AUTOMATIC DIFFERENTIATION TECHNIQUES USED IN JUMP

Miles Lubin, Iain Dunning, and Joey Huchette

June 22, 2016

MIT Operations Research Center

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

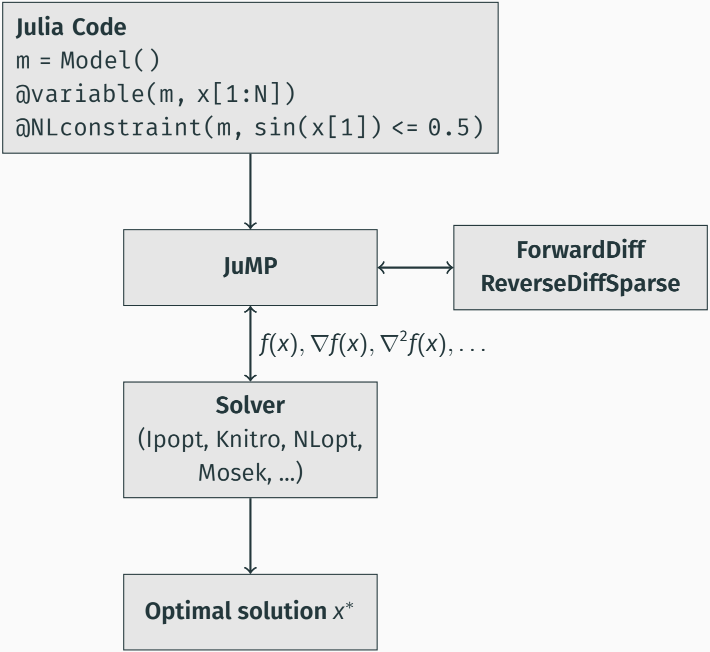
`http://www.juliaopt.org/`

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

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

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

http://nbviewer.ipython.org/github/JuliaOpt/
juliaopt-notebooks/blob/master/notebooks/
JuMP-Rocket.ipynb


Julia Code
m = Model()
@variable(m, x[1:N])
@NLconstraint(m, sin(x[1]) <= 0.5)
JuMP
ForwardDiff
ReverseDiffSparse
$f(x), \nabla f(x), \nabla^2 f(x), \ldots$
Solver
(Ipopt, Knitro, NLopt,
Mosek, ...)
Optimal solution x^*

Will discuss how JuMP computes derivatives: algorithms and data structures.

Related work:

- Machine Learning: TensorFlow, Torch, etc.
- Statistics: Stan
- PDEs: FEniCS, UFL
- Control: CasADi

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

Assume function f is given in the form,

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

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

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

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

What's the computational cost to compute a gradient?

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 $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$.

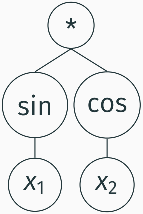
- Usually implemented by *interpreting* each instruction
- Why not just generate new code and compile it instead?
  - Let compiler optimize, essentially as fast as hand-written derivatives
  - Not a new idea, but historically hard to implement and difficult to use (e.g., AMPL's `nlc`)
  - In Julia, implementation easy but compilation time was prohibitive, so we now "interpret" expressions (ReverseDiffSparse v0.3 $\to$ v0.5)

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, implementation easy but compilation time was prohibitive, so we now "interpret" expressions (ReverseDiffSparse v0.3 $\to$ v0.5)
  - See also ReverseDiffSource.jl

Recall each operation g_i is associated with a set S_i – list of inputs.
Useful to think of operations as *nodes in a graph*, inputs as children.

Example: $\sin(x_1)\cos(x_2)$



Call this *expression tree* (or *expression graph*).

DATA STRUCTURE FOR EXPRESSION TREES

- JuMP's expression trees (loops unrolled) can easily have millions of nodes
- May have thousands of such constraints in a given optimization problem
- Billions of long-lived GC'd objects floating around is not great for performance in Julia

- JuMP's expression trees (loops unrolled) can easily have millions of nodes
- May have thousands of such constraints in a given optimization problem
- Billions of long-lived GC'd objects floating around is not great for performance in Julia

Problem: Design an efficient data structure for expression trees with a constant number of GC'd objects, regardless of size of tree.

- `Graphs` and `LightGraphs` use `Vector{Vector}` for list of children.

Problem: Design an efficient data structure for expression trees with a constant number of GC'd objects, regardless of size of tree.

Problem: Design an efficient data structure for expression trees with a constant number of GC'd objects, regardless of size of tree.

Solution: Use a single vector of `immutables`. Each element stores the index to its parent. Order the vector so that a linear pass corresponds to running function forward or backward. (c.f. "tapes")

Problem: Design an efficient data structure for expression trees with a constant number of GC'd objects, regardless of size of tree.

Solution: Use a single vector of `immutables`. Each element stores the index to its parent. Order the vector so that a linear pass corresponds to running function forward or backward. (c.f. "tapes")

That form makes it easy to access parents but not list of children. Use a CSC sparse matrix with children on the columns (adjacency matrix). (conversion code)

Problem: Design an efficient data structure for expression trees with a constant number of GC'd objects, regardless of size of tree.

Solution: Use a single vector of `immutables`. Each element stores the index to its parent. Order the vector so that a linear pass corresponds to running function forward or backward. (c.f. "tapes")

That form makes it easy to access parents but not list of children. Use a CSC sparse matrix with children on the columns (adjacency matrix). (conversion code)

Final data structure per expression tree:

- `Vector` of `immutables`
- `SparseMatrixCSC`

JuMP uses **forward-mode AD** (see Jarrett's talk next) for:

- Second-order derivatives, composed with reverse mode
- Gradients of user-defined functions

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

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

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

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

Hessian-vector product $\nabla^2 f(x)d$ is directional derivative of $\nabla f(x)$, can compute in $O(1)$ evaluations of f using forward mode ad composed with reverse mode.

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 $\nabla^2 f(x)$.

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).

```
function squareroot(x)
    z = x # Initial starting point for Newton's method
    while abs(z*z - x) > 1e-13
        z = z - (z*z-x)/(2z)
    end
    return z
end
JuMP.register(:squareroot, 1, squareroot, autodiff=true)

m = Model()
@variable(m, x[1:2], start=0.5)
@objective(m, Max, sum(x))
@NLconstraint(m, squareroot(x[1]^2+x[2]^2) <= 1)
solve(m)
```

Limitations:

- Function must accept generic number type, follow guidelines for ForwardDiff.jl
- No Hessians yet
- Low-dimensional functions only, no vector input

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!**

```
alpha = 350
h     = 1/N

m = Model()

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

@NLobjective(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})

@NLconstraint(m, cons1[i=1:N],
    x[i+1] - x[i] - 0.5*h*(sin(t[i+1])+sin(t[i])) == 0)
@constraint(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.)

Instance	**JuMP**	**Commercial** AMPL	GAMS	**Open-source** Pyomo	YALMIP
clnlbeam-5	12	0	0	5	76
clnlbeam-50	14	2	3	44	>600
clnlbeam-500	38	22	35	453	>600
acpower-1	18	0	0	3	-
acpower-10	21	1	2	26	-
acpower-100	66	14	16	261	-

`clnlbeam` has diagonal Hessian, `acpower` complex hessian structure.

Pyomo uses AMPL's open-source AD library. YALMIP pure MATLAB.

Table: Time (sec.) to evaluate derivatives (including gradients, Jacobians, and Hessians) during 3 iterations, as reported by Ipopt.

		Commercial	
Instance	**JuMP**	AMPL	GAMS
clnlbeam-5	0.03	0.03	0.09
clnlbeam-50	0.39	0.34	0.74
clnlbeam-500	4.72	3.40	15.69
acpower-1	0.07	0.02	0.06
acpower-10	0.66	0.30	0.53
acpower-100	6.11	3.20	18.13

Thank you to Julia developers, JuliaOpt contributors, JuMP users, JuliaCon organizers, and the audience!

More on AD in JuMP:
`http://arxiv.org/abs/1508.01982`

Explanation of reverse mode inspired by Justin Domke's blog post