A Workflow for Designing Optimization Methods in the Julia Language

Optimization Days 2016

Abel Soares Siqueira
Mathematics Department, Federal University of Paraná, Curitiba/PR, Brazil

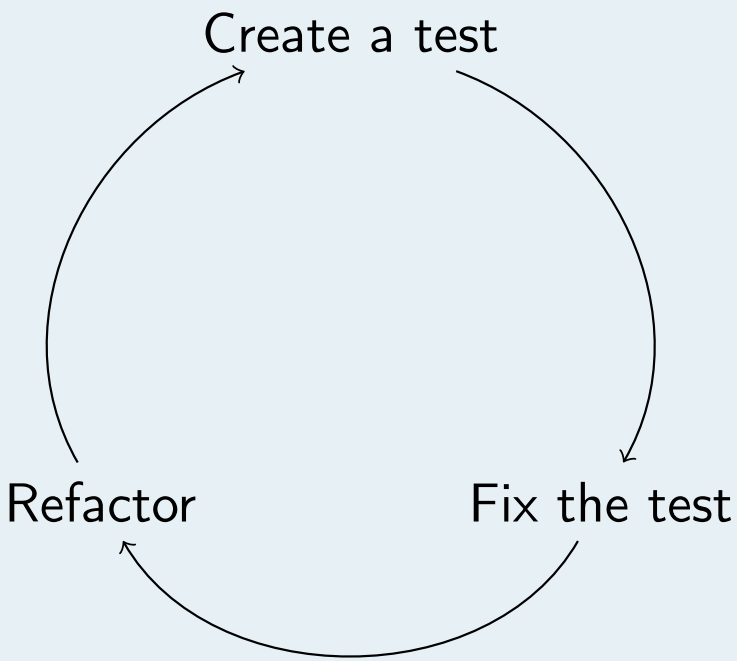
Dominique Orban
Mathematics and Industrial Engineering Department, École Polytechnique de Montréal

May, 4th

"... We should forget about the small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**..."
Donald Knuth

"The strategy is definitely: first **make it work**, then **make it right**, and, finally, **make it fast**."
Kent Beck

Test Driven Development



Problems as blocks

- Simplest representative problems;
- Classes of problems;
- Selection of problems from some specific repository.

Test nonzero exit flag

- Infinite loop;
- Budget limitations;
- Domain error.

Scale the problem

- Dense matrices;
- Inplace operations;
- Small efficiencies.

Outline

1. Use a test library (`FactCheck.jl`);

Outline

1. Use a test library (`FactCheck.jl`);
2.
   1. Write small problem;
   2. Write simplest code solving it (theory ✓, efficiency ✗);
   3. Repeat

Outline

1. Use a test library (`FactCheck.jl`);
2.
   1. Write small problem;
   2. Write simplest code solving it (theory ✓, efficiency ✗);
   3. Repeat
3. Write limitation tests (time, iteration, etc.) and code for it;

Outline

1. Use a test library (`FactCheck.jl`);
2.
   1. Write small problem;
   2. Write simplest code solving it (theory ✓, efficiency ✗);
   3. Repeat
3. Write limitation tests (time, iteration, etc.) and code for it;
4.
   1. Write class os problems (hundreds of runs, randomized, larger size);
   2. Write code solving it;
   3. Repeat

Outline

1. Use a test library (`FactCheck.jl`);
2.
   1. Write small problem;
   2. Write simplest code solving it (theory ✓, efficiency ✗);
   3. Repeat
3. Write limitation tests (time, iteration, etc.) and code for it;
4.
   1. Write class os problems (hundreds of runs, randomized, larger size);
   2. Write code solving it;
   3. Repeat
5.
   1. Choose problems from specific repository;
   2. Try to solve it;
   3. Repeat

Outline

1. Use a test library (`FactCheck.jl`);
2.
   1. Write small problem;
   2. Write simplest code solving it (theory ✓, efficiency ✗);
   3. Repeat
3. Write limitation tests (time, iteration, etc.) and code for it;
4.
   1. Write class os problems (hundreds of runs, randomized, larger size);
   2. Write code solving it;
   3. Repeat
5.
   1. Choose problems from specific repository;
   2. Try to solve it;
   3. Repeat
6. Improve the code.

Julia

- High level, High performance;
- Open source, multiplatform;
- Great C/Fortran interface;
- Easy syntax;
- Good practices (git, automated testing, code coverage);

Framework

- Develop with `Optimize.jl`
- Create the tests with `NLPModels.jl`;
- Easily access CUTEst with `CUTEst.jl`.

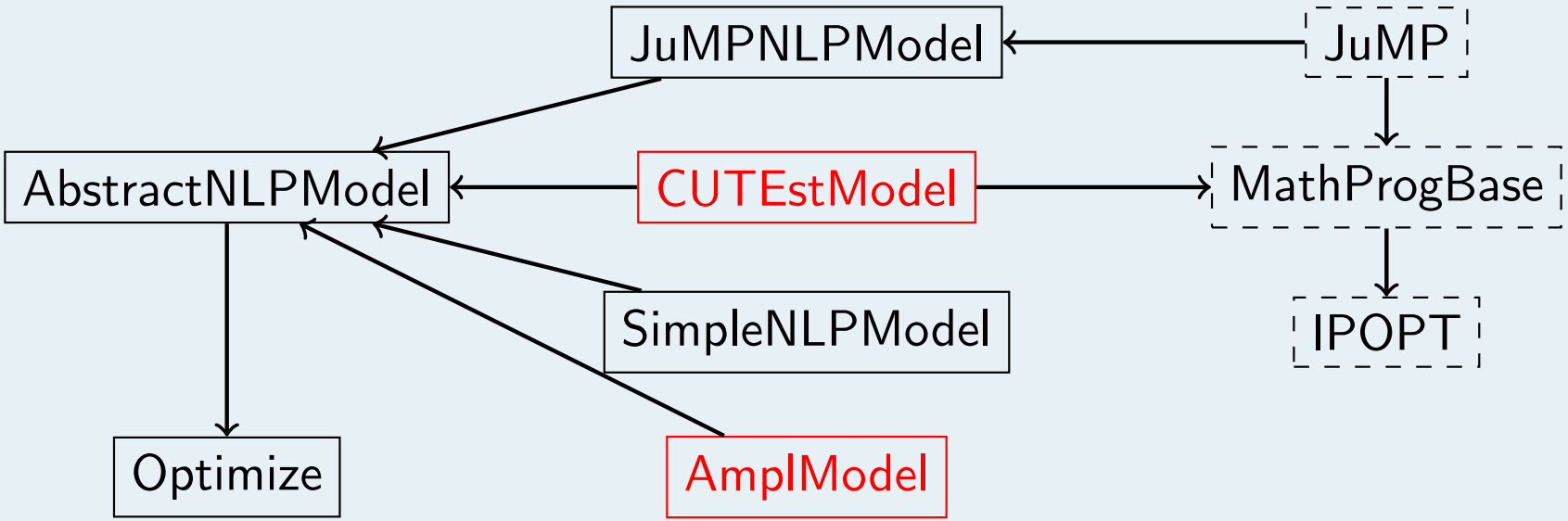
Optimize.jl

- Methods;
- Auxiliary algorithms and tools (trust region, line search, etc.);
- Testing and benchmarking.

NLPModels.jl

- Define `AbstractNLPModel`;
- Define `JuMPNLPModel` and `SimpleNLPModel`;
- AMPL and CUTEst models are derived from it;
- Allows future models.

NLPModels.jl



CUTEst

- Repository of Nonlinear Optimization problems;
- Provides subroutines to obtain the problem's information;
- Decodes the problem, compiles your code with the problem's and runs your main code;
- Widely used.

CUTEst.jl

- Easy to install;
- Easy to use;
- Helps in many stages of the development.

CUTEst (Fortran)

```
... !open problem and define variables
CALL cutest_cdimen(st, ifile, n, m)
if (m.GT.0) THEN
  STOP
ENDIF
CALL cutest_usetup(st, ifile, 7, 11, n, x, bl, bu)
CALL cutest_ufn(st, n, x, f)
... !close and end
```

CUTEst.jl

```
Pkg.add("CUTEst") # Once (Eventually)
```

```
using CUTEst

nlp = CUTEstModel("ROSENBR")
x = nlp.meta.x0

f = obj(nlp, x)
cutest_finalize(nlp)
```

Flavors

```
CALL cutest_ufn(st, n, x, f)
CALL cutest_cfn(st, n, m, x, f, c)
```

Wrapper

```
st = Cint[0]
f = [0.0]
c = zeros(m)
ufn(st, Cint[n], x, f)
cfn(st, Cint[n], Cint[m], x, f, c)
```

Flavors

```
CALL cutest_ufn(st, n, x, f)
CALL cutest_cfn(st, n, m, x, f, c)
```

Julian way

```
f = ufn(nlp, x)
f = ufn(n, x)
f, c = cfn(nlp, x)
f, c = cfn(n, m, x)
f = cfn!(nlp, x, c) # Inplace
f = cfn!(n, m, x, c) # Inplace
```

Flavors

```
CALL cutest_ufn(st, n, x, f)
CALL cutest_cfn(st, n, m, x, f, c)
```

AbstractModel way

```
f = obj(nlp, x)
c = cons(nlp, x)
f = objcons(nlp, x) # If unconstrained
f, c = objcons(nlp, x) # If constrained
```

Some functions

```
g = grad(nlp, x)
H = hess(nlp, x) # Sparse
H = hess(nlp, x, y) # Sparse
hrow, hcol, hval = hess_coord(nlp, x)
hrow, hcol, hval = hess_coord(nlp, x, y)
J = jac(nlp, x) # Sparse
jrow, jcol, jval = jac_coord(nlp, x)
Hv = hprod(nlp, x, v)
```

TRON: A practical example

Newton's Method for Large Bound-Constrained Optimization Problems

Chih-Jen Lin and Jorge J. Moré

SIAM Journal on Optimization Vol. 9, No. 4, pp. 1100-1127, 1999.

$$\min f(x) \quad \text{s. to} \quad x \in \Omega,$$

where Ω is

$$\Omega = \{x \in \mathbb{R}^n \mid \ell \leq x \leq u\},$$

but can be extended to

$$\Omega = \{x \in \mathbb{R}^n \mid \ell \leq c_i^T x \leq u,\ i \in \mathcal{I}\}.$$

Outline of iteration k

1. Compute a model

$$m_k(d) = \frac{1}{2} d^T B_k d + d^T g_k.$$

2. Compute a gradient step $s_k = P[x_k - \alpha_k g_k] - x_k$ such that

$$m_k(s_k) \leq \mu_0 g_k^T s_k, \qquad \text{and} \qquad \|s_k\| \leq \mu_1 \Delta_k.$$

3. Compute a step d_k better than s_k, i.e., further minimizing m_k with $\|d_k\| \leq \mu_1 \Delta_k$, and without leaving the bounds at $x_k + s_k$.
4. Update x_k and Δ_k using Trust Region rules.

Simplest problems

- $\min \; f(x) = \frac{1}{2}(x_1^2 + x_2^2)$;
- $\min \; f(x) = \frac{1}{2}(x_1^2 + x_2^2)$, subject to $1 \leq x_1, x_2 \leq 2$;
- $\min \; f(x) = \frac{1}{2}(x_1^2 + x_2^2)$, subject to $-1 \leq x_1, x_2 \leq 2$;
- $\min \; f(x) = \frac{1}{2}(x_1^2 + x_2^2)$, subject to $-1 \leq x_1 \leq 1, \; 1 \leq x_2 \leq 2$;
- $\min \; f(x) = \frac{1}{2}(x_1^2 + x_2^2)$, subject to $0 \leq x_1 \leq 1, \; 1 \leq x_2 \leq 2$;
- Rosenbrock with and without bounds;

Minimum implementation (satisfying theory)

1. Use $B_k = \nabla^2 f(x_k)$;
2. Use simple backtracking to find s_k;
3. Use $d_k = s_k$;
4. Use the Trust Region implemented in the framework (similar enough).

FactCheck

```
using FactCheck


facts("Simple test") do
  x0 = [1.2; 1.8]
  f(x) = dot(x,x)/2
  g(x) = x
  H(x) = eye(2)
  l = [1.0; 1.0]
  u = [2.0; 2.0]
  nlp = SimpleNLPModel(x0, f, grad=g, hess=H, lvar=l,
    uvar=u)
```

```
  x, fx, dual = tron(nlp)
  @fact x --> roughly(1)
  @fact fx --> roughly(f(1))
  @fact dual --> roughly(0.0)
end
```

Minimum implementation

```
function tron(nlp; m0 = 1e-2, m1 = 1.0)
  f(x) = obj(nlp, x)
  g(x) = grad(nlp, x)
  H(x) = hess(nlp, x)

  P(x) = max(min(x, u), l)
  dual(x) = norm(P(x - g(x)) - x)

  tr = TrustRegion(100.0)
  D() = get_property(tr, :radius)

  x = nlp.meta.x0
```

```
while dual(x) > 1e-6
  q(d) = 0.5*dot(d, H(x) * d) + dot(d, g(x))
  s(a) = P(x - a*g(x)) - x

  a = 1
  while q(s(a)) > m0*dot(g(x), s(a)) ||
      norm(s(a)) > m1*D()
    a *= 0.9
  end
  xp = x + s(a)
  rho = ratio(f(x), f(xp), q(xp-x))
  if acceptable(tr, rho)
    x = xp
  end
  update!(tr, rho, norm(s(a)))
```

```
  end
  return x, f(x), dual(x)
end
```

Limits

```
f(x) = begin sleep(0.1); sum(exp(x)) end
g(x) = begin sleep(0.1); exp(x) end
H(x) = begin sleep(0.1); spdiagm(exp(x), 0, 10, 10) end
x0 = 10*ones(10)
```

Class os tests

- $\min\ f(x) = \frac{1}{2}(x - r)^T Q^T \Lambda Q(x - r) + 1$, where
  - $r = (1, \ldots, 1)^T$;
  - Q is an orthogonal matrix;
  - $\Lambda = \text{diag}(10^{-2}, \ldots, 1)^T$
- $\min\ f(x) = \frac{1}{2}x^T Bx + g^T x$, subject to $\ell \leq x \leq u$, where
  - $B = Q^T \Lambda Q > 0$.
  - Build solution and choose g
- Generalized rosenbrock with bounds.

Improvements

- Matrix-free (hprod instead of hess);
- Store repeated function calls;
- Use inplace operations;

Future Work

- Implement various methods;
- Create a documentation/tutorial/example;
- Improve the benchmark;
- Benchmark many methods;
- Problem selector (Simple and CUTEst).

Thanks