JuMP: Nonlinear Modeling with Exact Hessians in Julia

Miles Lubin, Iain Dunning, and Joey Huchette

MIT Operations Research Center

INFORMS 2014 – November 9, 2014

- Challenges traditional performance trade-offs: **high-level**, **dynamic**, *and* **fast**
- Familiar syntax for Python and MATLAB users
- Technical advances that can change how we compute in the field of Operations Research (Lubin and Dunning, forthcoming in IJOC)

JuMP

- Solver-independent, fast, extensible, open-source algebraic modeling language for Mathematical Programming embedded in Julia
    - cf. AMPL, GAMS, Pyomo, PuLP, YALMIP, ...

JuMP

- Solver-independent, fast, extensible, open-source algebraic modeling language for Mathematical Programming embedded in Julia
    - cf. AMPL, GAMS, Pyomo, PuLP, YALMIP, ...
- Version 0.1 released in October 2013 (LP, QP, MILP)
- Version 0.2 released in December 2013 (Advanced MILP)

  $\uparrow$ Iain Dunning's talk tomorrow
- Version 0.5 released in May 2014 (**NLP**)

Nonlinear modeling

$$\begin{aligned} &\min\ f(x) \\ &\text{s.t. } g(x) \le 0 \end{aligned}$$

- User inputs closed-form expressions for f and g
- Modeling language communicates with solver to provide derivatives
  - Traditionally, Hessian of Lagrangian:

$$\nabla^2 f(x) + \sum_i \lambda_i \nabla^2 g(x)$$

State of the art

NL files: AMPL (or others...) write .nl file to disk, solver uses `asl` library to read and query derivatives

- Gay, D. 1997. Hooking your solver to AMPL. Technical report 97-4-06. Bell Laboratories.

State of the art

NL files: AMPL (or others...) write .nl file to disk, solver uses `asl` library to read and query derivatives

- Gay, D. 1997. Hooking your solver to AMPL. Technical report 97-4-06. Bell Laboratories.

Can we do better?

- Improve performance by avoiding writing to disk
- Flexibility of lightweight, pure-Julia implementation

Methods for computing derivatives

- Symbolic
  - Does not scale well to second-order derivatives
- Automatic Differentiation (AD)
  - Reverse mode
  - Forward mode

Reverse mode AD in 2 slides

Assume function f is given in the form,

```
**function** $f(x_1, x_2, \dots, x_n)$
    **for** $i = n+1, n+2, \dots, N$ **do**
        $x_i \leftarrow g_i(x_{S_i})$
    **end for**
    **return** $x_N$
**end function**
```

- S_i – input to ith operation, subset of $\{1, 2, \dots, i-1\}$, $(|S_i| \leq 2)$
- g_i – "basic" operation: $+, *, \text{sqrt}, \sin, \exp, \log, \dots$

Then

$$\frac{\partial f(x)}{\partial x_i} = \frac{\partial x_N}{\partial x_i} = \sum_{j:i \in S_j} \frac{\partial x_N}{\partial x_j} \frac{\partial g_j(x_{S_j})}{\partial x_i}$$

Note $i \in S_j$ implies $j > i$, which means that we can **compute all partials by running the function in reverse**:

```
$\frac{\partial x_N}{\partial x_N} \leftarrow 1$
for $i = N-1, N-2, \ldots, 2, 1$ do
    if  $i > n$  then
        for $k \in S_i$ do
            Compute and store $\frac{\partial g_i(x_{S_i})}{\partial x_k}$
        end for
    end if
    $\frac{\partial x_N}{\partial x_i} \leftarrow \sum_{j: i \in S_j} \frac{\partial x_N}{\partial x_j} \frac{\partial g_j(x_{S_j})}{\partial x_i}$
end for
```

At the end we obtain

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n} \right)$$

Discussion

- Can all functions be represented in the procedural form?
- What's the computational cost to compute a gradient?

Discussion

- Can all functions be represented in the procedural form?
  - Trivial for closed-form algebraic expressions (good for JuMP)
  - Yes in general, but sequence of operations may change over domain
- What's the computational cost to compute a gradient?

Discussion

- Can all functions be represented in the procedural form?
    - Trivial for closed-form algebraic expressions (good for JuMP)
    - Yes in general, but sequence of operations may change over domain
- What's the computational cost to compute a gradient?
    - $O(1)$ function evaluations! (c.f. $O(n)$ for finite differences)
    - $O(\#operations)$ storage

Example

$$f(x_1, x_2) = \sin(x_1)\exp(x_2)$$

```
**function** $f(x_1, x_2)$
    $x_3 \leftarrow \sin(x_1)$
    $x_4 \leftarrow \exp(x_2)$
    $x_5 \leftarrow x_3 * x_4$
    **return** $x_5$
**end function**
```

```
function $\nabla f(x_1, x_2)$
    $x_3 \leftarrow \sin(x_1)$
    $x_4 \leftarrow \exp(x_2)$
    $x_5 \leftarrow x_3 * x_4$
    $z_5 \leftarrow 1$
    $z_4 \leftarrow x_3$
    $z_3 \leftarrow x_4$
    $z_2 \leftarrow z_4 \exp(x_2)$
    $z_1 \leftarrow z_3 \cos(x_1)$
    return $(z_1, z_2)$
end function
```

$z_i := \frac{\partial x_5}{\partial x_i}$

One can view reverse-mode AD as a method for *transforming code* to compute a function $f : \mathbb{R}^n \to \mathbb{R}$ into code to compute the gradient function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$.

- Usually implemented by *interpreting* each instruction

One can view reverse-mode AD as a method for *transforming code* to compute a function $f : \mathbb{R}^n \to \mathbb{R}$ into code to compute the gradient function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$.

- Usually implemented by *interpreting* each instruction
- Why not just generate new code and compile it instead?
  - Let compiler optimize, essentially as fast as hand-written derivatives
  - Not a new idea, but historically hard to implement and difficult to use (e.g., AMPL's `nlc`)

One can view reverse-mode AD as a method for *transforming code* to compute a function $f : \mathbb{R}^n \to \mathbb{R}$ into code to compute the gradient function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$.

- Usually implemented by *interpreting* each instruction
- Why not just generate new code and compile it instead?
  - Let compiler optimize, essentially as fast as hand-written derivatives
  - Not a new idea, but historically hard to implement and difficult to use (e.g., AMPL's `nlc`)
- In Julia, **easy to manipulate and compile expressions at runtime**, so this is what we do!
  - 500 lines of code, transparent to the user

Forward-mode AD

- $f(x + y\epsilon) = f(x) + yf'(x)\epsilon$
- Idea: extend all operations to carry first-order taylor expansion terms

- Does this require access to the “procedural form”?

- What’s the computational cost?

- Does this require access to the "procedural form"?
  - No, implement via operator overloading*
  - Write generic (templated) code in Julia
- What's the computational cost?

Wait, isn't operator overloading slow?

```
*(z::Dual, w::Dual) = dual(real(z)*real(w),
      epsilon(z)*real(w)+real(z)*epsilon(w))
julia> code_native(*,(Dual{Float64},Dual{Float64}))
        push    RBP
        mov     RBP, RSP
        vmulsd  XMM3, XMM0, XMM3
        vmulsd  XMM1, XMM1, XMM2
        vaddsd  XMM1, XMM1, XMM3
        vmulsd  XMM0, XMM0, XMM2
        pop     RBP
        ret
```

- Efficient code for *immutable* types

- Does this require access to the "procedural form"?
  - No, implement via operator overloading
  - Write generic (templated) code in Julia
- What's the computational cost?

- Does this require access to the “procedural form”?
  - No, implement via operator overloading
  - Write generic (templated) code in Julia
- What’s the computational cost?
  - *Directional derivatives* in $O(1)$ evaluations, like finite differencing
  - So $O(n)$ evaluations for Jacobian of $f : \mathbb{R}^n \to \mathbb{R}^k$
  - Doesn’t scale like reverse-mode for gradients, but...

Computing Hessians

Efficient interior-point solvers (Ipopt, ...) need the $n \times n$ Hessian matrix:

$$\nabla^2 f(x)_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}.$$

The Jacobian of $\nabla f(x)$ is $\nabla^2 f(x)$. So compute full Hessian matrix in $O(n)$ evaluations of f.

Computing Hessians

Efficient interior-point solvers (Ipopt, ...) need the $n \times n$ Hessian matrix:

$$\nabla^2 f(x)_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}.$$

The Jacobian of $\nabla f(x)$ is $\nabla^2 f(x)$. So compute full Hessian matrix in $O(n)$ evaluations of f.

Alternatively: *Hessian-vector* product $\nabla^2 f(x)d$ is directional derivative of $\nabla f(x)$, can compute in $O(1)$ evaluations of f.

Exploiting sparsity

Usually Hessian matrix is *very* sparse.

If diagonal, just need to evaluate $\nabla^2 f(x) d$ with vector $d = (1, \cdots, 1)$ to "recover" all nonzero entries of $\nabla^2 f(x)$.

Exploiting sparsity

Usually Hessian matrix is *very* sparse.

If diagonal, just need to evaluate $\nabla^2 f(x) d$ with vector $d = (1, \cdots, 1)$ to "recover" all nonzero entries of $\nabla^2 f(x)$.

In general, what is the smallest number of Hessian-vector products needed to recover all nonzero elements of $\nabla^2 f(x)$?

- Acyclic graph coloring problem, NP-Hard (Coleman and Cai, 1986)
- We implement the coloring heuristic of Gebremedhin et al (2009).

Benchmarks

Model generation time: Time between user pressing enter and solver starting

Function evaluation time: Time evaluating derivatives

```
Total CPU secs in IPOPT (w/o function evaluations)   =    224.725
Total CPU secs in NLP function evaluations           =     29.510
```

Performance goal: **Don't be the bottleneck!**

clnlbeam model

```
alpha = 350
h     = 1/N

m = Model()

@defVar(m, -1 <= t[1:(N+1)] <= 1)
@defVar(m, -0.05 <= x[1:(N+1)] <= 0.05)
@defVar(m, u[1:(N+1)])

@setNLObjective(m, Min, sum{ 0.5*h*(u[i+1]^2+u[i]^2) +
                             0.5*alpha*h*(cos(t[i+1]) +
                               cos(t[i])), i=1:N})

@addNLConstraint(m, cons1[i=1:N],
    x[i+1] - x[i] - 0.5*h*(sin(t[i+1])+sin(t[i])) == 0)
@addConstraint(m, cons2[i=1:N],
    t[i+1] - t[i] - (0.5h)*u[i+1] - (0.5h)*u[i] == 0)
```

Table: Model generation time (sec.)

N =	JuMP	AMPL	Pyomo	YALMIP
5,000	0.6	0.2	4.8	116.6
50,000	1.9	2.8	44.2	OOM
500,000	17.2	211.6	636.1	OOM

OOM = Exceeded 64GB of RAM!
Model has $3N$ variables and $2N$ constraints. Diagonal Hessian.
Pyomo writes .nl files. YALMIP pure MATLAB.
For $N = 500,000$, .nl file is 180MB.

Table: Hessian evaluation time (sec.)

N =	JuMP	`asl`
5,000	0.004	0.002
50,000	0.055	0.042
500,000	0.573	0.438

`asl`: AMPL & Pyomo. YALMIP does not provide Hessians.

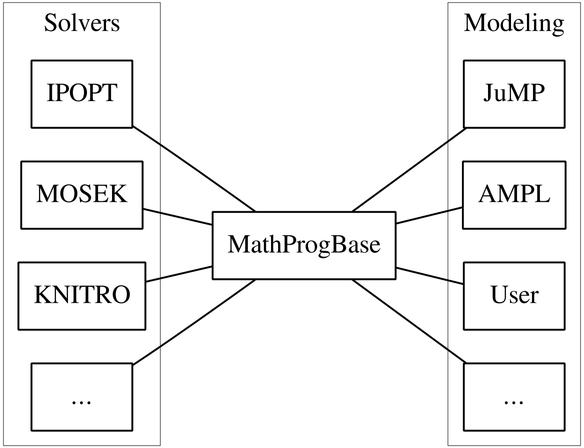
Connecting to solvers

JuMP uses solver-independent **MathProgBase** interface for connecting to solvers.

For LP/MILP: CPLEX, Clp, Cbc, ECOS, GLPK, Gurobi, Mosek

For **NLP**: Ipopt, KNITRO, Mosek, NLopt

All interfaces *in-memory*. **Order of magnitude easier** to interface with C and Fortran from Julia compared with Python and MATLAB.


Solvers
IPOPT
MOSEK
KNITRO
...
MathProgBase
Modeling
JuMP
AMPL
User
...

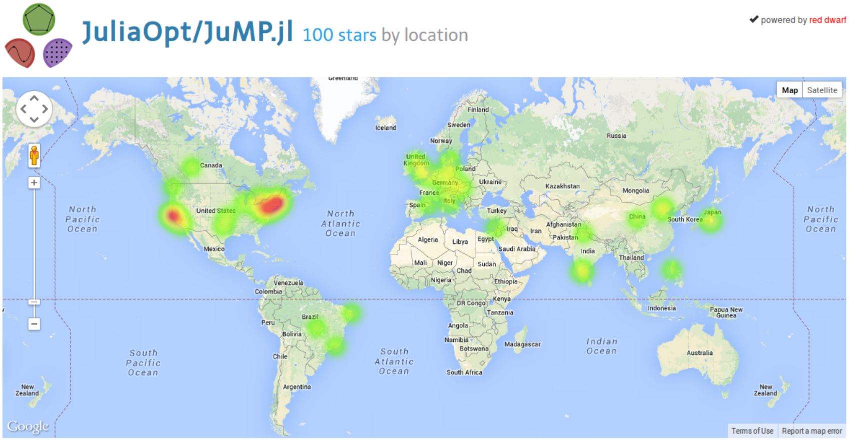
Availability

`http://github.com/JuliaOpt/JuMP.jl`

- Available via Julia package manager
- Easy installation of open-source solvers on all platforms[1]
- LGPL license

[1] Thanks to many contributors

Who's using JuMP?



- 4,000 monthly hits to GitHub page (50% from outside of USA)
- "Integer Programming" and "Optimization Methods" courses at MIT
- ...

Thank you!

References

- M. Lubin and I. Dunning, “Computing in Operations Research using Julia”, *INFORMS Journal on Computing*, forthcoming.
  - Early paper, does not include description of automatic differentiation
- A. H. Gebremedhin et al., “Efficient computation of sparse hessians using coloring and automatic differentiation”, *INFORMS Journal on Computing*, 2009.
  - Graph coloring algorithm used by JuMP
- Blog post by Justin Domke
  - Simple explanation of reverse-mode AD
- ReverseDiffSparse.jl and DualNumbers.jl
  - Modular implementations of reverse mode and forward mode AD used by JuMP