Automatic differentiation in Julia

Miles Lubin and Jarrett Revels

17th Euro AD Workshop

August 20, 2015

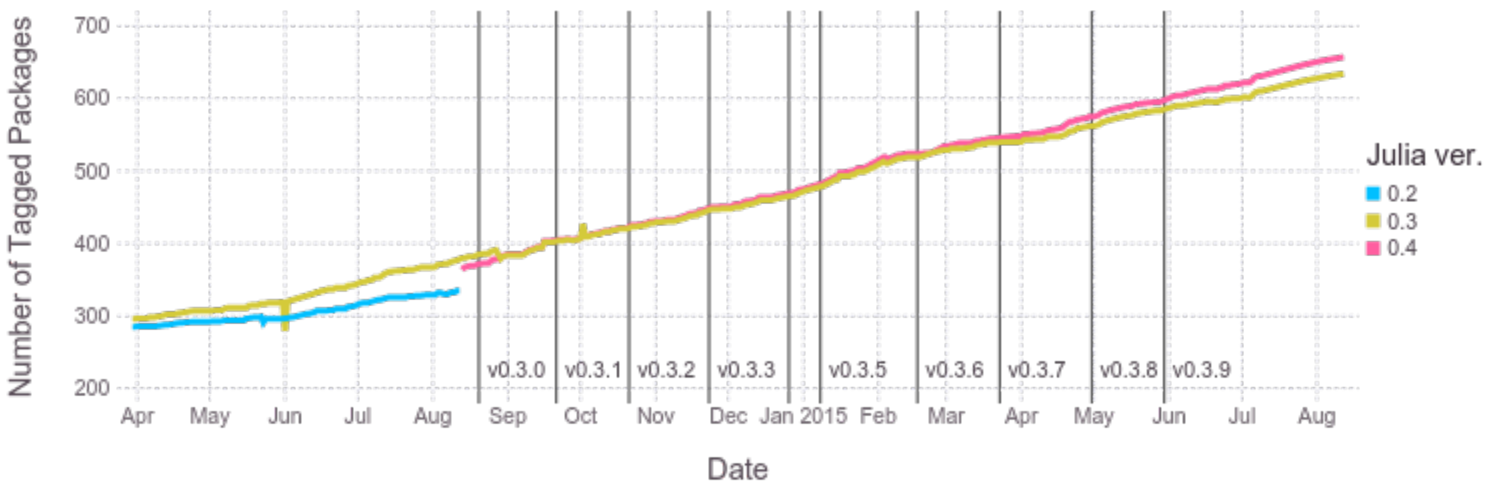
Why Julia?

- Fast like C++, high level like Python and Matlab
  - That’s the idea at least
- Solves the “multiple-language problem” in technical computing

Julia timeline

- Julia publicly announced, 2012
- Julia 0.1 release, February 2013
- Julia 0.2 release, November 2013
- 1st annual JuliaCon held in Chicago, 2014
- Julia 0.3 release, August 2014
- 2nd annual JuliaCon held at MIT, 2015
- Julia 0.4 release, Soon™

It's not 1.0, but people find it useful...



Total number of packages by Julia version

JuliaDiff is a:

- [Web page]
- [Github organization]

to help organize the development of AD tools in Julia.

Most people interact with JuliaDiff through:

```
optimize(f, method=:l_bfgs, autodiff=true)
```

```julia
function rosenbrock100(x::Vector)
  out = zero(eltype(x))
  for i in 1:div(length(x),2)
    out += 100*(x[2i-1]^2 - x[2i])^2 + (x[2i-1]-1)^2
  end
  out
end

@time optimize(rosenbrock100, zeros(100),
      method = :l_bfgs, iterations=21)
# elapsed time: 0.003834211 seconds
# Value of Function at Minimum: 3.419262

@time optimize(rosenbrock100, zeros(100),
      method = :l_bfgs, iterations=21, autodiff=true)
# elapsed time: 0.002318992 seconds
# Value of Function at Minimum: 0.000000
```

Outline

- Introduction to technical features of Julia interesting for AD
- ForwardDiff package
- JuMP modeling language for optimization

Follow along

https://juliabox.org/

https://github.com/mlubin/EuroAD2015

JuMP - a modeling language for linear and nonlinear optimization

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

- *All functions given as closed-form algebraic expressions*

State of the art

Commercial tools:

- AMPL (Gay, Fourer)
  - De-facto standard .nl exchange format
- GAMS

Open source:

- Pyomo
  - Writes to .nl format, doesn’t implement AD
- YALMIP
  - Not large scale, no hessians
- CasADi

What JuMP looks like…

```
m = Model(solver=IpoptSolver())
@defVar(m, x[1:n])
@setNLObjective(m, Min,
   sum{ exp(x[i]^2), i = 1:n} )
@addNLConstraint(m,
  prod{ x[i], i=1:n} <= 1)
solve(m)
```

JuMP is a domain-specific language

```
myset = [“cat”, “dog”]
@defVar(m, x[myset])
@addNLConstraint(m,
   sin(x[“dog”]) <= 0.5)
```

Useful for indexing over:

- Edges in a graph
- Types of widgets to produce
- ...

JuMP is a domain-specific language

```
@defVar(m, l[i] ≤ x[i=1:N] ≤ 2i)
@defVar(m, t ≥ 0)
@addNLConstraint(m, limit[i=(3:N-1)],
            exp(x[i]) ≤ t)
```

Equivalent to

```
for i in 3:N-1
  @addNLConstraint(m, exp(x[i]) ≤ t)
end
```

Easy to query derivatives

```
m = Model(); @defVar(m, x); @defVar(m, y)
@setNLObjective(m, Min, sin(x) + sin(y))


values = [2.0, 3.0]
d = JuMPNLPEvaluator(m); initialize(d, [:Grad])
objval = eval_f(d, values)
∇f = zeros(2)
eval_grad_f(d, ∇f, values)
# ∇f == [cos(2.0), cos(3.0)]
```

Benchmarks

- Build model in memory, prepare AD
  - **Model generation time**
- Give to Ipopt for 5 iterations, report time spent in NLP evaluations (incl. **gradients**, **jacobians**, **hessian of the lagrangian**)

https://github.com/mlubin/JuMPSupplement

Model generation time (sec.)

		Commercial		Open-source	
Instance	**JuMP**	AMPL	GAMS	Pyomo	YALMIP
clnlbeam-5	9	0	0	5	117
clnlbeam-50	11	2	3	43	>600
clnlbeam-500	28	21	34	424	>600
acpower-1	22	0	0	3	-
acpower-10	28	1	6	26	-
acpower-100	54	16	471	263	-

Derivative evaluation time (sec.)

		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	0.08	0.02	0.19
acpower-10	0.81	0.35	5.07
acpower-100	9.28	3.42	424.89

What do we do?

- `sum{}` and `prod{}` translated to accumulation loops
  - AMPL flattens out
- Apply Reverse-mode AD to this expression graph
  - Recompute instead of storing intermediate terms inside loops
  - Fuse reverse and forward mode of top-level loops (thanks Paul)

What do we do?

- Applying reverse mode, use Julia’s code generation facilities to **generate and compile a function at runtime** which evaluates the gradient
- This gives us gradients and Jacobians

From gradients to Hessians

- Apply forward-mode to gradient functions to evaluate Hessian-vector product
- Acyclic graph coloring heuristic of Gebremedhin et. al (2009)

$$
\begin{bmatrix}
h_{11} & h_{12} & & h_{14} & \\
h_{12} & h_{22} & h_{23} & & \\
& h_{23} & h_{33} & & \\
h_{14} & & & h_{44} & \\
& & & & h_{55}
\end{bmatrix}
\begin{bmatrix}
1 & \\
& 1 \\
1 & \\
& 1 \\
& 1
\end{bmatrix}
=
\begin{bmatrix}
h_{11} & h_{12} + h_{14} \\
h_{12} + h_{23} & h_{22} \\
h_{33} & h_{23} \\
h_{14} & h_{44} \\
& h_{55}
\end{bmatrix}
$$

Discussion

- Loops versus flattened out expression graphs
  - Algebraic simplifications? Presolve?
  - Composability

Thanks!