



SIAM Conference on Optimization, May 2008

# On Automatic Differentiation for Optimization

David M. Gay

Optimization and Uncertainty Estimation

<http://www.sandia.gov/~dmgay>

*dmgay@sandia.gov*

+1-505-284-1456

Sandia is a multiprogram laboratory operated by Sandia Corporation,  
a Lockheed Martin Company, for the United States Department of Energy's  
National Nuclear Security Administration under contract DE-AC04-94AL85000.



# Outline

- Why AD?
- Forward and Backward
- Implementation approaches
- Sacado package in Trilinos
- Hessian-vector products
- Concluding remarks



## Why AD?

Some algorithms need gradients and perhaps Hessians. Possibilities...

- Finite-differences
  - + work with black boxes
  - but can be expensive
  - and introduce truncation error.



## Why AD? (cont'd)

- Analytic derivatives
  - + no truncation error
  - + available from symbolic-computation packages
  - tedious and error-prone if done by hand
  - can be inefficient
  - possible interfacing issues



## Why AD? (cont'd)

- Automatic Differentiation (AD)
  - + no truncation error (uses chain rule)
  - + reverse mode = efficient for gradients
  - + sometimes easy to use
  - can take lots of memory
  - possible interfacing issues
  - if-then-else: which side at break?



## Forward and Backward

Two modes:

- Forward: recur partials (w.r.t. independent variables) of operands at each operation
  - + good locality and memory use
  - + for  $n = 1$  can compute high-order deriv's (Taylor series)
  - slow for large  $n$  ( $\#$  indep. vars)



## Forward and Backward (cont'd)

- Backward: recur partials of final result w.r.t. intermediate results
  - +  $f$  and  $\nabla f$  in time proportional to computing  $f$
  - memory use proportional to number of operations



## Forward and Backward (cont'd)

- Backward: recur partials of final result w.r.t. intermediate results
  - +  $f$  and  $\nabla f$  in time proportional to computing  $f$
  - memory use proportional to number of operations





## Implementation Approaches

Implementations must augment function computations with recurrence of partial derivatives. *Logically equivalent to obtaining and manipulating an expression graph.*

- Preprocessor consumes source code (e.g., C or Fortran) and emits modified source.
  - Examples: AUGMENT, ADIFOR, ADIC



## Implementation Approaches (cont'd)

- Operator overloading in some programming languages, such as C++ or Fortran
  - Examples: ADOL-C, ADOL-F, Sacado
- Modeling language (manipulates expression graph behind the scenes)
  - Examples: AMPL, GAMS

Many tools exist; <http://www.autodiff.org> lists 29.



## Implementation: Reverse-mode Inner Loops

Reverse-mode derivative propagation: all multiplications and additions. Op'ns of form

$$a \leftarrow a + b \times c$$

AMPL/solver interface lib.:

```
do *d->a.rp += *d->b.rp * *d->c.rp;  
while(d = d->next);
```

Sacado:

```
do d->c->aval += *d->a * d->b->aval;  
while((d = d->next));
```



## Sacado

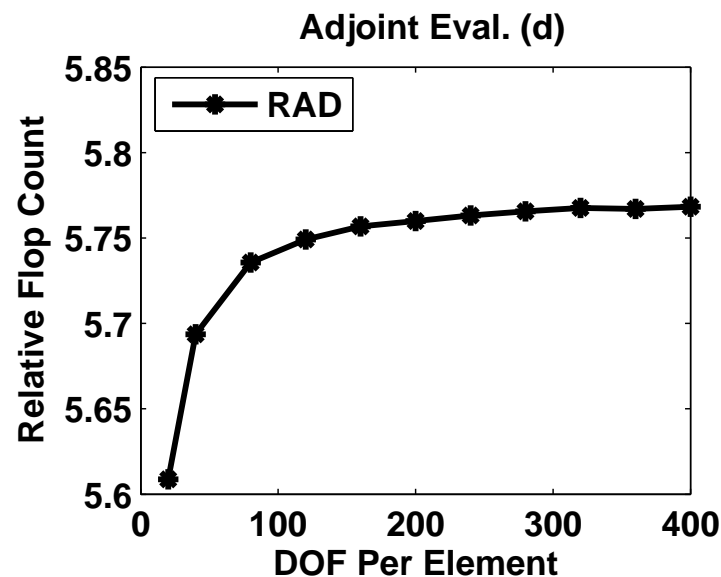
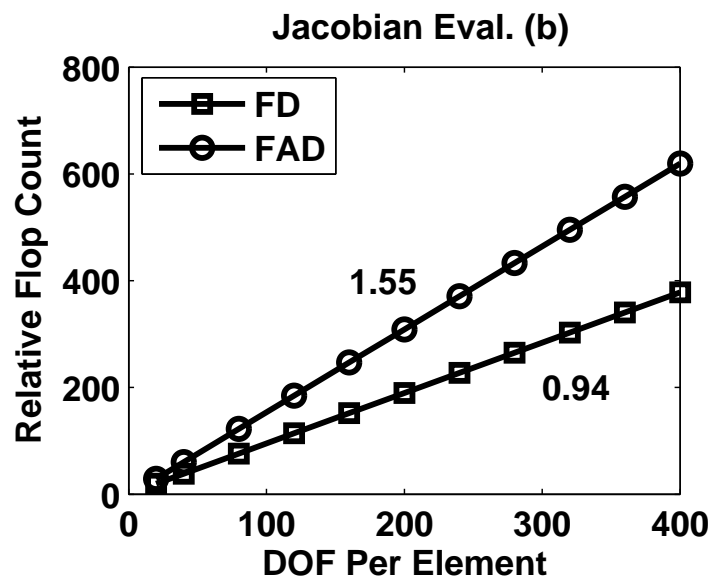
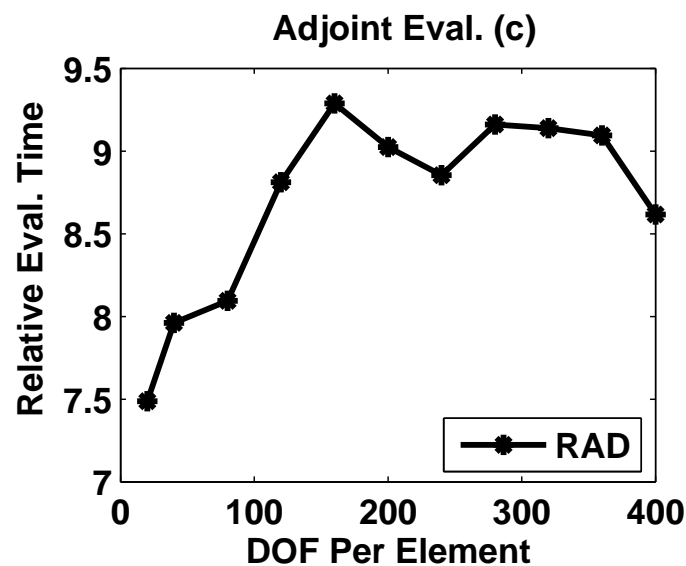
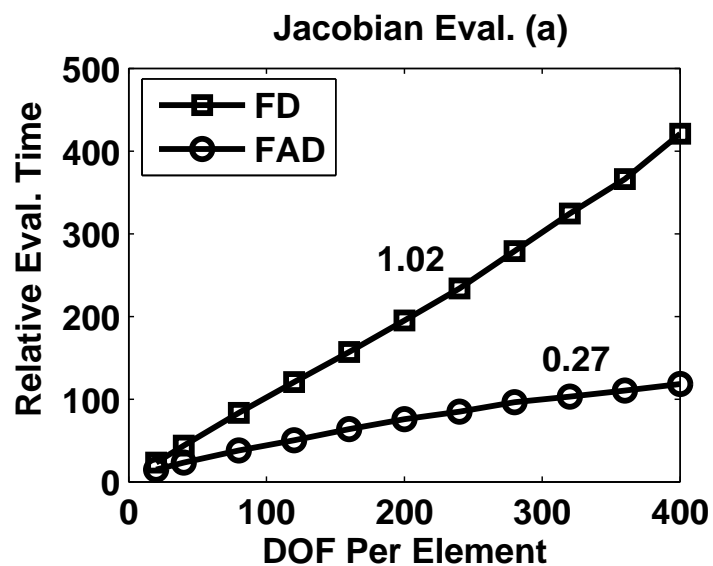
*Trilinos* = collection of open-source tools for scientific computing in C++; see

<http://trilinos.sandia.gov>

*Sacado* = Trilinos AD package (templated)

- Forward AD = rewrite of FAD package of Di Césaré and Pironneau; uses *expression templates*.
- Reverse AD = RAD (written by dmg).
- Taylor poly's ( $n = 1$  fwd) by Eric Phipps.

# Sacado results in Charon





## `n1c` for Optimized Gradients

Seeing larger expression graphs gives more opportunity for optimizing the computation.

- ADIC optimizes per C statement, mixing forward and reverse, in overall forward evaluation.
- *n1c* program sees entire function evaluation in `.n1` file, emits C or Fortran avoiding needless ops.



## Timings on Protein-Folding Example

Eval style	sec/eval	rel.
Compiled C, no grad.	2.92e−5	1.0
Sacado RAD	1.90e−4	6.5
<i>nlc</i>	4.78e−5	1.6
ASL, fg mode	9.94e−5	3.4
ASL, pfgh mode	1.26e−4	4.3

Eval. times, protein folding ( $n = 66$ )



## Hessian-vector Products

Several approaches...

- RAD  $\circ$  FAD: `ADvar<SFad<double>,1> >`
- FAD  $\circ$  RAD: `SFad<ADvar<double>,1>`
- Custom mixture: `Rad2::ADvar<double>`
- AMPL/solver interface library: find,  
exploit partial separability automatically:

$$f(x) = \sum_i \theta_i \left( \sum_j f_{ij}(U_{ij}x) \right) .$$





## Hessian-vector timings

Eval style	sec/eval	rel.
RAD $\circ$ FAD	4.70e-4	18.6
FAD $\circ$ RAD	1.07e-3	42.3
ASL, Custom mixture	2.27e-4	9.0
ASL, pfgh mode	2.53e-5	1.0

Seconds per Hessian-vector prod

$$f = \frac{1}{2}x^T Q x, n = 100.$$



## Concluding Remarks

- $\exists$  many possibilities, each with advantages and disadvantages. Having several tools helps, especially for treating hot spots.
- C++ — like looking through a keyhole; Seeing more expression graph can help.
- AD can save human time.
- AD may give faster, more accurate computation.
- Room for more tools to optimize evals.



## Some Pointers

<http://www.autodiff.org>

<http://trilinos.sandia.gov>

<http://www.sandia.gov/~dmgay>