(model)

"SolverName() attribute not implemented by the optimizer."

#### source

## start\_value

JuMP.start\_value - Function.

start\_value(con\_ref::ConstraintRef)

Return the primal start value (MOI.ConstraintPrimalStart) of the constraint con\_ref.

If no primal start value has been set, start\_value will return nothing.

See also set\_start\_value.

#### Example

```
julia> model = Model();
```

julia> @variable(model, x, start = 2.0);

```
julia> @constraint(model, c, [2x] in Nonnegatives())
c : [2 x] ∈ Nonnegatives()
```

julia> set\_start\_value(c, [4.0])

```
julia> start_value(c)
1-element Vector{Float64}:
4.0
```

julia> set\_start\_value(c, nothing)

```
julia> start_value(c)
```

#### source

start\_value(v::GenericVariableRef)

Return the start value (MOI.VariablePrimalStart) of the variable v. Note: VariablePrimalStarts are sometimes called "MIP-starts" or "warmstarts".

See also: has\_start\_value, set\_start\_value.

```
julia> model = Model();
```

julia> @variable(model, x, start = 1.5);

julia> @variable(model, y);

julia> has\_start\_value(x)
true

julia> has\_start\_value(y)
false

julia> start\_value(x)
1.5

julia> set\_start\_value(y, 2.0)

julia> has\_start\_value(y)
true

julia> start\_value(y)
2.0

### source

### termination\_status

JuMP.termination\_status - Function.

termination\_status(model::GenericModel)

Return a MOI.TerminationStatusCode describing why the solver stopped (that is, the MOI.TerminationStatus attribute).

## Example

julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

```
julia> termination_status(model)
OPTIMIZE_NOT_CALLED::TerminationStatusCode = 0
```

#### source

## time\_limit\_sec

JuMP.time\_limit\_sec - Function.

time\_limit\_sec(model::GenericModel)

Return the time limit (in seconds) of the model.

Returns nothing if unset.

See also: set\_time\_limit\_sec, unset\_time\_limit\_sec.

### Example

```
julia> import Ipopt
```

julia> model = Model(Ipopt.Optimizer);

```
julia> time_limit_sec(model)
```

julia> set\_time\_limit\_sec(model, 60.0)

julia> time\_limit\_sec(model)
60.0

julia> unset\_time\_limit\_sec(model)

julia> time\_limit\_sec(model)

#### source

## triangle\_vec

JuMP.triangle\_vec - Function.

```
triangle_vec(matrix::Matrix)
```

Return the upper triangle of a matrix concatenated into a vector in the order required by JuMP and Math-OptInterface for Triangle sets.

## Example

```
julia> model = Model();
julia> @variable(model, X[1:3, 1:3], Symmetric);
julia> @variable(model, t)
t
julia> @constraint(model, [t; triangle_vec(X)] in MOI.RootDetConeTriangle(3))
[t, X[1,1], X[1,2], X[2,2], X[1,3], X[2,3], X[3,3]] ∈ MathOptInterface.RootDetConeTriangle(3)
```

### source

#### unfix

JuMP.unfix - Function.

unfix(v::GenericVariableRef)

Delete the fixing constraint of a variable.

Error if one does not exist.

See also FixRef, is\_fixed, fix\_value, fix.

### Example

julia> model = Model();

julia> @variable(model, x == 1);

julia> is\_fixed(x)
true

julia> unfix(x)

julia> is\_fixed(x)
false

#### source

#### unregister

JuMP.unregister - Function.

unregister(model::GenericModel, key::Symbol)

Unregister the name key from model so that a new variable, constraint, or expression can be created with the same key.

Note that this will not delete the object model[key]; it will just remove the reference at model[key]. To delete the object, use delete as well.

See also: delete, object\_dictionary.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, x)
ERROR: An object of name x is already attached to this model. If this
 is intended, consider using the anonymous construction syntax, for example,
 'x = @variable(model, [1:N], ...)` where the name of the object does
 not appear inside the macro.
 Alternatively, use `unregister(model, :x)` to first unregister
```

```
the existing name from the model. Note that this will not delete the
object; it will just remove the reference at `model[:x]`.
Stacktrace:
[...]
julia> num_variables(model)
1
julia> unregister(model, :x)
julia> @variable(model, x)
x
julia> num_variables(model)
2
```

#### source

### unsafe\_backend

JuMP.unsafe\_backend - Function.

unsafe\_backend(model::GenericModel)

Return the innermost optimizer associated with the JuMP model model.

This function should only be used by advanced users looking to access low-level solver-specific functionality. It has a high-risk of incorrect usage. We strongly suggest you use the alternative suggested below.

See also: backend.

To obtain the index of a variable or constraint in the unsafe backend, use optimizer\_index.

### **Unsafe behavior**

This function is unsafe for two main reasons.

First, the formulation and order of variables and constraints in the unsafe backend may be different to the variables and constraints in model. This can happen because of bridges, or because the solver requires the variables or constraints in a specific order. In addition, the variable or constraint index returned by index at the JuMP level may be different to the index of the corresponding variable or constraint in the unsafe\_backend. There is no solution to this. Use the alternative suggested below instead.

Second, the unsafe\_backend may be empty, or lack some modifications made to the JuMP model. Thus, before calling unsafe\_backend you should first call MOI.Utilities.attach\_optimizer to ensure that the backend is synchronized with the JuMP model.

```
julia> import HiGHS
```

julia> model = Model(HiGHS.Optimizer)
A JuMP Model
+ solver: HiGHS

## CHAPTER 22. DOCSTRINGS

```
+ objective_sense: FEASIBILITY_SENSE
+ num_variables: 0
+ num_constraints: 0
L Names registered in the model: none
```

julia> MOI.Utilities.attach\_optimizer(model)

julia> inner = unsafe\_backend(model)
A HiGHS model with 0 columns and 0 rows.

Moreover, if you modify the JuMP model, the reference you have to the backend (that is, inner in the example above) may be out-dated, and you should call MOI.Utilities.attach\_optimizer again.

This function is also unsafe in the reverse direction: if you modify the unsafe backend, for example, by adding a new constraint to inner, the changes may be silently discarded by JuMP when the JuMP model is modified or solved.

### Alternative

Instead of unsafe\_backend, create a model using direct\_model and call backend instead.

For example, instead of:

```
julia> import HiGHS
julia> model = Model(HiGHS.Optimizer);
julia> set_silent(model)
julia> @variable(model, x >= 0)
x
julia> MOI.Utilities.attach_optimizer(model)
A HiGHS model with 1 columns and 0 rows.
julia> optimizer_index(x)
MOI.VariableIndex(1)
```

```
julia> import HiGHS
julia> model = direct_model(HiGHS.Optimizer());
julia> set_silent(model)
julia> @variable(model, x >= 0)
x
julia> highs = backend(model) # No need to call `attach_optimizer`.
A HiGHS model with 1 columns and 0 rows.
```

julia> index(x)
MOI.VariableIndex(1)

source

## unset\_binary

JuMP.unset\_binary - Function.

unset\_binary(variable\_ref::GenericVariableRef)

Remove the binary constraint on the variable variable\_ref.

See also BinaryRef, is\_binary, set\_binary.

## Example

julia> model = Model();

julia> @variable(model, x, Bin);

**julia**> is\_binary(x) true

julia> unset\_binary(x)

```
julia> is_binary(x)
false
```

### source

## unset\_integer

JuMP.unset\_integer - Function.

```
unset_integer(variable_ref::GenericVariableRef)
```

Remove the integrality constraint on the variable variable\_ref.

Errors if one does not exist.

See also IntegerRef, is\_integer, set\_integer.

### Example

julia> model = Model();

julia> @variable(model, x, Int);

julia> is\_integer(x)
true

julia> unset\_integer(x)
julia> is\_integer(x)

false

### source

# unset\_silent

JuMP.unset\_silent - Function.

unset\_silent(model::GenericModel)

Neutralize the effect of the set\_silent function and let the solver attributes control the verbosity.

See also: set\_silent.

## Example

julia> import Ipopt julia> model = Model(Ipopt.Optimizer); julia> set\_silent(model) julia> get\_attribute(model, MOI.Silent()) true julia> unset\_silent(model)

julia> get\_attribute(model, MOI.Silent())
false

#### source

## unset\_time\_limit\_sec

JuMP.unset\_time\_limit\_sec - Function.

unset\_time\_limit\_sec(model::GenericModel)

Unset the time limit of the solver.

See also: set\_time\_limit\_sec, time\_limit\_sec.

julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);
julia> time\_limit\_sec(model)
julia> set\_time\_limit\_sec(model, 60.0)
julia> time\_limit\_sec(model)
60.0
julia> unset\_time\_limit\_sec(model)
julia> time\_limit\_sec(model)

#### source

# upper\_bound

JuMP.upper\_bound - Function.

upper\_bound(v::GenericVariableRef)

Return the upper bound of a variable.

Error if one does not exist.

See also UpperBoundRef, has\_upper\_bound, set\_upper\_bound, delete\_upper\_bound.

## Example

```
julia> model = Model();
```

julia> @variable(model, x <= 1.0);</pre>

julia> upper\_bound(x)
1.0

### source

### value

JuMP.value - Function.

value(con\_ref::ConstraintRef; result::Int = 1)

Return the primal value of constraint con\_ref associated with result index result of the most-recent solution returned by the solver.

That is, if con\_ref is the reference of a constraint func-in-set, it returns the value of func evaluated at the value of the variables (given by value(::GenericVariableRef)).

Use primal\_status to check if a result exists before asking for values.

See also: result\_count.

### Note

For scalar constraints, the constant is moved to the set so it is not taken into account in the primal value of the constraint. For instance, the constraint @constraint(model, 2x + 3y + 1 == 5) is transformed into 2x + 3y-in-MOI.EqualTo(4) so the value returned by this function is the evaluation of 2x + 3y.

source

value(var\_value::Function, con\_ref::ConstraintRef)

Evaluate the primal value of the constraint con\_ref using var\_value(v) as the value for each variable v.

source

value(v::GenericVariableRef; result = 1)

Return the value of variable v associated with result index result of the most-recent returned by the solver.

Use primal\_status to check if a result exists before asking for values.

See also: result\_count.

source

value(var\_value::Function, v::GenericVariableRef)

Evaluate the value of the variable v as var\_value(v).

source

value(var\_value::Function, ex::GenericAffExpr)

Evaluate ex using var\_value(v) as the value for each variable v.

source

value(v::GenericAffExpr; result::Int = 1)

Return the value of the GenericAffExpr v associated with result index result of the most-recent solution returned by the solver.

See also: result\_count.

source

value(var\_value::Function, ex::GenericQuadExpr)

Evaluate ex using  $var_value(v)$  as the value for each variable v.

source

```
value(v::GenericQuadExpr; result::Int = 1)
```

Return the value of the GenericQuadExpr v associated with result index result of the most-recent solution returned by the solver.

Replaces getvalue for most use cases.

See also: result\_count.

source

value(p::NonlinearParameter)

Return the current value stored in the nonlinear parameter p.

#### Example

julia> model = Model();

```
julia> @NLparameter(model, p == 10)
p == 10.0
```

```
julia> value(p)
10.0
```

#### source

value(ex::NonlinearExpression; result::Int = 1)

Return the value of the NonlinearExpression ex associated with result index result of the most-recent solution returned by the solver.

See also: result\_count.

source

value(var\_value::Function, ex::NonlinearExpression)

Evaluate ex using var\_value(v) as the value for each variable v.

source

value(c::NonlinearConstraintRef; result::Int = 1)

Return the value of the NonlinearConstraintRef c associated with result index result of the most-recent solution returned by the solver.

See also: result\_count.

source

value(var\_value::Function, c::NonlinearConstraintRef)

Evaluate c using var\_value(v) as the value for each variable v.

source

## value\_type

JuMP.value\_type - Function.

value\_type(::Type{<:Union{AbstractModel,AbstractVariableRef}})</pre>

Return the return type of value for variables of that model. It defaults to Float64 if it is not implemented.

### Example

julia> value\_type(GenericModel{BigFloat})
BigFloat

### source

## variable\_by\_name

JuMP.variable\_by\_name - Function.

```
variable_by_name(
 model::AbstractModel,
 name::String,
)::Union{AbstractVariableRef,Nothing}
```

Returns the reference of the variable with name attribute name or Nothing if no variable has this name attribute. Throws an error if several variables have name as their name attribute.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> variable_by_name(model, "x")
x
julia> @variable(model, base_name="x")
x
julia> variable_by_name(model, "x")
ERROR: Multiple variables have the name x.
Stacktrace:
[1] error(::String) at ./error.jl:33
```
```
[2] get(::MOIU.Model{Float64}, ::Type{MathOptInterface.VariableIndex}, ::String) at
→ /home/blegat/.julia/dev/MathOptInterface/src/Utilities/model.jl:222
 [3] get at /home/blegat/.julia/dev/MathOptInterface/src/Utilities/universalfallback.jl:201
\hookrightarrow [inlined]
[4]
\rightarrow get(::MathOptInterface.Utilities.CachingOptimizer{MathOptInterface.AbstractOptimizer,MathOptInterface.Utilitie
→ /home/blegat/.julia/dev/MathOptInterface/src/Utilities/cachingoptimizer.jl:490
[5] variable_by_name(::GenericModel, ::String) at
→ /home/blegat/.julia/dev/JuMP/src/variables.jl:268
[6] top-level scope at none:0
julia> var = @variable(model, base_name="y")
julia> variable_by_name(model, "y")
V
julia> set_name(var, "z")
julia> variable_by_name(model, "y")
julia> variable_by_name(model, "z")
Ζ
julia> @variable(model, u[1:2])
2-element Vector{VariableRef}:
 u[1]
 u[2]
julia> variable_by_name(model, "u[2]")
u[2]
```

### source

variable\_ref\_type

JuMP.variable\_ref\_type - Function.

```
variable_ref_type(:::Union{F,Type{F}}) where {F}
```

A helper function used internally by JuMP and some JuMP extensions. Returns the variable type associated with the model or expression type F.

```
source
```

# vectorize

JuMP.vectorize - Function.

```
vectorize(matrix::AbstractMatrix, ::Shape)
```

Convert the matrix into a vector according to Shape.

source

# write\_to\_file

JuMP.write\_to\_file - Function.

```
write_to_file(
 model::GenericModel,
 filename::String;
 format::MOI.FileFormats.FileFormat = MOI.FileFormats.FORMAT_AUTOMATIC,
 kwargs...,
)
```

Write the JuMP model model to filename in the format format.

See MOI.FileFormats.FileFormat for a list of supported formats.

### Compression

If the filename ends in .gz, the file will be compressed using GZip.

If the filename ends in .bz2, the file will be compressed using BZip2.

## **Keyword arguments**

Other kwargs are passed to the Model constructor of the chosen format.

For details, see the docstring each file format's Model constructor. For example, MOI.FileFormats.MPS.Model.

```
julia> model = Model();
julia> @variable(model, x >= 0);
julia> @objective(model, Min, 2 * x + 1);
julia> filename = joinpath(mktempdir(), "model.mps");
julia> write_to_file(model, filename; generic_names = true)
julia> print(read(filename, String))
NAME
ROWS
N OBJ
COLUMNS
 OBJ
 2
 C1
RHS
 rhs
 OBJ
 -1
RANGES
BOUNDS
 C1
 0
LO bounds
 PL bounds
 C1
ENDATA
```

### Solver-specific formats

write\_to\_file calls a Julia function from the MathOptInterface package that is independent of the choice
of solver. That is, it does not call a solver's internal API to write a model to disk.

For MPS files in particular, write\_to\_file may not support the full range of features that a solver's internal API supports. This is because some solvers have defined solver-specific extensions to the MPS format, whereas our Julia implementation supports only features which are standardized across multiple solvers.

To write a file to disk using the solver's internal API, use direct\_model and call the solver's C API. For example:

```
julia> import HiGHS
julia> model = direct_model(HiGHS.Optimizer());
julia> set_silent(model)
julia> @variable(model, x >= 0);
julia> @objective(model, Min, 2 * x + 1);
julia> filename = joinpath(mktempdir(), "model.mps");
julia> HiGHS.Highs_writeModel(backend(model), filename);
julia> print(read(filename, String))
NAME
ROWS
N Obj
COLUMNS
 2
 Х
 Obj
RHS
 RHS V
 -1
 Obj
ENDATA
```

### source

# AbstractConstraint

JuMP.AbstractConstraint - Type.

### abstract type AbstractConstraint

An abstract base type for all constraint types. AbstractConstraints store the function and set directly, unlike ConstraintRefs that are merely references to constraints stored in a model. AbstractConstraints do not need to be attached to a model.

source

### AbstractJuMPScalar

JuMP.AbstractJuMPScalar - Type.

AbstractJuMPScalar <: MutableArithmetics.AbstractMutable</pre>

## Abstract base type for all scalar types

The subtyping of AbstractMutable will allow calls of some Base functions to be redirected to a method in MA that handles type promotion more carefully (for example the promotion in sparse matrix products in SparseArrays usually does not work for JuMP types) and exploits the mutability of AffExpr and QuadExpr.

source

## AbstractModel

JuMP.AbstractModel - Type.

AbstractModel

An abstract type that should be subtyped for users creating JuMP extensions.

source

### AbstractScalarSet

JuMP.AbstractScalarSet - Type.

AbstractScalarSet

An abstract type for defining new scalar sets in JuMP.

Implement moi\_set(::AbstractScalarSet) to convert the type into an MOI set.

See also: moi\_set.

source

### AbstractShape

JuMP.AbstractShape - Type.

AbstractShape

Abstract vectorizable shape. Given a flat vector form of an object of shape shape, the original object can be obtained by reshape\_vector.

source

### AbstractVariable

JuMP.AbstractVariable - Type.

### AbstractVariable

Variable returned by build\_variable. It represents a variable that has not been added yet to any model.
It can be added to a given model with add\_variable.

source

# AbstractVariableRef

JuMP.AbstractVariableRef - Type.

AbstractVariableRef

Variable returned by add\_variable. Affine (resp. quadratic) operations with variables of type V<:AbstractVariableRef and coefficients of type T create a GenericAffExpr{T,V} (resp. GenericQuadExpr{T,V}).

### source

### AbstractVectorSet

JuMP.AbstractVectorSet - Type.

AbstractVectorSet

An abstract type for defining new sets in JuMP.

Implement moi\_set(::AbstractVectorSet, dim::Int) to convert the type into an MOI set.

See also: moi\_set.

source

# AffExpr

JuMP.AffExpr - Type.

AffExpr

Alias for GenericAffExpr{Float64, VariableRef}, the specific GenericAffExpr used by JuMP.

source

## ArrayShape

JuMP.ArrayShape - Type.

ArrayShape{N}(dims::NTuple{N,Int}) where {N}

An AbstractShape that represents array-valued constraints.

# Example

```
julia> model = Model();
julia> @variable(model, x[1:2, 1:3]);
julia> c = @constraint(model, x >= 0, Nonnegatives())
[x[1,1] x[1,2] x[1,3]
x[2,1] x[2,2] x[2,3]] ∈ Nonnegatives()
julia> shape(constraint_object(c))
ArrayShape{2}((2, 3))
```

source

## **BinaryRef**

JuMP.BinaryRef - Function.

BinaryRef(v::GenericVariableRef)

Return a constraint reference to the constraint constraining v to be binary. Errors if one does not exist.

See also is\_binary, set\_binary, unset\_binary.

# Example

julia> model = Model();

julia> @variable(model, x, Bin);

julia> BinaryRef(x)
x binary

### source

# BridgeableConstraint

JuMP.BridgeableConstraint - Type.

```
BridgeableConstraint(
 constraint::C,
 bridge_type::B;
 coefficient_type::Type{T} = Float64,
) where {C<:AbstractConstraint,B<:Type{<:MOI.Bridges.AbstractBridge},T}</pre>
```

An AbstractConstraint representing that constraint that can be bridged by the bridge of type bridge\_type{coefficient Adding a BridgeableConstraint to a model is equivalent to:

```
add_bridge(model, bridge_type; coefficient_type = coefficient_type)
add_constraint(model, constraint)
```

## Example

Given a new scalar set type CustomSet with a bridge CustomBridge that can bridge F-in-CustomSet constraints, when the user does:

```
model = Model()
@variable(model, x)
@constraint(model, x + 1 in CustomSet())
optimize!(model)
```

with an optimizer that does not support F-in-CustomSet constraints, the constraint will not be bridged unless they first call add\_bridge(model, CustomBridge).

In order to automatically add the CustomBridge to any model to which an F-in-CustomSet is added, add the following method:

```
function JuMP.build_constraint(
 error_fn::Function,
 func::AbstractJuMPScalar,
 set::CustomSet,
)
 constraint = ScalarConstraint(func, set)
 return BridgeableConstraint(constraint, CustomBridge)
end
```

### Note

JuMP extensions should extend JuMP.build\_constraint only if they also defined CustomSet, for three reasons:

- 1. It is problematic if multiple extensions overload the same JuMP method.
- A missing method will not inform the users that they forgot to load the extension module defining the build\_constraint method.
- 3. Defining a method where neither the function nor any of the argument types are defined in the package is called *type piracy* and is discouraged in the Julia style guide.

### source

# ComplexPlane

JuMP.ComplexPlane - Type.

ComplexPlane

Complex plane object that can be used to create a complex variable in the @variable macro.

Consider the following example:

```
julia> model = Model();
julia> @variable(model, x in ComplexPlane())
real(x) + imag(x) im
julia> all_variables(model)
2-element Vector{VariableRef}:
real(x)
imag(x)
```

We see in the output of the last command that two real variables were created. The Julia variable x binds to an affine expression in terms of these two variables that parametrize the complex plane.

source

### ComplexVariable

JuMP.ComplexVariable - Type.

ComplexVariable{S,T,U,V} <: AbstractVariable</pre>

A struct used when adding complex variables.

See also: ComplexPlane.

source

# ConstraintNotOwned

JuMP.ConstraintNotOwned - Type.

```
struct ConstraintNotOwned{C<:ConstraintRef} <: Exception
 constraint_ref::C
end</pre>
```

An error thrown when the constraint constraint\_ref was used in a model different to owner\_model (constraint\_ref).

Stacktrace:
[...]

source

# ConstraintRef

JuMP.ConstraintRef - Type.

ConstraintRef

Holds a reference to the model and the corresponding MOI.ConstraintIndex.

source

# FixRef

JuMP.FixRef - Function.

FixRef(v::GenericVariableRef)

Return a constraint reference to the constraint fixing the value of v.

Errors if one does not exist.

See also is\_fixed, fix\_value, fix, unfix.

### Example

```
julia> model = Model();
julia> @variable(model, x == 1);
julia> FixRef(x)
x = 1
```

### source

# GenericAffExpr

JuMP.GenericAffExpr - Type.

```
mutable struct GenericAffExpr{CoefType,VarType} <: AbstractJuMPScalar
 constant::CoefType
 terms::OrderedDict{VarType,CoefType}
end
```

An expression type representing an affine expression of the form:  $\sum a_i x_i + c$ .

# Fields

- .constant: the constant c in the expression.
- .terms: an OrderedDict, with keys of VarType and values of CoefType describing the sparse vector a.

# Example

```
x[1] => 3.0
```

# source

### GenericModel

```
JuMP.GenericModel - Type.
```

```
GenericModel{T}([optimizer_factory]; kwargs...) where {T<:Real}</pre>
```

Create a new instance of a JuMP model.

If optimizer\_factory is provided, the model is initialized with the optimizer using set\_optimizer(model, optimizer\_factory; kwargs...). See set\_optimizer for details on kwargs.

If optimizer\_factory is not provided, use set\_optimizer to set the optimizer before calling optimize!.

### Value type T

Passing a type other than Float64 as the value type T is an advanced operation. The value type must match that expected by the chosen optimizer. Consult the optimizers documentation for details.

If not documented, assume that the optimizer supports only Float64.

Choosing an unsupported value type will throw an MOI.UnsupportedConstraint or an MOI.UnsupportedAttribute error, the timing of which (during the model construction or during a call to optimize!) depends on how the solver is interfaced to JuMP.

# Example

julia> model = GenericModel{BigFloat}();

julia> typeof(model)
GenericModel{BigFloat}

### source

## GenericNonlinearExpr

JuMP.GenericNonlinearExpr - Type.

```
GenericNonlinearExpr{V}(head::Symbol, args::Vector{Any})
GenericNonlinearExpr{V}(head::Symbol, args::Any...)
```

The scalar-valued nonlinear function head(args...), represented as a symbolic expression tree, with the call operator head and ordered arguments in args.

V is the type of AbstractVariableRef present in the expression, and is used to help dispatch JuMP extensions.

## head

The head::Symbol must be an operator supported by the model.

The default list of supported univariate operators is given by:

MOI.Nonlinear.DEFAULT\_UNIVARIATE\_OPERATORS

and the default list of supported multivariate operators is given by:

MOI.Nonlinear.DEFAULT\_MULTIVARIATE\_OPERATORS

Additional operators can be add using @operator.

See the full list of operators supported by a MOI.ModelLike by querying the MOI.ListOfSupportedNonlinearOperators attribute.

### args

The vector args contains the arguments to the nonlinear function. If the operator is univariate, it must contain one element. Otherwise, it may contain multiple elements.

Given a subtype of AbstractVariableRef, V, for GenericNonlinearExpr{V}, each element must be one of the following:

- A constant value of type <: Real
- A V
- A GenericAffExpr{T,V}
- A GenericQuadExpr{T,V}
- A GenericNonlinearExpr{V}

where T<:Real and T == value\_type(V).

### **Unsupported operators**

If the optimizer does not support head, an MOI.UnsupportedNonlinearOperator error will be thrown.

There is no guarantee about when this error will be thrown; it may be thrown when the function is first added to the model, or it may be thrown when optimize! is called.

### Example

To represent the function  $f(x) = sin(x)^2$ , do:

source

### **GenericQuadExpr**

JuMP.GenericQuadExpr - Type.

```
mutable struct GenericQuadExpr{CoefType,VarType} <: AbstractJuMPScalar
aff::GenericAffExpr{CoefType,VarType}
terms::OrderedDict{UnorderedPair{VarType}, CoefType}
end
```

An expression type representing an quadratic expression of the form:  $\sum q_{i,j}x_ix_j + \sum a_ix_i + c$ .

## Fields

- .aff: an GenericAffExpr representing the affine portion of the expression.
- .terms: an OrderedDict, with keys of UnorderedPair{VarType} and values of CoefType, describing the sparse list of terms q.

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> expr = 2.0 * x[1]^2 + x[1] * x[2] + 3.0 * x[1] + 4.0
2 x[1]² + x[1]*x[2] + 3 x[1] + 4
julia> expr.aff
3 x[1] + 4
julia> expr.terms
OrderedCollections.OrderedDict{UnorderedPair{VariableRef}, Float64} with 2 entries:
UnorderedPair{VariableRef}(x[1], x[1]) => 2.0
UnorderedPair{VariableRef}(x[1], x[2]) => 1.0
```

## **GenericReferenceMap**

JuMP.GenericReferenceMap - Type.

### GenericReferenceMap{T}

Mapping between variable and constraint reference of a model and its copy. The reference of the copied model can be obtained by indexing the map with the reference of the corresponding reference of the original model.

source

# GenericVariableRef

JuMP.GenericVariableRef - Type.

GenericVariableRef{T} <: AbstractVariableRef</pre>

Holds a reference to the model and the corresponding MOI.VariableIndex.

source

# GreaterThanZero

JuMP.GreaterThanZero - Type.

GreaterThanZero()

A struct used to intercept when  $\geq$  or  $\geq$  is used in a macro via operator\_to\_set.

This struct is not the same as Nonnegatives so that we can disambiguate  $x \ge y$  and  $x \ge y$  in Nonnegatives().

This struct is not intended for general usage, but it may be useful to some JuMP extensions.

# Example

```
julia> operator_to_set(error, Val(:>=))
GreaterThanZero()
```

### source

## HermitianMatrixAdjointShape

JuMP.HermitianMatrixAdjointShape - Type.

HermitianMatrixAdjointShape(side\_dimension)

The dual\_shape of HermitianMatrixShape. This shape is not intended for regular use.

source

### **HermitianMatrixShape**

JuMP.HermitianMatrixShape - Type.

```
HermitianMatrixShape(
 side_dimension::Int;
 needs_adjoint_dual::Bool = false,
)
```

The shape object for a Hermitian square matrix of side\_dimension rows and columns.

The vectorized form corresponds to MOI.HermitianPositiveSemidefiniteConeTriangle.

## needs\_adjoint\_dual

By default, the dual\_shape of HermitianMatrixShape is also HermitianMatrixShape. This is true for cases such as a LinearAlgebra.Hermitian matrix in HermitianPSDCone.

However, JuMP also supports LinearAlgebra.Hermitian matrix in Zeros, which is interpreted as an elementwise equality constraint. By exploiting symmetry, we pass only the upper triangle of the equality constraints. This works for the primal, but it leads to a factor of 2 difference in the off-diagonal dual elements. (The dual value of the (i, j) element in the triangle formulation should be divided by 2 when spread across the (i, j) and (j, i) elements in the square matrix formulation.) If the constraint has this dual inconsistency, set needs\_adjoint\_dual = true.

source

### HermitianMatrixSpace

JuMP.HermitianMatrixSpace - Type.

HermitianMatrixSpace()

Use in the @variable macro to constrain a matrix of variables to be hermitian.

### Example

```
julia> model = Model();
```

### source

### HermitianPSDCone

JuMP.HermitianPSDCone - Type.

### HermitianPSDCone

Hermitian positive semidefinite cone object that can be used to create a Hermitian positive semidefinite square matrix in the @variable and @constraint macros.

### Example

Consider the following example:

```
julia> model = Model();
```

julia> c = constraint\_object(VariableInSetRef(H));

### julia> c.func

9-element Vector{VariableRef}:
 real(H[1,1])
 real(H[1,2])
 real(H[2,2])
 real(H[1,3])
 real(H[2,3])
 real(H[3,3])
 imag(H[1,2])
 imag(H[1,3])

### julia> c.set

MathOptInterface.HermitianPositiveSemidefiniteConeTriangle(3)

We see in the output of the last commands that 9 real variables were created. The matrix H constrains affine expressions in terms of these 9 variables that parametrize a Hermitian matrix.

## source

## IntegerRef

JuMP.IntegerRef - Function.

IntegerRef(v::GenericVariableRef)

Return a constraint reference to the constraint constraining v to be integer.

Errors if one does not exist.

See also is\_integer, set\_integer, unset\_integer.

julia> model = Model();

julia> @variable(model, x, Int);

julia> IntegerRef(x)
x integer

source

# LPMatrixData

JuMP.LPMatrixData - Type.

LPMatrixData{T}

The struct returned by lp\_matrix\_data. See lp\_matrix\_data for a description of the public fields.

source

# LessThanZero

JuMP.LessThanZero - Type.

GreaterThanZero()

A struct used to intercept when  $\leq$  or  $\leq$  is used in a macro via operator\_to\_set.

This struct is not the same as Nonpositives so that we can disambiguate  $x \le y$  and  $x \ge y$  in Nonpositives().

This struct is not intended for general usage, but it may be useful to some JuMP extensions.

# Example

```
julia> operator_to_set(error, Val(:<=))
LessThanZero()</pre>
```

### source

# LinearTermIterator

JuMP.LinearTermIterator - Type.

LinearTermIterator{GAE<:GenericAffExpr}</pre>

A struct that implements the iterate protocol in order to iterate over tuples of (coefficient, variable) in the GenericAffExpr.

source

# LowerBoundRef

JuMP.LowerBoundRef - Function.

```
LowerBoundRef(v::GenericVariableRef)
```

Return a constraint reference to the lower bound constraint of v.

Errors if one does not exist.

See also has\_lower\_bound, lower\_bound, set\_lower\_bound, delete\_lower\_bound.

# Example

```
julia> model = Model();
```

```
julia> @variable(model, x >= 1.0);
```

```
julia> LowerBoundRef(x)
x \ge 1
```

### source

### Model

JuMP.Model - Type.

Model([optimizer\_factory;] add\_bridges::Bool = true)

Create a new instance of a JuMP model.

If optimizer\_factory is provided, the model is initialized with thhe optimizer returned by MOI.instantiate(optimizer\_fact

If optimizer\_factory is not provided, use set\_optimizer to set the optimizer before calling optimize!.

If add\_bridges, JuMP adds a MOI.Bridges.LazyBridgeOptimizer to automatically reformulate the problem into a form supported by the optimizer.

```
julia> import Ipopt
julia> model = Model(Ipopt.Optimizer);
julia> solver_name(model)
"Ipopt"
julia> import HiGHS
julia> import MultiObjectiveAlgorithms as MOA
julia> model = Model(() -> MOA.Optimizer(HiGHS.Optimizer); add_bridges = false);
```

# ModelMode

JuMP.ModelMode - Type.

ModelMode

An enum to describe the state of the CachingOptimizer inside a JuMP model.

See also: mode.

# Values

Possible values are:

AUTOMATIC : moi\_backend field holds a CachingOptimizer in AUTOMATIC mode.

- MANUAL : moi\_backend field holds a CachingOptimizer in MANUAL mode.
- DIRECT : moi\_backend field holds an AbstractOptimizer. No extra copy of the model is stored. The moi\_backend must support add\_constraint etc.

### source

# NLPEvaluator

JuMP.NLPEvaluator - Function.

```
NLPEvaluator(
 model::Model,
 __differentiation_backend::MOI.Nonlinear.AbstractAutomaticDifferentiation =
 MOI.Nonlinear.SparseReverseMode(),
)
```

Return an MOI.AbstractNLPEvaluator constructed from model

Warning

Before using, you must initialize the evaluator using MOI.initialize.

### Experimental

These features may change or be removed in any future version of JuMP.

Pass \_differentiation\_backend to specify the differentiation backend used to compute derivatives.

source

# NoOptimizer

JuMP.NoOptimizer - Type.

struct NoOptimizer <: Exception end</pre>

An error thrown when no optimizer is set and one is required.

The optimizer can be provided to the Model constructor or by calling set\_optimizer.

# Example

```
julia> model = Model();
julia> optimize!(model)
ERROR: NoOptimizer()
Stacktrace:
[...]
```

source

### NonlinearExpr

JuMP.NonlinearExpr - Type.

NonlinearExpr

Alias for GenericNonlinearExpr{VariableRef}, the specific GenericNonlinearExpr used by JuMP.

source

# NonlinearOperator

JuMP.NonlinearOperator - Type.

NonlinearOperator(func::Function, head::Symbol)

A callable struct (functor) representing a function named head.

When called with AbstractJuMPScalars, the struct returns a GenericNonlinearExpr.

When called with non-JuMP types, the struct returns the evaluation of func(args...).

Unless head is special-cased by the optimizer, the operator must have already been added to the model using add\_nonlinear\_operator or @operator.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> f(x::Float64) = x^2
f (generic function with 1 method)
julia> \frac{\frac{1}{1}}{f(x::Float64)} = 2 * x
\frac{\frac{1}{2}}{f(generic function with 1 method)}
```

```
julia> \nabla^2 f(x::Float64) = 2.0
\nabla^2 f (generic function with 1 method)
```

**julia**> @operator(model, op\_f, 1, f,  $\nabla f$ ,  $\nabla^2 f$ ) NonlinearOperator(f, :op\_f)

julia> bar = NonlinearOperator(f, :op\_f) NonlinearOperator(f, :op\_f)

julia> @objective(model, Min, bar(x))
op\_f(x)

**julia**> bar(2.0) 4.0

### source

# Nonnegatives

JuMP.Nonnegatives - Type.

Nonnegatives()

The JuMP equivalent of the MOI.Nonnegatives set, in which the dimension is inferred from the corresponding function.

# Example

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

julia> @constraint(model, x in Nonnegatives())
[x[1], x[2]] ∈ Nonnegatives()

```
julia> A = [1 2; 3 4];
```

```
julia> b = [5, 6];
```

julia> @constraint(model, A \* x >= b)
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ Nonnegatives()

### source

### Nonpositives

JuMP.Nonpositives - Type.

## Nonpositives()

The JuMP equivalent of the MOI. Nonpositives set, in which the dimension is inferred from the corresponding function.

## Example

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

```
julia> @constraint(model, x in Nonpositives())
[x[1], x[2]] ∈ Nonpositives()
```

```
julia> A = [1 2; 3 4];
```

julia> b = [5, 6];

julia> @constraint(model, A \* x <= b)
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ Nonpositives()</pre>

### source

# **OptimizationSense**

JuMP.OptimizationSense - Type.

**OptimizationSense** 

An Enum for the value of the ObjectiveSense attribute.

### Values

# MIN\_SENSE

The goal is to minimize the objective function.

# MAX\_SENSE

The goal is to maximize the objective function.

## FEASIBILITY\_SENSE

The model does not have an objective function.

source

# **OptimizeNotCalled**

JuMP.OptimizeNotCalled - Type.

## struct OptimizeNotCalled <: Exception end</pre>

An error thrown when a result attribute cannot be queried before optimize! is called.

# julia> import Ipopt julia> model = Model(Ipopt.Optimizer); julia> objective\_value(model) ERROR: OptimizeNotCalled() Stacktrace: [...]

### source

Example

### PSDCone

JuMP.PSDCone - Type.

PSDCone

Positive semidefinite cone object that can be used to constrain a square matrix to be positive semidefinite in the @constraint macro.

If the matrix has type Symmetric then the columns vectorization (the vector obtained by concatenating the columns) of its upper triangular part is constrained to belong to the MOI.PositiveSemidefiniteConeTriangle set, otherwise its column vectorization is constrained to belong to the MOI.PositiveSemidefiniteConeSquare set.

### Example

Non-symmetric case:

```
julia> model = Model();
```

```
julia> @variable(model, x);
```

```
julia> a = [x 2x; 2x x];
```

julia> b = [1 2; 2 4];

julia> cref = @constraint(model, a >= b, PSDCone())
[x - 1 2 x - 2
 2 x - 2 x - 4] ∈ PSDCone()

julia> jump\_function(constraint\_object(cref))

4-element Vector{AffExpr}: x - 1 2 x - 2 2 x - 2 x - 4

```
julia> moi_set(constraint_object(cref))
MathOptInterface.PositiveSemidefiniteConeSquare(2)
```

### Symmetric case:

```
julia> using LinearAlgebra # For Symmetric

julia> model = Model();

julia> @variable(model, x);

julia> a = [x 2x; 2x x];

julia> b = [1 2; 2 4];

julia> cref = @constraint(model, Symmetric(a - b) in PSDCone())

[x - 1 2 x - 2

... x - 4] E PSDCone()

julia> jump_function(constraint_object(cref))

3-element Vector{AffExpr}:

x - 1

2 x - 2

x - 4
```

julia> moi\_set(constraint\_object(cref))
MathOptInterface.PositiveSemidefiniteConeTriangle(2)

# source

### Parameter

JuMP.Parameter - Type.

Parameter(value)

A short-cut for the MOI. Parameter set.

# Example

julia> model = Model();

julia> @variable(model, x in Parameter(2))

```
julia> print(model)
Feasibility
Subject to
x ∈ MathOptInterface.Parameter{Float64}(2.0)
```

source

# ParameterRef

JuMP.ParameterRef - Function.

ParameterRef(x::GenericVariableRef)

Return a constraint reference to the constraint constraining x to be a parameter.

Errors if one does not exist.

See also is\_parameter, set\_parameter\_value, parameter\_value.

### Example

р

julia> model = Model();

```
julia> @variable(model, p in Parameter(2))
```

julia> @variable(model, x);

```
julia> ParameterRef(x)
ERROR: Variable x is not a parameter.
Stacktrace:
[...]
```

### source

# QuadExpr

JuMP.QuadExpr - Type.

QuadExpr

An alias for GenericQuadExpr{Float64, VariableRef}, the specific GenericQuadExpr used by JuMP.

source

# QuadTermIterator

JuMP.QuadTermIterator - Type.

QuadTermIterator{GQE<:GenericQuadExpr}</pre>

A struct that implements the iterate protocol in order to iterate over tuples of (coefficient, variable, variable) in the GenericQuadExpr.

source

## ReferenceMap

JuMP.ReferenceMap - Type.

### GenericReferenceMap{T}

Mapping between variable and constraint reference of a model and its copy. The reference of the copied model can be obtained by indexing the map with the reference of the corresponding reference of the original model.

source

# ResultStatusCode

JuMP.ResultStatusCode - Type.

ResultStatusCode

An Enum of possible values for the PrimalStatus and DualStatus attributes.

The values indicate how to interpret the result vector.

### Values

**NO SOLUTION** 

The result vector is empty.

FEASIBLE POINT

The result vector is a feasible point.

NEARLY\_FEASIBLE\_POINT

The result vector is feasible if some constraint tolerances are relaxed.

INFEASIBLE\_POINT

The result vector is an infeasible point.

**INFEASIBILITY CERTIFICATE** 

The result vector is an infeasibility certificate.

If the PrimalStatus is INFEASIBILITY\_CERTIFICATE, then the primal result vector is a certificate of dual infeasibility.

If the DualStatus is INFEASIBILITY\_CERTIFICATE, then the dual result vector is a proof of primal infeasibility.

NEARLY\_INFEASIBILITY\_CERTIFICATE

The result satisfies a relaxed criterion for a certificate of infeasibility.

If the PrimalStatus is NEARLY\_INFEASIBILITY\_CERTIFICATE, then the primal result vector is a certificate of dual infeasibility.

If the DualStatus is NEARLY\_INFEASIBILITY\_CERTIFICATE, then the dual result vector is a proof of primal infeasibility.

# **REDUCTION\_CERTIFICATE**

The result vector is an ill-posed certificate; see this article for details.

If the PrimalStatus is REDUCTION\_CERTIFICATE, then the primal result vector is a proof that the dual problem is ill-posed.

If the DualStatus is REDUCTION\_CERTIFICATE, then the dual result vector is a proof that the primal is ill-posed.

# NEARLY\_REDUCTION\_CERTIFICATE

The result satisfies a relaxed criterion for an ill-posed certificate.

UNKNOWN\_RESULT\_STATUS

The result vector contains a solution with an unknown interpretation. Check the solver log for more details.

### **OTHER\_RESULT\_STATUS**

The result vector contains a solution with an interpretation not covered by one of the statuses defined above. Check the solver log for more details.

source

# RotatedSecondOrderCone

JuMP.RotatedSecondOrderCone - Type.

RotatedSecondOrderCone

Rotated second order cone object that can be used to constrain the square of the euclidean norm of a vector x to be less than or equal to 2tu where t and u are nonnegative scalars.

This is a shortcut for the MOI.RotatedSecondOrderCone set.

## Example

The following constrains  $||(x-1, x-2)||_2^2 \le 2tx$  and  $t, x \ge 0$ :

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, t)
t
julia> @constraint(model, [t, x, x-1, x-2] in RotatedSecondOrderCone())
```

```
[t, x, x - 1, x - 2] E MathOptInterface.RotatedSecondOrderCone(4)
```

# source

# S0S1

JuMP.SOS1 - Type.

```
S0S1(weights = Real[])
```

The SOS1 (Special Ordered Set of Type 1) set constrains a vector x to the set where at most one variable can take a non-zero value, and all other elements are zero.

The weights vector, if specified, induces an ordering of the variables; as such, it should contain unique values. The weights vector must have the same number of elements as the vector x, and the element weights[i] corresponds to element x[i]. If not provided, the weights vector defaults to weights[i] = i.

This is a shortcut for the MOI.SOS1 set.

# Example

```
julia> model = Model();
julia> @variable(model, x[1:3] in SOS1([4.1, 3.2, 5.0]))
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
julia> print(model)
Feasibility
Subject to
 [x[1], x[2], x[3]] ∈ MathOptInterface.SOS1{Float64}([4.1, 3.2, 5.0])
```

### source

## S0S2

JuMP.SOS2 - Type.

S0S2(weights = Real[])

The SOS2 (Special Ordered Set of Type 2) set constrains a vector x to the set where at most two variables can take a non-zero value, and all other elements are zero. In addition, the two non-zero values must be consecutive given the ordering of the x vector induced by weights.

The weights vector, if specified, induces an ordering of the variables; as such, it must contain unique values. The weights vector must have the same number of elements as the vector x, and the element weights[i] corresponds to element x[i]. If not provided, the weights vector defaults to weights[i] = i.

This is a shortcut for the MOI.SOS2 set.

```
julia> model = Model();
```

```
julia> @variable(model, x[1:3] in SOS2([4.1, 3.2, 5.0]))
3-element Vector{VariableRef}:
```

x[1] x[2] x[3]

# **julia> print(model)** Feasibility

```
Subject to
[x[1], x[2], x[3]] ∈ MathOptInterface.S0S2{Float64}([4.1, 3.2, 5.0])
```

# source

# ScalarConstraint

JuMP.ScalarConstraint - Type.

### struct ScalarConstraint

The data for a scalar constraint.

See also the documentation on JuMP's representation of constraints for more background.

# Fields

- . func: field contains a JuMP object representing the function
- .set: field contains the MOI set

# Example

A scalar constraint:

```
julia> model = Model();
julia> @variable(model, x);
julia> @constraint(model, c, 2x <= 1)
c : 2 x ≤ 1
julia> object = constraint_object(c)
ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(2 x,
→ MathOptInterface.LessThan{Float64}(1.0))
julia> typeof(object)
```

ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}

```
julia> object.func
2 x
```

```
julia> object.set
MathOptInterface.LessThan{Float64}(1.0)
```

# ScalarShape

JuMP.ScalarShape - Type.

ScalarShape()

An AbstractShape that represents scalar constraints.

# Example

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> c = @constraint(model, x[2] <= 1);
julia> shape(constraint_object(c))
```

ScalarShape()

## source

# ScalarVariable

JuMP.ScalarVariable - Type.

ScalarVariable{S,T,U,V} <: AbstractVariable</pre>

A struct used when adding variables.

See also: add\_variable.

source

# SecondOrderCone

JuMP.SecondOrderCone - Type.

SecondOrderCone

Second order cone object that can be used to constrain the euclidean norm of a vector x to be less than or equal to a nonnegative scalar t.

This is a shortcut for the MOI.SecondOrderCone set.

# Example

The following constrains  $||(x-1, x-2)||_2 \le t$  and  $t \ge 0$ :

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, t)
t
julia> @constraint(model, [t, x-1, x-2] in SecondOrderCone())
[t, x - 1, x - 2] ∈ MathOptInterface.SecondOrderCone(3)
```

source

## Semicontinuous

JuMP.Semicontinuous - Type.

Semicontinuous(lower, upper)

A short-cut for the MOI. Semicontinuous set.

This short-cut is useful because it automatically promotes lower and upper to the same type, and converts them into the element type supported by the JuMP model.

# Example

### source

## Semiinteger

JuMP.Semiinteger - Type.

Semiinteger(lower, upper)

A short-cut for the MOI.Semiinteger set.

This short-cut is useful because it automatically promotes lower and upper to the same type, and converts them into the element type supported by the JuMP model.

# CHAPTER 22. DOCSTRINGS

julia> model = Model();

julia> @variable(model, x in Semiinteger(3, 5))
x
julia> print(model)
Feasibility
Subject to
x ∈ MathOptInterface.Semiinteger{Int64}(3, 5)

source

# SensitivityReport

JuMP.SensitivityReport - Type.

SensitivityReport

See lp\_sensitivity\_report.

source

### SkewSymmetricMatrixShape

JuMP.SkewSymmetricMatrixShape - Type.

SkewSymmetricMatrixShape

Shape object for a skew symmetric square matrix of side\_dimension rows and columns. The vectorized form contains the entries of the upper-right triangular part of the matrix (without the diagonal) given column by column (or equivalently, the entries of the lower-left triangular part given row by row). The diagonal is zero.

source

# **SkewSymmetricMatrixSpace**

JuMP.SkewSymmetricMatrixSpace - Type.

SkewSymmetricMatrixSpace()

Use in the @variable macro to constrain a matrix of variables to be skew-symmetric.

Example

-Q[1,2] 0

```
julia> model = Model();
julia> @variable(model, Q[1:2, 1:2] in SkewSymmetricMatrixSpace())
2×2 Matrix{AffExpr}:
0 Q[1,2]
```

### source

### SkipModelConvertScalarSetWrapper

JuMP.SkipModelConvertScalarSetWrapper - Type.

SkipModelConvertScalarSetWrapper(set::MOI.AbstractScalarSet)

JuMP uses model\_convert to automatically promote MOI.AbstractScalarSet sets to the same value\_type as the model.

In cases there this is undesirable, wrap the set in SkipModelConvertScalarSetWrapper to pass the set un-changed to the solver.

#### Warning

This struct is intended for use internally by JuMP extensions. You should not need to use it in regular JuMP code.

## Example

```
julia> model = Model();
```

julia> @variable(model, x);

```
julia> @constraint(model, x in MOI.EqualTo(1 // 2))
x = 0.5
```

```
julia> @constraint(model, x in SkipModelConvertScalarSetWrapper(MOI.EqualTo(1 // 2)))
x = 1//2
```

### source

# SquareMatrixShape

JuMP.SquareMatrixShape - Type.

SquareMatrixShape

Shape object for a square matrix of side\_dimension rows and columns. The vectorized form contains the entries of the matrix given column by column (or equivalently, the entries of the lower-left triangular part given row by row).

source

## SymmetricMatrixAdjointShape

JuMP.SymmetricMatrixAdjointShape - Type.

SymmetricMatrixAdjointShape(side\_dimension)

The dual\_shape of SymmetricMatrixShape.

This shape is not intended for regular use.

source

### SymmetricMatrixShape

JuMP.SymmetricMatrixShape - Type.

```
SymmetricMatrixShape(
 side_dimension::Int;
 needs_adjoint_dual::Bool = false,
)
```

The shape object for a symmetric square matrix of side\_dimension rows and columns.

The vectorized form contains the entries of the upper-right triangular part of the matrix given column by column (or equivalently, the entries of the lower-left triangular part given row by row).

## needs\_adjoint\_dual

By default, the dual\_shape of SymmetricMatrixShape is also SymmetricMatrixShape. This is true for cases such as a LinearAlgebra.Symmetric matrix in PSDCone.

However, JuMP also supports LinearAlgebra.Symmetric matrix in Zeros, which is interpreted as an elementwise equality constraint. By exploiting symmetry, we pass only the upper triangle of the equality constraints. This works for the primal, but it leads to a factor of 2 difference in the off-diagonal dual elements. (The dual value of the (i, j) element in the triangle formulation should be divided by 2 when spread across the (i, j) and (j, i) elements in the square matrix formulation.) If the constraint has this dual inconsistency, set needs\_adjoint\_dual = true.

source

# SymmetricMatrixSpace

JuMP.SymmetricMatrixSpace - Type.

SymmetricMatrixSpace()

Use in the @variable macro to constrain a matrix of variables to be symmetric.

```
julia> model = Model();
```

```
julia> @variable(model, Q[1:2, 1:2] in SymmetricMatrixSpace())
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
Q[1,1] Q[1,2]
Q[1,2] Q[2,2]
```

### TerminationStatusCode

JuMP.TerminationStatusCode - Type.

TerminationStatusCode

An Enum of possible values for the TerminationStatus attribute.

This attribute explains why the optimizer stopped executing in the most recent call to optimize!.

### Values

## **OPTIMIZE\_NOT\_CALLED**

The algorithm has not started.

### **OPTIMAL**

The algorithm found a globally optimal solution.

## INFEASIBLE

The algorithm proved that no primal feasible solution exists.

## DUAL\_INFEASIBLE

The algorithm proved that no dual feasible solution exists.

To check if the primal problem is feasible, set the objective sense to FEASIBILITY\_SENSE and re-solve the problem.

If a primal feasible point does not exist, the original problem is both primal and dual infeasible.

If a primal feasible solution exists, this status typically implies that the problem is unbounded, with some technical exceptions (for example, if the problem is a conic optimization problem in which strong duality does not hold).

The technical exceptions do not apply to linear programs. The combination of DUAL\_INFEASIBLE and a primal feasible point means that the primal linear program is unbounded.

## LOCALLY\_SOLVED

The algorithm converged to a stationary point, local optimal solution, could not find directions for improvement, or otherwise completed its search without global guarantees.

### LOCALLY\_INFEASIBLE

The algorithm converged to an infeasible point or otherwise completed its search without finding a feasible solution, without guarantees that no feasible solution exists.

If you know a primal feasible solution exists, use VariablePrimalStart to provide a feasible starting point to the solver.

### INFEASIBLE\_OR\_UNBOUNDED

The algorithm stopped because it proved that the problem is infeasible or unbounded, without distinguishing between the two cases.

To distinguish between the two cases, set the objective sense to FEASIBILITY\_SENSE and re-solve the problem. If a primal feasible point exists, the original problem is unbounded. If a primal feasible point does not exist, the original problem is infeasible.

## ALMOST\_OPTIMAL

The algorithm found a globally optimal solution to relaxed tolerances.

## ALMOST\_INFEASIBLE

The algorithm concluded that no feasible solution exists within relaxed tolerances.

ALMOST DUAL INFEASIBLE

The algorithm concluded that no dual bound exists for the problem within relaxed tolerances.

# ALMOST\_LOCALLY\_SOLVED

The algorithm converged to a stationary point, local optimal solution, or could not find directions for improvement within relaxed tolerances.

### **ITERATION\_LIMIT**

An iterative algorithm stopped after conducting the maximum number of iterations.

### TIME LIMIT

The algorithm stopped after a user-specified computation time.

This status may be returned in relation to the TimeLimitSec attribute, or some other solver-specific attribute.

### NODE LIMIT

A branch-and-bound algorithm stopped because it explored a maximum number of nodes in the branchand-bound tree.

This status may be returned in relation to the NodeLimit attribute, or some other solver-specific attribute.

### SOLUTION LIMIT

The algorithm stopped because it found the required number of solutions. This is often used in MIPs to get the solver to return the first feasible solution it encounters.

This status may be returned in relation to the SolutionLimit attribute, or some other solver-specific attribute.

### MEMORY LIMIT

The algorithm stopped because it ran out of memory.

## **OBJECTIVE\_LIMIT**

The algorithm stopped because it found a solution better than a minimum limit set by the user.

This status may be returned in relation to the ObjectiveLimit attribute, or some other solver-specific attribute.

# NORM\_LIMIT

The algorithm stopped because the norm of an iterate became too large.

This typically means that the primal problem is unbounded, but that the solver could not prove so.

### OTHER\_LIMIT

The algorithm stopped due to a limit not covered by one of the \_LIMIT\_ statuses above.

### **SLOW PROGRESS**

The algorithm stopped because it was unable to continue making progress towards the solution.

## NUMERICAL\_ERROR

The algorithm stopped because it encountered unrecoverable numerical error.

## INVALID\_MODEL

The algorithm stopped because the model is invalid.

The reason for this return code is solver-specific, but common causes are that the problem has zero variables or constraints, or that the problem data contains an invalid number such as NaN.

## INVALID\_OPTION

The algorithm stopped because it was provided an invalid option.

### **INTERRUPTED**

The algorithm stopped because of an interrupt signal.

This typically means that the solver was interrupted by the user with CTRL+C.

## **OTHER ERROR**

The algorithm stopped because of an error not covered by one of the statuses defined above. Check the solver log for further details.

## source

# UnorderedPair

JuMP.UnorderedPair - Type.

```
UnorderedPair(a::T, b::T)
```

A wrapper type used by GenericQuadExpr with fields .a and .b.

# Example

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2]);
```

julia> expr = 2.0 \* x[1] \* x[2]
2 x[1]\*x[2]

### julia> expr.terms

OrderedCollections.OrderedDict{UnorderedPair{VariableRef}, Float64} with 1 entry: UnorderedPair{VariableRef}(x[1], x[2]) => 2.0

# source

## UpperBoundRef

JuMP.UpperBoundRef - Function.

UpperBoundRef(v::GenericVariableRef)
Return a constraint reference to the upper bound constraint of v.

Errors if one does not exist.

See also has\_upper\_bound, upper\_bound, set\_upper\_bound, delete\_upper\_bound.

### Example

```
julia> model = Model();
```

julia> @variable(model, x <= 1.0);</pre>

julia> UpperBoundRef(x)
x ≤ 1

source

### VariableConstrainedOnCreation

JuMP.VariableConstrainedOnCreation - Type.

```
VariableConstrainedOnCreation <: AbstractVariable</pre>
```

Variable scalar\_variables constrained to belong to set.

Adding this variable can be understood as doing:

```
function JuMP.add_variable(
 model::GenericModel,
 variable::VariableConstrainedOnCreation,
 names,
)
 var_ref = add_variable(model, variable.scalar_variable, name)
 add_constraint(model, VectorConstraint(var_ref, variable.set))
 return var_ref
end
```

but adds the variables with MOI.add\_constrained\_variable(model, variable.set) instead.

source

#### VariableInSetRef

JuMP.VariableInSetRef - Function.

```
VariableInSetRef(
 model::GenericModel,
 x::Union{AbstractJuMPScalar,AbstractArray{<:AbstractJuMPScalar}},
)</pre>
```

Return the constraint reference associated with x when it is constrained on creation.

A variable is constrained on creation if it uses the x in S or x, set = S syntax in the @variable macro. This function errors if x was not constrained on creation. To check if the variable was constrained on creation, use is variable in set.

#### Exceptions

This function does not apply for variable bounds or integrality restrictions of a scalar variable. For example:

```
julia> model = Model();
julia> @variable(model, x >= 0, Int)
x
julia> is_variable_in_set(x)
false
```

Use instead IntegerRef, BinaryRef, LowerBoundRef, UpperBoundRef, and FixRef.

```
julia> model = Model();
julia> @variable(model, x >= 0, Int)
x
julia> IntegerRef(x)
x integer
julia> LowerBoundRef(x)
x ≥ 0
```

```
julia> model = Model();
julia> @variable(model, x[1:2, 1:2], PSD)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
x[1,1] x[1,2]
 x[1,2] x[2,2]
julia> is_variable_in_set(x)
true
julia> c = VariableInSetRef(x)
[x[1,1] x[1,2]
 x[2,2]] \in PSDCone()

julia> @variable(model, y)
julia> is_variable_in_set(y)
false
julia> @variable(model, z in Semicontinuous(1, 2))
Ζ
```

julia> is\_variable\_in\_set(z)
true

julia> c\_z = VariableInSetRef(z)
z ∈ MathOptInterface.Semicontinuous{Int64}(1, 2)

### source

# VariableInfo

JuMP.VariableInfo - Type.

VariableInfo{S,T,U,V}

A struct by JuMP internally when creating variables. This may also be used by JuMP extensions to create new types of variables.

See also: ScalarVariable.

source

#### VariableNotOwned

JuMP.VariableNotOwned - Type.

```
struct VariableNotOwned{V<:AbstractVariableRef} <: Exception
 variable::V
end</pre>
```

The variable variable was used in a model different to owner\_model(variable).

source

### VariableRef

JuMP.VariableRef – Type.

GenericVariableRef{T} <: AbstractVariableRef</pre>

Holds a reference to the model and the corresponding MOI.VariableIndex.

source

### VariablesConstrainedOnCreation

JuMP.VariablesConstrainedOnCreation - Type.

VariablesConstrainedOnCreation <: AbstractVariable</pre>

Vector of variables scalar\_variables constrained to belong to set. Adding this variable can be thought as doing:

```
function JuMP.add_variable(
 model::GenericModel,
 variable::VariablesConstrainedOnCreation,
 names,
)
 v_names = vectorize(names, variable.shape)
 var_refs = add_variable.(model, variable.scalar_variables, v_names)
 add_constraint(model, VectorConstraint(var_refs, variable.set))
 return reshape_vector(var_refs, variable.shape)
end
```

but adds the variables with MOI.add\_constrained\_variables(model, variable.set) instead. See the MOI documentation for the difference between adding the variables with MOI.add\_constrained\_variables and adding them with MOI.add\_variables and adding the constraint separately.

#### source

## VectorConstraint

JuMP.VectorConstraint - Type.

#### struct VectorConstraint

The data for a vector constraint.

See also the documentation on JuMP's representation of constraints.

### Fields

- func: field contains a JuMP object representing the function
- set: field contains the MOI set.
- shape: field contains an AbstractShape matching the form in which the constraint was constructed (for example, by using matrices or flat vectors).

#### Example

julia> model = Model();

julia> @variable(model, x[1:3]);

julia> @constraint(model, c, x in SecondOrderCone())
c : [x[1], x[2], x[3]] ∈ MathOptInterface.SecondOrderCone(3)

#### julia> object = constraint\_object(c)

```
VectorConstraint{VariableRef, MathOptInterface.SecondOrderCone, VectorShape}(VariableRef[x[1],

→ x[2], x[3]], MathOptInterface.SecondOrderCone(3), VectorShape())
```

#### julia> typeof(object)

VectorConstraint{VariableRef, MathOptInterface.SecondOrderCone, VectorShape}

```
julia> object.func
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
```

## julia> object.set

MathOptInterface.SecondOrderCone(3)

julia> object.shape
VectorShape()

### source

## VectorShape

JuMP.VectorShape - Type.

VectorShape()

An AbstractShape that represents vector-valued constraints.

### Example

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2]);
```

```
julia> c = @constraint(model, x in SOS1());
```

julia> shape(constraint\_object(c))
VectorShape()

#### source

### Zeros

JuMP.Zeros - Type.

Zeros()

The JuMP equivalent of the MOI.Zeros set, in which the dimension is inferred from the corresponding function.

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

```
julia> @constraint(model, x in Zeros())
[x[1], x[2]] ∈ Zeros()
```

```
julia> A = [1 2; 3 4];
```

julia> b = [5, 6];

```
julia> @constraint(model, A * x == b)
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ Zeros()
```

### source

## ALMOST\_DUAL\_INFEASIBLE

JuMP.ALMOST\_DUAL\_INFEASIBLE - Constant.

ALMOST\_DUAL\_INFEASIBLE::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm concluded that no dual bound exists for the problem within relaxed tolerances.

source

### ALMOST\_INFEASIBLE

JuMP.ALMOST\_INFEASIBLE - Constant.

ALMOST\_INFEASIBLE::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm concluded that no feasible solution exists within relaxed tolerances.

source

## ALMOST\_LOCALLY\_SOLVED

JuMP.ALMOST\_LOCALLY\_SOLVED - Constant.

ALMOST\_LOCALLY\_SOLVED:::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm converged to a stationary point, local optimal solution, or could not find directions for improvement within relaxed tolerances.

source

### ALMOST\_OPTIMAL

JuMP.ALMOST\_OPTIMAL - Constant.

ALMOST\_OPTIMAL:::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm found a globally optimal solution to relaxed tolerances.

source

### AUTOMATIC

JuMP.AUTOMATIC - Constant.

moi\_backend field holds a CachingOptimizer in AUTOMATIC mode.

source

## DIRECT

JuMP.DIRECT - Constant.

moi\_backend field holds an AbstractOptimizer. No extra copy of the model is stored. The moi\_backend must support add\_constraint etc.

source

### DUAL\_INFEASIBLE

JuMP.DUAL\_INFEASIBLE - Constant.

DUAL\_INFEASIBLE::TerminationStatusCode

An instance of the TerminationStatusCode enum.

#### About

The algorithm proved that no dual feasible solution exists.

To check if the primal problem is feasible, set the objective sense to FEASIBILITY\_SENSE and re-solve the problem.

If a primal feasible point does not exist, the original problem is both primal and dual infeasible.

If a primal feasible solution exists, this status typically implies that the problem is unbounded, with some technical exceptions (for example, if the problem is a conic optimization problem in which strong duality does not hold).

The technical exceptions do not apply to linear programs. The combination of DUAL\_INFEASIBLE and a primal feasible point means that the primal linear program is unbounded.

source

### FEASIBILITY\_SENSE

JuMP.FEASIBILITY\_SENSE - Constant.

FEASIBILITY\_SENSE::OptimizationSense

An instance of the OptimizationSense enum.

#### About

The model does not have an objective function.

source

#### FEASIBLE\_POINT

JuMP.FEASIBLE\_POINT - Constant.

FEASIBLE POINT::ResultStatusCode

An instance of the ResultStatusCode enum.

### About

The result vector is a feasible point.

source

#### INFEASIBILITY\_CERTIFICATE

JuMP.INFEASIBILITY\_CERTIFICATE - Constant.

INFEASIBILITY\_CERTIFICATE::ResultStatusCode

An instance of the ResultStatusCode enum.

### About

The result vector is an infeasibility certificate.

If the PrimalStatus is INFEASIBILITY\_CERTIFICATE, then the primal result vector is a certificate of dual infeasibility.

If the DualStatus is INFEASIBILITY\_CERTIFICATE, then the dual result vector is a proof of primal infeasibility.

source

### INFEASIBLE

JuMP.INFEASIBLE - Constant.

INFEASIBLE::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm proved that no primal feasible solution exists.

source

### INFEASIBLE\_OR\_UNBOUNDED

JuMP.INFEASIBLE\_OR\_UNBOUNDED - Constant.

INFEASIBLE\_OR\_UNBOUNDED::TerminationStatusCode

An instance of the TerminationStatusCode enum.

#### About

The algorithm stopped because it proved that the problem is infeasible or unbounded, without distinguishing between the two cases.

To distinguish between the two cases, set the objective sense to FEASIBILITY\_SENSE and re-solve the problem. If a primal feasible point exists, the original problem is unbounded. If a primal feasible point does not exist, the original problem is infeasible.

source

#### INFEASIBLE\_POINT

JuMP.INFEASIBLE\_POINT - Constant.

INFEASIBLE\_POINT::ResultStatusCode

An instance of the ResultStatusCode enum.

#### About

The result vector is an infeasible point.

source

## INTERRUPTED

JuMP.INTERRUPTED - Constant.

INTERRUPTED::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm stopped because of an interrupt signal.

This typically means that the solver was interrupted by the user with CTRL+C.

source

### INVALID\_MODEL

JuMP.INVALID\_MODEL - Constant.

INVALID\_MODEL::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm stopped because the model is invalid.

The reason for this return code is solver-specific, but common causes are that the problem has zero variables or constraints, or that the problem data contains an invalid number such as NaN.

### source

#### INVALID\_OPTION

JuMP.INVALID\_OPTION - Constant.

INVALID\_OPTION::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm stopped because it was provided an invalid option.

source

### ITERATION\_LIMIT

JuMP.ITERATION\_LIMIT - Constant.

ITERATION\_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

An iterative algorithm stopped after conducting the maximum number of iterations.

source

### LOCALLY\_INFEASIBLE

JuMP.LOCALLY\_INFEASIBLE - Constant.

LOCALLY\_INFEASIBLE::TerminationStatusCode

An instance of the TerminationStatusCode enum.

#### About

The algorithm converged to an infeasible point or otherwise completed its search without finding a feasible solution, without guarantees that no feasible solution exists.

If you know a primal feasible solution exists, use VariablePrimalStart to provide a feasible starting point to the solver.

source

### LOCALLY\_SOLVED

JuMP.LOCALLY\_SOLVED - Constant.

LOCALLY\_SOLVED::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm converged to a stationary point, local optimal solution, could not find directions for improvement, or otherwise completed its search without global guarantees.

source

## MANUAL

JuMP.MANUAL - Constant.

moi\_backend field holds a CachingOptimizer in MANUAL mode.

source

## MAX\_SENSE

JuMP.MAX\_SENSE - Constant.

MAX\_SENSE::OptimizationSense

An instance of the OptimizationSense enum.

## About

The goal is to maximize the objective function.

source

### MEMORY\_LIMIT

JuMP.MEMORY\_LIMIT - Constant.

MEMORY\_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm stopped because it ran out of memory.

### source

## MIN\_SENSE

JuMP.MIN\_SENSE - Constant.

MIN\_SENSE::OptimizationSense

An instance of the OptimizationSense enum.

### About

The goal is to minimize the objective function.

source

## NEARLY\_FEASIBLE\_POINT

JuMP.NEARLY\_FEASIBLE\_POINT - Constant.

NEARLY\_FEASIBLE\_POINT::ResultStatusCode

An instance of the ResultStatusCode enum.

### About

The result vector is feasible if some constraint tolerances are relaxed.

source

## NEARLY\_INFEASIBILITY\_CERTIFICATE

JuMP.NEARLY\_INFEASIBILITY\_CERTIFICATE - Constant.

NEARLY\_INFEASIBILITY\_CERTIFICATE::**ResultStatusCode** 

An instance of the ResultStatusCode enum.

#### About

The result satisfies a relaxed criterion for a certificate of infeasibility.

If the PrimalStatus is NEARLY\_INFEASIBILITY\_CERTIFICATE, then the primal result vector is a certificate of dual infeasibility.

If the DualStatus is NEARLY\_INFEASIBILITY\_CERTIFICATE, then the dual result vector is a proof of primal infeasibility.

source

### NEARLY\_REDUCTION\_CERTIFICATE

JuMP.NEARLY\_REDUCTION\_CERTIFICATE - Constant.

NEARLY\_REDUCTION\_CERTIFICATE::ResultStatusCode

An instance of the ResultStatusCode enum.

### About

The result satisfies a relaxed criterion for an ill-posed certificate.

source

### NODE\_LIMIT

JuMP.NODE\_LIMIT - Constant.

NODE\_LIMIT:: TerminationStatusCode

An instance of the TerminationStatusCode enum.

#### About

A branch-and-bound algorithm stopped because it explored a maximum number of nodes in the branchand-bound tree.

This status may be returned in relation to the NodeLimit attribute, or some other solver-specific attribute.

source

### NORM\_LIMIT

JuMP.NORM\_LIMIT - Constant.

NORM\_LIMIT:: TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm stopped because the norm of an iterate became too large.

This typically means that the primal problem is unbounded, but that the solver could not prove so.

source

### NO\_SOLUTION

JuMP.NO\_SOLUTION - Constant.

N0\_SOLUTION::ResultStatusCode

An instance of the ResultStatusCode enum.

### About

The result vector is empty.

source

### NUMERICAL\_ERROR

JuMP.NUMERICAL\_ERROR - Constant.

NUMERICAL\_ERROR::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm stopped because it encountered unrecoverable numerical error.

source

### **OBJECTIVE\_LIMIT**

JuMP.OBJECTIVE\_LIMIT - Constant.

OBJECTIVE\_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm stopped because it found a solution better than a minimum limit set by the user.

This status may be returned in relation to the **ObjectiveLimit** attribute, or some other solver-specific attribute.

source

### OPTIMAL

JuMP.OPTIMAL - Constant.

OPTIMAL::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm found a globally optimal solution.

source

## OPTIMIZE\_NOT\_CALLED

JuMP.OPTIMIZE\_NOT\_CALLED - Constant.

OPTIMIZE\_NOT\_CALLED::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm has not started.

source

### OTHER\_ERROR

JuMP.OTHER\_ERROR - Constant.

OTHER\_ERROR::TerminationStatusCode

An instance of the TerminationStatusCode enum.

## About

The algorithm stopped because of an error not covered by one of the statuses defined above. Check the solver log for further details.

source

## OTHER\_LIMIT

JuMP.OTHER\_LIMIT - Constant.

OTHER\_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm stopped due to a limit not covered by one of the \_LIMIT\_ statuses above.

source

## OTHER\_RESULT\_STATUS

JuMP.OTHER\_RESULT\_STATUS - Constant.

OTHER\_RESULT\_STATUS::ResultStatusCode

An instance of the ResultStatusCode enum.

### About

The result vector contains a solution with an interpretation not covered by one of the statuses defined above. Check the solver log for more details.

source

### **REDUCTION\_CERTIFICATE**

JuMP.REDUCTION\_CERTIFICATE - Constant.

REDUCTION\_CERTIFICATE::ResultStatusCode

An instance of the ResultStatusCode enum.

#### About

The result vector is an ill-posed certificate; see this article for details.

If the PrimalStatus is REDUCTION\_CERTIFICATE, then the primal result vector is a proof that the dual problem is ill-posed.

If the DualStatus is REDUCTION\_CERTIFICATE, then the dual result vector is a proof that the primal is ill-posed.

source

### SLOW\_PROGRESS

JuMP.SLOW PROGRESS - Constant.

SLOW\_PROGRESS::TerminationStatusCode

An instance of the TerminationStatusCode enum.

## About

The algorithm stopped because it was unable to continue making progress towards the solution.

source

## SOLUTION\_LIMIT

JuMP.SOLUTION\_LIMIT - Constant.

SOLUTION\_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm stopped because it found the required number of solutions. This is often used in MIPs to get the solver to return the first feasible solution it encounters.

This status may be returned in relation to the SolutionLimit attribute, or some other solver-specific attribute.

source

### TIME\_LIMIT

JuMP.TIME\_LIMIT - Constant.

TIME\_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

### About

The algorithm stopped after a user-specified computation time.

This status may be returned in relation to the TimeLimitSec attribute, or some other solver-specific attribute.

source

## UNKNOWN\_RESULT\_STATUS

JuMP.UNKNOWN\_RESULT\_STATUS - Constant.

UNKNOWN\_RESULT\_STATUS::ResultStatusCode

An instance of the ResultStatusCode enum.

### About

The result vector contains a solution with an unknown interpretation. Check the solver log for more details.

source

#### op\_and

JuMP.op and - Constant.

op\_and(x, y)

A function that falls back to x & y, but when called with JuMP variables or expressions, returns a GenericNonlinearExpr. **Example** 

```
julia> model = Model();
julia> @variable(model, x);
julia> op_and(true, false)
false
julia> op_and(true, x)
true && x
```

### source

#### op\_equal\_to

JuMP.op\_equal\_to - Constant.

```
op_equal_to(x, y)
```

A function that falls back to x == y, but when called with JuMP variables or expressions, returns a GenericNonlinearExpr.

### Example

```
julia> model = Model();
julia> @variable(model, x);
julia> op_equal_to(2, 2)
true
julia> op_equal_to(x, 2)
x == 2
```

#### source

## op\_greater\_than\_or\_equal\_to

JuMP.op\_greater\_than\_or\_equal\_to - Constant.

op\_greater\_than\_or\_equal\_to(x, y)

A function that falls back to x >= y, but when called with JuMP variables or expressions, returns a GenericNonlinearExpr.

```
julia> model = Model();
julia> @variable(model, x);
julia> op_greater_than_or_equal_to(2, 2)
true
julia> op_greater_than_or_equal_to(x, 2)
x >= 2
```

#### source

### op\_less\_than\_or\_equal\_to

JuMP.op\_less\_than\_or\_equal\_to - Constant.

op\_less\_than\_or\_equal\_to(x, y)

A function that falls back to x <= y, but when called with JuMP variables or expressions, returns a GenericNonlinearExpr.

### Example

```
julia> model = Model();
julia> @variable(model, x);
julia> op_less_than_or_equal_to(2, 2)
true
julia> op_less_than_or_equal_to(x, 2)
x <= 2</pre>
```

#### source

### op\_or

JuMP.op\_or - Constant.

op\_or(x, y)

A function that falls back to x | y, but when called with JuMP variables or expressions, returns a GenericNonlinearExpr.

```
julia> model = Model();
julia> @variable(model, x);
julia> op_or(true, false)
```

true

```
julia> op_or(true, x)
true || x
```

source

## op\_strictly\_greater\_than

JuMP.op\_strictly\_greater\_than - Constant.

op\_strictly\_greater\_than(x, y)

A function that falls back to x > y, but when called with JuMP variables or expressions, returns a GenericNonlinearExpr.

## Example

```
julia> model = Model();
julia> @variable(model, x);
julia> op_strictly_greater_than(1, 2)
false
julia> op_strictly_greater_than(x, 2)
x > 2
```

## source

## op\_strictly\_less\_than

JuMP.op\_strictly\_less\_than - Constant.

op\_strictly\_less\_than(x, y)

A function that falls back to x < y, but when called with JuMP variables or expressions, returns a GenericNonlinearExpr.

```
julia> model = Model();
julia> @variable(model, x);
julia> op_strictly_less_than(1, 2)
true
julia> op_strictly_less_than(x, 2)
x < 2</pre>
```

### Base.empty!(::GenericModel)

Base.empty! - Method.

empty!(model::GenericModel)::GenericModel

Empty the model, that is, remove all variables, constraints and model attributes but not optimizer attributes. Always return the argument.

Note: removes extensions data.

### Example

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> isempty(model)
false
julia> empty!(model)
A JuMP Model
+ solver: none
+ objective_sense: FEASIBILITY_SENSE
+ num_variables: 0
+ num_constraints: 0
L Names registered in the model: none
```

julia> print(model)
Feasibility
Subject to

julia> isempty(model)
true

source

Base.isempty(::GenericModel)

Base.isempty - Method.

isempty(model::GenericModel)

Verifies whether the model is empty, that is, whether the MOI backend is empty and whether the model is in the same state as at its creation, apart from optimizer attributes.

```
julia> model = Model();
julia> isempty(model)
true
```

```
julia> @variable(model, x[1:2]);
julia> isempty(model)
false
```

source

#### Base.copy(::AbstractModel)

Base.copy - Method.

copy(model::AbstractModel)

Return a copy of the model model. It is similar to copy\_model except that it does not return the mapping between the references of model and its copy.

### Note

Model copy is not supported in DIRECT mode, that is, when a model is constructed using the direct\_model constructor instead of the Model constructor. Moreover, independently on whether an optimizer was provided at model construction, the new model will have no optimizer, that is, an optimizer will have to be provided to the new model in the optimize! call.

#### Example

In the following example, a model model is constructed with a variable x and a constraint cref. It is then copied into a model new\_model with the new references assigned to x\_new and cref\_new.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @constraint(model, cref, x == 2)
cref : x = 2
julia> new_model = copy(model);
julia> x_new = model[:x]
x
julia> cref_new = model[:cref]
cref : x = 2
```

## source

Base.write(::I0, ::GenericModel; ::MOI.FileFormats.FileFormat)

Base.write - Method.

```
Base.write(
 io::IO,
 model::GenericModel;
 format::MOI.FileFormats.FileFormat = MOI.FileFormats.FORMAT_MOF,
 kwargs...,
)
```

Write the JuMP model model to io in the format format.

See MOI.FileFormats.FileFormat for a list of supported formats.

Other kwargs are passed to the Model constructor of the chosen format.

### **Keyword arguments**

Other kwargs are passed to the Model constructor of the chosen format.

For details, see the docstring each file format's Model constructor. For example, MOI.FileFormats.MPS.Model.

### Example

```
julia> model = Model();
julia> @variable(model, x >= 0);
julia> @objective(model, Min, 2 * x + 1);
julia> io = IOBuffer();
julia> write(io, model; format = MOI.FileFormats.FORMAT_MPS);
julia> seekstart(io);
julia> print(read(io, String))
NAME
ROWS
N OBJ
COLUMNS
 Х
 OBJ
 2
RHS
 rhs OBJ -1
RANGES
BOUNDS
LO bounds x
 0
PL bounds x
ENDATA
```

#### source

### MOI.Utilities.reset\_optimizer(::GenericModel)

MathOptInterface.Utilities.reset\_optimizer - Method.

MOIU.reset\_optimizer(model::GenericModel)

Call MOIU.reset\_optimizer on the backend of model. Cannot be called in direct mode.

source

#### MOI.Utilities.drop\_optimizer(::GenericModel)

MathOptInterface.Utilities.drop\_optimizer - Method.

MOIU.drop\_optimizer(model::GenericModel)

Call MOIU.drop\_optimizer on the backend of model.

Cannot be called in direct mode.

source

### MOI.Utilities.attach\_optimizer(::GenericModel)

MathOptInterface.Utilities.attach\_optimizer - Method.

MOIU.attach\_optimizer(model::GenericModel)

Call MOIU.attach\_optimizer on the backend of model.

Cannot be called in direct mode.

source

## @NLconstraint

JuMP.@NLconstraint - Macro.

@NLconstraint(model::GenericModel, expr)

Add a constraint described by the nonlinear expression expr. See also @constraint.

### Compat

This macro is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling. In most cases, you can replace @NLconstraint with @constraint.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @NLconstraint(model, sin(x) <= 1)
sin(x) - 1.0 ≤ 0
julia> @NLconstraint(model, [i = 1:3], sin(i * x) <= 1 / i)
3-element Vector{NonlinearConstraintRef{ScalarShape}}:
(sin(1.0 * x) - 1.0 / 1.0) - 0.0 ≤ 0
(sin(2.0 * x) - 1.0 / 2.0) - 0.0 ≤ 0
(sin(3.0 * x) - 1.0 / 3.0) - 0.0 ≤ 0
```

source

### @NLconstraints

JuMP.@NLconstraints - Macro.

@NLconstraints(model, args...)

Adds multiple nonlinear constraints to model at once, in the same fashion as the @NLconstraint macro.

The model must be the first argument, and multiple constraints can be added on multiple lines wrapped in a begin ... end block.

The macro returns a tuple containing the constraints that were defined.

#### Compat

This macro is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling. In most cases, you can replace @NLconstraints with @constraints.

```
end)
((t - sqrt(x ^ 2.0 + y ^ 2.0)) - 0.0 \ge 0, NonlinearConstraintRef{ScalarShape}[(z[1] - log(4.0))
\rightarrow - 0.0 \le 0, (z[2] - log(5.0)) - 0.0 \le 0])
```

#### source

#### @NLexpression

JuMP.@NLexpression - Macro.

@NLexpression(args...)

Efficiently build a nonlinear expression which can then be inserted in other nonlinear constraints and the objective. See also [@expression].

#### Compat

This macro is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling. In most cases, you can replace @NLexpression with @expression.

#### Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @variable(model, y)
y
julia> @NLexpression(model, my_expr, sin(x)^2 + cos(x^2))
subexpression[1]: sin(x) ^ 2.0 + cos(x ^ 2.0)
julia> @NLconstraint(model, my_expr + y >= 5)
(subexpression[1] + y) - 5.0 ≥ 0
```

julia> @NLobjective(model, Min, my\_expr)

Indexing over sets and anonymous expressions are also supported:

```
julia> @NLexpression(model, my_expr_1[i=1:3], sin(i * x))
3-element Vector{NonlinearExpression}:
 subexpression[2]: sin(1.0 * x)
 subexpression[3]: sin(2.0 * x)
 subexpression[4]: sin(3.0 * x)
julia> my_expr_2 = @NLexpression(model, log(1 + sum(exp(my_expr_1[i]) for i in 1:2)))
subexpression[5]: log(1.0 + (exp(subexpression[2]) + exp(subexpression[3])))
```

#### @NLexpressions

JuMP.@NLexpressions - Macro.

@NLexpressions(model, args...)

Adds multiple nonlinear expressions to model at once, in the same fashion as the @NLexpression macro.

The model must be the first argument, and multiple expressions can be added on multiple lines wrapped in a begin ... end block.

The macro returns a tuple containing the expressions that were defined.

### Compat

This macro is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling. In most cases, you can replace @NLexpressions with @expressions.

#### Example

#### source

#### @NLobjective

JuMP.@NLobjective - Macro.

@NLobjective(model, sense, expression)

Add a nonlinear objective to model with optimization sense sense. sense must be Max or Min.

#### Compat

This macro is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling. In most cases, you can replace @NLobjective with @objective.

#### Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> @NLobjective(model, Max, 2x + 1 + sin(x))
julia> print(model)
Max 2.0 * x + 1.0 + sin(x)
Subject to
```

source

#### @NLparameter

JuMP.@NLparameter - Macro.

@NLparameter(model, param == value)

Create and return a nonlinear parameter param attached to the model model with initial value set to value. Nonlinear parameters may be used only in nonlinear expressions.

### Example

```
julia> model = Model();
julia> @NLparameter(model, x == 10)
x == 10.0
julia> value(x)
10.0
```

@NLparameter(model, value = param\_value)

Create and return an anonymous nonlinear parameter param attached to the model model with initial value set to param\_value. Nonlinear parameters may be used only in nonlinear expressions.

```
julia> model = Model();
julia> x = @NLparameter(model, value = 10)
parameter[1] == 10.0
julia> value(x)
10.0
```

```
@NLparameter(model, param_collection[...] == value_expr)
```

Create and return a collection of nonlinear parameters param\_collection attached to the model model with initial value set to value\_expr (may depend on index sets). Uses the same syntax for specifying index sets as @variable.

### Example

```
julia> model = Model();
julia> @NLparameter(model, y[i = 1:3] == 2 * i)
3-element Vector{NonlinearParameter}:
parameter[1] == 2.0
parameter[2] == 4.0
parameter[3] == 6.0
julia> value(y[2])
4.0
```

@NLparameter(model, [...] == value\_expr)

Create and return an anonymous collection of nonlinear parameters attached to the model model with initial value set to value\_expr (may depend on index sets). Uses the same syntax for specifying index sets as @variable.

## Compat

This macro is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling. In most cases, you can replace a call like @NLparameter(model, p == value) with @variable(model, p in Parameter(value)).

### Example

```
julia> model = Model();
```

```
julia> y = @NLparameter(model, [i = 1:3] == 2 * i)
3-element Vector{NonlinearParameter}:
 parameter[1] == 2.0
 parameter[2] == 4.0
 parameter[3] == 6.0
julia> value(y[2])
4.0
```

### source

#### @NLparameters

JuMP.@NLparameters - Macro.

```
@NLparameters(model, args...)
```

Create and return multiple nonlinear parameters attached to model model, in the same fashion as @NLparameter macro.

The model must be the first argument, and multiple parameters can be added on multiple lines wrapped in a begin ... end block. Distinct parameters need to be placed on separate lines as in the following example.

The macro returns a tuple containing the parameters that were defined.

Compat
This macro is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling. In most cases, you can replace a call like
<pre>@NLparameters(model, begin     p == value end)</pre>
with
<pre>@variables(model, begin     p in Parameter(value) end)</pre>

## Example

#### source

## add\_nonlinear\_constraint

JuMP.add\_nonlinear\_constraint - Function.

```
add_nonlinear_constraint(model::Model, expr::Expr)
```

Add a nonlinear constraint described by the Julia expression ex to model.

## CHAPTER 22. DOCSTRINGS

This function is most useful if the expression ex is generated programmatically, and you cannot use @NLconstraint.

## Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

#### Notes

• You must interpolate the variables directly into the expression expr.

#### Example

```
julia> model = Model();
```

julia> @variable(model, x);

```
julia> add_nonlinear_constraint(model, :((x) + (x)^2 \le 1))
(x + x ^ 2.0) - 1.0 \le 0
```

```
× · × 2:0) - 1:0 S
```

#### source

#### add\_nonlinear\_expression

JuMP.add nonlinear expression - Function.

```
add_nonlinear_expression(model::Model, expr::Expr)
```

Add a nonlinear expression expr to model.

This function is most useful if the expression expr is generated programmatically, and you cannot use @NLexpression.

#### Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

#### Notes

• You must interpolate the variables directly into the expression expr.

## Example

```
julia> model = Model();
```

```
julia> @variable(model, x);
```

```
julia> add_nonlinear_expression(model, :($(x) + $(x)^2))
subexpression[1]: x + x ^ 2.0
```

#### source

#### add\_nonlinear\_parameter

JuMP.add\_nonlinear\_parameter - Function.

add\_nonlinear\_parameter(model::Model, value::Real)

Add an anonymous parameter to the model.

#### Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

source

### all\_nonlinear\_constraints

JuMP.all\_nonlinear\_constraints - Function.

```
all_nonlinear_constraints(model::GenericModel)
```

Return a vector of all nonlinear constraint references in the model in the order they were added to the model.

Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

This function returns only the constraints added with @NLconstraint and add\_nonlinear\_constraint. It does not return GenericNonlinearExpr constraints.

source

#### get\_optimizer\_attribute

JuMP.get\_optimizer\_attribute - Function.

```
get_optimizer_attribute(
 model::Union{GenericModel,MOI.OptimizerWithAttributes},
 attr::Union{AbstractString,MOI.AbstractOptimizerAttribute},
)
```

Return the value associated with the solver-specific attribute attr.

If attr is an AbstractString, this is equivalent to get\_optimizer\_attribute(model, MOI.RawOptimizerAttribute(name)

### Compat

This method will remain in all v1.X releases of JuMP, but it may be removed in a future v2.0 release. We recommend using get\_attribute instead.

See also: set\_optimizer\_attribute, set\_optimizer\_attributes.

### Example

```
julia> import Ipopt
```

```
julia> model = Model(Ipopt.Optimizer);
```

```
julia> get_optimizer_attribute(model, MOI.Silent())
false
```

### source

#### nonlinear\_constraint\_string

JuMP.nonlinear\_constraint\_string - Function.

```
nonlinear_constraint_string(
 model::GenericModel,
 mode::MIME,
 c::_NonlinearConstraint,
)
```

Return a string representation of the nonlinear constraint c belonging to model, given the mode.

## Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

source

### nonlinear\_dual\_start\_value

JuMP.nonlinear\_dual\_start\_value - Function.

nonlinear\_dual\_start\_value(model::Model)

Return the current value of the MOI attribute MOI.NLPBlockDualStart.

### Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

### nonlinear\_expr\_string

JuMP.nonlinear\_expr\_string - Function.

```
nonlinear_expr_string(
 model::GenericModel,
 mode::MIME,
 c::MOI.Nonlinear.Expression,
)
```

Return a string representation of the nonlinear expression c belonging to model, given the mode.

```
Compat
This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface
documented in Nonlinear Modeling.
```

### source

### nonlinear\_model

JuMP.nonlinear\_model - Function.

```
nonlinear_model(
 model::GenericModel;
 force::Bool = false,
)::Union{MOI.Nonlinear.Model,Nothing}
```

If model has nonlinear components, return a MOI.Nonlinear.Model, otherwise return nothing.

If force, always return a MOI.Nonlinear.Model, and if one does not exist for the model, create an empty one.

## Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

#### source

## num\_nonlinear\_constraints

JuMP.num\_nonlinear\_constraints - Function.

num\_nonlinear\_constraints(model::GenericModel)

Returns the number of nonlinear constraints associated with the model.



This function counts only the constraints added with @NLconstraint and add\_nonlinear\_constraint. It does not count GenericNonlinearExpr constraints.

#### source

### register

JuMP.register - Function.

```
register(
 model::Model,
 op::Symbol,
 dimension::Integer,
 f::Function;
 autodiff:Bool = false,
)
```

Register the user-defined function f that takes dimension arguments in model as the symbol op.

The function f must support all subtypes of Real as arguments. Do not assume that the inputs are Float64.

#### Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

## Notes

- For this method, you must explicitly set autodiff = true, because no user-provided gradient function ∇f is given.
- Second-derivative information is only computed if dimension == 1.
- op does not have to be the same symbol as f, but it is generally more readable if it is.

```
julia> model = Model();
julia> @variable(model, x)
x
julia> f(x::T) where {T<:Real} = x^2
f (generic function with 1 method)
julia> register(model, :foo, 1, f; autodiff = true)
julia> @NLobjective(model, Min, foo(x))
```

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

```
julia> g(x::T, y::T) where {T<:Real} = x * y
g (generic function with 1 method)</pre>
```

```
julia> register(model, :g, 2, g; autodiff = true)
```

```
julia> @NLobjective(model, Min, g(x[1], x[2]))
```

### source

```
register(
 model::Model,
 s::Symbol,
 dimension::Integer,
 f::Function,
 Vf::Function;
 autodiff:Bool = false,
)
```

Register the user-defined function f that takes dimension arguments in model as the symbol s. In addition, provide a gradient function  $\nabla f$ .

The functions fand  $\nabla f$  must support all subtypes of Real as arguments. Do not assume that the inputs are Float64.

# Compat This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

#### Notes

- If the function f is univariate (that is, dimension == 1), ∇f must return a number which represents the first-order derivative of the function f.
- If the function f is multi-variate, ∇f must have a signature matching ∇f(g::AbstractVector{T}, args::T...) where {T<:Real}, where the first argument is a vector g that is modified in-place with the gradient.</li>
- If autodiff = true and dimension == 1, use automatic differentiation to compute the second-order derivative information. If autodiff = false, only first-order derivative information will be used.
- s does not have to be the same symbol as f, but it is generally more readable if it is.
```
julia> model = Model();
```

julia> @variable(model, x)
x
julia> f(x::T) where {T<:Real} = x^2</pre>

f (generic function with 1 method)

```
julia> ∇f(x::T) where {T<:Real} = 2 * x
∇f (generic function with 1 method)
```

julia> register(model, :foo, 1, f, Vf; autodiff = true)

```
julia> @NLobjective(model, Min, foo(x))
```

```
julia> model = Model();
```

```
julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

```
julia> g(x::T, y::T) where {T<:Real} = x * y
g (generic function with 1 method)</pre>
```

```
julia> function ∇g(g::AbstractVector{T}, x::T, y::T) where {T<:Real}
 g[1] = y
 g[2] = x
 return
 end</pre>
```

 $\nabla g$  (generic function with 1 method)

julia> register(model, :g, 2, g, ∇g)

```
julia> @NLobjective(model, Min, g(x[1], x[2]))
```

#### source

```
register(
 model::Model,
 s::Symbol,
 dimension::Integer,
 f::Function,
 ∇f::Function,
 ∇²f::Function,
)
```

Register the user-defined function f that takes dimension arguments in model as the symbol s. In addition, provide a gradient function  $\nabla f$  and a hessian function  $\nabla^2 f$ .

 $\nabla f$  and  $\nabla^2 f$  must return numbers corresponding to the first- and second-order derivatives of the function f respectively.

# Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

#### Notes

- Because automatic differentiation is not used, you can assume the inputs are all Float64.
- This method will throw an error if dimension > 1.
- s does not have to be the same symbol as f, but it is generally more readable if it is.

# Example

```
julia> model = Model();
julia> @variable(model, x)
x
julia> f(x::Float64) = x^2
f (generic function with 1 method)
julia> \nabla f(x::Float64) = 2 * x
\nabla f (generic function with 1 method)
julia> \nabla^2 f(x::Float64) = 2.0
\nabla^2 f (generic function with 1 method)
```

```
julia> register(model, :foo, 1, f, \nabla f, \nabla^2 f)
```

```
julia> @NLobjective(model, Min, foo(x))
```

#### source

## set\_nonlinear\_dual\_start\_value

JuMP.set\_nonlinear\_dual\_start\_value - Function.

```
set_nonlinear_dual_start_value(
 model::Model,
 start::Union{Nothing,Vector{Float64}},
)
```

Set the value of the MOI attribute MOI.NLPBlockDualStart.

# Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

The start vector corresponds to the Lagrangian duals of the nonlinear constraints, in the order given by all\_nonlinear\_constraints. That is, you must pass a single start vector corresponding to all of the nonlinear constraints in a single function call; you cannot set the dual start value of nonlinear constraints one-by-one. The example below demonstrates how to use all\_nonlinear\_constraints to create a mapping between the nonlinear constraint references and the start vector.

Pass nothing to unset a previous start.

# Example

```
julia> model = Model();
julia> @variable(model, x[1:2]);
julia> nl1 = @NLconstraint(model, x[1] <= sqrt(x[2]));
julia> nl2 = @NLconstraint(model, x[1] >= exp(x[2]));
julia> start = Dict(nl1 => -1.0, nl2 => 1.0);
julia> start_vector = [start[con] for con in all_nonlinear_constraints(model)]
2-element Vector{Float64}:
-1.0
1.0
julia> set_nonlinear_dual_start_value(model, start_vector)
julia> nonlinear_dual_start_value(model)
2-element Vector{Float64}:
-1.0
1.0
```

#### source

# set\_nonlinear\_objective

JuMP.set\_nonlinear\_objective - Function.

```
set_nonlinear_objective(
 model::Model,
 sense::MOI.OptimizationSense,
 expr::Expr,
)
```

Set the nonlinear objective of model to the expression expr, with the optimization sense sense.

This function is most useful if the expression expr is generated programmatically, and you cannot use @NLobjective.

# Compat This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

#### Notes

- You must interpolate the variables directly into the expression expr.
- You must use MIN\_SENSE or MAX\_SENSE instead of Min and Max.

# Example

```
julia> model = Model();
julia> @variable(model, x);
julia> set_nonlinear_objective(model, MIN_SENSE, :($(x) + $(x)^2))
```

source

# set\_normalized\_coefficients

JuMP.set\_normalized\_coefficients - Function.

```
set_normalized_coefficients(
 constraint::ConstraintRef{<:AbstractModel,<:MOI.ConstraintIndex{F}},
 variable::AbstractVariableRef,
 new_coefficients::Vector{Tuple{Int64,T}},
) where {T,F<:Union{MOI.VectorAffineFunction{T},MOI.VectorQuadraticFunction{T}}}</pre>
```

A deprecated method that now redirects to set\_normalized\_coefficient.

source

## set\_optimizer\_attribute

JuMP.set\_optimizer\_attribute - Function.

```
set_optimizer_attribute(
 model::Union{GenericModel,MOI.OptimizerWithAttributes},
 attr::Union{AbstractString,MOI.AbstractOptimizerAttribute},
 value,
)
```

Set the solver-specific attribute attr in model to value.

Ifattris an AbstractString, this is equivalent to set\_optimizer\_attribute(model, MOI.RawOptimizerAttribute(name)
value).

# Compat This method will remain in all v1.X releases of JuMP, but it may be removed in a future v2.0 release. We recommend using set\_attribute instead.

See also: set\_optimizer\_attributes, get\_optimizer\_attribute.

# Example

julia> model = Model();

```
julia> set_optimizer_attribute(model, MOI.Silent(), true)
```

#### source

## set\_optimizer\_attributes

JuMP.set\_optimizer\_attributes - Function.

```
set_optimizer_attributes(
 model::Union{GenericModel,MOI.OptimizerWithAttributes},
 pairs::Pair...,
)
```

Given a list of attribute => value pairs, calls set\_optimizer\_attribute(model, attribute, value) for each pair.

# Compat This method will remain in all v1.X releases of JuMP, but it may be removed in a future v2.0 release. We recommend using set\_attributes instead.

See also: set\_optimizer\_attribute, get\_optimizer\_attribute.

## Example

julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> set\_optimizer\_attributes(model, "tol" => 1e-4, "max\_iter" => 100)

is equivalent to:

```
julia> import Ipopt
```

```
julia> model = Model(Ipopt.Optimizer);
```

julia> set\_optimizer\_attribute(model, "tol", 1e-4)

```
julia> set_optimizer_attribute(model, "max_iter", 100)
```

### source

# set\_value

JuMP.set\_value - Function.

set\_value(p::NonlinearParameter, v::Number)

Store the value v in the nonlinear parameter p.

# Compat

This function is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

# Example

```
julia> model = Model();
```

julia> @NLparameter(model, p == 0)
p == 0.0
julia> set\_value(p, 5)
5
julia> value(p)

#### source

5.0

## NonlinearConstraintIndex

JuMP.NonlinearConstraintIndex - Type.

ConstraintIndex

An index to a nonlinear constraint that is returned by add\_constraint.

Given data::Model and c::ConstraintIndex, use data[c] to retrieve the corresponding Constraint.

source

# NonlinearConstraintRef

JuMP.NonlinearConstraintRef - Type.

NonlinearConstraintRef

# Compat

This type is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

source

# NonlinearExpression

JuMP.NonlinearExpression - Type.

NonlinearExpression <: AbstractJuMPScalar

A struct to represent a nonlinear expression.

Create an expression using @NLexpression.

Compat

This type is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

## source

# NonlinearParameter

JuMP.NonlinearParameter - Type.

NonlinearParameter <: AbstractJuMPScalar

A struct to represent a nonlinear parameter.

Create a parameter using @NLparameter.

#### Compat

This type is part of the legacy nonlinear interface. Consider using the new nonlinear interface documented in Nonlinear Modeling.

source

# 22.2 JuMP.Containers

# JuMP.Containers

This page lists the public API of JuMP.Containers.

# Info

This page is an unstructured list of the JuMP.Containers API. For a more structured overview, read the Manual or Tutorial parts of this documentation.

Load all of the public the API into the current scope with:

using JuMP.Containers

Alternatively, load only the module with:

import JuMP.Containers

and then prefix all calls with JuMP.Containers. to create JuMP.Containers.<NAME>.

#### DenseAxisArray

JuMP.Containers.DenseAxisArray - Type.

DenseAxisArray(data::Array{T, N}, axes...) where {T, N}

Construct a JuMP array with the underlying data specified by the data array and the given axes. Exactly N axes must be provided, and their lengths must match size(data) in the corresponding dimensions.

## Example

```
julia> array = Containers.DenseAxisArray([1 2; 3 4], [:a, :b], 2:3)
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
 Dimension 1, [:a, :b]
 Dimension 2, 2:3
And data, a 2×2 Matrix{Int64}:
 1 2
 3 4
julia> array[:b, 3]
4
```

### source

DenseAxisArray{T}(undef, axes...) where T

Construct an uninitialized DenseAxisArray with element-type T indexed over the given axes.

#### Example

```
julia> array = Containers.DenseAxisArray{Float64}(undef, [:a, :b], 1:2);
```

```
julia> fill!(array, 1.0)
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
 Dimension 1, [:a, :b]
 Dimension 2, 1:2
And data, a 2×2 Matrix{Float64}:
 1.0 1.0
 1.0
```

julia> array[:a, 2] = 5.0

5.0

```
julia> array[:a, 2]
5.0
```

#### julia> array

```
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
 Dimension 1, [:a, :b]
 Dimension 2, 1:2
And data, a 2×2 Matrix{Float64}:
 1.0 5.0
 1.0
```

source

# **SparseAxisArray**

JuMP.Containers.SparseAxisArray - Type.

N-dimensional array with elements of type T where only a subset of the entries are defined. The entries with indices idx = (i1, i2, ..., iN) in keys(data) has value data[idx].

Note that, as opposed to SparseArrays.AbstractSparseArray, the missing entries are not assumed to be zero(T), they are simply not part of the array. This means that the result of map(f, sa::SparseAxisArray) or f.(sa::SparseAxisArray) has the same sparsity structure as sa, even if f(zero(T)) is not zero.

## Example

```
julia> using OrderedCollections: OrderedDict
julia> dict = OrderedDict((:a, 2) => 1.0, (:a, 3) => 2.0, (:b, 3) => 3.0)
OrderedDict{Tuple{Symbol, Int64}, Float64} with 3 entries:
 (:a, 2) => 1.0
 (:a, 3) => 2.0
 (:b, 3) => 3.0
julia> array = Containers.SparseAxisArray(dict)
SparseAxisArray{Float64, 2, Tuple{Symbol, Int64}} with 3 entries:
 [a, 2] = 1.0
 [a, 3] = 2.0
 [b, 3] = 3.0
julia> array[:b, 3]
3.0
```

source

#### Containers.@container

JuMP.Containers.@container - Macro.

```
@container([i=..., j=..., ...], expr[, container = :Auto])
```

Create a container with indices i, j, ... and values given by expr that may depend on the value of the indices.

```
@container(ref[i=..., j=..., ...], expr[, container = :Auto])
```

Same as above but the container is assigned to the variable of name ref.

The type of container can be controlled by the container keyword.

# Note

When the index set is explicitly given as 1:n for any expression n, it is transformed to Base.OneTo(n) before being given to container.

# Example

```
julia> Containers.@container([i = 1:3, j = 1:3], i + j)
3×3 Matrix{Int64}:
 2 3 4
 3 4 5
 4 5 6
julia> I = 1:3
1:3
julia> Containers.@container(x[i = I, j = I], i + j);
julia> x
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
 Dimension 1, 1:3
 Dimension 2, 1:3
And data, a 3×3 Matrix{Int64}:
2 3 4
3 4 5
 4 5 6
julia> Containers.@container([i = 2:3, j = 1:3], i + j)
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
 Dimension 1, 2:3
 Dimension 2, Base.OneTo(3)
And data, a 2×3 Matrix{Int64}:
3 4 5
 4 5 6
```

julia> Containers.@container([i = 1:3, j = 1:3; i <= j], i + j)
SparseAxisArray{Int64, 2, Tuple{Int64, Int64}} with 6 entries:
 [1, 1] = 2</pre>

 $\begin{bmatrix} 1, 2 \end{bmatrix} = 3 \\ \begin{bmatrix} 1, 3 \end{bmatrix} = 4 \\ \begin{bmatrix} 2, 2 \end{bmatrix} = 4 \\ \begin{bmatrix} 2, 3 \end{bmatrix} = 5 \\ \begin{bmatrix} 3, 3 \end{bmatrix} = 6$ 

#### source

### Containers.container

JuMP.Containers.container - Function.

```
container(f::Function, indices[[, ::Type{C} = AutoContainerType], names])
```

Create a container of type C with index names names, indices indices and values at given indices given by f.

If the method with names is not specialized on Type{C}, it falls back to calling container(f, indices, c) for backwards compatibility with containers not supporting index names.

## Example

```
julia> Containers.container((i, j) -> i + j, Containers.vectorized_product(Base.OneTo(3),
\hookrightarrow Base.OneTo(3))
3×3 Matrix{Int64}:
2 3 4
3 4 5
 4 5 6
julia> Containers.container((i, j) -> i + j, Containers.vectorized_product(1:3, 1:3))
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
 Dimension 1, 1:3
 Dimension 2, 1:3
And data, a 3×3 Matrix{Int64}:
2 3 4
 3 4 5
 4 5 6
julia> Containers.container((i, j) -> i + j, Containers.vectorized_product(2:3, Base.OneTo(3)))
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
 Dimension 1, 2:3
 Dimension 2, Base.OneTo(3)
And data, a 2×3 Matrix{Int64}:
3 4 5
4 5 6
julia> Containers.container((i, j) -> i + j, Containers.nested(() -> 1:3, i -> i:3, condition =
\hookrightarrow (i, j) -> isodd(i) || isodd(j)))
SparseAxisArray{Int64, 2, Tuple{Int64, Int64}} with 5 entries:
 [1, 1] = 2
 [1, 2] = 3
 [1, 3] = 4
 [2, 3] = 5
 [3, 3] = 6
```

### source

### Containers.rowtable

JuMP.Containers.rowtable - Function.

```
rowtable([f::Function=identity,] x; [header::Vector{Symbol} = Symbol[]])
```

Applies the function f to all elements of the variable container x, returning the result as a Vector of NamedTuples, where header is a vector containing the corresponding axis names.

If x is an N-dimensional array, there must be N+1 names, so that the last name corresponds to the result of f(x[i]).

If header is left empty, then the default header is [:x1, :x2, ..., :xN, :y].

# Info

A Vector of NamedTuples implements the Tables.jl interface, and so the result can be used as input for any function that consumes a 'Tables.jl' compatible source.

# Example

```
julia> @variable(model, x[i=1:2, j=i:2] >= 0, start = i+j);
```

```
julia> Containers.rowtable(start_value, x; header = [:i, :j, :start])
3-element Vector{@NamedTuple{i::Int64, j::Int64, start::Float64}}:
 (i = 1, j = 1, start = 2.0)
```

(i = 1, j = 2, start = 3.0) (i = 2, j = 2, start = 4.0)

julia> model = Model();

#### julia> Containers.rowtable(x)

```
3-element Vector{@NamedTuple{x1::Int64, x2::Int64, y::VariableRef}}:
 (x1 = 1, x2 = 1, y = x[1,1])
 (x1 = 1, x2 = 2, y = x[1,2])
 (x1 = 2, x2 = 2, y = x[2,2])
```

#### source

## Containers.default\_container

JuMP.Containers.default\_container - Function.

default\_container(indices)

If indices is a NestedIterator, return a SparseAxisArray. Otherwise, indices should be a VectorizedProductIterator and the function returns Array if all iterators of the product are Base.OneTo and returns DenseAxisArray otherwise.

source

# Containers.nested

JuMP.Containers.nested - Function.

nested(iterators...; condition = (args...) -> true)

Create a NestedIterator.

## Example

# julia> collect(iterator)

```
3-element Vector{Tuple{Int64, String}}:
 (1, "A")
 (1, "B")
 (2, "B")
```

## source

# Containers.vectorized\_product

JuMP.Containers.vectorized\_product - Function.

vectorized\_product(iterators...)

Created a VectorizedProductIterator.

# Example

```
julia> iterator = Containers.vectorized_product(1:2, ["A", "B"]);
```

```
julia> collect(iterator)
2×2 Matrix{Tuple{Int64, String}}:
 (1, "A") (1, "B")
 (2, "A") (2, "B")
```

## Containers.build\_error\_fn

```
JuMP.Containers.build_error_fn - Function.
```

build\_error\_fn(macro\_name, args, source)

Return a function that can be used in place of Base.error, but which additionally prints the macro from which it was called.

## source

### Containers.parse\_macro\_arguments

JuMP.Containers.parse\_macro\_arguments - Function.

```
parse_macro_arguments(
 error_fn::Function,
 args;
 valid_kwargs::Union{Nothing,Vector{Symbol}} = nothing,
 num_positional_args::Union{Nothing,Int,UnitRange{Int}} = nothing,
)
```

Returns a Tuple{Vector{Any},Dict{Symbol,Any}} containing the ordered positional arguments and a dictionary mapping the keyword arguments.

This specially handles the distinction of @foo(key = value) and @foo(; key = value) in macros.

An error is thrown if multiple keyword arguments are passed with the same key.

If valid\_kwargs is a Vector{Symbol}, an error is thrown if a keyword is not in valid\_kwargs.

If num\_positional\_args is not nothing, an error is thrown if the number of positional arguments is not in num\_positional\_args.

source

# Containers.parse\_ref\_sets

JuMP.Containers.parse\_ref\_sets - Function.

```
parse_ref_sets(
 error_fn::Function,
 expr;
 invalid_index_variables::Vector{Symbol} = Symbol[],
)
```

Helper function for macros to construct container objects.

#### Warning

This function is for advanced users implementing JuMP extensions. See container\_code for more details.

Arguments

- error\_fn: a function that takes a String and throws an error, potentially annotating the input string
  with extra information such as from which macro it was thrown from. Use error if you do not want
  a modified error message.
- expr: an Expr that specifies the container, for example, :(x[i = 1:3, [:red, :blue], k = S; i + k <= 6])</li>

# Returns

- 1. name: the name of the container, if given, otherwise nothing
- index\_vars: a Vector{Any} of names for the index variables, for example, [:i, gensym(), :k]. These may also be expressions, like :((i, j)) from a call like :(x[(i, j) in S]).
- 3. indices: an iterator over the indices, for example, Containers.NestedIterator

#### Example

See container\_code for a worked example.

source

# Containers.build\_name\_expr

JuMP.Containers.build\_name\_expr - Function.

```
build_name_expr(
 name::Union{Symbol,Nothing},
 index_vars::Vector,
 kwargs::Dict{Symbol,Any},
)
```

Returns an expression for the name of a container element, where name and index\_vars are the values returned by parse\_ref\_sets and kwargs is the dictionary returned by parse\_macro\_arguments.

This assumes that the key in kwargs used to over-ride the name choice is : base\_name.

### Example

```
julia> Containers.build_name_expr(:x, [:i, :j], Dict{Symbol,Any}())
:(string("x", "[", string($(Expr(:escape, :i))), ",", string($(Expr(:escape, :j))), "]"))
julia> Containers.build_name_expr(nothing, [:i, :j], Dict{Symbol,Any}())
""
julia> Containers.build_name_expr(:y, [:i, :j], Dict{Symbol,Any}(:base_name => "y"))
:(string("y", "[", string($(Expr(:escape, :i))), ",", string($(Expr(:escape, :j))), "]"))
```

### source

#### Containers.add\_additional\_args

JuMP.Containers.add\_additional\_args - Function.

```
add_additional_args(
 call::Expr,
 args::Vector,
 kwargs::Dict{Symbol,Any};
 kwarg_exclude::Vector{Symbol} = Symbol[],
)
```

Add the positional arguments args to the function call expression call, escaping each argument expression.

This function is able to incorporate additional positional arguments to calls that already have keyword arguments.

source

# Containers.container\_code

JuMP.Containers.container\_code - Function.

```
container_code(
 index_vars::Vector{Any},
 indices::Expr,
 code,
 requested_container::Union{Symbol,Expr,Dict{Symbol,Any}},
)
```

Used in macros to construct a call to container. This should be used in conjunction with parse\_ref\_sets.

#### Arguments

- index\_vars::Vector{Any}: a vector of names for the indices of the container. These may also be expressions, like :((i, j)) from a call like :(x[(i, j) in S]).
- indices::Expr: an expression that evaluates to an iterator of the indices.
- code: an expression or literal constant for the value to be stored in the container as a function of the named index vars.
- requested\_container: passed to the third argument of container. For built-in JuMP types, choose one of :Array, :DenseAxisArray, :SparseAxisArray, or :Auto. For a user-defined container, this expression must evaluate to the correct type. You may also pass the kwargs dictionary from parse\_macro\_arguments.

#### Warning

In most cases, you should esc(code) before passing it to container\_code.

# Example

```
julia> macro foo(ref_sets, code)
 name, index_vars, indices =
 Containers.parse_ref_sets(error, ref_sets)
 @assert name !== nothing # Anonymous container not supported
```

container =
 Containers.container\_code(index\_vars, indices, esc(code), :Auto)
 return quote
 \$(esc(name)) = \$container
 end
 end
 end
 end
 @foo (macro with 1 method)

julia> @foo(x[i=1:2, j=["A", "B"]], j^i);

# **julia**> x

```
2-dimensional DenseAxisArray{String,2,...} with index sets:
 Dimension 1, Base.OneTo(2)
 Dimension 2, ["A", "B"]
And data, a 2×2 Matrix{String}:
 "A" "B"
 "AA" "BB"
```

#### source

# Containers.AutoContainerType

JuMP.Containers.AutoContainerType - Type.

AutoContainerType

Pass AutoContainerType to container to let the container type be chosen based on the type of the indices using default\_container.

source

### Containers.NestedIterator

JuMP.Containers.NestedIterator - Type.

```
struct NestedIterator{T}
 iterators::T # Tuple of functions
 condition::Function
end
```

Iterators over the tuples that are produced by a nested for loop.

Construct a NestedIterator using nested.

Example

julia> iterators = (() -> 1:2, (i,) -> ["A", "B"]);

julia> condition = (i, j) -> isodd(i) || j == "B";

julia> x = Containers.NestedIterator(iterators, condition);

# is the same as

## source

# Containers.VectorizedProductIterator

JuMP.Containers.VectorizedProductIterator - Type.

```
struct VectorizedProductIterator{T}
 prod::Iterators.ProductIterator{T}
end
```

A wrapper type for Iterators.ProuctIterator that discards shape information and returns a Vector.

Construct a VectorizedProductIterator using vectorized\_product.

source

Part V

**Background Information** 

# **Chapter 23**

# Algebraic modeling languages

JuMP is an algebraic modeling language for mathematical optimization written in the Julia language. In this page, we explain what an algebraic modeling language actually is.

# 23.1 What is an algebraic modeling language?

If you have taken a class in mixed-integer linear programming, you will have seen a formulation like:

$$\min c^{\top} x \\ \text{s.t.} Ax = b \\ x \ge 0 \\ x_i \in \mathbb{Z}, \quad \forall i \in \mathcal{I}$$

where c, A, and b are appropriately sized vectors and matrices of data, and  $\mathcal{I}$  denotes the set of variables that are integer.

Solvers expect problems in a *standard form* like this because it limits the types of constraints that they need to consider. This makes writing a solver much easier.

What is a solver? A solver is a software package that computes solutions to one or more classes of problems. For example, HiGHS is a solver for linear programming (LP) and mixed integer programming (MIP) problems. It incorporates algorithms such as the simplex method and the interior-point method. JuMP currently supports a number of open-source and commercial solvers, which can be viewed in the Supported-solvers table.

Despite the textbook view of a linear program, you probably formulated problems algebraically like so:

$$\max \sum_{i=1}^{n} c_i x_i$$
  
s.t.  $\sum_{i=1}^{n} w_i x_i \le b$   
 $x_i \ge 0 \quad \forall i = 1, \dots, n$   
 $x_i \in \mathbb{Z} \quad \forall i = 1, \dots, n.$ 

Info

Do you recognize this formulation? It's the knapsack problem.

Users prefer to write problems in *algebraic form* because it is more convenient. For example, we used  $\leq b$ , even though the standard form only supported constraints of the form Ax = b.

We could convert our knapsack problem into the standard form by adding a new slack variable  $x_0$ :

$$\max \sum_{i=1}^{n} c_i x_i$$
  
s.t. $x_0 + \sum_{i=1}^{n} w_i x_i = b$   
 $x_i \ge 0 \quad \forall i = 0, \dots, n$   
 $x_i \in \mathbb{Z} \quad \forall i = 1, \dots, n.$ 

However, as models get more complicated, this manual conversion becomes more and more error-prone.

An algebraic modeling language is a tool that simplifies the translation between the algebraic form of the modeler, and the standard form of the solver.

Each algebraic modeling language has two main parts:

- 1. A domain specific language for the user to write down problems in algebraic form.
- 2. A converter from the algebraic form into a standard form supported by the solver (and back again).

Part 2 is less trivial than it might seem, because each solver has a unique application programming interface (API) and data structure for representing optimization models and obtaining results.

JuMP uses the MathOptInterface.jl package to abstract these differences between solvers.

## What is MathOptInterface?

MathOptInterface (MOI) is an abstraction layer designed to provide an interface to mathematical optimization solvers so that users do not need to understand multiple solver-specific APIs. MOI can be used directly, or through a higher-level modeling interface like JuMP.

There are three main parts to MathOptInterface:

- 1. A solver-independent API that abstracts concepts such as adding and deleting variables and constraints, setting and getting parameters, and querying results. For more information on the MathOptInterface API, read the documentation.
- 2. An automatic rewriting system based on equivalent formulations of a constraint. For more information on this rewriting system, read the LazyBridgeOptimizer section of the manual, and our paper on arXiv.
- 3. Utilities for managing how and when models are copied to solvers. For more information on this, read the CachingOptimizer section of the manual.

# 23.2 From user to solver

This section provides a brief summary of the steps that happen in order to translate the model that the user writes into a model that the solver understands.

# Step I: writing in algebraic form

JuMP provides the first part of an algebraic modeling language using the <code>@variable</code>, <code>@objective</code>, and <code>@constraint</code> macros.

For example, here's how we write the knapsack problem in JuMP:

```
julia> using JuMP, HiGHS
julia> function algebraic_knapsack(c, w, b)
 n = length(c)
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 @variable(model, x[1:n] >= 0, Int)
 @objective(model, Max, sum(c[i] * x[i] for i = 1:n))
 @constraint(model, sum(w[i] * x[i] for i = 1:n) <= b)</pre>
 optimize!(model)
 if termination_status(model) != OPTIMAL
 error("Not solved correctly")
 end
 return value.(x)
 end
algebraic_knapsack (generic function with 1 method)
julia> algebraic_knapsack([1, 2], [0.5, 0.5], 1.25)
```

```
2-element Vector{Float64}:
0.0
2.0
```

This formulation is compact, and it closely matches the algebraic formulation of the model we wrote out above.

### Step II: algebraic to functional

For the next step, JuMP's macros re-write the variables and constraints into a functional form. Here's what the JuMP code looks like after this step:

julia> using JuMP, HiGHS

```
julia> function nonalgebraic_knapsack(c, w, b)
```

```
n = length(c)
 model = Model(HiGHS.Optimizer)
 set_silent(model)
 x = [VariableRef(model) for i = 1:n]
 for i = 1:n
 set_lower_bound(x[i], 0)
 set_integer(x[i])
 set_name(x[i], "x[$i]")
 end
 obj = AffExpr(0.0)
 for i = 1:n
 add_to_expression!(obj, c[i], x[i])
 end
 set_objective(model, MAX_SENSE, obj)
 lhs = AffExpr(0.0)
 for i = 1:n
 add_to_expression!(lhs, w[i], x[i])
 end
 con = build_constraint(error, lhs, MOI.LessThan(b))
 add_constraint(model, con)
 optimize!(model)
 if termination_status(model) != OPTIMAL
 error("Not solved correctly")
 end
 return value.(x)
 end
nonalgebraic_knapsack (generic function with 1 method)
julia> nonalgebraic_knapsack([1, 2], [0.5, 0.5], 1.25)
2-element Vector{Float64}:
0.0
```

Hopefully you agree that the macro version is much easier to read.

# Part III: JuMP to MathOptInterface

2.0

In the third step, JuMP converts the functional form of the problem, that is, nonalgebraic\_knapsack, into the MathOptInterface API:

```
MOI.set(model, MOI.ObjectiveSense(), MOI.MAX_SENSE)
 obj = MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(c, x), 0.0)
 MOI.set(model, MOI.ObjectiveFunction{typeof(obj)}(), obj)
 MOI.add constraint(
 model,
 MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(w, x), 0.0),
 MOI.LessThan(b),
)
 MOI.optimize!(model)
 if MOI.get(model, MOI.TerminationStatus()) != MOI.OPTIMAL
 error("Not solved correctly")
 end
 return MOI.get.(model, MOI.VariablePrimal(), x)
 end
mathoptinterface_knapsack (generic function with 1 method)
julia> mathoptinterface_knapsack(HiGHS.Optimizer, [1.0, 2.0], [0.5, 0.5], 1.25)
2-element Vector{Float64}:
```

0.0 2.0

The code is becoming more verbose and looking less like the mathematical formulation that we started with.

# Step IV: MathOptInterface to HiGHS

As a final step, the HiGHS.jl package converts the MathOptInterface form, that is, mathoptinterface\_knapsack, into a HiGHS-specific API:

```
julia> using HiGHS
julia> function highs_knapsack(c, w, b)
 n = length(c)
 model = Highs create()
 Highs_setBoolOptionValue(model, "output_flag", false)
 for i in 1:n
 Highs_addCol(model, c[i], 0.0, Inf, 0, C_NULL, C_NULL)
 Highs_changeColIntegrality(model, i-1, 1)
 end
 Highs_changeObjectiveSense(model, -1)
 Highs_addRow(
 model,
 -Inf,
 b,
 Cint(length(w)),
 collect(Cint(0):Cint(n-1)),
 w,
)
 Highs_run(model)
 if Highs_getModelStatus(model) != kHighsModelStatusOptimal
 error("Not solved correctly")
 end
 x = fill(NaN, 2)
 Highs_getSolution(model, x, C_NULL, C_NULL, C_NULL)
 Highs_destroy(model)
```

# return x

end

highs\_knapsack (generic function with 1 method)

```
julia> highs_knapsack([1.0, 2.0], [0.5, 0.5], 1.25)
2-element Vector{Float64}:
 0.0
 2.0
```

We've now gone from a algebraic model that looked identical to the mathematical model we started with, to a verbose function that uses HiGHS-specific functionality.

The difference between algebraic\_knapsack and highs\_knapsack highlights the benefit that algebraic modeling languages provide to users. Moreover, if we used a different solver, the solver-specific function would be entirely different. A key benefit of an algebraic modeling language is that you can change the solver without needing to rewrite the model.

# **Chapter 24**

# **Bibliography**

- Barvinok, A. (2002). A course in convexity. Vol. 54 of Graduate studies in mathematics (American Mathematical Society).
- Ben-Tal, A. and Nemirovski, A. (2001). *Lectures on Modern Convex Optimization* (Society for Industrial and Applied Mathematics).
- Bertsimas, D.; Gupta, V. and Kallus, N. (2018). Data-driven robust optimization. Mathematical Programming 167, 235–292.
- Betts, J. T. (2010). *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Second Edition (Society for Industrial and Applied Mathematics).
- Boyd, S. and Vandenberghe, L. (2004). Convex Optimization (Cambridge University Press, Cambridge).
- Bukhsh, W. A.; Grothey, A.; McKinnon, K. I. and Trodden, P. A. (2013). Local Solutions of the Optimal Power Flow Problem. IEEE Transactions on Power Systems 28, 4780–4788.
- Cornuéjols, G.; Peña, J. and Tütüncü, R. (2018). *Optimization Methods in Finance*. 2 Edition (Cambridge University Press).
- D'Aertrycke, G.; Ehrenmann, A.; Ralph, D. and Smeers, Y. (2017). *Risk trading in capacity equilibrium models* (Cambridge Working Papers in Economics (CWPE)).
- Ferris, M. C.; Mangasarian, O. L. and Wright, S. J. (2007). *Linear Programming with MATLAB* (Society for Industrial and Applied Mathematics).
- Goemans, M. X. and Williamson, D. P. (1995). *Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming*. J. ACM **42**, 1115–1145.
- Jabr, R. A. (2012). Exploiting Sparsity in SDP Relaxations of the OPF Problem. IEEE Transactions on Power Systems **27**, 1138–1139.
- Knuth, D. E. (1994). The sandwich theorem. The Electronic Journal of Combinatorics 1.
- Krasko, V. and Rebennack, S. (2017). Global Optimization: Optimal Power Flow Problem. In: Advances and Trends in Optimization with Engineering Applications, edited by Terlaky, T.; Anjos, M. F. and Ahmed, S. (Society for Industrial and Applied Mathematics, Philadelphia, PA); Chapter 15, pp. 187–205.
- Linial, N. (2002). Finite Metric Spaces: Combinatorics, Geometry and Algorithms. In: Proceedings of the Eighteenth Annual Symposium on Computational Geometry, SCG '02 (Association for Computing Machinery, New York, NY, USA); p. 63.
- Matoušek, J. (2013). *Lectures on discrete geometry*. Vol. 212 no. 1 of *Graduate Texts in Mathematics* (Springer Science & Business Media).

- Peng, J. and Wei, Y. (2007). *Approximating K-means-type Clustering via Semidefinite Programming*. SIAM Journal on Optimization **18**, 186–205.
- Recht, B.; Fazel, M. and Parrilo, P. A. (2010). *Guaranteed Minimum-Rank Solutions of Linear Matrix Equations* via Nuclear Norm Minimization. SIAM Review **52**, 471–501.
- Zimmerman, R. D.; Murillo-Sánchez, C. E. and Thomas, R. J. (2011). *MATPOWER: Steady-State Operations, Planning, and Analysis Tools for Power Systems Research and Education.* IEEE Transactions on Power Systems **26**, 12–19.

Part VI

**Developer Docs** 

# **Chapter 25**

# Contributing

# 25.1 How to contribute to JuMP

Welcome, this document explains some ways you can contribute to JuMP.

# **Code of Conduct**

This project and everyone participating in it is governed by the JuMP Code of Conduct. By participating, you are expected to uphold this code.

# Join the community forum

First up, join the community forum.

The forum is a good place to ask questions about how to use JuMP. You can also use the forum to discuss possible feature requests and bugs before raising a GitHub issue (more on this below).

Aside from asking questions, the easiest way you can contribute to JuMP is to help answer questions on the forum.

# Join the developer chatroom

If you're interested in contributing code to JuMP, the next place to join is the developer chatroom. Let us know what you have in mind, and we can point you in the right direction.

# Improve the documentation

Chances are, if you asked (or answered) a question on the community forum, then it is a sign that the documentation could be improved. Moreover, since it is your question, you are probably the best-placed person to improve it.

The docs are written in Markdown and are built using Documenter.jl. You can find the source of all the docs here.

If your change is small (like fixing typos, or one or two sentence corrections), the easiest way to do this is via GitHub's online editor. (GitHub has help on how to do this.)

If your change is larger, or touches multiple files, you will need to make the change locally and then use Git to submit a pull request. (See Contribute code to JuMP below for more on this.)

# Тір

If you need any help, come join the developer chatroom and we will walk you through the process.

# File a bug report

Another way to contribute to JuMP is to file bug reports.

Make sure you read the info in the box where you write the body of the issue before posting. You can also find a copy of that info here.

Tip

If you're unsure whether you have a real bug, post on the community forum first. Someone will either help you fix the problem, or let you know the most appropriate place to open a bug report.

# Contribute code to JuMP

Finally, you can also contribute code to JuMP.

#### Warning

If you do not have experience with Git, GitHub, and Julia development, the first steps can be a little daunting. However, there are lots of tutorials available online, including these for:

- GitHub
- Git and GitHub
- Git
- Julia package development

If you need any help, come join the developer chatroom and we will walk you through the process.

Once you are familiar with Git and GitHub, the workflow for contributing code to JuMP is similar to the following:

#### Step 1: decide what to work on

The first step is to find an open issue (or open a new one) for the problem you want to solve. Then, *before* spending too much time on it, discuss what you are planning to do in the issue to see if other contributors are fine with your proposed changes. Getting feedback early can improve code quality, and avoid time spent writing code that does not get merged into JuMP.

## Тір

At this point, remember to be patient and polite; you may get a *lot* of comments on your issue. However, do not be afraid. Comments mean that people are willing to help you improve the code that you are contributing to JuMP.

## Step 2: fork JuMP

Go to https://github.com/jump-dev/JuMP.jl and click the "Fork" button in the top-right corner. This will create a copy of JuMP under your GitHub account.

## Step 3: install JuMP locally

Open Julia and run:

] dev JuMP

This will download the JuMP Git repository to ~/.julia/dev/JuMP. If you're on Windows, this will be C:\\Users\\<my\_name>\\.jul

#### Warning

] command means "first type ] to enter the Julia pkg mode, then type the rest. Don't copy-paste the code directly.

# Step 4: checkout a new branch

#### Note

In the following, replace any instance of GITHUB\_ACCOUNT with your GitHub user name.

The next step is to checkout a development branch. In a terminal (or command prompt on Windows), run:

\$ cd ~/.julia/dev/JuMP

\$ git remote add GITHUB\_ACCOUNT https://github.com/GITHUB\_ACCOUNT/JuMP.jl.git

- \$ git checkout master
- \$ git pull
- \$ git checkout -b my\_new\_branch

#### Tip

Lines starting with \$ mean "run these in a terminal (command prompt on Windows)."

#### Step 5: make changes

Now make any changes to the source code inside the  ${\sim}/.julia/dev/JuMP$  directory.

Make sure you:

- Follow the Style guide and run JuliaFormatter
- · Add tests and documentation for any changes or new features

#### Tip

When you change the source code, you'll need to restart Julia for the changes to take effect. This is a pain, so install Revise.jl.

## Step 6a: test your code changes

To test that your changes work, run the JuMP test-suite by opening Julia and running:

```
cd("~/.julia/dev/JuMP")
] activate .
] test
```

#### Warning

Running the tests might take a long time ( $\sim$ 10-15 minutes).

# Тір

If you're using Revise.jl, you can also run the tests by calling include:

include("test/runtests.jl")

This can be faster if you want to re-run the tests multiple times.

#### Step 6b: test your documentation changes

Open Julia, then run:

```
cd("~/.julia/dev/JuMP/docs")
] activate .
include("src/make.jl")
```

#### Warning

Building the documentation might take a long time (~10 minutes).

# Тір

If there's a problem with the tests that you don't know how to fix, don't worry. Continue to step 5, and one of the JuMP contributors will comment on your pull request telling you how to fix things.

## Step 7: make a pull request

Once you've made changes, you're ready to push the changes to GitHub. Run:

```
$ cd ~/.julia/dev/JuMP
```

\$ git add .

- \$ git commit -m "A descriptive message of the changes"
- \$ git push -u GITHUB\_ACCOUNT my\_new\_branch

# CHAPTER 25. CONTRIBUTING

Then go to https://github.com/jump-dev/JuMP.jl and follow the instructions that pop up to open a pull request.

## Step 8: respond to comments

At this point, remember to be patient and polite; you may get a *lot* of comments on your pull request. However, do not be afraid. A lot of comments means that people are willing to help you improve the code that you are contributing to JuMP.

To respond to the comments, go back to step 5, make any changes, test the changes in step 6, and then make a new commit in step 7. Your PR will automatically update.

## Step 9: cleaning up

Once the PR is merged, clean-up your Git repository ready for the next contribution.

- \$ cd ~/.julia/dev/JuMP
- \$ git checkout master

\$ git pull

# Note

If you have suggestions to improve this guide, please make a pull request. It's particularly helpful if you do this after your first pull request because you'll know all the parts that could be explained better.

# **Chapter 26**

# **Extensions**

# 26.1 Extensions

JuMP provides a variety of ways to extend the basic modeling functionality.

Тір
This documentation in this section is still a work-in-progress. The best place to look for ideas and help when writing a new JuMP extension are existing JuMP extensions. Examples include:
• BilevelJuMP.jl
• Coluna.jl
• InfiniteOpt.jl
• Plasmo.jl
• PolyJuMP.jl
• SDDP.jl
StochasticPrograms.jl
SumOfSquares.jl
• vOptGeneric.jl

# Compatibility

When writing JuMP extensions, you should carefully consider the compatibility guarantees that JuMP makes. In particular:

- All functions, structs, and constants which do not begin with an underscore (\_) are public. These are always safe to use, and they should all have corresponding documentation.
- All identifiers beginning with an underscore (\_) are private. These are not safe to use, because they may break in any JuMP release, including patch releases.

Unless explicitly mentioned in the documentation, all fields of a struct are private. These are not safe to
use, because they may break in any JuMP release, including patch releases. An example of a field which
is safe to use is the model.ext extension dictionary, which is documented in The extension dictionary.

In general, we strongly encourage you to use only the public API of JuMP. If you are missing a feature, please open a GitHub issue.

However, if you *do* use the private API (for example, because your feature request has not been implemented yet), then you must carefully restrict the versions of JuMP that your package is compatible with in the Project.toml file. The easiest way to do this is via the hyphen specifiers. For example, if your package supports all JuMP versions between v1.0.0 and v1.1.1, do:

JuMP = "1.0.0 - 1.1.1"

Then, whenever JuMP releases a new version, you should check if your package is still compatible and update the bound accordingly.

## Define a new set

To define a new set for JuMP, subtype MOI.AbstractScalarSet or MOI.AbstractVectorSet and implement Base.copy for the set.

However, for vector-sets, this requires the user to specify the dimension argument to their set, even though we could infer it from the length of x!

You can make a more user-friendly set by subtyping AbstractVectorSet and implementing moi\_set.

```
julia> struct NewVectorSet <: JuMP.AbstractVectorSet end
julia> JuMP.moi_set(::NewVectorSet, dim::Int) = NewMOIVectorSet(dim)
julia> @constraint(model, x in NewVectorSet())
[x[1], x[2]] ∈ NewMOIVectorSet(2)
```

# **Extend** @variable

Just as Bin and Int create binary and integer variables, you can extend the @variable macro to create new types of variables. Here is an explanation by example, where we create a AddTwice type, that creates a tuple of two JuMP variables instead of a single variable.

First, create a new struct. This can be anything. Our struct holds a VariableInfo object that stores bound information, and whether the variable is binary or integer.

Second, implement build\_variable, which takes :: Type{AddTwice} as an argument, and returns an instance of AddTwice. Note that you can also receive keyword arguments.

```
julia> function JuMP.build_variable(
 _err::Function,
 info::JuMP.VariableInfo,
 ::Type{AddTwice};
 kwargs...
)
 println("Can also use $kwargs here.")
 return AddTwice(info)
end
```

Third, implement add\_variable, which takes the instance of AddTwice from the previous step, and returns something. Typically, you will want to call add\_variable here. For example, our AddTwice call is going to add two JuMP variables.

```
julia> function JuMP.add_variable(
 model::JuMP.Model,
 duplicate::AddTwice,
 name::String,
)
 a = JuMP.add_variable(
 model,
 JuMP.ScalarVariable(duplicate.info),
 "$(name)_a",
)
 b = JuMP.add_variable(
 model.
 JuMP.ScalarVariable(duplicate.info),
 "$(name)_<mark>b</mark>",
)
 return (a, b)
 end
```

Now AddTwice can be passed to @variable similar to Bin or Int, or through the variable\_type keyword. However, now it adds two variables instead of one.

```
julia> model = Model();
```

```
julia> @variable(model, x[i=1:2], variable_type = AddTwice, kw = i)
Can also use Base.Pairs(:kw => 1) here.
Can also use Base.Pairs(:kw => 2) here.
2-element Vector{Tuple{VariableRef, VariableRef}}:
 (x[1]_a, x[1]_b)
```
```
(x[2]_a, x[2]_b)
julia> num_variables(model)
4
julia> first(x[1])
x[1]_a
julia> last(x[2])
x[2]_b
```

# Extend @constraint

The @constraint macro has three steps that can be intercepted and extended: parse time, build time, and add time.

# Parse

To extend the @constraint macro at parse time, implement one of the following methods:

- parse\_constraint\_head
- parse\_constraint\_call

#### Warning

julia> using JuMP

Extending the constraint macro at parse time is an advanced operation and has the potential to interfere with existing JuMP syntax. Please discuss with the developer chatroom before publishing any code that implements these methods.

parse\_constraint\_head should be implemented to intercept an expression based on the .head field of Base.Expr.
For example:

```
julia> const MutableArithmetics = JuMP._MA;
julia> model = Model(); @variable(model, x);
julia> function JuMP.parse_constraint_head(
 error_fn::Function,
 ::Val{:=},
 lhs,
 rhs,
)
 println("Rewriting = as ==")
 new_lhs, parse_code = MutableArithmetics.rewrite(lhs)
 build_code = :(
 build_constraint($(error_fn), $(new_lhs), MOI.EqualTo($(rhs)))
)
 return false, parse_code, build_code
```

end

```
julia> @constraint(model, x + x = 1.0)
Rewriting = as ==
2 x = 1
```

parse\_constraint\_call should be implemented to intercept an expression of the form Expr(:call, op, args...). For example:

```
julia> using JuMP
julia> const MutableArithmetics = JuMP. MA;
julia> model = Model(); @variable(model, x);
julia> function JuMP.parse_constraint_call(
 error_fn::Function,
 is_vectorized::Bool,
 ::Val{:my_equal_to},
 lhs,
 rhs.
)
 println("Rewriting my_equal_to to ==")
 new_lhs, parse_code = MutableArithmetics.rewrite(lhs)
 build_code = if is_vectorized
 :(build_constraint($(error_fn), $(new_lhs), MOI.EqualTo($(rhs)))
)
 else
 :(build_constraint.($(error_fn), $(new_lhs), MOI.EqualTo($(rhs))))
 end
 return parse_code, build_code
 end
```

julia> @constraint(model, my\_equal\_to(x + x, 1.0))
Rewriting my\_equal\_to to ==
2 x = 1

#### Тір

When parsing a constraint you can recurse into sub-constraint (for example, the {expr} in  $z \rightarrow x <= 1$ ) by calling parse\_constraint.

To prevent JuMP from promoting the set to the same value type as the model, use SkipModelConvertScalarSetWrapper.

# Build

To extend the @constraint macro at build time, implement a new build\_constraint method.

This may mean implementing a method for a specific function or set created at parse time, or it may mean implementing a method which handles additional positional arguments.

build\_constraint must return an AbstractConstraint, which can either be an AbstractConstraint already supported by JuMP, for example, ScalarConstraint or VectorConstraint, or a custom AbstractConstraint with a corresponding add\_constraint method (see Add).

# Тір

The easiest way to extend @constraint is via an additional positional argument to build\_constraint.

Here is an example of adding extra arguments to build\_constraint:

```
julia> model = Model(); @variable(model, x);
julia> struct MyConstrType end
julia> function JuMP.build_constraint(
 error_fn::Function,
 f::JuMP.GenericAffExpr,
 set::MOI.EqualTo,
 extra::Type{MyConstrType};
 d = 0,
)
 new_set = MOI.LessThan(set.value + d)
 return JuMP.build_constraint(error_fn, f, new_set)
 end
julia> @constraint(model, my_con, x == 0, MyConstrType, d = 2)
```

```
my_con : x \le 2
```

# Note

Only a single positional argument can be given to a particular constraint. Extensions that seek to pass multiple arguments (for example, Foo and Bar) should combine them into one argument type (for example, FooBar).

# Add

build\_constraint returns an AbstractConstraint object. To extend @constraint at add time, define a subtype of AbstractConstraint, implement build\_constraint to return an instance of the new type, and then implement add\_constraint.

Here is an example:

julia> model = Model(); @variable(model, x);

```
f::AffExpr,
 set::MOI.AbstractScalarSet,
 extra::MyTag,
)
 return MyConstraint(extra.name, f, set)
 end
julia> function JuMP.add_constraint(
 model::Model,
 con::MyConstraint,
 name::String,
)
 return add_constraint(
 model,
 ScalarConstraint(con.f, con.s),
 "$(con.name)[$(name)]",
)
 end
julia> @constraint(model, my_con, 2x <= 1, MyTag("my_prefix"))</pre>
my_prefix[my_con] : 2 \times - 1 \le 0
```

# The extension dictionary

Every JuMP model has a field .ext::Dict{Symbol,Any} that can be used by extensions. This is useful if your extensions to @variable and @constraint need to store information between calls.

The most common way to initialize a model with information in the .ext dictionary is to provide a new constructor:

```
julia> function MyModel()
 model = Model()
 model.ext[:MyModel] = 1
 return model
 end
MyModel (generic function with 1 method)
julia> model = MyModel()
A JuMP Model
 solver: none
 bojective_sense: FEASIBILITY_SENSE
 h num_variables: 0
 h num_constraints: 0
 L Names registered in the model: none
julia> model.ext
Dict{Symbol, Any} with 1 entry:
```

:MyModel => 1

If you define extension data, implement copy\_extension\_data to support copy\_model.

#### **Defining new JuMP models**

If extending individual calls to @variable and @constraint is not sufficient, it is possible to implement a new model via a subtype of AbstractModel. You can also define new AbstractVariableRefs to create different types of JuMP variables.

#### Warning

Extending JuMP in this manner is an advanced operation. We strongly encourage you to consider how you can use the methods mentioned in the previous sections to achieve your aims instead of defining new model and variable types. Consult the developer chatroom before starting work on this.

If you define new types, you will need to implement a considerable number of methods, and doing so will require a detailed understanding of the JuMP internals. Therefore, the list of methods to implement is currently undocumented.

The easiest way to extend JuMP by defining a new model type is to follow an existing example. A simple example to follow is the JuMPExtension module in the JuMP test suite. The best example of an external JuMP extension that implements an AbstractModel is InfiniteOpt.jl.

#### Testing JuMP extensions

The JuMP test suite contains a large number of tests for JuMP extensions. You can run these tests by copying the MIT-licensed Kokako.jl file in the JuMP tests into your /test folder, and then adding this snippet to your /test/runtests.jl file:

```
using MyJuMPExtension
import JuMP
include("Kokako.jl")
const MODULES_T0_TEST = Kokako.include_modules_to_test(JuMP)
Kokako.run_tests(
 MODULES_T0_TEST,
 MyJuMPExtension.MyModel,
 MyJuMPExtension.MyVariableRef;
 test_prefix = "test_extension_",
)
```

#### Set an optimize! hook

Some extensions require modification to the problem after the user has finished constructing the problem, but before optimize! is called. For these situations, JuMP provides set\_optimize\_hook, which lets you intercept the optimize! call.

Here's a simple example of adding an optimize hook that extends optimize! to take a keyword argument silent:

julia> using JuMP, HiGHS julia> model = Model(HiGHS.Optimizer); julia> @variable(model, x >= 1.5, Int); julia> @objective(model, Min, x);

```
julia> function silent_hook(model; silent::Bool)
 if silent
 set_silent(model)
 else
 unset_silent(model)
 end
 ## Make sure you set ignore_optimize_hook = true, or we'll
 ## recursively enter the optimize hook!
 return optimize!(model; ignore_optimize_hook = true)
 end
silent_hook (generic function with 1 method)
julia> set_optimize_hook(model, silent_hook)
silent_hook (generic function with 1 method)
julia> optimize!(model; silent = true)
julia> optimize!(model; silent = false)
MIP has 0 rows; 1 cols; 0 nonzeros; 1 integer variables (0 binary)
Coefficient ranges:
Cost [1e+00, 1e+00]
 Bound [2e+00, 2e+00]
Assessing feasibility of MIP using primal feasibility and integrality tolerance of
 1e-06
Solution hasnummaxsumColinfeasibilities000Integer infeasibilities000
Row infeasibilities 0
 Θ
 0
 Θ
 Θ
Row
 residuals
 Θ
Presolving model
0 rows, 0 cols, 0 nonzeros 0s
0 rows, 0 cols, 0 nonzeros 0s
Presolve: Optimal
Src: B => Branching; C => Central rounding; F => Feasibility pump; H => Heuristic; L => Sub-MIP;
 P => Empty MIP; R => Randomized rounding; S => Solve LP; T => Evaluate node; U => Unbounded;
 z => Trivial zero; l => Trivial lower; u => Trivial upper; p => Trivial point; X => User
\hookrightarrow solution
 Nodes
 | B&B Tree | Objective Bounds
 | Dynamic
 Work
→ Constraints |
Src Proc. InQueue | Leaves Expl. | BestBound
 Gap | Cuts InLp
 BestSol
→ Confl. | LpIters
 Time
0 0
↔ 0 0 0.0s
 0 0.00% 2
 2
 0.00% 0
 0
Solving report
 Status
 Optimal
 Primal bound 2
 Dual bound
 2
 Gap
 0% (tolerance: 0.01%)
 P-D integral 0
 Solution status feasible
 2 (objective)
```

```
0 (bound viol.)
 0 (int. viol.)
 0 (row viol.)
Timing
 0.00 (total)
 0.00 (presolve)
 0.00 (solve)
 0.00 (postsolve)
Max sub-MIP depth 0
Nodes
 Θ
Repair LPs
 0 (0 feasible; 0 iterations)
LP iterations 0 (total)
 0 (strong br.)
 0 (separation)
 0 (heuristics)
```

# Creating new container types

JuMP macros (for example, @variable) accept a container keyword argument to force the type of container that is chosen. By default, JuMP supports container = Array, container = DenseAxisArray, container = SparseAxisArray and container = Auto. You can extend support to user-defined types by implementing Containers.container.

For example, here is a container that reverses the order of the indices:

```
julia> struct Foo end
julia> function Containers.container(f::Function, indices, ::Type{Foo})
 return reverse([f(i...) for i in indices])
 end
julia> model = Model();
julia> @variable(model, x[1:3], container = Foo)
3-element Vector{VariableRef}:
x[3]
x[2]
x[1]
julia> x[1]
x[3]
julia> @variable(model, y[1:3, 1:2], container = Foo)
3×2 Matrix{VariableRef}:
y[3,2] y[3,1]
y[2,2] y[2,1]
y[1,2] y[1,1]
julia> y[1, 1]
y[3,2]
julia> @variable(model, z[i=1:3; isodd(i)], container = Foo)
2-element Vector{VariableRef}:
z[3]
z[1]
```

```
julia> z[2]
z[1]
```

If you are a general user, you should not need to create a new cor following User-defined containers and create a new container using star	ntainer type. Instead, consider ndard Julia syntax. For example:
<pre>julia&gt; model = Model();</pre>	
<pre>julia&gt; @variable(model, x[1:3]) 3-element Vector{VariableRef}:     x[1]     x[2]     x[3]</pre>	
<pre>julia&gt; y = reverse(x) 3-element Vector{VariableRef}:     x[3]     x[2]     x[1]</pre>	

# Performance tips for extensions

The function-in-set design of MathOptInterface causes type stability issues in Julia if you try to iterate over all of the constraints in a model. The easiest way to fix this is to use a function barrier.

```
For example, instead of:
```

function all\_names\_slow(model)

```
names = Set{String}()
 for ci in all_constraints(model)
 push!(names, name(ci))
 end
 return names
end
use:
function function_barrier(names, model, ::Type{F}, ::Type{S}) where {F,S}
 for ci in all_constraints(model, F, S)
 push!(names, name(ci))
 end
 return
end
function all_names_fast(model)
 names = Set{String}()
 for (F, S) in list_of_constraint_types(model)
```

# CHAPTER 26. EXTENSIONS

\_function\_barrier(names, model, F, S) end return names end

# Note

It is important to explicitly type the F and S arguments. If you leave them untyped, for example, function \_function\_barrier(names, model, F, S), Julia will not specialize the function calls and performance will not be improved.

# Chapter 27

# **Custom binaries**

# 27.1 How to use a custom binary

Many solvers are not written in Julia, but instead in languages like C or C++. JuMP interacts with these solvers through binary dependencies.

For many open-source solvers, we automatically install the appropriate binary when you run Pkg.add("Solver"). For example, Pkg.add("ECOS") will also install the ECOS binary.

This page explains how this installation works, and how you can use a custom binary.

Compat

These instructions require Julia 1.6 or later.

### Background

Each solver that JuMP supports is structured as a Julia package. For example, the interface for the ECOS solver is provided by the ECOS.jl package.

# Тір

This page uses the example of ECOS.jl because it is simple to compile. Other solvers follow similar conventions. For example, the interface to the Clp solver is provided by Clp.jl.

The ECOS.jl package provides an interface between the C API of ECOS and MathOptInterface. However, it does not handle the installation of the solver binary; that is the job for a JLL package.

A JLL is a Julia package that wraps a pre-compiled binary. Binaries are built using Yggdrasil (for example, ECOS) and hosted in the JuliaBinaryWrappers GitHub repository (for example, ECOS\_jll.jl).

JLL packages contain little code. Their only job is to dlopen a dynamic library, along with any dependencies.

JLL packages manage their binary dependencies using Julia's artifact system. Each JLL package has an Artifacts.toml file which describes where to find each binary artifact for each different platform that it might be installed on. Here is the Artifacts.toml file for ECOS\_jll.jl.

The binaries installed by the JLL package should be sufficient for most users. In rare cases, however, you may require a custom binary. The two main reasons to use a custom binary are:

• You want a binary with custom compilation settings (for example, debugging)

You want a binary with a set of dependencies that are not available on Yggdrasil (for example, a commercial solver like Gurobi or CPLEX).

The following sections explain how to replace the binaries provided by a JLL package with the custom ones you have compiled. As a reminder, we use ECOS as an example for simplicity, but the steps are the same for other solvers.

#### Explore the JLL you want to override

The first step is to explore the structure and filenames of the JLL package we want to override.

Find the location of the files using .artifact\_dir:

julia> using ECOS\_jll

```
julia> ECOS_jll.artifact_dir
"/Users/oscar/.julia/artifacts/2addb75332eff5a1657b46bb6bf30d2410bc7ecf"
```

### Тір

This path may be different on other machines.

Here is what it contains:

```
julia> readdir(ECOS_jll.artifact_dir)
4-element Vector{String}:
 "include"
 "lib"
 "logs"
 "share"
julia> readdir(joinpath(ECOS_jll.artifact_dir, "lib"))
1-element Vector{String}:
 "libecos.dylib"
```

Other solvers may have a bin directory containing executables. To use a custom binary of ECOS, we need to replace /lib/libecos.dylib with our custom binary.

# Compile a custom binary

The next step is to compile a custom binary. Because ECOS is written in C with no dependencies, this is easy to do if you have a C compiler:

```
oscar@Oscars-MBP jll_example % git clone https://github.com/embotech/ecos.git
[... lines omitted ...]
oscar@Oscars-MBP jll_example % cd ecos
oscar@Oscars-MBP ecos % make shared
[... many lines omitted...]
oscar@Oscars-MBP ecos % mkdir lib
oscar@Oscars-MBP ecos % cp libecos.dylib lib
```

# CHAPTER 27. CUSTOM BINARIES

#### Warning

Compiling custom solver binaries is an advanced operation. Due to the complexities of compiling various solvers, the JuMP community is unable to help you diagnose and fix compilation issues.

After this compilation step, we now have a folder /tmp/jll\_example/ecos that contains lib and include directories with the same files as ECOS\_jll:

```
julia> readdir(joinpath("ecos", "lib"))
1-element Vector{String}:
 "libecos.dylib"
```

# **Overriding a single library**

To override the libecos library, we need to know what ECOS\_jll calls it. (In most cases, it will also be libecos, but not always.)

There are two ways you can check.

- 1. Check the bottom of the JLL's GitHub README. For example, ECOS\_jll has a single LibraryProduct called libecos.
- 2. Type ECOS\_jll. and the press the [TAB] key twice to auto-complete available options:

JULIA> ECUS_JIL.				
LIBPATH	PATH_list	best_wrapper	<pre>get_libecos_path</pre>	libecos_handle
LIBPATH_list	init	dev_jll	is_available	libecos_path
PATH	artifact_dir	<pre>find_artifact_dir</pre>	libecos	

Here you can see there is libecos, and more usefully for us, libecos\_path.

Once you know the name of the variable to override (the one that ends in \_path), use Preferences.jl to specify a new path:

```
using Preferences
set_preferences!(
 "LocalPreferences.toml",
 "ECOS_jll",
 "libecos_path" => "/tmp/jll_example/ecos/lib/libecos"
)
```

This will create a file in your current directory called LocalPreferences.toml with the contents:

```
[ECOS_jll]
libecos_path = "/tmp/jll_example/ecos/lib/libecos"
```

Now if you restart Julia, you will see:

```
julia> using ECOS_jll
```

```
julia> ECOS_jll.libecos
"/tmp/jll_example/ecos/lib/libecos"
```

To go back to using the default library, just delete the LocalPreferences.toml file.

#### **Overriding an entire artifact**

Sometimes a solver may provide a number of libraries and executables, and specifying the path for each of the becomes tedious. In this case, we can use Julia's Override.toml to replace an entire artifact.

Overriding an entire artifact requires you to replicate the structure and contents of the JLL package that we explored above.

In most cases you need only reproduce the include, lib, and bin directories (if they exist). You can safely ignore any logs or share directories. Take careful note of what files each directory contains and what they are called.

For our ECOS example, we already reproduced the structure when we compiled ECOS.

So, now we need to tell Julia to use our custom installation instead of the default. We can do this by making an override file at ~/.julia/artifacts/Overrides.toml.

Overrides.toml has the following content:

```
Override for ECOS_jll
2addb75332eff5a1657b46bb6bf30d2410bc7ecf = "/tmp/jll_example/ecos"
```

where 2addb75332eff5a1657b46bb6bf30d2410bc7ecf is the folder from the original ECOS\_jll.artifact\_dir and "/tmp/jll\_example/ecos" is the location of our new installation. Replace these as appropriate for your system.

If you restart Julia after creating the override file, you will see:

julia> using ECOS\_jll

```
julia> ECOS_jll.artifact_dir
"/tmp/jll_example/ecos"
```

Now when we use ECOS it will use our custom binary.

# Using Cbc with a custom binary

As a second example, we demonstrate how to use Cbc.jl with a custom binary.

#### Explore the JLL you want to override

First, let's check where Cbc\_jll is installed:

julia> using Cbc\_jll

julia> Cbc\_jll.artifact\_dir

```
"/Users/oscar/.julia/artifacts/e481bc81db5e229ba1f52b2b4bd57484204b1b06"
julia> readdir(Cbc_jll.artifact_dir)
5-element Vector{String}:
"bin"
 "include"
 "lib"
"logs"
"share"
julia> readdir(joinpath(Cbc_jll.artifact_dir, "bin"))
1-element Vector{String}:
"cbc"
julia> readdir(joinpath(Cbc_jll.artifact_dir, "lib"))
10-element Vector{String}:
"libCbc.3.10.5.dylib"
 "libCbc.3.dylib"
 "libCbc.dylib"
"libCbcSolver.3.10.5.dylib"
"libCbcSolver.3.dylib"
"libCbcSolver.dylib"
"lib0siCbc.3.10.5.dylib"
"libOsiCbc.3.dylib"
"libOsiCbc.dylib"
"pkgconfig"
```

#### Compile a custom binary

Next, we need to compile Cbc. Cbc can be difficult to compile (it has a lot of dependencies), but for macOS users there is a homebrew recipe:

```
(base) oscar@Oscars-MBP jll_example % brew install cbc
[... lines omitted ...]
(base) oscar@Oscars-MBP jll_example % brew list cbc
/usr/local/Cellar/cbc/2.10.5/bin/cbc
/usr/local/Cellar/cbc/2.10.5/lib/libCbc.3.10.5.dylib
/usr/local/Cellar/cbc/2.10.5/lib/libCbcSolver.3.10.5.dylib
/usr/local/Cellar/cbc/2.10.5/lib/libOsiCbc.3.10.5.dylib
/usr/local/Cellar/cbc/2.10.5/lib/libOsiCbc.3.10.5.dylib
/usr/local/Cellar/cbc/2.10.5/lib/libOsiCbc.3.10.5.dylib
/usr/local/Cellar/cbc/2.10.5/lib/libOsiCbc.3.10.5.dylib
/usr/local/Cellar/cbc/2.10.5/lib/libOsiCbc.3.10.5.dylib
/usr/local/Cellar/cbc/2.10.5/lib/ (6 other files)
/usr/local/Cellar/cbc/2.10.5/share/cbc/ (59 files)
/usr/local/Cellar/cbc/2.10.5/share/coin/ (4 files)
```

### **Override single libraries**

. . . . . . . . .

To use Preferences.jl to override specific libraries we first check the names of each library in Cbc\_jll:

julia> Cbc_jll.			
LIBPATH	cbc	<pre>get_libcbcsolver_path</pre>	libOsiCbc_path
LIBPATH_list	cbc_path	is_available	libcbcsolver

PATH	dev_jll	libCbc	libcbcsolver_handle
PATH_list	find_artifact_dir	libCbc_handle	libcbcsolver_path
init	get_cbc_path	libCbc_path	
artifact_dir	<pre>get_libCbc_path</pre>	libOsiCbc	
best_wrapper	get_lib0siCbc_path	lib0siCbc_handle	

Then we add the following to LocalPreferences.toml:

```
[Cbc_jll]
cbc_path = "/usr/local/Cellar/cbc/2.10.5/bin/cbc"
libCbc_path = "/usr/local/Cellar/cbc/2.10.5/lib/libCbc.3.10.5"
libOsiCbc_path = "/usr/local/Cellar/cbc/2.10.5/lib/libOsiCbc.3.10.5"
libcbcsolver_path = "/usr/local/Cellar/cbc/2.10.5/lib/libCbcSolver.3.10.5"
```

# Info

Note that capitalization matters, so libcbcsolver\_path corresponds to libCbcSolver.3.10.5.

# **Override entire artifact**

To use the homebrew install as our custom binary we add the following to ~/.julia/artifacts/0verrides.toml:

```
Override for Cbc_jll
e481bc81db5e229ba1f52b2b4bd57484204b1b06 = "/usr/local/Cellar/cbc/2.10.5"
```

# **Chapter 28**

# **Style Guide**

# 28.1 Style guide and design principles

# Style guide

This section describes the coding style rules that apply to JuMP code and that we recommend for JuMP models and surrounding Julia code. The motivations for a style guide include:

- conveying best practices for writing readable and maintainable code
- reducing the amount of time spent on bike-shedding by establishing basic naming and formatting conventions
- lowering the barrier for new contributors by codifying the existing practices (for example, you can be more confident your code will pass review if you follow the style guide)

In some cases, the JuMP style guide diverges from the Julia style guide. All such cases will be explicitly noted and justified.

The JuMP style guide adopts many recommendations from the Google style guides.

#### Info

The style guide is always a work in progress, and not all JuMP code follows the rules. When modifying JuMP, please fix the style violations of the surrounding code (that is, leave the code tidier than when you started). If large changes are needed, consider separating them into another PR.

## JuliaFormatter

JuMP uses JuliaFormatter.jl as an auto-formatting tool.

We use the options contained in .JuliaFormatter.toml.

To format code, cd to the JuMP directory, then run:

```
] add JuliaFormatter
using JuliaFormatter
format("docs")
format("src")
format("test")
```



The following sections outline extra style guide points that are not fixed automatically by JuliaFormatter.

#### Abstract types and composition

Specifying types for method arguments is mostly optional in Julia. The benefit of abstract method arguments is that it enables functions and types from one package to be used with functions and types from another package via multiple dispatch.

However, abstractly typed methods have two main drawbacks:

- It's possible to find out that you are working with unexpected types deep in the call chain, potentially leading to hard-to-diagnose MethodErrors.
- 2. Untyped function arguments can lead to correctness problems if the user's choice of input type does not satisfy the assumptions made by the author of the function.

As a motivating example, consider the following function:

```
julia> function my_sum(x)
 y = 0.0
 for i in 1:length(x)
 y += x[i]
 end
 return y
end
my_sum (generic function with 1 method)
```

This function contains a number of implicit assumptions about the type of x:

- x supports 1-based getindex and implements length
- The element type of x supports addition with 0.0, and then with the result of x + 0.0.

#### Info

As a motivating example for the second point, VariableRef plus Float64 produces an AffExpr. Do not assume that +(::A, ::B) produces an instance of the type A or B.

my\_sum works as expected if the user passes in Vector{Float64}:

```
julia> my_sum([1.0, 2.0, 3.0])
6.0
```

but it doesn't respect input types, for example returning a Float64 if the user passes Vector{Int}:

```
julia> my_sum([1, 2, 3])
6.0
```

but it throws a MethodError if the user passes String:

```
julia> my_sum("abc")
ERROR: MethodError: no method matching +(::Float64, ::Char)
[...]
```

This particular MethodError is hard to debug, particularly for new users, because it mentions +, Float64, and Char, none of which were called or passed by the user.

#### Dealing with MethodErrors

This section diverges from the Julia style guide, as well as other common guides like SciML. The following suggestions are intended to provide a friendlier experience for novice Julia programmers, at the cost of limiting the power and flexibility of advanced Julia programmers.

Code should follow the MethodError principle:

The MethodError principle

A user should see a MethodError only for methods that they called directly.

Bad:

```
_internal_function(x::Integer) = x + 1
The user sees a MethodError for _internal_function when calling
public_function("a string"). This is not very helpful.
public_function(x) = _internal_function(x)
```

Good:

```
_internal_function(x::Integer) = x + 1
The user sees a MethodError for public_function when calling
public_function("a string"). This is easy to understand.
public_function(x::Integer) = _internal_function(x)
```

If it is hard to provide an error message at the top of the call chain, then the following pattern is also ok:

```
_internal_function(x::Integer) = x + 1
function _internal_function(x)
 error(
 "Internal error. This probably means that you called " *
 "`public_function()`s with the wrong type.",
)
end
public_function(x) = _internal_function(x)
```

CHAPTER 28. STYLE GUIDE

#### **Dealing with correctness**

Dealing with correctness is harder, because Julia has no way of formally specifying interfaces that abstract types must implement. Instead, here are two options that you can use when writing and interacting with generic code:

#### Option 1: use concrete types and let users extend new methods.

In this option, explicitly restrict input arguments to concrete types that are tested and have been validated for correctness. For example:

```
6.0
```

Using concrete types satisfies the MethodError principle:

```
julia> my_sum_option_1("abc")
ERROR: MethodError: no method matching my_sum_option_1(::String)
```

and it allows other types to be supported in future by defining new methods:

```
julia> function my_sum_option_1(x::Array{T,N}) where {T<:Number,N}
 y = zero(T)
 for i in eachindex(x)
 y += x[i]
 end
 return y
 end
my sum option 1 (generic function with 2 methods)</pre>
```

Importantly, these methods do not have to be defined in the original package.

Info Some usage of abstract types is okay. For example, in my\_sum\_option\_1, we allowed the element type, T, to be a subtype of Number. This is fairly safe, but it still has an implicit assumption that T supports zero(T) and +(::T, ::T).

## Option 2: program defensively, and validate all assumptions.

An alternative is to program defensively, and to rigorously document and validate all assumptions that the code makes. In particular:

1. All assumptions on abstract types that aren't guaranteed by the definition of the abstract type (for example, optional methods without a fallback) should be documented.

- If practical, the assumptions should be checked in code, and informative error messages should be provided to the user if the assumptions are not met. In general, these checks may be expensive, so you should prefer to do this once, at the highest level of the call-chain.
- 3. Tests should cover for a range of corner cases and argument types.

```
For example:
```

```
.....
 test_my_sum_defensive_assumptions(x::AbstractArray{T}) where {T}
Test the assumptions made by `my_sum_defensive`.
function test_my_sum_defensive_assumptions(x::AbstractArray{T}) where {T}
 try
 # Some types may not define zero.
 @assert zero(T) isa T
 # Check iteration supported
 @assert iterate(x) isa Union{Nothing,Tuple{T,Int}}
 # Check that + is defined
 @assert +(zero(T), zero(T)) isa Any
 catch err
 error(
 "Unable to call my_sum_defensive(::$(typeof(x))) because " *
 "it failed an internal assumption",
)
 end
 return
end
.....
 my_sum_defensive(x::AbstractArray{T}) where {T}
Return the sum of the elements in the abstract array `x`.
Assumptions
This function makes the following assumptions:
* That `zero(T)` is defined
\ast That `x` supports the iteration interface
* That `+(::T, ::T)` is defined
....
function my_sum_defensive(x::AbstractArray{T}) where {T}
 test_my_sum_defensive_assumptions(x)
 y = zero(T)
 for xi in x
 y += xi
 end
 return y
end
output
my_sum_defensive
```

This function works on Vector{Float64}:

```
julia> my_sum_defensive([1.0, 2.0, 3.0])
6.0
```

as well as Matrix{Rational{Int}}:

```
julia> my_sum_defensive([(1//2) + (4//3)im; (6//5) + (7//11)im])
17//10 + 65//33*im
```

and it throws an error when the assumptions aren't met:

```
julia> my_sum_defensive(['a', 'b', 'c'])
ERROR: Unable to call my_sum_defensive(::Vector{Char}) because it failed an internal assumption
[...]
```

As an alternative, you may choose not to call test\_my\_sum\_defensive\_assumptions within my\_sum\_defensive, and instead ask users of my\_sum\_defensive to call it in their tests.

#### Juxtaposed multiplication

Only use juxtaposed multiplication when the right-hand side is a symbol.

Good:

2x # Acceptable if there are space constraints. 2 \* x # This is preferred if space is not an issue. 2 \* (x + 1)

#### Bad:

2(x + 1)

#### **Empty vectors**

For a type T, T[] and Vector{T}() are equivalent ways to create an empty vector with element type T. Prefer T[] because it is more concise.

#### Comments

For non-native speakers and for general clarity, comments in code must be proper English sentences with appropriate punctuation.

Good:

# This is a comment demonstrating a good comment.

#### Bad:

# a bad comment

#### JuMP macro syntax

For consistency, always use parentheses.

Good:

@variable(model, x >= 0)

#### Bad:

@variable model x >= 0

For consistency, always use constant \* variable as opposed to variable \* constant. This makes it easier to read models in ambiguous cases like a \* x.

#### Good:

a = 4
@constraint(model, 3 \* x <= 1)
@constraint(model, a \* x <= 1)</pre>

### Bad:

a = 4
@constraint(model, x \* 3 <= 1)
@constraint(model, x \* a <= 1)</pre>

In order to reduce boilerplate code, prefer the plural form of macros over lots of repeated calls to singular forms.

Good:

Bad:

```
@variable(model, x >= 0)
@variable(model, y >= 1)
@variable(model, z <= 2)</pre>
```

An exception is made for calls with many keyword arguments, since these need to be enclosed in parentheses in order to parse properly.

Acceptable:

```
@variable(model, x >= 0, start = 0.0, base_name = "my_x")
@variable(model, y >= 1, start = 2.0)
@variable(model, z <= 2, start = -1.0)</pre>
```

Also acceptable:

While we always use in for for-loops, it is acceptable to use = in the container declarations of JuMP macros.

Okay:

@variable(model, x[i=1:3])

Also okay:

@variable(model, x[i in 1:3])

#### Naming

```
module SomeModule end
function some_function end
const SOME_CONSTANT = ...
struct SomeStruct
 some_field::SomeType
end
@enum SomeEnum ENUM_VALUE_A ENUM_VALUE_B
some_local_variable = ...
some_file.jl # Except for ModuleName.jl.
```

### Exported and non-exported names

Begin private module level functions and constants with an underscore. All other objects in the scope of a module should be exported. (See JuMP.jl for an example of how to do this.)

Names beginning with an underscore should only be used for distinguishing between exported (public) and non-exported (private) objects. Therefore, never begin the name of a local variable with an underscore.

```
module MyModule
export public_function, PUBLIC_CONSTANT
```

```
function _private_function()
 local_variable = 1
 return
end
function public_function end
const _PRIVATE_CONSTANT = 3.14159
const PUBLIC_CONSTANT = 1.41421
end
```

#### Use of underscores within names

The Julia style guide recommends avoiding underscores "when readable," for example, haskey, isequal, remotecall, and remotecall\_fetch. This convention creates the potential for unnecessary bikeshedding and also forces the user to recall the presence/absence of an underscore, for example, "was that argument named basename or base\_name?". For consistency, *always use underscores* in variable names and function names to separate words.

#### Use of !

Julia has a convention of appending ! to a function name if the function modifies its arguments. We recommend to:

- Omit ! when the name itself makes it clear that modification is taking place, for example, add\_constraint and set\_name. We depart from the Julia style guide because ! does not provide a reader with any additional information in this case, and adherence to this convention is not uniform even in base Julia itself (consider Base.println and Base.finalize).
- Use ! in all other cases. In particular it can be used to distinguish between modifying and non-modifying variants of the same function like scale and scale!.

Note that ! is *not* a self-documenting feature because it is still ambiguous which arguments are modified when multiple arguments are present. Be sure to document which arguments are modified in the method's docstring.

See also the Julia style guide recommendations for ordering of function arguments.

#### Abbreviations

Abbreviate names to make the code more readable, not to save typing. Don't arbitrarily delete letters from a word to abbreviate it (for example, indx). Use abbreviations consistently within a body of code (for example, do not mix con and constr, idx and indx).

Common abbreviations:

- num for number
- con for constraint

#### No one-letter variable names

Where possible, avoid one-letter variable names.

Use model = Model() instead of m = Model()

Exceptions are made for indices in loops.

#### @enum vs. Symbol

The @enum macro lets you define types with a finite number of values that are explicitly enumerated (like enum in C/C++). Symbols are lightweight strings that are used to represent identifiers in Julia (for example, :x).

@enum provides type safety and can have docstrings attached to explain the possible values. Use @enums when applicable, for example, for reporting statuses. Use strings to provide long-form additional information like error messages.

Use of Symbol should typically be reserved for identifiers, for example, for lookup in the JuMP model (model[:my variable]).

#### using vs. import

using ModuleName brings all symbols exported by the module ModuleName into scope, while import ModuleName brings only the module itself into scope. (See the Julia manual) for examples and more details.

For the same reason that from <module> import \* is not recommended in python (PEP 8), avoid using ModuleName except in throw-away scripts or at the REPL. The using statement makes it harder to track where symbols come from and exposes the code to ambiguities when two modules export the same symbol.

Prefer using ModuleName: x, p to import ModuleName.x, ModuleName.p and import MyModule: x, p because the import versions allow method extension without qualifying with the module name.

Similarly, using ModuleName: ModuleName is an acceptable substitute for import ModuleName, because it does not bring all symbols exported by ModuleName into scope. However, we prefer import ModuleName for consistency.

#### Documentation

This section describes the writing style that should be used when writing documentation for JuMP (and supporting packages).

We can recommend the documentation style guides by Divio, Google, and Write the Docs as general reading for those writing documentation. This guide delegates a thorough handling of the topic to those guides and instead elaborates on the points more specific to Julia and documentation that use Documenter.

- Be concise
- · Use lists instead of long sentences
- Use numbered lists when describing a sequence, for example, (1) do X, (2) then Y
- · Use bullet points when the items are not ordered
- · Example code should be covered by doctests
- When a word is a Julia symbol and not an English word, enclose it with backticks. In addition, if it has
  a docstring in this doc add a link using @ref. If it is a plural, add the "s" after the closing backtick. For
  example,

```
[`VariableRef`](@ref)s
```

• Use @meta blocks for TODOs and other comments that shouldn't be visible to readers. For example,

```
```@meta
# TODO: Mention also X, Y, and Z.
```

Docstrings

- Every exported object needs a docstring
- All examples in docstrings should be jldoctests
- Always use complete English sentences with proper punctuation
- Do not terminate lists with punctuation (for example, as in this doc)

Here is an example:

```
"""
signature(args; kwargs...)
Short sentence describing the function.
Optional: add a slightly longer paragraph describing the function.
## Notes
- List any notes that the user should be aware of
## Example
```jldoctest
julia> 1 + 1
2
```
"""
```

Testing

Use a module to encapsulate tests, and structure all tests as functions. This avoids leaking local variables between tests.

Here is a basic skeleton:

module TestPkg
using Test
function runtests()

```
for name in names(@__MODULE__; all = true)
        if startswith("$(name)", "test_")
            @testset "$(name)" begin
                getfield(@__MODULE__, name)()
            end
        end
    end
    return
end
helper_function() = 2
function test_addition()
   @test 1 + 1 == _helper_function()
    return
end
end # module TestPkg
TestPkg.runtests()
```

Break the tests into multiple files, with one module per file, so that subsets of the codebase can be tested by calling include with the relevant file.

Chapter 29

Conventions

29.1 Conventions for interfacing between JuMP and MathOptInterface

The purpose of this guide is to document the conventions that we have developed for interfacing between JuMP and MathOptInterface.

Attributes

JuMP provides get_attribute and set_attribute as thin shims for MOI.get and MOI.set. However, there are two common cases where the thin shims are not sufficient:

- when the value of the attribute is an AbstractJuMPScalar that needs to be mapped to and from MOI.AbstractFunction
- when the value of the attribute depends on the shape of the constraint, for example, when getting MOI.ConstraintDualStart of matrix-valued constraints.

In these two cases, the convention is to keep get_attribute as a thin shim that does not modify the attribute value, and to develop new functions that modify or reshape the value as appropriate.

As an example, JuMP provides dual_start_value and set_dual_start_value to get and set the MOI.ConstraintDualStart in the original matrix shape, while get_attribute and set_attribute can be used to get and set the value in the vectorized shape:

```
julia> using JuMP
julia> model = Model();
julia> @variable(model, x[1:2, 1:2], PSD);
julia> c = VariableInSetRef(x);
julia> set_dual_start_value(c, [1 0; 0 1])
julia> dual_start_value(c)
2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
1.0 0.0
0.0 1.0
```

julia> get_attribute(c, MOI.ConstraintDualStart())

```
3-element Vector{Float64}:
    1.0
    0.0
    1.0

julia> set_attribute(c, MOI.ConstraintDualStart(), [2.0, -1.0, 1.0])

julia> dual_start_value(c)
2×2 LinearAlgebra.Symmetric{Float64, Matrix{Float64}}:
    2.0 -1.0
    -1.0 1.0
```

unset_ methods

There are a variety of attributes in JuMP and MOI that can be "set" and "unset." For example, there is MOI.Silent, and the corresponding set_silent and unset_silent.

Note how set_silent and unset_silent take a single argument (the model), where set_silent(model) corresponds to MOI.set(model, MOI.Silent(), true) and unset_silent(model) corresponds to MOI.set(model, MOI.Silent(), false). We could have instead implemented a single method set_silent(model, flag::Bool) that corresponded to MOI.set(model, MOI.Silent(), flag). Another example is unset_time_limit_sec, which is equivalent to set_time_limit_sec(model, nothing).

We have come to regard the unset_design as a mistake, because it leads to a proliferation of unique function names instead of leveraging Julia's strength for multiple dispatch.

The existing unset_names are retained for backwards compatibility, but, going forward, provide a single set_ method and document what value type should be provided to restore the model to the default setting. Thus, we have set_string_names_on_creation, but no corresponding unset_string_names_on_creation.

Chapter 30

Roadmap

30.1 Development roadmap

The JuMP developers have compiled this roadmap document to share their plans and goals with the JuMP community. Contributions to roadmap issues are especially invited.

Most of these issues will require changes to both JuMP and MathOptInterface, and are non-trivial in their implementation. They are in no particular order, but represent broad themes that we see as areas in which JuMP could be improved.

- Support nonlinear expressions with vector-valued inputs and outputs. There are a few related components:
 - Representing terms like log(det(X)) as necessary for Convex.jl
 - Automatic differentiation of terms with vector inputs and outputs
 - User-defined functions with vector-as opposed to scalar-inputs, which is particularly useful for optimal control problems
 - User-defined functions with vector outputs, avoiding the need for User-defined operators with vector outputs
- Add support for modeling with SI units. The UnitJuMP.jl extension is a good proof of concept for what this would look like. We want to make units a first-class concept in JuMP. See #1350 for more details.

Completed

- **Done #3106** Make nonlinear programming a first-class citizen. There have been many issues and discussions about this: currently nonlinear constraints are handled through a MOI.NLPBlock and have various limitations and restrictions.
 - https://github.com/jump-dev/JuMP.jl/issues/1185
 - https://github.com/jump-dev/JuMP.jl/issues/1198
 - https://github.com/jump-dev/JuMP.jl/issues/2788
 - https://github.com/jump-dev/MathOptInterface.jl/issues/846
 - https://github.com/jump-dev/MathOptInterface.jl/issues/1397
- **Done #3385** Add support for coefficient types other than Float64: https://github.com/jump-dev/JuMP.jl/issues/2025 Since the very beginning, JuMP has hard-coded the coefficient type as Float64. This has made it impossible to support solvers which can use other types such as BigFloat or Rational{BigInt}.

- **Done #3385** Add support for constraint programming: https://github.com/jump-dev/JuMP.jl/issues/2227 JuMP has a strong focus on linear, conic and nonlinear optimization problems. We want to add better support for constraint programming.
- **Done #3176** Add support for multiobjective problems: https://github.com/jump-dev/JuMP.jl/issues/2099 JuMP is restricted to problems with scalar-valued objectives. We want to extend this to vector-valued problems.
- **Done #3629** Refactor the internal code of JuMP's macros. The code in src/macros.jl is some of the oldest part of JuMP and is difficult to read, modify, and extend. We should overhaul the internals of JuMP's macros--without making user-visible breaking changes--to improve their long-term maintainability.

Chapter 31

Checklists

31.1 Checklists

The purpose of this page is to collate a series of checklists for commonly performed changes to the source code of JuMP.

In each case, copy the checklist into the description of the pull request.

Making a release

In preparation for a release, use the following checklist. These steps can be done in the same commit, or separately. The last commit should have the message "Prep for vX.Y.Z."

Pre-release

- [] Check that the pinned packages in `docs/Project.toml` are updated. We pin the versions so that changes in the solvers (changes in printing, small numeric changes) do not break the printing of the JuMP docs in arbitrary commits.
- [] Check that the `rev` fields in `docs/packages.toml` are updated. We pin the versions of solvers and extensions to ensure that changes to their READMEs do not break the JuMP docs in arbitrary commits, and to ensure that the versions are compatible with the latest JuMP and MathOptInterface releases.
- [] Check compat of `DimensionalData` in `Project.toml`
- [] Check compat of `MacroTools` in `Project.toml`
- [] Update `docs/src/changelog.md`
- [] Run https://github.com/jump-dev/JuMP.jl/actions/workflows/extension-tests.yml using a `workflow_dispatch` trigger to check for any changes in JuMP that broke extensions.
- [] Change the version number in `Project.toml`
- [] The commit messages in this PR do not contain `[ci skip]`
- ## The release
- [] After merging this pull request, comment `[at]JuliaRegistrator register` in the GitHub commit. This should automatically publish a new version to the Julia registry, as well as create a tag, and rebuild the documentation for this tag.

These steps can take quite a bit of time (1 hour or more), so don't be

surprised if the new documentation takes a while to appear. In addition, the links in the README will be broken until JuliaHub fetches the new version on their servers.

```
## Post-release
```

- [] Once the tag is created, update the relevant `release-` branch. The latest release branch at the time of writing is `release-1.0` (we haven't back-ported any patches that needed to create a `release-1.Y` branch). To to update the release branch with the v1.10.0 tag, do: ``` git checkout release-1.0 git pull git merge v1.10.0 git push

Adding a new solver to the documentation

Use the following checklist when adding a new solver to the JuMP documentation.

Basic

- [] Check that the solver is a registered Julia package
- [] Check that the solver supports the long-term support release of Julia
- [] Check that the solver has a MathOptInterface wrapper
- [] Check that the tests call `MOI.Test.runtests`. Some test excludes are permissible, but the reason for skipping a particular test should be documented.
- [] Check that the README and/or documentation provides an example of how to use the solver with JuMP

Documentation

- [] Add a new row to the table in `docs/src/installation.md`

Optional

- [] Add package metadata to `docs/packages.toml`

Adding a new shape

Use the following checklist when adding a new AbstractShape

Basic

- [] Add a new subtype of `AbstractShape`
- [] Implement `vectorize(data, ::NewShape)::Vector`
- [] Implement `reshape_vector(vector, ::NewShape)`
- [] Implement `dual_shape`, or verify that the shape is self-dual
- [] Add the tests from https://github.com/jump-dev/JuMP.jl/pull/3816

Part VII

MathOptInterface

Chapter 32

Introduction

32.1 Introduction

Warning

This documentation in this section is a copy of the official MathOptInterface documentation available at https://jump.dev/MathOptInterface.jl/v1.40.1. It is included here to make it easier to link concepts between JuMP and MathOptInterface.

What is MathOptInterface?

MathOptInterface.jl (MOI) is an abstraction layer designed to provide a unified interface to mathematical optimization solvers so that users do not need to understand multiple solver-specific APIs.

Тір

This documentation is aimed at developers writing software interfaces to solvers and modeling languages using the MathOptInterface API. If you are a user interested in solving optimization problems, we encourage you instead to use MOI through a higher-level modeling interface like JuMP or Convex.jl.

How the documentation is structured

Having a high-level overview of how this documentation is structured will help you know where to look for certain things.

- The **Tutorials** section contains articles on how to use and implement the MathOptInteraface API. Look here if you want to write a model in MOI, or write an interface to a new solver.
- The Manual contains short code-snippets that explain how to use the MOI API. Look here for more details on particular areas of MOI.
- The **Background** section contains articles on the theory behind MathOptInterface. Look here if you want to understand *why*, rather than *how*.
- The **API Reference** contains a complete list of functions and types that comprise the MOI API. Look here if you want to know how to use (or implement) a particular function.
- The **Submodules** section contains stand-alone documentation for each of the submodules within MOI. These submodules are not required to interface a solver with MOI, but they make the job much easier.

Citing MathOptInterface

If you find MathOptInterface useful in your work, we kindly request that you cite the following paper:

```
@article{legat2021mathoptinterface,
    title={{MathOptInterface}: a data structure for mathematical optimization problems},
    author={Legat, Beno{\^\i}t and Dowson, Oscar and Garcia, Joaquim Dias and Lubin, Miles},
    journal={INFORMS Journal on Computing},
    year={2021},
    doi={10.1287/ijoc.2021.1067},
    publisher={INFORMS}
}
```

A preprint of this paper is freely available.

32.2 Motivation

MathOptInterface (MOI) is a replacement for MathProgBase, the first-generation abstraction layer for mathematical optimization previously used by JuMP and Convex.jl.

To address a number of limitations of MathProgBase, MOI is designed to:

- Be simple and extensible
 - unifying linear, quadratic, and conic optimization,
 - seamlessly facilitating extensions to essentially arbitrary constraints and functions (for example, indicator constraints, complementarity constraints, and piecewise-linear functions)
- Be fast
 - by allowing access to a solver's in-memory representation of a problem without writing intermediate files (when possible)
 - by using multiple dispatch and avoiding requiring containers of non-concrete types
- · Allow a solver to return multiple results (for example, a pool of solutions)
- Allow a solver to return extra arbitrary information via attributes (for example, variable- and constraintwise membership in an irreducible inconsistent subset for infeasibility analysis)
- Provide a greatly expanded set of status codes explaining what happened during the optimization procedure
- · Enable a solver to more precisely specify which problem classes it supports
- Enable both primal and dual warm starts
- Enable adding and removing both variables and constraints by indices that are not required to be consecutive
- · Enable any modification that the solver supports to an existing model
- Avoid requiring the solver wrapper to store an additional copy of the problem data
Chapter 33

Tutorials

33.1 Solving a problem using MathOptInterface

In this tutorial we demonstrate how to use MathOptInterface to solve the binary-constrained knapsack problem:

$$\max c^{\top} x$$

s.t. $w^{\top} x \le C$
 $x_i \in \{0, 1\}, \quad \forall i = 1, \dots, n$

Required packages

Load the MathOptInterface module and define the shorthand MOI:

import MathOptInterface as MOI

As an optimizer, we choose GLPK:

using GLPK
optimizer = GLPK.Optimizer()

Define the data

We first define the constants of the problem:

```
julia> c = [1.0, 2.0, 3.0]
3-element Vector{Float64}:
    1.0
    2.0
    3.0

julia> w = [0.3, 0.5, 1.0]
3-element Vector{Float64}:
    0.3
    0.5
    1.0
```

julia> C = 3.2 3.2

Add the variables

julia> x = MOI.add_variables(optimizer, length(c));

Set the objective

julia> MOI.set(optimizer, MOI.ObjectiveSense(), MOI.MAX_SENSE)

Тір

```
MOI.ScalarAffineTerm.(c, x) is a shortcut for [MOI.ScalarAffineTerm(c[i], x[i]) for i = 1:3]. This is Julia's broadcast syntax in action, and is used quite often throughout MOI.
```

Add the constraints

We add the knapsack constraint and integrality constraints:

```
julia> MOI.add_constraint(
    optimizer,
    MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(w, x), 0.0),
    MOI.LessThan(C),
);
```

Add integrality constraints:

```
julia> for x_i in x
MOI.add_constraint(optimizer, x_i, MOI.ZeroOne())
end
```

Optimize the model

julia> MOI.optimize!(optimizer)

Understand why the solver stopped

The first thing to check after optimization is why the solver stopped, for example, did it stop because of a time limit or did it stop because it found the optimal solution?

```
julia> MOI.get(optimizer, MOI.TerminationStatus())
OPTIMAL::TerminationStatusCode = 1
```

Looks like we found an optimal solution.

Understand what solution was returned

```
julia> MOI.get(optimizer, MOI.ResultCount())
1
```

julia> MOI.get(optimizer, MOI.PrimalStatus())
FEASIBLE POINT::ResultStatusCode = 1

julia> MOI.get(optimizer, MOI.DualStatus())
N0 SOLUTION::ResultStatusCode = 0

Query the objective

```
What is its objective value?
```

```
julia> MOI.get(optimizer, MOI.ObjectiveValue())
6.0
```

Query the primal solution

And what is the value of the variables x?

```
julia> MOI.get(optimizer, MOI.VariablePrimal(), x)
3-element Vector{Float64}:
    1.0
    1.0
    1.0
```

33.2 Implementing a solver interface

This guide outlines the basic steps to implement an interface to MathOptInterface for a new solver.

Danger

Implementing an interface to MathOptInterface for a new solver is a lot of work. Before starting, we recommend that you join the Developer chatroom and explain a little bit about the solver you are wrapping. If you have questions that are not answered by this guide, please ask them in the Developer chatroom so we can improve this guide.

A note on the API

The API of MathOptInterface is large and varied. In order to support the diversity of solvers and use-cases, we make heavy use of duck-typing. That is, solvers are not expected to implement the full API, nor is there a well-defined minimal subset of what must be implemented. Instead, you should implement the API as necessary to make the solver function as you require.

The main reason for using duck-typing is that solvers work in different ways and target different use-cases.

For example:

- Some solvers support incremental problem construction, support modification after a solve, and have native support for things like variable names.
- Other solvers are "one-shot" solvers that require all of the problem data to construct and solve the problem in a single function call. They do not support modification or things like variable names.
- Other "solvers" are not solvers at all, but things like file readers. These may only support functions like read_from_file, and may not even support the ability to add variables or constraints directly.
- Finally, some "solvers" are layers which take a problem as input, transform it according to some rules, and pass the transformed problem to an inner solver.

Preliminaries

Before starting on your wrapper, you should do some background research and make the solver accessible via Julia.

Decide if MathOptInterface is right for you

The first step in writing a wrapper is to decide whether implementing an interface is the right thing to do.

MathOptInterface is an abstraction layer for unifying *constrained* mathematical optimization solvers. If your solver doesn't fit in the category, for example, it implements a derivative-free algorithm for unconstrained objective functions, MathOptInterface may not be the right tool for the job.



Find a similar solver already wrapped

The next step is to find (if possible) a similar solver that is already wrapped. Although not strictly necessary, this will be a good place to look for inspiration when implementing your wrapper.

The JuMP documentation has a good list of solvers, along with the problem classes they support.

Tip

If you're not sure which solver is most similar, ask in the Developer chatroom.

Create a low-level interface

Before writing a MathOptInterface wrapper, you first need to be able to call the solver from Julia.

Wrapping solvers written in Julia

If your solver is written in Julia, there's nothing to do here. Go to the next section.

Wrapping solvers written in C

Julia is well suited to wrapping solvers written in C.

| Info | | |
|---------------------------------------|--|--|
| This is <i>not</i> true
interface. | for C++. If you have a solver written in C++, first write a C interface, then wrap the C | |

Before writing a MathOptInterface wrapper, there are a few extra steps.

Create a JLL

If the C code is publicly available under an open source license, create a JLL package via Yggdrasil. The easiest way to do this is to copy an existing solver. Good examples to follow are the COIN-OR solvers.

Warning

Building the solver via Yggdrasil is non-trivial. Please ask the Developer chatroom for help.

If the code is commercial or not publicly available, the user will need to manually install the solver. See Gurobi.jl or CPLEX.jl for examples of how to structure this.

Use Clang.jl to wrap the C API

The next step is to use Clang.jl to automatically wrap the C API. The easiest way to do this is to follow an example. Good examples to follow are Cbc.jl and HiGHS.jl.

Sometimes, you will need to make manual modifications to the resulting files.

Solvers written in other languages

Ask the Developer chatroom for advice. You may be able to use one of the JuliaInterop packages to call out to the solver.

For example, SeDuMi.jl uses MATLAB.jl to call the SeDuMi solver written in MATLAB.

Structuring the package

Structure your wrapper as a Julia package. Consult the Julia documentation if you haven't done this before.

MOI solver interfaces may be in the same package as the solver itself (either the C wrapper if the solver is accessible through C, or the Julia code if the solver is written in Julia, for example), or in a separate package which depends on the solver package.

Note

The JuMP core contributors request that you do not use "JuMP" in the name of your package without prior consent.

Your package should have the following structure:

```
/.github
    /workflows
        ci.yml
        format_check.yml
        TagBot.yml
/gen
    gen.jl # Code to wrap the C API
/src
   NewSolver.jl
   /gen
        libnewsolver api.jl
        libnewsolver_common.jl
    /MOI_wrapper
        MOI_wrapper.jl
        other_files.jl
/test
    runtests.jl
    /MOI_wrapper
        MOI_wrapper.jl
.gitignore
.JuliaFormatter.toml
README.md
LICENSE.md
Project.toml
```

- The /.github folder contains the scripts for GitHub actions. The easiest way to write these is to copy the ones from an existing solver.
- The /gen and /src/gen folders are only needed if you are wrapping a solver written in C.
- The /src/MOI_wrapper folder contains the Julia code for the MOI wrapper.
- The /test folder contains code for testing your package. See Setup tests for more information.
- The .JuliaFormatter.toml and .github/workflows/format_check.yml enforce code formatting using JuliaFormatter.jl. Check existing solvers or JuMP.jl for details.

Documentation

Your package must include documentation explaining how to use the package. The easiest approach is to include documentation in your README.md. A more involved option is to use Documenter.jl.

Examples of packages with README-based documentation include:

- Cbc.jl
- HiGHS.jl
- SCS.jl

Examples of packages with Documenter-based documentation include:

- Alpine.jl
- COSMO.jl
- Juniper.jl

Setup tests

Tip

The best way to implement an interface to MathOptInterface is via test-driven development.

The MOI.Test submodule contains a large test suite to help check that you have implemented things correctly. Follow the guide How to test a solver to set up the tests for your package.

Run the tests frequently when developing. However, at the start there is going to be a lot of errors. Start by excluding large classes of tests (for example, exclude = ["test_basic_", "test_model_"], implement any missing methods until the tests pass, then remove an exclusion and repeat.

Initial code

By this point, you should have a package setup with tests, formatting, and access to the underlying solver. Now it's time to start writing the wrapper.

The Optimizer object

The first object to create is a subtype of AbstractOptimizer. This type is going to store everything related to the problem.

By convention, these optimizers should not be exported and should be named PackageName.Optimizer.

```
import MathOptInterface as MOI
struct Optimizer <: MOI.AbstractOptimizer
    # Fields go here
end</pre>
```

Optimizer objects for C solvers

Warning

This section is important if you wrap a solver written in C.

Wrapping a solver written in C will require the use of pointers, and for you to manually free the solver's memory when the Optimizer is garbage collected by Julia.

Never pass a pointer directly to a Julia ccall function.

Instead, store the pointer as a field in your Optimizer, and implement Base.cconvert and Base.unsafe_convert. Then you can pass Optimizer to any ccall function that expects the pointer.

In addition, make sure you implement a finalizer for each model you create.

If newsolver_createProblem() is the low-level function that creates the problem pointer in C, and newsolver_freeProblem(::Pt is the low-level function that frees memory associated with the pointer, your Optimizer() function should look like this:

```
struct Optimizer <: MOI.AbstractOptimizer
    ptr::Ptr{Cvoid}

function Optimizer()
    ptr = newsolver_createProblem()
    model = Optimizer(ptr)
    finalizer(model) do m
        newsolver_freeProblem(m)
        return
    end
    return model
end
Base.cconvert(::Type{Ptr{Cvoid}}, model::Optimizer) = model.ptr</pre>
```

Implement methods for Optimizer

All Optimizers must implement the following methods:

- empty!
- is_empty

Other methods, detailed below, are optional or depend on how you implement the interface.

Tip

For this and all future methods, read the docstrings to understand what each method does, what it expects as input, and what it produces as output. If it isn't clear, let us know and we will improve the docstrings. It is also very helpful to look at an existing wrapper for a similar solver.

You should also implement Base.summary(:::I0, ::Optimizer) to print a nice string when someone shows your model. For example

```
function Base.summary(io::I0, model::Optimizer)
    return print(io, "NewSolver with the pointer $(model.ptr)")
end
```

Implement attributes

MathOptInterface uses attributes to manage different aspects of the problem.

For each attribute

- get gets the current value of the attribute
- set sets a new value of the attribute. Not all attributes can be set. For example, the user can't modify the SolverName.

• supports returns a Bool indicating whether the solver supports the attribute.

Info

Use attribute_value_type to check the value expected by a given attribute. You should make sure that your get function correctly infers to this type (or a subtype of it).

Each column in the table indicates whether you need to implement the particular method for each attribute.

| Attribute | get | set | supports |
|-----------------------|-----|-----|----------|
| SolverName | Yes | No | No |
| SolverVersion | Yes | No | No |
| RawSolver | Yes | No | No |
| Name | Yes | Yes | Yes |
| Silent | Yes | Yes | Yes |
| TimeLimitSec | Yes | Yes | Yes |
| ObjectiveLimit | Yes | Yes | Yes |
| SolutionLimit | Yes | Yes | Yes |
| NodeLimit | Yes | Yes | Yes |
| Raw0ptimizerAttribute | Yes | Yes | Yes |
| Number0fThreads | Yes | Yes | Yes |
| AbsoluteGapTolerance | Yes | Yes | Yes |
| RelativeGapTolerance | Yes | Yes | Yes |

For example:

MOI.supports(::Optimizer, ::MOI.Silent) = true

Define supports_constraint

The next step is to define which constraints and objective functions you plan to support.

For each function-set constraint pair, define supports_constraint:

```
function MOI.supports_constraint(
    ::Optimizer,
    ::Type{MOI.VariableIndex},
    ::Type{MOI.ZeroOne},
```

) return true end

To make this easier, you may want to use Unions:

```
function MOI.supports_constraint(
    ::Optimizer,
    ::Type{MOI.VariableIndex},
    ::Type{<:Union{MOI.LessThan,MOI.GreaterThan,MOI.EqualTo}},
)
    return true
end</pre>
```

Тір

Only support a constraint if your solver has native support for it.

The big decision: incremental modification?

Now you need to decide whether to support incremental modification or not.

Incremental modification means that the user can add variables and constraints one-by-one without needing to rebuild the entire problem, and they can modify the problem data after an optimize! call. Supporting incremental modification means implementing functions like add_variable and add_constraint.

The alternative is to accept the problem data in a single optimize! or copy_to function call. Because these functions see all of the data at once, it can typically call a more efficient function to load data into the underlying solver.

Good examples of solvers supporting incremental modification are MILP solvers like GLPK.jl and Gurobi.jl. Examples of non-incremental solvers are AmpINLWriter.jl and SCS.jl

It is possible for a solver to implement both approaches, but you should probably start with one for simplicity.



In general, supporting incremental modification is more work, and it usually requires some extra book-keeping. However, it provides a more efficient interface to the solver if the problem is going to be resolved multiple times with small modifications. Moreover, once you've implemented incremental modification, it's usually not much extra work to add a copy_to interface. The converse is not true.

Tip

If this is your first time writing an interface, start with the one-shot optimize!.

The non-incremental interface

There are two ways to implement the non-incremental interface. The first uses a two-argument version of optimize!, the second implements copy_to followed by the one-argument version of optimize!.

If your solver does not support modification, and requires all data to solve the problem in a single function call, you should implement the "one-shot" optimize!.

• optimize!(::ModelLike, ::ModelLike)

If your solver separates data loading and the actual optimization into separate steps, implement the copy_to interface.

- copy_to(::ModelLike, ::ModelLike)
- optimize!(::ModelLike)

The incremental interface

Warning

Writing this interface is a lot of work. The easiest way is to consult the source code of a similar solver.

To implement the incremental interface, implement the following functions:

- add_variable
- add_variables
- add_constraint
- add_constraints
- is_valid
- delete
- optimize!(::ModelLike)

Info

Solvers do not have to support AbstractScalarFunction in GreaterThan, LessThan, EqualTo, or Interval with a nonzero constant in the function. Throw ScalarFunctionConstantNotZero if the function constant is not zero.

In addition, you should implement the following model attributes:

| Attribute | get | set | supports |
|----------------------------------|-----|-----|----------|
| ListOfModelAttributesSet | Yes | No | No |
| <pre>ObjectiveFunctionType</pre> | Yes | No | No |
| ObjectiveFunction | Yes | Yes | Yes |
| ObjectiveSense | Yes | Yes | Yes |
| Name | Yes | Yes | Yes |

Variable-related attributes:

Constraint-related attributes:

| Attribute | get | set | supports |
|-----------------------------------|-----|-----|----------|
| ListOfVariableAttributesSet | Yes | No | No |
| ListOfVariablesWithAttributeSet | Yes | No | No |
| Number0fVariables | Yes | No | No |
| ListOfVariableIndices | Yes | No | No |
| | | | |
| Attribute | get | set | supports |
| ListOfConstraintAttributesSet | Yes | No | No |
| ListOfConstraintsWithAttributeSet | Yes | No | No |
| Number0fConstraints | Yes | No | No |
| ListOfConstraintTypesPresent | Yes | No | No |
| ConstraintFunction | Yes | Yes | No |
| ConstraintSet | Yes | Yes | No |

Modifications

If your solver supports modifying data in-place, implement modify for the following AbstractModifications:

- ScalarConstantChange
- ScalarCoefficientChange
- ScalarQuadraticCoefficientChange
- VectorConstantChange
- MultirowChange

Variables constrained on creation

Some solvers require variables be associated with a set *when they are created*. This conflicts with the incremental modification approach, since you cannot first add a free variable and then constrain it to the set.

If this is the case, implement:

- add_constrained_variable
- add_constrained_variables
- supports_add_constrained_variables

By default, MathOptInterface assumes solvers support free variables. If your solver does not support free variables, define:

```
MOI.supports_add_constrained_variables(::Optimizer, ::Type{Reals}) = false
```

Incremental and copy_to

If you implement the incremental interface, you have the option of also implementing copy_to.

If you don't want to implement copy_to, for example, because the solver has no API for building the problem in a single function call, define the following fallback:

```
MOI.supports_incremental_interface(::Optimizer) = true
function MOI.copy_to(dest::Optimizer, src::MOI.ModelLike)
    return MOI.Utilities.default_copy_to(dest, src)
end
```

Names

Regardless of which interface you implement, you have the option of implementing the Name attribute for variables and constraints:

| Attribute | get | set | supports |
|----------------|-----|-----|----------|
| VariableName | Yes | Yes | Yes |
| ConstraintName | Yes | Yes | Yes |

If you implement names, you must also implement the following three methods:

```
function MOI.get(model::Optimizer, ::Type{MOI.VariableIndex}, name::String)
    return # The variable named `name`.
end
function MOI.get(model::Optimizer, ::Type{MOI.ConstraintIndex}, name::String)
    return # The constraint any type named `name`.
end
function MOI.get(
    model::Optimizer,
    ::Type{MOI.ConstraintIndex{F,S}},
    name::String,
) where {F,S}
    return # The constraint of type F-in-S named `name`.
end
```

These methods have the following rules:

- · If there is no variable or constraint with the name, return nothing
- If there is a single variable or constraint with that name, return the variable or constraint
- If there are multiple variables or constraints with the name, throw an error.

Warning

You should *not* implement ConstraintName for VariableIndex constraints. If you implement ConstraintName for other constraints, you can add the following two methods to disable ConstraintName for VariableIndex constraints.

```
function MOI.supports(
    ::Optimizer,
    ::MOI.ConstraintName,
    ::Type{<:MOI.ConstraintIndex{MOI.VariableIndex,<:MOI.AbstractScalarSet}},
)
    return throw(MOI.VariableIndexConstraintNameError())
end
function MOI.set(
    ::Optimizer,
    ::MOI.ConstraintName,
    ::MOI.ConstraintIndex{MOI.VariableIndex,<:MOI.AbstractScalarSet},
    ::String,
)
    return throw(MOI.VariableIndexConstraintNameError())
end</pre>
```

Solutions

Implement optimize! to solve the model:

• optimize!

All Optimizers must implement the following attributes:

- DualStatus
- PrimalStatus
- RawStatusString
- ResultCount
- TerminationStatus

Info

You only need to implement get for solution attributes. Don't implement set or supports.

Note

Solver wrappers should document how the low-level statuses map to the MOI statuses. Statuses like NEARLY_FEASIBLE_POINT and INFEASIBLE_POINT, are designed to be used when the solver explicitly indicates that relaxed tolerances are satisfied or the returned point is infeasible, respectively.

You should also implement the following attributes:

- ObjectiveValue
- SolveTimeSec
- VariablePrimal

Тір

Attributes like VariablePrimal and ObjectiveValue are indexed by the result count. Use MOI.check_result_index_bounds(model, attr) to throw an error if the attribute is not available.

If your solver returns dual solutions, implement:

- ConstraintDual
- DualObjectiveValue

For integer solvers, implement:

- ObjectiveBound
- RelativeGap

If applicable, implement:

- SimplexIterations
- BarrierIterations
- NodeCount

If your solver uses the Simplex method, implement:

• ConstraintBasisStatus

If your solver accepts primal or dual warm-starts, implement:

- VariablePrimalStart
- ConstraintDualStart

Other tips

Here are some other points to be aware of when writing your wrapper.

Unsupported constraints at runtime

In some cases, your solver may support a particular type of constraint (for example, quadratic constraints), but only if the data meets some condition (for example, it is convex).

In this case, declare that you support the constraint, and throw AddConstraintNotAllowed.

Dealing with multiple variable bounds

MathOptInterface uses VariableIndex constraints to represent variable bounds. Defining multiple variable bounds on a single variable is not allowed.

Throw LowerBoundAlreadySet or UpperBoundAlreadySet if the user adds a constraint that results in multiple bounds.

Only throw if the constraints conflict. It is okay to add VariableIndex-in-GreaterThan and then VariableIndex-in-LessThan, but not VariableIndex-in-Interval and then VariableIndex-in-LessThan,

Expect duplicate coefficients

Solvers must expect that functions such as ScalarAffineFunction and VectorQuadraticFunction may contain duplicate coefficients.

For example, ScalarAffineFunction([ScalarAffineTerm(x, 1), ScalarAffineTerm(x, 1)], 0.0).

Use Utilities.canonical to return a new function with the duplicate coefficients aggregated together.

Don't modify user-data

All data passed to the solver must be copied immediately to internal data structures. Solvers may not modify any input vectors and must assume that input vectors will not be modified by users in the future.

This applies, for example, to the terms vector in ScalarAffineFunction. Vectors returned to the user, for example, via ObjectiveFunction or ConstraintFunction attributes, must not be modified by the solver afterwards. The in-place version of get! can be used by users to avoid extra copies in this case.

Column Generation

There is no special interface for column generation. If the solver has a special API for setting coefficients in existing constraints when adding a new variable, it is possible to queue modifications and new variables and then call the solver's API once all of the new coefficients are known.

Solver-specific attributes

You don't need to restrict yourself to the attributes defined in the MathOptInterface.jl package.

Solver-specific attributes should be specified by creating an appropriate subtype of AbstractModelAttribute, AbstractOptimizerAttribute, AbstractVariableAttribute, or AbstractConstraintAttribute.

For example, Gurobi.jl adds attributes for multiobjective optimization by defining:

```
struct NumberOfObjectives <: MOI.AbstractModelAttribute end
function MOI.set(model::Optimizer, ::NumberOfObjectives, n::Integer)
    # Code to set NumberOfObjectives
    return
end
function MOI.get(model::Optimizer, ::NumberOfObjectives)
    n = # Code to get NumberOfObjectives
    return n
end</pre>
```

Then, the user can write:

```
model = Gurobi.Optimizer()
MOI.set(model, Gurobi.NumberofObjectives(), 3)
```

33.3 Transitioning from MathProgBase

MathOptInterface is a replacement for MathProgBase.jl. However, it is not a direct replacement.

Transitioning a solver interface

MathOptInterface is more extensive than MathProgBase which may make its implementation seem daunting at first. There are however numerous utilities in MathOptInterface that the simplify implementation process.

For more information, read Implementing a solver interface.

Transitioning the high-level functions

MathOptInterface doesn't provide replacements for the high-level interfaces in MathProgBase. We recommend you use JuMP as a modeling interface instead.

Tip

```
If you haven't used JuMP before, start with the tutorial Getting started with JuMP
```

linprog

Here is one way of transitioning from linprog:

```
using JuMP
```

```
function linprog(c, A, sense, b, l, u, solver)
   N = length(c)
   model = Model(solver)
   @variable(model, l[i] <= x[i=1:N] <= u[i])</pre>
   @objective(model, Min, c' * x)
   eq_rows, ge_rows, le_rows = sense .== '=', sense .== '>', sense .== '<'
   @constraint(model, A[eq_rows, :] * x .== b[eq_rows])
   @constraint(model, A[ge_rows, :] * x .>= b[ge_rows])
   @constraint(model, A[le_rows, :] * x .<= b[le_rows])</pre>
   optimize!(model)
   return (
        status = termination_status(model),
        objval = objective_value(model),
        sol = value.(x)
    )
end
```

mixintprog

Here is one way of transitioning from mixintprog:

using JuMP

```
function mixintprog(c, A, rowlb, rowub, vartypes, lb, ub, solver)
   N = length(c)
   model = Model(solver)
   @variable(model, lb[i] <= x[i=1:N] <= ub[i])</pre>
    for i in 1:N
        if vartypes[i] == :Bin
            set_binary(x[i])
        elseif vartypes[i] == :Int
            set_integer(x[i])
        end
   end
   @objective(model, Min, c' * x)
   @constraint(model, rowlb .<= A * x .<= rowub)</pre>
   optimize!(model)
    return (
        status = termination_status(model),
        objval = objective_value(model),
        sol = value.(x)
    )
end
```

quadprog

using JuMP

Here is one way of transitioning from quadprog:

```
function quadprog(c, Q, A, rowlb, rowub, lb, ub, solver)
N = length(c)
model = Model(solver)
@variable(model, lb[i] <= x[i=1:N] <= ub[i])
@objective(model, Min, c' * x + 0.5 * x' * Q * x)
@constraint(model, rowlb .<= A * x .<= rowub)
optimize!(model)
return (
status = termination_status(model),
objval = objective_value(model),
sol = value.(x)
)
end</pre>
```

33.4 Implementing a constraint bridge

This guide outlines the basic steps to create a new bridge from a constraint expressed in the formalism Function-in-Set.

Preliminaries

First, decide on the set you want to bridge. Then, study its properties: the most important one is whether the set is scalar or vector, which impacts the dimensionality of the functions that can be used with the set.

- A scalar function only has one dimension. MOI defines three types of scalar functions: a variable (VariableIndex), an affine function (ScalarAffineFunction), or a quadratic function (ScalarQuadraticFunction).
- A vector function has several dimensions (at least one). MOI defines three types of vector functions: several variables (VectorOfVariables), an affine function (VectorAffineFunction), or a quadratic function (VectorQuadraticFunction). The main difference with scalar functions is that the order of dimensions can be very important: for instance, in an indicator constraint (Indicator), the first dimension indicates whether the constraint about the second dimension is active.

To explain how to implement a bridge, we present the example of Bridges.Constraint.FlipSignBridge. This bridge maps <= (LessThan) constraints to >= (GreaterThan) constraints. This corresponds to reversing the sign of the inequality. We focus on scalar affine functions (we disregard the cases of a single variable or of quadratic functions). This example is a simplified version of the code included in MOI.

Four mandatory parts in a constraint bridge

The first part of a constraint bridge is a new concrete subtype of Bridges.Constraint.AbstractBridge. This type must have fields to store all the new variables and constraints that the bridge will add. Typically, these types are parametrized by the type of the coefficients in the model.

Then, three sets of functions must be defined:

- 1. Bridges.Constraint.bridge_constraint: this function implements the bridge and creates the required variables and constraints.
- supports_constraint: these functions must return true when the combination of function and set is supported by the bridge. By default, the base implementation always returns false and the bridge does not have to provide this implementation.
- Bridges.added_constrained_variable_types and Bridges.added_constraint_types: these functions return the types of variables and constraints that this bridge adds. They are used to compute the set of other bridges that are required to use the one you are defining, if need be.

More functions can be implemented, for instance to retrieve properties from the bridge or deleting a bridged constraint.

1. Structure for the bridge

A typical struct behind a bridge depends on the type of the coefficients that are used for the model (typically Float64, but coefficients might also be integers or complex numbers).

This structure must hold a reference to all the variables and the constraints that are created as part of the bridge.

The type of this structure is used throughout MOI as an identifier for the bridge. It is passed as argument to most functions related to bridges.

The best practice is to have the name of this type end with Bridge.

In our example, the bridge maps any ScalarAffineFunction{T}-in-LessThan{T} constraint to a single ScalarAffineFunction{T in-GreaterThan{T} constraint. The affine function has coefficients of type T. The bridge is parametrized with T, so that the constraint that the bridge creates also has coefficients of type T.

```
struct SignBridge{T<:Number} <: Bridges.Constraint.AbstractBridge
constraint::ConstraintIndex{ScalarAffineFunction{T}, GreaterThan{T}}
end
```

2. Bridge creation

The function Bridges.Constraint.bridge_constraint is called whenever the bridge is instantiated for a specific model, with the given function and set. The arguments to bridge_constraint are similar to add_constraint, with the exception of the first argument: it is the Type of the struct defined in the first step (for our example, Type{SignBridge{T}}).

bridge_constraint returns an instance of the struct defined in the first step. the first step.

In our example, the bridge constraint could be defined as:

```
function Bridges.Constraint.bridge_constraint(
    ::Type{SignBridge{T}}, # Bridge to use.
    model::ModelLike, # Model to which the constraint is being added.
    f::ScalarAffineFunction{T}, # Function to rewrite.
    s::LessThan{T}, # Set to rewrite.
) where {T}
    # Create the variables and constraints required for the bridge.
    con = add_constraint(model, -f, GreaterThan(-s.upper))
    # Return an instance of the bridge type with a reference to all the
    # variables and constraints that were created in this function.
    return SignBridge(con)
end
```

3. Supported constraint types

The function supports_constraint determines whether the bridge type supports a given combination of function and set.

This function must closely match bridge_constraint, because it will not be called if supports_constraint returns false.

```
function supports_constraint(
    ::Type{SignBridge{T}}, # Bridge to use.
    ::Type{ScalarAffineFunction{T}}, # Function to rewrite.
    ::Type{LessThan{T}}, # Set to rewrite.
) where {T}
    # Do some computation to ensure that the constraint is supported.
    # Typically, you can directly return true.
    return true
end
```

4. Metadata about the bridge

To determine whether a bridge can be used, MOI uses a shortest-path algorithm that uses the variable types and the constraints that the bridge can create. This information is communicated from the bridge to MOI using

the functions Bridges.added_constrained_variable_types and Bridges.added_constraint_types. Both return lists of tuples: either a list of 1-tuples containing the variable types (typically, ZeroOne or Integer) or a list of 2-tuples contained the functions and sets (like ScalarAffineFunction{T}-GreaterThan).

For our example, the bridge does not create any constrained variables, and only ScalarAffineFunction{T}-in-GreaterThan{T} constraints:

```
function Bridges.added_constrained_variable_types(::Type{SignBridge{T}}) where {T}
    # The bridge does not create variables, return an empty list of tuples:
    return Tuple{Type}[]
end
function Bridges.added_constraint_types(::Type{SignBridge{T}}) where {T}
    return Tuple{Type,Type}[
        # One element per F-in-S the bridge creates.
        (ScalarAffineFunction{T}, GreaterThan{T}),
    ]
end
```

A bridge that creates binary variables would rather have this definition of added_constrained_variable_types:

```
function Bridges.added_constrained_variable_types(::Type{SomeBridge{T}}) where {T}
    # The bridge only creates binary variables:
    return Tuple{Type}[(ZeroOne,)]
end
```

Warning

If you declare the creation of constrained variables in added_constrained_variable_types, the corresponding constraint type VariableIndex must not be indicated in added_constraint_types. This would restrict the use of the bridge to solvers that can add such a constraint after the variable is created.

More concretely, *if* you declare in added_constrained_variable_types that your bridge creates binary variables (ZeroOne), *and if* you never add such a constraint afterward (you do not call add_constraint(model, var, ZeroOne())), then you must *not* list (VariableIndex, ZeroOne) in added_constraint_types.

Typically, the function Bridges.Constraint.concrete_bridge_type does not have to be defined for most bridges.

Bridge registration

For a bridge to be used by MOI, it must be known by MOI.

SingleBridgeOptimizer

The first way to do so is to create a single-bridge optimizer. This type of optimizer wraps another optimizer and adds the possibility to use only one bridge. It is especially useful when unit testing bridges.

It is common practice to use the same name as the type defined for the bridge (SignBridge, in our example) without the suffix Bridge.

```
CHAPTER 33. TUTORIALS
```

```
const Sign{T,0T<: ModelLike} =
   SingleBridgeOptimizer{SignBridge{T}, 0T}</pre>
```

In the context of unit tests, this bridge is used in conjunction with a Utilities.MockOptimizer:

```
mock = Utilities.MockOptimizer(
    Utilities.UniversalFallback(Utilities.Model{Float64}()),
)
bridged_mock = Sign{Float64}(mock)
```

New bridge for a LazyBridgeOptimizer

Typical user-facing models for MOI are based on Bridges.LazyBridgeOptimizer. For instance, this type of model is returned by Bridges.full_bridge_optimizer. These models can be added more bridges by using Bridges.add_bridge:

```
inner_optimizer = Utilities.Model{Float64}()
optimizer = Bridges.full_bridge_optimizer(inner_optimizer, Float64)
Bridges.add_bridge(optimizer, SignBridge{Float64})
```

Bridge improvements

Attribute retrieval

Like models, bridges have attributes that can be retrieved using get and set. The most important ones are the number of variables and constraints, but also the lists of variables and constraints.

In our example, we only have one constraint and only have to implement the NumberOfConstraints and ListOfConstraintIndices attributes:

```
function get(
    ::SignBridge{T},
    ::NumberOfConstraints{
        ScalarAffineFunction{T},
        GreaterThan{T},
    },
) where {T}
    return 1
end
function get(
    bridge::SignBridge{T},
    ::ListOfConstraintIndices{
        ScalarAffineFunction{T},
        GreaterThan{T},
    },
) where {T}
    return [bridge.constraint]
end
```

You must implement one such pair of functions for each type of constraint the bridge adds to the model.

Warning

Avoid returning a list from the bridge object without copying it. Users must be able to change the contents of the returned list without altering the bridge object.

For variables, the situation is simpler. If your bridge creates new variables, you must implement the NumberOfVariables and ListOfVariableIndices attributes. However, these attributes do not have parameters, unlike their constraint counterparts. Only two functions suffice:

```
function get(
    ::SignBridge{T},
    ::NumberOfVariables,
) where {T}
    return 0
end
function get(
    ::SignBridge{T},
    ::ListOfVariableIndices,
) where {T}
    return VariableIndex[]
end
```

In order for the user to be able to access the function and set of the original constraint, the bridge needs to implement getters for the ConstraintFunction and ConstraintSet attributes:

```
function get(
    model::MOI.ModelLike,
    attr::MOI.ConstraintFunction,
    bridge::SignBridge,
)
    return -MOI.get(model, attr, bridge.constraint)
end
function get(
    model::MOI.ModelLike,
    attr::MOI.ConstraintSet,
    bridge::SignBridge,
)
    set = MOI.get(model, attr, bridge.constraint)
    return MOI.LessThan(-set.lower)
end
```

Warning

Alternatively, one could store the original function and set in SignBridge during Bridges.Constraint.bridge_constraint to make these getters simpler and more efficient. On the other hand, this will increase the memory footprint of the bridges as the garbage collector won't be able to delete that object. The convention is to not store the function in the bridge and not care too much about the efficiency of these getters. If the user needs efficient getters for ConstraintFunction then they should use a Utilities.CachingOptimizer.

Model modifications

To avoid copying the model when the user request to change a constraint, MOI provides modify. Bridges can also implement this API to allow certain changes, such as coefficient changes.

In our case, a modification of a coefficient in the original constraint (for example, replacing the value of the coefficient of a variable in the affine function) must be transmitted to the constraint created by the bridge, but with a sign change.

```
function modify(
    model::ModelLike,
    bridge::SignBridge,
    change::ScalarCoefficientChange,
)
    modify(
        model,
        bridge.constraint,
        ScalarCoefficientChange(change.variable, -change.new_coefficient),
    )
    return
end
```

Bridge deletion

When a bridge is deleted, the constraints it added must be deleted too.

```
function delete(model::ModelLike, bridge::SignBridge)
    delete(model, bridge.constraint)
    return
end
```

33.5 Manipulating expressions

This guide highlights a syntactically appealing way to build expressions at the MOI level, but also to look at their contents. It may be especially useful when writing models or bridge code.

Creating functions

This section details the ways to create functions with MathOptInterface.

Creating scalar affine functions

The simplest scalar function is simply a variable:

```
julia> x = MOI.add_variable(model) # Create the variable x
MOI.VariableIndex(1)
```

This type of function is extremely simple; to express more complex functions, other types must be used. For instance, a ScalarAffineFunction is a sum of linear terms (a factor times a variable) and a constant. Such an object can be built using the standard constructor:

```
julia> f = MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1, x)], 2) # x + 2
(2) + (1) MOI.VariableIndex(1)
```

However, you can also use operators to build the same scalar function:

julia> f = x + 2
(2) + (1) MOI.VariableIndex(1)

Creating scalar quadratic functions

Scalar quadratic functions are stored in ScalarQuadraticFunction objects, in a way that is highly similar to scalar affine functions. You can obtain a quadratic function as a product of affine functions:

```
julia> 1 * x * x
(0) + 1.0 MOI.VariableIndex(1)<sup>2</sup>
julia> f * f # (x + 2)<sup>2</sup>
(4) + (2) MOI.VariableIndex(1) + (2) MOI.VariableIndex(1) + 1.0 MOI.VariableIndex(1)<sup>2</sup>
julia> f<sup>2</sup> # (x + 2)<sup>2</sup> too
(4) + (2) MOI.VariableIndex(1) + (2) MOI.VariableIndex(1) + 1.0 MOI.VariableIndex(1)<sup>2</sup>
```

Creating vector functions

A vector function is a function with several values, irrespective of the number of input variables. Similarly to scalar functions, there are three main types of vector functions: VectorOfVariables, VectorAffineFunction, and VectorQuadraticFunction.

The easiest way to create a vector function is to stack several scalar functions using Utilities.vectorize. It takes a vector as input, and the generated vector function (of the most appropriate type) has each dimension corresponding to a dimension of the vector.

Warning

Utilities.vectorize only takes a vector of similar scalar functions: you cannot mix VariableIndex and ScalarAffineFunction, for instance. In practice, it means that Utilities.vectorize([x, f]) does not work; you should rather use Utilities.vectorize([1 * x, f]) instead to only have ScalarAffineFunction objects.

Canonicalizing functions

In more advanced use cases, you might need to ensure that a function is "canonical." Functions are stored as an array of terms, but there is no check that these terms are redundant: a ScalarAffineFunction object might have two terms with the same variable, like x + x + 1. These terms could be merged without changing the semantics of the function: 2x + 1.

Working with these objects might be cumbersome. Canonicalization helps maintain redundancy to zero.

Utilities.is_canonical checks whether a function is already in its canonical form:

```
julia> MOI.Utilities.is_canonical(f + f) # (x + 2) + (x + 2) is stored as x + x + 4 false
```

Utilities.canonical returns the equivalent canonical version of the function:

```
julia> MOI.Utilities.canonical(f + f) # Returns 2x + 4
(4) + (2) MOI.VariableIndex(1)
```

Exploring functions

At some point, you might need to dig into a function, for instance to map it into solver constructs.

Vector functions

Utilities.scalarize returns a vector of scalar functions from a vector function:

```
julia> MOI.Utilities.scalarize(g) # Returns a vector [f, 2 * f].
2-element Vector{MathOptInterface.ScalarAffineFunction{Int64}}:
(2) + (1) MOI.VariableIndex(1)
(4) + (2) MOI.VariableIndex(1)
```

Note

Utilities.eachscalar returns an iterator on the dimensions, which serves the same purpose as Utilities.scalarize.

output_dimension returns the number of dimensions of the output of a function:

```
julia> MOI.output_dimension(g)
2
```

33.6 Latency

MathOptInterface suffers the "time-to-first-solve" problem of start-up latency.

This hurts both the user- and developer-experience of MathOptInterface. In the first case, because simple models have a multi-second delay before solving, and in the latter, because our tests take so long to run.

This page contains some advice on profiling and fixing latency-related problems in the MathOptInterface.jl repository.

Background

Before reading this part of the documentation, you should familiarize yourself with the reasons for latency in Julia and how to fix them.

- · Read the blogposts on julialang.org on precompilation and SnoopCompile
- Read the SnoopCompile documentation.
- Watch Tim Holy's talk at JuliaCon 2021
- Watch the package development workshop at JuliaCon 2021

Causes

There are three main causes of latency in MathOptInterface:

- 1. A large number of types
- 2. Lack of method ownership
- 3. Type-instability in the bridge layer

A large number of types

Julia is very good at specializing method calls based on the input type. Each specialization has a compilation cost, but the benefit of faster run-time performance.

The best-case scenario is for a method to be called a large number of times with a single set of argument types. The worst-case scenario is for a method to be called a single time for a large set of argument types.

Because of MathOptInterface's function-in-set formulation, we fall into the worst-case situation.

This is a fundamental limitation of Julia, so there isn't much we can do about it. However, if we can precompile MathOptInterface, much of the cost can be shifted from start-up latency to the time it takes to precompile a package on installation.

However, there are two things which make MathOptInterface hard to precompile.

Lack of method ownership

Lack of method ownership happens when a call is made using a mix of structs and methods from different modules. Because of this, no single module "owns" the method that is being dispatched, and so it cannot be precompiled.

Тір

This is a slightly simplified explanation. Read the precompilation tutorial for a more in-depth discussion on back-edges.

Unfortunately, the design of MOI means that this is a frequent occurrence: we have a bunch of types in MOI.Utilities that wrap types defined in external packages (for example, the Optimizers), which implement methods of functions defined in MOI (for example, add_variable, add_constraint).

Here's a simple example of method-ownership in practice:

```
module MyMOI
struct Wrapper{T}
    inner::T
end
optimize!(x::Wrapper) = optimize!(x.inner)
end # MyMOI
module MyOptimizer
using ..MyMOI
struct Optimizer end
MyMOI.optimize!(x::Optimizer) = 1
end # MyOptimizer
using SnoopCompile
model = MyMOI.Wrapper(MyOptimizer.Optimizer())
julia> tinf = @snoopi_deep MyMOI.optimize!(model)
InferenceTimingNode: 0.008256/0.008543 on InferenceFrameInfo for Core.Compiler.Timings.ROOT() with
\hookrightarrow 1 direct children
```

The result is that there was one method that required type inference. If we visualize tinf:

```
using ProfileView
ProfileView.view(flamegraph(tinf))
```

we see a flamegraph with a large red-bar indicating that the method MyMOI.optimize(MyMOI.Wrapper{MyOptimizer.Optimizer} cannot be precompiled.

To fix this, we need to designate a module to "own" that method (that is, create a back-edge). The easiest way to do this is for MyOptimizer to call MyMOI.optimize(MyMOI.Wrapper{MyOptimizer.Optimizer}) during using MyOptimizer. Let's see that in practice:

```
module MyMOI
struct Wrapper{T}
    inner::T
end
optimize(x::Wrapper) = optimize(x.inner)
end # MyMOI
module MyOptimizer
using ...MyMOI
struct Optimizer end
MyMOI.optimize(x::Optimizer) = 1
# The syntax of this let-while loop is very particular:
# * `let ... end` keeps everything local to avoid polluting the MyOptimizer
   namespace
#
# * `while true ... break end` runs the code once, and forces Julia to compile
    the inner loop, rather than interpret it.
#
let
   while true
        model = MyMOI.Wrapper(Optimizer())
       MyMOI.optimize(model)
        break
```

```
end
end
end # MyOptimizer
using SnoopCompile
model = MyMOI.Wrapper(MyOptimizer.Optimizer())
julia> tinf = @snoopi_deep MyMOI.optimize(model)
InferenceTimingNode: 0.006822/0.006822 on InferenceFrameInfo for Core.Compiler.Timings.ROOT() with

→ 0 direct children
```

There are now 0 direct children that required type inference because the method was already stored in MyOptimizer!

Unfortunately, this trick only works if the call-chain is fully inferrable. If there are breaks (due to type instability), then the benefit of doing this is reduced. And unfortunately for us, the design of MathOptInterface has a lot of type instabilities.

Type instability in the bridge layer

Most of MathOptInterface is pretty good at ensuring type-stability. However, a key component is not type stable, and that is the bridging layer.

In particular, the bridging layer defines Bridges.LazyBridgeOptimizer, which has fields like:

```
struct LazyBridgeOptimizer
    constraint_bridge_types::Vector{Any}
    constraint_node::Dict{Tuple{Type,Type},ConstraintNode}
    constraint_types::Vector{Tuple{Type,Type}}
end
```

This is because the LazyBridgeOptimizer needs to be able to deal with any *function-in-set* type passed to it, and we also allow users to pass additional bridges that they defined in external packages.

So to recap, MathOptInterface suffers package latency because:

- 1. there are a large number of types and functions
- 2. and these are split between multiple modules, including external packages
- 3. and there are type-instabilities like those in the bridging layer.

Resolutions

There are no magic solutions to reduce latency. Issue #1313 tracks progress on reducing latency in MathOpt-Interface.

A useful script is the following (replace GLPK as needed):

```
import GLPK
import MathOptInterface as MOI
function example_diet(optimizer, bridge)
    category_data = [
        1800.0 2200.0;
```

```
Inf;
      91.0
       0.0 65.0;
       0.0 1779.0
]
cost = [2.49, 2.89, 1.50, 1.89, 2.09, 1.99, 2.49, 0.89, 1.59]
food_data = [
    410 24 26 730;
   420 32 10 1190;
   560 20 32 1800;
   380 4 19 270;
   320 12 10 930;
   320 15 12 820;
   320 31 12 1230;
    100 8 2.5 125;
    330 8 10 180
]
bridge_model = if bridge
    MOI.instantiate(optimizer; with_bridge_type=Float64)
else
    MOI.instantiate(optimizer)
end
model = MOI.Utilities.CachingOptimizer(
    MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}()),
    MOI.Utilities.AUTOMATIC,
)
MOI.Utilities.reset_optimizer(model, bridge_model)
MOI.set(model, MOI.Silent(), true)
nutrition = MOI.add_variables(model, size(category_data, 1))
for (i, v) in enumerate(nutrition)
    MOI.add constraint(model, v, MOI.GreaterThan(category data[i, 1]))
    MOI.add_constraint(model, v, MOI.LessThan(category_data[i, 2]))
end
buy = MOI.add_variables(model, size(food_data, 1))
MOI.add_constraint.(model, buy, MOI.GreaterThan(0.0))
MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)
f = MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(cost, buy), 0.0)
MOI.set(model, MOI.ObjectiveFunction{typeof(f)}(), f)
for (j, n) in enumerate(nutrition)
    f = MOI.ScalarAffineFunction(
        MOI.ScalarAffineTerm.(food_data[:, j], buy),
        0.0.
    )
    push!(f.terms, MOI.ScalarAffineTerm(-1.0, n))
    MOI.add_constraint(model, f, MOI.EqualTo(0.0))
end
MOI.optimize!(model)
term_status = MOI.get(model, MOI.TerminationStatus())
@assert term_status == MOI.OPTIMAL
MOI.add_constraint(
    model,
    MOI.ScalarAffineFunction(
        MOI.ScalarAffineTerm. (1.0, [buy[end-1], buy[end]]),
        0.0,
    ).
    MOI.LessThan(6.0),
```



Figure 33.1: flamegraph

```
)
MOI.optimize!(model)
@assert MOI.get(model, MOI.TerminationStatus()) == MOI.INFEASIBLE
return
end
if length(ARGS) > 0
bridge = get(ARGS, 2, "") != "--no-bridge"
println("Running: $(ARGS[1]) $(get(ARGS, 2, ""))")
@time example_diet(GLPK.Optimizer, bridge)
@time example_diet(GLPK.Optimizer, bridge)
exit(0)
end
```

You can create a flame-graph via

```
using SnoopCompile
tinf = @snoopi_deep example_diet(GLPK.Optimizer, true)
using ProfileView
ProfileView.view(flamegraph(tinf))
```

Here's how things looked in mid-August 2021:

There are a few opportunities for improvement (non-red flames, particularly on the right). But the main problem is a large red (non-precompilable due to method ownership) flame.

Chapter 34

Manual

34.1 Standard form problem

MathOptInterface represents optimization problems in the standard form:

$$\min_{x \in \mathbb{R}^n} \qquad f_0(x) \tag{34.1}$$
 s.t.
$$f_i(x) \in \mathcal{S}_i \qquad i = 1 \dots m \tag{34.2}$$

where:

- the functions f_0, f_1, \ldots, f_m are specified by <code>AbstractFunction</code> objects
- the sets $\mathcal{S}_1,\ldots,\mathcal{S}_m$ are specified by <code>AbstractSet</code> objects

| Тір | |
|---|--|
| For more information on this standard form, read our paper. | |

MOI defines some commonly used functions and sets, but the interface is extensible to other sets recognized by the solver.

Functions

The function types implemented in MathOptInterface.jl are:

Extensions for nonlinear programming are present but not yet well documented.

One-dimensional sets

The one-dimensional set types implemented in MathOptInterface.jl are:

Vector cones

The vector-valued set types implemented in MathOptInterface.jl are:

| Function | Description |
|------------------------|--|
| VariableIndex | x_j , the projection onto a single coordinate defined by a variable index $j.$ |
| Vector0fVariables | The projection onto multiple coordinates (that is, extracting a sub-vector). |
| ScalarAffineFunction | $a^Tx + b$, where a is a vector and b scalar. |
| ScalarNonlinearFunctio | n $f(x)$, where f is a nonlinear function. |
| VectorAffineFunction | Ax + b, where A is a matrix and b is a vector. |
| ScalarQuadraticFunctio | $\ln \frac{1}{2}x^TQx + a^Tx + b$, where Q is a symmetric matrix, a is a vector, and b is a |
| | constant. |
| VectorQuadraticFunctio | A vector of scalar-valued quadratic functions. |
| VectorNonlinearFunctio | $\inf f(x)$, where f is a vector-valued nonlinear function. |

| Set | Description |
|---------------------------------|--|
| LessThan(u) | $(-\infty, u]$ |
| <pre>GreaterThan(l)</pre> | $[l,\infty)$ |
| EqualTo(v) | $\{v\}$ |
| <pre>Interval(l, u)</pre> | [l, u] |
| Integer() | \mathbb{Z} |
| ZeroOne() | $\{0, 1\}$ |
| <pre>Semicontinuous(l, u)</pre> | $\{0\}\cup [l,u]$ |
| <pre>Semiinteger(l, u)</pre> | $\{0\} \cup \{l, l+1, \dots, u-1, u\}$ |

| Set | Description |
|----------------------------------|--|
| Reals(d) | \mathbb{R}^{d} |
| Zeros(d) | 0^d |
| Nonnegatives(d) | $\{x \in \mathbb{R}^d : x \ge 0\}$ |
| Nonpositives(d) | $\{x \in \mathbb{R}^d : x \le 0\}$ |
| SecondOrderCone(d) | $\{(t,x)\in\mathbb{R}^d:t\geq\ x\ _2\}$ |
| RotatedSecondOrderCone(d) | $\{(t, u, x) \in \mathbb{R}^d : 2tu \ge \ x\ _2^2, t \ge 0, u \ge 0\}$ |
| <pre>ExponentialCone()</pre> | $\{(x,y,z)\in \mathbb{R}^3: y\exp(x/y)\leq z,y>0\}$ |
| <pre>DualExponentialCone()</pre> | $\{(u,v,w)\in\mathbb{R}^3:-u\exp(v/u)\leq\exp(1)w,u<0\}$ |
| <pre>GeometricMeanCone(d)</pre> | $\{(t,x)\in\mathbb{R}^{1+n}:x\geq 0,t\leq\sqrt[n]{x_1x_2\cdots x_n}\}$ where n is $d-1$ |
| PowerCone(a) | $\{(x,y,z)\in\mathbb{R}^3: x^{\alpha}y^{1-\alpha}\geq z , x\geq 0, y\geq 0\}$ |
| DualPowerCone(α) | $\{(u, v, w) \in \mathbb{R}^3 : \left(\frac{u}{\alpha}\right)^{\alpha} \left(\frac{v}{1-\alpha}\right)^{1-\alpha} \ge w , u, v \ge 0\}$ |
| NormOneCone(d) | $\{(t,x) \in \mathbb{R}^d : t \ge \sum_i x_i \}$ |
| NormInfinityCone(d) | $\{(t,x) \in \mathbb{R}^d : t \ge \max_i x_i \}$ |
| RelativeEntropyCone(d) | $\{(u, v, w) \in \mathbb{R}^d : u \ge \sum_i w_i \log(\frac{w_i}{v_i}), v_i \ge 0, w_i \ge 0\}$ |
| HyperRectangle(l, u) | $\{x \in \mathbb{R}^d : x_i \in [l_i, u_i] \forall i = 1, \dots, d\}$ |
| NormCone(p, d) | $\{(t,x) \in \mathbb{R}^d : t \ge \left(\sum_i x_i ^p\right)^{\frac{1}{p}}\}$ |

Matrix cones

The matrix-valued set types implemented in MathOptInterface.jl are:

Some of these cones can take two forms: XXXConeTriangle and XXXConeSquare.

In XXXConeTriangle sets, the matrix is assumed to be symmetric, and the elements are provided by a vector, in which the entries of the upper-right triangular part of the matrix are given column by column (or equivalently, the entries of the lower-left triangular part are given row by row).

| Set | Description |
|-------------------------------|---|
| RootDetConeTriangle(d) | $\{(t,X) \in \mathbb{R}^{1+d(1+d)/2} : t \le t$ |
| | $\det(X)^{1/d}, X$ is the upper triangle of a PSD matrix} |
| RootDetConeSquare(d) | $\{(t,X) \in \mathbb{R}^{1+d^2} : t \le \det(X)^{1/d}, X \text{ is a PSD matrix}\}$ |
| PositiveSemidefiniteConeTri | angle ($\{X \in \mathbb{R}^{d(d+1)/2} : X 	ext{ is the upper triangle of a PSD matrix}\}$ |
| PositiveSemidefiniteConeSqu | are (d) $\{X\in \mathbb{R}^{d^2}: X 	ext{ is a PSD matrix}\}$ |
| LogDetConeTriangle(d) | $\{(t,u,X)\in\mathbb{R}^{2+d(1+d)/2}:t\leq$ |
| | $u\log(\det(X/u)), X$ is the upper triangle of a PSD matrix, $u>0\}$ |
| LogDetConeSquare(d) | $\{(t, u, X) \in \mathbb{R}^{2+d^2} : t \le$ |
| | $u\log(\det(X/u)), X$ is a PSD matrix, $u>0\}$ |
| NormSpectralCone(r, c) | $\{(t,X)\in\mathbb{R}^{1+r	imes c}:t\geq\sigma_1(X),X	ext{ is a }r	imes c	ext{ matrix}\}$ |
| NormNuclearCone(r, c) | $\{(t,X)\in \mathbb{R}^{1+r	imes c}:t\geq \sum_i\sigma_i(X),X	ext{ is a }r	imes c	ext{ matrix}\}$ |
| HermitianPositiveSemidefini | teConellhe conectillermitian positive semidefinite matrices, with |
| side_dimension rows and colum | ins. |
| Scaled(S) | The set S scaled so that Utilities.set_dot corresponds to |
| | LinearAlgebra.dot |

In XXXConeSquare sets, the entries of the matrix are given column by column (or equivalently, row by row), and the matrix is constrained to be symmetric. As an example, given a 2-by-2 matrix of variables X and a one-dimensional variable t, we can specify a root-det constraint as [t, X11, X12, X22] \in RootDetConeTriangle or [t, X11, X12, X21, X22] \in RootDetConeSquare.

We provide both forms to enable flexibility for solvers who may natively support one or the other. Transformations between XXXConeTriangle and XXXConeSquare are handled by bridges, which removes the chance of conversion mistakes by users or solver developers.

Multi-dimensional sets with combinatorial structure

Other sets are vector-valued, with a particular combinatorial structure. Read their docstrings for more information on how to interpret them.

| Set | Description |
|------------------|---|
| S0S1 | A Special Ordered Set (SOS) of Type I |
| S0S2 | A Special Ordered Set (SOS) of Type II |
| Indicator | A set to specify an indicator constraint |
| Complements | A set to specify a mixed complementarity constraint |
| AllDifferent | The all_different global constraint |
| BinPacking | The bin_packing global constraint |
| Circuit | The circuit global constraint |
| CountAtLeast | The at_least global constraint |
| CountBelongs | The nvalue global constraint |
| CountDistinct | The distinct global constraint |
| CountGreaterThan | The count_gt global constraint |
| Cumulative | The cumulative global constraint |
| Path | The path global constraint |
| Table | The table global constraint |

34.2 Models

The most significant part of MOI is the definition of the **model API** that is used to specify an instance of an optimization problem (for example, by adding variables and constraints). Objects that implement the model API must inherit from the ModelLike abstract type.

Notably missing from the model API is the method to solve an optimization problem. ModelLike objects may store an instance (for example, in memory or backed by a file format) without being linked to a particular solver. In addition to the model API, MOI defines AbstractOptimizer and provides methods to solve the model and interact with solutions. See the Solutions section for more details.

Info

Throughout the rest of the manual, model is used as a generic ModelLike, and optimizer is used as a generic AbstractOptimizer.

Tip

MOI does not export functions, but for brevity we often omit qualifying names with the MOI module. Best practice is to have

import MathOptInterface as MOI

and prefix all MOI methods with MOI. in user code. If a name is also available in base Julia, we always explicitly use the module prefix, for example, with MOI.get.

Attributes

Attributes are properties of the model that can be queried and modified. These include constants such as the number of variables in a model NumberOfVariables), and properties of variables and constraints such as the name of a variable (VariableName).

There are four types of attributes:

- Model attributes (subtypes of AbstractModelAttribute) refer to properties of a model.
- Optimizer attributes (subtypes of AbstractOptimizerAttribute) refer to properties of an optimizer.
- Constraint attributes (subtypes of AbstractConstraintAttribute) refer to properties of an individual constraint.
- Variable attributes (subtypes of AbstractVariableAttribute) refer to properties of an individual variable.

Some attributes are values that can be queried by the user but not modified, while other attributes can be modified by the user.

All interactions with attributes occur through the get and set functions.

Consult the docstrings of each attribute for information on what it represents.
ModelLike API

The following attributes are available:

- ListOfConstraintAttributesSet
- ListOfConstraintIndices
- ListOfConstraintTypesPresent
- ListOfConstraintsWithAttributeSet
- ListOfModelAttributesSet
- ListOfVariableAttributesSet
- ListOfVariableIndices
- ListOfVariablesWithAttributeSet
- NumberOfConstraints
- NumberOfVariables
- Name
- ObjectiveFunction
- ObjectiveFunctionType
- ObjectiveSense

AbstractOptimizer API

The following attributes are available:

- DualStatus
- PrimalStatus
- RawStatusString
- ResultCount
- TerminationStatus
- BarrierIterations
- DualObjectiveValue
- NodeCount
- NumberOfThreads
- ObjectiveBound
- ObjectiveValue
- RelativeGap
- RawOptimizerAttribute

- RawSolver
- Silent
- SimplexIterations
- SolverName
- SolverVersion
- SolveTimeSec
- TimeLimitSec
- ObjectiveLimit
- SolutionLimit
- NodeLimit
- AutomaticDifferentiationBackend

34.3 Variables

Add a variable

Use add_variable to add a single variable.

```
julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)
```

add_variable returns a VariableIndex type, which is used to refer to the added variable in other calls.

Check if a VariableIndex is valid using is_valid.

julia> MOI.is_valid(model, x)
true

Use add_variables to add a number of variables.

```
julia> y = MOI.add_variables(model, 2)
2-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(2)
MOI.VariableIndex(3)
```

Warning

The integer does not necessarily correspond to the column inside an optimizer.

Delete a variable

Delete a variable using delete.

```
julia> MOI.delete(model, x)
```

```
julia> MOI.is_valid(model, x)
false
```

Warning

Not all ModelLike models support deleting variables. A DeleteNotAllowed error is thrown if this is not supported.

Variable attributes

The following attributes are available for variables:

- VariableName
- VariablePrimalStart
- VariablePrimal

Get and set these attributes using get and set.

```
julia> MOI.set(model, MOI.VariableName(), x, "var_x")
julia> MOI.get(model, MOI.VariableName(), x)
"var_x"
```

34.4 Constraints

Add a constraint

Use add_constraint to add a single constraint.

add_constraint returns a ConstraintIndex type, which is used to refer to the added constraint in other calls.

Check if a ConstraintIndex is valid using is_valid.

```
julia> MOI.is_valid(model, c)
true
```

Use add_constraints to add a number of constraints of the same type.

```
julia> c = MOI.add_constraints(
            model,
            [x[1], x[2]],
            [MOI.GreaterThan(0.0), MOI.GreaterThan(1.0)]
        )
2-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
        → MathOptInterface.GreaterThan{Float64}}:
MathOptInterface.GreaterThan{Float64}}:
        MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
        → MathOptInterface.GreaterThan{Float64}}(1)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
        → MathOptInterface.GreaterThan{Float64}}(2)
```

This time, a vector of ConstraintIndex are returned.

Use supports_constraint to check if the model supports adding a constraint type.

Delete a constraint

Use delete to delete a constraint.

```
julia> MOI.delete(model, c)
julia> MOI.is_valid(model, c)
false
```

Constraint attributes

The following attributes are available for constraints:

- ConstraintName
- ConstraintPrimalStart
- ConstraintDualStart
- ConstraintPrimal
- ConstraintDual
- ConstraintBasisStatus
- ConstraintFunction
- CanonicalConstraintFunction
- ConstraintSet

Get and set these attributes using get and set.

```
julia> MOI.set(model, MOI.ConstraintName(), c, "con_c")
julia> MOI.get(model, MOI.ConstraintName(), c)
"con_c"
```

Constraints by function-set pairs

Below is a list of common constraint types and how they are represented as function-set pairs in MOI. In the notation below, x is a vector of decision variables, x_i is a scalar decision variable, α, β are scalar constants, a, b are constant vectors, A is a constant matrix and \mathbb{R}_+ (resp. \mathbb{R}_-) is the set of non-negative (resp. non-positive) real numbers.

Linear constraints

| Mathematical Constraint | MOI Function | MOI Set |
|--------------------------------|----------------------|--------------|
| $a^T x \leq \beta$ | ScalarAffineFunction | LessThan |
| $a^T x \ge \alpha$ | ScalarAffineFunction | GreaterThan |
| $a^T x = \beta$ | ScalarAffineFunction | EqualTo |
| $\alpha \leq a^T x \leq \beta$ | ScalarAffineFunction | Interval |
| $x_i \leq \beta$ | VariableIndex | LessThan |
| $x_i \ge \alpha$ | VariableIndex | GreaterThan |
| $x_i = \beta$ | VariableIndex | EqualTo |
| $\alpha \le x_i \le \beta$ | VariableIndex | Interval |
| $Ax + b \in \mathbb{R}^n_+$ | VectorAffineFunction | Nonnegatives |
| $Ax + b \in \mathbb{R}^n$ | VectorAffineFunction | Nonpositives |
| Ax + b = 0 | VectorAffineFunction | Zeros |

By convention, solvers are not expected to support nonzero constant terms in the ScalarAffineFunctions the first four rows of the preceding table because they are redundant with the parameters of the sets. For example, encode $2x + 1 \le 2$ as $2x \le 1$.

Constraints with VariableIndex in LessThan, GreaterThan, EqualTo, or Interval sets have a natural interpretation as variable bounds. As such, it is typically not natural to impose multiple lower- or upper-bounds on the same variable, and the solver interfaces will throw respectively LowerBoundAlreadySet or UpperBoundAlreadySet.

Moreover, adding two VariableIndex constraints on the same variable with the same set is impossible because they share the same index as it is the index of the variable, see ConstraintIndex.

It is natural, however, to impose upper- and lower-bounds separately as two different constraints on a single variable. The difference between imposing bounds by using a single Interval constraint and by using separate LessThan and GreaterThan constraints is that the latter will allow the solver to return separate dual multipliers for the two bounds, while the former will allow the solver to return only a single dual for the interval constraint.

Conic constraints

where \mathcal{E} is the exponential cone (see ExponentialCone), \mathcal{S}_+ is the set of positive semidefinite symmetric matrices, A is an affine map that outputs symmetric matrices and B is an affine map that outputs square matrices.

| Mathematical Constraint | MOI Function | MOI Set |
|---|----------------------|----------------------------------|
| $ Ax+b _2 \le c^T x + d$ | VectorAffineFunction | SecondOrderCone |
| $y \ge \ x\ _2$ | Vector0fVariables | SecondOrderCone |
| $2yz \ge x _2^2, y, z \ge 0$ | Vector0fVariables | RotatedSecondOrderCone |
| $(a_1^T x + b_1, a_2^T x + b_2, a_3^T x + b_3) \in \mathcal{E}$ | VectorAffineFunction | ExponentialCone |
| $A(x) \in \mathcal{S}_+$ | VectorAffineFunction | PositiveSemidefiniteConeTriangle |
| $B(x) \in \mathcal{S}_+$ | VectorAffineFunction | PositiveSemidefiniteConeSquare |
| $x \in \mathcal{S}_+$ | Vector0fVariables | PositiveSemidefiniteConeTriangle |
| $x \in \mathcal{S}_+$ | Vector0fVariables | PositiveSemidefiniteConeSquare |

| Mathematical Constraint | MOI Function | MOI Set |
|-------------------------------------|-------------------------|--------------------------|
| $\frac{1}{2}x^TQx + a^Tx + b \ge 0$ | ScalarQuadraticFunction | GreaterThan |
| $\frac{1}{2}x^TQx + a^Tx + b \le 0$ | ScalarQuadraticFunction | LessThan |
| $\frac{1}{2}x^TQx + a^Tx + b = 0$ | ScalarQuadraticFunction | EqualTo |
| Bilinear matrix inequality | VectorQuadraticFunction | PositiveSemidefiniteCone |

Quadratic constraints

Note

For more details on the internal format of the quadratic functions see ScalarQuadraticFunction or VectorQuadraticFunction.

Discrete and logical constraints

| Mathematical Constraint | MOI Function | MOI Set |
|---|-------------------|----------------|
| $x_i \in \mathbb{Z}$ | VariableIndex | Integer |
| $x_i \in \{0, 1\}$ | VariableIndex | Zero0ne |
| $x_i \in \{0\} \cup [l,u]$ | VariableIndex | Semicontinuous |
| $x_i \in \{0\} \cup \{l, l+1, \dots, u-1, u\}$ | VariableIndex | Semiinteger |
| At most one component of x can be nonzero | VectorOfVariables | S0S1 |
| At most two components of \boldsymbol{x} can be nonzero, and if so they must be | VectorOfVariables | S0S2 |
| adjacent components | | |
| $y = 1 \implies a^T x \in S$ | VectorAffineFunct | ionIndicator |

JuMP mapping

The following bullet points show examples of how JuMP constraints are translated into MOI function-set pairs:

- @constraint(m, 2x + y <= 10) becomes ScalarAffineFunction-in-LessThan
- @constraint(m, 2x + y >= 10) becomes ScalarAffineFunction-in-GreaterThan
- @constraint(m, 2x + y == 10) becomes ScalarAffineFunction-in-EqualTo
- @constraint(m, 0 <= 2x + y <= 10) becomes ScalarAffineFunction-in-Interval
- @constraint(m, 2x + y in ArbitrarySet()) becomes ScalarAffineFunction-in-ArbitrarySet.

Variable bounds are handled in a similar fashion:

- @variable(m, x <= 1) becomes VariableIndex-in-LessThan
- @variable(m, x >= 1) becomes VariableIndex-in-GreaterThan

One notable difference is that a variable with an upper and lower bound is translated into two constraints, rather than an interval, that is:

• @variable(m, 0 <= x <= 1) becomes VariableIndex-in-LessThan and VariableIndex-in-GreaterThan.

34.5 Solutions

Solving and retrieving the results

Once an optimizer is loaded with the objective function and all of the constraints, we can ask the solver to solve the model by calling optimize!.

MOI.optimize!(optimizer)

Why did the solver stop?

The optimization procedure may stop for a number of reasons. The TerminationStatus attribute of the optimizer returns a TerminationStatusCode object which explains why the solver stopped.

The termination statuses distinguish between proofs of optimality, infeasibility, local convergence, limits, and termination because of something unexpected like invalid problem data or failure to converge.

A typical usage of the TerminationStatus attribute is as follows:

```
status = MOI.get(optimizer, TerminationStatus())
if status == MOI.OPTIMAL
    # Ok, we solved the problem!
else
    # Handle other cases.
end
```

After checking the TerminationStatus, check ResultCount. This attribute returns the number of results that the solver has available to return. A result is defined as a primal-dual pair, but either the primal or the dual may be missing from the result. While the OPTIMAL termination status normally implies that at least one result is available, other statuses do not. For example, in the case of infeasibility, a solver may return no result or a proof of infeasibility. The ResultCount attribute distinguishes between these two cases.

Primal solutions

Use the PrimalStatus optimizer attribute to return a ResultStatusCode describing the status of the primal solution.

Common returns are described below in the Common status situations section.

Query the primal solution using the VariablePrimal and ConstraintPrimal attributes.

Query the objective function value using the ObjectiveValue attribute.

Dual solutions

Warning

See Duality for a discussion of the MOI conventions for primal-dual pairs and certificates.

Use the DualStatus optimizer attribute to return a ResultStatusCode describing the status of the dual solution.

Query the dual solution using the ConstraintDual attribute.

Query the dual objective function value using the DualObjectiveValue attribute.

Common status situations

The sections below describe how to interpret typical or interesting status cases for three common classes of solvers. The example cases are illustrative, not comprehensive. Solver wrappers may provide additional information on how the solver's statuses map to MOI statuses.

Info

* in the tables indicate that multiple different values are possible.

Primal-dual convex solver

Linear programming and conic optimization solvers fall into this category.

| What happened? | TerminationSta Res | ₅ultCoun | t PrimalStatus | DualStatus | |
|--|---------------------------|----------|---------------------|------------------------|------|
| Proved optimality | OPTIMAL | 1 | FEASIBLE_POINT | FEASIBLE_POINT | |
| Proved infeasible | INFEASIBLE | 1 | NO_SOLUTION | INFEASIBILITY_CERTIFI | CATE |
| Optimal within relaxed | ALMOST_OPTIMAL | 1 | FEASIBLE_POINT | FEASIBLE_POINT | |
| tolerances | | | | | |
| Optimal within relaxed tolerances | ALMOST_OPTIMAL | 1 | ALMOST_FEASIBLE_POI | NALMOST_FEASIBLE_POINT | |
| Detected an unbounded ray
of the primal | DUAL_INFEASIBLE | 1 | INFEASIBILITY_CERTI | FICATE NO_SOLUTION | |
| Stall | SLOW_PROGRESS | 1 | * | * | |

Global branch-and-bound solvers

Mixed-integer programming solvers fall into this category.

| What happened? | TerminationStatus F | ResultCount | PrimalStatus | DualStatus |
|------------------------------------|-----------------------|-------------|-----------------|--------------|
| Proved optimality | OPTIMAL | 1 | FEASIBLE_POINT | NO_SOLUTION |
| Presolve detected infeasibility or | INFEASIBLE_OR_UNBOUND | DED 0 | NO_SOLUTION | NO_SOLUTION |
| unboundedness | | | | |
| Proved infeasibility | INFEASIBLE | 0 | NO_SOLUTION | NO_SOLUTION |
| Timed out (no solution) | TIME_LIMIT | 0 | NO_SOLUTION | NO_SOLUTION |
| Timed out (with a solution) | TIME_LIMIT | 1 | FEASIBLE_POINT | NO_SOLUTION |
| CPXMIP_OPTIMAL_INFEAS | ALMOST_OPTIMAL | 1 | INFEASIBLE_POIN | NTO_SOLUTION |
| | | | | |

Info

CPXMIP_OPTIMAL_INFEAS is a CPLEX status that indicates that a preprocessed problem was solved to optimality, but the solver was unable to recover a feasible solution to the original problem. Handling this status was one of the motivating drivers behind the design of MOI.

Local search solvers

Nonlinear programming solvers fall into this category. It also includes non-global tree search solvers like Juniper.

| What happened? | TerminationStatus | ResultCou | ın₽rimalStatus | DualStatus |
|------------------------------------|--------------------|-----------|----------------|------------------|
| Converged to a stationary point | LOCALLY_SOLVED | 1 | FEASIBLE_POIN | TFEASIBLE_POINT |
| Completed a non-global tree search | LOCALLY_SOLVED | 1 | FEASIBLE_POIN | NTFEASIBLE_POINT |
| (with a solution) | | | | |
| Converged to an infeasible point | LOCALLY_INFEASIBLE | 1 | INFEASIBLE_PO |)INT * |
| Completed a non-global tree search | LOCALLY_INFEASIBLE | 0 | NO_SOLUTION | NO_SOLUTION |
| (no solution found) | | | | |
| Iteration limit | ITERATION_LIMIT | 1 | * | * |
| Diverging iterates | NORM_LIMIT or | 1 | * | * |
| | OBJECTIVE_LIMIT | | | |

Querying solution attributes

Some solvers will not implement every solution attribute. Therefore, a call like MOI.get(model, MOI.SolveTimeSec()) may throw an UnsupportedAttribute error.

If you need to write code that is agnostic to the solver (for example, you are writing a library that an end-user passes their choice of solver to), you can work-around this problem using a try-catch:

```
function get_solve_time(model)
try
return MOI.get(model, MOI.SolveTimeSec())
catch err
if err isa MOI.UnsupportedAttribute
return NaN # Solver doesn't support. Return a placeholder value.
end
rethrow(err) # Something else went wrong. Rethrow the error
end
end
```

If, *after careful profiling*, you find that the try-catch is taking a significant portion of your runtime, you can improve performance by caching the result of the try-catch:

```
mutable struct CachedSolveTime{M}
    model::M
    supports_solve_time::Bool
    CachedSolveTime(model::M) where {M} = new(model, true)
end
function get_solve_time(model::CachedSolveTime)
    if !model.supports_solve_time
```

```
return NaN
end
try
return MOI.get(model, MOI.SolveTimeSec())
catch err
if err isa MOI.UnsupportedAttribute
model.supports_solve_time = false
return NaN
end
rethrow(err) # Something else went wrong. Rethrow the error
end
end
```

34.6 Problem modification

In addition to adding and deleting constraints and variables, MathOptInterface supports modifying, in-place, coefficients in the constraints and the objective function of a model.

These modifications can be grouped into two categories:

- modifications which replace the set of function of a constraint with a new set or function
- modifications which change, in-place, a component of a function

Warning

Some ModelLike objects do not support problem modification.

Modify the set of a constraint

Use set and ConstraintSet to modify the set of a constraint by replacing it with a new instance of the same type.

However, the following will fail as the new set is of a different type to the original set:

```
julia> MOI.set(model, MOI.ConstraintSet(), c, MOI.GreaterThan(2.0))
ERROR: [...]
```

Special cases: set transforms

If our constraint is an affine inequality, then this corresponds to modifying the right-hand side of a constraint in linear programming.

In some special cases, solvers may support efficiently changing the set of a constraint (for example, from LessThan to GreaterThan). For these cases, MathOptInterface provides the transform method.

The transform function returns a new constraint index, and the old constraint index (that is, c) is no longer valid.

```
julia> c = MOI.add_constraint(
    model,
    MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
    MOI.LessThan(1.0),
)
```

MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64}, → MathOptInterface.LessThan{Float64}}(1)

```
julia> new_c = MOI.transform(model, c, MOI.GreaterThan(2.0));
```

```
julia> MOI.is_valid(model, c)
false
```

```
julia> MOI.is_valid(model, new_c)
true
```

Note

transform cannot be called with a set of the same type. Use set instead.

Modify the function of a constraint

Use set and ConstraintFunction to modify the function of a constraint by replacing it with a new instance of the same type.

```
julia> MOI.set(model, MOI.ConstraintFunction(), c, new_f);
```

CHAPTER 34. MANUAL

```
julia> MOI.get(model, MOI.ConstraintFunction(), c) \approx new_f true
```

However, the following will fail as the new function is of a different type to the original function:

```
julia> MOI.set(model, MOI.ConstraintFunction(), c, x)
ERROR: [...]
```

Modify constant term in a scalar function

Use modify and ScalarConstantChange to modify the constant term in a ScalarAffineFunction or ScalarQuadraticFunction.

```
julia> c = MOI.add_constraint(
    model,
    MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
    MOI.EqualTo(1.0),
    )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
```

→ MathOptInterface.EqualTo{Float64}}(1)

```
julia> MOI.modify(model, c, MOI.ScalarConstantChange(1.0));
```

```
julia> new_f = MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 1.0);
```

```
<code>julia> MOI.get(model, MOI.ConstraintFunction(), c) \approx new_f true</code>
```

ScalarConstantChange can also be used to modify the objective function by passing an instance of ObjectiveFunction:

Modify constant terms in a vector function

Use modify and VectorConstantChange to modify the constant vector in a VectorAffineFunction or VectorQuadraticFunction

```
julia> c = MOI.add_constraint(
           model.
           MOI.VectorAffineFunction([
                   MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, x)),
                   MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(2.0, x)),
               ],
               [0.0, 0.0],
           ),
           MOI.Nonnegatives(2),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
→ MathOptInterface.Nonnegatives}(1)
julia> MOI.modify(model, c, MOI.VectorConstantChange([3.0, 4.0]));
julia> new_f = MOI.VectorAffineFunction(
           [
        MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, x)),
       MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(2.0, x)),
          1.
           [3.0, 4.0],
       );
julia> MOI.get(model, MOI.ConstraintFunction(), c) ~ new_f
```

```
true
```

Modify affine coefficients in a scalar function

Use modify and ScalarCoefficientChange to modify the affine coefficient of a ScalarAffineFunction or ScalarQuadraticFunction.

ScalarCoefficientChange can also be used to modify the objective function by passing an instance of ObjectiveFunction.

Modify quadratic coefficients in a scalar function

Use modify and ScalarQuadraticCoefficientChange to modify the quadratic coefficient of a ScalarQuadraticFunction.

```
julia> model = MOI.Utilities.Model{Float64}();
julia> x = MOI.add_variables(model, 2);
julia> c = MOI.add_constraint(
           model,
          1.0 * x[1] * x[1] + 2.0 * x[1] * x[2],
          MOI.EqualTo(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarQuadraticFunction{Float64},
→ MathOptInterface.EqualTo{Float64}}(1)
julia> MOI.modify(
           model,
           с.
           MOI.ScalarQuadraticCoefficientChange(x[1], x[1], 3.0),
       );
julia> MOI.modify(
          model,
           с.
           MOI.ScalarQuadraticCoefficientChange(x[1], x[2], 4.0),
      );
julia> new_f = 1.5 * x[1] * x[1] + 4.0 * x[1] * x[2];
julia> MOI.get(model, MOI.ConstraintFunction(), c) ~ new_f
true
```

ScalarQuadraticCoefficientChange can also be used to modify the objective function by passing an instance of ObjectiveFunction.

Modify affine coefficients in a vector function

Use modify and MultirowChange to modify a vector of affine coefficients in a VectorAffineFunction or a VectorQuadraticFunction.

```
julia> c = MOI.add_constraint(
    model,
    MOI.VectorAffineFunction([
        MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, x)),
        MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(2.0, x)),
        l,
        [0.0, 0.0],
        ),
        MOI.Nonnegatives(2),
    )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
        → MathOptInterface.Nonnegatives}(1)
julia> MOI.modify(model, c, MOI.MultirowChange(x, [(1, 3.0), (2, 4.0)]));
julia> new_f = MOI.VectorAffineFunction(
        [
```

```
MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(3.0, x)),
MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(4.0, x)),
],
[0.0, 0.0],
);
```

<code>julia> MOI.get(model, MOI.ConstraintFunction(), c) \approx new_f true</code>

Chapter 35

Background

35.1 Duality

Conic duality is the starting point for MOI's duality conventions. When all functions are affine (or coordinate projections), and all constraint sets are closed convex cones, the model may be called a conic optimization problem.

For a minimization problem in geometric conic form, the primal is:

$$\min_{x \in \mathbb{R}^n} \qquad \qquad a_0^T x + b_0 \tag{35.1}$$

s.t.
$$A_i x + b_i \in \mathcal{C}_i$$
 $i = 1 \dots m$ (35.2)

and the dual is a maximization problem in standard conic form:

$$\max_{y_1,\dots,y_m} \qquad -\sum_{i=1}^m b_i^T y_i + b_0 \tag{35.3}$$

s.t.
$$a_0 - \sum_{i=1}^m A_i^T y_i = 0$$
 (35.4)

$$y_i \in \mathcal{C}_i^*$$
 $i = 1 \dots m$ (35.5)

where each C_i is a closed convex cone and C_i^* is its dual cone. For a maximization problem in geometric conic form, the primal is:

$$\max_{x \in \mathbb{R}^n} \qquad \qquad a_0^T x + b_0 \tag{35.6}$$

s.t.
$$A_i x + b_i \in \mathcal{C}_i$$
 $i = 1 \dots m$ (35.7)

and the dual is a minimization problem in standard conic form:

$$\min_{y_1,...,y_m} \sum_{i=1}^m b_i^T y_i + b_0$$
(35.8)

s.t.
$$a_0 + \sum_{i=1}^m A_i^T y_i = 0$$
 (35.9)

$$y_i \in \mathcal{C}^*_i$$
 $i = 1 \dots m$ (35.10)

A linear inequality constraint $a^Tx + b \ge c$ is equivalent to $a^Tx + b - c \in \mathbb{R}_+$, and $a^Tx + b \le c$ is equivalent to $a^Tx + b - c \in \mathbb{R}_-$. Variable-wise constraints are affine constraints with the appropriate identity mapping in place of A_i .

For the special case of minimization LPs, the MOI primal form can be stated as:

s.t.

$$\min_{x \in \mathbb{R}^n} \qquad \qquad a_0^T x + b_0 \tag{35.11}$$

 $A_1 x \ge b_1$ (35.12) $A_2 x \leq b_2$ (35.13)

$$A_3 x = b_3$$
 (35.14)

(35.14)

By applying the stated transformations to conic form, taking the dual, and transforming back into linear inequality form, one obtains the following dual:

$$\max_{y_1, y_2, y_3} \qquad b_1^T y_1 + b_2^T y_2 + b_3^T y_3 + b_0 \tag{35.15}$$

$$A_1^T y_1 + A_2^T y_2 + A_3^T y_3 = a_0$$
(35.16)

 $y_1 \ge 0$ (35.17) $y_2 \leq 0$ (35.18)

For maximization LPs, the MOI primal form can be stated as:

s.t.

s.t.

s.t.

$$\max_{x \in \mathbb{R}^n} \qquad \qquad a_0^T x + b_0 \tag{35.19}$$

$$A_1 x \ge b_1 \tag{35.20}$$

$$A_2 x \le b_2 \tag{35.21}$$

$$A_3 x = b_3$$
 (35.22)

and similarly, the dual is:

| \min_{y_1,y_2,y_3} | $ \lim_{y_2,y_3} -b_1^T y_1 - b_2^T y_2 - b_3^T y_3 + b_0 $ | |
|----------------------|---|---------|
| | $AT \rightarrow AT \rightarrow AT$ | (25.24) |

$$A_1^T y_1 + A_2^T y_2 + A_3^T y_3 = -a_0$$
(35.24)
$$y_1 \ge 0$$
(35.25)

$$y_1 \ge 0$$
 (35.25)
 $y_2 \le 0$ (35.26)

Warning

For the LP case, the signs of the feasible dual variables depend only on the sense of the corresponding primal inequality and not on the objective sense.

Duality and scalar product

The scalar product is different from the canonical one for the sets PositiveSemidefiniteConeTriangle, LogDetConeTriangle, RootDetConeTriangle.

If the set C_i of the section Duality is one of these three cones, then the rows of the matrix A_i corresponding to off-diagonal entries are twice the value of the coefficients field in the VectorAffineFunction for the corresponding rows. See PositiveSemidefiniteConeTriangle for details.

Dual for problems with quadratic functions

Quadratic Programs (QPs)

For quadratic programs with only affine conic constraints,

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} x^T Q_0 x + a_0^T x + b_0$$

s.t. $A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \dots m$

with cones $\mathcal{C}_i \subseteq \mathbb{R}^{m_i}$ for $i = 1, \dots, m$, consider the Lagrangian function

$$L(x,y) = \frac{1}{2}x^{T}Q_{0}x + a_{0}^{T}x + b_{0} - \sum_{i=1}^{m} y_{i}^{T}(A_{i}x + b_{i}).$$

Let z(y) denote $\sum_{i=1}^m A_i^T y_i - a_0$, the Lagrangian can be rewritten as

$$L(x,y) = \frac{1}{2}x^{T}Q_{0}x - z(y)^{T}x + b_{0} - \sum_{i=1}^{m} y_{i}^{T}b_{i}.$$

The condition $\nabla_x L(x,y)=0$ gives

$$0 = \nabla_x L(x, y) = Q_0 x + a_0 - \sum_{i=1}^m y_i^T b_i$$

which gives $Q_0 x = z(y)$. This allows to obtain that

$$\min_{x \in \mathbb{R}^n} L(x, y) = -\frac{1}{2}x^T Q_0 x + b_0 - \sum_{i=1}^m y_i^T b_i$$

so the dual problem is

$$\max_{y_i \in \mathcal{C}_i^*} \min_{x \in \mathbb{R}^n} -\frac{1}{2} x^T Q_0 x + b_0 - \sum_{i=1}^m y_i^T b_i.$$

If Q_0 is invertible, we have $\boldsymbol{x} = Q_0^{-1} \boldsymbol{z}(\boldsymbol{y})$ hence

$$\min_{x \in \mathbb{R}^n} L(x, y) = -\frac{1}{2} z(y)^T Q_0^{-1} z(y) + b_0 - \sum_{i=1}^m y_i^T b_i$$

so the dual problem is

$$\max_{y_i \in \mathcal{C}_i^*} -\frac{1}{2} z(y)^T Q_0^{-1} z(y) + b_0 - \sum_{i=1}^m y_i^T b_i.$$

Quadratically Constrained Quadratic Programs (QCQPs)

Given a problem with both quadratic function and quadratic objectives:

$$\min_{x \in \mathbb{R}^n} \qquad \qquad \frac{1}{2} x^T Q_0 x + a_0^T x + b_0$$

s.t.
$$\frac{1}{2} x^T Q_i x + a_i^T x + b_i \in \mathcal{C}_i \qquad \qquad i = 1 \dots m.$$

with cones $\mathcal{C}_i \subseteq \mathbb{R}$ for $i=1\dots m$, consider the Lagrangian function

$$L(x,y) = \frac{1}{2}x^{T}Q_{0}x + a_{0}^{T}x + b_{0} - \sum_{i=1}^{m} y_{i}(\frac{1}{2}x^{T}Q_{i}x + a_{i}^{T}x + b_{i})$$

A pair of primal-dual variables (x^\star,y^\star) is optimal if

• x^{\star} is a minimizer of

$$\min_{x \in \mathbb{R}^n} L(x, y^\star).$$

That is,

$$0 = \nabla_x L(x, y^*) = Q_0 x + a_0 - \sum_{i=1}^m y_i^* (Q_i x + a_i).$$

• and y^{\star} is a maximizer of

$$\max_{y_i \in \mathcal{C}_i^*} L(x^\star, y).$$

That is, for all i = 1, ..., m, $\frac{1}{2}x^TQ_ix + a_i^Tx + b_i$ is either zero or in the normal cone of \mathcal{C}_i^* at y^* . For instance, if \mathcal{C}_i is $\{z \in \mathbb{R} : z \leq 0\}$, this means that if $\frac{1}{2}x^TQ_ix + a_i^Tx + b_i$ is nonzero at x^* then $y_i^* = 0$. This is the classical complementary slackness condition.

If C_i is a vector set, the discussion remains valid with $y_i(\frac{1}{2}x^TQ_ix + a_i^Tx + b_i)$ replaced with the scalar product between y_i and the vector of scalar-valued quadratic functions.

Dual for square semidefinite matrices

The set PositiveSemidefiniteConeTriangle is a self-dual. That is, querying ConstraintDual of a PositiveSemidefiniteConeT constraint returns a vector that is itself a member of PositiveSemidefiniteConeTriangle.

However, the dual of PositiveSemidefiniteConeSquare is not so straight forward. This section explains the duality convention we use, and how it is derived.

Info

If you have a PositiveSemidefiniteConeSquare constraint, the result matrix A from ConstraintDual is not positive semidefinite. However, $A + A^{\top}$ is positive semidefinite.

Let S_+ be the cone of symmetric semidefinite matrices in the $\frac{n(n+1)}{2}$ dimensional space of symmetric $\mathbb{R}^{n \times n}$ matrices. That is, S_+ is the set PositiveSemidefiniteConeTriangle. It is well known that S_+ is a self-dual proper cone.

Let \mathcal{P}_+ be the cone of symmetric semidefinite matrices in the n^2 dimensional space of $\mathbb{R}^{n \times n}$ matrices. That is \mathcal{P}_+ is the set PositiveSemidefiniteConeSquare.

In addition, let \mathcal{D}_+ be the cone of matrices A such that $A + A^\top \in \mathcal{P}_+$.

 \mathcal{P}_+ is not proper because it is not solid (it is not n^2 dimensional), so it is not necessarily true that $\mathcal{P}_+^{**} = \mathcal{P}_+$.

However, this is the case, because we will show that $\mathcal{P}^*_+ = \mathcal{D}_+$ and $\mathcal{D}^*_+ = \mathcal{P}_+$.

First, let us see why $\mathcal{P}^*_+ = \mathcal{D}_+$.

If B is symmetric, then

$$\langle A, B \rangle = \langle A^{\top}, B^{\top} \rangle = \langle A^{\top}, B \rangle$$

so

$$2\langle A, B \rangle = \langle A, B \rangle + \langle A^{\top}, B \rangle = \langle A + A^{\top}, B \rangle.$$

Therefore, $\langle A, B \rangle \geq 0$ for all $B \in \mathcal{P}_+$ if and only if $\langle A + A^{\top}, B \rangle \geq 0$ for all $B \in \mathcal{P}_+$. Since $A + A^{\top}$ is symmetric, and we know that \mathcal{S}_+ is self-dual, we have shown that \mathcal{P}_+^* is the set of matrices A such that $A + A^{\top} \in \mathcal{P}_+$.

Second, let us see why $\mathcal{D}^*_+ = \mathcal{P}_+.$

Since $A \in \mathcal{D}_+$ implies that $A^{\top} \in \mathcal{D}_+$, $B \in \mathcal{D}_+^*$ means that $\langle A + A^{\top}, B \rangle \ge 0$ for all $A \in \mathcal{D}_+$, and hence $B \in mathcal P_+$.

To see why it should be symmetric, simply notice that if $B_{i,j} < B_{j,i}$, then $\langle A, B \rangle$ can be made arbitrarily small by setting $A_{i,j} = A_{i,j} + s$ and $A_{j,i} = A_{j,i} - s$, with s arbitrarily large, and A stays in \mathcal{D}_+ because $A + A^\top$ does not change.

Typically, the primal/dual pair for semidefinite programs is presented as:

$$\min\langle C, X \rangle$$
 (35.27)

s.t.
$$\langle A_k, X \rangle = b_k \forall k$$
 (35.28)

$$X \in \mathcal{S}_+ \tag{35.29}$$

with the dual

$$\max\sum_{k} b_k y_k \tag{35.30}$$

s.t.
$$C - \sum A_k y_k \in \mathcal{S}_+$$
 (35.31)

If we allow ${\cal A}_k$ to be non-symmetric, we should instead use:

$$\min\langle C, X \rangle$$
 (35.32)

s.t.
$$\langle A_k, X \rangle = b_k \forall k$$
 (35.33)

$$X \in \mathcal{D}_+ \tag{35.34}$$

with the dual

$$\max \sum b_k y_k \tag{35.35}$$

s.t.
$$C - \sum A_k y_k \in \mathcal{P}_+$$
 (35.36)

This is implemented as:

$$\min\langle C, Z \rangle + \langle C - C^{\top}, S \rangle \tag{35.37}$$

s.t.
$$\langle A_k, Z \rangle + \langle A_k - A_k^{\top}, S \rangle = b_k \forall k$$
 (35.38)

$$Z \in \mathcal{S}_+$$
 (35.39)

with the dual

$$\max \sum b_k y_k \tag{35.40}$$

s.t.
$$C + C^{\top} - \sum (A_k + A_k^{\top}) y_k \in \mathcal{S}_+$$
 (35.41)

$$C - C^{\top} - \sum (A_k - A_k^{\top}) y_k = 0$$
(35.42)

and we recover $Z = X + X^{\top}$.

35.2 Infeasibility certificates

When given a conic problem that is infeasible or unbounded, some solvers can produce a certificate of infeasibility. This page explains what a certificate of infeasibility is, and the related conventions that MathOptInterface adopts.

Conic duality

MathOptInterface uses conic duality to define infeasibility certificates. A full explanation is given in the section Duality, but here is a brief overview.

Minimization problems

For a minimization problem in geometric conic form, the primal is:

$$\min_{x \in \mathbb{R}^n} \qquad a_0^{\top} x + b_0 \tag{35.43}$$
 s.t.
$$A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \dots m, \tag{35.44}$$

and the dual is a maximization problem in standard conic form:

$$\max_{y_1,...,y_m} -\sum_{i=1}^m b_i^\top y_i + b_0$$
(35.45)

s.t.
$$a_0 - \sum_{i=1}^m A_i^\top y_i = 0$$
 (35.46)

$$y_i \in \mathcal{C}_i^* \qquad \qquad i = 1 \dots m, \qquad (35.47)$$

where each \mathcal{C}_i is a closed convex cone and \mathcal{C}_i^* is its dual cone.

Maximization problems

For a maximization problem in geometric conic form, the primal is:

$$\max_{x \in \mathbb{R}^n} \qquad a_0^\top x + b_0 \tag{35.48}$$

s.t.
$$A_i x + b_i \in \mathcal{C}_i$$
 $i = 1 \dots m,$ (35.49)

and the dual is a minimization problem in standard conic form:

$$\min_{y_1,...,y_m} \sum_{i=1}^m b_i^\top y_i + b_0$$
(35.50)

s.t.
$$a_0 + \sum_{i=1}^m A_i^\top y_i = 0$$
 (35.51)

$$y_i \in \mathcal{C}_i^*$$
 $i = 1 \dots m.$ (35.52)

Unbounded problems

A problem is unbounded if and only if:

- 1. there exists a feasible primal solution
- 2. the dual is infeasible.

A feasible primal solution—if one exists—can be obtained by setting ObjectiveSense to FEASIBILITY_SENSE before optimizing. Therefore, most solvers stop after they prove the dual is infeasible via a certificate of dual infeasibility, but *before* they have found a feasible primal solution. This is also the reason that MathOptInterface defines the DUAL_INFEASIBLE status instead of UNBOUNDED.

A certificate of dual infeasibility is an improving ray of the primal problem. That is, there exists some vector d such that for all $\eta > 0$:

$$A_i(x+\eta d)+b_i\in \mathcal{C}_i, \ i=1\ldots m,$$

and (for minimization problems):

$$a_0^{\top}(x+\eta d) + b_0 < a_0^{\top}x + b_0,$$

for any feasible point x. The latter simplifies to $a_0^{\top}d < 0$. For maximization problems, the inequality is reversed, so that $a_0^{\top}d > 0$.

If the solver has found a certificate of dual infeasibility:

- TerminationStatus must be DUAL INFEASIBLE
- PrimalStatus must be INFEASIBILITY_CERTIFICATE
- VariablePrimal must be the corresponding value of d
- ConstraintPrimal must be the corresponding value of $A_i d$
- ObjectiveValue must be the value $a_0^{\top} d$. Note that this is the value of the objective function at d, ignoring the constant b_0.

Note

The choice of whether to scale the ray d to have magnitude 1 is left to the solver.

Infeasible problems

A certificate of primal infeasibility is an improving ray of the dual problem. However, because infeasibility is independent of the objective function, we first homogenize the primal problem by removing its objective.

For a minimization problem, a dual improving ray is some vector d such that for all $\eta > 0$:

$$-\sum_{i=1}^{m} A_i^{\top}(y_i + \eta d_i) = 0$$
(35.53)

$$(y_i + \eta d_i) \in \mathcal{C}_i^* \qquad \qquad i = 1 \dots m, \tag{35.54}$$

and:

$$-\sum_{i=1}^{m} b_i^{\top}(y_i + \eta d_i) > -\sum_{i=1}^{m} b_i^{\top} y_i,$$

for any feasible dual solution y. The latter simplifies to $-\sum_{i=1}^{m} b_i^{\top} d_i > 0$. For a maximization problem, the inequality is $\sum_{i=1}^{m} b_i^{\top} d_i < 0$. (Note that these are the same inequality, modulo a - sign.)

If the solver has found a certificate of primal infeasibility:

- TerminationStatus must be INFEASIBLE
- DualStatus must be INFEASIBILITY_CERTIFICATE
- ConstraintDual must be the corresponding value of d
- DualObjectiveValue must be the value $-\sum_{i=1}^{m} b_i^{\top} d_i$ for minimization problems and $\sum_{i=1}^{m} b_i^{\top} d_i$ for maximization problems.

Note

The choice of whether to scale the ray d to have magnitude 1 is left to the solver.

Infeasibility certificates of variable bounds

Many linear solvers (for example, Gurobi) do not provide explicit access to the primal infeasibility certificate of a variable bound. However, given a set of linear constraints:

$$l_A \le Ax \le u_A \tag{35.55}$$

$$l_x \le x \le u_x,\tag{35.56}$$

the primal certificate of the variable bounds can be computed using the primal certificate associated with the affine constraints, d. (Note that d will have one element for each row of the A matrix, and that some or all of the elements in the vectors l_A and u_A may be $\pm \infty$. If both l_A and u_A are finite for some row, the corresponding element in 'd must be 0.)

Given d, compute $\bar{d} = d^{\top}A$. If the bound is finite, a certificate for the lower variable bound of x_i is $\max\{\bar{d}_i, 0\}$, and a certificate for the upper variable bound is $\min\{\bar{d}_i, 0\}$.

35.3 Naming conventions

MOI follows several conventions for naming functions and structures. These should also be followed by packages extending MOI.

Sets

Sets encode the structure of constraints. Their names should follow the following conventions:

- Abstract types in the set hierarchy should begin with Abstract and end in Set, for example, AbstractScalarSet, AbstractVectorSet.
- Vector-valued conic sets should end with Cone, for example, NormInfinityCone, SecondOrderCone.
- Vector-valued Cartesian products should be plural and not end in Cone, for example, Nonnegatives, not NonnegativeCone.
- Matrix-valued conic sets should provide two representations: ConeSquare and ConeTriangle, for example, RootDetConeTriangle and RootDetConeSquare. See Matrix cones for more details.
- Scalar sets should be singular, not plural, for example, Integer, not Integers.
- As much as possible, the names should follow established conventions in the domain where this set is used: for instance, convex sets should have names close to those of CVX, and constraint-programming sets should follow MiniZinc's constraints.

Chapter 36

API Reference

36.1 Standard form

Functions

MathOptInterface.AbstractFunction - Type.

AbstractFunction

Abstract supertype for function objects.

Required methods

All functions must implement:

- Base.copy
- Base.isapprox
- constant

Abstract subtypes of AbstractFunction may require additional methods to be implemented.

source

MathOptInterface.output_dimension - Function.

output_dimension(f::AbstractFunction)

 $Return 1 if f is an \ Abstract Scalar Function, or the number of output components if f is an \ Abstract Vector Function.$

source

MathOptInterface.constant - Function.

```
constant(f::AbstractFunction[, ::Type{T}]) where {T}
```

Returns the constant term of a scalar-valued function, or the constant vector of a vector-valued function. If f is untyped and T is provided, returns zero(T).

source

```
constant(set::Union{EqualTo,GreaterThan,LessThan,Parameter})
```

Returns the constant term of the set set.

Example

```
julia> MOI.constant(MOI.GreaterThan(1.0))
1.0
julia> MOI.constant(MOI.LessThan(2.5))
2.5
julia> MOI.constant(MOI.EqualTo(3))
3
julia> MOI.constant(MOI.Parameter(4.5))
4.5
```

source

Scalar functions

MathOptInterface.AbstractScalarFunction - Type.

abstract type AbstractScalarFunction <: AbstractFunction</pre>

Abstract supertype for scalar-valued AbstractFunctions.

source

MathOptInterface.VariableIndex - Type.

VariableIndex

A type-safe wrapper for Int64 for use in referencing variables in a model. To allow for deletion, indices need not be consecutive.

source

MathOptInterface.ScalarAffineTerm - Type.

ScalarAffineTerm{T}(coefficient::T, variable::VariableIndex) where {T}

Represents the scalar-valued term coefficient * variable.

Example

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

```
julia> MOI.ScalarAffineTerm(2.0, x)
MathOptInterface.ScalarAffineTerm{Float64}(2.0, MOI.VariableIndex(1))
```

source

MathOptInterface.ScalarAffineFunction - Type.

```
ScalarAffineFunction{T}(
    terms::Vector{ScalarAffineTerm{T}},
    constant::T,
) where {T}
```

Represents the scalar-valued affine function $a^{\top}x + b$, where:

- $a^{\top}x$ is represented by the vector of ScalarAffineTerms
- *b* is a scalar constant::T

Duplicates

Duplicate variable indices in terms are accepted, and the corresponding coefficients are summed together.

Example

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
julia> terms = [MOI.ScalarAffineTerm(2.0, x), MOI.ScalarAffineTerm(3.0, x)]
2-element Vector{MathOptInterface.ScalarAffineTerm{Float64}}:
MathOptInterface.ScalarAffineTerm{Float64}(2.0, MOI.VariableIndex(1))
MathOptInterface.ScalarAffineTerm{Float64}(3.0, MOI.VariableIndex(1))
```

```
julia> f = MOI.ScalarAffineFunction(terms, 4.0)
4.0 + 2.0 MOI.VariableIndex(1) + 3.0 MOI.VariableIndex(1)
```

source

MathOptInterface.ScalarQuadraticTerm - Type.

```
ScalarQuadraticTerm{T}(
    coefficient::T,
    variable_1::VariableIndex,
    variable_2::VariableIndex,
) where {T}
```

Represents the scalar-valued term cx_ix_i where c is coefficient, x_i is variable_1 and x_i is variable_2.

Example

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

```
julia> MOI.ScalarQuadraticTerm(2.0, x, x)
MathOptInterface.ScalarQuadraticTerm{Float64}(2.0, MOI.VariableIndex(1), MOI.VariableIndex(1))
```

source

MathOptInterface.ScalarQuadraticFunction - Type.

```
ScalarQuadraticFunction{T}(
    quadratic_terms::Vector{ScalarQuadraticTerm{T}},
    affine_terms::Vector{ScalarAffineTerm{T}},
    constant::T,
) wher {T}
```

The scalar-valued quadratic function $\frac{1}{2}x^{\top}Qx + a^{\top}x + b$, where:

- Q is the symmetric matrix given by the vector of ScalarQuadraticTerms
- $a^{\top}x$ is a sparse vector given by the vector of ScalarAffineTerms
- *b* is the scalar constant::T.

Duplicates

Duplicate indices in quadratic_terms or affine_terms are accepted, and the corresponding coefficients are summed together.

In quadratic_terms, "mirrored" indices, (q, r) and (r, q) where r and q are VariableIndexes, are considered duplicates; only one needs to be specified.

The 0.5 factor

Coupled with the interpretation of mirrored indices, the 0.5 factor in front of the Q matrix is a common source of bugs.

As a rule, to represent $a * x^2 + b * x * y$:

- The coefficient *a* in front of squared variables (diagonal elements in *Q*) must be doubled when creating a ScalarQuadraticTerm
- The coefficient b in front of off-diagonal elements in Q should be left as b, be cause the mirrored index b * y * x will be implicitly added.

Example

To represent the function $f(x, y) = 2 * x^2 + 3 * x * y + 4 * x + 5$, do:

```
julia> x = MOI.VariableIndex(1);
julia> y = MOI.VariableIndex(2);
julia> constant = 5.0;
julia> affine_terms = [MOI.ScalarAffineTerm(4.0, x)];
julia> quadratic_terms = [
            MOI.ScalarQuadraticTerm(4.0, x, x), # Note the changed coefficient
            MOI.ScalarQuadraticTerm(3.0, x, y),
            ]
2-element Vector{MathOptInterface.ScalarQuadraticTerm{Float64}:
            MathOptInterface.ScalarQuadraticTerm{Float64}:
            MathOptInterface.ScalarQuadraticTerm{Float64}(3.0, MOI.VariableIndex(1), MOI.VariableIndex(1))
            MathOptInterface.ScalarQuadraticTerm{Float64}(3.0, MOI.VariableIndex(1), MOI.VariableIndex(2))
julia> f = MOI.ScalarQuadraticFunction(quadratic_terms, affine_terms, constant)
5.0 + 4.0 MOI.VariableIndex(1) + 2.0 MOI.VariableIndex(1)<sup>2</sup> + 3.0
            → MOI.VariableIndex(1)*MOI.VariableIndex(2)
```

source

MathOptInterface.ScalarNonlinearFunction - Type.

```
ScalarNonlinearFunction(head::Symbol, args::Vector{Any})
```

The scalar-valued nonlinear function head (args...), represented as a symbolic expression tree, with the call operator head and ordered arguments in args.

head

The head::Symbol must be an operator supported by the model.

The default list of supported univariate operators is given by:

• Nonlinear.DEFAULT_UNIVARIATE_OPERATORS

and the default list of supported multivariate operators is given by:

• Nonlinear.DEFAULT_MULTIVARIATE_OPERATORS

Additional operators can be registered by setting a UserDefinedFunction attribute.

See the full list of operators supported by a ModelLike by querying ListOfSupportedNonlinearOperators.

args

The vector args contains the arguments to the nonlinear function. If the operator is univariate, it must contain one element. Otherwise, it may contain multiple elements.

Each element must be one of the following:

- A constant value of type T<:Real
- A VariableIndex

- A ScalarAffineFunction
- A ScalarQuadraticFunction
- A ScalarNonlinearFunction

Unsupported operators

If the optimizer does not support head, an UnsupportedNonlinearOperator error will be thrown.

There is no guarantee about when this error will be thrown; it may be thrown when the function is first added to the model, or it may be thrown when optimize! is called.

Example

To represent the function $f(x) = sin(x)^2$, do:

source

Vector functions

MathOptInterface.AbstractVectorFunction - Type.

abstract type AbstractVectorFunction <: AbstractFunction</pre>

Abstract supertype for vector-valued AbstractFunctions.

Required methods

All subtypes of AbstractVectorFunction must implement:

• output dimension

source

MathOptInterface.VectorOfVariables - Type.

VectorOfVariables(variables::Vector{VariableIndex}) <: AbstractVectorFunction</pre>

The vector-valued function f(x) = variables, where variables is a subset of VariableIndexes in the model.

The list of variables may contain duplicates.

Example

```
julia> x = MOI.VariableIndex.(1:2)
2-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(1)
MOI.VariableIndex(2)

/
MOI.VariableIndex(1)
MOI.VariableIndex(2)

julia> MOI.output_dimension(f)
3
```

source

MathOptInterface.VectorAffineTerm - Type.

```
VectorAffineTerm{T}(
    output_index::Int64,
    scalar_term::ScalarAffineTerm{T},
) where {T}
```

A VectorAffineTerm is a scalar_term that appears in the output_index row of the vector-valued VectorAffineFunction or VectorQuadraticFunction.

Example

```
julia> x = MOI.VariableIndex(1);
```

```
julia> MOI.VectorAffineTerm(Int64(2), MOI.ScalarAffineTerm(3.0, x))
MathOptInterface.VectorAffineTerm{Float64}(2, MathOptInterface.ScalarAffineTerm{Float64}(3.0,
→ MOI.VariableIndex(1)))
```

source

MathOptInterface.VectorAffineFunction - Type.

```
VectorAffineFunction{T}(
    terms::Vector{VectorAffineTerm{T}},
    constants::Vector{T},
) where {T}
```

The vector-valued affine function Ax + b, where:

- Ax is the sparse matrix given by the vector of VectorAffineTerms
- b is the vector constants

Duplicates

Duplicate indices in the A are accepted, and the corresponding coefficients are summed together.

Example

```
julia> x = MOI.VariableIndex(1);
julia> terms = [
            MOI.VectorAffineTerm(Int64(1), MOI.ScalarAffineTerm(2.0, x)),
            MOI.VectorAffineTerm(Int64(2), MOI.ScalarAffineTerm(3.0, x)),
        ];
julia> f = MOI.VectorAffineFunction(terms, [4.0, 5.0])
[4.0 + 2.0 MOI.VariableIndex(1)]
[5.0 + 3.0 MOI.VariableIndex(1)]
julia> MOI.output_dimension(f)
2
```

source

MathOptInterface.VectorQuadraticTerm - Type.

```
VectorQuadraticTerm{T}(
    output_index::Int64,
    scalar_term::ScalarQuadraticTerm{T},
) where {T}
```

A VectorQuadraticTerm is a ScalarQuadraticTerm scalar_term that appears in the output_index row of the vector-valued VectorQuadraticFunction.

Example

julia> x = MOI.VariableIndex(1);

```
julia> MOI.VectorQuadraticTerm(Int64(2), MOI.ScalarQuadraticTerm(3.0, x, x))
MathOptInterface.VectorQuadraticTerm{Float64}(2,
```

```
\hookrightarrow MathOptInterface.ScalarQuadraticTerm{Float64}(3.0, MOI.VariableIndex(1),
```

```
→ MOI.VariableIndex(1)))
```

source

MathOptInterface.VectorQuadraticFunction - Type.

```
VectorQuadraticFunction{T}(
    quadratic_terms::Vector{VectorQuadraticTerm{T}},
    affine_terms::Vector{VectorAffineTerm{T}},
    constants::Vector{T},
) where {T}
```

The vector-valued quadratic function with ith component ("output index") defined as $\frac{1}{2}x^{\top}Q_ix + a_i^{\top}x + b_i$, where:

- $\frac{1}{2}x^{\top}Q_ix$ is the symmetric matrix given by the VectorQuadraticTerm elements in quadratic_terms with output_index == i
- a_i x is the sparse vector given by the VectorAffineTerm elements in affine_terms with output_index
 i
- b_i is a scalar given by constants[i]

Duplicates

Duplicate indices in quadratic_terms and affine_terms with the same output_index are handled in the same manner as duplicates in ScalarQuadraticFunction.

Example

```
julia> x = MOI.VariableIndex(1);
julia> y = MOI.VariableIndex(2);
julia> constants = [4.0, 5.0];
julia> affine_terms = [
           MOI.VectorAffineTerm(Int64(1), MOI.ScalarAffineTerm(2.0, x)),
           MOI.VectorAffineTerm(Int64(2), MOI.ScalarAffineTerm(3.0, x)),
       ];
julia> quad_terms = [
        MOI.VectorQuadraticTerm(Int64(1), MOI.ScalarQuadraticTerm(2.0, x, x)),
        MOI.VectorQuadraticTerm(Int64(2), MOI.ScalarQuadraticTerm(3.0, x, y)),
           1;
julia> f = MOI.VectorQuadraticFunction(quad_terms, affine_terms, constants)
4.0 + 2.0 MOI.VariableIndex(1) + 1.0 MOI.VariableIndex(1)<sup>2</sup>
[5.0 + 3.0 MOI.VariableIndex(1) + 3.0 MOI.VariableIndex(1)*MOI.VariableIndex(2)
julia> MOI.output_dimension(f)
2
```

source

MathOptInterface.VectorNonlinearFunction - Type.

VectorNonlinearFunction(args::Vector{ScalarNonlinearFunction})

The vector-valued nonlinear function composed of a vector of ScalarNonlinearFunction.

args

The vector args contains the scalar elements of the nonlinear function. Each element must be a ScalarNonlinearFunction, but if you pass a Vector{Any}, the elements can be automatically converted from one of the following:

- A constant value of type T<:Real
- A VariableIndex
- A ScalarAffineFunction
- A ScalarQuadraticFunction
- A ScalarNonlinearFunction

Example

To represent the function $f(x) = [sin(x)^2, x]$, do:

```
julia> MOI.VectorNonlinearFunction([g, x])
```

```
^(sin(MOI.VariableIndex(1)), 2.0)
+(MOI.VariableIndex(1))
```

Note the automatic conversion from x to +(x).

source

Sets

MathOptInterface.AbstractSet - Type.

AbstractSet

Abstract supertype for set objects used to encode constraints.

Required methods

For sets of type S with isbitstype(S) == false, you must implement:

- Base.copy(set::S)
- Base.:(==)(x::S, y::S)

Subtypes of AbstractSet such as AbstractScalarSet and AbstractVectorSet may prescribe additional required methods.

Optional methods

You may optionally implement:

• dual_set

• dual_set_type

Note for developers

When creating a new set, the set struct must not contain any VariableIndex or ConstraintIndex objects.

source

MathOptInterface.AbstractScalarSet - Type.

AbstractScalarSet

Abstract supertype for subsets of \mathbb{R} .

source

MathOptInterface.AbstractVectorSet - Type.

AbstractVectorSet

Abstract supertype for subsets of \mathbb{R}^n for some n.

Required methods

All AbstractVectorSets of type S must implement:

- dimension, unless the dimension is stored in the set.dimension field
- Utilities.set_dot, unless the dot product between two vectors in the set is equivalent to LinearAlgebra.dot.

source

Utilities

MathOptInterface.dimension - Function.

dimension(set::AbstractSet)

Return the output_dimension that an AbstractFunction should have to be used with the set set.

Example

```
julia> MOI.dimension(MOI.Reals(4))
4
julia> MOI.dimension(MOI.LessThan(3.0))
1
julia> MOI.dimension(MOI.PositiveSemidefiniteConeTriangle(2))
3
```
MathOptInterface.dual_set - Function.

dual_set(set::AbstractSet)

Return the dual set of set, that is the dual cone of the set. This follows the definition of duality discussed in Duality.

See Dual cone for more information.

If the dual cone is not defined it returns an error.

Example

julia> MOI.dual_set(MOI.Reals(4))
MathOptInterface.Zeros(4)

julia> MOI.dual_set(MOI.SecondOrderCone(5))
MathOptInterface.SecondOrderCone(5)

julia> MOI.dual_set(MOI.ExponentialCone())
MathOptInterface.DualExponentialCone()

source

MathOptInterface.dual_set_type - Function.

```
dual_set_type(S::Type{<:AbstractSet})</pre>
```

Return the type of dual set of sets of type S, as returned by dual_set. If the dual cone is not defined it returns an error.

Example

julia> MOI.dual_set_type(MOI.Reals)
MathOptInterface.Zeros

julia> MOI.dual_set_type(MOI.SecondOrderCone)
MathOptInterface.SecondOrderCone

julia> MOI.dual_set_type(MOI.ExponentialCone)
MathOptInterface.DualExponentialCone

source

MathOptInterface.constant - Method.

constant(set::Union{EqualTo,GreaterThan,LessThan,Parameter})

Returns the constant term of the set set.

Example

```
julia> MOI.constant(MOI.GreaterThan(1.0))
1.0
julia> MOI.constant(MOI.LessThan(2.5))
2.5
julia> MOI.constant(MOI.EqualTo(3))
3
julia> MOI.constant(MOI.Parameter(4.5))
4.5
```

MathOptInterface.supports_dimension_update - Function.

supports_dimension_update(S::Type{<:MOI.AbstractVectorSet})</pre>

Return a Bool indicating whether the elimination of any dimension of n-dimensional sets of type S give an n-1-dimensional set S. By default, this function returns false so it should only be implemented for sets that supports dimension update.

For instance, supports_dimension_update(MOI.Nonnegatives) is true because the elimination of any dimension of the n-dimensional nonnegative orthant gives the n-1-dimensional nonnegative orthant. However supports_dimension_update(MOI.ExponentialCone) is false.

source

MathOptInterface.update_dimension - Function.

update_dimension(s::AbstractVectorSet, new_dim::Int)

Returns a set with the dimension modified to new_dim.

source

Scalar sets

List of recognized scalar sets.

MathOptInterface.GreaterThan - Type.

GreaterThan{T<:Real}(lower::T)</pre>

The set $[lower, \infty) \subseteq \mathbb{R}$.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

source

MathOptInterface.LessThan - Type.

LessThan{T<:Real}(upper::T)</pre>

The set $(-\infty, upper] \subseteq \mathbb{R}$.

Example

julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

source

MathOptInterface.EqualTo - Type.

EqualTo{T<:Number}(value::T)</pre>

The set containing the single point $\{value\} \subseteq \mathbb{R}$.

Example

julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

source

MathOptInterface.Interval - Type.

Interval{T<:Real}(lower::T, upper::T)</pre>

The interval $[lower, upper] \subseteq \mathbb{R} \cup \{-\infty, +\infty\}.$

If lower or upper is - Inf or Inf, respectively, the set is interpreted as a one-sided interval.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

source

MathOptInterface.Integer - Type.

Integer()

The set of integers, \mathbb{Z} .

Example

julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

julia> MOI.add_constraint(model, x, MOI.Integer())
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}()

source

MathOptInterface.ZeroOne - Type.

ZeroOne()

The set $\{0, 1\}$.

Variables belonging to the Zero0ne set are also known as "binary" variables.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

```
julia> MOI.add_constraint(model, x, MOI.ZeroOne())
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(1)
```

source

MathOptInterface.Semicontinuous - Type.

```
Semicontinuous{T<:Real}(lower::T, upper::T)</pre>
```

The set $\{0\} \cup [lower, upper]$.

Example

julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

```
julia> MOI.add_constraint(model, x, MOI.Semicontinuous(2.0, 3.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
→ MathOptInterface.Semicontinuous{Float64}}(1)
```

source

MathOptInterface.Semiinteger - Type.

Semiinteger{T<:Real}(lower::T, upper::T)</pre>

The set $\{0\} \cup \{lower, lower + 1, \dots, upper - 1, upper\}$.

Note that if lower and upper are not equivalent to an integer, then the solver may throw an error, or it may round up lower and round down upper to the nearest integers.

Example

julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

MathOptInterface.Parameter - Type.

Parameter{T<:Number}(value::T)</pre>

The set containing the single point $\{value\} \subseteq \mathbb{R}$.

The Parameter set is conceptually similar to the EqualTo set, except that a variable constrained to the Parameter set cannot have other constraints added to it, and the Parameter set can never be deleted. Thus, solvers are free to treat the variable as a constant, and they need not add it as a decision variable to the model.

Because of this behavior, you must add parameters using add_constrained_variable, and solvers should declare supports_add_constrained_variable and not supports_constraint for the Parameter set.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

```
julia> MOI.set(model, MOI.ConstraintSet(), ci, MOI.Parameter(3.0))
```

```
julia> MOI.get(model, MOI.ConstraintSet(), ci)
MathOptInterface.Parameter{Float64}(3.0)
```

source

Vector sets

List of recognized vector sets.

MathOptInterface.Reals - Type.

Reals(dimension::Int)

The set $\mathbb{R}^{dimension}$ (containing all points) of non-negative dimension dimension.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Reals(3))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables, MathOptInterface.Reals}(1)

source

MathOptInterface.Zeros - Type.

Zeros(dimension::Int)

The set $\{0\}^{dimension}$ (containing only the origin) of non-negative dimension dimension.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

```
julia> x = MOI.add_variables(model, 3);
```

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Zeros(3))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables, MathOptInterface.Zeros}(1)

source

MathOptInterface.Nonnegatives - Type.

```
Nonnegatives(dimension::Int)
```

The nonnegative orthant $\{x \in \mathbb{R}^{dimension} : x \ge 0\}$ of non-negative dimension dimension.

Example

julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variables(model, 3);

source

MathOptInterface.Nonpositives - Type.

Nonpositives(dimension::Int)

The nonpositive orthant $\{x \in \mathbb{R}^{dimension} : x \leq 0\}$ of non-negative dimension dimension.

Example

julia> model = MOI.Utilities.Model{Float64}();

```
julia> x = MOI.add_variables(model, 3);
```

MathOptInterface.NormInfinityCone - Type.

```
NormInfinityCone(dimension::Int)
```

The ℓ_{∞} -norm cone $\{(t, x) \in \mathbb{R}^{dimension} : t \ge ||x||_{\infty} = \max_{i} |x_{i}|\}$ of dimension dimension.

The dimension must be at least 1.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
julia> t = MOI.add_variable(model)
MOI.VariableIndex(1)
julia> x = MOI.add_variables(model, 3);
```

source

MathOptInterface.NormOneCone - Type.

```
NormOneCone(dimension::Int)
```

The ℓ_1 -norm cone $\{(t,x) \in \mathbb{R}^{dimension} : t \ge \|x\|_1 = \sum_i |x_i|\}$ of dimension dimension.

The dimension must be at least 1.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
julia> t = MOI.add_variable(model)
MOI.VariableIndex(1)
```

```
julia> x = MOI.add_variables(model, 3);
```

source

MathOptInterface.NormCone - Type.

```
NormCone(p::Float64, dimension::Int)
```

The $\ell_p\text{-norm cone }\{(t,x)\in \mathbb{R}^{dimension}: t\geq \left(\sum\limits_i |x_i|^p\right)^{\frac{1}{p}}\}$ of dimension dimension.

The dimension must be at least 1.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
julia> t = MOI.add_variable(model)
MOI.VariableIndex(1)
```

julia> x = MOI.add_variables(model, 3);

source

MathOptInterface.SecondOrderCone - Type.

```
SecondOrderCone(dimension::Int)
```

The second-order cone (or Lorenz cone or ℓ_2 -norm cone) $\{(t, x) \in \mathbb{R}^{dimension} : t \ge ||x||_2\}$ of dimension dimension.

The dimension must be at least 1.

Example

julia> model = MOI.Utilities.Model{Float64}();

julia> t = MOI.add_variable(model)
MOI.VariableIndex(1)

julia> x = MOI.add_variables(model, 3);

source

MathOptInterface.RotatedSecondOrderCone - Type.

RotatedSecondOrderCone(dimension::Int)

The rotated second-order cone $\{(t, u, x) \in \mathbb{R}^{dimension} : 2tu \ge ||x||_2^2, t, u \ge 0\}$ of dimension dimension. The dimension must be at least 2.

Example

source

MathOptInterface.GeometricMeanCone - Type.

GeometricMeanCone(dimension::Int)

The geometric mean cone $\{(t,x) \in \mathbb{R}^{n+1} : x \ge 0, t \le \sqrt[n]{x_1x_2\cdots x_n}\}$, where dimension = n + 1 >= 2.

Duality note

The dual of the geometric mean cone is $\{(u,v) \in \mathbb{R}^{n+1} : u \leq 0, v \geq 0, -u \leq n \sqrt[n]{\prod_i v_i}\}$, where dimension = n + 1 >= 2.

Example

```
MOI.VectorOfVariables([t; x]),
MOI.GeometricMeanCone(4),
)
```

MathOptInterface.ExponentialCone - Type.

ExponentialCone()

The 3-dimensional exponential cone $\{(x, y, z) \in \mathbb{R}^3 : y \exp(x/y) \le z, y > 0\}.$

Example

source

MathOptInterface.DualExponentialCone - Type.

DualExponentialCone()

The 3-dimensional dual exponential cone $\{(u, v, w) \in \mathbb{R}^3 : -u \exp(v/u) \le \exp(1)w, u < 0\}$.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variables(model, 3);

source

MathOptInterface.PowerCone - Type.

PowerCone{T<:Real}(exponent::T)</pre>

The 3-dimensional power cone $\{(x, y, z) \in \mathbb{R}^3 : x^{exponent}y^{1-exponent} \ge |z|, x \ge 0, y \ge 0\}$ with parameter exponent.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
julia> x = MOI.add_variables(model, 3);
```

MathOptInterface.DualPowerCone - Type.

DualPowerCone{T<:Real}(exponent::T)</pre>

The 3-dimensional power cone $\{(u, v, w) \in \mathbb{R}^3 : (\frac{u}{exponent})^{exponent}(\frac{v}{1-exponent})^{1-exponent} \ge |w|, u \ge 0, v \ge 0\}$ with parameter exponent.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variables(model, 3);

source

MathOptInterface.RelativeEntropyCone - Type.

RelativeEntropyCone(dimension::Int)

The relative entropy cone $\{(u, v, w) \in \mathbb{R}^{1+2n} : u \geq \sum_{i=1}^n w_i \log(\frac{w_i}{v_i}), v_i \geq 0, w_i \geq 0\}$, where dimension = 2n + 1 >= 1.

Duality note

The dual of the relative entropy cone is $\{(u, v, w) \in \mathbb{R}^{1+2n} : \forall i, w_i \ge u(\log(\frac{u}{v_i}) - 1), v_i \ge 0, u > 0\}$ of dimension dimension = 2n + 1.

Example

julia> model = MOI.Utilities.Model{Float64}(); julia> u = MOI.add_variable(model); julia> v = MOI.add_variables(model, 3); julia> w = MOI.add_variables(model, 3);

```
julia> MOI.add_constraint(
            model,
            MOI.VectorOfVariables([u; v; w]),
            MOI.RelativeEntropyCone(7),
            )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
            ↔ MathOptInterface.RelativeEntropyCone}(1)
```

MathOptInterface.NormSpectralCone - Type.

NormSpectralCone(row_dim::Int, column_dim::Int)

The epigraph of the matrix spectral norm (maximum singular value function) $\{(t, X) \in \mathbb{R}^{1+row_d im \times column_d im} : t \ge \sigma_1(X)\}$, where σ_i is the *i*th singular value of the matrix X of non-negative row dimension row_dim and column dimension column dim.

The matrix X is vectorized by stacking the columns, matching the behavior of Julia's vec function.

Example

source

MathOptInterface.NormNuclearCone - Type.

NormNuclearCone(row_dim::Int, column_dim::Int)

The epigraph of the matrix nuclear norm (sum of singular values function) $\{(t, X) \in \mathbb{R}^{1+row_d im \times column_d im} : t \ge \sum_i \sigma_i(X)\}$, where σ_i is the *i*th singular value of the matrix X of non-negative row dimension row_dim and column dimension column_dim.

The matrix X is vectorized by stacking the columns, matching the behavior of Julia's vec function.

Example

source

MathOptInterface.SOS1 - Type.

```
SOS1{T<:Real}(weights::Vector{T})</pre>
```

The set corresponding to the Special Ordered Set (SOS) constraint of Type I.

Of the variables in the set, at most one can be nonzero.

The weights induce an ordering of the variables such that the *k*th element in the set corresponds to the *k*th weight in weights. Solvers may use these weights to improve the efficiency of the solution process, but the ordering does not change the set of feasible solutions.

Example

source

MathOptInterface.SOS2 - Type.

```
SOS2{T<:Real}(weights::Vector{T})</pre>
```

The set corresponding to the Special Ordered Set (SOS) constraint of Type II.

The weights induce an ordering of the variables such that the *k*th element in the set corresponds to the *k*th weight in weights. Therefore, the weights must be unique.

Of the variables in the set, at most two can be nonzero, and if two are nonzero, they must be adjacent in the ordering of the set.

Example

source

MathOptInterface.Indicator - Type.

```
Indicator{ACTIVATE_ON_ZERO}(set::AbstractScalarSet)
Indicator{ACTIVATE_ON_ONE}(set::AbstractScalarSet)
```

The set corresponding to an indicator constraint.

The type parameter must be an ActivationCondition.

When the type parameter is ACTIVATE_ON_ZERO, this means:

 $\{(y,x)\in\{0,1\}\times\mathbb{R}: y=0\implies x\in set\}$

When the type parameter is ACTIVATE_ON_ONE, this means:

 $\{(y,x) \in \{0,1\} \times \mathbb{R} : y = 1 \implies x \in set\}$

Notes

Most solvers expect that the first row of the function is interpretable as a VariableIndex (for example, 1.0 * x + 0.0), and that the variable is constrained to the ZeroOne set. An error will be thrown if this is not the case.

Example

The constraint $\{(y, x) \in \{0, 1\} \times \mathbb{R}^2 : y = 1 \implies x_1 + x_2 \le 9\}$ is defined as

```
julia> model = MOI.Utilities.Model{Float64}();
julia> x = MOI.add_variables(model, 2);
julia> y, _ = MOI.add_constrained_variable(model, MOI.ZeroOne());
julia> f = MOI.Utilities.vectorize([y, 1.0 * x[1] + 1.0 * x[2]])
[0.0 + 1.0 MOI.VariableIndex(3) ]
[0.0 + 1.0 MOI.VariableIndex(1) + 1.0 MOI.VariableIndex(2)]
julia> s = MOI.Indicator{MOI.ACTIVATE_ON_ONE}(MOI.LessThan(9.0))
MathOptInterface.Indicator{MOI.ACTIVATE_ON_ONE}(MOI.LessThan(9.0))
MathOptInterface.LessThan{Float64}(MathOptInterface.LessThan{Float64}(9.0))
julia> MOI.add_constraint(model, f, s)
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
→ MathOptInterface.Indicator{MathOptInterface.ACTIVATE_ON_ONE,
```

→ MathOptInterface.LessThan{Float64}}(1)

The constraint $\{(y, x) \in \{0, 1\} \times \mathbb{R} : y = 0 \implies x = 0\}$ is defined as

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variable(model);

julia> y, _ = MOI.add_constrained_variable(model, MOI.ZeroOne());

```
julia> f = MOI.VectorOfVariables([y, x]);
```

```
julia> s = MOI.Indicator{MOI.ACTIVATE_ON_ZERO}(MOI.EqualTo(0.0))
```

MathOptInterface.Indicator{MathOptInterface.ACTIVATE_ON_ZERO, ↔ MathOptInterface.EqualTo{Float64}(MathOptInterface.EqualTo{Float64}(0.0))

julia> MOI.add_constraint(model, f, s)

MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,

- $\hookrightarrow \quad \texttt{MathOptInterface.Indicator{MathOptInterface.ACTIVATE_ON_ZERO,}$
- → MathOptInterface.EqualTo{Float64}})(1)

source

MathOptInterface.ActivationCondition - Type.

ActivationCondition

Activation condition for an indicator constraint.

The enum value is used as first type parameter of Indicator{A,S}.

Values

ACTIVATE_ON_ZERO

The indicator constraint holds when the binary variable is zero.

ACTIVATE_ON_ONE

The indicator constraint holds when the binary variable is one.

source

MathOptInterface.ACTIVATE_ON_ZERO - Constant.

ACTIVATE_ON_ZER0::ActivationCondition

An instance of the ActivationCondition enum.

About

The indicator constraint holds when the binary variable is zero.

source

MathOptInterface.ACTIVATE_ON_ONE - Constant.

ACTIVATE_ON_ONE::ActivationCondition

An instance of the ActivationCondition enum.

About

The indicator constraint holds when the binary variable is one.

source

MathOptInterface.Complements - Type.

Complements(dimension::Base.Integer)

The set corresponding to a mixed complementarity constraint.

Complementarity constraints should be specified with an AbstractVectorFunction-in-Complements(dimension) constraint.

The dimension of the vector-valued function F must be dimension. This defines a complementarity constraint between the scalar function F[i] and the variable in F[i + dimension/2]. Thus, F[i + dimension/2] must be interpretable as a single variable x_i (for example, 1.0 * x + 0.0), and dimension must be even.

The mixed complementarity problem consists of finding x_i in the interval [lb, ub] (that is, in the set Interval(lb, ub)), such that the following holds:

- 1. F_i(x) == 0 if lb_i < x_i < ub_i
- 2. F_i(x) >= 0 if lb_i == x_i
- 3. F_i(x) <= 0 if x_i == ub_i

Classically, the bounding set for x_i is Interval(0, Inf), which recovers: $0 \le F_i(x) \perp x_i \ge 0$, where the \perp operator implies $F_i(x) * x_i = 0$.

Example

The problem:

x -in- Interval(-1, 1)
[-4 * x - 3, x] -in- Complements(2)

defines the mixed complementarity problem where the following holds:

1. -4 * x - 3 == 0 if -1 < x < 12. $-4 * x - 3 \ge 0$ if x == -13. -4 * x - 3 <= 0 if x == 1

There are three solutions:

x = -3/4 with F(x) = 0
 x = -1 with F(x) = 1
 x = 1 with F(x) = -7

julia> model = MOI.Utilities.Model{Float64}();

julia> x, _ = MOI.add_constrained_variable(model, MOI.Interval(-1.0, 1.0));

The function F can also be defined in terms of single variables. For example, the problem:

[x_3, x_4] -in- Nonnegatives(2)
[x_1, x_2, x_3, x_4] -in- Complements(4)

defines the complementarity problem where $0 \le x_1 \perp x_3 \ge 0$ and $0 \le x_2 \perp x_4 \ge 0$.

julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variables(model, 4);

julia> MOI.add_constraint(

CHAPTER 36. API REFERENCE

```
model,
MOI.VectorOfVariables(x),
MOI.Complements(4),
)
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
→ MathOptInterface.Complements}(1)
```

source

MathOptInterface.HyperRectangle - Type.

```
HyperRectangle(lower::Vector{T}, upper::Vector{T}) where {T}
```

The set $\{x \in \mathbb{R}^d : x_i \in [lower_i, upper_i] \forall i = 1, \dots, d\}.$

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

```
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,

→ MathOptInterface.HyperRectangle{Float64}}(1)
```

MOI.HyperRectangle(zeros(3), ones(3)),

source

)

MathOptInterface.Scaled - Type.

```
struct Scaled{S<:AbstractVectorSet} <: AbstractVectorSet
    set::S
end</pre>
```

Given a vector $a \in \mathbb{R}^d$ and a set representing the set $S \in \mathbb{R}^d$ such that Utilities.set_dot for $x \in S$ and $y \in S^*$ is

$$\sum_{i=1}^{d} a_i x_i y_i$$

the set Scaled(set) is defined as

$$\{(\sqrt{a_1}x_1, \sqrt{a_2}x_2, \dots, \sqrt{a_d}x_d) : x \in S\}$$

Example

This can be used to scale a vector of numbers

```
1.0
1.4142135623730951
1.0
```

It can be also used to scale a vector of function

```
julia> model = MOI.Utilities.Model{Float64}();
julia> x = MOI.add_variables(model, 3);
julia> func = MOI.VectorOfVariables(x)
[MOI.VariableIndex(1)]
MOI.VariableIndex(2)]
MOI.VariableIndex(3)]
julia> set = MOI.PositiveSemidefiniteConeTriangle(2)
MathOptInterface.PositiveSemidefiniteConeTriangle(2)
julia> MOI.Utilities.operate(*, Float64, Diagonal(a), func)
[0.0 + 1.0 MOI.VariableIndex(1)]
```

```
|0.0 + 1.0 MOI.VariableIndex(1)

|0.0 + 1.4142135623730951 MOI.VariableIndex(2)|

|0.0 + 1.0 MOI.VariableIndex(3)
```

source

Constraint programming sets

MathOptInterface.AllDifferent - Type.

AllDifferent(dimension::Int)

The set $\{x \in \mathbb{Z}^d\}$ such that no two elements in x take the same value and dimension = d.

Also known as

This constraint is called all_different in MiniZinc, and is sometimes also called distinct.

Example

```
To enforce x[1] != x[2] AND x[1] != x[3] AND x[2] != x[3]:
```

```
julia> model = MOI.Utilities.Model{Float64}();
```

```
julia> x = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(1)
MOI.VariableIndex(2)
MOI.VariableIndex(3)
```

source

MathOptInterface.BinPacking - Type.

```
BinPacking(c::T, w::Vector{T}) where {T}
```

The set $\{x \in \mathbb{Z}^d\}$ where d = length(w), such that each item i in 1:d of weight w[i] is put into bin x[i], and the total weight of each bin does not exceed c.

There are additional assumptions that the capacity, c, and the weights, w, must all be non-negative.

The bin numbers depend on the bounds of x, so they may be something other than the integers 1:d.

Also known as

This constraint is called bin packing in MiniZinc.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

```
julia> bins = MOI.add_variables(model, 5)
5-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(1)
MOI.VariableIndex(2)
MOI.VariableIndex(3)
MOI.VariableIndex(4)
MOI.VariableIndex(5)
```

julia> weights = Float64[1, 1, 2, 2, 3]

```
5-element Vector{Float64}:
1.0
1.0
2.0
 2.0
```

3.0

julia> MOI.add_constraint.(model, bins, MOI.Integer())

5-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,

```
→ MathOptInterface.Integer}}:
```

MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(1) MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(2) MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(3) MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(4) MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(5)

julia> MOI.add constraint.(model, bins, MOI.Interval(4.0, 6.0))

```
5-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
→ MathOptInterface.Interval{Float64}}:
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
→ MathOptInterface.Interval{Float64}}(1)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
→ MathOptInterface.Interval{Float64}}(2)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
\leftrightarrow MathOptInterface.Interval{Float64}}(3)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
→ MathOptInterface.Interval{Float64}}(4)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
\hookrightarrow MathOptInterface.Interval{Float64}}(5)
julia> MOI.add constraint(model, MOI.VectorOfVariables(bins), MOI.BinPacking(3.0, weights))
```

```
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
```

```
→ MathOptInterface.BinPacking{Float64}}(1)
```

source

MathOptInterface.Circuit - Type.

Circuit(dimension::Int)

The set $\{x \in \{1..d\}^d\}$ that constraints x to be a circuit, such that $x_i = j$ means that j is the successor of i, and dimension = d.

Graphs with multiple independent circuits, such as [2, 1, 3] and [2, 1, 4, 3], are not valid.

Also known as

This constraint is called circuit in MiniZinc, and it is equivalent to forming a (potentially sub-optimal) tour in the travelling salesperson problem.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

```
julia> x = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
```

3-element Vector{MathOptInterface.VariableIndex}: MOI.VariableIndex(1) MOI.VariableIndex(2) MOI.VariableIndex(3)

source

MathOptInterface.CountAtLeast - Type.

CountAtLeast(n::Int, d::Vector{Int}, set::Set{Int})

The set $\{x \in \mathbb{Z}^{d_1+d_2+\ldots d_N}\}$, where x is partitioned into N subsets ($\{x_1, \ldots, x_{d_1}\}$, $\{x_{d_1+1}, \ldots, x_{d_1+d_2}\}$ and so on), and at least n elements of each subset take one of the values in set.

Also known as

This constraint is called at_least in MiniZinc.

Example

To ensure that 3 appears at least once in each of the subsets $\{a, b\}$ and $\{b, c\}$:

julia> model = MOI.Utilities.Model{Float64}();

```
julia> x, d, set = [a, b, b, c], [2, 2], [3]
```

(MathOptInterface.VariableIndex[MOI.VariableIndex(1), MOI.VariableIndex(2), ↔ MOI.VariableIndex(2), MOI.VariableIndex(3)], [2, 2], [3])

source

MathOptInterface.CountBelongs - Type.

```
CountBelongs(dimenson::Int, set::Set{Int})
```

The set $\{(n, x) \in \mathbb{Z}^{1+d}\}$, such that n elements of the vector x take on of the values in set and dimension = 1 + d.

Also known as

This constraint is called among by MiniZinc.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
julia> n, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(1), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
→ MathOptInterface.Integer}(1))
julia> x = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(2)
 MOI.VariableIndex(3)
 MOI.VariableIndex(4)
julia> set = Set([3, 4, 5])
Set{Int64} with 3 elements:
 5
 4
  3
julia> MOI.add_constraint(model, MOI.VectorOfVariables([n; x]), MOI.CountBelongs(4, set))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
→ MathOptInterface.CountBelongs}(1)
```

source

MathOptInterface.CountDistinct - Type.

CountDistinct(dimension::Int)

The set $\{(n,x)\in\mathbb{Z}^{1+d}\}$, such that the number of distinct values in x is n and dimension = 1 + d.

Also known as

This constraint is called nvalues in MiniZinc.

Example

To model:

- if n == 1, then x[1] == x[2] == x[3]
- if n == 2, then
 - x[1] == x[2] != x[3] or

```
- x[1] != x[2] == x[3] or
```

```
- x[1] == x[3] != x[2]
```

• if n == 3, then x[1] != x[2], x[2] != x[3] and x[3] != x[1]

```
julia> model = MOI.Utilities.Model{Float64}();
```

```
julia> x = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
```

3-element Vector{MathOptInterface.VariableIndex}: MOI.VariableIndex(2) MOI.VariableIndex(3) MOI.VariableIndex(4)

Relationship to AllDifferent

When the first element is d, CountDistinct is equivalent to an AllDifferent constraint.

source

MathOptInterface.CountGreaterThan - Type.

CountGreaterThan(dimension::Int)

The set $\{(c, y, x) \in \mathbb{Z}^{1+1+d}\}$, such that c is strictly greater than the number of occurences of y in x and dimension = 1 + 1 + d.

Also known as

This constraint is called count gt in MiniZinc.

Example

julia> model = MOI.Utilities.Model{Float64}();

```
(MOI.VariableIndex(2), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,

→ MathOptInterface.Integer}(2))
```

```
julia> x = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(3)
MOI.VariableIndex(4)
```

CHAPTER 36. API REFERENCE

MOI.VariableIndex(5)

source

MathOptInterface.Cumulative - Type.

Cumulative(dimension::Int)

The set $\{(s, d, r, b) \in \mathbb{Z}^{3n+1}\}$, representing the cumulative global constraint, where n == length(s) == length(r) == length(b) and dimension = 3n + 1.

Cumulative requires that a set of tasks given by start times s, durations d, and resource requirements r, never requires more than the global resource bound b at any one time.

Also known as

This constraint is called cumulative in MiniZinc.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

```
julia> s = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(1)
MOI.VariableIndex(2)
```

MOI.VariableIndex(3)

```
julia> d = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
```

3-element Vector{MathOptInterface.VariableIndex}: MOI.VariableIndex(4) MOI.VariableIndex(5)

```
MOI.VariableIndex(6)
```

julia> r = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]

3-element Vector{MathOptInterface.VariableIndex}: MOI.VariableIndex(7) MOI.VariableIndex(8)

MOI.VariableIndex(9)

julia> b, _ = MOI.add_constrained_variable(model, MOI.Integer())

(MOI.VariableIndex(10), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, → MathOptInterface.Integer}(10))

MathOptInterface.Path - Type.

```
Path(from::Vector{Int}, to::Vector{Int})
```

Given a graph comprised of a set of nodes 1..N and a set of arcs 1..E represented by an edge from node from[i] to node to[i], Path constrains the set $(s, t, ns, es) \in (1..N) \times (1..E) \times \{0, 1\}^N \times \{0, 1\}^E$, to form subgraph that is a path from node s to node t, where node n is in the path if ns[n] is 1, and edge e is in the path if es[e] is 1.

The path must be acyclic, and it must traverse all nodes n for which ns[n] is 1, and all edges e for which es[e] is 1.

Also known as

This constraint is called path in MiniZinc.

Example

julia> model = MOI.Utilities.Model{Float64}();

```
julia> N, E = 4, 5
(4, 5)
julia> from = [1, 1, 2, 2, 3]
5-element Vector{Int64}:
    1
    2
    2
    3
julia> to = [2, 3, 3, 4, 4]
5-element Vector{Int64}:
    2
```

julia> s, _ = MOI.add_constrained_variable(model, MOI.Integer())

(MOI.VariableIndex(1), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, → MathOptInterface.Integer}(1))

julia> t, _ = MOI.add_constrained_variable(model, MOI.Integer())

(MOI.VariableIndex(2), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, → MathOptInterface.Integer}(2))

julia> ns = MOI.add_variables(model, N)

4-element Vector{MathOptInterface.VariableIndex}:

- MOI.VariableIndex(3)
- MOI.VariableIndex(4)
- MOI.VariableIndex(5)
- MOI.VariableIndex(6)

julia> MOI.add_constraint.(model, ns, MOI.ZeroOne())

4-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,

```
\hookrightarrow MathOptInterface.ZeroOne}}:
```

MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(3)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(4)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(5)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(6)

julia> es = MOI.add_variables(model, E)

```
5-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(7)
MOI.VariableIndex(8)
MOI.VariableIndex(9)
MOI.VariableIndex(10)
MOI.VariableIndex(11)
```

julia> MOI.add_constraint.(model, es, MOI.ZeroOne())

5-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,

```
→ MathOptInterface.ZeroOne}}:
```

```
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(7)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(8)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(9)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(10)
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(11)
```

julia> MOI.add_constraint(model, MOI.VectorOfVariables([s; t; ns; es]), MOI.Path(from, to))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables, MathOptInterface.Path}(1)

source

MathOptInterface.Reified - Type.

Reified(set::AbstractSet)

The constraint $[z; f(x)] \in Reifield(S)$ ensures that $f(x) \in S$ if and only if z == 1, where $z \in \{0, 1\}$.

julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}());

```
julia> x = MOI.add_variable(model)
MOI.VariableIndex(2)
```

```
julia> MOI.add_constraint(
    model,
    MOI.Utilities.vectorize([z, 2.0 * x]),
    MOI.Reified(MOI.GreaterThan(1.0)),
)
```

MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64}, → MathOptInterface.Reified{MathOptInterface.GreaterThan{Float64}}}(1) MathOptInterface.Table - Type.

```
Table(table::Matrix{T}) where {T}
```

The set $\{x \in \mathbb{R}^d\}$ where d = size(table, 2), such that x belongs to one row of table. That is, there exists some j in 1:size(table, 1), such that x[i] = table[j, i] for all i=1:size(table, 2).

Also known as

This constraint is called table in MiniZinc.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

```
julia> x = MOI.add_variables(model, 3)
3-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(1)
MOI.VariableIndex(2)
MOI.VariableIndex(3)
julia> table = Float64[1 1 0; 0 1 1; 1 0 1; 1 1 1]
4×3 Matrix{Float64}:
1.0 1.0 0.0
0.0 1.0 1.0
1.0 0.0 1.0
1.0 1.0
julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Table(table))
```

MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables, ↔ MathOptInterface.Table{Float64}}(1)

source

Matrix sets

Matrix sets are vectorized to be subtypes of AbstractVectorSet.

For sets of symmetric matrices, storing both the (i, j) and (j, i) elements is redundant. Use the AbstractSymmetricMatrixSe set to represent only the vectorization of the upper triangular part of the matrix.

When the matrix of expressions constrained to be in the set is not symmetric, and hence additional constraints are needed to force the equality of the (i, j) and (j, i) elements, use the AbstractSymmetricMatrixSetSquare set.

The Bridges.Constraint.SquareBridge can transform a set from the square form to the triangular_form by adding appropriate constraints if the (i, j) and (j, i) expressions are different.

MathOptInterface.AbstractSymmetricMatrixSetTriangle - Type.

abstract type AbstractSymmetricMatrixSetTriangle <: AbstractVectorSet end</pre>

Abstract supertype for subsets of the (vectorized) cone of symmetric matrices, with side_dimension rows and columns. The entries of the upper-right triangular part of the matrix are given column by column

(or equivalently, the entries of the lower-left triangular part are given row by row). A vectorized cone of dimension n corresponds to a square matrix with side dimension $\sqrt{1/4 + 2n} - 1/2$. (Because a $d \times d$ matrix has d(d+1)/2 elements in the upper or lower triangle.)

Example

The matrix

| [1 | 2 | 4 |
|------------|---|---|
| 2 | 3 | 5 |
| 4 | 5 | 6 |

has side_dimension 3 and vectorization (1, 2, 3, 4, 5, 6).

Note

Two packed storage formats exist for symmetric matrices, the respective orders of the entries are:

- upper triangular column by column (or lower triangular row by row);
- lower triangular column by column (or upper triangular row by row).

The advantage of the first format is the mapping between the (i, j) matrix indices and the k index of the vectorized form. It is simpler and does not depend on the side dimension of the matrix. Indeed,

- the entry of matrix indices (i, j) has vectorized index k = div((j 1) * j, 2) + i if $i \le j$ and k = div((i 1) * i, 2) + j if $j \le i$;
- and the entry with vectorized index k has matrix indices i = div(1 + isqrt(8k 7), 2) and j = k div((i 1) * i, 2) or j = div(1 + isqrt(8k 7), 2) and i = k div((j 1) * j, 2).

Duality note

The scalar product for the symmetric matrix in its vectorized form is the sum of the pairwise product of the diagonal entries plus twice the sum of the pairwise product of the upper diagonal entries; see [p. 634, 1]. This has important consequence for duality.

Consider for example the following problem (PositiveSemidefiniteConeTriangle is a subtype of AbstractSymmetricMatri:

| $\max_{x \in \mathbb{R}}$ | x |
|---------------------------|---|
| s.t. | $(1,-x,1)\in$ PositiveSemidefiniteConeTriangle $(2).$ |

The dual is the following problem

| $\min_{x \in \mathbb{R}^3}$ | $y_1 + y_3$ |
|-----------------------------|--|
| s.t. | $2y_2 = 1$ |
| | $y\in PositiveSemidefiniteConeTriangle(2)$ |

Why do we use $2y_2$ in the dual constraint instead of y_2 ? The reason is that $2y_2$ is the scalar product between y and the symmetric matrix whose vectorized form is (0, 1, 0). Indeed, with our modified scalar products we have

$$\langle (0,1,0), (y_1,y_2,y_3) \rangle = \operatorname{trace} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} y_1 & y_2 \\ y_2 & y_3 \end{pmatrix} = 2y_2.$$

References

[1] Boyd, S. and Vandenberghe, L.. Convex optimization. Cambridge university press, 2004.

source

MathOptInterface.AbstractSymmetricMatrixSetSquare - Type.

abstract type AbstractSymmetricMatrixSetSquare <: AbstractVectorSet end</pre>

Abstract supertype for subsets of the (vectorized) cone of symmetric matrices, with side_dimension rows and columns. The entries of the matrix are given column by column (or equivalently, row by row). The matrix is both constrained to be symmetric and to have its triangular_form belong to the corresponding set. That is, if the functions in entries (i, j) and (j, i) are different, then a constraint will be added to make sure that the entries are equal.

Example

PositiveSemidefiniteConeSquare is a subtype of AbstractSymmetricMatrixSetSquare and constraining the matrix

$$\begin{bmatrix} 1 & -y \\ -z & 0 \end{bmatrix}$$

to be symmetric positive semidefinite can be achieved by constraining the vector (1, -z, -y, 0) (or (1, -y, -z, 0)) to belong to the PositiveSemidefiniteConeSquare(2). It both constrains y = z and (1, -y, 0) (or (1, -z, 0)) to be in PositiveSemidefiniteConeTriangle(2), since triangular_form(PositiveSemidefiniteConeSquare) is PositiveSemidefiniteConeTriangle.

source

MathOptInterface.side_dimension - Function.

```
side_dimension(
    set::Union{
        AbstractSymmetricMatrixSetTriangle,
        AbstractSymmetricMatrixSetSquare,
        HermitianPositiveSemidefiniteConeTriangle,
    },
)
```

Side dimension of the matrices in set.

Convention

By convention, the side dimension should be stored in the side_dimension field. If this is not the case for a subtype of AbstractSymmetricMatrixSetTriangle, or AbstractSymmetricMatrixSetSquare you must implement this method.

source

MathOptInterface.triangular_form - Function.

```
triangular_form(S::Type{<:AbstractSymmetricMatrixSetSquare})
triangular_form(set::AbstractSymmetricMatrixSetSquare)</pre>
```

Return the AbstractSymmetricMatrixSetTriangle corresponding to the vectorization of the upper triangular part of matrices in the AbstractSymmetricMatrixSetSquare set.

source

List of recognized matrix sets.

MathOptInterface.PositiveSemidefiniteConeTriangle - Type.

PositiveSemidefiniteConeTriangle(side_dimension::Int) <: AbstractSymmetricMatrixSetTriangle</pre>

The (vectorized) cone of symmetric positive semidefinite matrices, with non-negative side_dimension rows and columns.

See AbstractSymmetricMatrixSetTriangle for more details on the vectorized form.

source

MathOptInterface.PositiveSemidefiniteConeSquare - Type.

PositiveSemidefiniteConeSquare(side_dimension::Int) <: AbstractSymmetricMatrixSetSquare</pre>

The cone of symmetric positive semidefinite matrices, with non-negative side length side_dimension.

See AbstractSymmetricMatrixSetSquare for more details on the vectorized form.

The entries of the matrix are given column by column (or equivalently, row by row).

The matrix is both constrained to be symmetric and to be positive semidefinite. That is, if the functions in entries (i, j) and (j, i) are different, then a constraint will be added to make sure that the entries are equal.

Example

Constraining the matrix

 $\begin{bmatrix} 1 & -y \\ -z & 0 \end{bmatrix}$

to be symmetric positive semidefinite can be achieved by constraining the vector (1, -z, -y, 0) (or (1, -y, -z, 0)) to belong to the PositiveSemidefiniteConeSquare(2).

It both constrains y = z and (1, -y, 0) (or (1, -z, 0)) to be in PositiveSemidefiniteConeTriangle(2).

source

MathOptInterface.HermitianPositiveSemidefiniteConeTriangle - Type.

HermitianPositiveSemidefiniteConeTriangle(side_dimension::Int) <: AbstractVectorSet</pre>

The (vectorized) cone of Hermitian positive semidefinite matrices, with non-negative side_dimension rows and columns.

Becaue the matrix is Hermitian, the diagonal elements are real, and the complex-valued lower triangular entries are obtained as the conjugate of corresponding upper triangular entries.

Vectorization format

The vectorized form starts with real part of the entries of the upper triangular part of the matrix, given column by column as explained in AbstractSymmetricMatrixSetSquare.

It is then followed by the imaginary part of the off-diagonal entries of the upper triangular part, also given column by column.

For example, the matrix

$$\begin{bmatrix} 1 & 2+7im & 4+8im \\ 2-7im & 3 & 5+9im \\ 4-8im & 5-9im & 6 \end{bmatrix}$$

has side_dimension 3 and is represented as the vector [1, 2, 3, 4, 5, 6, 7, 8, 9].

source

MathOptInterface.LogDetConeTriangle - Type.

LogDetConeTriangle(side_dimension::Int)

The log-determinant cone $\{(t, u, X) \in \mathbb{R}^{2+d(d+1)/2} : t \leq u \log(\det(X/u)), u > 0\}$, where the matrix X is represented in the same symmetric packed format as in the PositiveSemidefiniteConeTriangle.

The non-negative argument side_dimension is the side dimension of the matrix X, that is, its number of rows or columns.

Example

MathOptInterface.LogDetConeSquare - Type.

LogDetConeSquare(side_dimension::Int)

The log-determinant cone $\{(t, u, X) \in \mathbb{R}^{2+d^2} : t \leq u \log(\det(X/u)), X \text{ symmetric}, u > 0\}$, where the matrix X is represented in the same format as in the PositiveSemidefiniteConeSquare.

Similarly to PositiveSemidefiniteConeSquare, constraints are added to ensure that X is symmetric.

The non-negative argument side_dimension is the side dimension of the matrix X, that is, its number of rows or columns.

Example

source

MathOptInterface.RootDetConeTriangle - Type.

RootDetConeTriangle(side_dimension::Int)

The root-determinant cone $\{(t, X) \in \mathbb{R}^{1+d(d+1)/2} : t \leq \det(X)^{1/d}\}$, where the matrix X is represented in the same symmetric packed format as in the PositiveSemidefiniteConeTriangle.

The non-negative argument side_dimension is the side dimension of the matrix X, that is, its number of rows or columns.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> t = MOI.add_variable(model)
MOI.VariableIndex(1)

julia> X = MOI.add_variables(model, 3);

MathOptInterface.RootDetConeSquare - Type.

RootDetConeSquare(side_dimension::Int)

The root-determinant cone $\{(t, X) \in \mathbb{R}^{1+d^2} : t \leq \det(X)^{1/d}, X \text{ symmetric}\}$, where the matrix X is represented in the same format as PositiveSemidefiniteConeSquare.

Similarly to PositiveSemidefiniteConeSquare, constraints are added to ensure that X is symmetric.

The non-negative argument side_dimension is the side dimension of the matrix X, that is, its number of rows or columns.

Example

source

36.2 Models

Attribute interface

MathOptInterface.is_set_by_optimize - Function.

is_set_by_optimize(::AnyAttribute)

Return a Bool indicating whether the value of the attribute is set during an optimize! call, that is, the attribute is used to query the result of the optimization.

If an attribute can be set by the user, define is_copyable instead.

An attribute cannot be both is_copyable and is_set_by_optimize.

Default fallback

This function returns false by default so it should be implemented for attributes that are set by optimize!.

Undefined behavior

Querying the value of the attribute that is_set_by_optimize before a call to optimize! is undefined and depends on solver-specific behavior.

Example

julia> MOI.is_set_by_optimize(MOI.ObjectiveValue())
true

julia> MOI.is_set_by_optimize(MOI.VariableName())
false

source

MathOptInterface.is_copyable - Function.

is_copyable(::AnyAttribute)

Return a Bool indicating whether the value of the attribute may be copied during copy_to using set.

If an attribute is_copyable, then it cannot be modified by the optimizer, and get must always return the value that was set by the user.

If an attribute is the result of an optimization, define is_set_by_optimize instead.

An attribute cannot be both is_set_by_optimize and is_copyable.

Default fallback

By default is_copyable(attr) returns !is_set_by_optimize(attr), which is most probably true.

If an attribute should not be copied, define is_copyable(::MyAttribute) = false.

source

MathOptInterface.get - Function.

MOI.get(b::AbstractBridge, ::MOI.NumberOfVariables)::Int64

Return the number of variables created by the bridge b in the model.

See also MOI.NumberOfConstraints.

Implementation notes
• There is a default fallback, so you need only implement this if the bridge adds new variables.

source

```
MOI.get(b::AbstractBridge, ::MOI.ListOfVariableIndices)
```

Return the list of variables created by the bridge b.

See also MOI.ListOfVariableIndices.

Implementation notes

• There is a default fallback, so you need only implement this if the bridge adds new variables.

source

```
MOI.get(b::AbstractBridge, ::MOI.NumberOfConstraints{F,S})::Int64 where {F,S}
```

Return the number of constraints of the type F-in-S created by the bridge b.

See also MOI.NumberOfConstraints.

Implementation notes

 There is a default fallback, so you need only implement this for the constraint types returned by added_constraint_types.

source

MOI.get(b::AbstractBridge, ::MOI.ListOfConstraintIndices{F,S}) where {F,S}

Return a Vector{ConstraintIndex{F,S}} with indices of all constraints of type F-in-S created by the bride b.

See also MOI.ListOfConstraintIndices.

Implementation notes

 There is a default fallback, so you need only implement this for the constraint types returned by added_constraint_types.

source

```
function MOI.get(
    model::MOI.ModelLike,
    attr::MOI.AbstractConstraintAttribute,
    bridge::AbstractBridge,
)
```

Return the value of the attribute attr of the model model for the constraint bridged by bridge.

source

get(optimizer::AbstractOptimizer, attr::AbstractOptimizerAttribute)

Return an attribute attr of the optimizer optimizer.

get(model::ModelLike, attr::AbstractModelAttribute)

Return an attribute attr of the model model.

get(model::ModelLike, attr::AbstractVariableAttribute, v::VariableIndex)

If the attribute attr is set for the variable v in the model model, return its value, return nothing otherwise. If the attribute attr is not supported by model then an error should be thrown instead of returning nothing.

get(model::ModelLike, attr::AbstractVariableAttribute, v::Vector{VariableIndex})

Return a vector of attributes corresponding to each variable in the collection v in the model model.

get(model::ModelLike, attr::AbstractConstraintAttribute, c::ConstraintIndex)

If the attribute attr is set for the constraint c in the model model, return its value, return nothing otherwise. If the attribute attr is not supported by model then an error should be thrown instead of returning nothing.

```
get(
    model::ModelLike,
    attr::AbstractConstraintAttribute,
    c::Vector{ConstraintIndex{F,S}},
) where {F,S}
```

Return a vector of attributes corresponding to each constraint in the collection c in the model model.

```
get(model::ModelLike, ::Type{VariableIndex}, name::String)
```

If a variable with name name exists in the model model, return the corresponding index, otherwise return nothing. Errors if two variables have the same name.

```
get(
    model::ModelLike,
    ::Type{ConstraintIndex{F,S}},
    name::String,
) where {F,S}
```

If an F-in-S constraint with name name exists in the model model, return the corresponding index, otherwise return nothing. Errors if two constraints have the same name.

CHAPTER 36. API REFERENCE

```
get(model::ModelLike, ::Type{ConstraintIndex}, name::String)
```

If *any* constraint with name name exists in the model model, return the corresponding index, otherwise return nothing. This version is available for convenience but may incur a performance penalty because it is not type stable. Errors if two constraints have the same name.

source

get(model::GenericModel, attr::MathOptInterface.AbstractOptimizerAttribute)

Return the value of the attribute attr from the model's MOI backend.

source

get(model::GenericModel, attr::MathOptInterface.AbstractModelAttribute)

Return the value of the attribute attr from the model's MOI backend.

source

MathOptInterface.get! - Function.

```
get!(output, model::ModelLike, args...)
```

An in-place version of get.

The signature matches that of get except that the result is placed in the vector output.

source

MathOptInterface.set - Function.

```
function MOI.set(
    model::MOI.ModelLike,
    attr::MOI.AbstractConstraintAttribute,
    bridge::AbstractBridge,
    value,
)
```

Set the value of the attribute attr of the model model for the constraint bridged by bridge.

source

set(optimizer::AbstractOptimizer, attr::AbstractOptimizerAttribute, value)

Assign value to the attribute attr of the optimizer optimizer.

set(model::ModelLike, attr::AbstractModelAttribute, value)

Assign value to the attribute attr of the model model.

```
set(model::ModelLike, attr::AbstractVariableAttribute, v::VariableIndex, value)
```

Assign value to the attribute attr of variable v in model model.

```
set(
    model::ModelLike,
    attr::AbstractVariableAttribute,
    v::Vector{VariableIndex},
    vector_of_values,
)
```

Assign a value respectively to the attribute attr of each variable in the collection v in model model.

```
set(
    model::ModelLike,
    attr::AbstractConstraintAttribute,
    c::ConstraintIndex,
    value,
)
```

Assign a value to the attribute attr of constraint c in model model.

```
set(
    model::ModelLike,
    attr::AbstractConstraintAttribute,
    c::Vector{ConstraintIndex{F,S}},
    vector_of_values,
) where {F,S}
```

Assign a value respectively to the attribute attr of each constraint in the collection c in model model.

An UnsupportedAttribute error is thrown if model does not support the attribute attr (see supports) and a SetAttributeNotAllowed error is thrown if it supports the attribute attr but it cannot be set.

```
set(
    model::ModelLike,
    ::ConstraintSet,
    c::ConstraintIndex{F,S},
    set::S,
) where {F,S}
```

Change the set of constraint c to the new set set which should be of the same type as the original set.

```
set(
    model::ModelLike,
    ::ConstraintFunction,
    c::ConstraintIndex{F,S},
    func::F,
) where {F,S}
```

Replace the function in constraint c with func. F must match the original function type used to define the constraint.

Note Setting the constraint function is not allowed if F is VariableIndex; a SettingVariableIndexNotAllowed error is thrown instead. This is because, it would require changing the index c since the index of VariableIndex constraints must be the same as the index of the variable.

source

```
MOI.set(
    model::Optimizer,
    attr::ConstraintAttribute,
    ci::MOI.ConstraintIndex{MOI.ScalarAffineFunction{Float64}, <:Any},
    value
)</pre>
```

Set a constraint attribute.

source

```
MOI.set(
    model::Optimizer,
    attr::VariableAttribute,
    vi::MOI.VariableIndex,
    value
)
```

Set a variable attribute.

source

MOI.set(model::Optimizer, attr::ModelAttribute, value)

Set a model attribute.

source

MathOptInterface.supports - Function.

```
MOI.supports(
    model::MOI.ModelLike,
    attr::MOI.AbstractConstraintAttribute,
    BT::Type{<:AbstractBridge},
)</pre>
```

Return a Bool indicating whether BT supports setting attr to model.

source

supports(model::ModelLike, sub::AbstractSubmittable)::Bool

Return a Bool indicating whether model supports the submittable sub.

supports(model::ModelLike, attr::AbstractOptimizerAttribute)::Bool

Return a Bool indicating whether model supports the optimizer attribute attr. That is, it returns false if copy_to(model, src) shows a warning in case attr is in the ListOfOptimizerAttributesSet of src; see copy_to for more details on how unsupported optimizer attributes are handled in copy.

supports(model::ModelLike, attr::AbstractModelAttribute)::Bool

Return a Bool indicating whether model supports the model attribute attr. That is, it returns false if copy_to(model, src) cannot be performed in case attr is in the ListOfModelAttributesSet of src.

```
supports(
    model::ModelLike,
    attr::AbstractVariableAttribute,
    ::Type{VariableIndex},
)::Bool
```

Return a Bool indicating whether model supports the variable attribute attr. That is, it returns false if copy_to(model, src) cannot be performed in case attr is in the ListOfVariableAttributesSet of src.

```
supports(
    model::ModelLike,
    attr::AbstractConstraintAttribute,
    ::Type{ConstraintIndex{F,S}},
)::Bool where {F,S}
```

Return a Bool indicating whether model supports the constraint attribute attr applied to an F-in-S constraint. That is, it returns false if copy_to(model, src) cannot be performed in case attr is in the ListOfConstraintAttributesSet of src.

For all five methods, if the attribute is only not supported in specific circumstances, it should still return true.

Note that supports is only defined for attributes for which is_copyable returns true as other attributes do not appear in the list of attributes set obtained by ListOfXXXAttributesSet.

source

MathOptInterface.attribute_value_type - Function.

attribute_value_type(attr::AnyAttribute)

Given an attribute attr, return the type of value expected by get, or returned by set.

Notes

• Only implement this if it make sense to do so. If un-implemented, the default is Any.

source

Model interface

MathOptInterface.ModelLike - Type.

ModelLike

Abstract supertype for objects that implement the "Model" interface for defining an optimization problem.

source

MathOptInterface.is_empty - Function.

is_empty(model::ModelLike)

Returns false if the model has any model attribute set or has any variables or constraints.

Note that an empty model can have optimizer attributes set.

source

MathOptInterface.empty! - Function.

empty!(model::ModelLike)

Empty the model, that is, remove all variables, constraints and model attributes but not optimizer attributes.

source

MathOptInterface.write_to_file - Function.

write_to_file(model::ModelLike, filename::String)

Write the current model to the file at filename.

Supported file types depend on the model type.

source

MathOptInterface.read_from_file - Function.

read from_file(model::ModelLike, filename::String)

Read the file filename into the model model. If model is non-empty, this may throw an error.

Supported file types depend on the model type.

Note

Once the contents of the file are loaded into the model, users can query the variables via get(model, ListOfVariableIndices()). However, some filetypes, such as LP files, do not maintain an explicit ordering of the variables. Therefore, the returned list may be in an arbitrary order.

To avoid depending on the order of the indices, look up each variable index by name using get(model, VariableIndex, "name").

source

MathOptInterface.supports_incremental_interface - Function.

supports_incremental_interface(model::ModelLike)

Return a Bool indicating whether model supports building incrementally via add variable and add constraint.

The main purpose of this function is to determine whether a model can be loaded into model incrementally or whether it should be cached and copied at once instead.

source

MathOptInterface.copy_to - Function.

copy to(dest::ModelLike, src::ModelLike)::IndexMap

Copy the model from src into dest.

The target dest is emptied, and all previous indices to variables and constraints in dest are invalidated.

Returns an IndexMap object that translates variable and constraint indices from the src model to the corresponding indices in the dest model.

Notes

- If a constraint that in src is not supported by dest, then an UnsupportedConstraint error is thrown.
- If an AbstractModelAttribute, AbstractVariableAttribute, or AbstractConstraintAttribute is set in src but not supported by dest, then an UnsupportedAttribute error is thrown.

AbstractOptimizerAttributes are not copied to the dest model.

IndexMap

Implementations of copy_to must return an IndexMap. For technical reasons, this type is defined in the Utilities submodule as MOI.Utilities.IndexMap. However, since it is an integral part of the MOI API, we provide MOI.IndexMap as an alias.

Example

Given empty `ModelLike` objects `src` and `dest`.

```
x = add_variable(src)
```

is_valid(src, x) # true is_valid(dest, x) # false (`dest` has no variables)

```
index_map = copy_to(dest, src)
is_valid(dest, x) # false (unless index_map[x] == x)
is_valid(dest, index_map[x]) # true
```

source

MathOptInterface.IndexMap - Type.

IndexMap()

The dictionary-like object returned by copy_to.

IndexMap

Implementations of copy_to must return an IndexMap. For technical reasons, the IndexMap type is defined in the Utilities submodule as MOI.Utilities.IndexMap. However, since it is an integral part of the MOI API, we provide this MOI.IndexMap as an alias.

source

Model attributes

MathOptInterface.AbstractModelAttribute - Type.

AbstractModelAttribute

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of the model.

source

MathOptInterface.Name - Type.

Name()

An AbstractModelAttribute for the string identifying the model.

It has a default value of "" if not set.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.Name)::String
MOI.set(::Optimizer, ::MOI.Name, ::String)::Nothing
MOI.supports(::Optimizer, ::MOI.Name)::Bool
```

source

MathOptInterface.ObjectiveFunction - Type.

```
ObjectiveFunction{F<:AbstractScalarFunction}()</pre>
```

An AbstractModelAttribute for the objective function which has a type F<:AbstractScalarFunction.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ObjectiveFunction{F},
)::F where {F<:MOI.AbstractFunction}
MOI.set(
    ::Optimizer,
    ::MOI.ObjectiveFunction{F},
    ::F,
)::F where {F<:MOI.AbstractFunction}
MOI.supports(::Optimizer, ::MOI.ObjectiveFunction{<:MOI.AbstractFunction})::Bool</pre>
```

When implementing get, F may to be equivalent but not necessarily identical to the function type set by the user. If the objective function cannot be converted to F, an InexactError must be thrown.

source

MathOptInterface.ObjectiveFunctionType - Type.

ObjectiveFunctionType()

An AbstractModelAttribute for the type F of the objective function set using the ObjectiveFunction{F} attribute.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::0ptimizer,
    ::MOI.ObjectiveFunctionType,
)::Type{<:MOI.AbstractFunction}</pre>
```

They should not implement set or supports.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

julia> MOI.set(model, MOI.ObjectiveFunction{MOI.VariableIndex}(), x)

```
julia> MOI.get(model, MOI.ObjectiveFunctionType())
MathOptInterface.VariableIndex
```

source

MathOptInterface.ObjectiveSense - Type.

ObjectiveSense()

An AbstractModelAttribute for the objective sense of the objective function, which must be an OptimizationSense.

The default is FEASIBILITY_SENSE.

Interaction with ObjectiveFunction

Setting the sense to FEASIBILITY_SENSE unsets the ObjectiveFunction attribute. That is, if you first set ObjectiveFunction and then set ObjectiveSense to be FEASIBILITY_SENSE, no objective function will be passed to the solver.

In addition, some reformulations of ObjectiveFunction via bridges rely on the value of ObjectiveSense. Therefore, you should set ObjectiveSense before setting ObjectiveFunction.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.ObjectiveSense)::MOI.OptimizationSense
MOI.set(::Optimizer, ::MOI.ObjectiveSense, ::MOI.OptimizationSense)::Nothing
MOI.supports(::Optimizer, ::MOI.ObjectiveSense)::Bool
```

source

MathOptInterface.OptimizationSense - Type.

OptimizationSense

An Enum for the value of the ObjectiveSense attribute.

Values

MIN_SENSE

The goal is to minimize the objective function.

MAX SENSE

The goal is to maximize the objective function.

FEASIBILITY_SENSE

The model does not have an objective function.

source

MathOptInterface.MIN_SENSE - Constant.

MIN_SENSE::0ptimizationSense

An instance of the OptimizationSense enum.

About

The goal is to minimize the objective function.

source

MathOptInterface.MAX_SENSE - Constant.

MAX_SENSE::OptimizationSense

An instance of the OptimizationSense enum.

About

The goal is to maximize the objective function.

source

MathOptInterface.FEASIBILITY_SENSE - Constant.

FEASIBILITY_SENSE::0ptimizationSense

An instance of the OptimizationSense enum.

About

The model does not have an objective function.

source

MathOptInterface.NumberOfVariables - Type.

NumberOfVariables()

An AbstractModelAttribute for the number of variables in the model.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.NumberOfVariables)::Int64

They should not implement set or supports.

source

MathOptInterface.ListOfVariableIndices - Type.

ListOfVariableIndices()

An AbstractModelAttribute for querying the Vector{MOI.VariableIndex} of all [MOI.VariableIndex] present in the model.

Order

The variables must be returned in the order in which they were added to the model.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.ListOfVariableIndices)::Vector{MOI.VariableIndex}

They should not implement set or supports.

source

MathOptInterface.ListOfConstraintTypesPresent - Type.

```
ListOfConstraintTypesPresent()
```

An AbstractModelAttribute for the list of tuples of the form (F, S), indicating that the attribute NumberOfConstraints{F, S} has a value greater than zero.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ListOfConstraintTypesPresent,
)::Vector{Tuple{Type,Type}}
```

They should not implement set or supports.

source

MathOptInterface.NumberOfConstraints - Type.

NumberOfConstraints{F,S}()

An AbstractModelAttribute for querying the number of constraints of the type F-in-S present in the model.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.NumberOfConstraints{F,S},
)::Int64 where {F<:MOI.AbstractFunction,MOI.AbstractSet}</pre>
```

They should not implement set or supports.

source

MathOptInterface.ListOfConstraintIndices - Type.

```
ListOfConstraintIndices{F,S}()
```

An AbstractModelAttribute for the Vector{MOI.ConstraintIndex{F,S}} of all constraint indices of type F-in-S in the model.

Order

The constraints must be returned in the order in which they were added to the model.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ListOfConstraintIndices{F,S},
)::Vector{MOI.ConstraintIndex{F,S}} where {F<:MOI.AbstractFunction,MOI.AbstractSet}</pre>
```

They should not implement set or supports.

source

MathOptInterface.ListOfOptimizerAttributesSet - Type.

ListOfOptimizerAttributesSet()

An AbstractOptimizerAttribute for the Vector{AbstractOptimizerAttribute} of all optimizer attributes that were set.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ListOfOptimizerAttributesSet,
)::Vector{MOI.AbstractOptimizerAttribute}
```

They should not implement set or supports.

source

MathOptInterface.ListOfModelAttributesSet - Type.

ListOfModelAttributesSet()

An AbstractModelAttribute for the Vector{AbstractModelAttribute} of all model attributes attr such that:

- 1. is_copyable(attr) returns true, and
- 2. the attribute was set to the model

source

MathOptInterface.ListOfVariableAttributesSet - Type.

ListOfVariableAttributesSet()

An AbstractModelAttribute for the Vector{AbstractVariableAttribute} of all variable attributes attr such that:

- 1. is_copyable(attr) returns true
- 2. the attribute was set for at least one variable in the model

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ListOfVariableAttributesSet,
)::Vector{MOI.AbstractVariableAttribute}
```

They should not implement set or supports.

source

MathOptInterface.ListOfVariablesWithAttributeSet - Type.

ListOfVariablesWithAttributeSet(attr::AbstractVariableAttribute)

An AbstractModelAttribute for the Vector{MOI.VariableIndex} of all variables with the attribute attr set.

The returned list may not be minimal, so some elements may have their default value set.

Note

This is an optional attribute to implement. The default fallback is to get ListOfVariableIndices.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ListOfVariablesWithAttributeSet{<:MOI.AbstractVariableAttribute},
)::Vector{MOI.VarialbeIndex}</pre>
```

They should not implement set or supports.

source

MathOptInterface.ListOfConstraintAttributesSet - Type.

```
ListOfConstraintAttributesSet{F, S}()
```

An AbstractModelAttribute for the Vector{AbstractConstraintAttribute} of all constraint attributes attr such that:

- 1. is_copyable(attr) returns true and
- 2. the attribute was set to F-in-S constraints.

Note

The attributes ConstraintFunction and ConstraintSet should not be included in the list even if then have been set with set.

source

MathOptInterface.ListOfConstraintsWithAttributeSet - Type.

ListOfConstraintsWithAttributeSet{F,S}(attr:AbstractConstraintAttribute)

An AbstractModelAttribute for the Vector{ConstraintIndex{F,S}} of all constraints with the attribute attr set.

The returned list may not be minimal, so some elements may have their default value set.

Note

This is an optional attribute to implement. The default fallback is to get ListOfConstraintIndices.

source

MathOptInterface.UserDefinedFunction - Type.

```
UserDefinedFunction(name::Symbol, arity::Int) <: AbstractModelAttribute</pre>
```

Set this attribute to register a user-defined function by the name of name with arity arguments.

Once registered, name will appear in ListOfSupportedNonlinearOperators.

You cannot register multiple UserDefinedFunctions with the same name but different arity.

Value type

The value to be set is a tuple containing one, two, or three functions to evaluate the function, the first-order derivative, and the second-order derivative respectively. Both derivatives are optional, but if you pass the second-order derivative you must also pass the first-order derivative.

For univariate functions with arity = 1, the functions in the tuple must have the form:

- f(x::T)::T: returns the value of the function at x
- $\nabla f(x::T)::T$: returns the first-order derivative of f with respect to x
- $\nabla^2 f(x::T)::T$: returns the second-order derivative of f with respect to x.

For multivariate functions with arity > 1, the functions in the tuple must have the form:

- f(x::T...)::T: returns the value of the function at x
- ∇f(g::AbstractVector{T}, x::T...)::Nothing: fills the components of g, with g[i] being the first-order partial derivative of f with respect to x[i]
- ∇²f(H::AbstractMatrix{T}, x::T...)::Nothing: fills the non-zero components of H, with H[i, j] being the second-order partial derivative of f with respect to x[i] and then x[j]. H is initialized to the zero matrix, so you do not need to set any zero elements.

Example

```
julia> f(x, y) = x^2 + y^2
f (generic function with 1 method)
julia> function ∇f(g, x, y)
           g .= 2 * x, 2 * y
           return
       end
\nabla f (generic function with 1 method)
julia> function \nabla^2 f(H, x...)
           H[1, 1] = H[2, 2] = 2.0
           return
       end
\nabla^2 f (generic function with 1 method)
julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}());
julia> MOI.set(model, MOI.UserDefinedFunction(:f, 2), (f,))
julia> MOI.set(model, MOI.UserDefinedFunction(:g, 2), (f, ⊽f))
julia> MOI.set(model, MOI.UserDefinedFunction(:h, 2), (f, \nabla f, \nabla^2 f))
```

```
julia> x = MOI.add_variables(model, 2)
2-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(1)
MOI.VariableIndex(2)
```

julia> MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)

julia> obj_f = MOI.ScalarNonlinearFunction(:f, Any[x[1], x[2]])
f(MOI.VariableIndex(1), MOI.VariableIndex(2))

julia> MOI.set(model, MOI.ObjectiveFunction{typeof(obj_f)}(), obj_f)

```
julia> print(model)
Minimize ScalarNonlinearFunction:
  f(v[1], v[2])
```

Subject to:

source

MathOptInterface.ListOfSupportedNonlinearOperators - Type.

ListOfSupportedNonlinearOperators() <: AbstractModelAttribute</pre>

When queried with get, return a Vector{Symbol} listing the operators supported by the model. These operators may appear in the head field of ScalarNonlinearFunction.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ListOfSupportedNonlinearOperators,
)::Vector{Symbol}
```

They should not implement set or supports.

source

Optimizer interface

MathOptInterface.AbstractOptimizer - Type.

AbstractOptimizer <: ModelLike

Abstract supertype for objects representing an instance of an optimization problem tied to a particular solver. This is typically a solver's in-memory representation. In addition to ModelLike, AbstractOptimizer objects let you solve the model and query the solution.

source

MathOptInterface.OptimizerWithAttributes - Type.

```
struct OptimizerWithAttributes
    optimizer_constructor
    params::Vector{Pair{AbstractOptimizerAttribute,<:Any}}
end</pre>
```

Object grouping an optimizer constructor and a list of optimizer attributes. Instances are created with instantiate.

source

MathOptInterface.optimize! - Function.

optimize!(optimizer::AbstractOptimizer)

Optimize the problem contained in optimizer.

Before calling optimize!, the problem should first be constructed using the incremental interface (see supports_incremental_interface) or copy_to.

source

MathOptInterface.optimize! - Method.

optimize!(dest::AbstractOptimizer, src::ModelLike)::Tuple{IndexMap,Bool}

A "one-shot" call that copies the problem from src into dest and then uses dest to optimize the problem.

Returns a tuple of an IndexMap and a Bool copied.

- The IndexMap object translates variable and constraint indices from the src model to the corresponding indices in the dest optimizer. See copy_to for details.
- If copied == true, src was copied to dest and then cached, allowing incremental modification if supported by the solver.
- If copied == false, a cache of the model was not kept in dest. Therefore, only the solution information (attributes for which is_set_by_optimize is true) is available to query.

Note

The main purpose of optimize! method with two arguments is for use in Utilities.CachingOptimizer.

Relationship to the single-argument optimize!

The default fallback of optimize! (dest::AbstractOptimizer, src::ModelLike) is

```
function optimize!(dest::AbstractOptimizer, src::ModelLike)
    index_map = copy_to(dest, src)
    optimize!(dest)
    return index_map, true
end
```

Therefore, subtypes of AbstractOptimizer should either implement this two-argument method, or implement both copy_to(::Optimizer, ::ModelLike) and optimize!(::Optimizer).

source

MathOptInterface.instantiate - Function.

```
instantiate(
    optimizer_constructor,
    with_cache_type::Union{Nothing,Type} = nothing,
    with_bridge_type::Union{Nothing,Type} = nothing,
)
```

Create an instance of an optimizer by either:

- calling optimizer_constructor.optimizer_constructor() and setting the parameters in optimizer_constructor.p if optimizer_constructor is a OptimizerWithAttributes
- calling optimizer_constructor() if optimizer_constructor is callable.

withcachetype

If with_cache_type is not nothing, then the optimizer is wrapped in a Utilities.CachingOptimizer to store a cache of the model. This is most useful if the optimizer you are constructing does not support the incremental interface (see supports_incremental_interface).

withbridgetype

If with_bridge_type is not nothing, the optimizer is wrapped in a Bridges.full_bridge_optimizer, enabling all the bridges defined in the MOI.Bridges submodule with coefficient type with_bridge_type.

In addition, if the optimizer created by optimizer_constructor does not support the incremental interface (see supports_incremental_interface), then, irrespective of with_cache_type, the optimizer is wrapped in a Utilities.CachingOptimizer to store a cache of the bridged model.

If with_cache_type and with_bridge_type are both not nothing, then they must be the same type.

source

MathOptInterface.default_cache - Function.

default_cache(optimizer::ModelLike, ::Type{T}) where {T}

Return a new instance of the default model type to be used as cache for optimizer in a Utilities.CachingOptimizer for holding constraints of coefficient type T. By default, this returns Utilities.UniversalFallback(Utilities.Model{T}()) If copying from a instance of a given model type is faster for optimizer then a new method returning an instance of this model type should be defined.

source

Optimizer attributes

MathOptInterface.AbstractOptimizerAttribute - Type.

AbstractOptimizerAttribute

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of the optimizer.

Notes

The difference between AbstractOptimizerAttribute and AbstractModelAttribute lies in the behavior of is_empty, empty! and copy_to. Typically optimizer attributes affect only how the model is solved.

source

MathOptInterface.SolverName - Type.

SolverName()

An AbstractOptimizerAttribute for the string identifying the solver.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.SolverName)::String

They should not implement set or supports.

source

MathOptInterface.SolverVersion - Type.

SolverVersion()

An AbstractOptimizerAttribute for the string identifying the version of the solver.

Versioning systems

For solvers supporting semantic versioning, the SolverVersion should be a string of the form "vMA-JOR.MINOR.PATCH", so that it can be converted to a Julia VersionNumber (for example, 'VersionNumber("v1.2.3")).

We do not require Semantic Versioning because some solvers use alternate versioning systems. For example, CPLEX uses Calendar Versioning, so SolverVersion will return a string like "202001".

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.SolverVersion)::String

They should not implement set or supports.

source

MathOptInterface.Silent - Type.

Silent()

An AbstractOptimizerAttribute for silencing the output of an optimizer.

When set to true, this attribute takes precedence over any other attribute controlling verbosity and requires the optimizer to produce no output.

The default value is false which has no effect. In this case the verbosity is controlled by other optimizerspecific attributes.

Value and default

The provided value must be a Bool.

The default value is false.

Note

Every optimizer should have verbosity on by default. For instance, if a solver has a solver-specific log level attribute, the MOI implementation should set it to 1 by default. If the user sets Silent to true, then the log level should be set to 0, even if the user specifically sets a value of log level. If the value of Silent is false then the log level set to the solver is the value given by the user for this solver-specific parameter or 1 if none is given.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.Silent)::Bool
MOI.set(::Optimizer, ::MOI.Silent, ::Bool)::Nothing
MOI.supports(::Optimizer, ::MOI.Silent)::Bool
```

source

MathOptInterface.TimeLimitSec - Type.

TimeLimitSec()

An AbstractOptimizerAttribute for setting a time limit (in seconds) for a call to optimize!.

Value and default

The provided limit must be a Union{Nothing, Real}.

When set to nothing, it deactivates the time limit.

The default value is nothing.

TerminationStatus

The optimizer may stop when the SolveTimeSec is larger than the TimeLimitSec. If stopped because of this limit, the TerminationStatus must be TIME LIMIT.

Note that most optimizers do not strictly respect a time limit. Instead, they terminate at the first convenient time after the time limit has been exceeded. Thus, you may find that the SolveTimeSec exceeds the TimeLimitSec by a few seconds.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.TimeLimitSec)::Union{Nothing,Float64}
MOI.set(::Optimizer, ::MOI.TimeLimitSec, ::Union{Nothing,Real})::Nothing
MOI.supports(::Optimizer, ::MOI.TimeLimitSec)::Bool
```

source

MathOptInterface.ObjectiveLimit - Type.

ObjectiveLimit()

An AbstractOptimizerAttribute for setting a limit on the objective value.

Value and default

The provided limit must be a Union{Nothing,Real}.

When set to nothing, the limit reverts to the solver's default.

The default value is nothing.

TerminationStatus

The solver may stop when the ObjectiveValue is better (lower for minimization, higher for maximization) than the ObjectiveLimit. If stopped, because of this limit, the TerminationStatus should be OBJECTIVE_LIMIT.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.ObjectiveLimit)::Union{Nothing,Float64}
MOI.set(::Optimizer, ::MOI.ObjectiveLimit, ::Union{Nothing,Real})::Nothing
MOI.supports(::Optimizer, ::MOI.ObjectiveLimit)::Bool
```

source

MathOptInterface.SolutionLimit - Type.

SolutionLimit()

An AbstractOptimizerAttribute for setting a limit on the number of available feasible solutions.

Value and default

The provided limit must be a Union{Nothing, Int}.

When set to nothing, the limit reverts to the solver's default.

The default value is nothing.

Termination criteria

The solver may stop when the ResultCount is larger than or equal to the SolutionLimit. If stopped because of this attribute, the TerminationStatus must be SOLUTION LIMIT.

Solution quality

The quality of the available solutions is solver-dependent. The set of resulting solutions is not guaranteed to contain an optimal solution.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.SolutionLimit)::Union{Nothing,Int}
MOI.set(::Optimizer, ::MOI.SolutionLimit, ::Union{Nothing,Int})::Nothing
MOI.supports(::Optimizer, ::MOI.SolutionLimit)::Bool
```

source

MathOptInterface.NodeLimit - Type.

NodeLimit()

An AbstractOptimizerAttribute for setting a limit on the number of branch-and-bound nodes explored by a mixed-integer program (MIP) solver.

Value and default

The provided limit must be a Union{Nothing, Int}.

When set to nothing, the limit reverts to the solver's default.

The default value is nothing.

Termination criteria

The solver may stop when the NodeCount is larger than or equal to the NodeLimit. If stopped because of this attribute, the TerminationStatus must be NODE_LIMIT.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.NodeLimit)::Union{Nothing,Int}
MOI.set(::Optimizer, ::MOI.NodeLimit, ::Union{Nothing,Int})::Nothing
MOI.supports(::Optimizer, ::MOI.NodeLimit)::Bool
```

MathOptInterface.RawOptimizerAttribute - Type.

RawOptimizerAttribute(name::String)

An AbstractOptimizerAttribute for the solver-specific parameter identified by name.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.RawOptimizerAttribute)::Any
MOI.set(::Optimizer, ::MOI.RawOptimizerAttribute, ::Any)::Nothing
MOI.supports(::Optimizer, ::MOI.RawOptimizerAttribute)::Bool
```

source

MathOptInterface.NumberOfThreads - Type.

NumberOfThreads()

An AbstractOptimizerAttribute for setting the number of threads used for an optimization.

Value and default

The provided value must be nothing or a positive Int.

When set to nothing, the value reverts to the solver's default.

The default value is nothing.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.NumberOfThreads)::Union{Nothing,Int}
MOI.set(::Optimizer, ::MOI.NumberOfThreads, ::Union{Nothing,Int})::Nothing
MOI.supports(::Optimizer, ::MOI.NumberOfThreads)::Bool
```

source

MathOptInterface.RawSolver - Type.

RawSolver()

An AbstractModelAttribute for the object that may be used to access a solver-specific API for this optimizer.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.RawSolver)::Any

They should not implement set or supports.

source

MathOptInterface.AbsoluteGapTolerance - Type.

AbsoluteGapTolerance()

An AbstractOptimizerAttribute for setting the absolute gap tolerance for an optimization.

Definition

The mathematical definition of "absolute gap" and its allowed range are solver-dependent. However, most solvers that implement this attribute will stop once $|f - b| g_{abs}$, where b is the best bound, f is the best feasible objective value, and g_{abs} is the absolute gap.

Value and default

The provided value must be a Union{Nothing, Float64}.

When set to nothing, the limit reverts to the solver's default.

TerminationStatus

The optimizer may stop when the absolute difference between ObjectiveValue and ObjectiveBound is smaller than the AbsoluteGapTolerance. If stopped because of this limit, the TerminationStatus may be OPTIMAL.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.AbsoluteGapTolerance)::Union{Nothing,Float64}
MOI.set(
    ::Optimizer,
    ::MOI.AbsoluteGapTolerance,
    ::Union{Nothing,Float64},
 )::Nothing
MOI.supports(::Optimizer, ::MOI.AbsoluteGapTolerance)::Bool
```

source

MathOptInterface.RelativeGapTolerance - Type.

```
RelativeGapTolerance()
```

An AbstractOptimizerAttribute for setting the relative gap tolerance for an optimization.

Definition

The mathematical definition of "relative gap" and its allowed range are solver-dependent. Typically, solvers expect a value between 0.0 and 1.0.

Value and default

The provided value must be a Union{Nothing, Float64}.

When set to nothing, the limit reverts to the solver's default.

TerminationStatus

The optimizer may stop when the RelativeGap is smaller than the RelativeGapTolerance. If stopped because of this limit, the TerminationStatus may be OPTIMAL.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.RelativeGapTolerance)::Union{Nothing,Float64}
MOI.set(
    ::Optimizer,
    ::MOI.RelativeGapTolerance,
    ::Union{Nothing,Float64},
)::Nothing
MOI.supports(::Optimizer, ::MOI.RelativeGapTolerance)::Bool
```

source

MathOptInterface.AutomaticDifferentiationBackend - Type.

```
AutomaticDifferentiationBackend() <: AbstractOptimizerAttribute</pre>
```

An AbstractOptimizerAttribute for setting the automatic differentiation backend used by the solver.

The value must be a subtype of Nonlinear.AbstractAutomaticDifferentiation.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::0ptimizer,
    ::MOI.AutomaticDifferentiationBackend,
)::MOI.Nonlinear.AbstractAutomaticDifferentiation
MOI.set(
    ::0ptimizer,
    ::MOI.AutomaticDifferentiationBackend,
    ::MOI.Nonlinear.AbstractAutomaticDifferentiation,
)::Nothing
MOI.supports(::0ptimizer, ::MOI.AutomaticDifferentiationBackend)::Bool
```

source

List of attributes useful for optimizers

MathOptInterface.TerminationStatus - Type.

TerminationStatus()

An AbstractModelAttribute for the TerminationStatusCode explaining why the optimizer stopped.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.TerminationStatus)::MOI.TerminationStatusCode

They should not implement set or supports.

source

MathOptInterface.TerminationStatusCode - Type.

TerminationStatusCode

An Enum of possible values for the TerminationStatus attribute.

This attribute explains why the optimizer stopped executing in the most recent call to optimize!.

Values

OPTIMIZE NOT CALLED

The algorithm has not started.

OPTIMAL

The algorithm found a globally optimal solution.

INFEASIBLE

The algorithm proved that no primal feasible solution exists.

DUAL_INFEASIBLE

The algorithm proved that no dual feasible solution exists.

To check if the primal problem is feasible, set the objective sense to FEASIBILITY_SENSE and re-solve the problem.

If a primal feasible point does not exist, the original problem is both primal and dual infeasible.

If a primal feasible solution exists, this status typically implies that the problem is unbounded, with some technical exceptions (for example, if the problem is a conic optimization problem in which strong duality does not hold).

The technical exceptions do not apply to linear programs. The combination of DUAL_INFEASIBLE and a primal feasible point means that the primal linear program is unbounded.

LOCALLY_SOLVED

The algorithm converged to a stationary point, local optimal solution, could not find directions for improvement, or otherwise completed its search without global guarantees.

LOCALLY_INFEASIBLE

The algorithm converged to an infeasible point or otherwise completed its search without finding a feasible solution, without guarantees that no feasible solution exists.

If you know a primal feasible solution exists, use VariablePrimalStart to provide a feasible starting point to the solver.

INFEASIBLE OR UNBOUNDED

The algorithm stopped because it proved that the problem is infeasible or unbounded, without distinguishing between the two cases.

To distinguish between the two cases, set the objective sense to FEASIBILITY_SENSE and re-solve the problem. If a primal feasible point exists, the original problem is unbounded. If a primal feasible point does not exist, the original problem is infeasible.

ALMOST_OPTIMAL

The algorithm found a globally optimal solution to relaxed tolerances.

ALMOST INFEASIBLE

The algorithm concluded that no feasible solution exists within relaxed tolerances.

ALMOST_DUAL_INFEASIBLE

The algorithm concluded that no dual bound exists for the problem within relaxed tolerances.

ALMOST LOCALLY SOLVED

The algorithm converged to a stationary point, local optimal solution, or could not find directions for improvement within relaxed tolerances.

ITERATION_LIMIT

An iterative algorithm stopped after conducting the maximum number of iterations.

TIME_LIMIT

The algorithm stopped after a user-specified computation time.

This status may be returned in relation to the TimeLimitSec attribute, or some other solver-specific attribute.

NODE_LIMIT

A branch-and-bound algorithm stopped because it explored a maximum number of nodes in the branchand-bound tree.

This status may be returned in relation to the NodeLimit attribute, or some other solver-specific attribute.

SOLUTION LIMIT

The algorithm stopped because it found the required number of solutions. This is often used in MIPs to get the solver to return the first feasible solution it encounters.

This status may be returned in relation to the SolutionLimit attribute, or some other solver-specific attribute.

MEMORY_LIMIT

The algorithm stopped because it ran out of memory.

OBJECTIVE_LIMIT

The algorithm stopped because it found a solution better than a minimum limit set by the user.

This status may be returned in relation to the ObjectiveLimit attribute, or some other solver-specific attribute.

NORM LIMIT

The algorithm stopped because the norm of an iterate became too large.

This typically means that the primal problem is unbounded, but that the solver could not prove so.

OTHER_LIMIT

The algorithm stopped due to a limit not covered by one of the _LIMIT_ statuses above.

SLOW_PROGRESS

The algorithm stopped because it was unable to continue making progress towards the solution.

NUMERICAL ERROR

The algorithm stopped because it encountered unrecoverable numerical error.

INVALID MODEL

The algorithm stopped because the model is invalid.

The reason for this return code is solver-specific, but common causes are that the problem has zero variables or constraints, or that the problem data contains an invalid number such as NaN.

INVALID_OPTION

The algorithm stopped because it was provided an invalid option.

INTERRUPTED

The algorithm stopped because of an interrupt signal.

This typically means that the solver was interrupted by the user with CTRL+C.

OTHER ERROR

The algorithm stopped because of an error not covered by one of the statuses defined above. Check the solver log for further details.

source

MathOptInterface.OPTIMIZE_NOT_CALLED - Constant.

OPTIMIZE_NOT_CALLED::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm has not started.

source

MathOptInterface.OPTIMAL - Constant.

OPTIMAL::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm found a globally optimal solution.

source

MathOptInterface.INFEASIBLE - Constant.

INFEASIBLE::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm proved that no primal feasible solution exists.

source

MathOptInterface.DUAL_INFEASIBLE - Constant.

DUAL_INFEASIBLE::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm proved that no dual feasible solution exists.

To check if the primal problem is feasible, set the objective sense to FEASIBILITY_SENSE and re-solve the problem.

If a primal feasible point does not exist, the original problem is both primal and dual infeasible.

If a primal feasible solution exists, this status typically implies that the problem is unbounded, with some technical exceptions (for example, if the problem is a conic optimization problem in which strong duality does not hold).

The technical exceptions do not apply to linear programs. The combination of DUAL_INFEASIBLE and a primal feasible point means that the primal linear program is unbounded.

source

MathOptInterface.LOCALLY_SOLVED - Constant.

LOCALLY_SOLVED:::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm converged to a stationary point, local optimal solution, could not find directions for improvement, or otherwise completed its search without global guarantees.

source

MathOptInterface.LOCALLY_INFEASIBLE - Constant.

LOCALLY_INFEASIBLE::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm converged to an infeasible point or otherwise completed its search without finding a feasible solution, without guarantees that no feasible solution exists.

If you know a primal feasible solution exists, use VariablePrimalStart to provide a feasible starting point to the solver.

source

MathOptInterface.INFEASIBLE OR UNBOUNDED - Constant.

INFEASIBLE_OR_UNBOUNDED::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because it proved that the problem is infeasible or unbounded, without distinguishing between the two cases.

To distinguish between the two cases, set the objective sense to FEASIBILITY_SENSE and re-solve the problem. If a primal feasible point exists, the original problem is unbounded. If a primal feasible point does not exist, the original problem is infeasible.

source

MathOptInterface.ALMOST_OPTIMAL - Constant.

ALMOST_OPTIMAL:: TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm found a globally optimal solution to relaxed tolerances.

source

MathOptInterface.ALMOST_INFEASIBLE - Constant.

ALMOST_INFEASIBLE::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm concluded that no feasible solution exists within relaxed tolerances.

source

MathOptInterface.ALMOST_DUAL_INFEASIBLE - Constant.

ALMOST_DUAL_INFEASIBLE::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm concluded that no dual bound exists for the problem within relaxed tolerances.

source

MathOptInterface.ALMOST_LOCALLY_SOLVED - Constant.

ALMOST_LOCALLY_SOLVED::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm converged to a stationary point, local optimal solution, or could not find directions for improvement within relaxed tolerances.

source

MathOptInterface.ITERATION_LIMIT - Constant.

ITERATION_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

An iterative algorithm stopped after conducting the maximum number of iterations.

source

MathOptInterface.TIME_LIMIT - Constant.

TIME_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped after a user-specified computation time.

This status may be returned in relation to the TimeLimitSec attribute, or some other solver-specific attribute.

source

MathOptInterface.NODE_LIMIT - Constant.

NODE_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

A branch-and-bound algorithm stopped because it explored a maximum number of nodes in the branchand-bound tree.

This status may be returned in relation to the NodeLimit attribute, or some other solver-specific attribute.

source

MathOptInterface.SOLUTION_LIMIT - Constant.

SOLUTION_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because it found the required number of solutions. This is often used in MIPs to get the solver to return the first feasible solution it encounters.

This status may be returned in relation to the SolutionLimit attribute, or some other solver-specific attribute.

source

MathOptInterface.MEMORY_LIMIT - Constant.

MEMORY_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because it ran out of memory.

source

MathOptInterface.OBJECTIVE_LIMIT - Constant.

OBJECTIVE_LIMIT::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because it found a solution better than a minimum limit set by the user.

This status may be returned in relation to the **ObjectiveLimit** attribute, or some other solver-specific attribute.

source

MathOptInterface.NORM_LIMIT - Constant.

NORM_LIMIT:: TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because the norm of an iterate became too large.

This typically means that the primal problem is unbounded, but that the solver could not prove so.

source

MathOptInterface.OTHER_LIMIT - Constant.

OTHER_LIMIT:: TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped due to a limit not covered by one of the _LIMIT_ statuses above.

source

MathOptInterface.SLOW_PROGRESS - Constant.

SLOW_PROGRESS::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because it was unable to continue making progress towards the solution.

```
source
```

MathOptInterface.NUMERICAL_ERROR - Constant.

NUMERICAL_ERROR::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because it encountered unrecoverable numerical error.

source

MathOptInterface.INVALID_MODEL - Constant.

INVALID_MODEL::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because the model is invalid.

The reason for this return code is solver-specific, but common causes are that the problem has zero variables or constraints, or that the problem data contains an invalid number such as NaN.

source

MathOptInterface.INVALID_OPTION - Constant.

INVALID_OPTION::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because it was provided an invalid option.

source

MathOptInterface.INTERRUPTED - Constant.

INTERRUPTED::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because of an interrupt signal.

This typically means that the solver was interrupted by the user with CTRL+C.

source

MathOptInterface.OTHER_ERROR - Constant.

OTHER_ERROR::TerminationStatusCode

An instance of the TerminationStatusCode enum.

About

The algorithm stopped because of an error not covered by one of the statuses defined above. Check the solver log for further details.

source

MathOptInterface.PrimalStatus - Type.
PrimalStatus(result_index::Int = 1)

An AbstractModelAttribute for the ResultStatusCode of the primal result result index.

result_index

If result_index is omitted, it defaults to 1.

See ResultCount for information on how the results are ordered.

If result_index is larger than the value of ResultCount then NO_SOLUTION is returned.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.PrimalStatus)::MOI.ResultStatusCode

They should not implement set or supports.

source

MathOptInterface.DualStatus - Type.

DualStatus(result_index::Int = 1)

An AbstractModelAttribute for the ResultStatusCode of the dual result result_index.

result_index

See ResultCount for information on how the results are ordered.

If result_index is larger than the value of ResultCount then NO_SOLUTION is returned.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.PrimalStatus)::MOI.ResultStatusCode

They should not implement set or supports.

source

MathOptInterface.RawStatusString - Type.

RawStatusString()

An AbstractModelAttribute for a solver specific string explaining why the optimizer stopped.

Implementation

Optimizers should implement the following methods:

CHAPTER 36. API REFERENCE

MOI.get(::Optimizer, ::MOI.RawStatusString)::MOI.String

They should not implement set or supports.

source

MathOptInterface.ResultCount - Type.

ResultCount()

An AbstractModelAttribute for the number of results available.

Order of solutions

A number of attributes contain an index, result_index, which is used to refer to one of the available results. Thus, result_index must be an integer between 1 and the number of available results.

As a general rule, the first result (result_index = 1) is the most important result (for example, an optimal solution or an infeasibility certificate). Other results will typically be alternate solutions that the solver found during the search for the first result.

If a (local) optimal solution is available, that is, TerminationStatus is OPTIMAL or LOCALLY_SOLVED, the first result must correspond to the (locally) optimal solution. Other results may be alternative optimal solutions, or they may be other suboptimal solutions; use ObjectiveValue to distinguish between them.

If a primal or dual infeasibility certificate is available, that is, TerminationStatus is INFEASIBLE or DUAL_INFEASIBLE and the corresponding PrimalStatus or DualStatus is INFEASIBILITY_CERTIFICATE, then the first result must be a certificate. Other results may be alternate certificates, or infeasible points.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.ResultCount)::Int

They should not implement set or supports.

source

MathOptInterface.ObjectiveValue - Type.

ObjectiveValue(result_index::Int = 1)

An AbstractModelAttribute for the objective value of the primal solution result_index.

PrimalStatus

Before quering this attribute you should first check PrimalStatus to confirm that a primal solution is avaiable.

If the PrimalStatus is NO_SOLUTION the result of querying this attribute is undefined.

result_index

The optimizer may return multiple primal solutions. See ResultCount for information on how the results are ordered.

If the solver does not have a primal value for the objective because the result_index is beyond the available solutions (whose number is indicated by the ResultCount attribute), getting this attribute must throw a ResultIndexBoundsError.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ObjectiveValue,
)::Union{T,Vector{T}} where {T<:Real}</pre>
```

They should not implement set or supports.

```
source
```

MathOptInterface.DualObjectiveValue - Type.

```
DualObjectiveValue(result_index::Int = 1)
```

An AbstractModelAttribute for the value of the objective function of the dual solution result_index.

DualStatus

Before quering this attribute you should first check DualStatus to confirm that a dual solution is avaiable.

If the DualStatus is NO_SOLUTION the result of querying this attribute is undefined.

result_index

The optimizer may return multiple dual solutions. See ResultCount for information on how the results are ordered.

If the solver does not have a dual value for the objective because the result_index is beyond the available solutions (whose number is indicated by the ResultCount attribute), getting this attribute must throw a ResultIndexBoundsError.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.DualObjectiveValue,
)::Union{T,Vector{T}} where {T<:Real}</pre>
```

They should not implement set or supports.

source

MathOptInterface.ObjectiveBound - Type.

ObjectiveBound()

An AbstractModelAttribute for the best known bound on the optimal objective value.

source

MathOptInterface.RelativeGap - Type.

RelativeGap()

An AbstractModelAttribute for the final relative optimality gap.

Warning

The definition of this gap is solver-dependent. However, most solvers implementing this attribute define the relative gap as some variation of $\frac{|b-f|}{|f|}$, where b is the best bound and f is the best feasible objective value.

source

MathOptInterface.SolveTimeSec - Type.

SolveTimeSec()

An AbstractModelAttribute for the total elapsed solution time (in seconds) as reported by the optimizer.

source

MathOptInterface.SimplexIterations - Type.

SimplexIterations()

An AbstractModelAttribute for the cumulative number of simplex iterations while solving a problem.

For a mixed-integer program (MIP), the return value is the total simplex iterations for all nodes.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.SimplexIterations)::Int64

They should not implement set or supports.

source

MathOptInterface.BarrierIterations - Type.

BarrierIterations()

An AbstractModelAttribute for the cumulative number of barrier iterations while solving a problem.

For a mixed-integer program (MIP), the return value is the total barrier iterations for all nodes.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.BarrierIterations)::Int64

They should not implement set or supports.

source

MathOptInterface.NodeCount - Type.

NodeCount()

An AbstractModelAttribute for the total number of branch-and-bound nodes explored while solving a mixed-integer program (MIP).

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.NodeCount)::Int64

They should not implement set or supports.

source

ResultStatusCode

MathOptInterface.ResultStatusCode - Type.

ResultStatusCode

An Enum of possible values for the PrimalStatus and DualStatus attributes.

The values indicate how to interpret the result vector.

Values

NO_SOLUTION

The result vector is empty.

FEASIBLE_POINT

The result vector is a feasible point.

NEARLY_FEASIBLE_POINT

The result vector is feasible if some constraint tolerances are relaxed.

INFEASIBLE POINT

The result vector is an infeasible point.

INFEASIBILITY_CERTIFICATE

The result vector is an infeasibility certificate.

If the PrimalStatus is INFEASIBILITY_CERTIFICATE, then the primal result vector is a certificate of dual infeasibility.

If the DualStatus is INFEASIBILITY_CERTIFICATE, then the dual result vector is a proof of primal infeasibility.

NEARLY_INFEASIBILITY_CERTIFICATE

The result satisfies a relaxed criterion for a certificate of infeasibility.

If the PrimalStatus is NEARLY_INFEASIBILITY_CERTIFICATE, then the primal result vector is a certificate of dual infeasibility.

If the DualStatus is NEARLY_INFEASIBILITY_CERTIFICATE, then the dual result vector is a proof of primal infeasibility.

REDUCTION_CERTIFICATE

The result vector is an ill-posed certificate; see this article for details.

If the PrimalStatus is REDUCTION_CERTIFICATE, then the primal result vector is a proof that the dual problem is ill-posed.

If the DualStatus is REDUCTION_CERTIFICATE, then the dual result vector is a proof that the primal is ill-posed.

NEARLY REDUCTION CERTIFICATE

The result satisfies a relaxed criterion for an ill-posed certificate.

UNKNOWN_RESULT_STATUS

The result vector contains a solution with an unknown interpretation. Check the solver log for more details.

OTHER_RESULT_STATUS

The result vector contains a solution with an interpretation not covered by one of the statuses defined above. Check the solver log for more details.

source

MathOptInterface.NO_SOLUTION - Constant.

NO_SOLUTION::ResultStatusCode

An instance of the ResultStatusCode enum.

About

The result vector is empty.

source

MathOptInterface.FEASIBLE_POINT - Constant.

FEASIBLE POINT::ResultStatusCode

An instance of the ResultStatusCode enum.

About

The result vector is a feasible point.

source

MathOptInterface.NEARLY_FEASIBLE_POINT - Constant.

NEARLY_FEASIBLE_POINT::ResultStatusCode

An instance of the ResultStatusCode enum.

About

The result vector is feasible if some constraint tolerances are relaxed.

source

MathOptInterface.INFEASIBLE_POINT - Constant.

INFEASIBLE_POINT::ResultStatusCode

An instance of the ResultStatusCode enum.

About

The result vector is an infeasible point.

source

MathOptInterface.INFEASIBILITY_CERTIFICATE - Constant.

INFEASIBILITY_CERTIFICATE::ResultStatusCode

An instance of the ResultStatusCode enum.

About

The result vector is an infeasibility certificate.

If the PrimalStatus is INFEASIBILITY_CERTIFICATE, then the primal result vector is a certificate of dual infeasibility.

If the DualStatus is INFEASIBILITY_CERTIFICATE, then the dual result vector is a proof of primal infeasibility.

source

MathOptInterface.NEARLY_INFEASIBILITY_CERTIFICATE - Constant.

NEARLY_INFEASIBILITY_CERTIFICATE::**ResultStatusCode**

An instance of the ResultStatusCode enum.

About

The result satisfies a relaxed criterion for a certificate of infeasibility.

If the PrimalStatus is NEARLY_INFEASIBILITY_CERTIFICATE, then the primal result vector is a certificate of dual infeasibility.

If the DualStatus is NEARLY_INFEASIBILITY_CERTIFICATE, then the dual result vector is a proof of primal infeasibility.

source

MathOptInterface.REDUCTION_CERTIFICATE - Constant.

REDUCTION_CERTIFICATE::ResultStatusCode

An instance of the ResultStatusCode enum.

About

The result vector is an ill-posed certificate; see this article for details.

If the PrimalStatus is REDUCTION_CERTIFICATE, then the primal result vector is a proof that the dual problem is ill-posed.

If the DualStatus is REDUCTION_CERTIFICATE, then the dual result vector is a proof that the primal is ill-posed.

source

MathOptInterface.NEARLY_REDUCTION_CERTIFICATE - Constant.

NEARLY_REDUCTION_CERTIFICATE::ResultStatusCode

An instance of the ResultStatusCode enum.

About

The result satisfies a relaxed criterion for an ill-posed certificate.

source

MathOptInterface.UNKNOWN_RESULT_STATUS - Constant.

UNKNOWN_RESULT_STATUS::ResultStatusCode

An instance of the ResultStatusCode enum.

About

The result vector contains a solution with an unknown interpretation. Check the solver log for more details.

MathOptInterface.OTHER_RESULT_STATUS - Constant.

OTHER_RESULT_STATUS::ResultStatusCode

An instance of the ResultStatusCode enum.

About

The result vector contains a solution with an interpretation not covered by one of the statuses defined above. Check the solver log for more details.

source

Conflict Status

MathOptInterface.compute_conflict! - Function.

compute_conflict!(optimizer::AbstractOptimizer)

Computes a minimal subset of constraints such that the model with the other constraint removed is still infeasible.

Some solvers call a set of conflicting constraints an Irreducible Inconsistent Subsystem (IIS).

See also ConflictStatus and ConstraintConflictStatus.

Note

If the model is modified after a call to compute_conflict!, the implementor is not obliged to purge the conflict. Any calls to the above attributes may return values for the original conflict without a warning. Similarly, when modifying the model, the conflict can be discarded.

source

MathOptInterface.ConflictStatus - Type.

ConflictStatus()

An AbstractModelAttribute for the ConflictStatusCode explaining why [compute_conflict!] stopped when computing the conflict.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.ConflictStatus)::MOI.ConflictStatusCode

They should not implement set or supports.

source

MathOptInterface.ConflictStatusCode - Type.

ConflictStatusCode

An Enum of possible values for the ConflictStatus attribute.

This attribute is meant to explain the reason why the conflict finder stopped executing in the most recent call to compute_conflict!.

Values

COMPUTE_CONFLICT_NOT_CALLED

The function compute_conflict! has not yet been called.

NO_CONFLICT_EXISTS

There is no conflict because the problem is feasible.

NO_CONFLICT_FOUND

The solver could not find a conflict.

CONFLICT_FOUND

The solver found a conflict.

source

MathOptInterface.COMPUTE_CONFLICT_NOT_CALLED - Constant.

COMPUTE_CONFLICT_NOT_CALLED::ConflictStatusCode

An instance of the ConflictStatusCode enum.

About

The function compute_conflict! has not yet been called.

source

MathOptInterface.NO_CONFLICT_EXISTS - Constant.

N0_CONFLICT_EXISTS::ConflictStatusCode

An instance of the ConflictStatusCode enum.

About

There is no conflict because the problem is feasible.

source

MathOptInterface.NO_CONFLICT_FOUND - Constant.

NO_CONFLICT_FOUND:::ConflictStatusCode

An instance of the ConflictStatusCode enum.

About

The solver could not find a conflict.

source

MathOptInterface.CONFLICT_FOUND - Constant.

CONFLICT_FOUND::ConflictStatusCode

An instance of the ConflictStatusCode enum.

About

The solver found a conflict.

source

MathOptInterface.ConstraintConflictStatus - Type.

ConstraintConflictStatus()

A constraint attribute to query the ConflictParticipationStatusCode indicating whether the constraint participates in the conflict.

source

MathOptInterface.ConflictParticipationStatusCode - Type.

ConflictParticipationStatusCode

An Enum for the value of the ConstraintConflictStatus attribute.

This attribute is meant to indicate whether a given constraint participates or not in the last computed conflict.

Values

NOT_IN_CONFLICT

The constraint does not participate in the conflict.

IN_CONFLICT

The constraint participates in the conflict.

MAYBE_IN_CONFLICT

The solver was not able to prove whether the constraint is required to participate in the conflict.

source

MathOptInterface.NOT_IN_CONFLICT - Constant.

NOT_IN_CONFLICT::ConflictParticipationStatusCode

An instance of the ConflictParticipationStatusCode enum.

About

The constraint does not participate in the conflict.

source

MathOptInterface.IN_CONFLICT - Constant.

IN_CONFLICT::ConflictParticipationStatusCode

An instance of the ConflictParticipationStatusCode enum.

About

The constraint participates in the conflict.

source

MathOptInterface.MAYBE_IN_CONFLICT - Constant.

MAYBE_IN_CONFLICT::ConflictParticipationStatusCode

An instance of the ConflictParticipationStatusCode enum.

About

The solver was not able to prove whether the constraint is required to participate in the conflict.

source

36.3 Variables

Functions

MathOptInterface.add_variable - Function.

add_variable(model::ModelLike)::VariableIndex

Add a scalar variable to the model, returning a variable index.

A AddVariableNotAllowed error is thrown if adding variables cannot be done in the current state of the model model.

Example

julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

source

MathOptInterface.add_variables - Function.

```
add variables(model::ModelLike, n::Int)::Vector{VariableIndex}
```

Add n scalar variables to the model, returning a vector of variable indices.

An AddVariableNotAllowed error is thrown if adding variables cannot be done in the current state of the model model.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

```
julia> MOI.add_variables(model, 2)
2-element Vector{MathOptInterface.VariableIndex}:
    MOI.VariableIndex(1)
    MOI.VariableIndex(2)
```

source

MathOptInterface.add_constrained_variable - Function.

```
add_constrained_variable(
    model::ModelLike,
    set::AbstractScalarSet
)::Tuple{MOI.VariableIndex,
    MOI.ConstraintIndex{MOI.VariableIndex, typeof(set)}}
```

Add to model a scalar variable constrained to belong to set, returning the index of the variable created and the index of the constraint constraining the variable to belong to set.

By default, this function falls back to creating a free variable with add_variable and then constraining it to belong to set with add_constraint.

```
add_constrained_variable(
    model::ModelLike,
    set::Tuple{<:GreaterThan,<:LessThan},
)</pre>
```

A special-case method to add a scalar variable with a lower and upper bound.

This method should be implemented by optimizers which have native support for adding a variable with bounds and which cannot performantly modify the variable bounds after creation.

Example

source

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> set = (MOI.GreaterThan(1.0), MOI.LessThan(2.0));

julia> x, (c_l, c_u) = MOI.add_constrained_variable(model, set);

julia> c_l

MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, → MathOptInterface.GreaterThan{Float64}}(1)

julia> c_u

```
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,

→ MathOptInterface.LessThan{Float64}}(1)
```

julia> print(model)

```
Feasibility
```

Subject to:

```
VariableIndex-in-GreaterThan{Float64}
v[1] >= 1.0
```

VariableIndex-in-LessThan{Float64}
v[1] <= 2.0</pre>

source

MathOptInterface.add_constrained_variables - Function.

```
add_constrained_variables(
    model::ModelLike,
    sets::AbstractVector{<:AbstractScalarSet}
)::Tuple{
    Vector{MOI.VariableIndex},
    Vector{MOI.ConstraintIndex{MOI.VariableIndex,eltype(sets)}},
}</pre>
```

Add to model scalar variables constrained to belong to sets, returning the indices of the variables created and the indices of the constraints constraining the variables to belong to each set in sets. That is, if it returns variables and constraints, constraints[i] is the index of the constraint constraining variable[i] to belong to sets[i].

By default, this function falls back to calling add constrained variable on each set.

```
source
```

```
add_constrained_variables(
    model::ModelLike,
    set::AbstractVectorSet,
)::Tuple{
    Vector{MOI.VariableIndex},
    MOI.ConstraintIndex{MOI.VectorOfVariables,typeof(set)},
}
```

Add to model a vector of variables constrained to belong to set, returning the indices of the variables created and the index of the constraint constraining the vector of variables to belong to set.

By default, this function falls back to creating free variables with add_variables and then constraining it to belong to set with add_constraint.

source

MathOptInterface.supports_add_constrained_variable - Function.

```
supports_add_constrained_variable(
    model::ModelLike,
    S::Type{<:AbstractScalarSet}
)::Bool</pre>
```

Return a Bool indicating whether model supports constraining a variable to belong to a set of type S either on creation of the variable with add_constrained_variable or after the variable is created with add constraint.

By default, this function falls back to supports_add_constrained_variables(model, Reals) && supports_constraint(model. MOI.VariableIndex, S) which is the correct definition for most models.

Example

Suppose that a solver supports only two kind of variables: binary variables and continuous variables with a lower bound. If the solver decides not to support VariableIndex-in-Binary and VariableIndex-in-GreaterThan constraints, it only has to implement add_constrained_variable for these two sets which prevents the user to add both a binary constraint and a lower bound on the same variable. More-over, if the user adds a VariableIndex-in-GreaterThan constraint, implementing this interface (that is, supports_add_constrained_variables) enables the constraint to be transparently bridged into a supported constraint.

source

MathOptInterface.supports_add_constrained_variables - Function.

```
supports_add_constrained_variables(
    model::ModelLike,
    S::Type{<:AbstractVectorSet}
)::Bool</pre>
```

Return a Bool indicating whether model supports constraining a vector of variables to belong to a set of type S either on creation of the vector of variables with add_constrained_variables or after the variable is created with add constraint.

By default, if S is Reals then this function returns true and otherwise, it falls back to supports_add_constrained_variables (Reals) && supports_constraint(model, MOI.VectorOfVariables, S) which is the correct definition for most models.

Example

In the standard conic form (see Duality), the variables are grouped into several cones and the constraints are affine equality constraints. If Reals is not one of the cones supported by the solvers then it needs to implement supports_add_constrained_variables(::Optimizer, ::Type{Reals}) = false as free variables are not supported. The solvers should then implement supports_add_constrained_variables(::Optimizer, add_constrained_variables(::Optimizer, add_constrained_variables(::Opti

::Type{<:SupportedCones}) = true where SupportedCones is the union of all cone types that are supported; it does not have to implement the method supports_constraint(::Type{VectorOfVariables}, Type{<:SupportedCones}) as it should return false and it's the default. This prevents the user to constrain the same variable in two different cones. When a VectorOfVariables-in-S is added, the variables of the vector have already been created so they already belong to given cones. If bridges are enabled, the constraint will therefore be bridged by adding slack variables in S and equality constraints ensuring that the slack variables are equal to the corresponding variables of the given constraint function.

Note that there may also be sets for which !supports_add_constrained_variables(model, S) and supports_constraint(model, MOI.VectorOfVariables, S). For instance, suppose a solver supports positive semidefinite variable constraints and two types of variables: binary variables and nonnegative variables. Then the solver should support adding VectorOfVariables-in-PositiveSemidefiniteConeTriangle constraints, but it should not support creating variables constrained to belong to the PositiveSemidefiniteConeTriangle because the variables in PositiveSemidefiniteConeTriangle should first be created as either binary or non-negative.

source

MathOptInterface.is_valid - Method.

is_valid(model::ModelLike, index::Index)::Bool

Return a Bool indicating whether this index refers to a valid object in the model model.

source

MathOptInterface.delete - Method.

delete(model::ModelLike, index::Index)

Delete the referenced object from the model. Throw DeleteNotAllowed if if index cannot be deleted.

The following modifications also take effect if Index is VariableIndex:

- If index used in the objective function, it is removed from the function, that is, it is substituted for zero.
- For each func-in-set constraint of the model:
 - If func isa VariableIndex and func == index then the constraint is deleted.
 - If func isa VectorOfVariables and index in func.variables then
 - * if length(func.variables) == 1 is one, the constraint is deleted;
 - * iflength(func.variables) > 1 and supports_dimension_update(set) then then the variable is removed from func and set is replaced by update_dimension(set, MOI.dimension(set) - 1).
 - * Otherwise, a DeleteNotAllowed error is thrown.
 - Otherwise, the variable is removed from func, that is, it is substituted for zero.

source

MathOptInterface.delete - Method.

```
delete(model::ModelLike, indices::Vector{R<:Index}) where {R}</pre>
```

Delete the referenced objects in the vector indices from the model. It may be assumed that R is a concrete type. The default fallback sequentially deletes the individual items in indices, although specialized implementations may be more efficient.

source

Attributes

MathOptInterface.AbstractVariableAttribute - Type.

```
AbstractVariableAttribute
```

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of variables in the model.

```
source
```

MathOptInterface.VariableName - Type.

VariableName()

An AbstractVariableAttribute for a String identifying the variable.

The default name is "" if not set by the user.

Duplicate names

Two variables may have the same name; however, variables with duplicate names cannot be looked up using get.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
julia> x = MOI.add_variable(model);
julia> MOI.supports(model, MOI.VariableName(), MOI.VariableIndex)
true
julia> MOI.get(model, MOI.VariableName(), x)
""
julia> MOI.set(model, MOI.VariableName(), x, "x")
julia> MOI.get(model, MOI.VariableName(), x)
```

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.VariableName, ::MOI.VariableIndex)::String
MOI.set(::Optimizer, ::MOI.VariableName, ::MOI.VariableIndex, ::String)::Nothing
MOI.supports(::Optimizer, ::MOI.VariableName, ::Type{MOI.VariableIndex})::Bool
```

source

MathOptInterface.VariablePrimalStart - Type.

VariablePrimalStart()

An AbstractVariableAttribute for the initial assignment to the variable's primal value that the optimizer may use to warm-start the solve.

May be a number or nothing (unset).

Example

```
julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}());
julia> x = MOI.add_variable(model);
julia> MOI.supports(model, MOI.VariablePrimalStart(), MOI.VariableIndex)
true
julia> MOI.get(model, MOI.VariablePrimalStart(), x)
julia> MOI.set(model, MOI.VariablePrimalStart(), x, 1.0)
julia> MOI.get(model, MOI.VariablePrimalStart(), x)
1.0
```

Implementation

Optimizers should implement the following methods:

source

MathOptInterface.VariablePrimal - Type.

VariablePrimal(result_index::Int = 1)

An AbstractVariableAttribute for the variable's primal value in result result_index.

PrimalStatus

Before quering this attribute you should first check PrimalStatus to confirm that a primal solution is avaiable.

If the PrimalStatus is NO SOLUTION the result of querying this attribute is undefined.

result_index

The optimizer may return multiple primal solutions. See ResultCount for information on how the results are ordered.

If the solver does not have a primal value for the variable because the result_index is beyond the available solutions (whose number is indicated by the ResultCount attribute), getting this attribute must throw a ResultIndexBoundsError.

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.VariablePrimal, ::MOI.VariableIndex)::T
```

They should not implement set or supports.

source

MathOptInterface.VariableBasisStatus - Type.

```
VariableBasisStatus(result index::Int = 1)
```

An AbstractVariableAttribute for the BasisStatusCode of the variable in result result_index, with respect to a basic solution.

PrimalStatus

Before quering this attribute you should first check PrimalStatus to confirm that a primal solution is avaiable.

If the PrimalStatus is NO_SOLUTION the result of querying this attribute is undefined.

result_index

The optimizer may return multiple primal solutions. See ResultCount for information on how the results are ordered.

If the solver does not have a primal value for the variable because the result_index is beyond the available solutions (whose number is indicated by the ResultCount attribute), getting this attribute must throw a ResultIndexBoundsError.

Implementation

Optimizers should implement the following methods:

MOI.get(::Optimizer, ::MOI.VariableBasisStatus, ::MOI.VariableIndex)::MOI.BasisStatusCode

They should not implement set or supports.

source

36.4 Constraints

Types

MathOptInterface.ConstraintIndex - Type.

ConstraintIndex{F,S}

A type-safe wrapper for Int64 for use in referencing F-in-S constraints in a model.

The parameter F is the type of the function in the constraint, and the parameter S is the type of set in the constraint.

To allow for deletion, indices need not be consecutive.

Indices within a constraint type (that is, F-in-S) must be unique, but non-unique indices across different constraint types are allowed.

If F is VariableIndex then the index is equal to the index of the variable. That is for an index::ConstraintIndex{VariableIn we always have

```
index.value == MOI.get(model, MOI.ConstraintFunction(), index).value
```

source

Functions

MathOptInterface.is_valid - Method.

is_valid(model::ModelLike, index::Index)::Bool

Return a Bool indicating whether this index refers to a valid object in the model model.

source

MathOptInterface.add_constraint - Function.

MOI.add_constraint(map:::Map, vi::MOI.VariableIndex, set::MOI.AbstractScalarSet)

Record that a constraint vi-in-set is added and throws if a lower or upper bound is set by this constraint and such bound has already been set for vi.

source

add_constraint(model::ModelLike, func::F, set::S)::ConstraintIndex{F,S} where {F,S}

Add the constraint $f(x) \in S$ where f is defined by func, and S is defined by set.

Add the constraint $v \in S$ where v is the variable (or vector of variables) referenced by v and S is defined by set.

- An UnsupportedConstraint error is thrown if model does not support F-in-S constraints,
- a AddConstraintNotAllowed error is thrown if it supports F-in-S constraints but it cannot add the constraint in its current state and
- a ScalarFunctionConstantNotZero error may be thrown if func is an AbstractScalarFunction with nonzero constant and set is EqualTo, GreaterThan, LessThan or Interval.
- a LowerBoundAlreadySet error is thrown if F is a VariableIndex and a constraint was already added to this variable that sets a lower bound.
- a UpperBoundAlreadySet error is thrown if F is a VariableIndex and a constraint was already added to this variable that sets an upper bound.

source

MathOptInterface.add_constraints - Function.

```
add_constraints(model::ModelLike, funcs::Vector{F},

→ sets::Vector{S})::Vector{ConstraintIndex{F,S}} where {F,S}
```

Add the set of constraints specified by each function-set pair in funcs and sets. F and S should be concrete types. This call is equivalent to add_constraint.(model, funcs, sets) but may be more efficient.

source

MathOptInterface.transform - Function.

```
transform(
    model::ModelLike,
    c::ConstraintIndex{F,S1},
    newset::S2,
)::ConstraintIndex{F,S2}
```

Replace the set in constraint c with newset.

The constraint index c will no longer be valid, and the function returns a new constraint index with the correct type.

Solvers may only support a subset of constraint transforms that they perform efficiently (for example, changing from a LessThan to GreaterThan set). In addition, set modification (where S1 = S2) should be performed via the modify function.

Typically, the user should delete the constraint and add a new one.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
julia> x = MOI.add_variable(model);
julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0));
julia> print(model)
Feasibility
Subject to:
ScalarAffineFunction{Float64}-in-LessThan{Float64}
0.0 + 1.0 v[1] \le 2.0
julia> c2 = MOI.transform(model, c, MOI.GreaterThan(0.0))
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
→ MathOptInterface.GreaterThan{Float64}}(1)
julia> print(model)
Feasibility
Subject to:
ScalarAffineFunction{Float64}-in-GreaterThan{Float64}
0.0 + 1.0 \vee [1] >= 0.0
julia> MOI.is_valid(model, c)
```

```
false
```

source

MathOptInterface.supports_constraint - Function.

```
MOI.supports_constraint(
    BT::Type{<:AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet},
)::Bool</pre>
```

Return a Bool indicating whether the bridges of type BT support bridging F-in-S constraints.

Implementation notes

- This method depends only on the type of the inputs, not the runtime values.
- There is a default fallback, so you need only implement this method for constraint types that the bridge implements.

source

```
supports_constraint(
    model::ModelLike,
    ::Type{F},
```

::Type{S},
)::Bool where {F<:AbstractFunction,S<:AbstractSet}</pre>

Return a Bool indicating whether model supports F-in-S constraints, that is, copy_to(model, src) does not throw UnsupportedConstraint when src contains F-in-S constraints. If F-in-S constraints are only not supported in specific circumstances, for example, F-in-S constraints cannot be combined with another type of constraint, it should still return true.

source

Attributes

MathOptInterface.AbstractConstraintAttribute - Type.

AbstractConstraintAttribute

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of constraints in the model.

source

MathOptInterface.ConstraintName - Type.

ConstraintName()

An AbstractConstraintAttribute for a String identifying the constraint.

The default name is "" if not set by the user.

Duplicate names

Two constraints may have the same name; however, constraints with duplicate names cannot be looked up using get, regardless of whether they have the same F-in-S type.

VariableIndex connstraints

You should not implement ConstraintName for VariableIndex constraints.

Example

julia> model = MOI.Utilities.Model{Float64}(); julia> x = MOI.add_variable(model); julia> c = MOI.add_constraint(model, 1.0 * x, MOI.EqualTo(1.0)); julia> MOI.supports(model, MOI.ConstraintName(), typeof(c)) true julia> MOI.get(model, MOI.ConstraintName(), c) ""

```
julia> MOI.get(model, MOI.ConstraintName(), c)
"c"
julia> MOI.get(model, MOI.ConstraintIndex, "c")
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
→ MathOptInterface.EqualTo{Float64}}(1)
julia> F, S = MOI.ScalarAffineFunction{Float64}, MOI.EqualTo{Float64};
julia> MOI.get(model, MOI.ConstraintIndex{F,S}, "c")
```

```
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},

→ MathOptInterface.EqualTo{Float64}}(1)
```

Implementation

Optimizers should implement the following methods:

```
MOI.get(::Optimizer, ::MOI.ConstraintName, ::MOI.ConstraintIndex)::String
MOI.set(::Optimizer, ::MOI.ConstraintName, ::MOI.ConstraintIndex, ::String)::Nothing
MOI.supports(::Optimizer, ::MOI.ConstraintName, ::Type{<:MOI.ConstraintIndex})::Bool
MOI.get(::Optimizer, ::MOI.ConstraintIndex, ::MOI.ConstraintIndex,

                              ::MOI.ConstraintIndex, ::MOI.ConstraintIndex,
```

source

MathOptInterface.ConstraintPrimalStart - Type.

ConstraintPrimalStart()

An AbstractConstraintAttribute for the initial assignment to the constraint's ConstraintPrimal that the optimizer may use to warm-start the solve.

May be nothing (unset), a number for AbstractScalarFunction, or a vector for AbstractVectorFunction.

source

MathOptInterface.ConstraintDualStart - Type.

```
ConstraintDualStart()
```

An AbstractConstraintAttribute for the initial assignment to the constraint's ConstraintDual that the optimizer may use to warm-start the solve.

May be nothing (unset), a number for AbstractScalarFunction, or a vector for AbstractVectorFunction.

source

MathOptInterface.ConstraintPrimal - Type.

ConstraintPrimal(result_index::Int = 1)

An AbstractConstraintAttribute for the constraint's primal value in result result_index.

Definition

If the constraint is $f(x) \in S$, then in most cases the ConstraintPrimal is the value of f, evaluated at the corresponding VariablePrimal solution.

However, some conic solvers reformulate $b - Ax \in S$ to s = b - Ax and $s \in S$. These solvers may return the value of s for ConstraintPrimal, rather than b - Ax. (Although these are constrained by an equality constraint, due to numerical tolerances they may not be identical.)

PrimalStatus

Before quering this attribute you should first check PrimalStatus to confirm that a primal solution is avaiable.

If the PrimalStatus is NO_SOLUTION the result of querying this attribute is undefined.

result_index

The optimizer may return multiple primal solutions. See ResultCount for information on how the results are ordered.

If the solver does not have a primal value for the constraint because the result_index is beyond the available solutions (whose number is indicated by the ResultCount attribute), getting this attribute must throw a ResultIndexBoundsError.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ConstraintPrimal,
    ::MOI.ConstraintIndex{<:MOI.AbstractScalarFunction}
)::T
MOI.get(
    ::Optimizer,
    ::MOI.ConstraintPrimal,
    ::MOI.ConstraintIndex{<:MOI.AbstractVectorFunction}
)::Vector{T}</pre>
```

They should not implement set or supports.

source

MathOptInterface.ConstraintDual - Type.

ConstraintDual(result_index::Int = 1)

An AbstractConstraintAttribute for the constraint's dual value in result result_index.

DualStatus

Before quering this attribute you should first check DualStatus to confirm that a dual solution is avaiable.

If the DualStatus is NO_SOLUTION the result of querying this attribute is undefined.

result_index

The optimizer may return multiple dual solutions. See ResultCount for information on how the results are ordered.

If the solver does not have a dual value for the constraint because the result_index is beyond the available solutions (whose number is indicated by the ResultCount attribute), getting this attribute must throw a ResultIndexBoundsError.

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ConstraintDual,
    ::MOI.ConstraintIndex{<:MOI.AbstractScalarFunction}
)::T
MOI.get(
    ::Optimizer,
    ::MOI.ConstraintDual,
    ::MOI.ConstraintIndex{<:MOI.AbstractVectorFunction}
)::Vector{T}</pre>
```

They should not implement set or supports.

source

MathOptInterface.ConstraintBasisStatus - Type.

```
ConstraintBasisStatus(result_index::Int = 1)
```

An AbstractConstraintAttribute for the BasisStatusCode of the constraint in result result_index, with respect to a basic solution.

If result_index is omitted, it is 1 by default.

If the solver does not have a basis status for the constraint because the result_index is beyond the available solutions (whose number is indicated by the ResultCount attribute), getting this attribute must throw a ResultIndexBoundsError. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check PrimalStatus before accessing the ConstraintBasisStatus attribute.

See ResultCount for information on how the results are ordered.

Notes

For the basis status of a variable, query VariableBasisStatus.

ConstraintBasisStatus does not apply to VariableIndex constraints. You can infer the basis status of a VariableIndex constraint by looking at the result of VariableBasisStatus.

source

MathOptInterface.ConstraintFunction - Type.

ConstraintFunction()

An AbstractConstraintAttribute for the AbstractFunction object used to define the constraint.

It is guaranteed to be equivalent but not necessarily identical to the function provided by the user.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
julia> x = MOI.add_variable(model);
julia> c = MOI.add_constraint(model, 1.0 * x, MOI.GreaterThan(0.0));
julia> MOI.get(model, MOI.ConstraintFunction(), c)
0.0 + 1.0 MOI.VariableIndex(1)
julia> MOI.set(model, MOI.ConstraintFunction(), c, 2.0 * x)
julia> MOI.get(model, MOI.ConstraintFunction(), c)
0.0 + 2.0 MOI.VariableIndex(1)
```

Implementation

Optimizers should implement the following methods:

MOI.get(
 ::Optimizer,
 ::MOI.ConstraintFunction,
 ::MOI.ConstraintIndex{F,S},
)::F where {F,S}

If the optimizer supports modifying an existing function, it should implement:

```
MOI.set(
    ::Optimizer,
    ::MOI.ConstraintFunction,
    ::MOI.ConstraintIndex{F,S},
    ::F,
)::Nothing where {F,S}
```

It should not implement supports.

source

MathOptInterface.CanonicalConstraintFunction - Type.

```
CanonicalConstraintFunction()
```

An AbstractConstraintAttribute for a canonical representation of the AbstractFunction object used to define the constraint.

Getting this attribute is guaranteed to return a function that is equivalent but not necessarily identical to the function provided by the user.

Fallback

By default, MOI.get(model, MOI.CanonicalConstraintFunction(), ci) falls back to MOI.Utilities.canonical(MOI.get MOI.ConstraintFunction(), ci)).

However, if model knows that the constraint function is canonical then it can implement a specialized method that directly return the function without calling Utilities.canonical. Therefore, the value returned **cannot** be assumed to be a copy of the function stored in model.

Moreover, Utilities.Model checks with Utilities.is_canonical whether the function stored internally is already canonical and if it's the case, then it returns the function stored internally instead of a copy.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variable(model);

julia> c = MOI.add_constraint(model, 1.0 * x + 1.0 * x, MOI.GreaterThan(0.0));

```
julia> MOI.get(model, MOI.CanonicalConstraintFunction(), c)
0.0 + 2.0 MOI.VariableIndex(1)
```

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.CanonicalConstraintFunction,
    ::MOI.ConstraintIndex{F,S},
)::F where {F,S}
```

They should not implement set or supports.

source

MathOptInterface.ConstraintSet - Type.

ConstraintSet()

An AbstractConstraintAttribute for the AbstractSet object used to define the constraint.

Example

julia> model = MOI.Utilities.Model{Float64}();

```
julia> x = MOI.add_variable(model);
```

```
julia> c = MOI.add_constraint(model, x, MOI.GreaterThan(0.0));
julia> MOI.get(model, MOI.ConstraintSet(), c)
MathOptInterface.GreaterThan{Float64}(0.0)
julia> MOI.set(model, MOI.ConstraintSet(), c, MOI.GreaterThan(1.0))
julia> MOI.get(model, MOI.ConstraintSet(), c)
MathOptInterface.GreaterThan{Float64}(1.0)
```

Implementation

Optimizers should implement the following methods:

```
MOI.get(
    ::Optimizer,
    ::MOI.ConstraintSet,
    ::MOI.ConstraintIndex{F,S},
)::S where {F,S}
```

If the optimizer supports modifying an existing set, it should implement:

```
MOI.set(
    ::Optimizer,
    ::MOI.ConstraintSet,
    ::MOI.ConstraintIndex{F,S},
    ::S,
)::Nothing where {F,S}
```

It should not implement supports.

source

MathOptInterface.BasisStatusCode - Type.

BasisStatusCode

An Enum for the value of the ConstraintBasisStatus and VariableBasisStatus attributes, explaining the status of a given element with respect to an optimal solution basis.

Notes

When queried as part of ConstraintBasisStatus, NONBASIC_AT_LOWER and NONBASIC_AT_UPPER should be returned only for constraints with the Interval set. In this case, they are necessary to distinguish which side of the constraint is active. One-sided constraints (for example, LessThan and GreaterThan) should use NONBASIC instead of the NONBASIC_AT_* values.

This restriction does not apply to VariableBasisStatus, which should return NONBASIC_AT_* regardless of whether the alternative bound exists.

Values

BASIC

The element is in the basis.

NONBASIC

The element is not in the basis.

NONBASIC_AT_LOWER

The element is not in the basis and is at its lower bound.

NONBASIC_AT_UPPER

The element is not in the basis and is at its upper bound.

SUPER_BASIC

The element is not in the basis but is also not at one of its bounds.

In a linear program, this status occurs when a variable with no bounds is not in the basis, for example, because it takes the value 0.0.

source

MathOptInterface.BASIC - Constant.

BASIC::BasisStatusCode

An instance of the BasisStatusCode enum.

About

The element is in the basis.

source

MathOptInterface.NONBASIC - Constant.

NONBASIC::BasisStatusCode

An instance of the BasisStatusCode enum.

About

The element is not in the basis.

source

MathOptInterface.NONBASIC_AT_LOWER - Constant.

NONBASIC_AT_LOWER::BasisStatusCode

An instance of the BasisStatusCode enum.

About

The element is not in the basis and is at its lower bound.

source

MathOptInterface.NONBASIC_AT_UPPER - Constant.

NONBASIC_AT_UPPER::BasisStatusCode

An instance of the BasisStatusCode enum.

About

The element is not in the basis and is at its upper bound.

source

MathOptInterface.SUPER_BASIC - Constant.

SUPER BASIC::BasisStatusCode

An instance of the BasisStatusCode enum.

About

The element is not in the basis but is also not at one of its bounds.

In a linear program, this status occurs when a variable with no bounds is not in the basis, for example, because it takes the value 0.0.

source

36.5 Modifications

MathOptInterface.modify - Function.

```
modify(
    model::ModelLike,
    ci::ConstraintIndex,
    change::AbstractFunctionModification,
)
```

Apply the modification specified by change to the function of constraint ci.

An ModifyConstraintNotAllowed error is thrown if modifying constraints is not supported by the model model.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
julia> x = MOI.add_variable(model);
julia> ci = MOI.add_constraint(model, 1.0 * x, MOI.EqualTo(1.0));
julia> MOI.modify(model, ci, MOI.ScalarConstantChange(10.0))
```

```
julia> print(model)
```

Feasibility

Subject to:

```
ScalarAffineFunction{Float64}-in-EqualTo{Float64} 10.0 + 1.0 v[1] == 1.0
```

source

```
modify(
    model::ModelLike,
    cis::AbstractVector{<:ConstraintIndex},
    changes::AbstractVector{<:AbstractFunctionModification},
)</pre>
```

Apply multiple modifications specified by changes to the functions of constraints cis.

A ModifyConstraintNotAllowed error is thrown if modifying constraints is not supported by model.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variables(model, 2);

julia> ci = MOI.add_constraint.(model, 1.0 .* x, MOI.EqualTo(1.0));

julia> MOI.modify(model, ci, MOI.ScalarCoefficientChange.(x, [2.0, 0.5]))

```
julia> print(model)
Feasibility
```

Subject to:

```
ScalarAffineFunction{Float64}-in-EqualTo{Float64}
0.0 + 2.0 v[1] == 1.0
0.0 + 0.5 v[2] == 1.0
```

source

modify(model::ModelLike, ::ObjectiveFunction, change::AbstractFunctionModification)

Apply the modification specified by change to the objective function of model. To change the function completely, call set instead.

An ModifyObjectiveNotAllowed error is thrown if modifying objectives is not supported by the model model.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variable(model);

julia> MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)

julia> f = 1.0 * x;

```
julia> attr = MOI.ObjectiveFunction{typeof(f)}()
MathOptInterface.ObjectiveFunction{MathOptInterface.ScalarAffineFunction{Float64}}()
```

```
julia> MOI.set(model, attr, f)
```

julia> MOI.modify(model, attr, MOI.ScalarConstantChange(10.0))

```
julia> print(model)
Minimize ScalarAffineFunction{Float64}:
10.0 + 1.0 v[1]
```

Subject to:

source

```
modify(
    model::ModelLike,
    attr::ObjectiveFunction,
    changes::AbstractVector{<:AbstractFunctionModification},
)</pre>
```

Apply multiple modifications specified by changes to the functions of constraints cis.

A ModifyObjectiveNotAllowed error is thrown if modifying objective coefficients is not supported by model.

Example

Example

julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variables(model, 2);

julia> MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)

julia> f = 1.0 * x[1] + 1.0 * x[2];

```
julia> attr = MOI.ObjectiveFunction{typeof(f)}()
MathOptInterface.ObjectiveFunction{MathOptInterface.ScalarAffineFunction{Float64}}()
julia> MOI.set(model, attr, f)
julia> MOI.modify(model, attr, MOI.ScalarCoefficientChange.(x, [2.0, 0.5]))
julia> print(model)
```

```
Minimize ScalarAffineFunction{Float64}:
0.0 + 2.0 v[1] + 0.5 v[2]
```

Subject to:

source

MathOptInterface.AbstractFunctionModification - Type.

AbstractFunctionModification

An abstract supertype for structs which specify partial modifications to functions, to be used for making small modifications instead of replacing the functions entirely.

source

MathOptInterface.ScalarConstantChange - Type.

ScalarConstantChange{T}(new_constant::T)

A struct used to request a change in the constant term of a scalar-valued function.

Applicable to ScalarAffineFunction and ScalarQuadraticFunction.

source

MathOptInterface.VectorConstantChange - Type.

VectorConstantChange{T}(new_constant::Vector{T})

A struct used to request a change in the constant vector of a vector-valued function.

Applicable to VectorAffineFunction and VectorQuadraticFunction.

source

MathOptInterface.ScalarCoefficientChange - Type.

ScalarCoefficientChange{T}(variable::VariableIndex, new_coefficient::T)

A struct used to request a change in the linear coefficient of a single variable in a scalar-valued function.

Applicable to ScalarAffineFunction and ScalarQuadraticFunction.

source

MathOptInterface.ScalarQuadraticCoefficientChange - Type.

```
ScalarQuadraticCoefficientChange{T}(
    variable_1::VariableIndex,
    variable_2::VariableIndex,
    new_coefficient::T,
)
```

A struct used to request a change in the quadratic coefficient of a ScalarQuadraticFunction.

Scaling factors

A ScalarQuadraticFunction has an implicit 0.5 scaling factor in front of the Q matrix. This modification applies to terms in the Q matrix.

If variable_1 == variable_2, this modification sets the corresponding diagonal element of the Q matrix to new_coefficient.

If variable_1 != variable_2, this modification is equivalent to setting both the corresponding upperand lower-triangular elements of the Q matrix to new_coefficient.

As a consequence:

- to modify the term x² to become 2x², new_coefficient must be 4
- to modify the term xy to become 2xy, new_coefficient must be 2

source

MathOptInterface.MultirowChange - Type.

```
MultirowChange{T}(
    variable::VariableIndex,
    new_coefficients::Vector{Tuple{Int64,T}},
) where {T}
```

A struct used to request a change in the linear coefficients of a single variable in a vector-valued function.

New coefficients are specified by (output_index, coefficient) tuples.

Applicable to VectorAffineFunction and VectorQuadraticFunction.

source

36.6 Nonlinear programming

Types

MathOptInterface.AbstractNLPEvaluator - Type.

AbstractNLPEvaluator

Abstract supertype for the callback object that is used to query function values, derivatives, and expression graphs.

It is used in NLPBlockData.

Example

This example uses the Test. HS071 evaluator.

julia> evaluator = MOI.Test.HS071(true);

```
julia> supertype(typeof(evaluator))
MathOptInterface.AbstractNLPEvaluator
```

source

MathOptInterface.NLPBoundsPair - Type.

NLPBoundsPair(lower::Float64, upper::Float64)

A struct holding a pair of lower and upper bounds.

- Inf and Inf can be used to indicate no lower or upper bound, respectively.

Example

```
julia> bounds = MOI.NLPBoundsPair.([25.0, 40.0], [Inf, 40.0])
2-element Vector{MathOptInterface.NLPBoundsPair}:
MathOptInterface.NLPBoundsPair(25.0, Inf)
MathOptInterface.NLPBoundsPair(40.0, 40.0)
```

source

MathOptInterface.NLPBlockData - Type.

```
struct NLPBlockData
    constraint_bounds::Vector{NLPBoundsPair}
    evaluator::AbstractNLPEvaluator
    has_objective::Bool
end
```

A struct encoding a set of nonlinear constraints of the form $lb \leq g(x) \leq ub$ and, if has_objective == true, a nonlinear objective function f(x).

Nonlinear objectives override any objective set by using the ObjectiveFunction attribute.

The evaluator is a callback object that is used to query function values, derivatives, and expression graphs. If has_objective == false, then it is an error to query properties of the objective function, and in Hessian-of-the-Lagrangian queries, σ must be set to zero.

Note

Throughout the evaluator, all variables are ordered according to ListOfVariableIndices. Hence, MOI copies of nonlinear problems must not re-order variables.
Example

This example uses the Test.HS071 evaluator.

```
julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}());
```

julia> MOI.set(model, MOI.NLPBlock(), block)

source

Attributes

MathOptInterface.NLPBlock - Type.

NLPBlock()

An AbstractModelAttribute that stores an NLPBlockData, representing a set of nonlinear constraints, and optionally a nonlinear objective.

Example

This example uses the Test. HS071 evaluator.

```
julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}());
```

julia> MOI.set(model, MOI.NLPBlock(), block)

source

MathOptInterface.NLPBlockDual - Type.

NLPBlockDual(result_index::Int = 1)

An AbstractModelAttribute for the Lagrange multipliers on the constraints from the NLPBlock in result result_index.

If result_index is omitted, it is 1 by default.

Example

```
julia> MOI.NLPBlockDual()
MathOptInterface.NLPBlockDual(1)
```

julia> MOI.NLPBlockDual(2)
MathOptInterface.NLPBlockDual(2)

source

MathOptInterface.NLPBlockDualStart - Type.

NLPBlockDualStart()

An AbstractModelAttribute for the initial assignment of the Lagrange multipliers on the constraints from the NLPBlock that the solver may use to warm-start the solve.

Example

This example uses the Test.HS071 evaluator.

source

Functions

MathOptInterface.initialize - Function.

```
initialize(
    d::AbstractNLPEvaluator,
    requested_features::Vector{Symbol},
)::Nothing
```

Initialize d with the set of features in requested_features. Check features_available before calling initialize to see what features are supported by d.

Warning

This method must be called before any other methods.

Features

The following features are defined:

- :Grad: enables eval_objective_gradient
- :Jac: enables eval_constraint_jacobian and eval_constraint_gradient
- : JacVec: enables eval_constraint_jacobian_product and eval_constraint_jacobian_transpose_product
- :Hess: enables eval_hessian_lagrangian
- :HessVec: enables eval_hessian_lagrangian_product
- :ExprGraph: enables objective_expr and constraint_expr.

In all cases, including when requested_features is empty, eval_objective and eval_constraint are supported.

Example

This example uses the Test.HS071 evaluator.

julia> evaluator = MOI.Test.HS071(true);

julia> MOI.initialize(evaluator, [:Grad, :Jac])

source

MathOptInterface.features_available - Function.

features_available(d::AbstractNLPEvaluator)::Vector{Symbol}

Returns the subset of features available for this problem instance.

See initialize for the list of defined features.

Example

This example uses the Test.HS071 evaluator.

julia> evaluator = MOI.Test.HS071(true, true);

julia> MOI.features_available(evaluator)
6-element Vector{Symbol}:
 :Grad
 :Jac
 :JacVec
 :ExprGraph
 :Hess
 :HessVec

source

MathOptInterface.eval_objective - Function.

```
eval_objective(d::AbstractNLPEvaluator, x::AbstractVector{T})::T where {T}
```

Evaluate the objective f(x), returning a scalar value.

Initialize

Before calling this function, you must call initialize, but you do not need to pass a value.

Example

This example uses the Test. HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
julia> MOI.initialize(evaluator, Symbol[])
julia> MOI.eval_objective(evaluator, [1.0, 2.0, 3.0, 4.0])
27.0
```

source

MathOptInterface.eval_constraint - Function.

```
eval_constraint(
    d::AbstractNLPEvaluator,
    g::AbstractVector{T},
    x::AbstractVector{T},
)::Nothing where {T}
```

Given a set of vector-valued constraints $l \leq g(x) \leq u$, evaluate the constraint function g(x), storing the result in the vector g.

Initialize

Before calling this function, you must call initialize, but you do not need to pass a value.

Implementation notes

When implementing this method, you must not assume that g is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

Example

This example uses the Test.HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
julia> MOI.initialize(evaluator, Symbol[])
julia> g = fill(NaN, 2);
julia> MOI.eval_constraint(evaluator, g, [1.0, 2.0, 3.0, 4.0])
julia> g
```

```
2-element Vector{Float64}:
24.0
30.0
```

source

MathOptInterface.eval_objective_gradient - Function.

```
eval_objective_gradient(
    d::AbstractNLPEvaluator,
    grad::AbstractVector{T},
    x::AbstractVector{T},
)::Nothing where {T}
```

Evaluate the gradient of the objective function $grad = \nabla f(x)$ as a dense vector, storing the result in the vector grad.

Initialize

Before calling this function, you must call initialize with :Grad.

Implementation notes

When implementing this method, you must not assume that grad is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

Example

This example uses the Test.HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
```

julia> MOI.initialize(evaluator, Symbol[:Grad])

julia> grad = fill(NaN, 4);

julia> MOI.eval_objective_gradient(evaluator, grad, [1.0, 2.0, 3.0, 4.0])

julia> grad

```
4-element Vector{Float64}:
28.0
4.0
5.0
6.0
```

source

MathOptInterface.jacobian_structure - Function.

jacobian_structure(d::AbstractNLPEvaluator)::Vector{Tuple{Int64,Int64}}

Returns a vector of tuples, (row, column), where each indicates the position of a structurally nonzero $\nabla g_1(x)$

element in the Jacobian matrix: $J_g(x) = \begin{bmatrix} \nabla g_2(x) \\ \vdots \\ \nabla g_m(x) \end{bmatrix}$, where g_i is the ith component of the nonlinear

constraints g(x).

The indices are not required to be sorted and can contain duplicates, in which case the solver should combine the corresponding elements by adding them together.

The sparsity structure is assumed to be independent of the point x.

Initialize

Before calling this function, you must call initialize with : Jac.

Example

This example uses the Test. HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
```

julia> MOI.initialize(evaluator, Symbol[:Jac])

```
julia> MOI.jacobian_structure(evaluator)
```

```
8-element Vector{Tuple{Int64, Int64}}:
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
```

source

MathOptInterface.eval_constraint_gradient - Function.

```
eval_constraint_gradient(
   d::AbstractNLPEvaluator,
   ∇g::AbstractVector{T},
   x::AbstractVector{T},
   i::Int,
)::Nothing where {T}
```

Evaluate the gradient of constraint i, $\nabla q_i(x)$, and store the non-zero values in ∇g , corresponding to the structure returned by constraint_gradient_structure.

Implementation notes

When implementing this method, you must not assume that ∇q is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

Initialize

Before calling this function, you must call initialize with : Jac.

Example

This example uses the Test. HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
```

julia> MOI.initialize(evaluator, Symbol[:Jac])

julia> indices = MOI.constraint_gradient_structure(evaluator, 1);

```
julia> Vg = zeros(length(indices));
```

julia> MOI.eval_constraint_gradient(evaluator, ∇g, [1.0, 2.0, 3.0, 4.0], 1)

julia> ⊽g

```
4-element Vector{Float64}:
24.0
12.0
8.0
6.0
```

source

MathOptInterface.constraint_gradient_structure - Function.

constraint_gradient_structure(d::AbstractNLPEvaluator, i::Int)::Vector{Int64}

Returns a vector of indices, where each element indicates the position of a structurally nonzero element in the gradient of constraint $\nabla g_i(x)$.

The indices are not required to be sorted and can contain duplicates, in which case the solver should combine the corresponding elements by adding them together.

The sparsity structure is assumed to be independent of the point x.

Initialize

Before calling this function, you must call initialize with : Jac.

Example

This example uses the Test.HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
julia> MOI.initialize(evaluator, Symbol[:Jac])
julia> indices = MOI.constraint_gradient_structure(evaluator, 1)
4-element Vector{Int64}:
1
2
3
4
```

source

MathOptInterface.eval_constraint_jacobian - Function.

```
eval constraint jacobian(
   d::AbstractNLPEvaluator,
   J::AbstractVector{T},
   x::AbstractVector{T},
)::Nothing where {T}
```

 $\left[\nabla g_1(x) \right]$ Evaluates the sparse Jacobian matrix $J_q(x) =$

$$\begin{bmatrix} \nabla g_2(x) \\ \vdots \\ \nabla g_m(x) \end{bmatrix}.$$

The result is stored in the vector J in the same order as the indices returned by jacobian_structure.

Implementation notes

When implementing this method, you must not assume that J is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

Initialize

Before calling this function, you must call initialize with : Hess.

Example

This example uses the Test.HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
```

julia> MOI.initialize(evaluator, Symbol[:Jac])

julia> J_indices = MOI.jacobian_structure(evaluator);

julia> J = zeros(length(J_indices));

julia> MOI.eval_constraint_jacobian(evaluator, J, [1.0, 2.0, 3.0, 4.0])

julia> J

8-element Vector{Float64}: 24.0 12.0 8.0 6.0 2.0 4.0 6.0 8.0

source

MathOptInterface.eval_constraint_jacobian_product - Function.

```
eval_constraint_jacobian_product(
    d::AbstractNLPEvaluator,
    y::AbstractVector{T},
    x::AbstractVector{T},
    w::AbstractVector{T},
)::Nothing where {T}
```

Computes the Jacobian-vector product $\boldsymbol{y}=J_g(\boldsymbol{x})\boldsymbol{w}$, storing the result in the vector y.

The vectors have dimensions such that length(w) = length(x), and length(y) is the number of nonlinear constraints.

Implementation notes

When implementing this method, you must not assume that y is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

Initialize

Before calling this function, you must call initialize with :JacVec.

Example

This example uses the Test.HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
julia> MOI.initialize(evaluator, Symbol[:Jac, :JacVec])
julia> y = zeros(2);
julia> x = [1.0, 2.0, 3.0, 4.0];
julia> w = [1.5, 2.5, 3.5, 4.5];
julia> MOI.eval_constraint_jacobian_product(evaluator, y, x, w)
julia> y
2-element Vector{Float64}:
121.0
70.0
```

source

MathOptInterface.eval_constraint_jacobian_transpose_product - Function.

```
eval_constraint_jacobian_transpose_product(
    d::AbstractNLPEvaluator,
    y::AbstractVector{T},
    x::AbstractVector{T},
    w::AbstractVector{T},
)::Nothing where {T}
```

Computes the Jacobian-transpose-vector product $y = J_g(x)^T w$, storing the result in the vector y.

The vectors have dimensions such that length(y) == length(x), and length(w) is the number of nonlinear constraints.

Implementation notes

When implementing this method, you must not assume that y is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

Initialize

Before calling this function, you must call initialize with : JacVec.

Example

This example uses the Test.HS071 evaluator.

julia> evaluator = MOI.Test.HS071(true);

julia> MOI.initialize(evaluator, Symbol[:Jac, :JacVec])

julia> y = zeros(4);

julia> x = [1.0, 2.0, 3.0, 4.0];

julia> w = [1.5, 2.5];

julia> MOI.eval_constraint_jacobian_transpose_product(evaluator, y, x, w)

julia> y

```
4-element Vector{Float64}:
41.0
28.0
27.0
29.0
```

source

MathOptInterface.hessian_lagrangian_structure - Function.

```
hessian_lagrangian_structure(
    d::AbstractNLPEvaluator,
)::Vector{Tuple{Int64,Int64}}
```

Returns a vector of tuples, (row, column), where each indicates the position of a structurally nonzero element in the Hessian-of-the-Lagrangian matrix: $\nabla^2 f(x) + \sum_{i=1}^m \nabla^2 g_i(x)$.

The indices are not required to be sorted and can contain duplicates, in which case the solver should combine the corresponding elements by adding them together.

Any mix of lower and upper-triangular indices is valid. Elements (i, j) and (j, i), if both present, should be treated as duplicates.

The sparsity structure is assumed to be independent of the point x.

Initialize

Before calling this function, you must call initialize with : Hess.

Example

This example uses the Test.HS071 evaluator.

julia> evaluator = MOI.Test.HS071(true); julia> MOI.initialize(evaluator, Symbol[:Hess]) julia> MOI.hessian_lagrangian_structure(evaluator) 10-element Vector{Tuple{Int64, Int64}}: (1, 1) (2, 1) (2, 2) (3, 1) (3, 2) (3, 3) (4, 1) (4, 2) (4, 3) (4, 4)

source

MathOptInterface.hessian_objective_structure - Function.

```
hessian_objective_structure(
    d::AbstractNLPEvaluator,
)::Vector{Tuple{Int64,Int64}}
```

Returns a vector of tuples, (row, column), where each indicates the position of a structurally nonzero element in the Hessian matrix: $\nabla^2 f(x)$.

The indices are not required to be sorted and can contain duplicates, in which case the solver should combine the corresponding elements by adding them together.

Any mix of lower and upper-triangular indices is valid. Elements (i, j) and (j, i), if both present, should be treated as duplicates.

The sparsity structure is assumed to be independent of the point x.

Initialize

Before calling this function, you must call initialize with : Hess.

Example

This example uses the Test. HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
julia> MOI.initialize(evaluator, Symbol[:Hess])
julia> MOI.hessian_objective_structure(evaluator)
6-element Vector{Tuple{Int64, Int64}}:
(1, 1)
(2, 1)
```

(3, 1)
(4, 1)
(4, 2)
(4, 3)

source

MathOptInterface.hessian_constraint_structure - Function.

```
hessian_constraint_structure(
    d::AbstractNLPEvaluator,
    i::Int64,
)::Vector{Tuple{Int64,Int64}}
```

Returns a vector of tuples, (row, column), where each indicates the position of a structurally nonzero element in the Hessian matrix: $\nabla^2 g_i(x)$.

The indices are not required to be sorted and can contain duplicates, in which case the solver should combine the corresponding elements by adding them together.

Any mix of lower and upper-triangular indices is valid. Elements (i, j) and (j, i), if both present, should be treated as duplicates.

The sparsity structure is assumed to be independent of the point x.

Initialize

Before calling this function, you must call initialize with : Hess.

Example

This example uses the Test. HS071 evaluator.

julia> evaluator = MOI.Test.HS071(true);

julia> MOI.initialize(evaluator, Symbol[:Hess])

julia> MOI.hessian_constraint_structure(evaluator, 1)

6-element Vector{Tuple{Int64, Int64}}:
 (2, 1)
 (3, 1)
 (3, 2)
 (4, 1)
 (4, 2)
 (4, 3)

iulies MOI because constraint structure

julia> MOI.hessian_constraint_structure(evaluator, 2)

4-element Vector{Tuple{Int64, Int64}}:

(1, 1)

(2, 2)

(3, 3) (4, 4)

source

MathOptInterface.eval_hessian_objective - Function.

```
eval_hessian_objective(
    d::AbstractNLPEvaluator,
    H::AbstractVector{T},
    x::AbstractVector{T},
)::Nothing where {T}
```

This function computes the sparse Hessian matrix: $\nabla^2 f(x)$, storing the result in the vector H in the same order as the indices returned by hessian_objective_structure.

Implementation notes

When implementing this method, you must not assume that H is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

Initialize

Before calling this function, you must call initialize with : Hess.

Example

This example uses the Test. HS071 evaluator.

julia> evaluator = MOI.Test.HS071(true, true);

julia> MOI.initialize(evaluator, Symbol[:Hess])

julia> indices = MOI.hessian_objective_structure(evaluator);

julia> H = zeros(length(indices));

```
julia> x = [1.0, 2.0, 3.0, 4.0];
```

julia> MOI.eval_hessian_objective(evaluator, H, x)

julia> H

```
6-element Vector{Float64}:
8.0
4.0
4.0
7.0
1.0
1.0
```

source

MathOptInterface.eval_hessian_constraint - Function.

```
eval_hessian_constraint(
    d::AbstractNLPEvaluator,
    H::AbstractVector{T},
    x::AbstractVector{T},
    i::Int64,
)::Nothing where {T}
```

This function computes the sparse Hessian matrix: $\nabla^2 g_i(x)$, storing the result in the vector H in the same order as the indices returned by hessian_constraint_structure.

Implementation notes

When implementing this method, you must not assume that H is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

Initialize

Before calling this function, you must call initialize with : Hess.

Example

This example uses the Test. HS071 evaluator.

julia> evaluator = MOI.Test.HS071(true, true);

julia> MOI.initialize(evaluator, Symbol[:Hess])

julia> indices = MOI.hessian_constraint_structure(evaluator, 1);

julia> H = zeros(length(indices));

```
julia> x = [1.0, 2.0, 3.0, 4.0];
```

julia> MOI.eval_hessian_constraint(evaluator, H, x, 1)

julia> H

```
6-element Vector{Float64}:
12.0
8.0
4.0
6.0
3.0
2.0
```

source

MathOptInterface.eval_hessian_lagrangian - Function.

```
eval_hessian_lagrangian(
    d::AbstractNLPEvaluator,
    H::AbstractVector{T},
    x::AbstractVector{T},
    o::T,
    µ::AbstractVector{T},
)::Nothing where {T}
```

Given scalar weight σ and vector of constraint weights μ , this function computes the sparse Hessian-of-the-Lagrangian matrix: $\sigma \nabla^2 f(x) + \sum_{i=1}^m \mu_i \nabla^2 g_i(x)$, storing the result in the vector H in the same order as the indices returned by hessian_lagrangian_structure.

Implementation notes

When implementing this method, you must not assume that H is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

Initialize

Before calling this function, you must call initialize with : Hess.

Example

This example uses the Test. HS071 evaluator.

julia> evaluator = MOI.Test.HS071(true);

julia> MOI.initialize(evaluator, Symbol[:Hess])

julia> indices = MOI.hessian_lagrangian_structure(evaluator);

julia> H = zeros(length(indices));

julia> x = [1.0, 2.0, 3.0, 4.0];

julia> $\sigma = 1.0$;

julia> $\mu = [1.0, 1.0];$

<code>julia> MOI.eval_hessian_lagrangian(evaluator, H, x, σ , μ)</code>

julia> H

```
10-element Vector{Float64}:
10.0
16.0
2.0
12.0
4.0
2.0
13.0
4.0
3.0
2.0
```

source

MathOptInterface.eval_hessian_lagrangian_product - Function.

```
eval_hessian_lagrangian_product(
    d::AbstractNLPEvaluator,
    h::AbstractVector{T},
    x::AbstractVector{T},
    v::AbstractVector{T},
    o::T,
    µ::AbstractVector{T},
)::Nothing where {T}
```

Given scalar weight σ and vector of constraint weights μ , computes the Hessian-of-the-Lagrangian-vector product $h = \left(\sigma \nabla^2 f(x) + \sum_{i=1}^m \mu_i \nabla^2 g_i(x)\right) v$, storing the result in the vector h.

The vectors have dimensions such that length(h) == length(x) == length(v).

Implementation notes

When implementing this method, you must not assume that h is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

Initialize

Before calling this function, you must call initialize with :HessVec.

Example

This example uses the Test.HS071 evaluator.

julia> evaluator = MOI.Test.HS071(true, true);

julia> MOI.initialize(evaluator, Symbol[:HessVec])

julia> H = fill(NaN, 4);

julia> x = [1.0, 2.0, 3.0, 4.0];

julia> v = [1.5, 2.5, 3.5, 4.5];

julia> $\sigma = 1.0$;

julia> $\mu = [1.0, 1.0];$

<code>julia> MOI.eval_hessian_lagrangian_product(evaluator, H, x, v, σ , μ)</code>

julia> H

```
4-element Vector{Float64}:
155.5
61.0
48.5
49.0
```

source

MathOptInterface.objective_expr - Function.

objective_expr(d::AbstractNLPEvaluator)::Expr

Returns a Julia Expr object representing the expression graph of the objective function.

Format

The expression has a number of limitations, compared with arbitrary Julia expressions:

- All sums and products are flattened out as simple Expr(:+, ...) and Expr(:*, ...) objects.
- All decision variables must be of the form Expr(:ref, :x, MOI.VariableIndex(i)), where i is the *i*th variable in ListOfVariableIndices.
- There are currently no restrictions on recognized functions; typically these will be built-in Julia functions like ^, exp, log, cos, tan, sqrt, etc., but modeling interfaces may choose to extend these basic functions, or error if they encounter unsupported functions.

Initialize

Before calling this function, you must call initialize with : ExprGraph.

Example

This example uses the Test.HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
```

julia> MOI.initialize(evaluator, [:ExprGraph])

julia> MOI.objective_expr(evaluator)

```
:(x[MOI.VariableIndex(1)] * x[MOI.VariableIndex(4)] * (x[MOI.VariableIndex(1)] +

→ x[MOI.VariableIndex(2)] + x[MOI.VariableIndex(3)]) + x[MOI.VariableIndex(3)])
```

source

MathOptInterface.constraint_expr - Function.

constraint_expr(d::AbstractNLPEvaluator, i::Integer)::Expr

Returns a Julia Expr object representing the expression graph for the *i*th nonlinear constraint.

Format

The format is the same as objective_expr, with an additional comparison operator indicating the sense of and bounds on the constraint.

For single-sided comparisons, the body of the constraint must be on the left-hand side, and the right-hand side must be a constant.

For double-sided comparisons (that is, $l \le g(x) \le u$), the body of the constraint must be in the middle, and the left- and right-hand sides must be constants.

The bounds on the constraints must match the NLPBoundsPairs passed to NLPBlockData.

Initialize

Before calling this function, you must call initialize with : ExprGraph.

Example

This example uses the Test. HS071 evaluator.

```
julia> evaluator = MOI.Test.HS071(true);
```

julia> MOI.initialize(evaluator, [:ExprGraph])

julia> MOI.constraint_expr(evaluator, 1)

```
:(x[MOI.VariableIndex(1)] * x[MOI.VariableIndex(2)] * x[MOI.VariableIndex(3)] *
→ x[MOI.VariableIndex(4)] >= 25.0)
```

julia> MOI.constraint_expr(evaluator, 2)

```
:(x[MOI.VariableIndex(1)] ^ 2 + x[MOI.VariableIndex(2)] ^ 2 + x[MOI.VariableIndex(3)] ^ 2 + 

↔ x[MOI.VariableIndex(4)] ^ 2 == 40.0)
```

36.7 Callbacks

MathOptInterface.AbstractCallback - Type.

```
abstract type AbstractCallback <: AbstractModelAttribute end</pre>
```

Abstract type for a model attribute representing a callback function. The value set to subtypes of AbstractCallback is a function that may be called during optimize!. As optimize! is in progress, the result attributes (that is, the attributes attr such that is_set_by_optimize(attr)) may not be accessible from the callback, hence trying to get result attributes might throw a OptimizeInProgress error.

At most one callback of each type can be registered. If an optimizer already has a function for a callback type, and the user registers a new function, then the old one is replaced.

The value of the attribute should be a function taking only one argument, commonly called callback_data, that can be used for instance in LazyConstraintCallback, HeuristicCallback and UserCutCallback.

source

MathOptInterface.AbstractSubmittable - Type.

AbstractSubmittable

Abstract supertype for objects that can be submitted to the model.

source

MathOptInterface.submit - Function.

```
submit(
    optimizer::AbstractOptimizer,
    sub::AbstractSubmittable,
    values...,
)::Nothing
```

Submit values to the submittable sub of the optimizer optimizer.

An UnsupportedSubmittable error is thrown if model does not support the attribute attr (see supports) and a SubmitNotAllowed error is thrown if it supports the submittable sub but it cannot be submitted.

```
source
```

Attributes

MathOptInterface.CallbackNodeStatus - Type.

CallbackNodeStatus(callback_data)

An optimizer attribute describing the (in)feasibility of the primal solution available from CallbackVariablePrimal during a callback identified by callback_data.

Returns a CallbackNodeStatusCode Enum.

source

MathOptInterface.CallbackVariablePrimal - Type.

CallbackVariablePrimal(callback_data)

An AbstractVariableAttribute for the assignment to the variable's primal value during the callback identified by callback_data.

source

MathOptInterface.CallbackNodeStatusCode - Type.

CallbackNodeStatusCode

An Enum for the value of the CallbackNodeStatus attribute.

Values

CALLBACK_NODE_STATUS_INTEGER

The primal solution available from CallbackVariablePrimal is integer feasible.

CALLBACK_NODE_STATUS_FRACTIONAL

The primal solution available from CallbackVariablePrimal is integer infeasible.

CALLBACK NODE STATUS UNKNOWN

The status of the primal solution available from CallbackVariablePrimal is unknown.

source

MathOptInterface.CALLBACK_NODE_STATUS_INTEGER - Constant.

CALLBACK_NODE_STATUS_INTEGER::CallbackNodeStatusCode

An instance of the CallbackNodeStatusCode enum.

About

The primal solution available from CallbackVariablePrimal is integer feasible.

source

MathOptInterface.CALLBACK_NODE_STATUS_FRACTIONAL - Constant.

CALLBACK_NODE_STATUS_FRACTIONAL::CallbackNodeStatusCode

An instance of the CallbackNodeStatusCode enum.

About

The primal solution available from CallbackVariablePrimal is integer infeasible.

source

MathOptInterface.CALLBACK_NODE_STATUS_UNKNOWN - Constant.

CALLBACK NODE STATUS UNKNOWN::CallbackNodeStatusCode

An instance of the CallbackNodeStatusCode enum.

About

The status of the primal solution available from CallbackVariablePrimal is unknown.

source

Lazy constraints

MathOptInterface.LazyConstraintCallback - Type.

LazyConstraintCallback() <: AbstractCallback</pre>

The callback can be used to reduce the feasible set given the current primal solution by submitting a LazyConstraint. For instance, it may be called at an incumbent of a mixed-integer problem. Note that there is no guarantee that the callback is called at *every* feasible primal solution.

The current primal solution is accessed through CallbackVariablePrimal. Trying to access other result attributes will throw OptimizeInProgress as discussed in AbstractCallback.

Example

```
x = MOI.add_variables(optimizer, 8)
MOI.set(optimizer, MOI.LazyConstraintCallback(), callback_data -> begin
sol = MOI.get(optimizer, MOI.CallbackVariablePrimal(callback_data), x)
if # should add a lazy constraint
func = # computes function
set = # computes set
MOI.submit(optimizer, MOI.LazyConstraint(callback_data), func, set)
end
end)
```

source

MathOptInterface.LazyConstraint - Type.

LazyConstraint(callback data)

Lazy constraint func-in-set submitted as func, set. The optimal solution returned by VariablePrimal will satisfy all lazy constraints that have been submitted.

This can be submitted only from the LazyConstraintCallback. The field callback_data is a solver-specific callback type that is passed as the argument to the feasible solution callback.

Example

Suppose x and y are VariableIndexs of optimizer. To add a LazyConstraint for $2x + 3y \le 1$, write

```
func = 2.0x + 3.0y
set = MOI.LessThan(1.0)
MOI.submit(optimizer, MOI.LazyConstraint(callback_data), func, set)
```

inside a LazyConstraintCallback of data callback_data.

source

User cuts

MathOptInterface.UserCutCallback - Type.

UserCutCallback() <: AbstractCallback</pre>

The callback can be used to submit UserCut given the current primal solution. For instance, it may be called at fractional (that is, non-integer) nodes in the branch and bound tree of a mixed-integer problem. Note that there is not guarantee that the callback is called *everytime* the solver has an infeasible solution.

The infeasible solution is accessed through CallbackVariablePrimal. Trying to access other result attributes will throw OptimizeInProgress as discussed in AbstractCallback.

Example

```
x = MOI.add_variables(optimizer, 8)
MOI.set(optimizer, MOI.UserCutCallback(), callback_data -> begin
sol = MOI.get(optimizer, MOI.CallbackVariablePrimal(callback_data), x)
if # can find a user cut
func = # computes function
set = # computes set
MOI.submit(optimizer, MOI.UserCut(callback_data), func, set)
end
end
```

source

MathOptInterface.UserCut - Type.

UserCut(callback_data)

Constraint func-to-set suggested to help the solver detect the solution given by CallbackVariablePrimal as infeasible. The cut is submitted as the tuple (func, set)::Tuple{MOI.AbstractFunction,MOI.AbstractSet}.

Typically CallbackVariablePrimal will violate integrality constraints, and a cut would be of the form ScalarAffineFunction-in-LessThan or ScalarAffineFunction-in-GreaterThan.

Note that, as opposed to LazyConstraint, the provided constraint must not modify the feasible set. The constraint should be redundant, for example, it may be a consequence of affine and integrality constraints.

This can be submitted only from the UserCutCallback. The field callback_data is a solver-specific callback type that is passed as the argument to the infeasible solution callback.

Note that the solver may silently ignore the provided constraint.

source

Heuristic solutions

MathOptInterface.HeuristicCallback - Type.

```
HeuristicCallback() <: AbstractCallback</pre>
```

The callback can be used to submit HeuristicSolution given the current primal solution. For example, it may be called at fractional (that is, non-integer) nodes in the branch and bound tree of a mixed-integer problem. Note that there is no guarantee that the callback is called *every* time the solver has an infeasible solution.

The current primal solution is accessed through CallbackVariablePrimal. Trying to access other result attributes will throw OptimizeInProgress as discussed in AbstractCallback.

Example

```
x = MOI.add_variables(optimizer, 8)
MOI.set(optimizer, MOI.HeuristicCallback(), callback_data -> begin
sol = MOI.get(optimizer, MOI.CallbackVariablePrimal(callback_data), x)
if # can find a heuristic solution
values = # computes heuristic solution
MOI.submit(optimizer, MOI.HeuristicSolution(callback_data), x,
values)
end
end
```

source

MathOptInterface.HeuristicSolution - Type.

```
HeuristicSolution(callback_data)
```

Heuristically obtained feasible solution. The solution is submitted as (variables, values)::Tuple{Vector{MOI.VariableI} where values[i] gives the value of variables[i].

The submit call returns a HeuristicSolutionStatus indicating whether the provided solution was accepted or rejected.

This can be submitted only from the HeuristicCallback. The field callback_data is a solver-specific callback type that is passed as the argument to the heuristic callback.

Some solvers require a complete solution, others only partial solutions.

source

MathOptInterface.HeuristicSolutionStatus - Type.

HeuristicSolutionStatus

An Enum of possible return values for submit with HeuristicSolution.

This status informs whether the heuristic solution was accepted or rejected.

Values

HEURISTIC_SOLUTION_ACCEPTED

The heuristic solution was accepted.

HEURISTIC_SOLUTION_REJECTED

The heuristic solution was rejected.

HEURISTIC_SOLUTION_UNKNOWN

No information available on the acceptance.

source

MathOptInterface.HEURISTIC_SOLUTION_ACCEPTED - Constant.

HEURISTIC_SOLUTION_ACCEPTED::HeuristicSolutionStatus

An instance of the HeuristicSolutionStatus enum.

About

The heuristic solution was accepted.

source

MathOptInterface.HEURISTIC_SOLUTION_REJECTED - Constant.

HEURISTIC_SOLUTION_REJECTED::HeuristicSolutionStatus

An instance of the HeuristicSolutionStatus enum.

About

The heuristic solution was rejected.

source

MathOptInterface.HEURISTIC_SOLUTION_UNKNOWN - Constant.

HEURISTIC_SOLUTION_UNKNOWN::HeuristicSolutionStatus

An instance of the HeuristicSolutionStatus enum.

About

No information available on the acceptance.

source

36.8 Errors

When an MOI call fails on a model, precise errors should be thrown when possible instead of simply calling error with a message. The docstrings for the respective methods describe the errors that the implementation should throw in certain situations. This error-reporting system allows code to distinguish between internal errors (that should be shown to the user) and unsupported operations which may have automatic workarounds.

When an invalid index is used in an MOI call, an InvalidIndex is thrown:

MathOptInterface.InvalidIndex - Type.

```
struct InvalidIndex{IndexType<:Index} <: Exception
    index::IndexType
end</pre>
```

An error indicating that the index index is invalid.

source

When an invalid result index is used to retrieve an attribute, a ResultIndexBoundsError is thrown:

MathOptInterface.ResultIndexBoundsError - Type.

```
struct ResultIndexBoundsError{AttrType} <: Exception
    attr::AttrType
    result_count::Int
end</pre>
```

An error indicating that the requested attribute attr could not be retrieved, because the solver returned too few results compared to what was requested. For instance, the user tries to retrieve VariablePrimal(2) when only one solution is available, or when the model is infeasible and has no solution.

See also: check_result_index_bounds.

source

MathOptInterface.check_result_index_bounds - Function.

check_result_index_bounds(model::ModelLike, attr)

This function checks whether enough results are available in the model for the requested attr, using its result_index field. If the model does not have sufficient results to answer the query, it throws a ResultIndexBoundsError.

source

As discussed in JuMP mapping, for scalar constraint with a nonzero function constant, a ScalarFunctionConstantNotZero exception may be thrown:

MathOptInterface.ScalarFunctionConstantNotZero - Type.

```
struct ScalarFunctionConstantNotZero{T, F, S} <: Exception
    constant::T
end</pre>
```

An error indicating that the constant part of the function in the constraint F-in-S is nonzero.

source

Some VariableIndex constraints cannot be combined on the same variable:

MathOptInterface.LowerBoundAlreadySet - Type.

```
LowerBoundAlreadySet{S1, S2}
```

Error thrown when setting a VariableIndex-in-S2 when a VariableIndex-in-S1 has already been added and the sets S1, S2 both set a lower bound, that is, they are EqualTo, GreaterThan, Interval, Semicontinuous or Semiinteger.

source

MathOptInterface.UpperBoundAlreadySet - Type.

```
UpperBoundAlreadySet{S1, S2}
```

Error thrown when setting a VariableIndex-in-S2 when a VariableIndex-in-S1 has already been added and the sets S1, S2 both set an upper bound, that is, they are EqualTo, LessThan, Interval, Semicontinuous or Semiinteger.

source

As discussed in AbstractCallback, trying to get attributes inside a callback may throw:

MathOptInterface.OptimizeInProgress - Type.

```
struct OptimizeInProgress{AttrType<:AnyAttribute} <: Exception
    attr::AttrType
end</pre>
```

Error thrown from optimizer when MOI.get(optimizer, attr) is called inside an AbstractCallback while it is only defined once optimize! has completed. This can only happen when is_set_by_optimize(attr) is true.

source

Trying to submit the wrong type of AbstractSubmittable inside an AbstractCallback (for example, a UserCut inside a LazyConstraintCallback) will throw:

MathOptInterface.InvalidCallbackUsage - Type.

```
struct InvalidCallbackUsage{C, S} <: Exception
    callback::C
    submittable::S
end</pre>
```

An error indicating that submittable cannot be submitted inside callback.

For example, UserCut cannot be submitted inside LazyConstraintCallback.

source

The rest of the errors defined in MOI fall in two categories represented by the following two abstract types:

MathOptInterface.UnsupportedError - Type.

UnsupportedError <: Exception</pre>

Abstract type for error thrown when an element is not supported by the model.

source

MathOptInterface.NotAllowedError - Type.

NotAllowedError <: Exception

Abstract type for error thrown when an operation is supported but cannot be applied in the current state of the model.

source

The different UnsupportedError and NotAllowedError are the following errors:

MathOptInterface.UnsupportedAttribute - Type.

```
struct UnsupportedAttribute{AttrType} <: UnsupportedError
    attr::AttrType
    message::String
end</pre>
```

An error indicating that the attribute attr is not supported by the model, that is, that supports returns false.

source

MathOptInterface.GetAttributeNotAllowed - Type.

```
struct GetAttributeNotAllowed{AttrType} <: NotAllowedError
    attr::AttrType
    message::String
end</pre>
```

An error indicating that the attribute attr cannot be got for some reason (see the error string).

source

MathOptInterface.SetAttributeNotAllowed - Type.

```
struct SetAttributeNotAllowed{AttrType} <: NotAllowedError
    attr::AttrType
    message::String
end</pre>
```

An error indicating that the attribute attr is supported (see supports) but cannot be set for some reason (see the error string).

source

MathOptInterface.AddVariableNotAllowed - Type.

```
struct AddVariableNotAllowed <: NotAllowedError
    message::String # Human-friendly explanation why the attribute cannot be set
end</pre>
```

An error indicating that variables cannot be added to the model.

source

MathOptInterface.UnsupportedConstraint - Type.

```
struct UnsupportedConstraint{F<:AbstractFunction,S<:AbstractSet} <: UnsupportedError
    message::String
end</pre>
```

An error indicating that constraints of type F-in-S are not supported by the model, that is, that supports_constraint returns false.

source

MathOptInterface.AddConstraintNotAllowed - Type.

```
struct AddConstraintNotAllowed{F<:AbstractFunction, S<:AbstractSet} <: NotAllowedError
    message::String # Human-friendly explanation why the attribute cannot be set
end</pre>
```

An error indicating that constraints of type F-in-S are supported (see supports_constraint) but cannot be added.

source

MathOptInterface.ModifyConstraintNotAllowed - Type.

An error indicating that the constraint modification change cannot be applied to the constraint of index ci.

source

MathOptInterface.ModifyObjectiveNotAllowed - Type.

```
struct ModifyObjectiveNotAllowed{C<:AbstractFunctionModification} <: NotAllowedError
    change::C
    message::String
end</pre>
```

An error indicating that the objective modification change cannot be applied to the objective.

source

MathOptInterface.DeleteNotAllowed - Type.

```
struct DeleteNotAllowed{IndexType <: Index} <: NotAllowedError
    index::IndexType
    message::String
end</pre>
```

An error indicating that the index index cannot be deleted.

source

MathOptInterface.UnsupportedSubmittable - Type.

```
struct UnsupportedSubmittable{SubmitType} <: UnsupportedError
    sub::SubmitType
    message::String
end</pre>
```

An error indicating that the submittable sub is not supported by the model, that is, that supports returns false.

source

MathOptInterface.SubmitNotAllowed - Type.

```
struct SubmitNotAllowed{SubmitTyp<:AbstractSubmittable} <: NotAllowedError
    sub::SubmitType
    message::String
end</pre>
```

An error indicating that the submittable sub is supported (see supports) but cannot be added for some reason (see the error string).

source

MathOptInterface.UnsupportedNonlinearOperator - Type.

UnsupportedNonlinearOperator(head::Symbol[, message::String]) <: UnsupportedError</pre>

An error thrown by optimizers if they do not support the operator head in a ScalarNonlinearFunction.

Example

source

Note that setting the ConstraintFunction of a VariableIndex constraint is not allowed:

MathOptInterface.SettingVariableIndexNotAllowed - Type.

SettingVariableIndexNotAllowed()

Error type that should be thrown when the user calls set to change the ConstraintFunction of a VariableIndex constraint.

source

Chapter 37

Submodules

37.1 Benchmarks

Overview

The Benchmarks submodule

To aid the development of efficient solver wrappers, MathOptInterface provides benchmarking capability. Benchmarking a wrapper follows a two-step process.

First, prior to making changes, create a baseline for the benchmark results on a given benchmark suite as follows:

```
using SolverPackage # Replace with your choice of solver.
import MathOptInterface as MOI
suite = MOI.Benchmarks.suite() do
   SolverPackage.Optimizer()
end
MOI.Benchmarks.create_baseline(
   suite, "current"; directory = "/tmp", verbose = true
)
```

Use the exclude argument to Benchmarks.suite to exclude benchmarks that the solver doesn't support.

Second, after making changes to the package, re-run the benchmark suite and compare to the prior saved results:

```
using SolverPackage
import MathOptInterface as MOI
suite = MOI.Benchmarks.suite() do
    SolverPackage.Optimizer()
end
MOI.Benchmarks.compare_against_baseline(
    suite, "current"; directory = "/tmp", verbose = true
)
```

This comparison will create a report detailing improvements and regressions.

API Reference

Benchmarks

Functions to help benchmark the performance of solver wrappers. See The Benchmarks submodule for more details.

MathOptInterface.Benchmarks.suite - Function.

```
suite(
    new_model::Function;
    exclude::Vector{Regex} = Regex[]
)
```

Create a suite of benchmarks. new_model should be a function that takes no arguments, and returns a new instance of the optimizer you wish to benchmark.

Use exclude to exclude a subset of benchmarks.

Example

```
julia> MOI.Benchmarks.suite() do
    return GLPK.Optimizer()
    end
julia> MOI.Benchmarks.suite(; exclude = [r"delete"]) do
    return Gurobi.Optimizer()
    end
```

source

MathOptInterface.Benchmarks.create_baseline - Function.

create_baseline(suite, name::String; directory::String = ""; kwargs...)

Run all benchmarks in suite and save to files called name in directory.

Extra kwargs are based to BenchmarkTools.run.

Example

MathOptInterface.Benchmarks.compare_against_baseline - Function.

```
compare_against_baseline(
    suite, name::String; directory::String = "",
    report_filename::String = "report.txt"
)
```

Run all benchmarks in suite and compare against files called name in directory that were created by a call to create_baseline.

A report summarizing the comparison is written to report_filename in directory.

Extra kwargs are based to BenchmarkTools.run.

Example

source

37.2 Bridges

Overview

The Bridges submodule

The Bridges module simplifies the process of converting models between equivalent formulations.

Tip

Read our paper for more details on how bridges are implemented.

Why bridges?

A constraint can often be written in a number of equivalent formulations. For example, the constraint $l \leq a^{\top}x \leq u$ (ScalarAffineFunction-in-Interval) could be re-formulated as two constraints: $a^{\top}x \geq l$ (ScalarAffineFunction-in-GreaterThan) and $a^{\top}x \leq u$ (ScalarAffineFunction-in-LessThan). An alternative re-formulation is to add a dummy variable y with the constraints $l \leq y \leq u$ (VariableIndex-in-Interval) and $a^{\top}x - y = 0$ (ScalarAffineFunction-in-EqualTo).

To avoid each solver having to code these transformations manually, MathOptInterface provides bridges.

A bridge is a small transformation from one constraint type to another (potentially collection of) constraint type.

Because these bridges are included in MathOptInterface, they can be re-used by any optimizer. Some bridges also implement constraint modifications and constraint primal and dual translations.

Several bridges can be used in combination to transform a single constraint into a form that the solver may understand. Choosing the bridges to use takes the form of finding a shortest path in the hyper-graph of bridges. The methodology is detailed in the MOI paper.

The three types of bridges

There are three types of bridges in MathOptInterface:

- 1. Constraint bridges
- 2. Variable bridges
- 3. Objective bridges

Constraint bridges

Constraint bridges convert constraints formulated by the user into an equivalent form supported by the solver. Constraint bridges are subtypes of Bridges.Constraint.AbstractBridge.

The equivalent formulation may add constraints (and possibly also variables) in the underlying model.

In particular, constraint bridges can focus on rewriting the function of a constraint, and do not change the set. Function bridges are subtypes of Bridges.Constraint.AbstractFunctionConversionBridge.

Read the list of implemented constraint bridges for more details on the types of transformations that are available. Function bridges are Bridges.Constraint.ScalarFunctionizeBridge and Bridges.Constraint.VectorFunctionizeBri

Variable bridges

Variable bridges convert variables added by the user, either free with add_variable/add_variables, or constrained with add_constrained_variable/add_constrained_variables, into an equivalent form supported by the solver. Variable bridges are subtypes of Bridges.Variable.AbstractBridge.

The equivalent formulation may add constraints (and possibly also variables) in the underlying model.

Read the list of implemented variable bridges for more details on the types of transformations that are available.

Objective bridges

Objective bridges convert the ObjectiveFunction set by the user into an equivalent form supported by the solver. Objective bridges are subtypes of Bridges.Objective.AbstractBridge.

The equivalent formulation may add constraints (and possibly also variables) in the underlying model.

Read the list of implemented objective bridges for more details on the types of transformations that are available.

Bridges.full_bridge_optimizer

Tip

Unless you have an advanced use-case, this is probably the only function you need to care about.

To enable the full power of MathOptInterface's bridges, wrap an optimizer in a Bridges.full_bridge_optimizer.

```
julia> inner_optimizer = MOI.Utilities.Model{Float64}()
```

```
MOIU.Model{Float64}
```

```
bjectiveSense: FEASIBILITY_SENSE
```

bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}

```
- NumberOfVariables: 0
```

```
L NumberOfConstraints: 0
```

julia> optimizer = MOI.Bridges.full_bridge_optimizer(inner_optimizer, Float64)

- MOIB.LazyBridgeOptimizer{MOIU.Model{Float64}}
- Variable bridges: none
- Constraint bridges: none
- Objective bridges: none
- L model: MOIU.Model{Float64}
- bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
- NumberOfVariables: 0
- L NumberOfConstraints: 0

Now, use optimizer as normal, and bridging will happen lazily behind the scenes. By lazily, we mean that bridging will happen if and only if the constraint is not supported by the inner_optimizer.

Info

Most bridges are added by default in Bridges.full_bridge_optimizer. However, for technical reasons, some bridges are not added by default. Three examples include Bridges.Constraint.SOCtoPSDBridge, Bridges.Constraint.SOCtoNonConvexQuadBridge and Bridges.Constraint.RSOCtoNonConvexQuadBridge. See the docs of those bridges for more information.

Add a single bridge

If you don't want to use Bridges.full_bridge_optimizer, you can wrap an optimizer in a single bridge.

However, this will force the constraint to be bridged, even if the inner_optimizer supports it.

```
julia> inner_optimizer = MOI.Utilities.Model{Float64}();
```

julia> optimizer = MOI.Bridges.Constraint.SplitInterval{Float64}(inner_optimizer);

```
julia> x = MOI.add_variable(optimizer)
MOI.VariableIndex(1)
```

```
julia> MOI.add_constraint(optimizer, x, MOI.Interval(0.0, 1.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
→ MathOptInterface.Interval{Float64}}(1)
```

```
julia> MOI.get(optimizer, MOI.ListOfConstraintTypesPresent())
1-element Vector{Tuple{Type, Type}}:
  (MathOptInterface.VariableIndex, MathOptInterface.Interval{Float64})
```

```
julia> MOI.get(inner_optimizer, MOI.ListOfConstraintTypesPresent())
2-element Vector{Tuple{Type, Type}}:
  (MathOptInterface.VariableIndex, MathOptInterface.GreaterThan{Float64})
  (MathOptInterface.VariableIndex, MathOptInterface.LessThan{Float64})
```

Bridges.LazyBridgeOptimizer

If you don't want to use Bridges.full_bridge_optimizer, but you need more than a single bridge (or you want the bridging to happen lazily), you can manually construct a Bridges.LazyBridgeOptimizer.

First, wrap an inner optimizer:

```
julia> inner_optimizer = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}
- ObjectiveSense: FEASIBILITY SENSE
bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
- NumberOfVariables: 0
L NumberOfConstraints: 0
julia> optimizer = MOI.Bridges.LazyBridgeOptimizer(inner_optimizer)
MOIB.LazyBridgeOptimizer{MOIU.Model{Float64}}
- Variable bridges: none
- Constraint bridges: none
- Objective bridges: none
L model: MOIU.Model{Float64}
```

- F ObjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
- NumberOfVariables: 0
- L NumberOfConstraints: 0

Then use Bridges.add bridge to add individual bridges:

julia> MOI.Bridges.add_bridge(optimizer, MOI.Bridges.Constraint.SplitIntervalBridge{Float64})

julia> MOI.Bridges.add_bridge(optimizer, MOI.Bridges.Objective.FunctionizeBridge{Float64})

Now the constraints will be bridged only if needed:

```
julia> x = MOI.add_variable(optimizer)
MOI.VariableIndex(1)
```

julia> MOI.add_constraint(optimizer, x, MOI.Interval(0.0, 1.0)) MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, \hookrightarrow MathOptInterface.Interval{Float64}}(1)

```
julia> MOI.get(optimizer, MOI.ListOfConstraintTypesPresent())
1-element Vector{Tuple{Type, Type}}:
(MathOptInterface.VariableIndex, MathOptInterface.Interval{Float64})
```

julia> MOI.get(inner_optimizer, MOI.ListOfConstraintTypesPresent()) 1-element Vector{Tuple{Type, Type}}: (MathOptInterface.VariableIndex, MathOptInterface.Interval{Float64})

Implementation

Implementing a bridge

The easiest way to implement a bridge is to follow an existing example. There are three locations of bridges in the source code:

- Constraint bridges are stored in src/Bridges/Constraint/bridges
- Objective bridges are stored in src/Bridges/Objective/bridges
- Variable bridges are stored in src/Bridges/Variable/bridges

The Implementing a constraint bridge tutorial has a more detailed guide on what is required to implement a bridge.

When opening a pull request that adds a new bridge, use the checklist Adding a new bridge.

If you need help or advice, please contact the Developer Chatroom.

SetMap bridges

For constraint and variable bridges, a common reformulation is that $f(x) \in F$ is reformulated to $g(x) \in G$. In this case, no additional variables and constraints are added, and the bridge needs only a way to map between the functions f and g and the sets F and G.

To implementation a bridge of this form, subtype the abstract type Bridges.Constraint.SetMapBridge or Bridges.Variable.SetMapBridge and implement the API described in the docstring of each type.

final_touch

Some bridges require information from other parts of the model. One set of examples are the various combinatorial ToMILP bridges, such as Bridges.Constraint.SOSIToMILPBridge, which require knowledge of the variable bounds.

Bridges requiring information from other parts of the model should implement Bridges.final_touch and Bridges.needs_final_touch.

During the bridge's construction, store the function and set and make no changes to the underlying model. Then, in Bridges.final_touch, query the additional information and add the reformulated problem to the model.

When implementing, you must consider that:

- Bridges.final_touch may be called multiple times, so that your reformulation should be applied only if necessary. Sometimes the additional data will be the same, and sometimes it may be different.
- We do not currently support final_touch bridges that introduce constraints which also require a final_touch bridge. Therefore, you should implement final_touch only if necessary, and we recommend that you contact the Developer Chatroom for advice before doing so.

Testing

Use the Bridges.runtests function to test a bridge. It takes three arguments: the type of the bridge, the input model as a string, and the output model as a string.

Here is an example:
```
"""
variables: x
-1.0 * x <= -1.0
""",
)
Test Summary: | Pass Total Time
Bridges.runtests | 29 29 0.0s</pre>
```

There are a number of other useful keyword arguments.

- eltype can be used to specify the element type of the model (and bridge). It defaults to Float64.
- variable_start and constraint_start are used as the values to set the VariablePrimalStart and ConstraintPrimalStart attributes to. They default to 1.2. If you use a different eltype, you must set appropriate starting values of the same type. The default 1.2 was chosen to minimize the risk that the starting point is undefined, which could happen for common situations like 0.0 and 1.0. The tests associated with the starting values do not necessarily check for correctness, only that they can be set and get to produce the same result.
- print_inner_model can be used to print the reformulated output model from the bridge. This is especially helpful during debugging to see what the bridge is doing, and to spot mistakes. It defaults to false.

Here is an example:

```
julia> MOI.Bridges.runtests(
           MOI.Bridges.Constraint.GreaterToLessBridge,
           .....
           variables: x
          x >= 1
          ....
          .....
          variables: x
          ::Int: -1 * x <= -1
          ....
          eltype = Int,
           print_inner_model = true,
           variable_start = 2,
           constraint_start = 2,
      )
Feasibility
Subject to:
ScalarAffineFunction{Int64}-in-LessThan{Int64}
(0) - (1) \times <= (-1)
Test Summary: | Pass Total Time
Bridges.runtests | 29 0.0s
```

List of bridges

List of bridges

This section describes the Bridges.AbstractBridges that are implemented in MathOptInterface.

Constraint bridges

These bridges are subtypes of Bridges.Constraint.AbstractBridge.

MathOptInterface.Bridges.Constraint.AbstractFunctionConversionBridge - Type.

abstract type AbstractFunctionConversionBridge{F,S} <: AbstractBridge end</pre>

Abstract type to support writing bridges in which the function changes but the set does not.

By convention, the transformed function is stored in the .constraint field.

source

MathOptInterface.Bridges.Constraint.AbstractToIntervalBridge - Type.

AbstractToIntervalBridge{T<:AbstractFloat,S,F}</pre>

An abstract type that simplifies the creation of other bridges.

Warning

T must be a AbstractFloat type because otherwise typemin and typemax would either be not implemented (for example, BigInt), or would not give infinite value (for example, Int). For this reason, this bridge is only added to MOI.Bridges.full_bridge_optimizer when T is a subtype of AbstractFloat.

source

MathOptInterface.Bridges.Constraint.AllDifferentToCountDistinctBridge - Type.

AllDifferentToCountDistinctBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

AllDifferentToCountDistinctBridge implements the following reformulations:

- $x \in AllDifferent(d)$ to $(n, x) \in CountDistinct(1 + d)$ and n = d
- $f(x) \in AllDifferent(d)$ to $(d, f(x)) \in CountDistinct(1 + d)$

Source node

AllDifferentToCountDistinctBridge supports:

• F in MOI.AllDifferent

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

AllDifferentToCountDistinctBridge creates:

• F in MOI.CountDistinct

MOI.VariableIndex in MOI.EqualTo{T}

```
source
```

MathOptInterface.Bridges.Constraint.BinPackingToMILPBridge - Type.

BinPackingToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

BinPackingToMILPBridge implements the following reformulation:

• $x \in BinPacking(c, w)$ into a mixed-integer linear program.

Reformulation

The reformulation is non-trivial, and it depends on the finite domain of each variable x_i , which we as define $S_i = \{l_i, \ldots, u_i\}$.

First, we introduce new binary variables z_{ij} , which are 1 if variable x_i takes the value j in the optimal solution and 0 otherwise:

$$z_{ij} \in \{0, 1\} \quad \forall i \in 1 \dots d, j \in S_i$$
$$x_i - \sum_{j \in S_i} j \cdot z_{ij} = 0 \quad \forall i \in 1 \dots d$$
$$\sum_{j \in S_i} z_{ij} = 1 \quad \forall i \in 1 \dots d$$

Then, we add the capacity constraint for all possible bins j:

$$\sum_{i} w_i z_{ij} \le c \forall j \in \bigcup_{i=1,\dots,d} S_i$$

Source node

BinPackingToMILPBridge supports:

• F in MOI.BinPacking{T}

Target nodes

BinPackingToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}
- MOI.ScalarAffineFunction{T} in MOI.LessThan{T}

source

MathOptInterface.Bridges.Constraint.CircuitToMILPBridge - Type.

CircuitToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

CircuitToMILPBridge implements the following reformulation:

• $x \in Circuit(d)$ to the Miller-Tucker-Zemlin formulation of the Traveling Salesperson Problem.

Source node

CircuitToMILPBridge supports:

• F in MOI.Circuit

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

CircuitToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.VariableIndex in MOI.Integer
- MOI.VariableIndex in MOI.Interval{T}
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}
- MOI.ScalarAffineFunction{T} in MOI.LessThan{T}

source

MathOptInterface.Bridges.Constraint.ComplexNormInfinityToSecondOrderConeBridge - Type.

ComplexNormInfinityToSecondOrderConeBridge{T} <: Bridges.Constraint.AbstractBridge</pre>

ComplexNormInfinityToSecondOrderConeBridge implements the following reformulation:

• $(t, x) \in NormInfinity(1 + d)$ into $(t, real(x_i), imag(x_i)) \in SecondOrderCone()$ for all *i*.

Source node

ComplexNormInfinityToSecondOrderConeBridge supports:

• MOI.VectorAffineFunction{Complex{T}} in MOI.NormInfinityCone

Target nodes

ComplexNormInfinityToSecondOrderConeBridge creates:

• MOI.VectorAffineFunction{T} in MOI.SecondOrderCone

source

MathOptInterface.Bridges.Constraint.CountAtLeastToCountBelongsBridge - Type.

CountAtLeastToCountBelongsBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

CountAtLeastToCountBelongsBridge implements the following reformulation:

• $x \in \text{CountAtLeast}(n, d, S)$ to $(n_i, x_{d_i}) \in \text{CountBelongs}(1 + d, S)$ and $n_i \ge n$ for all i.

Source node

CountAtLeastToCountBelongsBridge supports:

• F in MOI.CountAtLeast

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

CountAtLeastToCountBelongsBridge creates:

- F in MOI.CountBelongs
- MOI.VariableIndex in MOI.GreaterThan{T}

source

MathOptInterface.Bridges.Constraint.CountBelongsToMILPBridge - Type.

CountBelongsToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

CountBelongsToMILPBridge implements the following reformulation:

• $(n, x) \in \text{CountBelongs}(1 + d, S)$ into a mixed-integer linear program.

Reformulation

The reformulation is non-trivial, and it depends on the finite domain of each variable x_i , which we as define $S_i = \{l_i, \ldots, u_i\}$.

First, we introduce new binary variables z_{ij} , which are 1 if variable x_i takes the value j in the optimal solution and 0 otherwise:

$$z_{ij} \in \{0,1\} \quad \forall i \in 1 \dots d, j \in S_i$$
$$x_i - \sum_{j \in S_i} j \cdot z_{ij} = 0 \quad \forall i \in 1 \dots d$$
$$\sum_{j \in S_i} z_{ij} = 1 \quad \forall i \in 1 \dots d$$

Finally, n is constrained to be the number of z_{ij} elements that are in S:

$$n - \sum_{i \in 1...d, j \in \mathcal{S}} z_{ij} = 0$$

Source node

CountBelongsToMILPBridge supports:

• F in MOI.CountBelongs

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

CountBelongsToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}

source

MathOptInterface.Bridges.Constraint.CountDistinctToMILPBridge - Type.

CountDistinctToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

 ${\tt CountDistinctToMILPBridge\ implements\ the\ following\ reformulation:}$

• $(n, x) \in \text{CountDistinct}(1 + d)$ into a mixed-integer linear program.

Reformulation

The reformulation is non-trivial, and it depends on the finite domain of each variable x_i , which we as define $S_i = \{l_i, \ldots, u_i\}$.

First, we introduce new binary variables z_{ij} , which are 1 if variable x_i takes the value j in the optimal solution and 0 otherwise:

$$z_{ij} \in \{0, 1\} \quad \forall i \in 1 \dots d, j \in S_i$$
$$x_i - \sum_{j \in S_i} j \cdot z_{ij} = 0 \quad \forall i \in 1 \dots d$$
$$\sum_{j \in S_i} z_{ij} = 1 \quad \forall i \in 1 \dots d$$

Then, we introduce new binary variables y_j , which are 1 if a variable takes the value j in the optimal solution and 0 otherwise.

$$y_j \in \{0,1\} \ \forall j \in \bigcup_{i=1,\dots,d} S_i$$
$$y_j \le \sum_{i \in 1\dots d: j \in S_i} z_{ij} \le M y_j \ \forall j \in \bigcup_{i=1,\dots,d} S_i$$

Finally, n is constrained to be the number of y_j elements that are non-zero:

$$n - \sum_{j \in \bigcup_{i=1,\dots,d} S_i} y_j = 0$$

Formulation (special case)

In the special case that the constraint is [2, x, y] in CountDistinct(3), then the constraint is equivalent to [x, y] in AllDifferent(2), which is equivalent to x != y.

$$(x - y \le -1) \lor (y - x \le -1)$$

which is equivalent to (for suitable M):

$$z \in \{0, 1\}$$
$$x - y - Mz \le -1$$
$$y - x - M(1 - z) \le -1$$

Source node

CountDistinctToMILPBridge supports:

• F in MOI.CountDistinct

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

CountDistinctToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}
- MOI.ScalarAffineFunction{T} in MOI.LessThan{T}

source

MathOptInterface.Bridges.Constraint.CountGreaterThanToMILPBridge - Type.

CountGreaterThanToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

CountGreaterThanToMILPBridge implements the following reformulation:

• $(c, y, x) \in CountGreaterThan()$ into a mixed-integer linear program.

Source node

CountGreaterThanToMILPBridge supports:

• F in MOI.CountGreaterThan

Target nodes

CountGreaterThanToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}

MOI.ScalarAffineFunction{T} in MOI.GreaterThan{T}

source

MathOptInterface.Bridges.Constraint.ExponentialConeToScalarNonlinearFunctionBridge - Type.

```
ExponentialConeToScalarNonlinearFunctionBridge{T,F} <:
    Bridges.Constraint.AbstractBridge</pre>
```

ExponentialConeToScalarNonlinearFunctionBridge implements the following reformulation:

• $(x, y, z) \in \text{ExponentialCone}()$ to $y \cdot exp(x/y)) - z \le 0, y \ge 0$.

Source node

ExponentialConeToScalarNonlinearFunctionBridge supports:

• F in MOI.ExponentialCone

Target nodes

ExponentialConeToScalarNonlinearFunctionBridge creates:

- MOI.ScalarNonlinearFunction in MOI.LessThan{T}
- MOI.ScalarAffineFunction in MOI.GreaterThan{T}

source

MathOptInterface.Bridges.Constraint.FlipSignBridge - Type.

FlipSignBridge{T,S1,S2,F,G}

An abstract type that simplifies the creation of other bridges.

source

MathOptInterface.Bridges.Constraint.FunctionConversionBridge - Type.

FunctionConversionBridge{T,F,G,S} <: AbstractFunctionConversionBridge{G,S}</pre>

FunctionConversionBridge implements the following reformulations:

• $g(x) \in S$ into $f(x) \in S$

for these pairs of functions:

- MOI.ScalarAffineFunctionto [MOI.ScalarQuadraticFunction'](@ref)
- MOI.ScalarQuadraticFunction to MOI.ScalarNonlinearFunction
- MOI.VectorAffineFunction to MOI.VectorQuadraticFunction

See also SetConversionBridge.

Source node

FunctionConversionBridge supports:

• G in S

Target nodes

FunctionConversionBridge creates:

• F in S

source

MathOptInterface.Bridges.Constraint.GeoMeanBridge - Type.

GeoMeanBridge{T,F,G,H} <: Bridges.Constraint.AbstractBridge</pre>

 $Geo Mean Bridge \ implements \ a \ reformulation \ from \ MOI. Geometric Mean Cone \ into \ MOI. Rotated \\ Second \\ Order \\ Cone. \\$

The reformulation is best described in an example.

Consider the cone of dimension 4:

$$t \le \sqrt[3]{x_1 x_2 x_3}$$

This can be rewritten as $\exists y \geq 0$ such that:

$$t \le y,$$

$$y^4 \le x_1 x_2 x_3 y.$$

Note that we need to create y and not use t^4 directly because t is not allowed to be negative.

This is equivalent to:

$$\begin{split} t &\leq \frac{y_1}{\sqrt{4}}, \\ y_1^2 &\leq 2y_2y_3, \\ y_2^2 &\leq 2x_1x_2, \\ y_3^2 &\leq 2x_3(y_1/\sqrt{4}) \\ y &\geq 0. \end{split}$$

More generally, you can show how the geometric mean code is recursively expanded into a set of new variables y in MOI.Nonnegatives, a set of MOI.RotatedSecondOrderCone constraints, and a MOI.LessThan constraint between t and y_1 .

Source node

GeoMeanBridge supports:

• H in MOI.GeometricMeanCone

Target nodes

GeoMeanBridge creates:

- F in MOI.LessThan{T}
- G in MOI.RotatedSecondOrderCone
- G in MOI.Nonnegatives

source

MathOptInterface.Bridges.Constraint.GeoMeanToPowerBridge - Type.

GeoMeanToPowerBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

GeoMeanToPowerBridge implements the following reformulation:

• $(y, x...) \in GeometricMeanCone(1+d)$ into $(x_1, t, y) \in PowerCone(1/d)$ and $(t, x_2, ..., x_d)$ in GeometricMeanCone which is then recursively expanded into more PowerCone constraints.

Source node

GeoMeanToPowerBridge supports:

• F in MOI.GeometricMeanCone

Target nodes

GeoMeanToPowerBridge creates:

- F in MOI.PowerCone{T}
- MOI.VectorOfVariables in MOI.Nonnegatives

source

MathOptInterface.Bridges.Constraint.GeoMeantoRelEntrBridge - Type.

GeoMeantoRelEntrBridge{T,F,G,H} <: Bridges.Constraint.AbstractBridge</pre>

GeoMeantoRelEntrBridge implements the following reformulation:

• $(u, w) \in GeometricMeanCone$ into $(0, w, (u + y)\mathbf{1}) \in RelativeEntropyCone$ and $y \ge 0$

Source node

GeoMeantoRelEntrBridge supports:

• H in MOI.GeometricMeanCone

Target nodes

GeoMeantoRelEntrBridge creates:

- G in MOI.RelativeEntropyCone
- F in MOI.Nonnegatives

Derivation

The derivation of the bridge is as follows:

$$\begin{split} (u,w) \in GeometricMeanCone \iff & u \leq \left(\prod_{i=1}^{n} w_{i}\right)^{1/n} \\ \iff & 0 \leq u+y \leq \left(\prod_{i=1}^{n} w_{i}\right)^{1/n}, y \geq 0 \\ \iff & 1 \leq \frac{\left(\prod_{i=1}^{n} w_{i}\right)^{1/n}}{u+y}, y \geq 0 \\ \iff & 1 \leq \left(\prod_{i=1}^{n} \frac{w_{i}}{u+y}\right)^{1/n}, y \geq 0 \\ \iff & 0 \leq \sum_{i=1}^{n} \log\left(\frac{w_{i}}{u+y}\right), y \geq 0 \\ \iff & 0 \geq \sum_{i=1}^{n} \log\left(\frac{u+y}{w_{i}}\right), y \geq 0 \\ \iff & 0 \geq \sum_{i=1}^{n} \log\left(\frac{u+y}{w_{i}}\right), y \geq 0 \\ \iff & 0 \geq \sum_{i=1}^{n} (u+y) \log\left(\frac{u+y}{w_{i}}\right), y \geq 0 \\ \iff & 0 \geq \sum_{i=1}^{n} (u+y) \log\left(\frac{u+y}{w_{i}}\right), y \geq 0 \\ \iff & 0 \geq \sum_{i=1}^{n} (u+y) \log\left(\frac{u+y}{w_{i}}\right), y \geq 0 \end{split}$$

This derivation assumes that u + y > 0, which is enforced by the relative entropy cone.

source

MathOptInterface.Bridges.Constraint.GreaterToIntervalBridge - Type.

GreaterToIntervalBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

GreaterToIntervalBridge implements the following reformulations:

• $f(x) \ge l$ into $f(x) \in [l, \infty)$

Source node

GreaterToIntervalBridge supports:

• F in MOI.GreaterThan{T}

Target nodes

GreaterToIntervalBridge creates:

• F in MOI.Interval{T}

source

MathOptInterface.Bridges.Constraint.GreaterToLessBridge - Type.

GreaterToLessBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

GreaterToLessBridge implements the following reformulation:

• $f(x) \ge l$ into $-f(x) \le -l$

Source node

GreaterToLessBridge supports:

• G in MOI.GreaterThan{T}

Target nodes

GreaterToLessBridge creates:

• F in MOI.LessThan{T}

source

MathOptInterface.Bridges.Constraint.HermitianToComplexSymmetricBridge - Type.

HermitianToComplexSymmetricBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

HermitianToSymmetricBridge implements the following reformulation:

• Hermitian positive semidefinite n x n represented as a vector of real entries with real and imaginary parts on different entries to a vector of complex entries.

See also MOI.Bridges.Constraint.HermitianToSymmetricPSDBridge.

Source node

HermitianToComplexSymmetricBridge supports:

• G in MOI.HermitianPositiveSemidefiniteConeTriangle

Target node

HermitianToComplexSymmetricBridge creates:

• F in MOI.PositiveSemidefiniteConeTriangle

Note that if G is MOI.VectorAffineFunction{T} then F will be MOI.VectorAffineFunction{Complex{T}} source

MathOptInterface.Bridges.Constraint.HermitianToSymmetricPSDBridge - Type.

HermitianToSymmetricPSDBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

HermitianToSymmetricPSDBridge implements the following reformulation:

• Hermitian positive semidefinite n x n complex matrix to a symmetric positive semidefinite 2n x 2n real matrix.

See also MOI.Bridges.Variable.HermitianToSymmetricPSDBridge.

Source node

HermitianToSymmetricPSDBridge supports:

• G in MOI.HermitianPositiveSemidefiniteConeTriangle

Target node

HermitianToSymmetricPSDBridge creates:

• F in MOI.PositiveSemidefiniteConeTriangle

Reformulation

The reformulation is best described by example.

The Hermitian matrix:

| x_{11} | $x_{12} + y_{12}im$ | $x_{13} + y_{13}im$ |
|---------------------|---------------------|---------------------|
| $x_{12} - y_{12}im$ | x_{22} | $x_{23} + y_{23}im$ |
| $x_{13} - y_{13}im$ | $x_{23} - y_{23}im$ | x_{33} |

is positive semidefinite if and only if the symmetric matrix:

| x_{11} | x_{12} | x_{13} | 0 | y_{12} | y_{13} |
|----------|----------|----------|-----------|-----------|----------|
| | x_{22} | x_{23} | $-y_{12}$ | 0 | y_{23} |
| | | x_{33} | $-y_{13}$ | $-y_{23}$ | 0 |
| | | | x_{11} | x_{12} | x_{13} |
| | | | | x_{22} | x_{23} |
| | | | | | x_{33} |

is positive semidefinite.

The bridge achieves this reformulation by constraining the above matrix to belong to the MOI.PositiveSemidefiniteConeTrisource

MathOptInterface.Bridges.Constraint.IndicatorActiveOnFalseBridge - Type.

IndicatorActiveOnFalseBridge{T,F,S} <: Bridges.Constraint.AbstractBridge</pre>

IndicatorActiveOnFalseBridge implements the following reformulation:

• $\neg z \implies f(x) \in S$ into $y \implies f(x) \in S$, z + y = 1, and $y \in \{0, 1\}$

Source node

IndicatorActiveOnFalseBridge supports:

MOI.VectorAffineFunction{T} in MOI.Indicator{MOI.ACTIVATE_ON_ZER0,S}

Target nodes

IndicatorActiveOnFalseBridge creates:

- MOI.VectorAffineFunction{T} in MOI.Indicator{MOI.ACTIVATE_ON_ONE,S}
- MOI.ScalarAffineFunction{T} in MOI.EqualTo
- MOI.VariableIndex in MOI.ZeroOne

source

MathOptInterface.Bridges.Constraint.IndicatorGreaterToLessThanBridge - Type.

IndicatorGreaterToLessThanBridge{T,A} <: Bridges.Constraint.AbstractBridge</pre>

IndicatorGreaterToLessThanBridge implements the following reformulation:

•
$$z \implies f(x) \ge l$$
 into $z \implies -f(x) \le -l$

Source node

IndicatorGreaterToLessThanBridge supports:

• MOI.VectorAffineFunction{T} in MOI.Indicator{A,MOI.GreaterThan{T}}

Target nodes

IndicatorGreaterToLessThanBridge creates:

• MOI.VectorAffineFunction{T} in MOI.Indicator{A,MOI.LessThan{T}}

source

MathOptInterface.Bridges.Constraint.IndicatorLessToGreaterThanBridge - Type.

IndicatorLessToGreaterThanBridge{T,A} <: Bridges.Constraint.AbstractBridge</pre>

IndicatorLessToGreaterThanBridge implements the following reformulations:

•
$$z \implies f(x) \le u$$
 into $z \implies -f(x) \ge -u$

IndicatorLessToGreaterThanBridge supports:

• MOI.VectorAffineFunction{T} in MOI.Indicator{A,MOI.LessThan{T}}

Target nodes

IndicatorLessToGreaterThanBridge creates:

• MOI.VectorAffineFunction{T} in MOI.Indicator{A,MOI.GreaterThan{T}}

source

MathOptInterface.Bridges.Constraint.IndicatorSOS1Bridge - Type.

IndicatorSOS1Bridge{T,S} <: Bridges.Constraint.AbstractBridge</pre>

IndicatorS0S1Bridge implements the following reformulation:

• $z \implies f(x) \in S$ into $f(x) + y \in S$, SOS1(y, z)

Warning

This bridge assumes that the solver supports $MOI.SOS1{T}$ constraints in which one of the variables (y) is continuous.

Source node

IndicatorSOS1Bridge supports:

• MOI.VectorAffineFunction{T} in MOI.Indicator{MOI.ACTIVATE_ON_ONE,S}

Target nodes

IndicatorSOS1Bridge creates:

- MOI.ScalarAffineFunction{T} in S
- MOI.VectorOfVariables in MOI.SOS1{T}

source

MathOptInterface.Bridges.Constraint.IndicatorSetMapBridge - Type.

IndicatorSetMapBridge{T,A,S1,S2} <: Bridges.Constraint.AbstractBridge</pre>

IndicatorSetMapBridge implements the following reformulations:

- $z \implies f(x) \ge l$ into $z \implies -f(x) \le -l$
- $z \implies f(x) \le u$ into $z \implies -f(x) \ge -u$

IndicatorSetMapBridge supports:

• MOI.VectorAffineFunction{T} in MOI.Indicator{A,S1}

Target nodes

IndicatorSetMapBridge creates:

• MOI.VectorAffineFunction{T} in MOI.Indicator{A,S2}

source

MathOptInterface.Bridges.Constraint.IndicatorToMILPBridge - Type.

IndicatorToMILPBridge{T,F,A,S} <: Bridges.Constraint.AbstractBridge</pre>

IndicatorToMILPBridge implements the following reformulation:

• $x \in \text{Indicator}(s)$ into a mixed-integer linear program.

Source node

IndicatorToMILPBridge supports:

• F in MOI.Indicator{A,S}

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

IndicatorToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in S

source

MathOptInterface.Bridges.Constraint.InequalityToComplementsBridge - Type.

InequalityToComplementsBridge{T,F,S,G} <: Bridges.Constraint.AbstractBridge</pre>

InequalityToComplementsBridge implements the following reformulations:

- $f(x) \ge b$ into $\exists y$ such that $f(x) b \perp y \ge 0$
- $f(x) \le b$ into $f(x) b \perp y \le 0$
- f(x) = b into $f(x) b \perp y$

InequalityToComplementsBridge supports:

- F in MOI.GreaterThan{T}
- F in MOI.LessThan{T}
- F in MOI.EqualTo

Target nodes

InequalityToComplementsBridge creates:

- MOI.VariableIndex in MOI.LessThan{T}
- MOI.VariableIndex in MOI.GreaterThan{T}
- G in MOI.Complements

source

MathOptInterface.Bridges.Constraint.IntegerToZeroOneBridge - Type.

IntegerToZeroOneBridge{T} <: Bridges.Constraint.AbstractBridge</pre>

IntegerToZeroOneBridge implements the following reformulation:

•
$$x \in \mathbf{Z}$$
 into $y_i \in \{0, 1\}$, $x == lb + \sum 2^{i-1}y_i$.

Source node

IntegerToZeroOneBridge supports:

• VariableIndex in MOI.Integer

Target nodes

IntegerToZeroOneBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}

Developer note

This bridge is implemented as a constraint bridge instead of a variable bridge because we don't want to substitute the linear combination of y for every instance of x. Doing so would be expensive and greatly reduce the sparsity of the constraints.

source

MathOptInterface.Bridges.Constraint.LessToGreaterBridge - Type.

LessToGreaterBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

LessToGreaterBridge implements the following reformulation:

•
$$f(x) \le u$$
 into $-f(x) \ge -u$

LessToGreaterBridge supports:

• G in MOI.LessThan{T}

Target nodes

LessToGreaterBridge creates:

• F in MOI.GreaterThan{T}

source

MathOptInterface.Bridges.Constraint.LessToIntervalBridge - Type.

LessToIntervalBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

LessToIntervalBridge implements the following reformulations:

• $f(x) \le u$ into $f(x) \in (-\infty, u]$

Source node

LessToIntervalBridge supports:

• F in MOI.LessThan{T}

Target nodes

LessToIntervalBridge creates:

• F in MOI.Interval{T}

source

MathOptInterface.Bridges.Constraint.LogDetBridge - Type.

LogDetBridge{T,F,G,H,I} <: Bridges.Constraint.AbstractBridge</pre>

The MOI.LogDetConeTriangle is representable by MOI.PositiveSemidefiniteConeTriangle and MOI.ExponentialCone constraints.

Indeed, $\log \det(X) = \sum_{i=1}^n \log(\delta_i)$ where δ_i are the eigenvalues of X.

Adapting the method from [1, p. 149], we see that $t \le u \log(\det(X/u))$ for u > 0 if and only if there exists a lower triangular matrix such that

$$\begin{pmatrix} X \\ \top & \text{Diag}() \end{pmatrix} \succeq 0$$
$$t - \sum_{i=1}^{n} u \log\left(\frac{ii}{u}\right) \le 0$$

Which we reformulate further into

$$\begin{pmatrix} X \\ \top & \text{Diag}() \end{pmatrix} \succeq 0 \\ (l_i, u, i_i) \in ExponentialCone \quad \forall i \\ t - \sum_{i=1}^n l_i \le 0$$

Source node

LogDetBridge supports:

• I in MOI.LogDetConeTriangle

Target nodes

LogDetBridge creates:

- F in MOI.PositiveSemidefiniteConeTriangle
- G in MOI.ExponentialCone
- H in MOI.LessThan{T}

[1] Ben-Tal, Aharon, and Arkadi Nemirovski. *Lectures on modern convex optimization: analysis, algorithms, and engineering applications*. Society for Industrial and Applied Mathematics, 2001.

source

MathOptInterface.Bridges.Constraint.MultiSetMapBridge - Type.

abstract type MultiSetMapBridge{T,S1,G} <: AbstractBridge end</pre>

Same as SetMapBridge but the output constraint type does not only depend on the input constraint type.

When subtyping MultiSetMapBridge, added_constraint_types and supports should additionally be implemented by the bridge.

For example, if a bridge BridgeType may create either a constraint of type F2-in-S2 or F3-in-S3, these methods should be implemented as follows:

```
function MOI.Bridges.added_constraint_types(
    ::Type{<:BridgeType{T,F2,F3}},
) where {T,F2,F3}
    return Tuple{Type,Type}[(F2, S2), (F3, S3)]
end</pre>
```

source

MathOptInterface.Bridges.Constraint.NonnegToNonposBridge - Type.

NonnegToNonposBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

NonnegToNonposBridge implements the following reformulation:

• $f(x) \in \mathbb{R}_+$ into $-f(x) \in \mathbb{R}_-$

Source node

NonnegToNonposBridge supports:

• G in MOI.Nonnegatives

Target nodes

NonnegToNonposBridge creates:

• F in MOI.Nonpositives

source

MathOptInterface.Bridges.Constraint.NonposToNonnegBridge - Type.

NonposToNonnegBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

NonposToNonnegBridge implements the following reformulation:

• $f(x) \in \mathbb{R}_{-}$ into $-f(x) \in \mathbb{R}_{+}$

Source node

NonposToNonnegBridge supports:

• G in MOI.Nonpositives

Target nodes

NonposToNonnegBridge creates:

• F in MOI.Nonnegatives

```
source
```

MathOptInterface.Bridges.Constraint.NormInfinityBridge - Type.

NormInfinityBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

NormInfinityBridge implements the following reformulation:

• $|x|_{\infty} \leq t$ into $[t - x_i, t + x_i] \in \mathbb{R}_+$.

Source node

NormInfinityBridge supports:

• G in MOI.NormInfinityCone{T}

Target nodes

NormInfinityBridge creates:

• F in MOI.Nonnegatives

source

MathOptInterface.Bridges.Constraint.NormInfinityConeToNormConeBridge - Type.

NormInfinityConeToNormConeBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

NormInfinityConeToNormConeBridge implements the following reformulations:

• (t, x)inNormInfinityCone(d) into (t, x)inNormCone(Inf, d)

Source node

NormInfinityConeToNormConeBridge supports:

• F in MOI.NormInfinityCone

Target nodes

NormInfinityConeToNormConeBridge creates:

• F in MOI.NormCone

source

MathOptInterface.Bridges.Constraint.NormNuclearBridge - Type.

NormNuclearBridge{T,F,G,H} <: Bridges.Constraint.AbstractBridge</pre>

NormNuclearBridge implements the following reformulation:

•
$$t \ge \sum_i \sigma_i(X)$$
 into $\begin{bmatrix} U & X^\top \\ X & V \end{bmatrix} \succeq 0$ and $2t \ge tr(U) + tr(V)$.

Source node

NormNuclearBridge supports:

• H in MOI.NormNuclearCone

Target nodes

NormNuclearBridge creates:

- F in MOI.GreaterThan{T}
- G in MOI.PositiveSemidefiniteConeTriangle

source

MathOptInterface.Bridges.Constraint.NormOneBridge - Type.

NormOneBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

NormOneBridge implements the following reformulation:

• $\sum |x_i| \leq t$ into $[t - \sum y_i, y_i - x_i, y_i + x_i] \in \mathbb{R}_+$.

Source node

NormOneBridge supports:

• G in MOI.NormOneCone{T}

Target nodes

NormOneBridge creates:

• F in MOI.Nonnegatives

source

MathOptInterface.Bridges.Constraint.NormOneConeToNormConeBridge - Type.

NormOneConeToNormConeBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

NormOneConeToNormConeBridge implements the following reformulations:

• (t, x)inNormOneCone(d) into (t, x)inNormCone(1, d)

Source node

NormOneConeToNormConeBridge supports:

• F in MOI.NormOneCone

Target nodes

NormOneConeToNormConeBridge creates:

• F in MOI.NormCone

source

MathOptInterface.Bridges.Constraint.NormSpectralBridge - Type.

NormSpectralBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

NormSpectralBridge implements the following reformulation:

•
$$t \ge \sigma_1(X)$$
 into $\begin{bmatrix} t\mathbf{I} & X^\top \\ X & t\mathbf{I} \end{bmatrix} \succeq 0$

Source node

NormSpectralBridge supports:

• G in MOI.NormSpectralCone

Target nodes

NormSpectralBridge creates:

• F in MOI.PositiveSemidefiniteConeTriangle

source

MathOptInterface.Bridges.Constraint.NormToPowerBridge - Type.

NormToPowerBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

NormToPowerBridge implements the following reformulation:

•
$$(t,x) \in NormCone(p,1+d)$$
 into $(r_i,t,x_i) \in PowerCone(1/p)$ for all i , and $\sum_i r_i = t$.

For details, see Alizadeh, F., and Goldfarb, D. (2001). "Second-order cone programming." Mathematical Programming, Series B, 95:3-51.

Source node

NormToPowerBridge supports:

• F in MOI.NormCone

Target nodes

NormToPowerBridge creates:

- F in MOI.PowerCone{T}
- MOI.ScalarAffineFunction in MOI.EqualTo

source

MathOptInterface.Bridges.Constraint.NumberConversionBridge - Type.

NumberConversionBridge{T,F1,S1,F2,S2} <: Bridges.Constraint.AbstractBridge</pre>

NumberConversionBridge implements the following reformulation:

• $f1(x) \in S1$ to $f2(x) \in S2$

where f and S are the same functional form, but differ in their coefficient type.

Source node

NumberConversionBridge supports:

• F1 in S1

Target node

NumberConversionBridge creates:

• F2 in S2

source

MathOptInterface.Bridges.Constraint.QuadtoSOCBridge - Type.

QuadtoSOCBridge{T} <: Bridges.Constraint.AbstractBridge</pre>

QuadtoS0CBridge converts quadratic inequalities

$$\frac{1}{2}x^TQx + a^Tx \le ub$$

into MOI.RotatedSecondOrderCone constraints, but it only applies when Q is positive definite.

This is because, if Q is positive definite, there exists U such that $Q = U^T U$, and so the inequality can then be rewritten as;

$$||Ux||_2^2 \le 2(-a^Tx + ub)$$

Therefore, QuadtoS0CBridge implements the following reformulations:

- $\frac{1}{2}x^TQx + a^Tx \leq ub$ into $(1, -a^Tx + ub, Ux) \in RotatedSecondOrderCone$ where $Q = U^TU$
- $\frac{1}{2}x^TQx + a^Tx \ge lb$ into $(1, a^Tx lb, Ux) \in RotatedSecondOrderCone$ where $-Q = U^TU$

QuadtoSOCBridge supports:

- MOI.ScalarAffineFunction{T} in MOI.LessThan{T}
- MOI.ScalarAffineFunction{T} in MOI.GreaterThan{T}

Target nodes

RelativeEntropyBridge creates:

MOI.VectorAffineFunction{T} in MOI.RotatedSecondOrderCone

Errors

This bridge errors if Q is not positive definite.

```
source
```

MathOptInterface.Bridges.Constraint.RSOCtoNonConvexQuadBridge - Type.

RSOCtoNonConvexQuadBridge{T} <: Bridges.Constraint.AbstractBridge</pre>

RSOCtoNonConvexQuadBridge implements the following reformulations:

• $||x||_2^2 \le 2tu$ into $\sum x^2 - 2tu \le 0$, $1t + 0 \ge 0$, and $1u + 0 \ge 0$.

The MOI.ScalarAffineFunctions 1t + 0 and 1u + 0 are used in case the variables have other bound constraints.

Warning

This transformation starts from a convex constraint and creates a non-convex constraint. Unless the solver has explicit support for detecting rotated second-order cones in quadratic form, this may (wrongly) be interpreted by the solver as being non-convex. Therefore, this bridge is not added automatically by MOI.Bridges.full_bridge_optimizer. Care is recommended when adding this bridge to a optimizer.

Source node

RSOCtoNonConvexQuadBridge supports:

MOI.VectorOfVariables in MOI.RotatedSecondOrderCone

Target nodes

RSOCtoNonConvexQuadBridge creates:

- MOI.ScalarQuadraticFunction{T} in MOI.LessThan{T}
- MOI.ScalarAffineFunction{T} in MOI.GreaterThan{T}

source

MathOptInterface.Bridges.Constraint.RSOCtoPSDBridge - Type.

```
RSOCtoPSDBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>
```

RSOCtoPSDBridge implements the following reformulation:

| • $ x _2^2 \leq 2t \cdot u$ into | $\left[\begin{array}{c}t\\x\end{array}\right]$ | $\begin{array}{c} x^{\top} \\ 2tu \mathbf{I} \end{array}$ | ≥ 0 |
|------------------------------------|--|---|----------|
|------------------------------------|--|---|----------|

Source node

RSOCtoPSDBridge supports:

• G in MOI.RotatedSecondOrderCone

Target nodes

RSOCtoPSDBridge creates:

• F in MOI.PositiveSemidefiniteConeTriangle

source

MathOptInterface.Bridges.Constraint.RSOCtoSOCBridge - Type.

RSOCtoSOCBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

RS0CtoS0CBridge implements the following reformulation:

• $||x||_2^2 \le 2tu$ into $||\frac{t-u}{\sqrt{2}}, x||_2 \le \frac{t+u}{\sqrt{2}}$

Source node

RSOCtoSOCBridge supports:

• G in MOI.RotatedSecondOrderCone

Target node

RSOCtoSOCBridge creates:

• F in MOI.SecondOrderCone

source

MathOptInterface.Bridges.Constraint.ReifiedAllDifferentToCountDistinctBridge - Type.

```
ReifiedAllDifferentToCountDistinctBridge{T,F} <:
Bridges.Constraint.AbstractBridge</pre>
```

ReifiedAllDifferentToCountDistinctBridge implements the following reformulations:

- $r \iff x \in \mathsf{AllDifferent}(d)$ to $r \iff (n, x) \in \mathsf{CountDistinct}(1 + d)$ and n = d
- $r \iff f(x) \in \text{AllDifferent}(d)$ to $r \iff (d, f(x)) \in \text{CountDistinct}(1+d)$

ReifiedAllDifferentToCountDistinctBridge supports:

• F in MOI.Reified{MOI.AllDifferent}

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

ReifiedAllDifferentToCountDistinctBridge creates:

- F in MOI.Reified{MOI.CountDistinct}
- MOI.VariableIndex in MOI.EqualTo{T}

source

MathOptInterface.Bridges.Constraint.ReifiedCountDistinctToMILPBridge - Type.

ReifiedCountDistinctToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

ReifiedCountDistinctToMILPBridge implements the following reformulation:

• $r \iff (n, x) \in \text{CountDistinct}(1 + d)$ into a mixed-integer linear program.

Reformulation

The reformulation is non-trivial, and it depends on the finite domain of each variable x_i , which we as define $S_i = \{l_i, \ldots, u_i\}$.

First, we introduce new binary variables z_{ij} , which are 1 if variable x_i takes the value j in the optimal solution and 0 otherwise:

$$z_{ij} \in \{0,1\} \quad \forall i \in 1 \dots d, j \in S_i$$
$$x_i - \sum_{j \in S_i} j \cdot z_{ij} = 0 \quad \forall i \in 1 \dots d$$
$$\sum_{j \in S_i} z_{ij} = 1 \quad \forall i \in 1 \dots d$$

Then, we introduce new binary variables y_j , which are 1 if a variable takes the value j in the optimal solution and 0 otherwise.

$$y_j \in \{0,1\} \ \forall j \in \bigcup_{i=1,\dots,d} S_i$$
$$y_j \le \sum_{i \in 1\dots d: j \in S_i} z_{ij} \le M y_j \ \forall j \in \bigcup_{i=1,\dots,d} S_i$$

Finally, n is constrained to be the number of y_j elements that are non-zero, with some slack:

$$n - \sum_{j \in \bigcup_{i=1,\dots,d} S_i} y_j = \delta^+ - \delta^-$$

And then the slack is constrained to respect the reif variable r:

$$\begin{split} & d_1 \leq \delta^+ \leq M d_1 \\ & d_2 \leq \delta^- \leq M d_s \\ & d_1 + d_2 + r = 1 \\ & d_1, d_2 \in \{0, 1\} \end{split}$$

Source node

ReifiedCountDistinctToMILPBridge supports:

• F in MOI.Reified{MOI.CountDistinct}

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

ReifiedCountDistinctToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}
- MOI.ScalarAffineFunction{T} in MOI.LessThan{T}

source

MathOptInterface.Bridges.Constraint.RelativeEntropyBridge - Type.

RelativeEntropyBridge{T,F,G,H} <: Bridges.Constraint.AbstractBridge</pre>

RelativeEntropyBridge implements the following reformulation that converts a MOI.RelativeEntropyCone into an MOI.ExponentialCone:

•
$$u \ge \sum_{i=1}^n w_i \log\left(\frac{w_i}{v_i}\right)$$
 into $y_i \ge 0$, $u \ge \sum_{i=1}^n y_i$, and $(-y_i, w_i, v_i) \in ExponentialCone$

Source node

RelativeEntropyBridge supports:

• H in MOI.RelativeEntropyCone

Target nodes

RelativeEntropyBridge creates:

• F in MOI.GreaterThan{T}

• G in MOI.ExponentialCone

source

MathOptInterface.Bridges.Constraint.RootDetBridge - Type.

RootDetBridge{T,F,G,H} <: Bridges.Constraint.AbstractBridge</pre>

The MOI.RootDetConeTriangle is representable by MOI.PositiveSemidefiniteConeTriangle and MOI.GeometricMeanCone constraints, see [1, p. 149].

Indeed, $t \leq \det(X)^{1/n}$ if and only if there exists a lower triangular matrix such that:

$$\begin{pmatrix} X \\ \top & \text{Diag}() \end{pmatrix} \succeq 0 \\ (t, \text{Diag}()) \in GeometricMeanCone$$

Source node

RootDetBridge supports:

• I in MOI.RootDetConeTriangle

Target nodes

RootDetBridge creates:

- F in MOI.PositiveSemidefiniteConeTriangle
- G in MOI.GeometricMeanCone

[1] Ben-Tal, Aharon, and Arkadi Nemirovski. *Lectures on modern convex optimization: analysis, algorithms, and engineering applications*. Society for Industrial and Applied Mathematics, 2001.

source

MathOptInterface.Bridges.Constraint.SOCtoNonConvexQuadBridge - Type.

SOCtoNonConvexQuadBridge{T} <: Bridges.Constraint.AbstractBridge</pre>

SOCtoNonConvexQuadBridge implements the following reformulations:

• $||x||_2 \leq t$ into $\sum x^2 - t^2 \leq 0$ and $1t + 0 \geq 0$

The MOI.ScalarAffineFunction 1t + 0 is used in case the variable has other bound constraints.

Warning

This transformation starts from a convex constraint and creates a non-convex constraint. Unless the solver has explicit support for detecting second-order cones in quadratic form, this may (wrongly) be interpreted by the solver as being non-convex. Therefore, this bridge is not added automatically by MOI.Bridges.full_bridge_optimizer. Care is recommended when adding this bridge to a optimizer.

SOCtoNonConvexQuadBridge supports:

• MOI.VectorOfVariables in MOI.SecondOrderCone

Target nodes

SOCtoNonConvexQuadBridge creates:

- MOI.ScalarQuadraticFunction{T} in MOI.LessThan{T}
- MOI.ScalarAffineFunction{T} in MOI.GreaterThan{T}

source

MathOptInterface.Bridges.Constraint.SOCtoPSDBridge - Type.

SOCtoPSDBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

SOCtoPSDBridge implements the following reformulation:

 $\bullet \ ||x||_2 \leq t \text{ into } \left[\begin{array}{cc} t & x^\top \\ x & t\mathbf{I} \end{array} \right] \succeq 0$

Warning

This bridge is not added by default by MOI.Bridges.full_bridge_optimizer because bridging second order cone constraints to semidefinite constraints can be achieved by the SOCtoRSOCBridge followed by the RSOCtoPSDBridge, while creating a smaller semidefinite constraint.

Source node

SOCtoPSDBridge supports:

• G in MOI.SecondOrderCone

Target nodes

SOCtoPSDBridge creates:

• F in MOI.PositiveSemidefiniteConeTriangle

source

MathOptInterface.Bridges.Constraint.SOCtoRSOCBridge - Type.

SOCtoRSOCBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

SOCtoRSOCBridge implements the following reformulation:

• $||x||_2 \le t$ into $(t+x_1)(t-x_1) \ge ||(x_2...,x_N)||_2^2$

Assumptions

• SOCtoRSOCBridge assumes that the length of x is at least one.

Source node

SOCtoRSOCBridge supports:

• G in MOI.SecondOrderCone

Target node

SOCtoRSOCBridge creates:

• F in MOI.RotatedSecondOrderCone

source

MathOptInterface.Bridges.Constraint.SOS1ToMILPBridge - Type.

SOS1ToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

S0S1ToMILPBridge implements the following reformulation:

• $x \in SOS1(d)$ into a mixed-integer linear program.

Source node

SOS1ToMILPBridge supports:

• F in MOI.SOS1

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

SOS1ToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}
- MOI.ScalarAffineFunction{T} in MOI.LessThan{T}

source

MathOptInterface.Bridges.Constraint.SOS2ToMILPBridge - Type.

SOS2ToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

SOS2ToMILPBridge implements the following reformulation:

• $x \in SOS2(d)$ into a mixed-integer linear program.

Source node

SOS2ToMILPBridge supports:

• F in MOI.SOS2

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

SOS2ToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}
- MOI.ScalarAffineFunction{T} in MOI.LessThan{T}

source

MathOptInterface.Bridges.Constraint.ScalarFunctionizeBridge - Type.

```
ScalarFunctionizeBridge{T,S} =
→ FunctionConversionBridge{T,MOI.ScalarAffineFunction{T},MOI.VariableIndex,S}
```

ScalarFunctionizeBridge implements the following reformulations:

• $x \in S$ into $1x + 0 \in S$

Source node

ScalarFunctionizeBridge supports:

• MOI.VariableIndex in S

Target nodes

ScalarFunctionizeBridge creates:

• MOI.ScalarAffineFunction{T} in S

source

MathOptInterface.Bridges.Constraint.ScalarSlackBridge - Type.

ScalarSlackBridge{T,F,S} <: Bridges.Constraint.AbstractBridge</pre>

ScalarSlackBridge implements the following reformulation:

• $f(x) \in S$ into f(x) - y == 0 and $y \in S$

ScalarSlackBridge supports:

• G in S, where G is not MOI. VariableIndex and S is not MOI. EqualTo

Target nodes

ScalarSlackBridge creates:

- F in MOI.EqualTo{T}
- MOI.VariableIndex in S

source

MathOptInterface.Bridges.Constraint.ScalarizeBridge - Type.

ScalarizeBridge{T,F,S}

ScalarizeBridge implements the following reformulations:

- $f(x) a \in \mathbb{R}_+$ into $f_i(x) \ge a_i$ for all i
- $f(x) a \in \mathbb{R}_{-}$ into $f_i(x) \le a_i$ for all i
- $f(x) a \in \{0\}$ into $f_i(x) == a_i$ for all i

Source node

ScalarizeBridge supports:

- G in MOI.Nonnegatives{T}
- G in MOI.Nonpositives{T}
- G in MOI.Zeros{T}

Target nodes

ScalarizeBridge creates:

F in S, where S is one of MOI.GreaterThan{T}, MOI.LessThan{T}, and MOI.EqualTo{T}, depending
on the type of the input set.

source

MathOptInterface.Bridges.Constraint.SecondOrderConeToNormConeBridge - Type.

SecondOrderConeToNormConeBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

SecondOrderConeToNormConeBridge implements the following reformulations:

• (t, x)inSecondOrderCone(d) into (t, x)inNormCone(2, d)

SecondOrderConeToNormConeBridge supports:

• F in MOI.SecondOrderCone

Target nodes

SecondOrderConeToNormConeBridge creates:

• F in MOI.NormCone

source

MathOptInterface.Bridges.Constraint.SemiToBinaryBridge - Type.

SemiToBinaryBridge{T,S} <: Bridges.Constraint.AbstractBridge</pre>

SemiToBinaryBridge implements the following reformulations:

•
$$x \in \{0\} \cup [l, u]$$
 into

$$\begin{aligned} x &\leq zu \\ x &\geq zl \\ z &\in \{0,1\} \end{aligned}$$

•
$$x \in \{0\} \cup \{l, \dots, u\}$$
 into

$$x \le zu$$
$$x \ge zl$$
$$z \in \{0, 1\}$$
$$x \in \mathbb{Z}$$

Source node

SemiToBinaryBridge supports:

- MOI.VariableIndex in MOI.Semicontinuous{T}
- MOI.VariableIndex in MOI.Semiinteger{T}

Target nodes

SemiToBinaryBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in MOI.LessThan{T}
- MOI.ScalarAffineFunction{T} in MOI.GreaterThan{T}
- MOI.VariableIndex{T} in MOI.Integer (if S is MOI.Semiinteger{T}

source

MathOptInterface.Bridges.Constraint.SetConversionBridge - Type.

```
SetConversionBridge{T,S2,S1,F} <:
    MOI.Bridges.Constraint.SetMapBridge{T,S2,S1,F,F}
```

SetConversionBridge implements the following reformulations:

• $f(x) \in S1$ into $f(x) \in S2$

In order to add this bridge, you need to create a bridge specific for a given type T and set S2:

```
MOI.Bridges.add_bridge(model, MOI.Bridges.Constraint.SetConversionBridge{T,S2})
```

In order to define a bridge with S2 specified but T unspecified, for example for JuMP.add_bridge, you can use

```
const MyBridge{T,S1,F} = MOI.Bridges.Constraint.SetConversionBridge{T,S2,S1,F}
```

See also FunctionConversionBridge.

Source node

SetConversionBridge supports:

• F in S1

Target nodes

SetConversionBridge creates:

• F in S2

source

MathOptInterface.Bridges.Constraint.SetDotInverseScalingBridge - Type.

SetDotInverseScalingBridge{T,S,F,G} <: Bridges.Constraint.AbstractBridge</pre>

SetDotInverseScalingBridge implements the reformulation from constraints in the MOI.Scaled{S} to constraints in the S.

Source node

SetDotInverseScalingBridge supports:

• G in MOI.Scaled{S}

Target node

SetDotInverseScalingBridge creates:

• F in S

source

MathOptInterface.Bridges.Constraint.SetDotScalingBridge - Type.

SetDotScalingBridge{T,S,F,G} <: Bridges.Constraint.AbstractBridge</pre>

SetDotScalingBridge implements the reformulation from constraints in S to constraints in MOI.Scaled{S}.

Source node

SetDotScalingBridge supports:

• G in S

Target node

SetDotScalingBridge creates:

• F in MOI.Scaled{S}

source

MathOptInterface.Bridges.Constraint.SetMapBridge - Type.

abstract type SetMapBridge{T,S2,S1,F,G} <: MultiSetMapBridge{T,S1,G} end</pre>

Consider two type of sets, S1 and S2, and a linear mapping A such that the image of a set of type S1 under A is a set of type S2.

A SetMapBridge{T,S2,S1,F,G} is a bridge that maps G-in-S1 constraints into F-in-S2 by mapping the function through A.

The linear map A is described by;

- MOI.Bridges.map_set
- MOI.Bridges.map_function.

Implementing a method for these two functions is sufficient to bridge constraints. However, in order for the getters and setters of attributes such as dual solutions and starting values to work as well, a method for the following functions must be implemented:

- MOI.Bridges.inverse_map_set
- MOI.Bridges.inverse_map_function
- MOI.Bridges.adjoint_map_function
- MOI.Bridges.inverse_adjoint_map_function

See the docstrings of each function to see which feature would be missing if it was not implemented for a given bridge.

source
MathOptInterface.Bridges.Constraint.SplitComplexEqualToBridge - Type.

SplitComplexEqualToBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

SplitComplexEqualToBridge implements the following reformulation:

• f(x) + g(x) * im = a + b * im into f(x) = a and g(x) = b

Source node

SplitComplexEqualToBridge supports:

• G in MOI.EqualTo{Complex{T}}

where G is a function with Complex coefficients.

Target nodes

SplitComplexEqualToBridge creates:

• F in MOI.EqualTo{T}

where F is the type of the real/imaginary part of G.

source

MathOptInterface.Bridges.Constraint.SplitComplexZerosBridge - Type.

SplitComplexZerosBridge{T,F,G} <: Bridges.Constraint.AbstractBridge</pre>

SplitComplexZerosBridge implements the following reformulation:

• $f(x) \in \{0\}^n$ into $\operatorname{Re}(f(x)) \in \{0\}^n$ and $\operatorname{Im}(f(x)) \in \{0\}^n$

Source node

SplitComplexZerosBridge supports:

• G in MOI.Zeros

where G is a function with Complex coefficients.

Target nodes

SplitComplexZerosBridge creates:

• F in MOI.Zeros

where F is the type of the real/imaginary part of G.

source

MathOptInterface.Bridges.Constraint.SplitHyperRectangleBridge - Type.

SplitHyperRectangleBridge{T,G,F} <: Bridges.Constraint.AbstractBridge</pre>

SplitHyperRectangleBridge implements the following reformulation:

• $f(x) \in \text{HyperRectangle}(l, u)$ to $[f(x) - l; u - f(x)] \in \mathbb{R}_+$.

Source node

SplitHyperRectangleBridge supports:

• F in MOI.HyperRectangle

Target nodes

SplitHyperRectangleBridge creates:

• G in MOI.Nonnegatives

source

MathOptInterface.Bridges.Constraint.SplitIntervalBridge - Type.

SplitIntervalBridge{T,F,S,LS,US} <: Bridges.Constraint.AbstractBridge</pre>

SplitIntervalBridge implements the following reformulations:

- $l \leq f(x) \leq u$ into $f(x) \geq l$ and $f(x) \leq u$
- f(x) = b into $f(x) \ge b$ and $f(x) \le b$
- $f(x) \in \{0\}$ into $f(x) \in \mathbb{R}_+$ and $f(x) \in \mathbb{R}_-$

Source node

SplitIntervalBridge supports:

- F in MOI.Interval{T}
- F in MOI.EqualTo{T}
- F in MOI.Zeros

Target nodes

SplitIntervalBridge creates:

- F in MOI.LessThan{T}
- F in MOI.GreaterThan{T}

or

- F in MOI.Nonnegatives
- F in MOI.Nonpositives

Note

If T<:AbstractFloat and S is MOI.Interval{T} then no lower (resp. upper) bound constraint is created if the lower (resp. upper) bound is typemin(T) (resp. typemax(T)). Similarly, when MOI.ConstraintSet is set, a lower or upper bound constraint may be deleted or created accordingly.

source

MathOptInterface.Bridges.Constraint.SquareBridge - Type.

SquareBridge{T,F,G,TT,ST} <: Bridges.Constraint.AbstractBridge</pre>

SquareBridge implements the following reformulations:

- $(t, u, X) \in LogDetConeSquare$ into (t, u, Y)inLogDetConeTriangle
- $(t, X) \in RootDetConeSquare$ into (t, Y)inRootDetConeTriangle
- $\bullet \ X \in AbstractSymmetricMatrixSetSquare \ {\rm into}\ YinAbstractSymmetricMatrixSetTriangle$

where Y is the upper triangluar component of X.

In addition, constraints are added as necessary to constrain the matrix X to be symmetric. For example, the constraint for the matrix:

$$\begin{pmatrix} 1 & 1+x & 2-3x \\ 1+x & 2+x & 3-x \\ 2-3x & 2+x & 2x \end{pmatrix}$$

can be broken down to the constraint of the symmetric matrix

$$\begin{pmatrix} 1 & 1+x & 2-3x \\ \cdot & 2+x & 3-x \\ \cdot & \cdot & 2x \end{pmatrix}$$

and the equality constraint between the off-diagonal entries (2, 3) and (3, 2) 3-x == 2+x. Note that no symmetrization constraint needs to be added between the off-diagonal entries (1, 2) and (2, 1) or between (1, 3) and (3, 1) because the expressions are the same.

Source node

SquareBridge supports:

• F in ST

Target nodes

SquareBridge creates:

• G in TT

source

MathOptInterface.Bridges.Constraint.TableToMILPBridge - Type.

TableToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge</pre>

TableToMILPBridge implements the following reformulation:

• $x \in Table(t)$ into

$$z_j \in \{0, 1\} \quad \forall i, j$$
$$\sum_{j=1}^n z_j = 1$$
$$\sum_{j=1}^n t_{ij} z_j = x_i \quad \forall i$$

Source node

TableToMILPBridge supports:

• F in MOI.Table{T}

Target nodes

TableToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}

source

MathOptInterface.Bridges.Constraint.ToScalarNonlinearBridge - Type.

ToScalarNonlinearBridge{T,G,S} <: AbstractFunctionConversionBridge{G,S}</pre>

ToScalarNonlinearBridge implements the following reformulation:

• $g(x) \in S$ into $f(x) \in S$

where g is an abstract scalar function and f is a MOI.ScalarNonlinearFunction.

Source node

ToScalarNonlinearBridge supports:

• G<:AbstractScalarFunction in S

Target nodes

ToScalarNonlinearBridge creates:

• MOI.ScalarNonlinearFunction in S

source

MathOptInterface.Bridges.Constraint.ToScalarQuadraticBridge - Type.

ToScalarQuadraticBridge{T,G,S} <: AbstractFunctionConversionBridge{G,S}</pre>

ToScalarQuadraticBridge implements the following reformulation:

• $g(x) \in S$ into $f(x) \in S$

where g is an abstract scalar function and f is a MOI.ScalarQuadraticFunction.

Source node

ToScalarQuadraticBridge supports:

• G<:AbstractScalarFunction in S

Target nodes

ToScalarQuadraticBridge creates:

• MOI.ScalarQuadraticFunction in S

source

MathOptInterface.Bridges.Constraint.ToVectorQuadraticBridge - Type.

ToVectorQuadraticBridge{T,G,S} <: AbstractFunctionConversionBridge{G,S}</pre>

ToVectorQuadraticBridge implements the following reformulation:

• $g(x) \in S$ into $f(x) \in S$

where g is an abstract vector function and f is a MOI.VectorQuadraticFunction.

Source node

ToVectorQuadraticBridge supports:

• G<:AbstractVectorFunction in S

Target nodes

ToVectorQuadraticBridge creates:

• MOI.VectorQuadraticFunction in S

source

MathOptInterface.Bridges.Constraint.VectorFunctionizeBridge - Type.

VectorFunctionizeBridge{T,S} = FunctionConversionBridge{T,MOI.VectorAffineFunction{T},S}

VectorFunctionizeBridge implements the following reformulations:

• $x \in S$ into $Ix + 0 \in S$

Source node

VectorFunctionizeBridge supports:

• MOI.VectorOfVariables in S

Target nodes

VectorFunctionizeBridge creates:

• MOI.VectorAffineFunction{T} in S

source

MathOptInterface.Bridges.Constraint.VectorSlackBridge - Type.

VectorSlackBridge{T,F,S} <: Bridges.Constraint.AbstractBridge</pre>

VectorSlackBridge implements the following reformulation:

• $f(x) \in S$ into $f(x) - y \in \{0\}$ and $y \in S$

Source node

VectorSlackBridge supports:

• G in S, where G is not MOI.VectorOfVariables and S is not MOI.Zeros

Target nodes

VectorSlackBridge creates:

- F in MOI.Zeros
- MOI.VectorOfVariables in S

source

MathOptInterface.Bridges.Constraint.VectorizeBridge - Type.

VectorizeBridge{T,F,S,G} <: Bridges.Constraint.AbstractBridge</pre>

VectorizeBridge implements the following reformulations:

- $g(x) \ge a$ into $[g(x) a] \in \mathbb{R}_+$
- $g(x) \leq a$ into $[g(x) a] \in \mathbb{R}_{-}$
- g(x) == a into $[g(x) a] \in \{0\}$

where T is the coefficient type of g(x) - a.

Source node

VectorizeBridge supports:

- G in MOI.GreaterThan{T}
- G in MOI.LessThan{T}
- G in MOI.EqualTo{T}

Target nodes

VectorizeBridge creates:

• F in S, where S is one of MOI.Nonnegatives, MOI.Nonpositives, MOI.Zeros depending on the type of the input set.

source

MathOptInterface.Bridges.Constraint.ZeroOneBridge - Type.

ZeroOneBridge{T} <: Bridges.Constraint.AbstractBridge</pre>

ZeroOneBridge implements the following reformulation:

• $x \in \{0, 1\}$ into $x \in \mathbb{Z}$, $1x \in [0, 1]$.

Note

ZeroOneBridge adds a linear constraint instead of adding variable bounds to avoid conflicting with bounds set by the user.

Source node

ZeroOneBridge supports:

• MOI.VariableIndex in MOI.ZeroOne

Target nodes

ZeroOneBridge creates:

- MOI.VariableIndex in MOI.Integer
- MOI.ScalarAffineFunction{T} in MOI.Interval{T}

source

Objective bridges

These bridges are subtypes of Bridges.Objective.AbstractBridge.

MathOptInterface.Bridges.Objective.FunctionConversionBridge - Type.

```
FunctionConversionBridge{T,F,G} <: AbstractBridge</pre>
```

FunctionConversionBridge implements the following reformulations:

- $\min\{g(x)\}$ into $\min\{f(x)\}$
- $\max\{g(x)\}$ into $\max\{f(x)\}$

for these pairs of functions:

- MOI.ScalarAffineFunctionto [MOI.ScalarQuadraticFunction'](@ref)
- MOI.ScalarQuadraticFunction to MOI.ScalarNonlinearFunction
- MOI.VectorAffineFunction to MOI.VectorQuadraticFunction

Source node

FunctionConversionBridge supports:

MOI.ObjectiveFunction{G}

Target nodes

FunctionConversionBridge creates:

One objective node: MOI.ObjectiveFunction{F}

source

MathOptInterface.Bridges.Objective.FunctionizeBridge - Type.

FunctionizeBridge{T,G} <: FunctionConversionBridge{T,MOI.ScalarAffineFunction{T},G}</pre>

FunctionizeBridge implements the following reformulations:

- $\min\{x\}$ into $\min\{1x+0\}$
- $\max\{x\}$ into $\max\{1x+0\}$

where T is the coefficient type of 1 and 0.

Source node

FunctionizeBridge supports:

MOI.ObjectiveFunction{G}

Target nodes

FunctionizeBridge creates:

• One objective node: MOI.ObjectiveFunction{MOI.ScalarAffineFunction{T}}

source

MathOptInterface.Bridges.Objective.QuadratizeBridge - Type.

QuadratizeBridge{T,G} <: FunctionConversionBridge{T,MOI.ScalarQuadraticFunction{T},G}</pre>

QuadratizeBridge implements the following reformulations:

- $\min\{a^{\top}x+b\}$ into $\min\{x^{\top}\mathbf{0}x+a^{\top}x+b\}$
- max{ $a^{\top}x + b$ } into max{ $x^{\top}\mathbf{0}x + a^{\top}x + b$ }

where T is the coefficient type of 0.

Source node

QuadratizeBridge supports:

MOI.ObjectiveFunction{G}

Target nodes

QuadratizeBridge creates:

• One objective node: MOI.ObjectiveFunction{MOI.ScalarQuadraticFunction{T}}

source

MathOptInterface.Bridges.Objective.SlackBridge - Type.

SlackBridge{T,F,G}

SlackBridge implements the following reformulations:

- $\min\{f(x)\}$ into $\min\{y \mid f(x) y \le 0\}$
- $\max\{f(x)\}$ into $\max\{y \mid f(x) y \ge 0\}$

where F is the type of f(x) - y, G is the type of f(x), and T is the coefficient type of f(x).

Source node

SlackBridge supports:

MOI.ObjectiveFunction{G}

Target nodes

SlackBridge creates:

- One variable node: MOI.VariableIndex in MOI.Reals
- One objective node: MOI.ObjectiveFunction{MOI.VariableIndex}
- One constraint node, that depends on the MOI.ObjectiveSense:
 - F-in-MOI.LessThan if MIN_SENSE
 - F-in-MOI.GreaterThan if MAX_SENSE

Warning

When using this bridge, changing the optimization sense is not supported. Set the sense to MOI.FEASIBILITY_SENSE first to delete the bridge, then set MOI.ObjectiveSense and re-add the objective.

source

MathOptInterface.Bridges.Objective.VectorFunctionizeBridge - Type.

VectorFunctionizeBridge{T,G} <: FunctionConversionBridge{T,MOI.VectorAffineFunction{T},G}</pre>

VectorFunctionizeBridge implements the following reformulations:

- $\min\{x\}$ into $\min\{1x+0\}$
- $\max\{x\}$ into $\max\{1x+0\}$

where T is the coefficient type of 1 and 0.

Source node

VectorFunctionizeBridge supports:

MOI.ObjectiveFunction{G}

Target nodes

VectorFunctionizeBridge creates:

• One objective node: MOI.ObjectiveFunction{MOI.VectorAffineFunction{T}}

source

MathOptInterface.Bridges.Objective.VectorSlackBridge - Type.

VectorSlackBridge{T,F,G}

VectorSlackBridge implements the following reformulations:

- $\min\{f(x)\}$ into $\min\{y \mid y f(x) \in \mathbb{R}_+\}$
- $\max\{f(x)\}$ into $\max\{y \mid f(x) y \in \mathbb{R}_+\}$

where F is the type of f(x) - y, G is the type of f(x), and T is the coefficient type of f(x).

Source node

VectorSlackBridge supports:

MOI.ObjectiveFunction{G}

Target nodes

VectorSlackBridge creates:

- One variable node: MOI.VectorOfVariables in MOI.Reals
- One objective node: MOI.ObjectiveFunction{MOI.VectorOfVariables}
- One constraint node: F-in-MOI.Nonnegatives

Warning

When using this bridge, changing the optimization sense is not supported. Set the sense to MOI.FEASIBILITY_SENSE first to delete the bridge, then set MOI.ObjectiveSense and re-add the objective.

source

Variable bridges

These bridges are subtypes of Bridges.Variable.AbstractBridge.

MathOptInterface.Bridges.Variable.FlipSignBridge - Type.

abstract type FlipSignBridge{T,S1,S2} <: SetMapBridge{T,S2,S1} end</pre>

An abstract type that simplifies the creation of other bridges.

source

MathOptInterface.Bridges.Variable.FreeBridge - Type.

FreeBridge{T} <: Bridges.Variable.AbstractBridge</pre>

FreeBridge implements the following reformulation:

• $x \in \mathbb{R}$ into $y, z \ge 0$ with the substitution rule x = y - z,

where T is the coefficient type of y - z.

Source node

FreeBridge supports:

• MOI.VectorOfVariables in MOI.Reals

Target nodes

FreeBridge creates:

• One variable node: MOI.VectorOfVariables in MOI.Nonnegatives

source

MathOptInterface.Bridges.Variable.HermitianToSymmetricPSDBridge - Type.

HermitianToSymmetricPSDBridge{T} <: Bridges.Variable.AbstractBridge</pre>

HermitianToSymmetricPSDBridge implements the following reformulation:

• Hermitian positive semidefinite n x n complex matrix to a symmetric positive semidefinite 2n x 2n real matrix satisfying equality constraints described below.

Source node

HermitianToSymmetricPSDBridge supports:

MOI.VectorOfVariables in MOI.HermitianPositiveSemidefiniteConeTriangle

Target node

HermitianToSymmetricPSDBridge creates:

- MOI.VectorOfVariables in MOI.PositiveSemidefiniteConeTriangle
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}

Reformulation

The reformulation is best described by example.

The Hermitian matrix:

$$\begin{bmatrix} x_{11} & x_{12} + y_{12}im & x_{13} + y_{13}im \\ x_{12} - y_{12}im & x_{22} & x_{23} + y_{23}im \\ x_{13} - y_{13}im & x_{23} - y_{23}im & x_{33} \end{bmatrix}$$

is positive semidefinite if and only if the symmetric matrix:

| x_{11} | x_{12} | x_{13} | 0 | y_{12} | y_{13} |
|----------|----------|----------|-----------|-----------|----------|
| | x_{22} | x_{23} | $-y_{12}$ | 0 | y_{23} |
| | | x_{33} | $-y_{13}$ | $-y_{23}$ | 0 |
| | | | x_{11} | x_{12} | x_{13} |
| | | | | x_{22} | x_{23} |
| | | | | | x_{33} |

is positive semidefinite.

The bridge achieves this reformulation by adding a new set of variables in MOI.PositiveSemidefiniteConeTriangle(6), and then adding three groups of equality constraints to:

- constrain the two x blocks to be equal
- force the diagonal of the y blocks to be 0
- force the lower triangular of the y block to be the negative of the upper triangle.

source

MathOptInterface.Bridges.Variable.NonposToNonnegBridge - Type.

NonposToNonnegBridge{T} <: Bridges.Variable.AbstractBridge</pre>

NonposToNonnegBridge implements the following reformulation:

+ $x\in \mathbb{R}_-$ into $y\in \mathbb{R}_+$ with the substitution rule x=-y,

where T is the coefficient type of -y.

Source node

NonposToNonnegBridge supports:

• MOI.VectorOfVariables in MOI.Nonpositives

Target nodes

NonposToNonnegBridge creates:

• One variable node: MOI.VectorOfVariables in MOI.Nonnegatives,

source

MathOptInterface.Bridges.Variable.ParameterToEqualToBridge - Type.

ParameterToEqualToBridge{T} <: Bridges.Variable.AbstractBridge</pre>

ParameterToEqualToBridge implements the following reformulation:

• $x \in Parameter(v)$ into x == v

Source node

ParameterToEqualToBridge supports:

• MOI.VariableIndex in MOI.Parameter

Target nodes

ParameterToEqualToBridge creates:

• One variable node: MOI.VariableIndex in MOI.EqualTo{T}

source

MathOptInterface.Bridges.Variable.RSOCtoPSDBridge - Type.

RSOCtoPSDBridge{T} <: Bridges.Variable.AbstractBridge</pre>

RSOCtoPSDBridge implements the following reformulation:

• $||x||_2^2 \leq 2tu$ where $t, u \geq 0$ into $Y \succeq 0$, with the substitution rule: $Y = \begin{bmatrix} t & x^\top \\ x & 2u\mathbf{I} \end{bmatrix}$.

Additional bounds are added to ensure the off-diagonals of the 2uI submatrix are 0, and linear constraints are added to ensure the diagonal of 2uI takes the same values.

As a special case, if |x|| = 0, then RSOCtoPSDBridge reformulates into $(t, u) \in \mathbb{R}_+$.

Source node

RSOCtoPSDBridge supports:

• MOI.VectorOfVariables in MOI.RotatedSecondOrderCone

Target nodes

RSOCtoPSDBridge creates:

- One variable node that depends on the input dimension:
 - MOI.VectorOfVariables in MOI.Nonnegatives if dimension is 1 or 2
 - MOI.VectorOfVariables in
 - MOI.PositiveSemidefiniteConeTriangle otherwise
- The constraint node MOI.VariableIndex in MOI.EqualTo
- The constant node MOI.ScalarAffineFunction in MOI.EqualTo

source

MathOptInterface.Bridges.Variable.RSOCtoSOCBridge - Type.

RSOCtoSOCBridge{T} <: Bridges.Variable.AbstractBridge</pre>

RSOCtoSOCBridge implements the following reformulation:

• $||x||_2^2 \leq 2tu$ into $||v||_2 \leq w$, with the substitution rules $t = \frac{w}{\sqrt{2}} + \frac{v_1}{\sqrt{2}}$, $u = \frac{w}{\sqrt{2}} - \frac{v_1}{\sqrt{2}}$, and $x = (v_2, \ldots, v_N)$.

Source node

RSOCtoSOCBridge supports:

• MOI.VectorOfVariables in MOI.RotatedSecondOrderCone

Target node

RSOCtoSOCBridge creates:

MOI.VectorOfVariables in MOI.SecondOrderCone

source

MathOptInterface.Bridges.Variable.SOCtoRSOCBridge - Type.

SOCtoRSOCBridge{T} <: Bridges.Variable.AbstractBridge</pre>

SOCtoRSOCBridge implements the following reformulation:

• $||x||_2 \le t$ into $2uv \ge ||w||_2^2$, with the substitution rules $t = \frac{u}{\sqrt{2}} + \frac{v}{\sqrt{2}}$, $x = (\frac{u}{\sqrt{2}} - \frac{v}{\sqrt{2}}, w)$.

Assumptions

• SOCtoRSOCBridge assumes that $|x| \ge 1$.

Source node

SOCtoRSOCBridge supports:

• MOI.VectorOfVariables in MOI.SecondOrderCone

Target node

SOCtoRSOCBridge creates:

• MOI.VectorOfVariables in MOI.RotatedSecondOrderCone

source

MathOptInterface.Bridges.Variable.SetMapBridge - Type.

abstract type SetMapBridge{T,S1,S2} <: AbstractBridge end</pre>

Consider two type of sets, S1 and S2, and a linear mapping A such that the image of a set of type S1 under A is a set of type S2.

A SetMapBridge{T,S1,S2} is a bridge that substitutes constrained variables in S2 into the image through A of constrained variables in S1.

The linear map A is described by:

- MOI.Bridges.map_set
- MOI.Bridges.map_function

Implementing a method for these two functions is sufficient to bridge constrained variables. However, in order for the getters and setters of attributes such as dual solutions and starting values to work as well, a method for the following functions must be implemented:

- MOI.Bridges.inverse_map_set
- MOI.Bridges.inverse_map_function

- MOI.Bridges.adjoint_map_function
- MOI.Bridges.inverse_adjoint_map_function.

See the docstrings of each function to see which feature would be missing if it was not implemented for a given bridge.

Fieldnames

If S1 and S2 are [MOI.AbstractScalarSet], the struct must have fields:

- variable::MOI.VariableIndex
- constraint::MOI.ConstraintIndex{MOI.VariableIndex,S1}

If S1 and S2 are [MOI.AbstractVectorSet], the struct must have fields:

- variable::Vector{MOI.VariableIndex}
- constraint::MOI.ConstraintIndex{MOI.VectorOfVariables,S1}

source

MathOptInterface.Bridges.Variable.VectorizeBridge - Type.

VectorizeBridge{T,S} <: Bridges.Variable.AbstractBridge</pre>

VectorizeBridge implements the following reformulations:

- $x \ge a$ into $[y] \in \mathbb{R}_+$ with the substitution rule x = a + y
- $x \leq a$ into $[y] \in \mathbb{R}_{-}$ with the substitution rule x = a + y
- x == a into $[y] \in \{0\}$ with the substitution rule x = a + y

where T is the coefficient type of a + y.

Source node

VectorizeBridge supports:

- MOI.VariableIndex in MOI.GreaterThan{T}
- MOI.VariableIndex in MOI.LessThan{T}
- MOI.VariableIndex in MOI.EqualTo{T}

Target nodes

VectorizeBridge creates:

• One variable node: MOI.VectorOfVariables in S, where S is one of MOI.Nonnegatives, MOI.Nonpositives, MOI.Zeros depending on the type of S.

source

MathOptInterface.Bridges.Variable.ZerosBridge - Type.

ZerosBridge{T} <: Bridges.Variable.AbstractBridge</pre>

ZerosBridge implements the following reformulation:

• $x \in \{0\}$ into the substitution rule x = 0,

where T is the coefficient type of 0.

Source node

ZerosBridge supports:

MOI.VectorOfVariables in MOI.Zeros

Target nodes

ZerosBridge does not create target nodes. It replaces all instances of x with 0 via substitution. This means that no variables are created in the underlying model.

Caveats

The bridged variables are similar to parameters with zero values. Parameters with non-zero values can be created with constrained variables in MOI.EqualTo by combining a VectorizeBridge and this bridge.

However, functions modified by ZerosBridge cannot be unbridged. That is, for a given function, we cannot determine if the bridged variables were used.

A related implication is that this bridge does not support MOI.ConstraintDual. However, if a MOI.Utilities.CachingOptimi is used, the dual can be determined by the bridged optimizer using MOI.Utilities.get_fallback because the caching optimizer records the unbridged function.

source

API Reference

Bridges

AbstractBridge API

MathOptInterface.Bridges.AbstractBridge - Type.

abstract type AbstractBridge end

An abstract type representing a bridged constraint or variable in a MOI.Bridges.AbstractBridgeOptimizer.

All bridges must implement:

- added_constrained_variable_types
- added_constraint_types
- MOI.get(::AbstractBridge, ::MOI.NumberOfVariables)
- MOI.get(::AbstractBridge, ::MOI.ListOfVariableIndices)
- MOI.get(::AbstractBridge, ::MOI.NumberOfConstraints)

MOI.get(::AbstractBridge, ::MOI.ListOfConstraintIndices)

Subtypes of AbstractBridge may have additional requirements. Consult their docstrings for details.

In addition, all subtypes may optionally implement the following constraint attributes with the bridge in place of the constraint index:

- MOI.ConstraintDual
- MOI.ConstraintPrimal

source

MathOptInterface.Bridges.added_constrained_variable_types - Function.

```
added_constrained_variable_types(
    BT::Type{<:AbstractBridge},
)::Vector{Tuple{Type}}</pre>
```

Return a list of the types of constrained variables that bridges of concrete type BT add.

Implementation notes

- This method depends only on the type of the bridge, not the runtime value. If the bridge *may* add a constrained variable, the type *must* be included in the return vector.
- If the bridge adds a free variable via MOI.add_variable or MOI.add_variables, the return vector must include (MOI.Reals,).

Example

source

MathOptInterface.Bridges.added_constraint_types - Function.

```
added_constraint_types(
    BT::Type{<:AbstractBridge},
)::Vector{Tuple{Type,Type}}</pre>
```

Return a list of the types of constraints that bridges of concrete type BT add.

Implementation notes

• This method depends only on the type of the bridge, not the runtime value. If the bridge *may* add a constraint, the type *must* be included in the return vector.

Example

source

MathOptInterface.get - Method.

MOI.get(b::AbstractBridge, ::MOI.NumberOfVariables)::Int64

Return the number of variables created by the bridge b in the model.

See also MOI.NumberOfConstraints.

Implementation notes

• There is a default fallback, so you need only implement this if the bridge adds new variables.

source

MathOptInterface.get - Method.

MOI.get(b::AbstractBridge, ::MOI.ListOfVariableIndices)

Return the list of variables created by the bridge b.

See also MOI.ListOfVariableIndices.

Implementation notes

• There is a default fallback, so you need only implement this if the bridge adds new variables.

source

MathOptInterface.get - Method.

MOI.get(b::AbstractBridge, ::MOI.NumberOfConstraints{F,S})::Int64 where {F,S}

Return the number of constraints of the type F-in-S created by the bridge b.

See also MOI.NumberOfConstraints.

Implementation notes

 There is a default fallback, so you need only implement this for the constraint types returned by added_constraint_types.

source

MathOptInterface.get - Method.

MOI.get(b::AbstractBridge, ::MOI.ListOfConstraintIndices{F,S}) where {F,S}

Return a Vector{ConstraintIndex{F,S}} with indices of all constraints of type F-in-S created by the bride b.

See also MOI.ListOfConstraintIndices.

Implementation notes

 There is a default fallback, so you need only implement this for the constraint types returned by added_constraint_types.

source

MathOptInterface.Bridges.needs_final_touch - Function.

needs_final_touch(bridge::AbstractBridge)::Bool

Return whether final touch is implemented by bridge.

source

MathOptInterface.Bridges.final_touch - Function.

final_touch(bridge::AbstractBridge, model::MOI.ModelLike)::Nothing

A function that is called immediately prior to MOI.optimize! to allow bridges to modify their reformulations with respect to other variables and constraints in model.

For example, if the correctness of bridge depends on the bounds of a variable or the fact that variables are integer, then the bridge can implement final_touch to check assumptions immediately before a call to MOI.optimize!.

If you implement this method, you must also implement needs_final_touch.

source

MathOptInterface.Bridges.bridging_cost - Function.

bridging_cost(b::AbstractBridgeOptimizer, S::Type{<:MOI.AbstractSet}})</pre>

Return the cost of bridging variables constrained in S on creation, is_bridged(b, S) is assumed to be true.

```
bridging_cost(
    b::AbstractBridgeOptimizer,
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet},
)</pre>
```

Return the cost of bridging F-in-S constraints.

is_bridged(b, S) is assumed to be true.

source

MathOptInterface.Bridges.runtests - Function.

```
runtests(
    Bridge::Type{<:AbstractBridge},
    input_fn::Function,
    output_fn::Function;
    variable_start = 1.2,
    constraint_start = 1.2,
    eltype = Float64,
    cannot_unbridge::Bool = false,
)</pre>
```

Run a series of tests that check the correctness of Bridge.

input_fn and output_fn are functions such that input_fn(model) and output_fn(model) load the corresponding model into model.

Set cannot_unbridge to true if the bridge transformation is not invertible. If Bridge is a variable bridge this allows Variable.unbridged_map to returns nothing so that the tests allow errors that can be raised due to this. If Bridge is a constraint bridge this allows the getter of MOI.ConstraintFunction and MOI.ConstraintPrimalStart to throw MOI.GetAttributeNotAllowed.

Example

```
julia> MOI.Bridges.runtests(
        MOI.Bridges.Constraint.ZeroOneBridge,
        model -> MOI.add_constrained_variable(model, MOI.ZeroOne()),
        model -> begin
            x, _ = MOI.add_constrained_variable(model, MOI.Integer())
            MOI.add_constraint(model, 1.0 * x, MOI.Interval(0.0, 1.0))
        end,
        )
Test Summary: | Pass Total Time
Bridges.runtests | 32 32 0.8s
```

source

```
runtests(
    Bridge::Type{<:AbstractBridge},
    input::String,
    output::String;</pre>
```

```
variable_start = 1.2,
constraint_start = 1.2,
eltype = Float64,
)
```

Run a series of tests that check the correctness of Bridge.

input and output are models in the style of MOI.Utilities.loadfromstring!.

Example

```
julia> MOI.Bridges.runtests(
    MOI.Bridges.Constraint.ZeroOneBridge,
    """
    variables: x
    x in ZeroOne()
    """,
    """
    variables: x
    x in Integer()
    1.0 * x in Interval(0.0, 1.0)
    """,
    )
Test Summary: | Pass Total Time
Bridges.runtests | 32 32 0.0s
```

source

Constraint bridge API

MathOptInterface.Bridges.Constraint.AbstractBridge - Type.

abstract type AbstractBridge <: MOI.Bridges.AbstractType</pre>

Subtype of MOI.Bridges.AbstractBridge for constraint bridges.

In addition to the required implementation described in MOI.Bridges.AbstractBridge, subtypes of AbstractBridge must additionally implement:

- MOI.supports_constraint(::Type{<:AbstractBridge}, ::Type{<:MOI.AbstractFunction}, ::Type{<:MOI.AbstractFunction}
- concrete bridge type
- bridge_constraint

source

MathOptInterface.Bridges.Constraint.SingleBridgeOptimizer - Type.

SingleBridgeOptimizer{BT<:AbstractBridge}(model::MOI.ModelLike)</pre>

Return AbstractBridgeOptimizer that always bridges any objective function supported by the bridge BT.

This is in contrast with the MOI.Bridges.LazyBridgeOptimizer, which only bridges the objective function if it is supported by the bridge BT and unsupported by model.

Example

Implementation notes

All bridges should simplify the creation of SingleBridgeOptimizers by defining a constant that wraps the bridge in a SingleBridgeOptimizer.

```
julia> const MyNewBridgeModel{T,OT<:MOI.ModelLike} =
MOI.Bridges.Constraint.SingleBridgeOptimizer{MyNewBridge{T},OT};
```

This enables users to create bridged models as follows:

```
julia> MyNewBridgeModel{Float64}(MOI.Utilities.Model{Float64}());
```

source

MathOptInterface.supports_constraint - Method.

```
MOI.supports_constraint(
    BT::Type{<:AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet},
)::Bool</pre>
```

Return a Bool indicating whether the bridges of type BT support bridging F-in-S constraints.

Implementation notes

- This method depends only on the type of the inputs, not the runtime values.
- There is a default fallback, so you need only implement this method for constraint types that the bridge implements.

source

MathOptInterface.Bridges.Constraint.concrete_bridge_type - Function.

```
concrete_bridge_type(
   BT::Type{<:AbstractBridge},
   F::Type{<:MOI.AbstractFunction},
   S::Type{<:MOI.AbstractSet}
)::Type</pre>
```

Return the concrete type of the bridge supporting F-in-S constraints.

This function can only be called if MOI.supports_constraint(BT, F, S) is true.

Example

The SplitIntervalBridge bridges a MOI.VariableIndex-in-MOI.Interval constraint into a MOI.VariableIndex-in-MOI.GreaterThan and a MOI.VariableIndex-in-MOI.LessThan constraint.

source

MathOptInterface.Bridges.Constraint.bridge_constraint - Function.

```
bridge_constraint(
    BT::Type{<:AbstractBridge},
    model::MOI.ModelLike,
    func::AbstractFunction,
    set::MOI.AbstractSet,
)::BT</pre>
```

Bridge the constraint func-in-set using bridge BT to model and returns a bridge object of type BT.

Implementation notes

• The bridge type BT should be a concrete type, that is, all the type parameters of the bridge must be set.

source

MathOptInterface.Bridges.Constraint.add_all_bridges - Function.

add_all_bridges(model, ::Type{T}) where {T}

Add all bridges defined in the Bridges.Constraint submodule to model.

The coefficient type used is T.

source

MathOptInterface.Bridges.Constraint.conversion_cost - Function.

```
conversion_cost(
    F::Type{<:MOI.AbstractFunction},
    G::Type{<:MOI.AbstractFunction},
)::Float64</pre>
```

Return a Float64 returning the cost of converting any function of type G to a function of type F with convert.

This cost is used to compute MOI.Bridges.bridging_cost.

The default cost is Inf, which means that MOI.Bridges.Constraint.FunctionConversionBridge should not attempt the conversion.

source

Objective bridge API

MathOptInterface.Bridges.Objective.AbstractBridge - Type.

abstract type AbstractBridge <: MOI.Bridges.AbstractBridge end</pre>

Subtype of MOI.Bridges.AbstractBridge for objective bridges.

In addition to the required implementation described in MOI.Bridges.AbstractBridge, subtypes of AbstractBridge must additionally implement:

- supports_objective_function
- concrete_bridge_type
- bridge_objective
- MOI.Bridges.set_objective_function_type

source

MathOptInterface.Bridges.Objective.SingleBridgeOptimizer - Type.

SingleBridgeOptimizer{BT<:AbstractBridge}(model::MOI.ModelLike)</pre>

Return AbstractBridgeOptimizer that always bridges any objective function supported by the bridge BT.

This is in contrast with the MOI.Bridges.LazyBridgeOptimizer, which only bridges the objective function if it is supported by the bridge BT and unsupported by model.

Example

Implementation notes

All bridges should simplify the creation of SingleBridgeOptimizers by defining a constant that wraps the bridge in a SingleBridgeOptimizer.

```
julia> const MyNewBridgeModel{T,OT<:MOI.ModelLike} =
MOI.Bridges.Objective.SingleBridgeOptimizer{MyNewBridge{T},OT};
```

This enables users to create bridged models as follows:

julia> MyNewBridgeModel{Float64}(MOI.Utilities.Model{Float64}());

source

MathOptInterface.Bridges.Objective.supports_objective_function - Function.

```
supports_objective_function(
    BT::Type{<:MOI.Bridges.Objective.AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
)::Bool</pre>
```

Return a Bool indicating whether the bridges of type BT support bridging objective functions of type F.

Implementation notes

- This method depends only on the type of the inputs, not the runtime values.
- There is a default fallback, so you need only implement this method For objective functions that the bridge implements.

source

MathOptInterface.Bridges.set_objective_function_type - Function.

```
set_objective_function_type(
    BT::Type{<:Objective.AbstractBridge},
)::Type{<:MOI.AbstractFunction}</pre>
```

Return the type of objective function that bridges of concrete type BT set.

Implementation notes

• This method depends only on the type of the bridge, not the runtime value.

Example

MathOptInterface.Bridges.Objective.concrete_bridge_type - Function.

```
concrete_bridge_type(
    BT::Type{<:MOI.Bridges.Objective.AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
)::Type</pre>
```

Return the concrete type of the bridge supporting objective functions of type F.

This function can only be called if MOI.supports_objective_function(BT, F) is true.

source

MathOptInterface.Bridges.Objective.bridge_objective - Function.

```
bridge_objective(
    BT::Type{<:MOI.Bridges.Objective.AbstractBridge},
    model::MOI.ModelLike,
    func::MOI.AbstractFunction,
)::BT</pre>
```

Bridge the objective function func using bridge BT to model and returns a bridge object of type BT.

Implementation notes

• The bridge type BT must be a concrete type, that is, all the type parameters of the bridge must be set.

source

MathOptInterface.Bridges.Objective.add_all_bridges - Function.

```
add_all_bridges(model, ::Type{T}) where {T}
```

Add all bridges defined in the Bridges.Objective submodule to model.

The coefficient type used is T.

source

Variable bridge API

MathOptInterface.Bridges.Variable.AbstractBridge - Type.

abstract type AbstractBridge <: MOI.Bridges.AbstractBridge end</pre>

Subtype of MOI.Bridges.AbstractBridge for variable bridges.

In addition to the required implementation described in MOI.Bridges.AbstractBridge, subtypes of AbstractBridge must additionally implement:

CHAPTER 37. SUBMODULES

- supports_constrained_variable
- concrete_bridge_type
- bridge_constrained_variable

source

MathOptInterface.Bridges.Variable.SingleBridgeOptimizer - Type.

```
SingleBridgeOptimizer{BT<:AbstractBridge}(model::MOI.ModelLike)</pre>
```

Return MOI.Bridges.AbstractBridgeOptimizer that always bridges any variables constrained on creation supported by the bridge BT.

This is in contrast with the MOI.Bridges.LazyBridgeOptimizer, which only bridges the variables constrained on creation if they are supported by the bridge BT and unsupported by model.

Warning

Two SingleBridgeOptimizers cannot be used together as both of them assume that the underlying model only returns variable indices with nonnegative values. Use MOI.Bridges.LazyBridgeOptimizer instead.

Example

Implementation notes

All bridges should simplify the creation of SingleBridgeOptimizers by defining a constant that wraps the bridge in a SingleBridgeOptimizer.

```
julia> const MyNewBridgeModel{T,OT<:MOI.ModelLike} =
MOI.Bridges.Variable.SingleBridgeOptimizer{MyNewBridge{T},OT};
```

This enables users to create bridged models as follows:

```
julia> MyNewBridgeModel{Float64}(MOI.Utilities.Model{Float64}());
```

source

MathOptInterface.Bridges.Variable.supports_constrained_variable - Function.

```
supports_constrained_variable(
    BT::Type{<:AbstractBridge},
    S::Type{<:MOI.AbstractSet},
)::Bool</pre>
```

Return a Bool indicating whether the bridges of type BT support bridging constrained variables in S. That is, it returns true if the bridge of type BT converts constrained variables of type S into a form supported by the solver.

Implementation notes

- This method depends only on the type of the bridge and set, not the runtime values.
- There is a default fallback, so you need only implement this method for sets that the bridge implements.

Example

source

MathOptInterface.Bridges.Variable.concrete_bridge_type - Function.

```
concrete_bridge_type(
    BT::Type{<:AbstractBridge},
    S::Type{<:MOI.AbstractSet},
)::Type</pre>
```

Return the concrete type of the bridge supporting variables in S constraints.

This function can only be called if MOI.supports_constrained_variable(BT, S) is true.

Example

As a variable in MOI.GreaterThan is bridged into variables in MOI.Nonnegatives by the VectorizeBridge:

source

MathOptInterface.Bridges.Variable.bridge_constrained_variable - Function.

```
bridge_constrained_variable(
    BT::Type{<:AbstractBridge},
    model::MOI.ModelLike,
    set::MOI.AbstractSet,
)::BT</pre>
```

Bridge the constrained variable in set using bridge BT to model and returns a bridge object of type BT.

Implementation notes

• The bridge type BT must be a concrete type, that is, all the type parameters of the bridge must be set.

```
source
```

MathOptInterface.Bridges.Variable.add_all_bridges - Function.

```
add_all_bridges(model, ::Type{T}) where {T}
```

Add all bridges defined in the Bridges.Variable submodule to model.

The coefficient type used is T.

```
source
```

MathOptInterface.Bridges.Variable.unbridged_map - Function.

```
unbridged_map(
    bridge::MOI.Bridges.Variable.AbstractBridge,
    vi::MOI.VariableIndex,
)
```

For a bridged variable in a scalar set, return a tuple of pairs mapping the variables created by the bridge to an affine expression in terms of the bridged variable vi.

```
unbridged_map(
    bridge::MOI.Bridges.Variable.AbstractBridge,
    vis::Vector{MOI.VariableIndex},
)
```

For a bridged variable in a vector set, return a tuple of pairs mapping the variables created by the bridge to an affine expression in terms of the bridged variable vis. If this method is not implemented, it falls back to calling the following method for every variable of vis.

```
unbridged_map(
    bridge::MOI.Bridges.Variable.AbstractBridge,
    vi::MOI.VariableIndex,
    i::MOI.Bridges.IndexInVector,
)
```

For a bridged variable in a vector set, return a tuple of pairs mapping the variables created by the bridge to an affine expression in terms of the bridged variable vi corresponding to the ith variable of the vector.

If there is no way to recover the expression in terms of the bridged variable(s) vi(s), return nothing. See ZerosBridge for an example of bridge returning nothing.

source

AbstractBridgeOptimizer API

MathOptInterface.Bridges.AbstractBridgeOptimizer - Type.

```
abstract type AbstractBridgeOptimizer <: MOI.AbstractOptimizer end</pre>
```

An abstract type that implements generic functions for bridges.

Implementation notes

By convention, the inner optimizer should be stored in a model field. If not, the optimizer must implement MOI.optimize!.

source

MathOptInterface.Bridges.bridged_variable_function - Function.

```
bridged_variable_function(
    b::AbstractBridgeOptimizer,
    vi::MOI.VariableIndex,
)
```

Return a MOI.AbstractScalarFunction of variables of b.model that equals vi. That is, if the variable vi is bridged, it returns its expression in terms of the variables of b.model. Otherwise, it returns vi.

source

MathOptInterface.Bridges.unbridged_variable_function - Function.

```
unbridged_variable_function(
    b::AbstractBridgeOptimizer,
    vi::MOI.VariableIndex,
)
```

Return a MOI.AbstractScalarFunction of variables of b that equals vi. That is, if the variable vi is an internal variable of b.model created by a bridge but not visible to the user, it returns its expression in terms of the variables of bridged variables. Otherwise, it returns vi.

source

MathOptInterface.Bridges.bridged_function - Function.

bridged_function(b::AbstractBridgeOptimizer, value)::typeof(value)

Substitute any bridged MOI.VariableIndex in value by an equivalent expression in terms of variables of b.model.

source

MathOptInterface.Bridges.supports_constraint_bridges - Function.

supports_constraint_bridges(b::AbstractBridgeOptimizer)::Bool

Return a Bool indicating if b supports MOI.Bridges.Constraint.AbstractBridge.

source

MathOptInterface.Bridges.recursive_model - Function.

recursive_model(b::AbstractBridgeOptimizer)

If a variable, constraint, or objective is bridged, return the context of the inner variables. For most optimizers, this should be b.model.

source

MathOptInterface.Bridges.FirstBridge - Type.

struct FirstBridge <: MOI.AbstractConstraintAttribute end</pre>

Returns the first bridge used to bridge the constraint.

Warning

The indices of the bridge correspond to internal indices and may not correspond to indices of the model this attribute is got from.

source

LazyBridgeOptimizer API

MathOptInterface.Bridges.LazyBridgeOptimizer - Type.

LazyBridgeOptimizer(model::MOI.ModelLike)

The LazyBridgeOptimizer is a bridge optimizer that supports multiple bridges, and only bridges things which are not supported by the internal model.

Internally, the LazyBridgeOptimizer solves a shortest hyper-path problem to determine which bridges to use.

In general, you should use full_bridge_optimizer instead of this constructor because full_bridge_optimizer automatically adds a large number of supported bridges.

See also: add_bridge, remove_bridge, has_bridge and full_bridge_optimizer.

Example

```
julia> model = MOI.Bridges.LazyBridgeOptimizer(MOI.Utilities.Model{Float64}());
```

julia> MOI.Bridges.add_bridge(model, MOI.Bridges.Variable.FreeBridge{Float64})

```
julia> MOI.Bridges.has_bridge(model, MOI.Bridges.Variable.FreeBridge{Float64})
true
```

source

MathOptInterface.Bridges.full_bridge_optimizer - Function.

```
full_bridge_optimizer(model::MOI.ModelLike, ::Type{T}) where {T}
```

Returns a LazyBridgeOptimizer bridging model for every bridge defined in this package (see below for the few exceptions) and for the coefficient type T, as well as the bridges in the list returned by the ListOfNonstandardBridges attribute.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> bridged_model = MOI.Bridges.full_bridge_optimizer(model, Float64);

Exceptions

The following bridges are not added by full_bridge_optimizer, except if they are in the list returned by the ListOfNonstandardBridges attribute:

- Constraint.SOCtoNonConvexQuadBridge
- Constraint.RSOCtoNonConvexQuadBridge](@ref)
- Constraint.SOCtoPSDBridge
- If T is not a subtype of AbstractFloat, subtypes of Constraint.AbstractToIntervalBridge
 - Constraint.GreaterToIntervalBridge
 - Constraint.LessToIntervalBridge)

See the docstring of the each bridge for the reason they are not added.

source

MathOptInterface.Bridges.ListOfNonstandardBridges - Type.

ListOfNonstandardBridges{T}() <: MOI.AbstractOptimizerAttribute</pre>

Any optimizer can be wrapped in a LazyBridgeOptimizer using full_bridge_optimizer. However, by default LazyBridgeOptimizer uses a limited set of bridges that are:

- 1. implemented in MOI.Bridges
- 2. generally applicable for all optimizers.

For some optimizers however, it is useful to add additional bridges, such as those that are implemented in external packages (for example, within the solver package itself) or only apply in certain circumstances (for example, Constraint.SOCtoNonConvexQuadBridge).

Such optimizers should implement the ListOfNonstandardBridges attribute to return a vector of bridge types that are added by full_bridge_optimizer in addition to the list of default bridges.

Note that optimizers implementing ListOfNonstandardBridges may require package-specific functions or sets to be used if the non-standard bridges are not added. Therefore, you are recommended to use model = MOI.instantiate(Package.Optimizer; with_bridge_type = T) instead of model = MOI.instantiate(Package.Optimiz See MOI.instantiate.

Example

An optimizer using a non-default bridge in MOI.Bridges

Solvers supporting MOI.ScalarQuadraticFunction can support MOI.SecondOrderCone and MOI.RotatedSecondOrderCone by defining:

```
function MOI.get(::MyQuadraticOptimizer, ::ListOfNonstandardBridges{Float64})
    return Type[
        MOI.Bridges.Constraint.SOCtoNonConvexQuadBridge{Float64},
        MOI.Bridges.Constraint.RSOCtoNonConvexQuadBridge{Float64},
    ]
end
```

An optimizer defining an internal bridge

Suppose an optimizer can exploit specific structure of a constraint, for example, it can exploit the structure of the matrix A in the linear system of equations A * x = b.

The optimizer can define the function:

```
struct MatrixAffineFunction{T} <: MOI.AbstractVectorFunction
    A::SomeStructuredMatrixType{T}
    b::Vector{T}
end</pre>
```

and then a bridge

struct MatrixAffineFunctionBridge{T} <: MOI.Constraint.AbstractBridge
 # ...
end
...</pre>

from VectorAffineFunction{T} to the MatrixAffineFunction. Finally, it defines:

```
function MOI.get(::Optimizer{T}, ::ListOfNonstandardBridges{T}) where {T}
    return Type[MatrixAffineFunctionBridge{T}]
end
```

source

```
MathOptInterface.Bridges.add_bridge - Function.
```

add_bridge(b::LazyBridgeOptimizer, BT::Type{<:AbstractBridge})</pre>

Enable the use of the bridges of type BT by b.

source

MathOptInterface.Bridges.remove_bridge - Function.

remove_bridge(b::LazyBridgeOptimizer, BT::Type{<:AbstractBridge})</pre>

Disable the use of the bridges of type BT by b.

source

MathOptInterface.Bridges.has_bridge - Function.

has_bridge(b::LazyBridgeOptimizer, BT::Type{<:AbstractBridge})</pre>

Return a Bool indicating whether the bridges of type BT are used by b.

source

MathOptInterface.Bridges.print_active_bridges - Function.

print_active_bridges([io::IO=stdout,] b::MOI.Bridges.LazyBridgeOptimizer)

Print the set of bridges that are active in the model b.

source

```
print_active_bridges(
    [io::I0=stdout,]
    b::MOI.Bridges.LazyBridgeOptimizer,
    F::Type{<:MOI.AbstractFunction}
)</pre>
```

Print the set of bridges required for an objective function of type F.

source

```
print_active_bridges(
    [io::I0=stdout,]
    b::MOI.Bridges.LazyBridgeOptimizer,
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet},
)</pre>
```

Print the set of bridges required for a constraint of type F-in-S.

```
print_active_bridges(
    [io:::I0=stdout,]
    b::MOI.Bridges.LazyBridgeOptimizer,
    S::Type{<:MOI.AbstractSet}
)</pre>
```

Print the set of bridges required for a variable constrained to set S.

source

source

MathOptInterface.Bridges.print_graph - Function.

print_graph([io::IO = stdout,] b::LazyBridgeOptimizer)

Print the hyper-graph containing all variable, constraint, and objective types that could be obtained by bridging the variables, constraints, and objectives that are present in the model by all the bridges added to b.

Each node in the hyper-graph corresponds to a variable, constraint, or objective type.

- Variable nodes are indicated by []
- Constraint nodes are indicated by ()
- Objective nodes are indicated by | |

The number inside each pair of brackets is an index of the node in the hyper-graph.

Note that this hyper-graph is the full list of possible transformations. When the bridged model is created, we select the shortest hyper-path from this graph, so many nodes may be un-used.

To see which nodes are used, call print_active_bridges.

For more information, see Legat, B., Dowson, O., Garcia, J., and Lubin, M. (2020). "MathOptInterface: a data structure for mathematical optimization problems." URL

source

MathOptInterface.Bridges.debug_supports_constraint - Function.

```
debug_supports_constraint(
    b::LazyBridgeOptimizer,
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet};
    io::IO = Base.stdout,
)</pre>
```

Prints to io explanations for the value of MOI. supports_constraint with the same arguments.

source
MathOptInterface.Bridges.debug_supports - Function.

```
debug_supports(
    b::LazyBridgeOptimizer,
    ::MOI.ObjectiveFunction{F};
    io::I0 = Base.stdout,
) where F
```

Prints to io explanations for the value of MOI.supports with the same arguments.

source

SetMap API

MathOptInterface.Bridges.MapNotInvertible - Type.

```
struct MapNotInvertible <: Exception
    message::String
end</pre>
```

An error thrown by inverse_map_function or inverse_adjoint_map_function indicating that the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge is not invertible.

source

MathOptInterface.Bridges.map_set - Function.

```
map_set(bridge::MOI.Bridges.AbstractBridge, set)
map_set(::Type{BT}, set) where {BT}
```

Return the image of set through the linear map A defined in Variable. SetMapBridge and Constraint. SetMapBridge.

This function is used for bridging the constraint and setting the MOI.ConstraintSet.

source

MathOptInterface.Bridges.inverse_map_set - Function.

```
inverse_map_set(bridge::MOI.Bridges.AbstractBridge, set)
inverse_map_set(::Type{BT}, set) where {BT}
```

Return the preimage of set through the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge.

This function is used for getting the MOI.ConstraintSet.

The method can alternatively be defined on the bridge type. This legacy interface is kept for backward compatibility.

source

MathOptInterface.Bridges.map_function - Function.

```
map_function(bridge::MOI.Bridges.AbstractBridge, func)
map_function(::Type{BT}, func) where {BT}
```

Return the image of func through the linear map A defined in Variable. SetMapBridge and Constraint. SetMapBridge.

This function is used for getting the MOI.ConstraintPrimal of variable bridges. For constraint bridges, this is used for bridging the constraint, setting the MOI.ConstraintFunction and MOI.ConstraintPrimalStart and modifying the function with MOI.modify.

The default implementation of Constraint.bridge_constraint uses map_function with the bridge type so if this function is defined on the bridge type, Constraint.bridge_constraint does not need to be implemented.

source

map_function(::Type{BT}, func, i::IndexInVector) where {BT}

Return the scalar function at the ith index of the vector function that would be returned by map_function(BT, func) except that it may compute the ith element. This is used by bridged_function and for getting the MOI.VariablePrimal and MOI.VariablePrimalStart of variable bridges.

source

MathOptInterface.Bridges.inverse_map_function - Function.

```
inverse_map_function(bridge::MOI.Bridges.AbstractBridge, func)
inverse_map_function(::Type{BT}, func) where {BT}
```

Return the image of func through the inverse of the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge.

This function is used by Variable.unbridged_map and for setting the MOI.VariablePrimalStart of variable bridges and for getting the MOI.ConstraintFunction, the MOI.ConstraintPrimal and the MOI.ConstraintPrimalStart of constraint bridges.

If the linear map A is not invertible, the error MapNotInvertible is thrown.

The method can alternatively be defined on the bridge type. This legacy interface is kept for backward compatibility.

source

MathOptInterface.Bridges.adjoint_map_function - Function.

```
adjoint_map_function(bridge::MOI.Bridges.AbstractBridge, func)
adjoint_map_function(::Type{BT}, func) where {BT}
```

Return the image of func through the adjoint of the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge.

This function is used for getting the MOI.ConstraintDual and MOI.ConstraintDualStart of constraint bridges.

The method can alternatively be defined on the bridge type. This legacy interface is kept for backward compatibility.

source

MathOptInterface.Bridges.inverse_adjoint_map_function - Function.

```
inverse_adjoint_map_function(bridge::MOI.Bridges.AbstractBridge, func)
inverse_adjoint_map_function(::Type{BT}, func) where {BT}
```

Return the image of func through the inverse of the adjoint of the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge.

This function is used for getting the MOI. ConstraintDual of variable bridges and setting the MOI. ConstraintDualStart of constraint bridges.

If the linear map A is not invertible, the error MapNotInvertible is thrown.

The method can alternatively be defined on the bridge type. This legacy interface is kept for backward compatibility.

source

Bridging graph API

MathOptInterface.Bridges.Graph - Type.

Graph()

A type-stable datastructure for computing the shortest hyperpath problem.

Nodes

There are three types of nodes in the graph:

- VariableNode
- ConstraintNode
- ObjectiveNode

Add nodes to the graph using add_node.

Edges

There are two types of edges in the graph:

- Edge
- ObjectiveEdge

Add edges to the graph using add_edge.

For the ability to add a variable constrained on creation as a free variable followed by a constraint, use set_variable_constraint_node.

Optimal hyper-edges

Use bridge_index to compute the minimum-cost bridge leaving a node.

Note that bridge_index lazy runs a Bellman-Ford algorithm to compute the set of minimum cost edges.
Thus, the first call to bridge_index after adding new nodes or edges will take longer than subsequent
calls.

source

MathOptInterface.Bridges.VariableNode - Type.

VariableNode(index::Int)

A node in Graph representing a variable constrained on creation.

source

MathOptInterface.Bridges.ConstraintNode - Type.

ConstraintNode(index::Int)

A node in Graph representing a constraint.

source

MathOptInterface.Bridges.ObjectiveNode - Type.

ObjectiveNode(index::Int)

A node in Graph representing an objective function.

source

MathOptInterface.Bridges.Edge - Type.

```
Edge(
    bridge_index::Int,
    added_variables::Vector{VariableNode},
    added_constraints::Vector{ConstraintNode},
    cost::Float64 = 1.0,
)
```

Return a new datastructure representing an edge in Graph that starts at a VariableNode or a ConstraintNode.

source

MathOptInterface.Bridges.ObjectiveEdge - Type.

```
ObjectiveEdge(
    bridge_index::Int,
    added_variables::Vector{VariableNode},
    added_constraints::Vector{ConstraintNode},
)
```

Return a new datastructure representing an edge in Graph that starts at an ObjectiveNode.

source

MathOptInterface.Bridges.add node - Function.

```
add_node(graph::Graph, ::Type{VariableNode})::VariableNode
add_node(graph::Graph, ::Type{ConstraintNode})::ConstraintNode
add_node(graph::Graph, ::Type{ObjectiveNode})::ObjectiveNode
```

Add a new node to graph.

source

MathOptInterface.Bridges.add_edge - Function.

```
add_edge(graph::Graph, node::VariableNode, edge::Edge)::Nothing
add_edge(graph::Graph, node::ConstraintNode, edge::Edge)::Nothing
add_edge(graph::Graph, node::ObjectiveNode, edge::ObjectiveEdge)::Nothing
```

Add edge to graph, where edge starts at node and connects to the nodes defined in edge.

source

MathOptInterface.Bridges.set_variable_constraint_node - Function.

```
set_variable_constraint_node(
    graph::Graph,
    variable_node::VariableNode,
    constraint_node::ConstraintNode,
    cost::Int,
)
```

As an alternative to variable_node, add a virtual edge to graph that represents adding a free variable, followed by a constraint of type constraint_node, with bridging cost cost.

Why is this needed?

Variables can either be added as a variable constrained on creation, or as a free variable which then has a constraint added to it.

source

MathOptInterface.Bridges.bridge_index - Function.

```
bridge_index(graph::Graph, node::VariableNode)::Int
bridge_index(graph::Graph, node::ConstraintNode)::Int
bridge_index(graph::Graph, node::ObjectiveNode)::Int
```

Return the optimal index of the bridge to chose from node.

source

MathOptInterface.Bridges.is_variable_edge_best - Function.

```
is variable edge best(graph::Graph, node::VariableNode)::Bool
```

Return a Bool indicating whether node should be added as a variable constrained on creation, or as a free variable followed by a constraint.

source

37.3 FileFormats

Overview

The FileFormats submodule

The FileFormats module provides functions for reading and writing MOI models using write_to_file and read_from_file.

Supported file types

You must read and write files to a FileFormats.Model object. Specific the file-type by passing a FileFormats.FileFormat enum. For example:

The Conic Benchmark Format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_CBF)
```

MOI.FileFormats.CBF.Model

```
bjectiveSense: FEASIBILITY_SENSE
```

- bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
- NumberOfVariables: 0
- L NumberOfConstraints: 0

The LP file format

- bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
- NumberOfVariables: 0
- L NumberOfConstraints: 0

The MathOptFormat file format

julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MOF)

MOI.FileFormats.MOF.Model

```
bjectiveSense: FEASIBILITY_SENSE
```

```
bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
```

```
- NumberOfVariables: 0
```

```
L NumberOfConstraints: 0
```

The MPS file format

julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MPS)

MOI.FileFormats.MPS.Model

ObjectiveSense: FEASIBILITY_SENSE

- bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
- NumberOfVariables: 0
- L NumberOfConstraints: 0

The NL file format

julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_NL)

MOI.FileFormats.NL.Model

bjectiveSense: unknown

- bjectiveFunctionType: unknown
- NumberOfVariables: unknown
- L NumberOfConstraints: unknown

The REW file format

julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_REW)

- MOI.FileFormats.MPS.Model
- bjectiveSense: FEASIBILITY_SENSE
- bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
- NumberOfVariables: 0
- L NumberOfConstraints: 0

Note that the REW format is identical to the MPS file format, except that all names are replaced with generic identifiers.

The SDPA file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_SDPA)
MOI.FileFormats.SDPA.Model
    ObjectiveSense: FEASIBILITY_SENSE
    ObjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
    NumberOfVariables: 0
    L NumberOfConstraints: 0
```

Write to file

To write a model src to a MathOptFormat file, use:

```
julia> src = MOI.Utilities.Model{Float64}();
```

julia> MOI.add_variable(src);

julia> dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MOF);

julia> MOI.copy_to(dest, src)

MathOptInterface.Utilities.IndexMap with 1 entry: MOI.VariableIndex(1) => MOI.VariableIndex(1)

julia> MOI.write_to_file(dest, "file.mof.json")

julia> print(read("file.mof.json", String))

Read from file

To read a MathOptFormat file, use:

julia> dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MOF);

julia> MOI.read_from_file(dest, "file.mof.json")

julia> MOI.get(dest, MOI.ListOfVariableIndices())
1-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(1)

julia> rm("file.mof.json") # Clean up after ourselves.

Detecting the file-type automatically

Instead of the format keyword, you can also use the filename keyword argument to FileFormats.Model. This will attempt to automatically guess the format from the file extension. For example:

```
julia> src = MOI.Utilities.Model{Float64}();
julia> dest = MOI.FileFormats.Model(filename = "file.cbf.gz");
julia> MOI.copy_to(dest, src)
MathOptInterface.Utilities.IndexMap()
julia> MOI.write_to_file(dest, "file.cbf.gz")
julia> src_2 = MOI.FileFormats.Model(filename = "file.cbf.gz");
julia> MOI.read_from_file(src_2, "file.cbf.gz")
julia> rm("file.cbf.gz") # Clean up after ourselves.
```

Note how the compression format (GZip) is also automatically detected from the filename.

Unsupported constraints

In some cases src may contain constraints that are not supported by the file format (for example, the CBF format supports integer variables but not binary). If so, copy src to a bridged model using Bridges.full_bridge_optimizer:

```
julia> src = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variable(src);

julia> MOI.add_constraint(src, x, MOI.ZeroOne());

julia> dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_CBF);

julia> bridged = MOI.Bridges.full_bridge_optimizer(dest, Float64);

julia> MOI.copy_to(bridged, src);

julia> MOI.write_to_file(dest, "my_model.cbf")

julia> rm("my_model.cbf") # Clean up after ourselves.

Note

Even after bridging, it may still not be possible to write the model to file because of unsupported constraints (for example, PSD variables in the LP file format).

Read and write to io

In addition to write_to_file and read_from_file, you can read and write directly from IO streams using Base.write and Base.read!:

```
julia> src = MOI.Utilities.Model{Float64}();
```

julia> dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MPS);

julia> MOI.copy_to(dest, src)
MathOptInterface.Utilities.IndexMap()

julia> io = IOBuffer();

julia> write(io, dest)

```
julia> seekstart(io);
```

julia> src_2 = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MPS);

julia> read!(io, src_2);

ScalarNonlinearFunction

By default, reading a .nl or .mof.json that contains nonlinear expressions will create an NLPBlock.

To instead read nonlinear expressions as ScalarNonlinearFunction, pass the use_nlp_block = false keyword argument to the Model constructor:

```
julia> model = MOI.FileFormats.Model(;
    format = MOI.FileFormats.FORMAT_MOF,
    use_nlp_block = false,
    );
julia> model = MOI.FileFormats.Model(;
    format = MOI.FileFormats.FORMAT_NL,
    use_nlp_block = false,
    );
```

Validating MOF files

MathOptFormat files are governed by a schema. Use JSONSchema.jl to check if a .mof.json file satisfies the schema.

First, construct the schema object as follows:

julia> import JSON, JSONSchema

julia> schema = JSONSchema.Schema(JSON.parsefile(MOI.FileFormats.MOF.SCHEMA_PATH))
A JSONSchema

Then, check if a model file is valid using isvalid:

```
julia> good_model = JSON.parse("""
    {
        "version": {
            "major": 1,
            "minor": 5
        },
        "variables": [{"name": "x"}],
        "objective": {"sense": "feasibility"},
        "constraints": []
    }
    """);
julia> isvalid(schema, good_model)
true
```

If we construct an invalid file, for example by mis-typing name as NaMe, the validation fails:

```
julia> bad_model = JSON.parse("""
    {
        "version": {
            "major": 1,
            "minor": 5
        },
        "variables": [{"NaMe": "x"}],
        "objective": {"sense": "feasibility"},
        "constraints": []
    }
    """);
```

```
julia> isvalid(schema, bad_model)
false
```

Use JSONSchema.validate to obtain more insight into why the validation failed:

```
julia> JSONSchema.validate(schema, bad_model)
Validation failed:
path: [variables][1]
instance: Dict{String, Any}("NaMe" => "x")
schema key: required
schema value: Any["name"]
```

API Reference

File Formats

Functions to help read and write MOI models to/from various file formats. See The FileFormats submodule for more details.

MathOptInterface.FileFormats.Model - Function.

```
Model(;
    format::FileFormat = FORMAT_AUTOMATIC,
    filename::Union{Nothing, String} = nothing,
    kwargs...
)
```

Return model corresponding to the FileFormats.FileFormat format, or, if format == FORMAT_AUTOMATIC, guess the format from filename.

The filename argument is only needed if format == FORMAT_AUTOMATIC.

kwargs are passed to the underlying model constructor.

source

MathOptInterface.FileFormats.FileFormat - Type.

FileFormat

List of accepted export formats.

Values

FORMAT_AUTOMATIC

Try to detect the file format based on the file name.

FORMAT_CBF

The Conic Benchmark format.

See FileFormats.CBF.Model for more details.

FORMAT_LP

The LP file format.

See FileFormats.LP.Model for more details.

FORMAT_MOF

The MathOptFormat file format.

See FileFormats.MOF.Model for more details.

FORMAT_MPS

The MPS file format.

See FileFormats.MPS.Model for more details.

FORMAT_NL

The AMPL .nl file format.

See FileFormats.NL.Model for more details.

FORMAT_REW

The .rew file format, which is equivalent to the MPS format (FileFormats.FORMAT_MPS) with the generic_names = true keyword argument set by default.

See FileFormats.MPS.Model for more details.

FORMAT_SDPA

The SemiDefinite Programming Algorithm format.

See FileFormats.SDPA.Model for more details.

source

MathOptInterface.FileFormats.FORMAT_AUTOMATIC - Constant.

FORMAT_AUTOMATIC::FileFormat

An instance of the FileFormat enum.

About

Try to detect the file format based on the file name.

source

MathOptInterface.FileFormats.FORMAT_CBF - Constant.

FORMAT_CBF::FileFormat

An instance of the FileFormat enum.

About

The Conic Benchmark format.

See FileFormats.CBF.Model for more details.

source

MathOptInterface.FileFormats.CBF.Model - Type.

Model()

Create an empty instance of FileFormats.CBF.Model.

source

MathOptInterface.FileFormats.FORMAT_LP - Constant.

FORMAT_LP::FileFormat

An instance of the FileFormat enum.

About

The LP file format.

See FileFormats.LP.Model for more details.

source

MathOptInterface.FileFormats.LP.Model - Type.

Model(; kwargs...)

Create an empty instance of FileFormats.LP.Model.

Keyword arguments are:

- maximum_length::Int=255: the maximum length for the name of a variable. lp_solve 5.0 allows only 16 characters, while CPLEX 12.5+ allow 255.
- warn::Bool=false: print a warning when variables or constraints are renamed.

source

MathOptInterface.FileFormats.FORMAT_MOF - Constant.

FORMAT_MOF::FileFormat

An instance of the FileFormat enum.

About

The MathOptFormat file format.

See FileFormats.MOF.Model for more details.

source

MathOptInterface.FileFormats.MOF.Model - Type.

Model(; kwargs...)

Create an empty instance of FileFormats.MOF.Model.

Keyword arguments are:

- print_compact::Bool=false: print the JSON file in a compact format without spaces or newlines.
- warn::Bool=false: print a warning when variables or constraints are renamed
- differentiation_backend::MOI.Nonlinear.AbstractAutomaticDifferentiation = MOI.Nonlinear.SparseRever automatic differentiation backend to use when reading models with nonlinear constraints and objectives.
- use_nlp_block::Bool=true: if true parse "ScalarNonlinearFunction" into an MOI.NLPBlock. If false, "ScalarNonlinearFunction" are parsed as MOI.ScalarNonlinearFunction functions.

source

MathOptInterface.FileFormats.FORMAT_MPS - Constant.

FORMAT_MPS::FileFormat

An instance of the FileFormat enum.

About

The MPS file format.

See FileFormats.MPS.Model for more details.

source

MathOptInterface.FileFormats.MPS.Model - Type.

Model(; kwargs...)

Create an empty instance of FileFormats.MPS.Model.

Keyword arguments are:

- warn::Bool=false: print a warning when variables or constraints are renamed.
- print_objsense::Bool=false: print the OBJSENSE section when writing
- generic_names::Bool=false: strip all names in the model and replace them with the generic names C\$i and R\$i for the i'th column and row respectively.
- quadratic_format::QuadraticFormat = kQuadraticFormatGurobi: specify the solver-specific extension used when writing the quadratic components of the model. Options are kQuadraticFormatGurobi, kQuadraticFormatCPLEX, and kQuadraticFormatMosek.

source

MathOptInterface.FileFormats.FORMAT_NL - Constant.

FORMAT_NL::FileFormat

An instance of the FileFormat enum.

About

The AMPL .nl file format.

See FileFormats.NL.Model for more details.

source

MathOptInterface.FileFormats.NL.Model - Type.

Model(; use_nlp_block::Bool = true)

Create a new Optimizer object.

source

MathOptInterface.FileFormats.FORMAT_REW - Constant.

FORMAT_REW::FileFormat

An instance of the FileFormat enum.

About

The .rew file format, which is equivalent to the MPS format (FileFormats.FORMAT_MPS) with the generic_names = true keyword argument set by default.

See FileFormats.MPS.Model for more details.

source

MathOptInterface.FileFormats.FORMAT_SDPA - Constant.

FORMAT_SDPA::FileFormat

An instance of the FileFormat enum.

About

The SemiDefinite Programming Algorithm format.

See FileFormats.SDPA.Model for more details.

source

MathOptInterface.FileFormats.SDPA.Model - Type.

Model(; number_type::Type = Float64)

Create an empty instance of FileFormats.SDPA.Model{number type}.

It is important to be aware that the SDPA file format is interpreted in *geometric* form and not *standard conic* form. The *standard conic* form and *geometric conic* form are two dual standard forms for semidefinite programs (SDPs). The *geometric conic* form of an SDP is as follows:

$$\min_{y \in \mathbb{R}^m} b^T y$$
s.t.
$$\sum_{i=1}^m A_i y_i - C \in \mathbb{K}$$
(37.2)

where \mathcal{K} is a cartesian product of nonnegative orthant and positive semidefinite matrices that align with a block diagonal structure shared with the matrices A i and C.

In other words, the geometric conic form contains free variables and affine constraints in either the nonnegative orthant or the positive semidefinite cone. That is, in the MathOptInterface's terminology, MOI.VectorAffineFunctionin-MOI.Nonnegatives and MOI.VectorAffineFunction-in-MOI.PositiveSemidefiniteConeTriangle constraints.

 $\overline{i=1}$

The corresponding *standard conic* form of the dual SDP is as follows:

$$\max_{X \in \mathbb{K}} \operatorname{tr}(CX)$$
(37.3)
s.t.
$$\operatorname{tr}(A;X) = b_i \qquad i = 1, \dots, m.$$
(37.4)

In other words, the standard conic form contains nonnegative and positive semidefinite variables with equality constraints. That is, in the MathOptInterface's terminology, MOI.VectorOfVariables-in-MOI.Nonnegatives, MOI.VectorOfVariables-in-MOI.PositiveSemidefiniteConeTriangle and MOI.ScalarAffineFunction-in-MOI.EqualTo constraints.

If a model is in standard conic form, use Dualization.jl to transform it into the geometric conic form before writting it. Otherwise, the nonnegative (resp. positive semidefinite) variables will be bridged into free variables with affine constraints constraining them to belong to the nonnegative orthant (resp. positive semidefinite cone) by the MOI.Bridges.Constraint.VectorFunctionizeBridge. Moreover, equality constraints will be bridged into pairs of affine constraints in the nonnegative orthant by the MOI.Bridges.Constraint.SplitInte and then the MOI.Bridges.Constraint.VectorizeBridge.

If a solver is in standard conic form, use Dualization.jl to transform the model read into standard conic form before copying it to the solver. Otherwise, the free variables will be bridged into pairs of variables in the nonnegative orthant by the MOI.Bridges.Variable.FreeBridge and affine constraints will be bridged into equality constraints by creating a slack variable by the MOI.Bridges.Constraint.VectorSlackBridge.

source

Other helpers

MathOptInterface.FileFormats.NL.SolFileResults - Type.

```
SolFileResults(
   filename::String,
   model::Model;
   suffix_lower_bound_duals::Vector{String} =
      ["ipopt_zL_out", "lower_bound_duals"],
   suffix_uuper_bound_duals::Vector{String} =
      ["ipopt_zU_out", "upper_bound_duals"],
)
```

Parse the .sol file filename created by solving model and return a SolFileResults struct.

The returned struct supports the MOI.get API for querying result attributes such as MOI.TerminationStatus, MOI.VariablePrimal, and MOI.ConstraintDual.

source

```
SolFileResults(
    raw_status::String,
    termination_status::MOI.TerminationStatusCode,
)
```

Return a SolFileResults struct with MOI.RawStatusString set to raw_status, MOI.TerminationStatus set to termination_status, and MOI.PrimalStatus and MOI.DualStatus set to NO_SOLUTION.

All other attributes are un-set.

source

37.4 Nonlinear

Overview

Nonlinear

The Nonlinear submodule contains data structures and functions for working with a nonlinear optimization problem in the form of an expression graph. This page explains the API and describes the rationale behind its design.

Standard form

Nonlinear programs (NLPs) are a class of optimization problems in which some of the constraints or the objective function are nonlinear:

$$\min_{x \in \mathbb{R}^n} f_0(x) \tag{37.5}$$

s.t.
$$l_j \le f_j(x) \le u_j$$
 $j = 1 \dots m$ (37.6)

There may be additional constraints, as well as things like variable bounds and integrality restrictions, but we do not consider them here because they are best dealt with by other components of MathOptInterface.

API overview

The core element of the Nonlinear submodule is Nonlinear.Model:

```
julia> const Nonlinear = MOI.Nonlinear;
```

julia> model = Nonlinear.Model()
A Nonlinear.Model with:
0 objectives
0 parameters
0 expressions
0 constraints

Nonlinear.Model is a mutable struct that stores all of the nonlinear information added to the model.

Decision variables

Decision variables are represented by VariableIndexes. The user is responsible for creating these using MOI.VariableIndex(i), where i is the column associated with the variable.

Expressions

The input data structure is a Julia Expr. The input expressions can incorporate VariableIndexes, but these must be interpolated into the expression with \$:

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

julia> input = :(1 + sin(\$x)^2) :(1 + sin(MathOptInterface.VariableIndex(1)) ^ 2)

There are a number of restrictions on the input Expr:

- It cannot contain macros
- · It cannot contain broadcasting
- It cannot contain splatting (except in limited situations)
- · It cannot contain linear algebra, such as matrix-vector products
- It cannot contain generator expressions, including sum(i for i in S)

Given an input expression, add an expression using Nonlinear.add_expression:

```
julia> expr = Nonlinear.add_expression(model, input)
MathOptInterface.Nonlinear.ExpressionIndex(1)
```

The return value, expr, is a Nonlinear.ExpressionIndex that can then be interpolated into other input expressions.

Looking again at model, we see:

CHAPTER 37. SUBMODULES

julia> model

A Nonlinear.Model with: 0 objectives 0 parameters

- 1 expression
- 0 constraints

Parameters

In addition to constant literals like 1 or 1.23, you can create parameters. Parameters are placeholders whose values can change before passing the expression to the solver. Create a parameter using Nonlinear.add_parameter, which accepts a default value:

```
julia> p = Nonlinear.add_parameter(model, 1.23)
MathOptInterface.Nonlinear.ParameterIndex(1)
```

The return value, p, is a Nonlinear. ParameterIndex that can then be interpolated into other input expressions.

Looking again at model, we see:

```
julia> model
A Nonlinear.Model with:
0 objectives
1 parameter
1 expression
0 constraints
```

Update a parameter as follows:

```
julia> model[p]
1.23
julia> model[p] = 4.56
4.56
julia> model[p]
4.56
```

Objectives

Set a nonlinear objective using Nonlinear.set_objective:

```
julia> Nonlinear.set_objective(model, :($p + $expr + $x))
```

```
julia> model
A Nonlinear.Model with:
1 objective
1 parameter
1 expression
0 constraints
```

Clear a nonlinear objective by passing nothing:

```
julia> Nonlinear.set_objective(model, nothing)
```

julia> model A Nonlinear.Model with: 0 objectives 1 parameter 1 expression 0 constraints

But we'll re-add the objective for later:

```
julia> Nonlinear.set_objective(model, :($p + $expr + $x));
```

Constraints

Add a constraint using Nonlinear.add_constraint:

```
julia> c = Nonlinear.add_constraint(model, :(1 + sqrt($x)), MOI.LessThan(2.0))
MathOptInterface.Nonlinear.ConstraintIndex(1)
```

julia> model

- A Nonlinear.Model with:
- 1 objective
- 1 parameter
- 1 expression
- 1 constraint

The return value, c, is a Nonlinear.ConstraintIndex that is a unique identifier for the constraint. Interval constraints are also supported:

```
julia> c2 = Nonlinear.add_constraint(model, :(1 + sqrt($x)), MOI.Interval(-1.0, 2.0))
MathOptInterface.Nonlinear.ConstraintIndex(2)
```

julia> model

A Nonlinear.Model with: 1 objective

- 1 parameter
- 1 expression
- 2 constraints

Delete a constraint using Nonlinear.delete:

julia> Nonlinear.delete(model, c2)

julia> model

A Nonlinear.Model with: 1 objective

- 1 parameter
- 1 expression
- 1 constraint

User-defined operators

By default, Nonlinear supports a wide range of univariate and multivariate operators. However, you can also define your own operators by *registering* them.

Univariate operators

Register a univariate user-defined operator using Nonlinear.register_operator:

```
julia> f(x) = 1 + sin(x)^2
f (generic function with 1 method)
```

julia> Nonlinear.register_operator(model, :my_f, 1, f)

Now, you can use :my_f in expressions:

```
julia> new_expr = Nonlinear.add_expression(model, :(my_f($x + 1)))
MathOptInterface.Nonlinear.ExpressionIndex(2)
```

By default, Nonlinear will compute first- and second-derivatives of the registered operator using ForwardDiff.jl. Override this by passing functions which compute the respective derivative:

```
julia> f'(x) = 2 * sin(x) * cos(x)
f' (generic function with 1 method)
```

julia> Nonlinear.register_operator(model, :my_f2, 1, f, f')

or

```
julia> f''(x) = 2 * (\cos(x)^2 - \sin(x)^2)
f'' (generic function with 1 method)
```

julia> Nonlinear.register_operator(model, :my_f3, 1, f, f', f'')

Multivariate operators

Register a multivariate user-defined operator using Nonlinear.register_operator:

```
julia> g(x...) = x[1]^2 + x[1] * x[2] + x[2]^2
g (generic function with 1 method)
```

julia> Nonlinear.register_operator(model, :my_g, 2, g)

Now, you can use :my_g in expressions:

```
julia> new_expr = Nonlinear.add_expression(model, :(my_g($x + 1, $x)))
MathOptInterface.Nonlinear.ExpressionIndex(3)
```

By default, Nonlinear will compute the gradient of the registered operator using ForwardDiff.jl. (Hessian information is not supported.) Override this by passing a function to compute the gradient:

julia> Nonlinear.register_operator(model, :my_g2, 2, g, ∇g)

MathOptInterface

MathOptInterface communicates the nonlinear portion of an optimization problem to solvers using concrete subtypes of AbstractNLPEvaluator, which implement the Nonlinear programming API.

Create an AbstractNLPEvaluator from Nonlinear.Model using Nonlinear.Evaluator.

Nonlinear.Evaluator requires an Nonlinear.AbstractAutomaticDifferentiation backend and an ordered list of the variables that are included in the model.

There following backends are available to choose from within MOI, although other packages may add more options by sub-typing Nonlinear.AbstractAutomaticDifferentiation:

- Nonlinear.ExprGraphOnly
- Nonlinear.SparseReverseMode.

```
julia> evaluator = Nonlinear.Evaluator(model, Nonlinear.ExprGraphOnly(), [x])
Nonlinear.Evaluator with available features:
    * :ExprGraph
```

The functions of the Nonlinear programming API implemented by Nonlinear.Evaluator depends upon the chosen Nonlinear.AbstractAutomaticDifferentiation backend.

The :ExprGraph feature means we can call objective_expr and constraint_expr to retrieve the expression graph of the problem. However, we cannot call gradient terms such as eval_objective_gradient because Nonlinear.ExprGraphOnly does not have the capability to differentiate a nonlinear expression.

If, instead, we pass Nonlinear.SparseReverseMode, then we get access to :Grad, the gradient of the objective function, :Jac, the Jacobian matrix of the constraints, :JacVec, the ability to compute Jacobian-vector products, and :ExprGraph.

However, before using the evaluator, we need to call initialize:

julia> MOI.initialize(evaluator, [:Grad, :Jac, :JacVec, :ExprGraph])

Now we can call methods like eval objective:

```
julia> x = [1.0]
1-element Vector{Float64}:
1.0
```

julia> MOI.eval_objective(evaluator, x)
7.268073418273571

and eval_objective_gradient:

```
julia> grad = [0.0]
1-element Vector{Float64}:
0.0
```

julia> MOI.eval_objective_gradient(evaluator, grad, x)

```
julia> grad
1-element Vector{Float64}:
1.909297426825682
```

Instead of passing Nonlinear.Evaluator directly to solvers, solvers query the NLPBlock attribute, which returns an NLPBlockData. This object wraps an Nonlinear.Evaluator and includes other information such as constraint bounds and whether the evaluator has a nonlinear objective. Create and set NLPBlockData as follows:

```
julia> block = MOI.NLPBlockData(evaluator);
```

```
julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}());
```

```
julia> MOI.set(model, MOI.NLPBlock(), block);
```

Warning

Only call NLPBlockData once you have finished modifying the problem in model.

Putting everything together, you can create a nonlinear optimization problem in MathOptInterface as follows:

```
import MathOptInterface as MOI
function build_model(
    model::MOI.ModelLike;
    backend::MOI.Nonlinear.AbstractAutomaticDifferentiation,
)
    x = MOI.add_variable(model)
    y = MOI.add_variable(model)
```

```
MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)
nl_model = MOI.Nonlinear.Model()
MOI.Nonlinear.set_objective(nl_model, :($x^2 + $y^2))
evaluator = MOI.Nonlinear.Evaluator(nl_model, backend, [x, y])
MOI.set(model, MOI.NLPBlock(), MOI.NLPBlockData(evaluator))
return
end
# Replace `model` and `backend` with your optimizer and backend of choice.
model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}())
build model(model; backend = MOI.Nonlinear.SparseReverseMode())
```

Expression-graph representation

Nonlinear.Model stores nonlinear expressions in Nonlinear.Expressions. This section explains the design of the expression graph data structure in Nonlinear.Expression.

Given a nonlinear function like $f(x) = sin(x)^2 + x$, a conceptual aid for thinking about the graph representation of the expression is to convert it into Polish prefix notation:

 $f(x, y) = (+ (^ (sin x) 2) x)$

This format identifies each operator (function), as well as a list of arguments. Operators can be univariate, like sin, or multivariate, like +.

A common way of representing Polish prefix notation in code is as follows:

This data structure follows our Polish prefix notation very closely, and we can easily identify the arguments to an operator. However, it has a significant draw-back: each node in the graph requires a Vector, which is heap-allocated and tracked by Julia's garbage collector (GC). For large models, we can expect to have millions of nodes in the expression graph, so this overhead quickly becomes prohibitive for computation.

An alternative is to record the expression as a linear tape:

```
julia> expr = Any[:+, 2, :^, 2, :sin, 1, x, 2.0, x]
9-element Vector{Any}:
    :+
2
    :^
2
    :sin
1
MOI.VariableIndex(1)
```

```
2.0
MOI.VariableIndex(1)
```

The Int after each operator Symbol specifies the number of arguments.

This data-structure is a single vector, which resolves our problem with the GC, but each element is the abstract type, Any, and so any operations on it will lead to slower dynamic dispatch. It's also hard to identify the children of each operation without reading the entire tape.

To summarize, representing expression graphs in Julia has the following challenges:

- Nodes in the expression graph should not contain a heap-allocated object
- All data-structures should be concretely typed
- It should be easy to identify the children of a node

Sketch of the design in Nonlinear

Nonlinear overcomes these problems by decomposing the data structure into a number of different concretetyped vectors.

First, we create vectors of the supported uni- and multivariate operators.

```
julia> const UNIVARIATE_OPERATORS = [:sin];
```

```
julia> const MULTIVARIATE_OPERATORS = [:+, :^];
```

In practice, there are many more supported operations than the ones listed here.

Second, we create an enum to represent the different types of nodes present in the expression graph:

In practice, there are node types other than the ones listed here.

Third, we create two concretely typed structs as follows:

```
julia> struct Node
    type::NodeType
    parent::Int
    index::Int
end
julia> struct Expression
    nodes::Vector{Node}
    values::Vector{Float64}
end
```

For each node node in the .nodes field, if node.type is:

- NODE_CALL_MULTIVARIATE, we look up MULTIVARIATE_OPERATORS[node.index] to retrieve the operator
- NODE_CALL_UNIVARIATE, we look up UNIVARIATE_OPERATORS[node.index] to retrieve the operator
- NODE_VARIABLE, we create MOI.VariableIndex(node.index)
- NODE_VALUE, we look up values[node.index]

The .parent field of each node is the integer index of the parent node in .nodes. For the first node, the parent is -1 by convention.

Therefore, we can represent our function as:

The ordering of the nodes in the tape must satisfy two rules:

- The children of a node must appear after the parent. This means that the tape is ordered topologically, so that a reverse pass of the nodes evaluates all children nodes before their parent
- The arguments for a CALL node are ordered in the tape based on the order in which they appear in the function call.

Design goals

This is less readable than the other options, but does this data structure meet our design goals?

Instead of a heap-allocated object for each node, we only have two Vectors for each expression, nodes and values, as well as two constant vectors for the OPERATORS. In addition, all fields are concretely typed, and there are no Union or Any types.

For our third goal, it is not easy to identify the children of a node, but it is easy to identify the *parent* of any node. Therefore, we can use Nonlinear.adjacency_matrix to compute a sparse matrix that maps parents to their children.

The design in practice

In practice, Node and Expression are exactly Nonlinear.Node and Nonlinear.Expression. However, Nonlinear.NodeType has more fields to account for comparison operators such as :>= and :<=, logic operators such as :&& and :||, nonlinear parameters, and nested subexpressions.

Moreover, instead of storing the operators as global constants, they are stored in Nonlinear.OperatorRegistry, and it also stores a vector of logic operators and a vector of comparison operators. In addition to Nonlinear.DEFAULT_UNIVARIATE and Nonlinear.DEFAULT_MULTIVARIATE_OPERATORS, you can register user-defined functions using Nonlinear.register_operat Nonlinear.Model is a struct that stores the Nonlinear.OperatorRegistry, as well as a list of parameters and subexpressions in the model.

ReverseAD

Nonlinear.ReverseAD is a submodule for computing derivatives of a nonlinear optimization problem using sparse reverse-mode automatic differentiation (AD).

This section does not attempt to explain how sparse reverse-mode AD works, but instead explains why MOI contains its own implementation, and highlights notable differences from similar packages.

Warning

Don't use the API in ReverseAD to compute derivatives. Instead, create a Nonlinear.Evaluator object with Nonlinear.SparseReverseMode as the backend, and then query the MOI API methods.

Design goals

The JuliaDiff organization maintains a list of packages for doing AD in Julia. At last count, there were at least ten packages——not including ReverseAD——for reverse-mode AD in Julia. ReverseAD exists because it has a different set of design goals.

- **Goal: handle scale and sparsity.** The types of nonlinear optimization problems that MOI represents can be large scale (10⁵ or more functions across 10⁵ or more variables) with very sparse derivatives. The ability to compute a sparse Hessian matrix is essential. To the best of our knowledge, ReverseAD is the only reverse-mode AD system in Julia that handles sparsity by default.
- Goal: limit the scope to improve robustness. Most other AD packages accept arbitrary Julia functions as input and then trace an expression graph using operator overloading. This means they must deal (or detect and ignore) with control flow, I/O, and other vagaries of Julia. In contrast, ReverseAD only accepts functions in the form of Nonlinear. Expression, which greatly limits the range of syntax that it must deal with. By reducing the scope of what we accept as input to functions relevant for mathematical optimization, we can provide a simpler implementation with various performance optimizations.
- Goal: provide outputs which match what solvers expect. Other AD packages focus on differentiating individual Julia functions. In contrast, ReverseAD has a very specific use-case: to generate outputs needed by the MOI nonlinear API. This means it needs to efficiently compute sparse Hessians, and it needs subexpression handling to avoid recomputing subexpressions that are shared between functions.

History

ReverseAD started life as ReverseDiffSparse.jl, development of which began in early 2014(!). This was well before the other AD packages started development. Because we had a well-tested, working AD in JuMP, there was less motivation to contribute to and explore other AD packages. The lack of historical interaction also meant that other packages were not optimized for the types of problems that JuMP is built for (that is, large-scale sparse problems). When we first created MathOptInterface, we kept the AD in JuMP to simplify the transition, and post-poned the development of a first-class nonlinear interface in MathOptInterface.

Prior to the introduction of Nonlinear, JuMP's nonlinear implementation was a confusing mix of functions and types spread across the code base and in the private _Derivatives submodule. This made it hard to swap the AD system for another. The main motivation for refactoring JuMP to create the Nonlinear submodule in MathOptInterface was to abstract the interface between JuMP and the AD system, allowing us to swap-in and test new AD systems in the future.

SymbolicAD

SymbolicAD

The Nonlinear.SymbolicAD submodule contains data structures and functions for working with the symbolic derivatives of a nonlinear optimization problem. This page explains the API and describes the rationale behind its design.

Background

The code in SymbolicAD is inspired by Hassan Hijazi's work on coin-or/gravity, a high-performance algebraic modeling language in C++.

Hassan made the following observations:

- For large scale models, symbolic differentiation is slower than other automatic differentiation techniques, such as the reverse mode algorithm implemented in MOI.Nonlinear.ReverseAD.
- However, most large-scale nonlinear programs have a lot of structure.
- Gravity asks the user to provide structure in the form of *template constraints*, where the user gives the symbolic form of the constraint as well as a set of data to convert from a symbolic form to the numerical form.
- Instead of differentiating each constraint in its numerical form, we can compute one symbolic derivative
 of the constraint in symbolic form, and then plug in the data in to get the numerical derivative of each
 function.
- As a final step, if users don't provide the structure, we can still infer it -perhaps with less accuracy-by comparing the expression tree of each constraint.

The symbolic differentiation approach of Gravity works well when the problem is large with few unique constraints. For example, a model like:

```
model = Model()
@variable(model, 0 <= x[1:10_000] <= 1)
@constraint(model, [i=1:10_000], sin(x[i]) <= 1)
@objective(model, Max, sum(x))</pre>
```

is ideal, because although the Jacobian matrix has 10,000 rows, we can compute the derivative of sin(x[i]) as cos(x[i]), and then fill in the Jacobian by evaluating the derivative function instead of having to differentiation 10,000 expressions.

The symbolic differentiation approach of Gravity works poorly if there are a large number of unique constraints in the model (which would require a lot of expressions to be symbolically differentiated), or if the nonlinear functions contain a large number of nonlinear terms (which would make the symbolic derivative expensive to compute).

SymbolicAD started life as MathOptSymbolicAD.jl, development of which began in early 2022. This initial version of SymbolicAD used the Symbolics.jl package to compute the symbolic derivatives. In 2025, we rewrote MathOptSymbolicAD.jl to remove the dependence on Symbolics.jl, and, since the rewrite depended only on MathOptInterface, we added it to MOI.Nonlinear as a new submodule.

For more details on MathOptSymbolicAD.jl, see Oscar's JuMP-dev 2022 talk, although note that the syntax has changed since the original recording.

Use SymbolicAD with JuMP

To use SymbolicAD with JuMP, set the AutomaticDifferentiationBackend attribute to Nonlinear. SymbolicMode:

```
using JuMP, Ipopt
model = Model(Ipopt.Optimizer)
set_attribute(
    model,
    MOI.AutomaticDifferentiationBackend(),
    MOI.Nonlinear.SymbolicMode(),
)
@variable(model, x[1:2])
@objective(model, Min, (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2)
optimize!(model)
```

To revert back to the default sparse reverse mode algorithm, set the AutomaticDifferentiationBackend attribute to Nonlinear.SparseReverseMode.

simplify

Use Nonlinear.SymbolicAD.simplify to simplify nonlinear expressions. The simplification algorithm performs simple rewrites such as lifting nested summations:

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

julia> MOI.Nonlinear.SymbolicAD.simplify(f)
4.0 + 3.0 MOI.VariableIndex(1)

and trivial identities such as $x^1 = x$:

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

```
julia> f = MOI.ScalarNonlinearFunction(:^, Any[x, 1])
^(MOI.VariableIndex(1), (1))
```

```
julia> MOI.Nonlinear.SymbolicAD.simplify(f)
MOI.VariableIndex(1)
```

The list of rewrites that will be made is intentionally limited to keep the codebase simple. Nonlinear.SymbolicAD is not a substitute for a Computer Algebraic System (CAS). For example, we do not detect the relationship $sin(x)^2 + cos(x)^2 = 1$:

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

In addition to Nonlinear.SymbolicAD.simplify, there is an in-place version Nonlinear.SymbolicAD.simplify! that may make changes to the existing function.

variables

Use Nonlinear.SymbolicAD.variables to return a sorted list of the variables that appear in the function:

```
julia> x = MOI.VariableIndex.(1:3)
3-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(1)
MOI.VariableIndex(2)
MOI.VariableIndex(3)
```

```
julia> f = MOI.ScalarNonlinearFunction(:atan, Any[x[3], 2.0 * x[1]])
atan(MOI.VariableIndex(3), 0.0 + 2.0 MOI.VariableIndex(1))
```

```
julia> MOI.Nonlinear.SymbolicAD.variables(f)
2-element Vector{MathOptInterface.VariableIndex}:
    MOI.VariableIndex(1)
```

derivative

MOI.VariableIndex(3)

Use Nonlinear.SymbolicAD.derivative to compute the symbolic derivative of a function with respect to a decision variable:

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

julia> f = MOI.ScalarNonlinearFunction(:sin, Any[x]) sin(MOI.VariableIndex(1))

```
julia> MOI.Nonlinear.SymbolicAD.derivative(f, x)
cos(MOI.VariableIndex(1))
```

Note that the resultant expression can often be simplified. Thus, in most cases you should call Nonlinear.SymbolicAD.simplify on the expression before using it in other places:

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

```
julia> f = MOI.ScalarNonlinearFunction(:sin, Any[x + 1.0])
sin(1.0 + 1.0 MOI.VariableIndex(1))
```

```
julia> df_dx = MOI.Nonlinear.SymbolicAD.derivative(f, x)
*(cos(1.0 + 1.0 MOI.VariableIndex(1)), 1.0)
```

```
julia> MOI.Nonlinear.SymbolicAD.simplify!(df_dx)
cos(1.0 + 1.0 MOI.VariableIndex(1))
```

gradient_and_hessian

In some cases, you may want to compute the gradient and (sparse) Hessian matrix of a function. One way to achieve this is by recursively calling Nonlinear.SymbolicAD.derivative on the result of Nonlinear.SymbolicAD.derivative for each variable in the list of Nonlinear.SymbolicAD.variables. But, to simplify the process, you should use Nonlinear.SymbolicAD.gradient_and_hessian:

```
julia> x = MOI.VariableIndex.(1:2);
```

```
julia> f = MOI.ScalarNonlinearFunction(:sin, Any[1.0 * x[1] + 2.0 * x[2]])
sin(0.0 + 1.0 MOI.VariableIndex(1) + 2.0 MOI.VariableIndex(2))
```

julia> y, Vf, H, V²f = MOI.Nonlinear.SymbolicAD.gradient_and_hessian(f);

julia> y

```
2-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(1)
MOI.VariableIndex(2)
```

julia> ⊽f

```
2-element Vector{Any}:
    cos(0.0 + 1.0 MOI.VariableIndex(1) + 2.0 MOI.VariableIndex(2))
    *(cos(0.0 + 1.0 MOI.VariableIndex(1) + 2.0 MOI.VariableIndex(2)), 2.0)
```

julia> H

```
3-element Vector{Tuple{Int64, Int64}}:
  (1, 1)
  (1, 2)
  (2, 2)
```

julia> ∇²f

```
3-element Vector{Any}:
    -(sin(0.0 + 1.0 MOI.VariableIndex(1) + 2.0 MOI.VariableIndex(2)))
    *(-(sin(0.0 + 1.0 MOI.VariableIndex(1) + 2.0 MOI.VariableIndex(2))), 2.0)
    *(-(sin(0.0 + 1.0 MOI.VariableIndex(1) + 2.0 MOI.VariableIndex(2))), 4.0)
```

- y is the list of variables that appear in f
- ∇f is the first partial derivatives of f with respect to each variable in y
- H and ∇² f form a sparse Hessian matrix, were H is the (row, column) index of each element, and the corresponding element in ∇² f is the second partial derivative of f with respect to y[row] and y[column].

Unlike Nonlinear.SymbolicAD.derivative, the gradient and Hessian expressions have already been simplified; you do not need to call Nonlinear.SymbolicAD.simplify.

API Reference

Nonlinear Modeling

More information can be found in the Nonlinear section of the manual.

MathOptInterface.Nonlinear.Model - Type.

Model()

The core datastructure for representing a nonlinear optimization problem.

It has the following fields:

- objective::Union{Nothing, Expression} : holds the nonlinear objective function, if one exists, otherwise nothing.
- expressions::Vector{Expression} : a vector of expressions in the model.
- constraints::OrderedDict{ConstraintIndex,Constraint}: a map from ConstraintIndex to the corresponding Constraint. An OrderedDict is used instead of a Vector to support constraint deletion.
- parameters::Vector{Float64} : holds the current values of the parameters.
- operators::OperatorRegistry: stores the operators used in the model.

source

Expressions

MathOptInterface.Nonlinear.ExpressionIndex - Type.

ExpressionIndex

An index to a nonlinear expression that is returned by add_expression.

Given data::Model and ex::ExpressionIndex, use data[ex] to retrieve the corresponding Expression.

source

MathOptInterface.Nonlinear.add_expression - Function.

```
add_expression(model::Model, expr)::ExpressionIndex
```

Parse expr into a Expression and add to model. Returns an ExpressionIndex that can be interpolated into other input expressions.

expr must be a type that is supported by parse_expression.

Example

```
julia> model = MOI.Nonlinear.Model();
```

```
julia> x = MOI.VariableIndex(1);
```

julia> ex = MOI.Nonlinear.add_expression(model, :(\$x^2 + 1))
MathOptInterface.Nonlinear.ExpressionIndex(1)

```
julia> MOI.Nonlinear.set_objective(model, :(sqrt($ex)))
```

source

Parameters

MathOptInterface.Nonlinear.ParameterIndex - Type.

ParameterIndex

An index to a nonlinear parameter that is returned by add_parameter. Given data::Model and p::ParameterIndex, use data[p] to retrieve the current value of the parameter and data[p] = value to set a new value.

source

MathOptInterface.Nonlinear.add_parameter - Function.

```
add_parameter(model::Model, value::Float64)::ParameterIndex
```

Add a new parameter to model with the default value value. Returns a ParameterIndex that can be interpolated into other input expressions and used to modify the value of the parameter.

Example

```
julia> model = MOI.Nonlinear.Model()
A Nonlinear.Model with:
0 objectives
0 parameters
0 expressions
0 constraints
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

1321

```
julia> p = MOI.Nonlinear.add_parameter(model, 1.2)
MathOptInterface.Nonlinear.ParameterIndex(1)
```

```
julia> c = MOI.Nonlinear.add_constraint(model, :($x^2 - $p), MOI.LessThan(0.0))
MathOptInterface.Nonlinear.ConstraintIndex(1)
```

source

Objectives

MathOptInterface.Nonlinear.set_objective - Function.

set_objective(model::Model, obj)::Nothing

Parse obj into a Expression and set as the objective function of model.

obj must be a type that is supported by parse_expression.

To remove the objective, pass nothing.

Example

```
julia> model = MOI.Nonlinear.Model()
A Nonlinear.Model with:
0 objectives
0 parameters
0 expressions
0 constraints
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
julia> MOI.Nonlinear.set_objective(model, :($x^2 + 1))
julia> MOI.Nonlinear.set_objective(model, x)
julia> MOI.Nonlinear.set_objective(model, nothing)
```

source

Constraints

MathOptInterface.Nonlinear.ConstraintIndex - Type.

ConstraintIndex

An index to a nonlinear constraint that is returned by add_constraint.

Given data::Model and c::ConstraintIndex, use data[c] to retrieve the corresponding Constraint.

source

MathOptInterface.Nonlinear.add_constraint - Function.

```
add_constraint(
    model::Model,
    func,
    set::Union{
        MOI.GreaterThan{Float64},
        MOI.LessThan{Float64},
        MOI.Interval{Float64},
        MOI.EqualTo{Float64},
        },
    )
```

Parse func and set into a Constraint and add to model. Returns a ConstraintIndex that can be used to delete the constraint or query solution information.

Example

julia> model = MOI.Nonlinear.Model();

julia> x = MOI.VariableIndex(1);

```
julia> c = MOI.Nonlinear.add_constraint(model, :($x^2), MOI.LessThan(1.0))
MathOptInterface.Nonlinear.ConstraintIndex(1)
```

source

MathOptInterface.Nonlinear.delete - Function.

delete(model::Model, c::ConstraintIndex)::Nothing

Delete the constraint index c from model.

Example

julia> model = MOI.Nonlinear.Model()
A Nonlinear.Model with:
0 objectives
0 parameters
0 expressions
0 constraints

julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)

julia> c = MOI.Nonlinear.add_constraint(model, :(\$x^2), MOI.LessThan(1.0))
MathOptInterface.Nonlinear.ConstraintIndex(1)

julia> model
A Nonlinear.Model with:
0 objectives

CHAPTER 37. SUBMODULES

0 parameters 0 expressions

1 constraint

julia> MOI.Nonlinear.delete(model, c)

julia> model

A Nonlinear.Model with:

0 objectives

0 parameters

- 0 expressions
- 0 constraints

source

User-defined operators

MathOptInterface.Nonlinear.OperatorRegistry - Type.

OperatorRegistry()

Create a new OperatorRegistry to store and evaluate univariate and multivariate operators.

source

MathOptInterface.Nonlinear.DEFAULT UNIVARIATE OPERATORS - Constant.

DEFAULT_UNIVARIATE_OPERATORS

The list of univariate operators that are supported by default.

Example

julia> MOI.Nonlinear.DEFAULT_UNIVARIATE_OPERATORS

73-element Vector{Symbol}: :+ :abs :sign :sqrt :cbrt :abs2 :inv :log :log10 :airybi :airyaiprime :airybiprime :besselj0 :besselj1
:bessely0 :bessely1 :erfcx :dawson

source

MathOptInterface.Nonlinear.DEFAULT_MULTIVARIATE_OPERATORS - Constant.

DEFAULT_MULTIVARIATE_OPERATORS

The list of multivariate operators that are supported by default.

Example

julia> MOI.Nonlinear.DEFAULT_MULTIVARIATE_OPERATORS
9-element Vector{Symbol}:
 :+
 : :*
 :^
 :/
 ifelse
 :atan
 :min
 :max

source

MathOptInterface.Nonlinear.register_operator - Function.

```
register_operator(
    model::Model,
    op::Symbol,
    nargs::Int,
    f::Function,
    [∇f::Function],
    [∇<sup>2</sup>f::Function],
)
```

Register the user-defined operator op with nargs input arguments in model.

Univariate functions

- f(x::T)::T must be a function that takes a single input argument x and returns the function evaluated at x. If ∇f and ∇² f are not provided, f must support any Real input type T.
- ∇f(x::T)::T is a function that takes a single input argument x and returns the first derivative of f with respect to x. If ∇² f is not provided, ∇f must support any Real input type T.
- ∇² f(x::T)::T is a function that takes a single input argument x and returns the second derivative of f with respect to x.

Multivariate functions

- f(x::T...)::T must be a function that takes a nargs input arguments x and returns the function evaluated at x. If ∇f and ∇² f are not provided, f must support any Real input type T.
- ∇f(g::AbstractVector{T}, x::T...)::T is a function that takes a cache vector g of length length(x),
 and fills each element g[i] with the partial derivative of f with respect to x[i].
- ∇²f(H::AbstractMatrix, x::T...)::T is a function that takes a matrix H and fills the lower-triangular components H[i, j] with the Hessian of f with respect to x[i] and x[j] for i >= j.

Notes for multivariate Hessians

- H has size(H) == (length(x), length(x)), but you must not access elements H[i, j] for i > j.
- H is dense, but you do not need to fill structural zeros.

source

MathOptInterface.Nonlinear.register_operator_if_needed - Function.

```
register_operator_if_needed(
    registry::OperatorRegistry,
    op::Symbol,
    nargs::Int,
    f::Function;
)
```

Similar to register_operator, but this function warns if the function is not registered, and skips silently if it already is.

```
source
```

MathOptInterface.Nonlinear.assert_registered - Function.

assert_registered(registry::OperatorRegistry, op::Symbol, nargs::Int)

Throw an error if op is not registered in registry with nargs arguments.

source

MathOptInterface.Nonlinear.check_return_type - Function.

check_return_type(::Type{T}, ret::S) where {T,S}

Overload this method for new types S to throw an informative error if a user-defined function returns the type S instead of T.

source

MathOptInterface.Nonlinear.eval_univariate_function - Function.

```
eval_univariate_function(
    registry::OperatorRegistry,
    op::Union{Symbol,Integer},
    x::T,
) where {T}
```

Evaluate the operator op(x)::T, where op is a univariate function in registry.

If op isa Integer, then op is the index in registry.univariate_operators[op].

Example

```
julia> r = MOI.Nonlinear.OperatorRegistry();
julia> MOI.Nonlinear.eval_univariate_function(r, :abs, -1.2)
1.2
julia> r.univariate_operators[3]
:abs
julia> MOI.Nonlinear.eval_univariate_function(r, 3, -1.2)
1.2
```

source

MathOptInterface.Nonlinear.eval_univariate_gradient - Function.

```
eval_univariate_gradient(
    registry::OperatorRegistry,
    op::Union{Symbol,Integer},
    x::T,
) where {T}
```

Evaluate the first-derivative of the operator op(x)::T, where op is a univariate function in registry.

If op isa Integer, then op is the index in registry.univariate_operators[op].

```
julia> r = MOI.Nonlinear.OperatorRegistry();
julia> MOI.Nonlinear.eval_univariate_gradient(r, :abs, -1.2)
-1.0
julia> r.univariate_operators[3]
:abs
julia> MOI.Nonlinear.eval_univariate_gradient(r, 3, -1.2)
-1.0
```

MathOptInterface.Nonlinear.eval_univariate_function_and_gradient - Function.

```
eval_univariate_function_and_gradient(
    registry::OperatorRegistry,
    op::Union{Symbol,Integer},
    x::T,
)::Tuple{T,T} where {T}
```

Evaluate the function and first-derivative of the operator op(x)::T, where op is a univariate function in registry.

If op isa Integer, then op is the index in registry.univariate_operators[op].

Example

julia> r = MOI.Nonlinear.OperatorRegistry();

```
julia> MOI.Nonlinear.eval_univariate_function_and_gradient(r, :abs, -1.2)
(1.2, -1.0)
```

```
julia> r.univariate_operators[3]
:abs
```

```
julia> MOI.Nonlinear.eval_univariate_function_and_gradient(r, 3, -1.2)
(1.2, -1.0)
```

source

MathOptInterface.Nonlinear.eval_univariate_hessian - Function.

```
eval_univariate_hessian(
    registry::OperatorRegistry,
    op::Union{Symbol,Integer},
    x::T,
) where {T}
```

Evaluate the second-derivative of the operator op(x)::T, where op is a univariate function in registry.

If op isa Integer, then op is the index in registry.univariate_operators[op].

```
julia> r = MOI.Nonlinear.OperatorRegistry();
julia> MOI.Nonlinear.eval_univariate_hessian(r, :sin, 1.0)
-0.8414709848078965
julia> r.univariate_operators[16]
:sin
julia> MOI.Nonlinear.eval_univariate_hessian(r, 16, 1.0)
-0.8414709848078965
```

julia> -sin(1.0) -0.8414709848078965

source

MathOptInterface.Nonlinear.eval_multivariate_function - Function.

```
eval_multivariate_function(
    registry::OperatorRegistry,
    op::Symbol,
    x::AbstractVector{T},
) where {T}
```

Evaluate the operator op(x)::T, where op is a multivariate function in registry.

source

MathOptInterface.Nonlinear.eval_multivariate_gradient - Function.

```
eval_multivariate_gradient(
    registry::0peratorRegistry,
    op::Symbol,
    g::AbstractVector{T},
    x::AbstractVector{T},
) where {T}
```

Evaluate the gradient of operator g $.= \nabla op(x)$, where op is a multivariate function in registry.

source

MathOptInterface.Nonlinear.eval_multivariate_hessian - Function.

```
eval_multivariate_hessian(
    registry::OperatorRegistry,
    op::Symbol,
    H::AbstractMatrix,
    x::AbstractVector{T},
)::Bool where {T}
```

Evaluate the Hessian of operator $\nabla^2 op(x)$, where op is a multivariate function in registry.

The Hessian is stored in the lower-triangular part of the matrix H.

Returns a Bool indicating whether non-zeros were stored in the matrix.

Note

Implementations of the Hessian operators will not fill structural zeros. Therefore, before calling this function you should pre-populate the matrix H with θ .

source

MathOptInterface.Nonlinear.eval_logic_function - Function.

```
eval_logic_function(
    registry::OperatorRegistry,
    op::Symbol,
    lhs::T,
    rhs::T,
)::Bool where {T}
```

Evaluate (lhs op rhs)::Bool, where op is a logic operator in registry.

source

MathOptInterface.Nonlinear.eval_comparison_function - Function.

```
eval_comparison_function(
    registry::OperatorRegistry,
    op::Symbol,
    lhs::T,
    rhs::T,
)::Bool where {T}
```

Evaluate (lhs op rhs)::Bool, where op is a comparison operator in registry.

source

Automatic-differentiation backends

MathOptInterface.Nonlinear.Evaluator - Type.

```
Evaluator(
    model::Model,
    backend::AbstractAutomaticDifferentiation,
    ordered_variables::Vector{MOI.VariableIndex},
)
```

Create Evaluator, a subtype of MOI.AbstractNLPEvaluator, from Model.

source

MathOptInterface.Nonlinear.AbstractAutomaticDifferentiation - Type.

AbstractAutomaticDifferentiation

An abstract type for extending Evaluator.

source

MathOptInterface.Nonlinear.ExprGraphOnly - Type.

ExprGraphOnly() <: AbstractAutomaticDifferentiation</pre>

 $The \ default \ implementation \ of \ Abstract \ Automatic \ Differentiation. \ The \ only \ supported \ feature \ is \ : \ ExprGraph.$

source

MathOptInterface.Nonlinear.SparseReverseMode - Type.

SparseReverseMode() <: AbstractAutomaticDifferentiation</pre>

An implementation of AbstractAutomaticDifferentiation that uses sparse reverse-mode automatic differentiation to compute derivatives. Supports all features in the MOI nonlinear interface.

source

MathOptInterface.Nonlinear.SymbolicMode - Type.

SymbolicMode() <: AbstractAutomaticDifferentiation</pre>

A type for setting as the value of the MOI.AutomaticDifferentiationBackend() attribute to enable symbolic automatic differentiation.

source

Data-structure

MathOptInterface.Nonlinear.Node - Type.

```
struct Node
    type::NodeType
    index::Int
    parent::Int
end
```

A single node in a nonlinear expression tree. Used by Expression.

See the MathOptInterface documentation for information on how the nodes and values form an expression tree.

source

MathOptInterface.Nonlinear.NodeType - Type.

NodeType

An enum describing the possible node types. Each Node has a .index field, which should be interpreted as follows:

- NODE_CALL_MULTIVARIATE: the index into operators.multivariate_operators
- NODE_CALL_UNIVARIATE: the index into operators.univariate_operators
- NODE_LOGIC: the index into operators.logic_operators
- NODE_COMPARISON: the index into operators.comparison_operators
- NODE_MOI_VARIABLE: the value of MOI.VariableIndex(index) in the user's space of the model.
- NODE_VARIABLE: the 1-based index of the internal vector
- NODE_VALUE: the index into the .values field of Expression
- NODE_PARAMETER: the index into data.parameters
- NODE_SUBEXPRESSION: the index into data.expressions

source

MathOptInterface.Nonlinear.Expression - Type.

```
struct Expression
    nodes::Vector{Node}
    values::Vector{Float64}
end
```

The core type that represents a nonlinear expression. See the MathOptInterface documentation for information on how the nodes and values form an expression tree.

source

MathOptInterface.Nonlinear.Constraint - Type.

```
struct Constraint
    expression::Expression
    set::Union{
        MOI.LessThan{Float64},
        MOI.GreaterThan{Float64},
        MOI.EqualTo{Float64},
        MOI.Interval{Float64},
    }
end
```

A type to hold information relating to the nonlinear constraint f(x) in S, where f(x) is defined by .expression, and S is .set.

source

MathOptInterface.Nonlinear.adjacency_matrix - Function.

adjacency_matrix(nodes::Vector{Node})

Compute the sparse adjacency matrix describing the parent-child relationships in nodes.

The element (i, j) is true if there is an edge *from* node[j] to node[i]. Since we get a column-oriented matrix, this gives us a fast way to look up the edges leaving any node (that is, the children).

source

MathOptInterface.Nonlinear.parse_expression - Function.

```
parse_expression(data::Model, input)::Expression
Parse input into a Expression.
source
parse_expression(
    data::Model,
    expr::Expression,
    input::Any,
    parent_index::Int,
)::Expression
```

Parse input into a Expression, and add it to expr as a child of expr.nodes[parent_index]. Existing subexpressions and parameters are stored in data.

You can extend parsing support to new types of objects by overloading this method with a different type on input::Any.

source

MathOptInterface.Nonlinear.convert_to_expr - Function.

```
convert to expr(data::Model, expr::Expression)
```

Convert the Expression expr into a Julia Expr.

- subexpressions are represented by a ExpressionIndex object.
- parameters are represented by a ParameterIndex object.
- variables are represented by an MOI.VariableIndex object.

source

```
convert_to_expr(
    evaluator::Evaluator,
    expr::Expression;
    moi_output_format::Bool,
)
```

Convert the Expression expr into a Julia Expr.

lf moi_output_format = true:

- subexpressions will be converted to Julia Expr and substituted into the output expression.
- · the current value of each parameter will be interpolated into the expression
- variables will be represented in the form x[MOI.VariableIndex(i)]

If moi_output_format = false:

- subexpressions will be represented by a ExpressionIndex object.
- parameters will be represented by a ParameterIndex object.
- variables will be represented by an MOI.VariableIndex object.

Warning

To use moi_output_format = true, you must have first called MOI.initialize with :ExprGraph as a requested feature.

source

MathOptInterface.Nonlinear.ordinal_index - Function.

ordinal_index(evaluator::Evaluator, c::ConstraintIndex)::Int

Return the 1-indexed value of the constraint index c in evaluator.

```
julia> model = MOI.Nonlinear.Model()
A Nonlinear.Model with:
 0 objectives
 0 parameters
 0 expressions
 0 constraints
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
julia> c1 = MOI.Nonlinear.add_constraint(model, :($x^2), MOI.LessThan(1.0))
MathOptInterface.Nonlinear.ConstraintIndex(1)
julia> c2 = MOI.Nonlinear.add_constraint(model, :($x^2), MOI.LessThan(1.0))
MathOptInterface.Nonlinear.ConstraintIndex(2)
julia> evaluator = MOI.Nonlinear.Evaluator(model)
Nonlinear.Evaluator with available features:
  * :ExprGraph
julia> MOI.initialize(evaluator, Symbol[])
julia> MOI.Nonlinear.ordinal_index(evaluator, c2) # Returns 2
julia> MOI.Nonlinear.delete(model, c1)
julia> evaluator = MOI.Nonlinear.Evaluator(model)
Nonlinear. Evaluator with available features:
  * :ExprGraph
```

```
julia> MOI.initialize(evaluator, Symbol[])
julia> MOI.Nonlinear.ordinal_index(evaluator, c2) # Returns 1
1
```

source

SymbolicAD

MathOptInterface.Nonlinear.SymbolicAD.simplify - Function.

simplify(f)

Return a simplified copy of the function f.

Warning

This function is not type stable by design.

Example

julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)

```
julia> f = MOI.ScalarNonlinearFunction(:^, Any[x, 1])
 ^(MOI.VariableIndex(1), (1))
```

julia> MOI.Nonlinear.SymbolicAD.simplify(f)
MOI.VariableIndex(1)

source

MathOptInterface.Nonlinear.SymbolicAD.simplify! - Function.

simplify!(f)

Simplify the function f in-place and return either the function f or a new object if f can be represented in a simpler type.

Warning

This function is not type stable by design.

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

:+,

```
julia> f = MOI.ScalarNonlinearFunction(
```

Any[MOI.ScalarNonlinearFunction(:+, Any[1.0, x]), 2.0 * x + 3.0],
)

+(+(1.0, MOI.VariableIndex(1)), 3.0 + 2.0 MOI.VariableIndex(1))

julia> MOI.Nonlinear.SymbolicAD.simplify!(f)
4.0 + 3.0 MOI.VariableIndex(1)

julia> f

```
+(1.0, MOI.VariableIndex(1), 3.0 + 2.0 MOI.VariableIndex(1))
```

source

simplify!(::Val{head}, f::MOI.ScalarNonlinearFunction)

Simplify the function f in-place and return either the function f or a new object if f can be represented in a simpler type.

Val

The head in Val{head} is taken from f.head. This function should be called as:

```
f = simplify!(Val(f.head), f)
```

Implementing a method that dispatches on head enables custom simplification rules for different operators without needing a giant switch statement.

Note

It is important that this function does not recursively call simplify!. Deal only with the immediate operator. The children arguments will already be simplified.

source

MathOptInterface.Nonlinear.SymbolicAD.variables - Function.

variables(f::Union{Real,MOI.AbstractScalarFunction})

Return a sorted list of the MOI.VariableIndex present in the function f.

```
julia> x = MOI.VariableIndex.(1:3)
3-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(1)
MOI.VariableIndex(2)
MOI.VariableIndex(3)
```

```
julia> f = MOI.ScalarNonlinearFunction(:atan, Any[x[3], 2.0 * x[1]])
atan(MOI.VariableIndex(3), 0.0 + 2.0 MOI.VariableIndex(1))
```

julia> MOI.Nonlinear.SymbolicAD.variables(f)
2-element Vector{MathOptInterface.VariableIndex}:
MOI.VariableIndex(1)
MOI.VariableIndex(3)

source

MathOptInterface.Nonlinear.SymbolicAD.derivative - Function.

derivative(f::Union{Real,MOI.AbstractScalarFunction}, x::MOI.VariableIndex)

Return an expression representing the partial derivative of f with respect to x.

Expression swelling

With few exceptions, the algorithm used to compute the derivative does not perform simplications. As a result, the returned expression may contain terms like *(false, g) that can be trivially simplified to false.

In most cases, you should call simplify! (derivative(f, x)) to return a simplified expression of the derivative.

Example

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

```
julia> f = MOI.ScalarNonlinearFunction(:sin, Any[x])
sin(MOI.VariableIndex(1))
```

```
julia> df_dx = MOI.Nonlinear.SymbolicAD.derivative(f, x)
cos(MOI.VariableIndex(1))
```

source

MathOptInterface.Nonlinear.SymbolicAD.gradient_and_hessian - Function.

```
gradient_and_hessian(
    [filter_fn::Function = x -> true,]
    f::MOI.AbstractScalarFunction,
)
```

Compute the symbolic gradient and Hessian of f, and return the result as a tuple of four elements:

- 1. x::Vector{MOI.VariableIndex}: the list of variables that appear in f
- 2. $\nabla f::Vector{Any}: a vector for the first partial derivative of f with respect to each element in x$

- 3. H::Vector{Tuple{Int,Int}}: a vector of (row, column) tuples that list the non-zero entries in the Hessian of f
- ∇²f::Vector{Any}: a vector of expressions, in the same order as H, for the non-zero entries in the Hessian of f

filter_fn

This argument is a function, filter_fn(::MOI.VariableIndex)::Bool that returns true if the gradient and Hessian of f should be computed with respect to it.

Use this argument to filter out constant parameters from decision variables.

source

37.5 Utilities

Overview

The Utilities submodule

The Utilities submodule provides a variety of functions and datastructures for managing MOI.ModelLike objects.

Utilities.Model

Utilities.Model provides an implementation of a ModelLike that efficiently supports all functions and sets defined within MOI. However, given the extensibility of MOI, this might not cover all use cases.

Create a model as follows:

Utilities.UniversalFallback

Utilities.UniversalFallback is a layer that sits on top of any ModelLike and provides non-specialized (slower) fallbacks for constraints and attributes that the underlying ModelLike does not support.

For example, Utilities.Model doesn't support some variable attributes like VariablePrimalStart, so JuMP uses a combination of Universal fallback and Utilities.Model as a generic problem cache:

```
julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}())
```

MOIU.UniversalFallback{MOIU.Model{Float64}}

```
bjectiveSense: FEASIBILITY_SENSE
```

bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}

```
- NumberOfVariables: 0
```

L NumberOfConstraints: 0

Warning

Adding a UniversalFallback means that your model will now support all constraints, even if the innermodel does not. This can lead to unexpected behavior.

Utilities.@model

For advanced use cases that need efficient support for functions and sets defined outside of MOI (but still known at compile time), we provide the Utilities.@model macro.

The @model macro takes a name (for a new type, which must not exist yet), eight tuples specifying the types of constraints that are supported, and then a Bool indicating the type is a subtype of MOI.AbstractOptimizer (if true) or MOI.ModelLike (if false).

The eight tuples are in the following order:

- 1. Un-typed scalar sets, for example, Integer
- 2. Typed scalar sets, for example, LessThan
- 3. Un-typed vector sets, for example, Nonnegatives
- 4. Typed vector sets, for example, PowerCone
- 5. Un-typed scalar functions, for example, VariableIndex
- 6. Typed scalar functions, for example, ScalarAffineFunction
- 7. Un-typed vector functions, for example, VectorOfVariables
- 8. Typed vector functions, for example, VectorAffineFunction

The tuples can contain more than one element. Typed-sets must be specified without their type parameter, for example, MOI.LessThan, not MOI.LessThan{Float64}.

Here is an example:

```
julia> MOI.Utilities.@model(
          MyNewModel,
          (MOI.Integer,),
                                         # Un-typed scalar sets
          (MOI.GreaterThan,),
                                         # Typed scalar sets
          (MOI.Nonnegatives,),
                                        # Un-typed vector sets
          (MOI.PowerCone,),
                                         # Typed vector sets
          (MOI.VariableIndex,),
                                        # Un-typed scalar functions
          (MOI.ScalarAffineFunction,), # Typed scalar functions
          (MOI.VectorOfVariables,),
                                        # Un-typed vector functions
          (MOI.VectorAffineFunction,), # Typed vector functions
          true,
                                         # <:MOI.AbstractOptimizer?</pre>
      )
```

MathOptInterface.Utilities.GenericOptimizer{T, MathOptInterface.Utilities.ObjectiveContainer{T}, → MathOptInterface.Utilities.VariablesContainer{T}, MyNewModelFunctionConstraints{T}} where T

julia> model = MyNewModel{Float64}()

MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64}, MOIU.VariablesContainer{Float64}, → MyNewModelFunctionConstraints{Float64}} ↓ ObjectiveSense: FEASIBILITY_SENSE

CHAPTER 37. SUBMODULES

bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}

```
- NumberOfVariables: 0
```

```
L NumberOfConstraints: 0
```

Warning

MyNewModel supports every VariableIndex-in-Set constraint, as well as VariableIndex, ScalarAffineFunction, and ScalarQuadraticFunction objective functions. Implement MOI.supports as needed to forbid constraint and objective function combinations.

As another example, PATHSolver, which only supports VectorAffineFunction-in-Complements defines its optimizer as:

```
julia> MOI.Utilities.@model(
    PathOptimizer,
    (), # Scalar sets
    (), # Typed scalar sets
    (MOI.Complements,), # Vector sets
    (), # Typed vector sets
    (), # Typed vector sets
    (), # Scalar functions
    (), # Typed scalar functions
    (), # Vector functions
    (MOI.VectorAffineFunction,), # Typed vector functions
    true, # is_optimizer
    )
MathOptInterface.Utilities.GenericOptimizer{T, MathOptInterface.Utilities.ObjectiveContainer{T},
    · MathOptInterface.Utilities.VariablesContainer{T},
```

← MathOptInterface.Utilities.VectorOfConstraints{MathOptInterface.VectorAffineFunction{T},

```
↔ MathOptInterface.Complements} where T
```

However, PathOptimizer does not support some VariableIndex-in-Set constraints, so we must explicitly define:

```
julia> function MOI.supports_constraint(
                ::PathOptimizer,
                ::Type{MOI.VariableIndex},
                ::Type{Union{<:MOI.Semiinteger,MOI.Semicontinuous,MOI.ZeroOne,MOI.Integer}}
        )
        return false
    end</pre>
```

Finally, PATH doesn't support an objective function, so we need to add:

julia> MOI.supports(::PathOptimizer, ::MOI.ObjectiveFunction) = false

Warning

This macro creates a new type, so it must be called from the top-level of a module, for example, it cannot be called from inside a function.

Utilities.CachingOptimizer

A [Utilities.CachingOptimizer] is an MOI layer that abstracts the difference between solvers that support incremental modification (for example, they support adding variables one-by-one), and solvers that require the entire problem in a single API call (for example, they only accept the A, b and c matrices of a linear program).

It has two parts:

- 1. A cache, where the model can be built and modified incrementally
- 2. An optimizer, which is used to solve the problem

```
julia> model = MOI.Utilities.CachingOptimizer(
          MOI.Utilities.Model{Float64}(),
          PathOptimizer{Float64}(),
      )
MOIU.CachingOptimizer
- state: EMPTY OPTIMIZER
- mode: AUTOMATIC
- model cache: MOIU.Model{Float64}
| | ObjectiveSense: FEASIBILITY_SENSE
| | NumberOfVariables: 0
 L NumberOfConstraints: 0
L optimizer: MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↔ MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
\hookrightarrow MOI.Complements}
 bjectiveSense: FEASIBILITY SENSE
  bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
  - NumberOfVariables: 0
 L NumberOfConstraints: 0
```

A Utilities.CachingOptimizer may be in one of three possible states:

- N0_0PTIMIZER: The CachingOptimizer does not have any optimizer.
- EMPTY_OPTIMIZER: The CachingOptimizer has an empty optimizer, and it is not synchronized with the cached model. Modifications are forwarded to the cache, but *not* to the optimizer.
- ATTACHED_OPTIMIZER: The CachingOptimizer has an optimizer, and it is synchronized with the cached model. Modifications are forwarded to the optimizer. If the optimizer does not support modifications, and error will be thrown.

Use Utilities.attach_optimizer to go from EMPTY_OPTIMIZER to ATTACHED_OPTIMIZER:

julia> MOI.Utilities.attach_optimizer(model)

julia> MOI.Utilities.state(model)
ATTACHED_OPTIMIZER::CachingOptimizerState = 2

Info

You must be in ATTACHED_OPTIMIZER to use optimize!.

Use Utilities.reset_optimizer to go from ATTACHED_OPTIMIZER to EMPTY_OPTIMIZER:

julia> MOI.Utilities.reset_optimizer(model)

```
julia> MOI.Utilities.state(model)
EMPTY_OPTIMIZER::CachingOptimizerState = 1
```

Info

Calling MOI.empty!(model) also resets the state to EMPTY_OPTIMIZER. So after emptying a model, the modification will only be applied to the cache.

Use Utilities.drop_optimizer to go from any state to NO_OPTIMIZER:

```
julia> MOI.Utilities.drop_optimizer(model)
```

julia> MOI.Utilities.state(model)
NO_OPTIMIZER::CachingOptimizerState = 0

julia> model

MOIU.CachingOptimizer
 state: N0_OPTIMIZER
 mode: AUTOMATIC
 model_cache: MOIU.Model{Float64}
 | ObjectiveSense: FEASIBILITY_SENSE
 | ObjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
 | NumberOfVariables: 0
 L NumberOfConstraints: 0
 optimizer: nothing

Pass an empty optimizer to Utilities.reset_optimizer to go from NO_OPTIMIZER to EMPTY_OPTIMIZER:

julia> MOI.Utilities.reset_optimizer(model, PathOptimizer{Float64}())

julia> MOI.Utilities.state(model) EMPTY_OPTIMIZER::CachingOptimizerState = 1

julia> model

MOIU.CachingOptimizer

- state: EMPTY_OPTIMIZER
- mode: AUTOMATIC
- h model_cache: MOIU.Model{Float64}
- | | ObjectiveSense: FEASIBILITY_SENSE
- | | NumberOfVariables: 0
- | L NumberOfConstraints: 0

L optimizer: MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},

- ↔ MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
- \hookrightarrow MOI.Complements}

CHAPTER 37. SUBMODULES

```
b ObjectiveSense: FEASIBILITY_SENSE
b ObjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
NumberOfVariables: 0
NumberOfConstraints: 0
```

Deciding when to attach and reset the optimizer is tedious, and you will often write code like this:

```
try
    # modification
catch
    MOI.Utilities.reset_optimizer(model)
    # Re-try modification
end
```

To make this easier, Utilities.CachingOptimizer has two modes of operation:

- AUTOMATIC: The CachingOptimizer changes its state when necessary. Attempting to add a constraint
 or perform a modification not supported by the optimizer results in a drop to EMPTY_OPTIMIZER mode.
- MANUAL: The user must change the state of the CachingOptimizer. Attempting to perform an operation in the incorrect state results in an error.

By default, AUTOMATIC mode is chosen. However, you can create a CachingOptimizer in MANUAL mode as follows:

julia> MOI.Utilities.reset_optimizer(model, PathOptimizer{Float64}())

julia> model

```
1343
```

```
bjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
```

- NumberOfVariables: 0

```
L NumberOfConstraints: 0
```

Printing

Use print to print the formulation of the model.

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

```
julia> MOI.set(model, MOI.VariableName(), x, "x_var")
```

julia> MOI.add_constraint(model, x, MOI.ZeroOne())
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}()

```
julia> MOI.set(model, MOI.ObjectiveFunction{typeof(x)}(), x)
```

```
julia> MOI.set(model, MOI.ObjectiveSense(), MOI.MAX_SENSE)
```

```
julia> print(model)
Maximize VariableIndex:
    x_var
```

Subject to:

VariableIndex-in-ZeroOne
x_var ∈ {0, 1}

Use Utilities.latex_formulation to display the model in LaTeX form:

```
julia> MOI.Utilities.latex_formulation(model)
```

```
$$ \begin{aligned}
\max\quad & x\_var \\
\text{Subject to}\\
& \text{VariableIndex-in-ZeroOne} \\
& x\_var \in \{0, 1\} \\
\end{aligned} $$
```

Тір

In IJulia, calling print or ending a cell with Utilities.latex_formulation will render the model in LaTeX.

Utilities.PenaltyRelaxation

Pass Utilities.PenaltyRelaxation to modify to relax the problem by adding penalized slack variables to the constraints. This is helpful when debugging sources of infeasible models.

```
julia> model = MOI.Utilities.Model{Float64}();
```

julia> x = MOI.add_variable(model);

julia> MOI.set(model, MOI.VariableName(), x, "x")

julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0));

julia> map = MOI.modify(model, MOI.Utilities.PenaltyRelaxation(Dict(c => 2.0)));

julia> print(model) Minimize CoolerAffineTure

```
Minimize ScalarAffineFunction{Float64}:
 0.0 + 2.0 v[2]
```

Subject to:

```
ScalarAffineFunction{Float64}-in-LessThan{Float64}
0.0 + 1.0 x - 1.0 v[2] <= 2.0</pre>
```

VariableIndex-in-GreaterThan{Float64} v[2] >= 0.0

julia> map[c] 0.0 + 1.0 MOI.VariableIndex(2)

You can also modify a single constraint using Utilities.ScalarPenaltyRelaxation:

```
julia> model = MOI.Utilities.Model{Float64}();
julia> x = MOI.add_variable(model);
julia> MOI.set(model, MOI.VariableName(), x, "x")
julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0));
julia> f = MOI.modify(model, c, MOI.Utilities.ScalarPenaltyRelaxation(2.0));
julia> print(model)
Minimize ScalarAffineFunction{Float64}:
0.0 + 2.0 v[2]
Subject to:
ScalarAffineFunction{Float64}-in-LessThan{Float64}
0.0 + 1.0 x - 1.0 v[2] <= 2.0
VariableIndex-in-GreaterThan{Float64}
v[2] >= 0.0
```

```
julia> f
0.0 + 1.0 MOI.VariableIndex(2)
```

Utilities.MatrixOfConstraints

The constraints of Utilities.Model are stored as a vector of tuples of function and set in a Utilities.VectorOfConstraints.

Other representations can be used by parameterizing the type Utilities.GenericModel (resp. Utilities.GenericOptimizer).

For example, if all non-VariableIndex constraints are affine, the coefficients of all the constraints can be stored in a single sparse matrix using Utilities.MatrixOfConstraints.

The constraints storage can even be customized up to a point where it exactly matches the storage of the solver of interest, in which case copy_to can be implemented for the solver by calling copy_to to this custom model.

For example, Clp.jl defines the following model:

```
julia> MOI.Utilities.@product of sets(
           SupportedSets,
           MOI.EqualTo{T},
           MOI.LessThan{T},
           MOI.GreaterThan{T},
           MOI.Interval{T},
       );
julia> const OptimizerCache = MOI.Utilities.GenericModel{
           Float64.
           MOI.Utilities.ObjectiveContainer{Float64},
           MOI.Utilities.VariablesContainer{Float64},
           MOI.Utilities.MatrixOfConstraints{
               Float64,
               MOI.Utilities.MutableSparseMatrixCSC{
                   # The data type of the coefficients
                   Float64,
                   # The data type of the variable indices
                   Cint,
                   # Can also be MOI.Utilities.OneBasedIndexing
                   MOI.Utilities.ZeroBasedIndexing,
               },
               MOI.Utilities.Hyperrectangle{Float64},
               SupportedSets{Float64},
          },
      };
```

Given the input model:

We can construct a new cached model and copy src to it:

```
julia> dest = OptimizerCache();
julia> index_map = MOI.copy_to(dest, src);
```

From dest, we can access the A matrix in sparse matrix form:

```
julia> A = dest.constraints.coefficients;
```

julia> A.n 3

julia> A.m 3

julia> A.colptr

4-element Vector{Int32}: 0 2 4 5

julia> A.rowval

5-element Vector{Int32}: 0 1 1 2 0

julia> A.nzval

5-element Vector{Float64}: -4.0 1.0 1.0 2.0 1.0

The lower and upper row bounds:

julia> row_bounds = dest.constraints.constants;

julia> row_bounds.lower
3-element Vector{Float64}:
 5.0
 Inf
 3.0

julia> row_bounds.upper
3-element Vector{Float64}:

5.0 1.0 Inf

The lower and upper variable bounds:

```
julia> dest.variables.lower
3-element Vector{Float64}:
    0.0
    -Inf
    5.0
julia> dest.variables.upper
3-element Vector{Float64}:
    1.0
    10.0
    5.0
```

Because of larger variations between solvers, the objective can be queried using the standard MOI methods:

```
julia> MOI.get(dest, MOI.ObjectiveSense())
MAX_SENSE::OptimizationSense = 1
julia> F = MOI.get(dest, MOI.ObjectiveFunctionType())
MathOptInterface.ScalarAffineFunction{Float64}
julia> F = MOI.get(dest, MOI.ObjectiveFunction{F}())
```

0.0 + 1.0 MOI.VariableIndex(1) + 2.0 MOI.VariableIndex(2) - 3.1 MOI.VariableIndex(3)

```
Thus, Clp.jl implements copy_to methods similar to the following:
```

```
# This method copies from the cache to the `Clp.Optimizer` object.
function MOI.copy_to(dest::Optimizer, src::OptimizerCache)
   @assert MOI.is_empty(dest)
   A = src.constraints.coefficients
    row_bounds = src.constraints.constants
    Clp_loadProblem(
        dest,
        A.n,
        Α.m,
        A.colptr,
        A.rowval,
        A.nzval,
        src.lower_bound,
        src.upper_bound,
        # (...) objective vector (omitted),
        row_bounds.lower,
        row_bounds.upper,
    )
    return MOI.Utilities.identity_index_map(src)
end
```

```
# This method copies from an arbitrary model to the optimizer, by the
# intermediate `OptimizerCache` representation.
function MOI.copy_to(dest::Optimizer, src::MOI.ModelLike)
    cache = OptimizerCache()
   index_map = MOI.copy_to(cache, src)
   MOI.copy_to(dest, cache)
   return index_map
end
# This is a special method that gets called in some cases when `OptimizerCache`
# is used as the backing data structure in a `MOI.Utilities.CachingOptimizer`.
# It is needed for performance, but not correctness.
function MOI.copy_to(
   dest::Optimizer,
   src::MOI.Utilities.UniversalFallback{OptimizerCache},
)
   MOI.Utilities.throw unsupported(src)
    return MOI.copy_to(dest, src.model)
end
```

Тір

For other examples of Utilities.MatrixOfConstraints, see:

- Cbc.jl
- ECOS.jl
- SCS.jl

ModelFilter

Utilities provides Utilities.ModelFilter as a useful tool to copy a subset of a model. For example, given an infeasible model, we can copy the irreducible infeasible subsystem (for models implementing ConstraintConflictStatus) as follows:

```
my_filter(::Any) = true
function my_filter(ci::MOI.ConstraintIndex)
    status = MOI.get(dest, MOI.ConstraintConflictStatus(), ci)
    return status != MOI.NOT_IN_CONFLICT
end
filtered_src = MOI.Utilities.ModelFilter(my_filter, src)
index_map = MOI.copy_to(dest, filtered_src)
```

Fallbacks

The value of some attributes can be inferred from the value of other attributes.

For example, the value of ObjectiveValue can be computed using ObjectiveFunction and VariablePrimal.

When a solver gives direct access to an attribute, it is better to return this value. However, if this is not the case, Utilities.get_fallback can be used instead. For example:

```
function MOI.get(model::Optimizer, attr::MOI.ObjectiveFunction)
    return MOI.Utilities.get_fallback(model, attr)
end
```

DoubleDicts

When writing MOI interfaces, we often need to handle situations in which we map ConstraintIndexs to different values. For example, to a string for ConstraintName.

One option is to use a dictionary like Dict{MOI.ConstraintIndex,String}. However, this incurs a performance cost because the key is not a concrete type.

The DoubleDicts submodule helps this situation by providing two types main types Utilities.DoubleDicts.DoubleDict and Utilities.DoubleDicts.IndexDoubleDict. These types act like normal dictionaries, but internally they use more efficient dictionaries specialized to the type of the function-set pair.

The most common usage of a DoubleDict is in the index_map returned by copy_to. Performance can be improved, by using a function barrier. That is, instead of code like:

```
index_map = MOI.copy_to(dest, src)
for (F, S) in MOI.get(src, MOI.ListOfConstraintTypesPresent())
    for ci in MOI.get(src, MOI.ListOfConstraintIndices{F,S}())
        dest_ci = index_map[ci]
        # ...
    end
end
```

use instead:

```
function function_barrier(
    dest,
    src,
    index_map::MOI.Utilities.DoubleDicts.IndexDoubleDictInner{F,S},
) where {F,S}
    for ci in MOI.get(src, MOI.ListOfConstraintIndices{F,S}())
        dest_ci = index_map[ci]
        # ...
    end
    return
end
index_map = MOI.copy_to(dest, src)
for (F, S) in MOI.get(src, MOI.ListOfConstraintTypesPresent())
        function_barrier(dest, src, index_map[F, S])
end
```

API Reference

Utilities.Model

MathOptInterface.Utilities.Model - Type.

MOI.Utilities.Model{T}() where {T}

An implementation of ModelLike that supports all functions and sets defined in MOI. It is parameterized by the coefficient type.

Example

```
julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}
    ObjectiveSense: FEASIBILITY_SENSE
    ObjectiveFunctionType: MOI.ScalarAffineFunction{Float64}
    NumberOfVariables: 0
    L NumberOfConstraints: 0
```

source

Utilities.UniversalFallback

MathOptInterface.Utilities.UniversalFallback - Type.

UniversalFallback

The UniversalFallback can be applied on a MOI.ModelLike model to create the model UniversalFallback (model) supporting *any* constraint and attribute. This allows to have a specialized implementation in model for performance critical constraints and attributes while still supporting other attributes with a small performance penalty. Note that model is unaware of constraints and attributes stored by UniversalFallback so this is not appropriate if model is an optimizer (for this reason, MOI.optimize! has not been implemented). In that case, optimizer bridges should be used instead.

```
source
```

Utilities.@model

MathOptInterface.Utilities.@model - Macro.

```
macro model(
    model_name,
    scalar_sets,
    typed_scalar_sets,
    vector_sets,
    typed_vector_sets,
    scalar_functions,
    typed_scalar_functions,
    vector_functions,
    typed_vector_functions,
    is_optimizer = false
)
```

Creates a type model_name implementing the MOI model interface and supporting all combinations of the provided functions and sets.

Each typed_ scalar/vector sets/functions argument is a tuple of types. A type is "typed" if it has a coefficient $\{T\}$ as the first type parameter.

Tuple syntax

To give no set/function, write (). To give one set or function X, write (X,).

is_optimizer

If is_optimizer = true, the resulting struct is a of GenericOptimizer, which is a subtype of MOI.AbstractOptimizer,
otherwise, it is a GenericModel, which is a subtype of MOI.ModelLike.

VariableIndex

- The function MOI.VariableIndex must not be given in scalar_functions.
- The model supports MOI.VariableIndex-in-S constraints where S is MOI.EqualTo, MOI.GreaterThan, MOI.LessThan, MOI.Interval, MOI.Integer, MOI.ZeroOne, MOI.Semicontinuous or MOI.Semiinteger.
- The sets supported with MOI.VariableIndex cannot be controlled from the macro; use UniversalFallback to support more sets.

Example

The model describing a linear program would be:

```
@model(
   LPModel,
                                                     # model name
                                                     # untyped scalar sets
   (),
    (MOI.EqualTo, MOI.GreaterThan, MOI.LessThan, MOI.Interval), # typed scalar sets
    (MOI.Zeros, MOI.Nonnegatives, MOI.Nonpositives), # untyped vector sets
   (),
                                                     # typed vector sets
   (),
                                                     # untyped scalar functions
    (MOI.ScalarAffineFunction,),
                                                     # typed scalar functions
    (MOI.VectorOfVariables.).
                                                     # untyped vector functions
    (MOI.VectorAffineFunction,),
                                                     # typed vector functions
   false,
                                                     # is_optimizer
)
```

```
source
```

MathOptInterface.Utilities.GenericModel - Type.

mutable struct GenericModel{T,0,V,C} <: AbstractModelLike{T}</pre>

Implements a model supporting coefficients of type T and:

- An objective function stored in .objective::0
- Variables and VariableIndex constraints stored in .variable_bounds::V
- F-in-S constraints (excluding VariableIndex constraints) stored in .constraints::C

All interactions take place via the MOI interface, so the types 0, V, and C must implement the API as needed for their functionality.

source

MathOptInterface.Utilities.GenericOptimizer - Type.

```
mutable struct GenericOptimizer{T,0,V,C} <: AbstractOptimizer{T}</pre>
```

Implements a model supporting coefficients of type T and:

- An objective function stored in .objective::0
- Variables and VariableIndex constraints stored in .variable_bounds::V
- F-in-S constraints (excluding VariableIndex constraints) stored in .constraints::C

All interactions take place via the MOI interface, so the types 0, V, and C must implement the API as needed for their functionality.

source

.objective

MathOptInterface.Utilities.ObjectiveContainer - Type.

```
ObjectiveContainer{T}
```

A helper struct to simplify the handling of objective functions in Utilities.Model.

source

.variables

MathOptInterface.Utilities.VariablesContainer - Type.

```
struct VariablesContainer{T} <: AbstractVectorBounds
    set_mask::Vector{UInt16}
    lower::Vector{T}
    upper::Vector{T}
end</pre>
```

A struct for storing variables and VariableIndex-related constraints. Used in MOI.Utilities.Model by default.

source

MathOptInterface.Utilities.FreeVariables - Type.

```
mutable struct FreeVariables <: MOI.ModelLike
    n::Int64
    FreeVariables() = new(0)
end</pre>
```

A struct for storing free variables that can be used as the variables field of GenericModel or GenericModel. It represents a model that does not support any constraint nor objective function.

Example

The following model type represents a conic model in geometric form. As opposed to VariablesContainer, FreeVariables does not support constraint bounds so they are bridged into an affine constraint in the MOI.Nonnegatives cone as expected for the geometric conic form.

```
julia> MOI.Utilities.@product_of_sets(
    Cones.
    MOI.Zeros,
   MOI.Nonnegatives,
    MOI.SecondOrderCone,
    MOI.PositiveSemidefiniteConeTriangle,
);
julia> const ConicModel{T} = MOI.Utilities.GenericOptimizer{
    т.
    MOI.Utilities.ObjectiveContainer{T},
    MOI.Utilities.FreeVariables,
    MOI.Utilities.MatrixOfConstraints{
        Т,
       MOI.Utilities.MutableSparseMatrixCSC{
           Τ,
           Int.
           MOI.Utilities.OneBasedIndexing,
       },
       Vector{T},
        Cones{T},
    },
};
julia> model = MOI.instantiate(ConicModel{Float64}, with_bridge_type=Float64);
julia> x = MOI.add_variable(model)
MathOptInterface.VariableIndex(1)
julia> c = MOI.add_constraint(model, x, MOI.GreaterThan(1.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
\rightarrow MathOptInterface.GreaterThan{Float64}}(1)
julia> MOI.Bridges.is_bridged(model, c)
true
julia> bridge = MOI.Bridges.bridge(model, c)
MathOptInterface.Bridges.Constraint.VectorizeBridge{Float64,
\hookrightarrow \mbox{MathOptInterface.VectorAffineFunction{Float64}, \mbox{MathOptInterface.Nonnegatives}, \label{eq:MathOptInterface}
\hookrightarrow MathOptInterface.Nonnegatives}(1), 1.0)
julia> bridge.vector_constraint
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
\hookrightarrow MathOptInterface.Nonnegatives}(1)
julia> MOI.Bridges.is_bridged(model, bridge.vector_constraint)
false
```

source

.constraints

MathOptInterface.Utilities.VectorOfConstraints - Type.

```
mutable struct VectorOfConstraints{
    F<:MOI.AbstractFunction,
    S<:MOI.AbstractSet,
} <: MOI.ModelLike
    constraints::CleverDicts.CleverDict{
        MOI.ConstraintIndex{F,S},
        Tuple{F,S},
        typeof(CleverDicts.key_to_index),
        typeof(CleverDicts.index_to_key),
    }
end</pre>
```

A struct storing F-in-S constraints as a mapping between the constraint indices to the corresponding tuple of function and set.

source

MathOptInterface.Utilities.StructOfConstraints - Type.

abstract type StructOfConstraints <: MOI.ModelLike end</pre>

A struct storing a subfields other structs storing constraints of different types.

See Utilities.@struct_of_constraints_by_function_types and Utilities.@struct_of_constraints_by_set_types.
source

MathOptInterface.Utilities.@struct_of_constraints_by_function_types - Macro.

Utilities.@struct_of_constraints_by_function_types(name, func_types...)

Given a vector of n function types (F1, F2,..., Fn) in func_types, defines a subtype of StructOfConstraints of name name and which type parameters {T, C1, C2, ..., Cn}. It contains n field where the ith field has type Ci and stores the constraints of function type Fi.

The expression Fi can also be a union in which case any constraint for which the function type is in the union is stored in the field with type Ci.

source

MathOptInterface.Utilities.@struct_of_constraints_by_set_types - Macro.

Utilities.@struct_of_constraints_by_set_types(name, func_types...)

Given a vector of n set types (S1, S2,..., Sn) in func_types, defines a subtype of StructOfConstraints of name name and which type parameters {T, C1, C2, ..., Cn}. It contains n field where the ith field has type Ci and stores the constraints of set type Si. The expression Si can also be a union in which case any constraint for which the set type is in the union is stored in the field with type Ci. This can be useful if Ci is a MatrixOfConstraints in order to concatenate the coefficients of constraints of several different set types in the same matrix.

source

MathOptInterface.Utilities.struct_of_constraint_code - Function.

struct_of_constraint_code(struct_name, types, field_types = nothing)

Given a vector of n Union{SymbolFun,_UnionSymbolFS{SymbolFun}} or Union{SymbolSet,_UnionSymbolFS{SymbolSet}} in types, defines a subtype of StructOfConstraints of name name and which type parameters {T, F1, F2, ..., Fn} if field_types is nothing and a {T} otherwise. It contains n field where the ith field has type Ci if field_types is nothing and type field_types[i] otherwise. If types is vector of Union{SymbolFun,_UnionSymbolFS{SymbolFun}} (resp. Union{SymbolSet,_UnionSymbolFS{SymbolSet}}) then the constraints of that function (resp. set) type are stored in the corresponding field.

This function is used by the macros @model, @struct_of_constraints_by_function_types and @struct_of_constraints_b

source

Caching optimizer

MathOptInterface.Utilities.CachingOptimizer - Type.

CachingOptimizer

CachingOptimizer is an intermediate layer that stores a cache of the model and links it with an optimizer. It supports incremental model construction and modification even when the optimizer doesn't.

Mode

A Utilities.CachingOptimizer has two modes of operation: Utilities.AUTOMATIC and Utilities.MANUAL. See their docstrings for details.

Use Utilities.mode to query the mode of a Utilities.CachingOptimizer.

State

A Utilities.CachingOptimizer may be in one of three possible states: NO_OPTIMIZER, Utilities.EMPTY_OPTIMIZER, and Utilities.ATTACHED_OPTIMIZER. See their docstrings for details.

Use Utilities.state to query the state of a Utilities.CachingOptimizer.

Constructor with optimizer

CachingOptimizer(cache::MOI.ModelLike, optimizer::AbstractOptimizer)

Creates a CachingOptimizer in AUTOMATIC mode, with the optimizer optimizer.

The type of the optimizer returned is CachingOptimizer{typeof(optimizer), typeof(cache)} so it does not support the function Utilities.reset_optimizer(::CachingOptimizer, new_optimizer) if the type of new_optimizer is different from the type of optimizer.

Constructor without optimizer

CachingOptimizer(cache::MOI.ModelLike, mode::CachingOptimizerMode)

Creates a CachingOptimizer in the NO_OPTIMIZER Utilities.CachingOptimizerState and the Utilities.CachingOptimizerState and the

The type of the optimizer returned is CachingOptimizer{MOI.AbstractOptimizer, typeof(cache)} so it *does* support the function reset_optimizer(::CachingOptimizer, new_optimizer) if the type of new_optimizer is different from the type of optimizer.

source

MathOptInterface.Utilities.CachingOptimizerState - Type.

CachingOptimizerState

A Utilities.CachingOptimizer may be in one of three possible states.

Values

NO_OPTIMIZER

The CachingOptimizer does not have any optimizer.

EMPTY OPTIMIZER

The CachingOptimizer has an optimizer. The optimizer is empty and it is not synchronized with the cached model.

ATTACHED_OPTIMIZER

The CachingOptimizer has an optimizer, and it is synchronized with the cached model.

source

MathOptInterface.Utilities.NO_OPTIMIZER - Constant.

NO_OPTIMIZER::CachingOptimizerState

An instance of the CachingOptimizerState enum.

About

The CachingOptimizer does not have any optimizer.

source

MathOptInterface.Utilities.EMPTY_OPTIMIZER - Constant.

EMPTY_OPTIMIZER::CachingOptimizerState

An instance of the CachingOptimizerState enum.

About

The CachingOptimizer has an optimizer. The optimizer is empty and it is not synchronized with the cached model.

source

MathOptInterface.Utilities.ATTACHED_OPTIMIZER - Constant.

ATTACHED_OPTIMIZER::CachingOptimizerState

An instance of the CachingOptimizerState enum.

About

The CachingOptimizer has an optimizer, and it is synchronized with the cached model.

source

MathOptInterface.Utilities.state - Function.

state(m::CachingOptimizer)::CachingOptimizerState

Returns the state of the CachingOptimizer m.

See Utilities.CachingOptimizer.

source

MathOptInterface.Utilities.CachingOptimizerMode - Type.

CachingOptimizerMode

A Utilities.CachingOptimizer has two modes of operation.

Values

MANUAL

The only methods that change the state of the CachingOptimizer are Utilities.reset_optimizer, Utilities.drop_optimizer, and Utilities.attach_optimizer.

Attempting to perform an operation in the incorrect state results in an error.

AUTOMATIC

The Utilities.CachingOptimizer changes its state when necessary. For example, MOI.optimize! will automatically call Utilities.attach_optimizer (an optimizer must have been previously set). Attempting to add a constraint or perform a modification not supported by the optimizer results in a drop to the EMPTY_OPTIMIZER state.

source

MathOptInterface.Utilities.AUTOMATIC - Constant.

AUTOMATIC::CachingOptimizerMode

An instance of the CachingOptimizerMode enum.

About

The Utilities.CachingOptimizer changes its state when necessary. For example, MOI.optimize! will automatically call Utilities.attach_optimizer (an optimizer must have been previously set). Attempting to add a constraint or perform a modification not supported by the optimizer results in a drop to the EMPTY_OPTIMIZER state.

source

MathOptInterface.Utilities.MANUAL - Constant.

MANUAL:: CachingOptimizerMode

An instance of the CachingOptimizerMode enum.

About

The only methods that change the state of the CachingOptimizer are Utilities.reset_optimizer, Utilities.drop_optimizer, and Utilities.attach_optimizer.

Attempting to perform an operation in the incorrect state results in an error.

source

MathOptInterface.Utilities.mode - Function.

mode(m::CachingOptimizer)::CachingOptimizerMode

Returns the operating mode of the CachingOptimizer m.

See Utilities.CachingOptimizer.

source

MathOptInterface.Utilities.attach_optimizer - Function.

attach_optimizer(model::CachingOptimizer)

Attaches the optimizer to model, copying all model data into it. Can be called only from the EMPTY_OPTIMIZER state. If the copy succeeds, the CachingOptimizer will be in state ATTACHED_OPTIMIZER after the call, otherwise an error is thrown; see MOI.copy_to for more details on which errors can be thrown.

source

MOIU.attach_optimizer(model::GenericModel)

Call MOIU.attach_optimizer on the backend of model.

Cannot be called in direct mode.

source

MathOptInterface.Utilities.reset_optimizer - Function.

reset_optimizer(m::CachingOptimizer, optimizer::MOI.AbstractOptimizer)

Sets or resets m to have the given empty optimizer optimizer.

Can be called from any state. An assertion error will be thrown if optimizer is not empty.

The CachingOptimizer m will be in state EMPTY_OPTIMIZER after the call.

source

reset_optimizer(m::CachingOptimizer)

Detaches and empties the current optimizer. Can be called from ATTACHED_OPTIMIZER or EMPTY_OPTIMIZER state. The CachingOptimizer will be in state EMPTY_OPTIMIZER after the call.

source

MOIU.reset_optimizer(model::GenericModel, optimizer::MOI.AbstractOptimizer)

Call MOIU.reset_optimizer on the backend of model.

Cannot be called in direct mode.

source

MOIU.reset_optimizer(model::GenericModel)

Call MOIU.reset_optimizer on the backend of model.

Cannot be called in direct mode.

source

MathOptInterface.Utilities.drop_optimizer - Function.
drop_optimizer(m::CachingOptimizer)

Drops the optimizer, if one is present. Can be called from any state. The CachingOptimizer will be in state NO_OPTIMIZER after the call.

source

MOIU.drop optimizer(model::GenericModel)

Call MOIU.drop_optimizer on the backend of model.

Cannot be called in direct mode.

source

Mock optimizer

MathOptInterface.Utilities.MockOptimizer - Type.

MockOptimizer

MockOptimizer is a fake optimizer especially useful for testing. Its main feature is that it can store the values that should be returned for each attribute.

source

Printing

MathOptInterface.Utilities.latex_formulation - Function.

latex_formulation(model::MOI.ModelLike; kwargs...)

Wrap model in a type so that it can be pretty-printed as text/latex in a notebook like IJulia, or in Documenter.

To render the model, end the cell with latex_formulation(model), or call display(latex_formulation(model)) in to force the display of the model from inside a function.

Possible keyword arguments are:

- simplify_coefficients : Simplify coefficients if possible by omitting them or removing trailing zeros.
- default_name : The name given to variables with an empty name.
- print_types : Print the MOI type of each function and set for clarity.

source

Copy utilities

MathOptInterface.Utilities.default_copy_to - Function.

default_copy_to(dest::MOI.ModelLike, src::MOI.ModelLike)

A default implementation of MOI.copy_to(dest, src) for models that implement the incremental interface, that is, MOI.supports_incremental_interface returns true.

source

MathOptInterface.Utilities.IndexMap - Type.

IndexMap()

The dictionary-like object returned by MOI.copy_to.

source

MathOptInterface.Utilities.identity_index_map - Function.

identity_index_map(model::MOI.ModelLike)

Return an IndexMap that maps all variable and constraint indices of model to themselves.

source

MathOptInterface.Utilities.ModelFilter - Type.

ModelFilter(filter::Function, model::MOI.ModelLike)

A layer to filter out various components of model.

The filter function takes a single argument, which is each element from the list returned by the attributes below. It returns true if the element should be visible in the filtered model and false otherwise.

The components that are filtered are:

- Entire constraint types via:
 - MOI.ListOfConstraintTypesPresent
- Individual constraints via:
 - MOI.ListOfConstraintIndices{F,S}
- Specific attributes via:
 - MOI.ListOfModelAttributesSet
 - MOI.ListOfConstraintAttributesSet
 - MOI.ListOfVariableAttributesSet

Warning

The list of attributes filtered may change in a future release. You should write functions that are generic and not limited to the five types listed above. Thus, you should probably define a fallback filter(::Any) = true.

Note

See below for examples of how this works.

This layer has a limited scope. It is intended by be used in conjunction with MOI.copy_to.

Example: copy model excluding integer constraints

Use the do syntax to provide a single function.

```
filtered_src = MOI.Utilities.ModelFilter(src) do item
    return item != (MOI.VariableIndex, MOI.Integer)
end
MOI.copy_to(dest, filtered_src)
```

Example: copy model excluding names

Use type dispatch to simplify the implementation:

```
my_filter(::Any) = true # Note the generic fallback
my_filter(::MOI.VariableName) = false
my_filter(::MOI.ConstraintName) = false
filtered_src = MOI.Utilities.ModelFilter(my_filter, src)
MOI.copy_to(dest, filtered_src)
```

Example: copy irreducible infeasible subsystem

```
my_filter(::Any) = true # Note the generic fallback
function my_filter(ci::MOI.ConstraintIndex)
    status = MOI.get(dest, MOI.ConstraintConflictStatus(), ci)
    return status != MOI.NOT_IN_CONFLICT
end
filtered_src = MOI.Utilities.ModelFilter(my_filter, src)
MOI.copy_to(dest, filtered_src)
```

source

MathOptInterface.Utilities.loadfromstring! - Function.

loadfromstring!(model, s)

A utility function to aid writing tests.

Warning

This function is not intended for widespread use. It is mainly used as a tool to simplify writing tests in MathOptInterface. Do not use it as an exchange format for storing or transmitting problem instances. Use the FileFormats submodule instead.

Example

```
julia> model = MOI.Utilities.Model{Float64}();
julia> MOI.Utilities.loadfromstring!(model, """
    variables: x, y, z
    constrainedvariable: [a, b, c] in Nonnegatives(3)
    minobjective::Float64: 2x + 3y
    con1: x + y <= 1.0
    con2: [x, y] in Nonnegatives(2)
    x >= 0.0
    """)
```

Notes

Special labels are:

- variables
- minobjective
- maxobjectives

Everything else denotes a constraint with a name.

Append :: T to use an element type of T when parsing the function.

Do not name VariableIndex constraints.

Exceptions

- x y does NOT currently parse. Instead, write x + -1.0 * y.
- x² does NOT currently parse. Instead, write x * x.

source

Penalty relaxation

MathOptInterface.Utilities.PenaltyRelaxation - Type.

```
PenaltyRelaxation(
    penalties = Dict{MOI.ConstraintIndex,Float64}();
    default::Union{Nothing,T} = 1.0,
)
```

A problem modifier that, when passed to MOI.modify, destructively modifies the model in-place to create a penalized relaxation of the constraints.

Warning

This is a destructive routine that modifies the model in-place. If you don't want to modify the original model, use JuMP.copy_model to create a copy before calling MOI.modify.

Reformulation

See Utilities.ScalarPenaltyRelaxation for details of the reformulation.

For each constraint ci, the penalty passed to Utilities.ScalarPenaltyRelaxation is get(penalties, ci, default). If the value is nothing, because ci does not exist in penalties and default = nothing, then the constraint is skipped.

Return value

MOI.modify(model, PenaltyRelaxation()) returns a Dict{MOI.ConstraintIndex,MOI.ScalarAffineFunction} that maps each constraint index to the corresponding y + z as a MOI.ScalarAffineFunction. In an optimal solution, query the value of these functions to compute the violation of each constraint.

Relax a subset of constraints

To relax a subset of constraints, pass a penalties dictionary and set default = nothing.

Supported constraint types

The penalty relaxation is currently limited to modifying MOI.ScalarAffineFunction and MOI.ScalarQuadraticFunction constraints in the linear sets MOI.LessThan, MOI.GreaterThan, MOI.EqualTo and MOI.Interval.

It does not include variable bound or integrality constraints, because these cannot be modified in-place.

To modify variable bounds, rewrite them as linear constraints.

Example

julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model);

julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0));

julia> map = MOI.modify(model, MOI.Utilities.PenaltyRelaxation(default = 2.0));

julia> print(model)

```
Minimize ScalarAffineFunction{Float64}:
 0.0 + 2.0 v[2]
```

Subject to:

```
ScalarAffineFunction{Float64}-in-LessThan{Float64}
0.0 + 1.0 v[1] - 1.0 v[2] <= 2.0</pre>
```

VariableIndex-in-GreaterThan{Float64}
v[2] >= 0.0

```
julia> map[c] isa MOI.ScalarAffineFunction{Float64}
true
```

```
julia> model = MOI.Utilities.Model{Float64}();
julia> x = MOI.add_variable(model);
julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0));
julia> map = MOI.modify(model, MOI.Utilities.PenaltyRelaxation(Dict(c => 3.0)));
julia> print(model)
```

```
Minimize ScalarAffineFunction{Float64}:
    0.0 + 3.0 v[2]
```

Subject to:

ScalarAffineFunction{Float64}-in-LessThan{Float64}
0.0 + 1.0 v[1] - 1.0 v[2] <= 2.0</pre>

```
VariableIndex-in-GreaterThan{Float64}
v[2] >= 0.0
```

julia> map[c] isa MOI.ScalarAffineFunction{Float64}
true

source

MathOptInterface.Utilities.ScalarPenaltyRelaxation - Type.

ScalarPenaltyRelaxation(penalty::T) where {T}

A problem modifier that, when passed to MOI.modify, destructively modifies the constraint in-place to create a penalized relaxation of the constraint.

Warning

This is a destructive routine that modifies the constraint in-place. If you don't want to modify the original model, use JuMP.copy_model to create a copy before calling MOI.modify.

Reformulation

The penalty relaxation modifies constraints of the form $f(x) \in S$ into $f(x) + y - z \in S$, where $y, z \ge 0$, and then it introduces a penalty term into the objective of $a \times (y + z)$ (if minimizing, else -a), where a is penalty

When S is MOI.LessThan or MOI.GreaterThan, we omit y or z respectively as a performance optimization.

Return value

MOI.modify(model, ci, ScalarPenaltyRelaxation(penalty)) returns y + z as a MOI.ScalarAffineFunction. In an optimal solution, query the value of this function to compute the violation of the constraint.

Example

julia> model = MOI.Utilities.Model{Float64}(); julia> x = MOI.add_variable(model); julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0)); julia> f = MOI.modify(model, c, MOI.Utilities.ScalarPenaltyRelaxation(2.0)); julia> print(model) Minimize ScalarAffineFunction{Float64}: 0.0 + 2.0 v[2]

Subject to:

ScalarAffineFunction{Float64}-in-LessThan{Float64}
0.0 + 1.0 v[1] - 1.0 v[2] <= 2.0</pre>

```
VariableIndex-in-GreaterThan{Float64}
v[2] >= 0.0
```

julia> f isa MOI.ScalarAffineFunction{Float64}
true

source

MatrixOfConstraints

MathOptInterface.Utilities.MatrixOfConstraints - Type.

```
mutable struct MatrixOfConstraints{T,AT,BT,ST} <: MOI.ModelLike
    coefficients::AT
    constants::BT
    sets::ST
    caches::Vector{Any}
    are_indices_mapped::Vector{BitSet}
    final_touch::Bool
end
```

Represent ScalarAffineFunction and VectorAffinefunction constraints in a matrix form where the linear coefficients of the functions are stored in the coefficients field, the constants of the functions or sets are stored in the constants field. Additional information about the sets are stored in the sets field.

This model can only be used as the constraints field of a MOI.Utilities.AbstractModel.

When the constraints are added, they are stored in the caches field. They are only loaded in the coefficients and constants fields once MOI.Utilities.final_touch is called. For this reason, MatrixOfConstraints should not be used by an incremental interface. Use MOI.copy_to instead.

The constraints can be added in two different ways:

- 1. With add_constraint, in which case a canonicalized copy of the function is stored in caches.
- 2. With pass_nonvariable_constraints, in which case the functions and sets are stored themselves in caches without mapping the variable indices. The corresponding index in caches is added in are_indices_mapped. This avoids doing a copy of the function in case the getter of CanonicalConstraintFunction does not make a copy for the source model, for example, this is the case of VectorOfConstraints.

We illustrate this with an example. Suppose a model is copied from a src::MOI.Utilities.Model to a bridged model with a MatrixOfConstraints. For all the types that are not bridged, the constraints will be copied with pass_nonvariable_constraints. Hence the functions stored in caches are exactly the same as the ones stored in src. This is ok since this is only during the copy_to operation during which src cannot be modified. On the other hand, for the types that are bridged, the functions added may

contain duplicates even if the functions did not contain duplicates in src so duplicates are removed with MOI.Utilities.canonical.

Interface

The .coefficients::AT type must implement:

- AT()
- MOI.empty(::AT)!
- MOI.Utilities.add_column
- MOI.Utilities.set_number_of_rows
- MOI.Utilities.allocate_terms
- MOI.Utilities.load_terms
- MOI.Utilities.final_touch

The .constants::BT type must implement:

- BT()
- Base.empty!(::BT)
- Base.resize(::BT)
- MOI.Utilities.load_constants
- MOI.Utilities.function constants
- MOI.Utilities.set_from_constants

The .sets::ST type must implement:

- ST()
- MOI.is_empty(::ST)
- MOI.empty(::ST)
- MOI.dimension(::ST)
- MOI.is_valid(::ST, ::MOI.ConstraintIndex)
- MOI.get(::ST, ::MOI.ListOfConstraintTypesPresent)
- MOI.get(::ST, ::MOI.NumberOfConstraints)
- MOI.get(::ST, ::MOI.ListOfConstraintIndices)
- MOI.Utilities.set_types
- MOI.Utilities.set_index
- MOI.Utilities.add_set
- MOI.Utilities.rows
- MOI.Utilities.final_touch

source

.coefficients

MathOptInterface.Utilities.add_column - Function.

add_column(coefficients)::Nothing

Tell coefficients to pre-allocate datastructures as needed to store one column.

source

MathOptInterface.Utilities.allocate_terms - Function.

allocate_terms(coefficients, index_map, func)::Nothing

Tell coefficients that the terms of the function func where the variable indices are mapped with index_map will be loaded with load_terms.

The function func must be canonicalized before calling allocate_terms. See is_canonical.

source

MathOptInterface.Utilities.set_number_of_rows - Function.

set_number_of_rows(coefficients, n)::Nothing

Tell coefficients to pre-allocate datastructures as needed to store n rows.

source

MathOptInterface.Utilities.load_terms - Function.

load_terms(coefficients, index_map, func, offset)::Nothing

Loads the terms of func to coefficients, mapping the variable indices with index_map.

The ith dimension of func is loaded at the (offset + i)th row of coefficients.

The function must be allocated first with allocate_terms.

The function func must be canonicalized, see is canonical.

source

MathOptInterface.Utilities.final_touch - Function.

final_touch(coefficients)::Nothing

Informs the coefficients that all functions have been added with load_terms. No more modification is allowed unless MOI.empty! is called.

final_touch(sets)::Nothing

Informs the sets that all functions have been added with add_set. No more modification is allowed unless MOI.empty! is called.

source

MathOptInterface.Utilities.extract_function - Function.

```
extract_function(coefficients, row::Integer, constant::T) where {T}
```

Return the MOI.ScalarAffineFunction{T} function corresponding to row row in coefficients.

```
extract_function(
    coefficients,
    rows::UnitRange,
    constants::Vector{T},
) where{T}
```

Return the MOI.VectorAffineFunction{T} function corresponding to rows rows in coefficients.

source

MathOptInterface.Utilities.MutableSparseMatrixCSC - Type.

```
mutable struct MutableSparseMatrixCSC{Tv,Ti<:Integer,I<:AbstractIndexing}
    indexing::I
    m::Int
    n::Int
    colptr::Vector{Ti}
    rowval::Vector{Ti}
    nzval::Vector{Tv}
    nz_added::Vector{Ti}
end</pre>
```

ena

Matrix type loading sparse matrices in the Compressed Sparse Column format. The indexing used is indexing, see AbstractIndexing. The other fields have the same meaning than for SparseArrays.SparseMatrixCSC except that the indexing is different unless indexing is OneBasedIndexing. In addition, nz_added is used to cache the number of non-zero terms that have been added to each column due to the incremental nature of load_terms.

The matrix is loaded in 5 steps:

- 1. MOI.empty! is called.
- 2. MOI.Utilities.add_column and MOI.Utilities.allocate_terms are called in any order.
- 3. MOI.Utilities.set_number_of_rows is called.
- 4. MOI.Utilities.load_terms is called for each affine function.
- 5. MOI.Utilities.final_touch is called.

source

MathOptInterface.Utilities.AbstractIndexing - Type.

abstract type AbstractIndexing end

Indexing to be used for storing the row and column indices of MutableSparseMatrixCSC. See ZeroBasedIndexing and OneBasedIndexing.

source

MathOptInterface.Utilities.ZeroBasedIndexing - Type.

struct ZeroBasedIndexing <: AbstractIndexing end</pre>

Zero-based indexing: the ith row or column has index i - 1. This is useful when the vectors of row and column indices need to be communicated to a library using zero-based indexing such as C libraries.

source

MathOptInterface.Utilities.OneBasedIndexing - Type.

struct ZeroBasedIndexing <: AbstractIndexing end</pre>

One-based indexing: the ith row or column has index i. This enables an allocation-free conversion of MutableSparseMatrixCSC to SparseArrays.SparseMatrixCSC.

source

.constants

MathOptInterface.Utilities.load_constants - Function.

load constants(constants, offset, func or set)::Nothing

This function loads the constants of func_or_set in constants at an offset of offset. Where offset is the sum of the dimensions of the constraints already loaded. The storage should be preallocated with resize! before calling this function.

This function should be implemented to be usable as storage of constants for MatrixOfConstraints.

The constants are loaded in three steps:

- 1. Base.empty! is called.
- 2. Base.resize! is called with the sum of the dimensions of all constraints.
- 3. MOI.Utilities.load_constants is called for each function for vector constraint or set for scalar constraint.

source

MathOptInterface.Utilities.function_constants - Function.

function_constants(constants, rows)

This function returns the function constants that were loaded with load_constants at the rows rows. This function should be implemented to be usable as storage of constants for MatrixOfConstraints. source

MathOptInterface.Utilities.set_from_constants - Function.

```
set_from_constants(constants, S::Type, rows)::S
```

This function returns an instance of the set S for which the constants where loaded with load_constants at the rows rows.

This function should be implemented to be usable as storage of constants for MatrixOfConstraints.

source

MathOptInterface.Utilities.modify_constants - Function.

```
modify_constants(constants, row::Integer, new_constant::T) where {T}
modify_constants(
    constants,
    rows::AbstractVector{<:Integer},
    new_constants::AbstractVector{T},
) where {T}</pre>
```

Modify constants in-place to store new_constant in the row row, or rows rows.

This function must be implemented to enable MOI.ScalarConstantChange and MOI.VectorConstantChange for MatrixOfConstraints.

source

MathOptInterface.Utilities.Hyperrectangle - Type.

```
struct Hyperrectangle{T} <: AbstractVectorBounds
    lower::Vector{T}
    upper::Vector{T}
end</pre>
```

A struct for the .constants field in MatrixOfConstraints.

source

.sets

MathOptInterface.Utilities.set_index - Function.

set_index(sets, ::Type{S})::Union{Int,Nothing} where {S<:MOI.AbstractSet}</pre>

Return an integer corresponding to the index of the set type in the list given by ${\tt set_types}.$

If S is not part of the list, return nothing.

source

MathOptInterface.Utilities.set_types - Function.

set_types(sets)::Vector{Type}

Return the list of the types of the sets allowed in sets.

source

MathOptInterface.Utilities.add_set - Function.

add_set(sets, i)::Int64

Add a scalar set of type index i.

add_set(sets, i, dim)::Int64

Add a vector set of type index i and dimension dim.

Both methods return a unique Int64 of the set that can be used to reference this set.

source

MathOptInterface.Utilities.rows - Function.

rows(sets, ci::MOI.ConstraintIndex)::Union{Int,UnitRange{Int}}

Return the rows in 1:MOI.dimension(sets) corresponding to the set of id ci.value.

For scalar sets, this returns an Int. For vector sets, this returns an UnitRange{Int}.

source

MathOptInterface.Utilities.num_rows - Function.

num_rows(sets::OrderedProductOfSets, ::Type{S}) where {S}

Return the number of rows corresponding to a set of type S. That is, it is the sum of the dimensions of the sets of type S.

source

MathOptInterface.Utilities.set_with_dimension - Function.

set_with_dimension(::Type{S}, dim) where {S<:MOI.AbstractVectorSet}</pre>

Returns the instance of S of MOI.dimension dim. This needs to be implemented for sets of type S to be useable with MatrixOfConstraints.

source

MathOptInterface.Utilities.ProductOfSets - Type.

abstract type ProductOfSets{T} end

Represents a cartesian product of sets of given types.

source

MathOptInterface.Utilities.MixOfScalarSets - Type.

abstract type MixOfScalarSets{T} <: ProductOfSets{T} end</pre>

Product of scalar sets in the order the constraints are added, mixing the constraints of different types.

Use @mix_of_scalar_sets to generate a new subtype.

source

MathOptInterface.Utilities.@mix_of_scalar_sets - Macro.

@mix_of_scalar_sets(name, set_types...)

Generate a new MixOfScalarSets subtype.

Example

```
julia> MOI.Utilities.@mix_of_scalar_sets(
        MixedIntegerLinearProgramSets,
        MOI.GreaterThan{T},
        MOI.LessThan{T},
        MOI.EqualTo{T},
        MOI.Integer,
        )
```

source

MathOptInterface.Utilities.OrderedProductOfSets - Type.

abstract type OrderedProductOfSets{T} <: ProductOfSets{T} end</pre>

Product of sets in the order the constraints are added, grouping the constraints of the same types contiguously.

Use @product_of_sets to generate new subtypes.

source

MathOptInterface.Utilities.@product_of_sets - Macro.

```
@product_of_sets(name, set_types...)
```

Generate a new OrderedProductOfSets subtype.

Example

```
julia> MOI.Utilities.@product_of_sets(
        LinearOrthants,
        MOI.Zeros,
        MOI.Nonnegatives,
        MOI.Nonpositives,
        MOI.ZeroOne,
)
```

source

Fallbacks

MathOptInterface.Utilities.get_fallback - Function.

get_fallback(model::MOI.ModelLike, ::MOI.ObjectiveValue)

Compute the objective function value using the VariablePrimal results and the ObjectiveFunction value.

```
source
```

```
get_fallback(
    model::MOI.ModelLike,
    ::MOI.DualObjectiveValue,
    ::Type{T},
)::T where {T}
```

Compute the dual objective value of type T using the ConstraintDual results and the ConstraintFunction and ConstraintSet values.

Note that the nonlinear part of the model is ignored.

source

```
get_fallback(
    model::MOI.ModelLike,
    ::MOI.ConstraintPrimal,
    constraint_index::MOI.ConstraintIndex,
)
```

Compute the value of the function of the constraint of index constraint_index using the VariablePrimal results and the ConstraintFunction values.

```
get_fallback(
    model::MOI.ModelLike,
    attr::MOI.ConstraintDual,
    ci::MOI.ConstraintIndex{Union{MOI.VariableIndex,MOI.VectorOfVariables}},
    ::Type{T} = Float64,
) where {T}
```

Compute the dual of the constraint of index ci using the ConstraintDual of other constraints and the ConstraintFunction values.

Throws an error if some constraints are quadratic or if there is one another MOI.VariableIndex-in-S or MOI.VectorOfVariables-in-S constraint with one of the variables in the function of the constraint ci.

source

source

Function utilities

The following utilities are available for functions:

MathOptInterface.Utilities.eval_variables - Function.

eval_variables(value_fn::Function, f::MOI.AbstractFunction)

Returns the value of function f if each variable index vi is evaluated as value_fn(vi).

Note that value_fn must return a Number. See substitute_variables for a similar function where value_fn returns an MOI.AbstractScalarFunction.

Warning

The two-argument version of eval_variables is deprecated and may be removed in MOI v2.0.0. Use the three-argument method eval_variables(::Function, ::MOI.ModelLike, ::MOI.AbstractFunction) instead.

source

MathOptInterface.Utilities.map_indices - Function.

map_indices(index_map::Function, attr::MOI.AnyAttribute, x::X)::X where {X}

Substitute any MOI.VariableIndex (resp. MOI.ConstraintIndex) in x by the MOI.VariableIndex (resp. MOI.ConstraintIndex) of the same type given by index_map(x).

When to implement this method for new types X

This function is used by implementations of MOI.copy_to on constraint functions, attribute values and submittable values. If you define a new attribute whose values x::X contain variable or constraint indices, you must also implement this function.

source

```
map_indices(
    variable_map::AbstractDict{T,T},
    x::X,
)::X where {T<:MOI.Index,X}</pre>
```

Shortcut for map_indices(vi -> variable_map[vi], x).

source

MathOptInterface.Utilities.substitute_variables - Function.

```
substitute_variables(variable_map::Function, x)
```

Substitute any MOI.VariableIndex in x by variable_map(x). The variable_map function returns either MOI.VariableIndex or MOI.ScalarAffineFunction, see eval_variables for a similar function where variable map returns a number.

This function is used by bridge optimizers on constraint functions, attribute values and submittable values when at least one variable bridge is used hence it needs to be implemented for custom types that are meant to be used as attribute or submittable value.

Note

When implementing a new method, don't use substitute_variables(::Function, because Julia will not specialize on it. Use instead substitute_variables(::F, ...) where {F<:Function}.

source

MathOptInterface.Utilities.filter_variables - Function.

filter_variables(keep::Function, f::AbstractFunction)

Return a new function f with the variable vi such that !keep(vi) removed.

WARNING: Don't define filter_variables(::Function, ...) because Julia will not specialize on this. Define instead filter_variables(::F, ...) where {F<:Function}.

source

MathOptInterface.Utilities.remove_variable - Function.

```
remove_variable(f::AbstractFunction, vi::VariableIndex)
```

Return a new function f with the variable vi removed.

```
source
remove_variable(
    f::MOI.AbstractFunction,
    s::MOI.AbstractSet,
    vi::MOI.VariableIndex,
)
```

Return a tuple (g, t) representing the constraint f-in-s with the variable vi removed. That is, the terms containing the variable vi in the function f are removed and the dimension of the set s is updated if needed (for example, when f is a VectorOfVariables with vi being one of the variables).

source

MathOptInterface.Utilities.all_coefficients - Function.

```
all_coefficients(p::Function, f::MOI.AbstractFunction)
```

Determine whether predicate p returns true for all coefficients of f, returning false as soon as the first coefficient of f for which p returns false is encountered (short-circuiting). Similar to all.

source

MathOptInterface.Utilities.unsafe_add - Function.

unsafe_add(t1::MOI.ScalarAffineTerm, t2::MOI.ScalarAffineTerm)

Sums the coefficients of t1 and t2 and returns an output MOI.ScalarAffineTerm. It is unsafe because it uses the variable of t1 as the variable of the output without checking that it is equal to that of t2.

source

unsafe_add(t1::MOI.ScalarQuadraticTerm, t2::MOI.ScalarQuadraticTerm)

Sums the coefficients of t1 and t2 and returns an output MOI.ScalarQuadraticTerm. It is unsafe because it uses the variable's of t1 as the variable's of the output without checking that they are the same (up to permutation) to those of t2.

source

unsafe_add(t1::MOI.VectorAffineTerm, t2::MOI.VectorAffineTerm)

Sums the coefficients of t1 and t2 and returns an output MOI.VectorAffineTerm. It is unsafe because it uses the output_index and variable of t1 as the output_index and variable of the output term without checking that they are equal to those of t2.

source

```
MathOptInterface.Utilities.isapprox_zero - Function.
```

```
isapprox_zero(f::MOI.AbstractFunction, tol)
```

Return a Bool indicating whether the function f is approximately zero using tol as a tolerance.

Important note

This function assumes that f does not contain any duplicate terms, you might want to first call canonical if that is not guaranteed.

Example

```
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)
```

```
julia> f = MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.([1, -1], [x, x]), 0)
(0) + (1) MOI.VariableIndex(1) - (1) MOI.VariableIndex(1)
```

```
julia> MOI.Utilities.isapprox_zero(f, le-8)
false
```

```
julia> MOI.Utilities.isapprox_zero(MOI.Utilities.canonical(f), le-8)
true
```

source

MathOptInterface.Utilities.modify_function - Function.

modify_function(f::AbstractFunction, change::AbstractFunctionModification)

Return a copy of the function f, modified according to change.

source

MathOptInterface.Utilities.zero_with_output_dimension - Function.

zero_with_output_dimension(::Type{T}, output_dimension::Integer) where {T}

Create an instance of type T with the output dimension output_dimension.

This is mostly useful in Bridges, when code needs to be agnostic to the type of vector-valued function that is passed in.

source

The following functions can be used to canonicalize a function:

MathOptInterface.Utilities.is_canonical - Function.

is_canonical(f::Union{ScalarAffineFunction, VectorAffineFunction})

Returns a Bool indicating whether the function is in canonical form. See canonical.

source

is_canonical(f::Union{ScalarQuadraticFunction, VectorQuadraticFunction})

Returns a Bool indicating whether the function is in canonical form. See canonical.

source

MathOptInterface.Utilities.canonical - Function.

canonical(f::MOI.AbstractFunction)

Returns the function in a canonical form, that is,

- A term appear only once.
- The coefficients are nonzero.
- The terms appear in increasing order of variable where there the order of the variables is the order of their value.
- For a AbstractVectorFunction, the terms are sorted in ascending order of output index.

The output of canonical can be assumed to be a copy of f, even for VectorOfVariables.

Example

julia> MOI.Utilities.canonical(f) 5.0 - 1.0 MOI.VariableIndex(1) + 2.0 MOI.VariableIndex(2)

source

MathOptInterface.Utilities.canonicalize! - Function.

canonicalize!(f::Union{ScalarAffineFunction, VectorAffineFunction})

Convert a function to canonical form in-place, without allocating a copy to hold the result. See canonical.

canonicalize!(f::Union{ScalarQuadraticFunction, VectorQuadraticFunction})

Convert a function to canonical form in-place, without allocating a copy to hold the result. See canonical.

The following functions can be used to manipulate functions with basic algebra: MathOptInterface.Utilities.scalar_type - Function.

scalar_type(F::Type{<:MOI.AbstractVectorFunction})</pre>

Type of functions obtained by indexing objects obtained by calling eachscalar on functions of type F.

source

MathOptInterface.Utilities.scalarize - Function.

scalarize(func::MOI.VectorOfVariables, ignore_constants::Bool = false)

Returns a vector of scalar functions making up the vector function in the form of a Vector{MOI.SingleVariable}. See also eachscalar.

source

scalarize(func::MOI.VectorAffineFunction{T}, ignore_constants::Bool = false)

Returns a vector of scalar functions making up the vector function in the form of a Vector {MOI.ScalarAffineFunction{T}}. See also eachscalar.

source

scalarize(func::MOI.VectorQuadraticFunction{T}, ignore_constants::Bool = false)

Returns a vector of scalar functions making up the vector function in the form of a Vector{MOI.ScalarQuadraticFunction{T} See also eachscalar.

source

MathOptInterface.Utilities.eachscalar - Function.

eachscalar(f::MOI.AbstractVectorFunction)

Returns an iterator for the scalar components of the vector function.

See also scalarize.

eachscalar(f::MOI.AbstractVector)

Returns an iterator for the scalar components of the vector.

source

MathOptInterface.Utilities.promote_operation - Function.

```
promote_operation(
    op::Function,
    ::Type{T},
    ArgsTypes::Type{<:Union{T,AbstractVector{T},MOI.AbstractFunction}}...,
) where {T<:Number}</pre>
```

Compute the return type of the call operate(op, T, args...), where the types of the arguments args are ArgsTypes.

One assumption is that the element type T is invariant under each operation. That is, op(::T, ::T)::T where op is a +, -, *, and /.

There are six methods for which we implement Utilities.promote_operation:

- 1. + a. promote_operation(::typeof(+), ::Type{T}, ::Type{F1}, ::Type{F2})
- 2. -a.promote_operation(::typeof(-), ::Type{T}, ::Type{F}) b.promote_operation(::typeof(-), ::Type{T}, ::Type{F1}, ::Type{F2})
- 3. *a.promote_operation(::typeof(*), ::Type{T}, ::Type{T}, ::Type{F}) b.promote_operation(::typeof(*), ::Type{T}, ::Type{F}, ::Type{T}) c. promote_operation(::typeof(*), ::Type{T}, ::Type{F1}, ::Type{F2}) where F1 and F2 are VariableIndex or ScalarAffineFunction d. promote_operation(::typeof(*), ::Type{T}, ::Type{<}, ::Type{F}</pre>
- 4. / a. promote_operation(::typeof(/), ::Type{T}, ::Type{F}, ::Type{T})
- 5. vcat a. promote_operation(::typeof(vcat), ::Type{T}, ::Type{F}...)
- 6. imag a. promote_operation(::typeof(imag), ::Type{T}, ::Type{F}) where F is VariableIndex
 or VectorOfVariables

In each case, F (or F1 and F2) is one of the ten supported types, with a restriction that the mathematical operation makes sense, for example, we don't define promote_operation(-, T, F1, F2) where F1 is a scalar-valued function and F2 is a vector-valued function. The ten supported types are:

- 1. ::T
- 2. ::VariableIndex
- 3. ::ScalarAffineFunction{T}
- 4. ::ScalarQuadraticFunction{T}
- 5. ::ScalarNonlinearFunction
- 6. ::AbstractVector{T}
- 7. ::VectorOfVariables
- 8. ::VectorAffineFunction{T}
- 9. ::VectorQuadraticFunction{T}

10. ::VectorNonlinearFunction

source

MathOptInterface.Utilities.operate - Function.

```
operate(
    op::Function,
    ::Type{T},
    args::Union{T,MOI.AbstractFunction}...,
)::MOI.AbstractFunction where {T<:Number}</pre>
```

Returns an MOI.AbstractFunction representing the function resulting from the operation op(args...) on functions of coefficient type T.

No argument can be modified.

Methods

- 1. +a. operate(::typeof(+), ::Type{T}, ::F1) b. operate(::typeof(+), ::Type{T}, ::F1, ::F2)
 c. operate(::typeof(+), ::Type{T}, ::F1...)
- 2. a. operate(::typeof(-), ::Type{T}, ::F) b. operate(::typeof(-), ::Type{T}, ::F1, ::F2)
- 4. / a. operate(::typeof(/), ::Type{T}, ::F, ::T)
- 5. vcat a. operate(::typeof(vcat), ::Type{T}, ::F...)
- 6. imag a. operate(::typeof(imag), ::Type{T}, ::F) where F is VariableIndex or VectorOfVariables

One assumption is that the element type T is invariant under each operation. That is, op(::T, ::T)::T where op is a +, -, *, and /.

In each case, F (or F1 and F2) is one of the ten supported types, with a restriction that the mathematical operation makes sense, for example, we don't define promote_operation(-, T, F1, F2) where F1 is a scalar-valued function and F2 is a vector-valued function. The ten supported types are:

- 1. ::T
- 2. ::VariableIndex
- 3. ::ScalarAffineFunction{T}
- 4. ::ScalarQuadraticFunction{T}
- 5. ::ScalarNonlinearFunction
- 6. ::AbstractVector{T}
- 7. ::VectorOfVariables
- 8. ::VectorAffineFunction{T}
- 9. ::VectorQuadraticFunction{T}
- 10. ::VectorNonlinearFunction

MathOptInterface.Utilities.operate! - Function.

```
operate!(
    op::Function,
    ::Type{T},
    args::Union{T,MOI.AbstractFunction}...,
)::MOI.AbstractFunction where {T<:Number}</pre>
```

Returns an MOI.AbstractFunction representing the function resulting from the operation op(args...) on functions of coefficient type T.

The first argument may be modified, in which case the return value is identical to the first argument. For operations which cannot be implemented in-place, this function returns a new object.

source

MathOptInterface.Utilities.operate_output_index! - Function.

```
operate_output_index!(
    op::Union{typeof(+),typeof(-)},
    ::Type{T},
    output_index::Integer,
    f::Union{AbstractVector{T},MOI.AbstractVectorFunction}
    g::Union{T,MOI.AbstractScalarFunction}...
) where {T<:Number}</pre>
```

Return an MOI.AbstractVectorFunction in which the scalar function in row output_index is the result of op(f[output_index], g).

The functions at output index different to output_index are the same as the functions at the same output index in func. The first argument may be modified.

Methods

```
    + a. operate_output_index!(+, ::Type{T}, ::Int, ::VectorF, ::ScalarF)
    - a. operate output index!(-, ::Type{T}, ::Int, ::VectorF, ::ScalarF)
```

source

MathOptInterface.Utilities.vectorize - Function.

vectorize(x::AbstractVector{<:Number})</pre>

Returns x.

source

vectorize(x::AbstractVector{MOI.VariableIndex})

Returns the vector of scalar affine functions in the form of a MOI. VectorAffineFunction{T}.

source

vectorize(funcs::AbstractVector{MOI.ScalarAffineFunction{T}}) where T

Returns the vector of scalar affine functions in the form of a MOI.VectorAffineFunction{T}.

source

vectorize(funcs::AbstractVector{MOI.ScalarQuadraticFunction{T}}) where T

Returns the vector of scalar quadratic functions in the form of a MOI. VectorQuadraticFunction $\{T\}$.

source

Constraint utilities

The following utilities are available for moving the function constant to the set for scalar constraints: MathOptInterface.Utilities.shift_constant - Function.

```
shift_constant(set::MOI.AbstractScalarSet, offset)
```

Returns a new scalar set new_set such that func-in-set is equivalent to func + offset-in-new_set.

Use supports_shift_constant to check if the set supports shifting:

```
if MOI.Utilities.supports_shift_constant(typeof(set))
    new_set = MOI.Utilities.shift_constant(set, -func.constant)
    func.constant = 0
    MOI.add_constraint(model, func, new_set)
else
    MOI.add_constraint(model, func, set)
end
```

Note for developers

Only define this function if it makes sense and you have implemented supports_shift_constant to return true.

Example

```
julia> set = MOI.Interval(-2.0, 3.0)
MathOptInterface.Interval{Float64}(-2.0, 3.0)
```

julia> MOI.Utilities.supports_shift_constant(typeof(set))
true

```
julia> MOI.Utilities.shift_constant(set, 1.0)
MathOptInterface.Interval{Float64}(-1.0, 4.0)
```

source

MathOptInterface.Utilities.supports_shift_constant - Function.

```
supports_shift_constant(::Type{S}) where {S<:MOI.AbstractSet}</pre>
```

Return true if shift_constant is defined for set S.

See also shift_constant.

Example

```
julia> MOI.Utilities.supports_shift_constant(MOI.Interval{Float64})
true
```

julia> MOI.Utilities.supports_shift_constant(MOI.ZeroOne)
false

source

MathOptInterface.Utilities.normalize_and_add_constraint - Function.

```
normalize_and_add_constraint(
    model::MOI.ModelLike,
    func::MOI.AbstractScalarFunction,
    set::MOI.AbstractScalarSet;
    allow_modify_function::Bool = false,
)
```

Adds the scalar constraint obtained by moving the constant term in func to the set in model. If allow_modify_function is true then the function func can be modified.

source

MathOptInterface.Utilities.normalize_constant - Function.

```
normalize_constant(
    func::MOI.AbstractScalarFunction,
    set::MOI.AbstractScalarSet;
    allow_modify_function::Bool = false,
)
```

Return the func-in-set constraint in normalized form. That is, if func is MOI.ScalarQuadraticFunction or MOI.ScalarAffineFunction, the constant is moved to the set. If allow_modify_function is true then the function func can be modified.

source

The following utility identifies those constraints imposing bounds on a given variable, and returns those bound values:

MathOptInterface.Utilities.get_bounds - Function.

```
get_bounds(model::MOI.ModelLike, ::Type{T}, x::MOI.VariableIndex)
```

Return a tuple (lb, ub) of type Tuple{T, T}, where lb and ub are lower and upper bounds, respectively, imposed on x in model.

source

```
get_bounds(
    model::MOI.ModelLike,
    bounds_cache::Dict{MOI.VariableIndex,NTuple{2,T}},
    f::MOI.ScalarAffineFunction{T},
) where {T} --> Union{Nothing,NTuple{2,T}}
```

Return the lower and upper bound of f as a tuple. If the domain is not bounded, return nothing.

```
source
```

```
get_bounds(
    model::MOI.ModelLike,
    bounds_cache::Dict{MOI.VariableIndex,NTuple{2,T}},
    x::MOI.VariableIndex,
) where {T} --> Union{Nothing,NTuple{2,T}}
```

Return the lower and upper bound of x as a tuple. If the domain is not bounded, return nothing.

Similar to get_bounds(::MOI.ModelLike, ::Type{T}, ::MOI.VariableIndex), except that the second argument is a cache which maps variables to their bounds and avoids repeated lookups.

source

The following utilities are useful when working with symmetric matrix cones.

MathOptInterface.Utilities.is_diagonal_vectorized_index - Function.

is_diagonal_vectorized_index(index::Base.Integer)

Return whether index is the index of a diagonal element in a MOI.AbstractSymmetricMatrixSetTriangle set.

source

MathOptInterface.Utilities.side_dimension_for_vectorized_dimension - Function.

side_dimension_for_vectorized_dimension(n::Integer)

Return the dimension d such that MOI.dimension(MOI.PositiveSemidefiniteConeTriangle(d)) is n.

source

Set utilities

The following utilities are available for sets:

```
MathOptInterface.Utilities.AbstractDistance - Type.
```

abstract type AbstractDistance end

An abstract type used to enable dispatch of Utilities.distance_to_set.

source

MathOptInterface.Utilities.ProjectionUpperBoundDistance - Type.

```
ProjectionUpperBoundDistance() <: AbstractDistance</pre>
```

An upper bound on the minimum distance between point and the closest feasible point in set.

Definition of distance

The minimum distance is computed as:

$$d(x,\mathcal{K}) = \min_{y \in \mathcal{K}} ||x - y||$$

where x is point and ${\mathcal K}$ is set. The norm is computed as:

$$||x|| = \sqrt{f(x, x, \mathcal{K})}$$

where f is Utilities.set dot.

In the default case, where the set does not have a specialized method for Utilities.set_dot, the norm is equivalent to the Euclidean norm $||x|| = \sqrt{\sum x_i^2}$.

Why an upper bound?

In most cases, distance_to_set should return the smallest upper bound, but it may return a larger value if the smallest upper bound is expensive to compute.

For example, given an epigraph from of a conic set, $\{(t, x)|f(x) \le t\}$, it may be simpler to return δ such that $f(x) \le t + \delta$, rather than computing the nearest projection onto the set.

If the distance is not the smallest upper bound, the docstring of the appropriate distance_to_set method *must* describe the way that the distance is computed.

source

MathOptInterface.Utilities.distance_to_set - Function.

```
distance_to_set(
    [d::AbstractDistance = ProjectionUpperBoundDistance()],]
    point::T,
    set::MOI.AbstractScalarSet,
```

```
) where {T}
distance_to_set(
    [d::AbstractDistance = ProjectionUpperBoundDistance(),]
    point::AbstractVector{T},
    set::MOI.AbstractVectorSet,
) where {T}
```

Compute the distance between point and set using the distance metric d. If point is in the set set, this function *must* return zero(T).

If d is omitted, the default distance is Utilities.ProjectionUpperBoundDistance.

source

distance to set(::**ProjectionUpperBoundDistance**, x, ::**MOI**.RotatedSecondOrderCone)

Let (t, u, y...) = x. Return the 2-norm of the vector d such that in x + d, u is projected to 1 if u <= 0, and t is increased such that x + d belongs to the set.

source

distance_to_set(::ProjectionUpperBoundDistance, x, ::MOI.ExponentialCone)

Let (u, v, w) = x. If v > 0, return the epigraph distance d such that (u, v, w + d) belongs to the set. If $v \le 0$ return the 2-norm of the vector d such that x + d = (u, 1, z) where z satisfies the constraints. source

distance_to_set(::ProjectionUpperBoundDistance, x, ::MOI.DualExponentialCone)

Let (u, v, w) = x. If u < 0, return the epigraph distance d such that (u, v, w + d) belongs to the set. If $u \ge 0$ return the 2-norm of the vector d such that x + d = (u, -1, z) where z satisfies the constraints. source

distance_to_set(::ProjectionUpperBoundDistance, x, ::MOI.GeometricMeanCone)

Let (t, y...) = x. If all y are non-negative, return the epigraph distance d such that (t + d, y...) belongs to the set.

If any y are strictly negative, return the 2-norm of the vector d that projects negative y elements to 0 and t to \mathbb{R}_- .

source

distance_to_set(::ProjectionUpperBoundDistance, x, ::MOI.PowerCone)

Let (a, b, c) = x. If a and b are non-negative, return the epigraph distance required to increase c such that the constraint is satisfied.

If a or b is strictly negative, return the 2-norm of the vector d such that in the vector x + d: c, and any negative a and b are projected to 0.

source

distance_to_set(::ProjectionUpperBoundDistance, x, ::MOI.DualPowerCone)

Let (a, b, c) = x. If a and b are non-negative, return the epigraph distance required to increase c such that the constraint is satisfied.

If a or b is strictly negative, return the 2-norm of the vector d such that in the vector x + d: c, and any negative a and b are projected to 0.

source

distance_to_set(::ProjectionUpperBoundDistance, x, ::MOI.NormOneCone)

Let (t, y...) = x. Return the epigraph distance d such that (t + d, y...) belongs to the set.

source

distance_to_set(::ProjectionUpperBoundDistance, x, ::MOI.NormInfinityCone)

Let (t, y...) = x. Return the epigraph distance d such that (t + d, y...) belongs to the set.

source

distance_to_set(::ProjectionUpperBoundDistance, x, ::MOI.RelativeEntropyCone)

Let (u, v..., w...) = x. If v and w are strictly positive, return the epigraph distance required to increase u such that the constraint is satisfied.

If any elements in v or w are non-positive, return the 2-norm of the vector d such that in the vector x + d: any non-positive elements in v and w are projected to 1, and u is projected such that the epigraph constraint holds.

source

distance_to_set(::ProjectionUpperBoundDistance, x, set::MOI.NormCone)

Let (t, y...) = x. Return the epigraph distance d such that (t + d, y...) belongs to the set.

source

```
distance_to_set(
    ::ProjectionUpperBoundDistance,
    x::AbstractVector,
    set::Union{
        MOI.PositiveSemidefiniteConeSquare,
        MOI.PositiveSemidefiniteConeTriangle,
    },
)
```

Let X be x reshaped into the appropriate matrix. The returned distance is $||X - Y||_2^2$ where Y is the eigen decomposition of X with negative eigen values removed.

source

MathOptInterface.Utilities.set_dot - Function.

set dot(x::AbstractVector, y::AbstractVector, set::AbstractVectorSet)

Return the scalar product between a vector x of the set set and a vector y of the dual of the set s.

source

set_dot(x, y, set::AbstractScalarSet)

Return the scalar product between a number x of the set set and a number y of the dual of the set s.

source

DoubleDicts

MathOptInterface.Utilities.DoubleDicts.DoubleDict - Type.

DoubleDict{V}

An optimized dictionary to map MOI.ConstraintIndex to values of type V.

Works as a AbstractDict{MOI.ConstraintIndex,V} with minimal differences.

If V is also a MOI.ConstraintIndex, use IndexDoubleDict.

Note that MOI.ConstraintIndex is not a concrete type, opposed to MOI.ConstraintIndex{MOI.VariableIndex, MOI.Integers}, which is a concrete type.

When looping through multiple keys of the same Function-in-Set type, use

inner = dict[F, S]

to return a type-stable DoubleDictInner.

source

MathOptInterface.Utilities.DoubleDicts.DoubleDictInner - Type.

DoubleDictInner{F,S,V}

A type stable inner dictionary of DoubleDict.

source

MathOptInterface.Utilities.DoubleDicts.IndexDoubleDict - Type.

IndexDoubleDict

A specialized version of [DoubleDict] in which the values are of type MOI.ConstraintIndex When looping through multiple keys of the same Function-in-Set type, use

inner = dict[F, S]

to return a type-stable IndexDoubleDictInner.

source

MathOptInterface.Utilities.DoubleDicts.IndexDoubleDictInner - Type.

IndexDoubleDictInner{F,S}

A type stable inner dictionary of IndexDoubleDict.

source

MathOptInterface.Utilities.DoubleDicts.outer_keys - Function.

```
outer_keys(d::AbstractDoubleDict)
```

Return an iterator over the outer keys of the AbstractDoubleDict d. Each outer key is a Tuple{Type, Type} so that a double loop can be easily used:

```
for (F, S) in DoubleDicts.outer_keys(dict)
    for (k, v) in dict[F, S]
        # ...
    end
end
```

For performance, it is recommended that the inner loop lies in a separate function to guarantee typestability. Some outer keys (F, S) might lead to an empty dict[F, S]. If you want only nonempty dict[F, S], use nonempty_outer_keys.

source

MathOptInterface.Utilities.DoubleDicts.nonempty_outer_keys - Function.

nonempty_outer_keys(d::AbstractDoubleDict)

Return a vector of outer keys of the AbstractDoubleDict d.

Only outer keys that have a nonempty set of inner keys will be returned.

Each outer key is a Tuple{Type, Type} so that a double loop can be easily used

```
for (F, S) in DoubleDicts.nonempty_outer_keys(dict)
    for (k, v) in dict[F, S]
        # ...
    end
end
For performance, it is recommended that the inner loop lies in a separate
function to guarantee type-stability.
```

If you want an iterator of all current outer keys, use [`outer_keys`](@ref).

source

37.6 Test

Overview

The Test submodule

The Test submodule provides tools to help solvers implement unit tests in order to ensure they implement the MathOptInterface API correctly, and to check for solver-correctness.

We use a centralized repository of tests, so that if we find a bug in one solver, instead of adding a test to that particular repository, we add it here so that all solvers can benefit.

How to test a solver

The skeleton below can be used for the wrapper test file of a solver named FooBar.

```
module TestFooBar
import FooBar
using Test
import MathOptInterface as MOI
const OPTIMIZER = MOI.instantiate(
   MOI.OptimizerWithAttributes(FooBar.Optimizer, MOI.Silent() => true),
)
const BRIDGED = MOI.instantiate(
   MOI.OptimizerWithAttributes(FooBar.Optimizer, MOI.Silent() => true),
   with_bridge_type = Float64,
)
# See the docstring of MOI.Test.Config for other arguments.
const CONFIG = MOI.Test.Config(
   # Modify tolerances as necessary.
   atol = 1e-6,
   rtol = 1e-6,
   # Use MOI.LOCALLY_SOLVED for local solvers.
   optimal_status = MOI.OPTIMAL,
   # Pass attributes or MOI functions to `exclude` to skip tests that
   # rely on this functionality.
```

```
exclude = Any[MOI.VariableName, MOI.delete],
)
.....
    runtests()
This function runs all functions in the this Module starting with `test_`.
.....
function runtests()
    for name in names(@__MODULE__; all = true)
        if startswith("$(name)", "test_")
            @testset "$(name)" begin
                getfield(@__MODULE__, name)()
            end
        end
    end
end
.....
    test_runtests()
This function runs all the tests in MathOptInterface.Test.
Pass arguments to `exclude` to skip tests for functionality that is not
implemented or that your solver doesn't support.
.....
function test_runtests()
    MOI.Test.runtests(
        BRIDGED,
        CONFIG,
        exclude = [
            "test attribute NumberOfThreads",
            "test_quadratic_",
        ],
        # This argument is useful to prevent tests from failing on future
        # releases of MOI that add new tests. Don't let this number get too far
        # behind the current MOI release though. You should periodically check
        # for new tests to fix bugs and implement new features.
        exclude_tests_after = v"0.10.5",
    )
    return
end
.....
    test SolverName()
You can also write new tests for solver-specific functionality. Write each new
test as a function with a name beginning with `test_`.
0.0.0
function test_SolverName()
    @test MOI.get(FooBar.Optimizer(), MOI.SolverName()) == "FooBar"
    return
end
end # module TestFooBar
```

CHAPTER 37. SUBMODULES

```
# This line at the end of the file runs all the tests!
TestFooBar.runtests()
```

Then modify your runtests.jl file to include the MOI_wrapper.jl file:

```
using Test
```

```
@testset "MOI" begin
    include("test/MOI_wrapper.jl")
end
```

Info

The optimizer BRIDGED constructed with instantiate automatically bridges constraints that are not supported by OPTIMIZER using the bridges listed in Bridges. It is recommended for an implementation of MOI to only support constraints that are natively supported by the solver and let bridges transform the constraint to the appropriate form. For this reason it is expected that tests may not pass if OPTIMIZER is used instead of BRIDGED.

How to debug a failing test

When writing a solver, it's likely that you will initially fail many tests. Some failures will be bugs, but other failures you may choose to exclude.

There are two ways to exclude tests:

• Exclude tests whose names contain a string using:

```
MOI.Test.runtests(
    model,
    config;
    exclude = String["test_to_exclude", "test_conic_"],
)
```

This will exclude tests whose name contains either of the two strings provided.

• Exclude tests which rely on specific functionality using:

```
MOI.Test.Config(exclude = Any[MOI.VariableName, MOI.optimize!])
```

This will exclude tests which use the MOI.VariableName attribute, or which call MOI.optimize!.

Each test that fails can be independently called as:

```
model = FooBar.Optimizer()
config = MOI.Test.Config()
MOI.empty!(model)
MOI.Test.test_category_name_that_failed(model, config)
```

You can look-up the source code of the test that failed by searching for it in the src/Test/test_category.jl file.



Periodically, you should re-run excluded tests to see if they now pass. The easiest way to do this is to swap the exclude keyword argument of runtests to include. For example:

```
MOI.Test.runtests(
    model,
    config;
    exclude = String["test_to_exclude", "test_conic_"],
)
```

becomes

```
MOI.Test.runtests(
    model,
    config;
    include = String["test_to_exclude", "test_conic_"],
)
```

How to add a test

To detect bugs in solvers, we add new tests to MOI.Test.

As an example, ECOS errored calling optimize! twice in a row. (See ECOS.jl PR #72.) We could add a test to ECOS.jl, but that would only stop us from re-introducing the bug to ECOS.jl in the future, but it would not catch other solvers in the ecosystem with the same bug. Instead, if we add a test to MOI.Test, then all solvers will also check that they handle a double optimize call.

For this test, we care about correctness, rather than performance. therefore, we don't expect solvers to efficiently decide that they have already solved the problem, only that calling optimize! twice doesn't throw an error or give the wrong answer.

Step 1

Install the MathOptInterface julia package in dev mode:

```
julia> ]
(@v1.6) pkg> dev MathOptInterface
```
From here on, proceed with making the following changes in the ~/.julia/dev/MathOptInterface folder (or equivalent dev path on your machine).

Step 3

Since the double-optimize error involves solving an optimization problem, add a new test to src/Test/test_solve.jl:

```
.....
   test_unit_optimize!_twice(model::MOI.ModelLike, config::Config)
Test that calling `MOI.optimize!` twice does not error.
This problem was first detected in ECOS.jl PR#72:
https://github.com/jump-dev/ECOS.jl/pull/72
......
function test_unit_optimize!_twice(
   model::MOI.ModelLike,
   config::Config{T},
) where {T}
   # Use the `@requires` macro to check conditions that the test function
   # requires to run. Models failing this `@requires` check will silently skip
   # the test.
   @requires MOI.supports_constraint(
       model,
       MOI.VariableIndex,
       MOI.GreaterThan{Float64},
   )
   @requires _supports(config, MOI.optimize!)
   # If needed, you can test that the model is empty at the start of the test.
   # You can assume that this will be the case for tests run via `runtests`.
   # User's calling tests individually need to call `MOI.empty!` themselves.
   @test MOI.is empty(model)
   # Create a simple model. Try to make this as simple as possible so that the
   # majority of solvers can run the test.
   x = MOI.add_variable(model)
   MOI.add_constraint(model, x, MOI.GreaterThan(one(T)))
   MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)
   MOI.set(
        model.
       MOI.ObjectiveFunction{MOI.VariableIndex}(),
        х,
   )
   # The main component of the test: does calling `optimize!` twice error?
   MOI.optimize!(model)
   MOI.optimize!(model)
   # Check we have a solution.
   @test MOI.get(model, MOI.TerminationStatus()) == MOI.OPTIMAL
   # There is a three-argument version of `Base.isapprox` for checking
   # approximate equality based on the tolerances defined in `config`:
   @test isapprox(MOI.get(model, MOI.VariablePrimal(), x), one(T), config)
   # For code-style, these tests should always `return` `nothing`.
    return
end
```

Info

Make sure the function is agnostic to the number type T; don't assume it is a Float64 capable solver.

We also need to write a test for the test. Place this function immediately below the test you just wrote in the same file:

```
function setup_test(
    ::typeof(test_unit_optimize!_twice),
    model::MOI.Utilities.MockOptimizer,
    ::Config,
)

MOI.Utilities.set_mock_optimize!(
    model,
    (mock::MOI.Utilities.MockOptimizer) -> MOIU.mock_optimize!(
    mock,
    MOI.OPTIMAL,
    (MOI.FEASIBLE_POINT, [1.0]),
    ),
    )
    return
end
```

Finally, you also need to implement Test.version_added. If we added this test when the latest released version of MOI was v0.10.5, define:

```
version_added(::typeof(test_unit_optimize!_twice)) = v"0.10.6"
```

Step 6

Commit the changes to git from ~/.julia/dev/MathOptInterface and submit the PR for review.

Tip If you need help writing a test, open an issue on GitHub, or ask the Developer Chatroom.

API Reference

The Test submodule

Functions to help test implementations of MOI. See The Test submodule for more details.

MathOptInterface.Test.Config - Type.

```
Config(
  ::Type{T} = Float64;
  atol::Real = Base.rtoldefault(T),
  rtol::Real = Base.rtoldefault(T),
  optimal_status::MOI.TerminationStatusCode = MOI.OPTIMAL,
  infeasible_status::MOI.TerminationStatusCode = MOI.INFEASIBLE,
  exclude::Vector{Any} = Any[],
) where {T}
```

Return an object that is used to configure various tests.

Configuration arguments

- atol::Real = Base.rtoldefault(T): Control the absolute tolerance used when comparing solutions.
- rtol::Real = Base.rtoldefault(T): Control the relative tolerance used when comparing solutions.
- optimal_status = MOI.OPTIMAL: Set to MOI.LOCALLY_SOLVED if the solver cannot prove global optimality.
- infeasible_status = MOI.INFEASIBLE: Set to MOI.LOCALLY_INFEASIBLE if the solver cannot prove global infeasibility.
- exclude = Vector{Any}: Pass attributes or functions to exclude to skip parts of tests that require certain functionality. Common arguments include:
 - MOI.delete to skip deletion-related tests
 - MOI.optimize! to skip optimize-related tests
 - MOI.ConstraintDual to skip dual-related tests
 - MOI.VariableName to skip setting variable names
 - MOI.ConstraintName to skip setting constraint names

Example

For a nonlinear solver that finds local optima and does not support finding dual variables or constraint names:

source

MathOptInterface.Test.runtests - Function.

```
runtests(
    model::MOI.ModelLike,
    config::Config;
    include::Vector{Union{String,Regex}} = String[],
    exclude::Vector{Union{String,Regex}} = String[],
    warn_unsupported::Bool = false,
    exclude_tests_after::VersionNumber = v"999.0.0",
    verbose::Bool = false,
    test_module = MathOptInterface.Test,
)
```

Run all tests in test_module, which defaults to MathOptInterface.Test, on model.

Configuration arguments

- config is a Test.Config object that can be used to modify the behavior of tests.
- If include is not empty, only run tests if an element from include occursin the name of the test.
- If exclude is not empty, skip tests if an element from exclude occursin the name of the test.
- exclude takes priority over include.
- If warn_unsupported is false, runtests will silently skip tests that fail with a MOI.NotAllowedError, MOI.UnsupportedError, or RequirementUnmet error. (The latter is thrown when an @requires statement returns false.) When warn_unsupported is true, a warning will be printed. For most cases the default behavior, false, is what you want, since these tests likely test functionality that is not supported by model. However, it can be useful to run warn_unsupported = true to check you are not skipping tests due to a missing supports_constraint method or equivalent.
- exclude_tests_after is a version number that excludes any tests to MOI added after that version number. This is useful for solvers who can declare a fixed set of tests, and not cause their tests to break if a new patch of MOI is released with a new test.
- verbose is a Bool that controls whether the name of the test is printed before executing it. This can be helpful when debugging.
- test_module is a Module where all the functions starting with test_ are considered as tests.

See also: setup_test.

Example

```
config = MathOptInterface.Test.Config()
MathOptInterface.Test.runtests(
    model,
    config;
    include = ["test_linear_", r"^test_model_Name$"],
    exclude = ["VariablePrimalStart"],
    warn_unsupported = true,
    verbose = true,
    exclude_tests_after = v"0.10.5",
)
```

source

MathOptInterface.Test.setup_test - Function.

setup_test(::typeof(f), model::MOI.ModelLike, config::Config)

Overload this method to modify model before running the test function f on model with config. You can also modify the fields in config (for example, to loosen the default tolerances).

This function should either return nothing, or return a function which, when called with zero arguments, undoes the setup to return the model to its previous state. You do not need to undo any modifications to config.

This function is most useful when writing new tests of the tests for MOI, but it can also be used to set test-specific tolerances, etc.

```
See also: runtests
```

Example

```
function MOI.Test.setup test(
    ::typeof(MOI.Test.test_linear_VariablePrimalStart_partial),
    mock::MOIU.MockOptimizer,
    ::MOI.Test.Config,
)
    MOIU.set_mock_optimize!(
        mock,
        (mock::MOIU.MockOptimizer) -> MOIU.mock_optimize!(mock, [1.0, 0.0]),
    )
    mock.eval_variable_constraint_dual = false
    function reset_function()
        mock.eval_variable_constraint_dual = true
        return
    end
    return reset_function
end
```

source

MathOptInterface.Test.version_added - Function.

```
version_added(::typeof(function_name))
```

Returns the version of MOI in which the test function_name was added.

This method should be implemented for all new tests.

See the exclude_tests_after keyword of runtests for more details.

source

MathOptInterface.Test.@requires - Macro.

@requires(x)

Check that the condition x is true. Otherwise, throw an RequirementUnmet error to indicate that the model does not support something required by the test function.

Example

```
@requires MOI.supports(model, MOI.Silent())
@test MOI.get(model, MOI.Silent())
```

source

MathOptInterface.Test.RequirementUnmet - Type.

```
RequirementUnmet(msg::String) <: Exception</pre>
```

An error for throwing in tests to indicate that the model does not support some requirement expected by the test function.

source

MathOptInterface.Test.HS071 - Type.

```
HS071(
    enable_hessian::Bool,
    enable_hessian_vector_product::Bool = false,
)
```

An MOI.AbstractNLPEvaluator for the problem:

$$\begin{split} \min x_1 * x_4 * (x_1 + x_2 + x_3) + x_3 \\ \text{subject to} & x_1 * x_2 * x_3 * x_4 \geq 25 \\ & x_1^2 + x_2^2 + x_3^2 + x_4^2 = 40 \\ & 1 \leq x_1, x_2, x_3, x_4 \leq 5 \end{split}$$

The optimal solution is [1.000, 4.743, 3.821, 1.379].

source

Chapter 38

Developer Docs

38.1 Checklists

The purpose of this page is to collate a series of checklists for commonly performed changes to the source code of MathOptInterface.

In each case, copy the checklist into the description of the pull request.

Making a release

Use this checklist when making a release of the MathOptInterface repository.

Basic

- [] `version` field of `Project.toml` has been updated
 - If a breaking change, increment the MAJOR field and reset others to $\ensuremath{0}$
 - If adding new features, increment the MINOR field and reset PATCH to $\boldsymbol{\theta}$
 - If adding bug fixes or documentation changes, increment the PATCH field

Documentation

- [] Add a new entry to `docs/src/changelog.md`, following existing style

Tests

- [] The `solver-tests.yml` GitHub action does not have unexpected failures. To run the action, go to: https://github.com/jump-dev/MathOptInterface.jl/actions/workflows/solver-tests.yml and click "Run workflow"
- [] If new tests were added, ensure that `MOI.Test.version_added` is implemented.

Adding a new set

Use this checklist when adding a new set to the MathOptInterface repository.

Basic

- [] Add a new `AbstractScalarSet` or `AbstractVectorSet` to `src/sets.jl`

- [] If `isbitstype(S) == false`, implement `Base.copy(set::S)`

- [] If `isbitstype(S) == false`, implement `Base.:(==)(x::S, y::S)`
- [] If an `AbstractVectorSet`, implement `dimension(set::S)`, unless the dimension is given by `set.dimension`.

Utilities

- [] If an `AbstractVectorSet`, implement `Utilities.set_dot`, unless the dot product between two vectors in the set is equivalent to `LinearAlgebra.dot`
- [] If an `AbstractVectorSet`, implement `Utilities.set_with_dimension` in `src/Utilities/matrix_of_constraints.jl`
- [] Add the set to the `@model` macro at the bottom of `src/Utilities.model.jl`

Documentation

- [] Add a docstring, which gives the mathematical definition of the set, along with an `## Example` block containing a `jldoctest`
- [] Add the docstring to `docs/src/reference/standard form.md`
- [] Add the set to the relevant table in `docs/src/manual/standard_form.md`

Tests

- [] Define a new `_set(::Type{S})` method in `src/Test/test_basic_constraint.jl` and add the name of the set to the list at the bottom of that files
- [] If the set has any checks in its constructor, add tests to `test/sets.jl`

MathOptFormat

- [] Open an issue at `https://github.com/jump-dev/MathOptFormat` to add support for the new set {{ replace with link to the issue }}

Optional

- [] Implement `dual_set(::S)` and `dual_set_type(::Type{S})`
- [] Add new tests to the `Test` submodule exercising your new set
- [] Add new bridges to convert your set into more commonly used sets

Adding a new bridge

Use this checklist when adding a new bridge to the MathOptInterface repository.

The steps are mostly the same, but locations depend on whether the bridge is a Constraint, Objective, or Variable bridge. In each case below, replace XXX with the appropriate type of bridge.

Basic

- [] Create a new file in `src/Bridges/XXX/bridges` named after the type of the bridge
- [] Define the bridge, following existing examples. The name of the bridge struct must end in `Bridge`
- [] Check if your bridge can be a subtype of [`MOI.Bridges.Constraint.SetMapBridge`](@ref)
- [] Define a new `const` that is a `SingleBridgeOptimizer` wrapping the new bridge. The name of the const must be the name of the bridge, less the `Bridge` suffix

 [] If the bridge should be enabled by default, add the bridge to `add_all_bridges` at the bottom of `src/Bridges/XXX/XXX.jl`

Tests

- [] Create a new file in the appropriate subdirectory of `tests/Bridges/XXX` named after the type of the bridge
- [] Use `MOI.Bridges.runtests` to test various inputs and outputs of the bridge
- [] If, after opening the pull request to add the bridge, some lines are not covered by the tests, add additional bridge-specific tests to cover the untested lines.

Documentation

- [] Add a docstring which uses the same template as existing bridges.

```
## Final touch
```

If the bridge depends on run-time values of other variables and constraints in the model:

- [] Implement `MOI.Utilities.needs_final_touch(::Bridge)`
- [] Implement `MOI.Utilities.final_touch(::Bridge, ::MOI.ModelLike)`
- [] Ensure that `final_touch` can be called multiple times in a row

Updating MathOptFormat

Use this checklist when updating the version of MathOptFormat.

Basic

```
    [] The file at `src/FileFormats/MOF/mof.schema.json` is updated
    [] The constant `_SUPPORTED_VERSIONS` is updated in
`src/FileFormats/MOF/MOF.jl`
```

New sets

```
[] New sets are added to the `@model` in `src/FileFormats/MOF/MOF.jl`
[] New sets are added to the `@enum` in `src/FileFormats/MOF/read.jl`
[] `set_to_moi` is defined for each set in `src/FileFormats/MOF/write.jl`
[] `head_name` is defined for each set in `src/FileFormats/MOF/write.jl`
[] A new unit test calling `_test_model_equality` is aded to `test/FileFormats/MOF/MOF.jl`
```

Tests

```
- [ ] The version field in `test/FileFormats/MOF/nlp.mof.json` is updated
```

Documentation

- [] The version fields are updated in `docs/src/submodules/FileFormats/overview.md`