DiffOpt.jl

Differentiable Optimization for JuMP

A tutorial

Joaquim Dias Garcia · Soma Energy / jump-dev · Edinburgh · 2026

What is differentiable optimization?

An optimization problem with parameters θ defines an implicit function:

$$x^\star(\theta) \;=\; \arg\min_x \; f(x;\theta) \quad \text{s.t.} \quad g(x;\theta) \le 0, \;\; h(x;\theta) = 0$$

Differentiable optimization = computing $\partial x^\star / \partial \theta$ — analytically.

Typically, through the KKT conditions.

Input: problem data θ
(objective coeffs, constraint coeffs, RHS, parameters)

→

`solve()` + KKT diff

→

Output: solution $x^\star(\theta)$
and its Jacobian w.r.t. θ

Implicit differentiation in one slide

The KKT conditions (or the HSDE) are a system of equations:

$$F(x^\star,\, \lambda^\star,\, \nu^\star;\, \theta) \;=\; 0$$

By the **implicit function theorem**, near a solution:

$$\frac{\partial x^\star}{\partial \theta} \;=\; -\left[\frac{\partial F}{\partial (x,\lambda,\nu)}\right]^{-1} \frac{\partial F}{\partial \theta}$$

> **What DiffOpt does for you:** it assembles that Jacobian using MOI to read your current primal/dual solution, factorises it once, and lets you push tangents through in either direction (forward or reverse) for the cost of one back-solve.

Why do we care?

Decision-Focused / Application-driven Learning

Application-driven learning: train a predictor so that the *downstream optimization decision* is good — not the prediction itself. Gradients flow through the optimal solution.

arXiv:2102.13273

Sobolev Training of Optimization Proxies

Distil a slow optimizer into a fast neural surrogate by matching not just the optimal solution but its *gradients* w.r.t. the inputs.

arXiv:2505.11342

Bilevel & strategic decisions

Energy-market bidding, model predictive control, robust design: the outer level needs gradients of the inner optimum.

SSRN:5043309

Sensitivity analysis

"How does my dispatch react to a 1% demand shock?", "What's $\partial\lambda/\partial d$ at the marginal generator?", $\partial w/\partial\sigma_{\max}$ in mean–variance.

What is DiffOpt.jl?

A Julia package for **differentiating the solution map** of an optimization problem (from JuMP).

- Problem classes: **LP, QP, Conic, NLP**
- Modes: **forward** and **reverse**
- Built **on top of JuMP and MOI** (modelling stays the same)

```
using JuMP, DiffOpt, HiGHS
model = DiffOpt.quadratic_diff_model(HiGHS.Optimizer)
@variable(model, x)
@variable(model, p in Parameter(4.0))
@constraint(model, x ⩾ 3p)
@objective(model, Min, 2x)
optimize!(model)
DiffOpt.set_forward_parameter(model, p, 1.0)
DiffOpt.forward_differentiate!(model)
DiffOpt.get_forward_variable(model, x)   # = 3.0
```

Two roles of DiffOpt

A — As an interface

A **unified API** for differentiable optimization, designed to extend JuMP and MOI to this domain.

The attribute system (`ForwardObjectiveFunction`, `ReverseVariablePrimal`, ...) was built to mirror MOI's own attribute design.

One day, these could be **upstreamed into MOI**.

B — As a helper

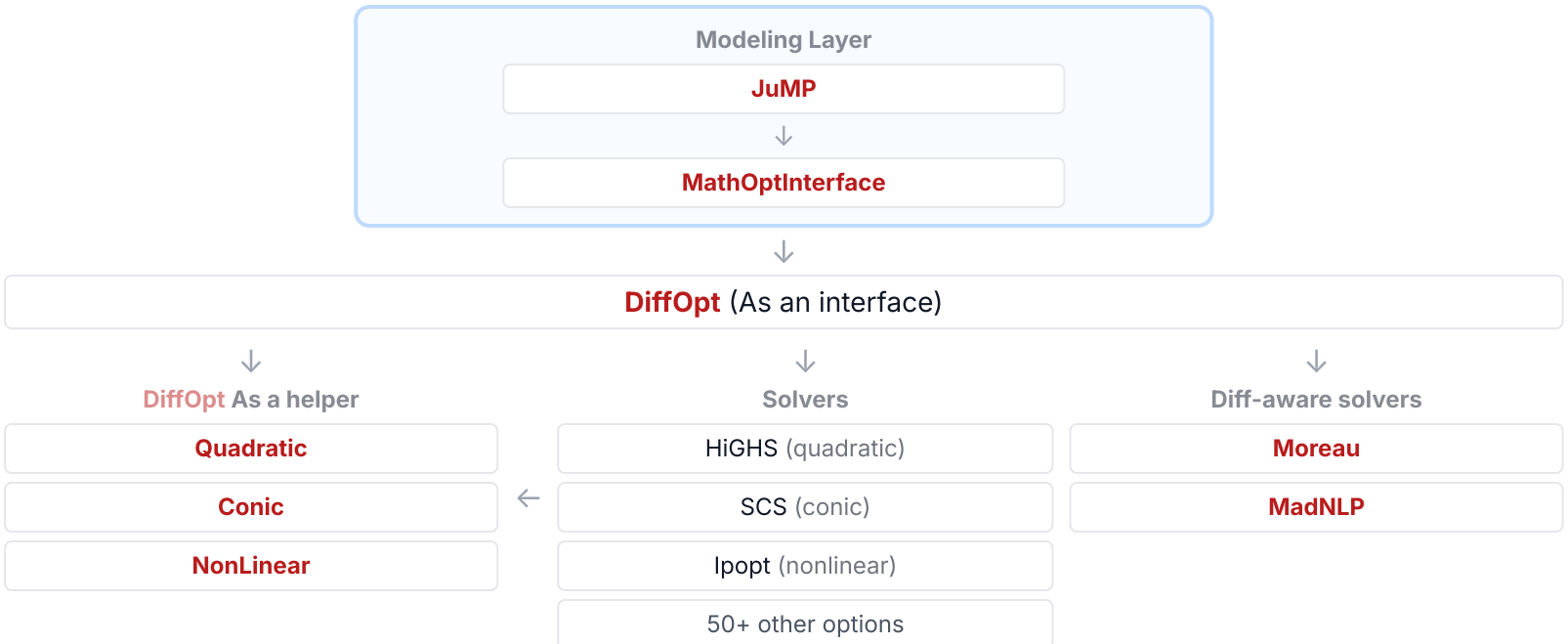
Most solvers **do not** ship with differentiation: HiGHS, Clp, ECOS, SCS, Ipopt, Xpress, ...

DiffOpt provides a **pluggable diff layer** over any MOI-compatible solver:

```
DiffOpt.diff_model(HiGHS.Optimizer)
DiffOpt.diff_model(SCS.Optimizer)
DiffOpt.diff_model(Ipopt.Optimizer)
DiffOpt.diff_model(Xpress.Optimizer)
```

Same API, different solver. The diff math lives in DiffOpt — the solver just solves.

Architecture



Forward vs reverse mode

Forward mode

Question: *Given a perturbation to data, how does the solution move?*

- **Input:** sensitivity on data (`Δθ`)
- **Output:** sensitivity on solution (`Δx*`)
- **Trigger:** `forward_differentiate!`
- **Best for:** few directions in `θ`, many outputs

```
Δθ  ⟶  [ DiffOpt ]  ⟶  Δx*
```

$$\mathrm{jvp}_f(x, v) = J_f(x) \cdot v$$

Reverse mode

Question: *Given a "wish" for the solution, which data points should I credit?*

- **Input:** sensitivity on solution (`x̄`)
- **Output:** sensitivity on data (`θ̄`)
- **Trigger:** `reverse_differentiate!`
- **Best for:** scalar loss, many parameters (ML)

```
x̄  ⟶  [ DiffOpt ]  ⟶  θ̄
```

$$\mathrm{vjp}_f(x, v) = v \cdot J_f(x)$$

Three problem-class backends

QuadraticProgram

src/QuadraticProgram/

Supports **LP, QP**

Differentiates the QP KKT system directly.

`quadratic_diff_model`

ConicProgram

src/ConicProgram/

Supports **LP, Conic**

Differentiates via the homogeneous self-dual embedding.

`conic_diff_model`

NonLinearProgram 🆕

src/NonLinearProgram/

Supports **LP, QP, NLP**

KKT differentiation for smooth, potentially nonconvex problems.

`nonlinear_diff_model`

And the auto-dispatching factory: `DiffOpt.diff_model(SolverOptimizer)` picks one for you.

Interchangeable backends

Problem class	`quadratic_diff_model`	`conic_diff_model`	`nonlinear_diff_model`
LP	✓	✓	✓
QP	✓	✓*	✓
Conic		✓	
NLP			✓

> Backends are only constrained by the problem classes. Not by the solver.

Why does it matter? Numerical stability and feature coverage differ. Some LP may differentiate more cleanly through the conic backend; a QP with active bounds may work better via NLP backend. You can swap one line and re-check.

The two interfaces

Functional API (low-level)

Set tangents **directly on problem data** via specialized helpers:

- `DiffOpt.set_forward_objective_function`
- `DiffOpt.set_forward_constraint_function`
- `DiffOpt.get_reverse_objective_function`
- `DiffOpt.get_reverse_constraint_function`

Talks to MOI in MOI's own language: scalar/vector affine functions, constraint indices, sign conventions.

Parametric API (high-level)

Declare parameters in JuMP, ask for sensitivities by name:

- `@variable(m, p in Parameter(...))`
- `DiffOpt.set_forward_parameter` / `DiffOpt.get_forward_variable`
- `DiffOpt.set_reverse_variable` / `DiffOpt.get_reverse_parameter`

Sensitivities **propagate automatically** to every place `p` appears in the model.

Pros / cons of each interface

Functional API

Pros:

- Most general: covers anything MOI can express
- Required by bridges to rewrite parametric problems
- Full control of which coefficients receive which tangent

Cons:

- Verbose: explicit constraint indices and functions
- Users must respect MOI sign conventions
- You bookkeep your own parameter↔coefficient map

Parametric API

Pros:

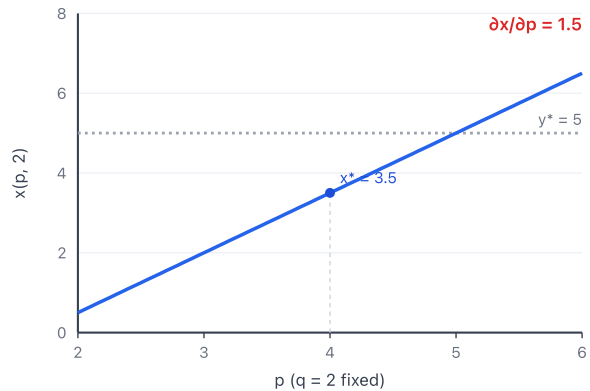
- Ergonomic: JuMP-native, declare once
- Auto-propagates a single parameter across the whole model
- Cleaner for ML / sensitivity workflows

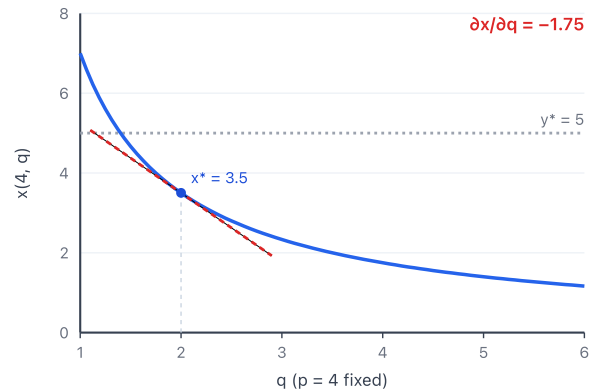
Cons:

- Built *on top of* the functional API / can't replace it
- Can be slower for very large models

One example, solved by hand first

$$\min_{x,\,y} \; 2x - y \quad \text{s.t.} \quad q\,x + y \;\geq\; 3p, \quad y \leq 5 \qquad \text{at} \;\; p = 4,\; q = 2$$





T he solution map is

$$x(p,q) = \frac{3p - 5}{q} = 3.5, \qquad y = 5$$

	$\partial/\partial p$	$\partial/\partial q$
x	1.5	-1.75
y	0	0

Sensitivity by finite differences

Before any auto-diff, the obvious thing: build the model **once**, update parameters, re-solve, divide.

```
using JuMP, DiffOpt, HiGHS

model = DiffOpt.quadratic_diff_model(HiGHS.Optimizer); set_silent(model)
@variable(model, x); @variable(model, y)
@variable(model, p in Parameter(4.0))
@variable(model, q in Parameter(2.0))
@constraint(model, q * x + y ⩾ 3 * p)
@constraint(model, y ⩽ 5)
@objective(model, Min, 2x - y)

function x_at(pv, qv)
    set_parameter_value(p, pv)
    set_parameter_value(q, qv)
    optimize!(model)
    return value(x)
end

h = 1e-4
(x_at(4 + h, 2) - x_at(4 - h, 2)) / (2h)    # ≈  1.5    (∂x/∂p)
(x_at(4, 2 + h) - x_at(4, 2 - h)) / (2h)    # ≈ -1.75   (∂x/∂q)
```

Parametric API: declare the parameters

Express the shared example with JuMP `Parameter` variables. `p` and `q` are first-class objects you can attach sensitivities to.

```
@variable(model, x)
@variable(model, y)
@variable(model, p in Parameter(4.0))   # ← first-class JuMP objects
@variable(model, q in Parameter(2.0))
@constraint(model, cons, q * x + y ≥ 3 * p)
@constraint(model, ybnd, y ≤ 5)
@objective(model, Min, 2x - y)
```

Parametric API: forward mode

```julia
using JuMP, DiffOpt, HiGHS
model = DiffOpt.quadratic_diff_model(HiGHS.Optimizer); set_silent(model)
@variable(model, x); @variable(model, y)
@variable(model, p in Parameter(4.0)); @variable(model, q in Parameter(2.0))
@constraint(model, cons, q * x + y ≥ 3 * p)
@constraint(model, ybnd, y ≤ 5)
@objective(model, Min, 2x - y)
optimize!(model)                                # x = 3.5, y = 5

DiffOpt.empty_input_sensitivities!(model)
DiffOpt.set_forward_parameter(model, p, 1.0)    # nudge p
DiffOpt.forward_differentiate!(model)
DiffOpt.get_forward_variable(model, x)          # →  1.5   = ∂x/∂p
DiffOpt.get_forward_variable(model, y)          # →  0.0   = ∂y/∂p
DiffOpt.get_forward_objective(model)            # →  3.0   = ∂obj/∂p

DiffOpt.empty_input_sensitivities!(model)
DiffOpt.set_forward_parameter(model, q, 1.0)    # nudge q
DiffOpt.forward_differentiate!(model)
DiffOpt.get_forward_variable(model, x)          # → -1.75  = ∂x/∂q
DiffOpt.get_forward_variable(model, y)          # →  0.0   = ∂y/∂q
DiffOpt.get_forward_objective(model)            # → -3.5   = ∂obj/∂q
```

Parametric API: reverse mode

```julia
using JuMP, DiffOpt, HiGHS
model = DiffOpt.quadratic_diff_model(HiGHS.Optimizer); set_silent(model)
@variable(model, x); @variable(model, y)
@variable(model, p in Parameter(4.0)); @variable(model, q in Parameter(2.0))
@constraint(model, cons, q * x + y ≥ 3 * p)
@constraint(model, ybnd, y ≤ 5)
@objective(model, Min, 2x - y)
optimize!(model)                                  # x = 3.5, y = 5

DiffOpt.set_reverse_variable(model, x, 1.0)       # gradients of x*
DiffOpt.reverse_differentiate!(model)
DiffOpt.get_reverse_parameter(model, p)           # →  1.5    = ∂x/∂p
DiffOpt.get_reverse_parameter(model, q)           # → -1.75   = ∂x/∂q

DiffOpt.empty_input_sensitivities!(model)
DiffOpt.set_reverse_objective(model, 1.0)         # gradients of obj*
DiffOpt.reverse_differentiate!(model)
DiffOpt.get_reverse_parameter(model, p)           # →  3.0    = ∂obj/∂p
DiffOpt.get_reverse_parameter(model, q)           # → -3.5    = ∂obj/∂q
```

Parametric API: under the hood

- `p`, `q` are JuMP variables bound by a `MOI.Parameter` set.
- POI (ParametricOptInterface) rewrites the model so the solver sees a *non-parametric* problem with the current values substituted in.
- DiffOpt tracks the parameter↔coefficient map POI used. Set a sensitivity on `p` or `q` and it lowers to the right `ForwardConstraintFunction` / `ForwardObjectiveFunction` calls automatically.
- The sign-convention bookkeeping is handled for you.

> You never track "which constraint did `q` appear in, and with what coefficient?". Declare once, differentiate everywhere.

Forward mode:
One linear solve *per parameter direction*.

Reverse mode:
One linear solve total, gradients for *all* parameters.

Functional API: same problem, no parameters

The functional API has **no parameters**. We differentiate the *raw data* of the model the solver actually sees.

The snapshot of the shared problem at `p = 4, q = 2`.

```
using JuMP, DiffOpt, HiGHS

model = DiffOpt.diff_model(HiGHS.Optimizer); set_silent(model)
@variable(model, x)
@variable(model, y)
@constraint(model, cons, 2x + y ⩾ 12)   # q·x + y ≥ 3p   at  q=2, p=4
@constraint(model, ybnd, y ⩽ 5)
@objective(model, Min, 2x - y)
optimize!(model)                         # ⇒ x = 3.5, y = 5
```

Note the substituted data: the `x`-coefficient **2** in `cons` is `q`; the RHS **12** is `3p`. Those are the knobs we will perturb.

Functional API: coefficients are the sensitivities

JuMP/MOI writes each constraint as `function in set`; the RHS lives in the set.
`ForwardConstraintFunction` is the **tangent of** `(function - RHS)` **w.r.t. your perturbation** θ.

Direction `p` — `p` lives only in `cons` (as the RHS `3p`)

Constraint: `q·x + y ≥ 3p`

Normalized: `q·x + y - 3p ≥ 0`

Sensitivity: `q·x + y - 3(p + Δp) ≥ 0`

$\partial/\partial p = -3$

```
DiffOpt.set_forward_constraint_function(
    model, cons, AffExpr(-3.0))
```

Direction `q` — `q` lives only in `cons` (as the `x`-coefficient)

Constraint: `q·x + y ≥ 3p`

Normalized: `q·x + y - 3p ≥ 0`

Sensitivity: `(q + Δq)·x + y - 3p ≥ 0`

$\partial/\partial q = x$

```
DiffOpt.set_forward_constraint_function(
    model, cons, 1.0x)
```

Nothing stops you from doing **both at once**: forward mode is linear, so a combined seed represents the sum of the two directions.

```
DiffOpt.set_forward_constraint_function(model, cons, 1.0x - 3.0)   # (Δq - 3Δp)
```

Functional API: forward

```julia
using JuMP, DiffOpt, HiGHS
model = DiffOpt.diff_model(HiGHS.Optimizer); set_silent(model)
@variable(model, x); @variable(model, y)
@constraint(model, cons, 2x + y ⩾ 12)
@constraint(model, ybnd, y ⩽ 5)
@objective(model, Min, 2x - y)
optimize!(model)                                  # x = 3.5, y = 5

DiffOpt.set_forward_constraint_function(model, cons, AffExpr(-3.0))   # ∂/∂p
DiffOpt.forward_differentiate!(model)
DiffOpt.get_forward_variable(model, x)            # →  1.5   = ∂x/∂p
DiffOpt.get_forward_variable(model, y)            # →  0.0   = ∂y/∂p

DiffOpt.empty_input_sensitivities!(model)
DiffOpt.set_forward_constraint_function(model, cons, 1.0x)            # ∂/∂q
DiffOpt.forward_differentiate!(model)
DiffOpt.get_forward_variable(model, x)            # → -1.75  = ∂x/∂q
DiffOpt.get_forward_variable(model, y)            # →  0.0   = ∂y/∂q
```

> Same as `set_forward_parameter(model, p, 1.0)`, and `set_forward_parameter(model, q, 1.0)`.

Forward is linear: combine perturbations

Seed both parameter directions **at once**. The seed is the sum of the individual partials; the response is the sum of the individual responses.

```julia
using JuMP, DiffOpt, HiGHS
model = DiffOpt.diff_model(HiGHS.Optimizer); set_silent(model)
@variable(model, x); @variable(model, y)
@constraint(model, cons, 2x + y ⩾ 12)
@constraint(model, ybnd, y ⩽ 5)
@objective(model, Min, 2x - y)
optimize!(model)                                  # x = 3.5, y = 5

DiffOpt.set_forward_constraint_function(model, cons, 1.0x - 3.0)   # ∂/∂q + ∂/∂p
DiffOpt.forward_differentiate!(model)
DiffOpt.get_forward_variable(model, x)            # → -0.25  = ∂x/∂p + ∂x/∂q
DiffOpt.get_forward_variable(model, y)            # →  0.0
```

> Forward mode is a *directional derivative*: one back-solve, one direction. The seed `1.0x - 3.0` represents `(Δp, Δq) = (1, 1)`, so `dx = 1.5 + (-1.75) = -0.25`. Any linear combination of parameter directions works the same way.

Functional API: reverse

```julia
using JuMP, DiffOpt, HiGHS
model = DiffOpt.diff_model(HiGHS.Optimizer); set_silent(model)
@variable(model, x); @variable(model, y)
@constraint(model, cons, 2x + y ⩾ 12)
@constraint(model, ybnd, y ⩽ 5)
@objective(model, Min, 2x - y)
optimize!(model)                                  # x = 3.5, y = 5

DiffOpt.set_reverse_variable(model, x, 1.0)       # credit for moving x
DiffOpt.reverse_differentiate!(model)

grad_obj  = DiffOpt.get_reverse_objective_function(model)
# grad_obj  == 0

grad_cons = DiffOpt.get_reverse_constraint_function(model, cons)
# grad_cons == -1.75x - 2.5y - 0.5
```

> The parametric API's `get_reverse_parameter(model, q)` = **−1.75** is just the `x`-coefficient of `grad_cons`.
> `get_reverse_parameter(model, p)` = **1.5** comes from the constant **−0.5** times `∂(-3p)/∂p = -3`.

References

Mathieu Besançon, Joaquim Dias Garcia, Benoît Legat, Akshay Sharma. *Flexible Differentiable Optimization via Model Transformations*. INFORMS J. Comput. 36(2), 456–478, 2024. DOI: 10.1287/ijoc.2022.0283 · arXiv: 2206.06135.

Andrew W. Rosemberg, Joaquim Dias Garcia, François Pacaud, Robert B. Parker, Benoît Legat, Kaarthik Sundar, Russell Bent, Pascal Van Hentenryck. *A General and Streamlined Differentiable Optimization Framework*. arXiv: 2510.25986, Oct 2025.

GitHub: https://github.com/jump-dev/DiffOpt.jl

Docs: https://jump.dev/DiffOpt.jl/stable

Try it out

```
using Pkg; Pkg.add("DiffOpt")

using JuMP, DiffOpt, SOLVER
model = DiffOpt.diff_model(SOLVER.Optimizer)
# ... your JuMP code, with @variable(model, p in Parameter(...))
```

Examples

`docs/src/examples/`

Ridge, SVM, portfolio, dispatch, inverse kinematics, ...

Issues

github.com/jump-dev/DiffOpt.jl

Questions?

Catch me after the talk, or on the JuMP Discourse.

Thank you.