Recent Advances in Optimization Solvers within JuliaSmoothOptimizers

A tale of solving large-scale optimization problems with JuliaSmoothOptimizers

`jso.dev`

Tangi Migot

`tangi.migot@gmail.com / Github: @tmigot`

joint work with D. Orban (Polytechnique Montréal)
and A.S. Siqueira (Netherlands eScience Center)

JuMP-dev 2024, Montréal, Canada, July 19-21th

Outline

Introduction: Nonlinear nonconvex optimization

Variables: $x \in X$ (take $\mathbb{R}^n$);

Cost: $f : X \to \mathbb{R}$;

Constraints: $C \subseteq X$, for instance described by inequalities (in this case $C = \{x : c_L \leq c(x) \leq c_U,\ \ell \leq x \leq u\}$) with $c : X \to \mathbb{R}^m$.

We denote

$$\min_{x \in X} f(x) \text{ s.t. } x \in C.$$

Numerics?

Tools: Use derivatives (tradeoff efficiency/guarantee);
Aim: Stationary points (local result).

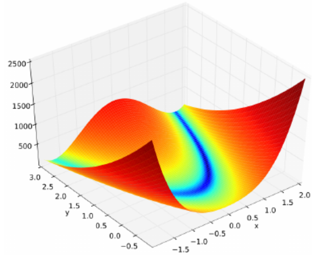


Figure : Rosenbrock's function. source: Wikipedia

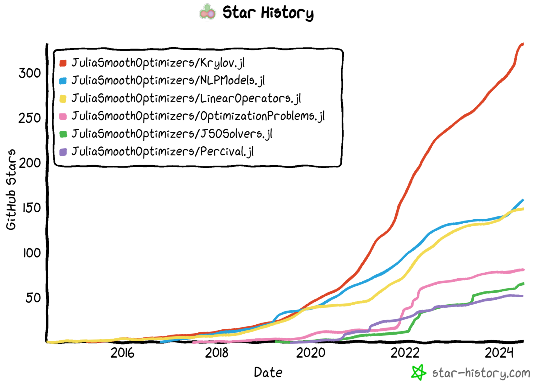
JuliaSmoothOptimizers: Brief genesis

- Scientific background: numerical linear algebra (NLA) and optimization (OPT);
- JuliaSmoothOptimizers: is an organization on GitHub containing a collection of Julia packages;
- The organization was first initiated in 2015 by D. Orban (@dpo) and A. Siqueira (@abelsiqueira). CUTEst, AmplNLReader, Krylov were among the first packages;
- Core contributors are mainly researchers in NLA and OPT;
- JSO is used in the classroom, to write research papers, and solve large problems;
- Checkout @abelsiqueira's talk**s** at the JuliaCon 24 `https://juliacon.org/2024/`
- The organization has a website `https://jso.dev/` with news, references and tutorials.

Thanks for making JSO possible

Some stats...

- Since 2019, **27 journal publications** and a book using JSO packages;
- **60 registered packages** for NLA and OPT



In this talk...

we will only talk about **smooth** nonlinear OPT.

How to maintain JSO: BestieTemplate.jl

- Template focused on best practice
- Can be applied to existing packages (and reapplied or updated)
- Test repo: `github.com/JSOSuite.jl`
- Formatting, citation file, workflows, docs, lint-checker, PR template
- Check @abelsiqueira's talk at JuliaCon

JSO: The environment

Models
- ADNLPModels.jl
- KnetNLPModels.jl
- LLSModels.jl
- ManualNLPModels.jl
- NLPModels.jl
- NLPModelsModifiers.jl
- QuadraticModels.jl
- PartiallySeparableNLP...
- PDENLPModels.jl

Pure Julia solvers
- CaNNOLeS.jl
- DCISolver.jl
- FletcherPenaltySolver.jl
- JSOSolvers.jl
- NCL.jl
- PartiallySeparableSolvers.jl
- Percival.jl
- RegularizedOptimization.jl
- RipQP.jl

Linear algebra
- AMD.jl
- Krylov.jl
- LDLFactorizations.jl
- LimitedLDLFactorizations.jl
- LinearOperators.jl
- SparseMatricesCOO.jl
- SuiteSparseMatrixCollection.jl

Tools
- BenchmarkProfiles.jl
- ExpressionTreeForge.jl
- JSOSuite.jl
- PartitionedStructures.jl
- ShiftedProximalOperators.jl
- Solver/NLPModelsTest.jl
- SolverBenchmark.jl
- SolverCore.jl
- SolverTools.jl

Models wrappers
- AMPLNLReader.jl
- NLPModelsJuMP.jl
- QPSReader.jl

Solver wrappers
- NLPModelsIpopt.jl
- NLPModelsKnitro.jl
- QuadraticModelsCPLEX.jl
- /Gurobi.jl/Xpress.jl

Linear algebra wrappers
- BasicLU.jl
- PROPACK.jl
- HSL.jl
- MUMPS.jl
- QRMumps.jl

Test problems
- CUTEst.jl
- NLSProblems.jl
- RegularizedProblems.jl
- BundleAdjustmentModels.jl
- OptimizationProblems.jl

An example of solver (1/3): ARCqK

Adaptive regularization with cubics (ARC) is an iterative algorithm for

$$\min_{x \in \mathbb{R}^n} \; f(x)$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is smooth. At each iteration, ARC solves a cubic unconstrained problem

$$\min_{d \in \mathbb{R}^n} \; f(x^k) + d^T \nabla f(x^k) + \frac{1}{2} d^T \nabla^2 f(x^k) d + \frac{1}{3\alpha} \|d\|^3$$

(VIP) Fun fact

of iterations to reach $\|\nabla f(x^k)\| \leq \epsilon$ is $\mathcal{O}(\epsilon^{-3/2})$ for ARC, but only $\mathcal{O}(\epsilon^{-2})$ for steepest descent or trust-region.

An example of solver (2/3): The main ingredient

As in trust-region methods, minimizing the cubic model involves solving the shifted linear system

$$(\nabla^2 f(x^k) + \lambda I)d = -\nabla f(x^k) \qquad \text{(LS)}$$

while searching an appropriate value of the shift $\lambda > 0$.

An example of solver (2/3): The main ingredient

As in trust-region methods, minimizing the cubic model involves solving the shifted linear system

$$(\nabla^2 f(x^k) + \lambda I)d = -\nabla f(x^k) \qquad \text{(LS)}$$

while searching an appropriate value of the shift $\lambda > 0$.

Key idea

- exploit Lanczos implementation of the conjugate gradient to solve factorization-free (LS) for several shifts
- only 1 matrix-vector product with the Hessian per iteration of conjugate gradient for several shifts.

J.P. Dussault, **T. Migot**, D. Orban
Scalable adaptive cubic regularization methods, Math. Prog., 2023.

An example of solver (3/3): Implementation

`Krylov.jl` + `NLPModels.jl` + `SolverCore.jl` = `AdaptiveRegularization.jl`

An example of solver (3/3): Implementation

`Krylov.jl` + `NLPModels.jl` + `SolverCore.jl` = `AdaptiveRegularization.jl`

`CUTEst.jl` + `SolverBenchmark.jl` = paper :)

Exploit structure: Partially separable solver

A **partially separable objective function** f sums element functions of smaller dimensions. An example of a separable objective function in $\mathbb{R}^5$:

$$f(x) := f_1(x_1, x_2, x_3) + f_2(x_3, x_4, x_5) + f_3(x_1, x_3, x_5) \tag{1}$$

and its Hessian

$\nabla^2 f =$  .

We want to build specialized methods such as **partitioned quasi-Newton method**. 

NLPModels API

One of the core packages in JSO is `NLPModels.jl`, which provides a standardized API for models of the form

$$\min_{x \in \mathbb{R}^n} \; f(x) \text{ s.t. } c_L \leq c(x) \leq c_U, \; \ell \leq x \leq u,$$

- provides access to objective and constraint functions
- in-place and out-of-place evaluation of the objective gradient, constraints, Jacobian and Hessian nonzero values

Function	API
$f(x)$	obj, objgrad, objcons
$\nabla f(x)$	grad, objgrad
$\nabla^2 f(x)$	hess, hess_op, hess_coord, hess_structure, hprod
$c(x)$	cons, objcons
$\nabla c(x)$	jac, jac_coord, jac_structure, jprod, jtprod, jac_op
$\nabla^2 f(x) + \sum_{i=1}^m y_i \nabla^2 c_i(x)$	hess, hess_coord, hess_structure, hprod, hess_op

There is more...

It has an API to access separately linear and nonlinear constraints.

Access derivatives

Most of the algorithms we will use rely on first and second-order derivatives either to:

- compute a factorization of a system involving Jacobian/Hessian matrices,
- or, compute Jacobian/Hessian-vector products.

The NLPModel API provides two ways to access second-order derivatives:

- Using COO-structure (vectors of rows, columns and values).
- Using linear operators (via `LinearOperators.jl`) to compute the matrix-vector products without storing the whole matrix.

JSO-compliant optimization solvers

There is a minimal set of rules to qualify a JSO-compliant solver:

- The input must be an instance of `AbstractNLPModel` as presented before
- The output has to include a `GenericExecutionStats` implemented in `SolverCore.jl` which gives the solution, optimal value, elapsed time, iterations, primal and dual feasibility, etc.

and that's it!

Note that the whole `NLPModel API` doesn't have to be implemented, only the methods required by the algorithm are needed.

Optimization solvers within JSO

One of the strength of the organization is the variety of available solvers included well-established codes such as:

- Artelys Knitro via `NLPModelsKnitro.jl`;
- Ipopt via `NLPModelsIpopt.jl`;
- Algencan via `NLPModelsAlgencan.jl`;

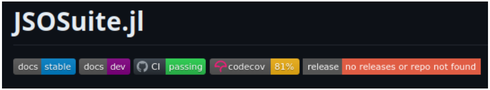
and JSO pure-Julia implementation such as

- `JSOSolvers.jl` and `AdaptiveRegularization.jl` (unconstrained + bounds);
- `RipQP.jl` (quadratic programs);
- `Percival.jl` (bounds + "=");
- `DCISolver.jl` ("=" only for now);
- `FletcherPenaltySolver.jl` ("=" only for now).

Remark

These solvers are indepent of the origin of the problem!

JSOSuite: Motivations



- We needed a tool that gives a simple and intuitive entry point in the JSO universe;
- Be able to easily prototype and compare solvers to find the right option.
- Collect information about a solver and select an appropriate solver (ongoing: automatic detection of linear constraints, NLS, etc.)
- Connect with `SolverBenchmark.jl` for benchmarks and collect results.

Implement the API

- `NLPModels.jl` define the `NLPModel` `API` for the abstract type `AbstractNLPModel`;
- Packages are making subtypes of `AbstractNLPModel` and implementing the API:
  - ADNLPModels: automatic differentiation;
  - ManualNLPModels: manually inputted functions.
- Some packages exploit the problem to provide more efficient implementations:
  - PDENLPModels;
  - PartiallySeparableNLPModels.jl;
  - KnetNLPModels.jl or FluxNLPModels.jl.
- There are also wrappers with optimization modeling languages: `NLPModelsJuMP.jl` for `JuMP.jl` and `MOI`, `CUTEst.jl` or `AmplNLReader.jl`.

ADNLPModels

The package `ADNLPModels.jl` provides AD-based model implementations that conform to the NLPModels API.

- It has no specific modeling constraints, and accepts directly Julia's `Function` type;
- This allow to define models for any floating-point type that supports arithmetic operations;
- It uses automatic differentiation modules such as `ForwardDiff.jl` or `ReverseDiff.jl` to compute derivatives;
- It is design with a backend organization that allow switching from one AD-module to another and build mixed-models;

Try out ADNLPModels 0.8.2 and 0.8.3

thanks to @amontoison, @gdalle and @adrhill there was massive improvement in Jacobian/Hessian computations.

Mixed NLPModels with ADNLPModels.jl

PDENLPModels.jl usually defined sparse Jacobian and Hessian matrices, but no operator-type product.

```
using ADNLPModels, PDENLPModels, PDEOptimizationProblems
model = steering() # Insantiate a NLPModel using PDENLPModels API
x0 = model.meta.x0
f(x; nlp = model) = obj(nlp, x)
c!(cx, x; nlp = model) = cons!(nlp, x, cx)
lcon, ucon = model.meta.lcon, model.meta.ucon
lvar, uvar = model.meta.lvar, model.meta.uvar
name = "AD-$(model.meta.name)"
nlp = ADNLPModel!(
    f, x0, lvar, uvar,
    c!, lcon, ucon,
    name = name,
    jacobian_backend = model,
    hessian_backend = model,
)
```

NLPModels: Things I won't talk about

`NLPModels.jl` define the `NLPModel API` for problems such as

$$\min_{x \in \mathbb{R}^n} f(x) \text{ s.t. } c_L \leq c(x) \leq c_U, \ \ell \leq x \leq u, \tag{2}$$

- It is possible to specialize the API for a subproblem of (1), for instance consider the nonlinear least squares (NLS):

$$\min_{x \in \mathbb{R}^n} \frac{1}{2}\|F(x)\|_2^2 \text{ s.t. } c_L \leq c(x) \leq c_U, \ \ell \leq x \leq u. \tag{3}$$

  `NLPModels.jl` also defines an API that access to the residual F and its derivatives.
- It is possible to modify NLPModel, for instance use a quasi Newton approximation, add slack variables, etc. The main package for these transformations is `NLPModelsModifiers.jl`.

An example of NLPModel modifier: TimerNLPModels.jl

```
using JSOSolvers, NLPModelsTest, TimerNLPModels
nlp = BROWNDEN()
timer_nlp = TimerNLPModel(nlp)
trunk(timer_nlp)
get_timer(timer_nlp) # screenshot of the result of this last line.
```

		Time			Allocations		
Tot / % measured:		842μs / 17.3%			19.2KiB / 8.2%		
Section	ncalls	time	%tot	avg	alloc	%tot	avg
grad!	10	66.2μs	45.5%	6.62μs	320B	20.0%	32.0B
obj	10	43.1μs	29.6%	4.31μs	320B	20.0%	32.0B
hess_op!	1	36.2μs	24.9%	36.2μs	960B	60.0%	960B

JSO GPU support

We recently started testing and supporting GPU data types across the organization

- `NLPModels.jl` with `NLPModelsTest.jl` now have tests for GPU compatible problems.
- The other NLPModel packages that are compatible are also tested.
- Solvers are being tested too: `TRUNK`, first-order solvers `R2`
- Soon: Equality-constrained solvers `FletcherPenaltySolver.jl` and `Percival.jl`
- To come later: `LBFGS`, bound-constrained solvers, and `Percival.jl` with inequalities.

Parameter optimization

What is a JSO-compliant solver?

- Minimal JSO-compliant: NLPModel as input, GenericExecutionStats as output;
- Efficient JSO-compliant: implement `solve!(nlp::AbstractNLPModel, solver::AbstractOptimizationSolver, stats::GenericExecutionStats)`

New change

We are adding a new structure for handling solver parameters `AbstractParameterSet`.

Parameter optimization

The strategy is as follows

- Use `AbstractParameterSet` in your solver, specify bounds, constraints, defaults for each parameter;
- Instantiate a `BBModel`, an unconstrained model that corresponds to the parameter optimization problem;
- Solve this problem (derivative-free solver, random search, etc.);
- Feed the solver with your optimized parameters.
- (Automatize this process in the repos)

Parameter optimization

```julia
using JSOSolvers, BBModels, SolverParameters
model = BBModel(
  LBFGSParameterSet{Float64}(), # AbstractParameterSet
  problems, # vector of AbstractNLPModel
  (nlp, p) -> lbfgs(nlp, mem = value(p.mem)),
  time_only, # time_only, memory_only, sumfc OR a hand-made function
  subset = (:mem,), # optimize only `mem`
)
vals = BBModels.random_search(model, verbose = 0)
```

Thank you for your attention!

`https://github.com/JuliaSmoothOptimizers/JSOSuite.jl`

Where to start with JuliaSmoothOptimizers?

- JSO website `https://jso.dev` with news, tutorials, etc.
- New contributors are always welcome! Feel free to say Hi! or discuss ideas, potential use-cases, etc.