Large Scale Optimization Solvers in Julia

A tale of solving large-scale optimization problems with JuliaSmoothOptimizers

Tangi Migot
Polytechnique Montréal
tangi.migot@gmail.com

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

Journées de l'Optimisation 2022, Montréal, May 17th

Outline

Introduction

Introduction: nonlinear optimization

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

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

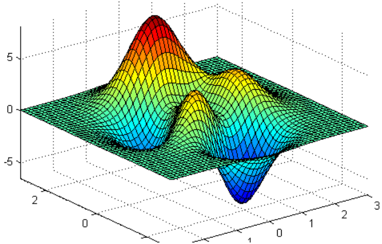
Constraints: $C \subseteq X$, for instance described by inequalities (in this case $C = \{x : g(x) \leq 0\}$) with $g : X \to \mathbb{R}^m$.

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).



Example: 2D Poisson-Boltzmann problem

Example

A typical example is the control problem of a 2D Poisson-Boltzman equation:

$$\begin{cases} \min\limits_{y\in H_0^1(\Omega),u\in L^2(\Omega)} \dfrac{1}{2}\displaystyle\int_\Omega |y - y_d(x)|^2 + \dfrac{1}{2}\alpha \int_\Omega |u|^2 dx, \\ \qquad\qquad \text{s.t.} \quad -\Delta y + \sinh y = h + u, \quad \text{in } \Omega := (-1,1)^2, \\ \qquad\qquad\qquad y = 0, \quad \text{in } \partial\Omega, \end{cases}$$

with the forcing term $h(x_1, x_2) = -\sin(\omega x_1)\sin(\omega x_2)$, $\omega = \pi - \frac{1}{8}$, and target state

$$y_d(x) = \begin{cases} 10 & \text{if } x \in [0.25, 0.75]^2, \\ 5 & \text{otherwise.} \end{cases}$$

Optimization over Partial Differential Equations

Hyperparameters in the model (u, c) can be refined by known data/measurements $\hat{y}$

PDE-constrained optimization problem:

$$\min_{y,u,c} \quad \text{cost}\left(y, \frac{\partial y}{\partial x}, u, c; \hat{y}\right)$$

$$\text{subject to } f\left(y, \frac{\partial y}{\partial x_i}, \frac{\partial^2 y}{\partial x_i \partial x_j}, u, c\right) = 0, \quad \text{over } x \in \Omega$$

unknown function

1^{st} and 2^{nd} partial derivatives of y

control and hyperparameter

1D (2D or 3D) domain

Challenge: Design codes for PDE-constrained optimization

PDE-constrained optimization: a Toolbox

The environment

JuliaSmoothOptimizers :

- a Github organization initiated in 2017 by D.Orban and A.Siqueira at Polytechnique Montréal
- Julia packages for linear algebra and continuous smooth optimization solvers

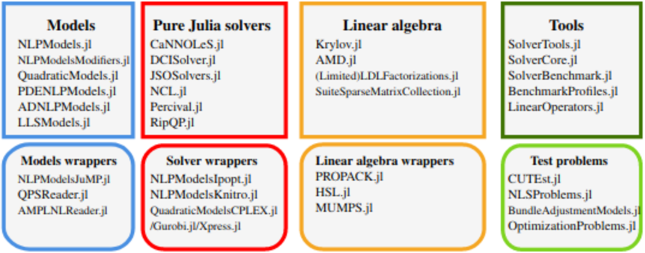


Fig. 2: Organization of the JSO packages in clusters.

Modeling

Model PDE-constrained optimization in Julia

$$\begin{cases} \min\limits_{y \in H_0^1(\Omega), u \in L^2(\Omega)} \frac{1}{2} \int_\Omega |y - y_d(x)|^2 + \frac{1}{2}\alpha \int_\Omega |u|^2 dx, \\ \qquad \text{s.t.} \quad -\Delta y + \sinh y = h + u, \quad \text{in } \Omega := (-1,1)^2, \\ \qquad\qquad y = 0, \quad \text{in } \partial\Omega, \end{cases}$$

We target a direct method that discretize the problem (domain/integral/partial derivatives) and convert it as a very large (but sparse and highly structured) nonlinear continuous optimization problem.

Challenge

Access a discretization of the domain and have the possibility to evaluate derivatives of the involved functions.

Finite-element methods

For PDEs, there are several ways to represent functions and derivatives as vectors:

- Finite difference methods: functions are represented on a grid, e.g., `DiffEqOperators.jl`, `InfiniteOpt.jl` or `Trixi.jl`.
- Finite volume methods: functions are represented by a discretization of its integral.
- Spectral methods: functions are represented by a global basis, e.g., `FFTW.jl` and `ApproxFun.jl`.
- Physics-informed neural networks: functions are represented by a neural networks, e.g., `NeuralPDE.jl`.
- **Finite element methods**: functions are represented by a local basis.

Finite-element methods

FE methods for discretization is a must for generic formulations.

- It is easy to increase the order of the elements or locally refine the mesh so that the physics fields can be approximated accurately.
- You can straightforwardly combine different kinds of approximation functions leading to mixed formulations.
- Finally, curved or irregular geometries of the domain are handled in a natural way.

The theory is much more difficult which explains the scarcity of implementations.
There exists a couple of packages for FE methods in Julia. The main are `FEniCS.jl`, `Ferrite.jl`, `FinEtools.jl`, `JuliaFEM.jl`, and `Gridap.jl`.

Gridap.jl for the FE discretization

We focus on the Gridap.jl as

- exclusively written in the Julia programming language
- supports a variety of different models, discretizations, and meshing possibilities
- has a very expressive API allowing to model complex PDEs with very few lines of code
- the user can write the underlying **weak form** with a syntax almost one-to-one to the mathematical notation.

```julia
using Gridap
#Domain: domain/partition
model = CartesianDiscreteModel((-1,1,-1,1), (n,n))
#Definition of the spaces:
order = 2
reffe = ReferenceFE(lagrangian, Float64, order)
Xpde = TestFESpace(model, reffe; conformity = :H1, dirichlet_tags = "boundary")
Ypde = TrialFESpace(Xpde, 0.0)
reffe_con = ReferenceFE(lagrangian, valuetype, 1)
Xcon = TestFESpace(model, reffe_con; conformity = :H1)
Ycon = TrialFESpace(Xcon)
#Integration machinery: triangulation / degree
dΩ = Measure(Triangulation(model), 1)
#Definition of constraint operator
h(x) = -sin((π - 1 / 8) * x[1]) * sin((π - 1 / 8) * x[2])
res(y, u, v) = ∫(∇(v) ⋅ ∇(y) + (sinh ∘ y) * v - u * v - v * h) * dΩ
```

PDENLPModels.jl

`PDENLPModels.jl` is a new Julia implementation for the modelization of optimization problems with a discretized partial differential equation (PDE) on Ω in the constraints of the form

$$\begin{aligned} \min_{y\in\mathcal{Y},u\in\mathcal{U},\theta\in\mathbb{R}^k} \quad & \int_\Omega J(y,u,\theta)d\Omega \\ \text{s. t.} \quad & c(y,u,\theta)=0, \quad \text{(the governing PDE on } \Omega) \\ & l_{y,u} \leq (y,u) \leq u_{y,u}, \quad \text{(functional bound constraints)} \\ & l_\theta \leq \theta \leq u_\theta, \quad \text{(bound constraints)} \end{aligned}$$

$J : \mathcal{Y} \times \mathcal{U} \times \mathbb{R}^k \to \mathbb{R}$ and $c : \mathcal{Y} \times \mathcal{U} \times \mathbb{R}^k \to \mathcal{C}$ are smooth mappings $(\mathcal{Y}, |\cdot|_\mathcal{Y})$, $(\mathcal{U}, |\cdot|_\mathcal{U})$, and $(\mathcal{C}, |\cdot|_\mathcal{C})$ are real Banach spaces. $l_\theta, u_\theta \in \mathbb{R}^k$ are bounds on the unknown θ, and $l_{y,u}, u_{y,u}$ are functional bounds $\Omega \to \mathcal{Y} \times \mathcal{U}$ on the unknown controls and states.

PDENLPModels.jl implements the NLPModel API

The package's main function exports GridapPDENLPModel that uses `Gridap.jl` for the discretization of the functional space by finite elements.

```
#Objective function:
yd(x) = min(x[1] - 0.25, 0.75 - x[1], x[2] - 0.25, 0.75 - x[2]) >= 0.0 ? 10.0 : 5.0
α = 1e-4
function f(y, u)
∫(0.5 * (yd - y) * (yd - y) + 0.5 * α * u * u) * dΩ
end
nlp = GridapPDENLPModel(xin, f, trian, Ypde, Ycon, Xpde, Xcon, op, name = "2D-Poisson Boltzman n=$n")
```

Model

The resulting model is an instance of an AbstractNLPModel, defined in `NLPModels.jl`.

NLPModels API

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

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

- provides access to objective and constraint functions
- in-place and out-of-place evaluation of the objective gradient, constraints, Jacobian and Hessian nonzero values
- a corresponding API dedicated to nonlinear least-squares models

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

Solvers

Solvers within JSO

Therefore, the package `PDENLPModels.jl` offers an interface between generic PDE-constrained optimization problems and cutting-edge optimization solvers such as:

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

and JSO pure-Julia implementation such as

- `Percival.jl` (bounds + "=")
- `DCISolver.jl` ("=" only)
- `FletcherPenaltyNLPSolver` (bounds + "=")

and basically any solver accepting an AbstractNLPModel as input, see JuliaSmoothOptimizers (JSO).

Remark

These solvers are indepent of the origin of the problem!

Access derivatives

Most of these solvers/algorithms 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.

Subproblem solvers

Most of these solvers/algorithms are iteratively solving subproblems of the form of simpler optimization problems (bound-constrained or unconstrained) or/and linear algebra systems (linear system, linear least squares, linear least-norm, ...).

- `JSOSolvers.jl` provides implementation of classical unconstrained/bound-constrained methods: lbfgs, tron, trunk (and their NLS versions);
- `LDLFactorizations.jl` and `HSL.jl` provide LDL factorization of sparse matrices.
- `Krylov.jl` contains over 30 implementation of iterative methods for various linear algebra systems (with GPU support).

DCISolver.jl

Each DCI iteration is a two-step process.

- Tangential step: approximately minimizes a quadratic model subject to linearized constraints within a trust region.
- Normal step: recenters feasibility by way of a trust cylinder, which is the set of points such that $\|h(x)\| \leq \rho$, where $\rho > 0$.

Each time the trust cylinder is violated during the tangential step, the normal step brings infeasibility back within prescribed limits. The radius ρ of the trust cylinder decreases with the iterations, so a feasible and optimal point results in the limit.

Bielschowsky, R. H., & Gomes, F. A..
Dynamic control of infeasibility in equality constrained optimization,
SIAM Journal on Optimization, 19:3, pp. 1299-1325, 2008.

FletcherPenaltyNLPSolver

The method uses Fletcher's penalty function:

$$\min_{x \in \mathbb{R}^n} f(x) - c(x)^T y_\sigma(x)$$

where

$$y_\sigma(x) \in \arg\min_y \frac{1}{2} \|\nabla c(x)^T y - \nabla f(x)\|_2^2 + \sigma c(x)^T y$$

Fun facts (1/2):

- This function is also smooth under classical assumptions
- The penalty function is exact, i.e. local minimizers are minimizers of the penalty function for σ sufficiently large.

Estrin, R., Friedlander, M. P., Orban, D., & Saunders, M. A. . Implementing a smooth exact penalty function for equality-constrained nonlinear optimization, *SIAM Journal on Scientific Computing*, 42:3, pp. A1809-A1835, 2020.

FletcherPenaltyNLPSolver

The method uses Fletcher's penalty function:

$$\min_{x \in \mathbb{R}^n} f(x) - c(x)^T y_\sigma(x)$$

where

$$y_\sigma(x) \in \arg\min_y \frac{1}{2} \|\nabla c(x)^T y - \nabla f(x)\|_2^2 + \sigma c(x)^T y$$

Fun facts (2/2):

- Evaluating the penalty function and its derivatives is the solution of a certain saddle-point system.
- If the system matrix is available explicitly, we can factorize it once and reuse the factors to evaluate the derivatives.
- The penalty function can also be adapted to be factorization-free by solving the linear system iteratively.

Percival.jl

- It is an implementation by Egmara Antunes dos Santos and Abel Soares Siqueira of a **matrix-free** augmented Lagrangian method.
- The method is designed for equality constraints and bounds.
- It uses an pure Julia implementation of `tron` to solve the bound-constrained subproblem.

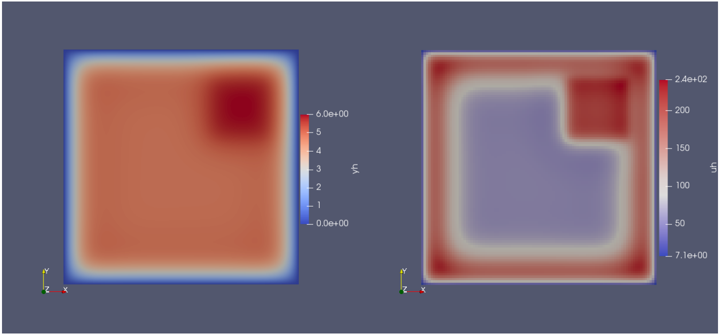
S. Arreckx, A. Lambe, Martins, J. R. R. A., & Orban, D.. A Matrix-Free Augmented Lagrangian Algorithm with Application to Large-Scale Structural Design Optimization., *Optimization And Engineering*, 17, pp. 359384, 2016.

Results

Solve our 2D Poisson-Boltzmann problem

`https://juliasmoothoptimizers.github.io/PDENLPModels.jl/dev/poisson-boltzman/`

Using Paraview we can print the vtk file obtained in Julia:



Distributed Poisson control problem with Dirichlet boundary conditions

https://jso-docs.github.io/solve-pdenlpmodels-with-jsosolvers/

https://tmigot.github.io/FletcherPenaltyNLPSolver/dev/example/

Benchmark CUTEst

We present the result of a benchmark of equality-constrained CUTEst problems with a maximum of 10000 variables and constraints (82 problems).
We compare DCISolver (using LDLFactorizations.jl for the tangential step), Ipopt, and Knitro with $max\_time = 20min$ and $tol = 10^{-5}$

`https://juliasmoothoptimizers.github.io/DCISolver.jl/dev/benchmark/`

Benchmark CUTEst: DCI-ldl vs Knitro

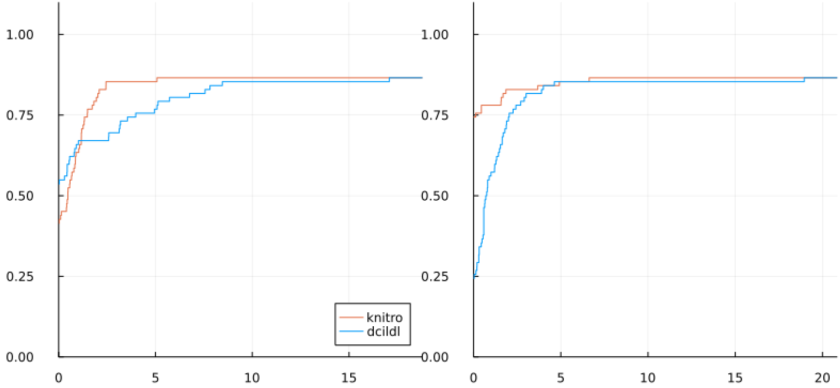


Figure : On the right with respect to time, and on the left with respect to number of evaluations of $f + c$.

Benchmark CUTEst: DCI-ldl vs Ipopt

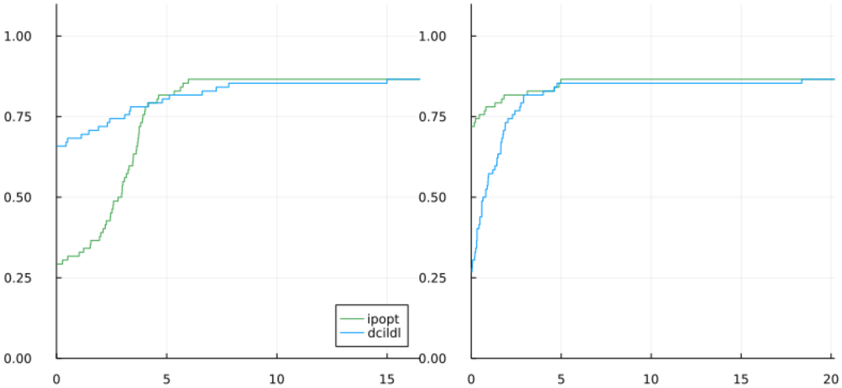


Figure : On the right with respect to time, and on the left with respect to number of evaluations of $f + c$.

Conclusions

PDE-constrained optimizer: a Toolbox

PDENLPModels.jl

Model the optimization problem and **pre-process** it as a (very large) continuous optimization problem.

use Gridap.jl (S. Badia & F. Verdusco, 2020) for the discretization of the PDE with finite-elements.

Solvers

JSO-interface to well-established solvers *Knitro* and *Ipopt*

Homemade solvers in pure Julia:

- **DCISolver.jl**
- **FletcherPenaltyNLPSolver.jl** (matrix-free !)
- **Percival.jl** (matrix-free !)

PDEOptimizationProblems.jl

Our collection of test problems and applications

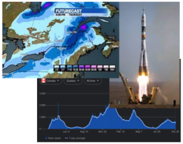
PDENLPModels.jl and co

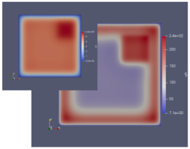
The ecosystem for PDE-constrained optimization in Julia

Within JuliaSmoothOptimizers we developed the whole process :

Real-world applications → Mathematical model implemented in PDENLPModels.jl → Solved with our solvers FletcherPenaltySolver.jl DCISolver.jl

Calibrated on PDEOptimizationProblems.jl → Get the solution to our application and ask more questions !





Perspectives:

- Maintain and improve this new ecosystem
- Handle more complex models (for instance bilevel programs)
- Tackle different applications

Thank you for your attention!

What I have used today

- CUTEst.jl : access the CUTEst test set in NLPModel format.
- NLPModelsModifiers.jl: to transform inequalities into bound constraints in one line via `SlackModel`.
- OptimizationProblems.jl: collection of test problems in JuMP and ADNLPModels format. (ps: great for 1st contribution!)
- SolverBenchmark.jl: run benchmark, generate performance profile and Latex tables.
- Stopping.jl: handle stopping criterion in your algorithms.