Automatic Differentiation in Computational Science

Boyana Norris

norris@mcs.anl.gov

http://www.mcs.anl.gov/~norris

Paul Hovland, Sri Hari Krishna Narayanan, Jean Utke

ASCAC Meeting, Nov. 10, 2010

Outline

- Why automatic differentiation?
- Application examples
- AD in a nutshell
- Research challenges and opportunities
- Summary

Why automatic differentiation?

- Given: some numerical model $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ implemented as a program
- Wanted: sensitivity analysis, optimization, parameter (state) estimation, higher order approximation, …

Why automatic differentiation? (cont.)

- Alternative #1: hand-coded derivatives
  - hand-coding is tedious and error-prone
  - coding time grows with program size and complexity
  - automatically generated code may be faster
  - no natural way to compute derivative matrix-vector products (Jv, J^Tv, Hv) without forming full matrix
  - maintenance is a problem (must maintain consistency)
- Alternative #2: finite difference approximations
  - introduce truncation error that in the best case halves the digits of accuracy
  - cost grows with number of independents
  - no natural way to compute J^Tv products

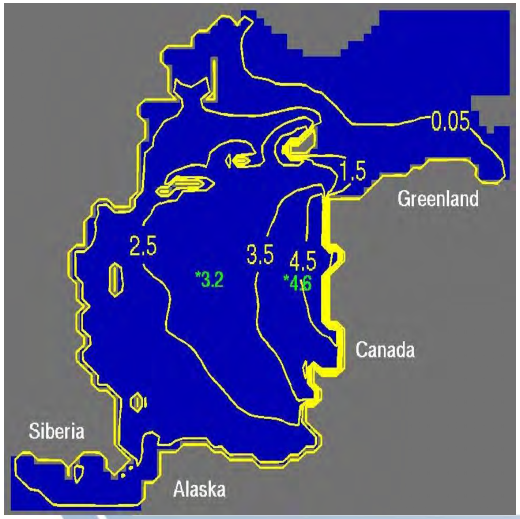
➔ use tools to do it at least semi-automatically!

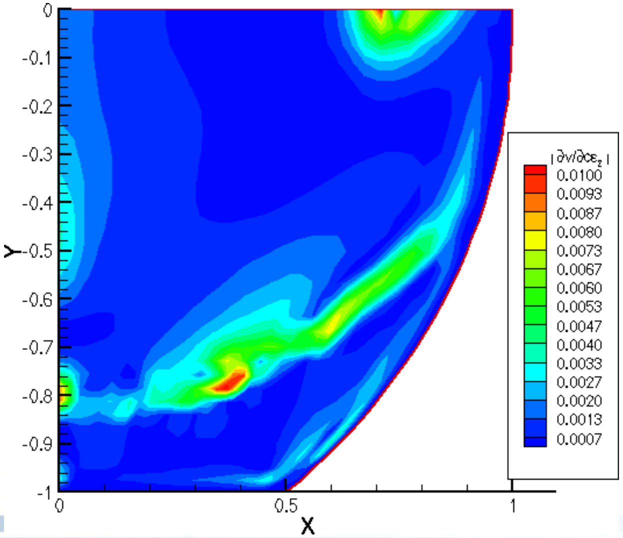
AD in computational science

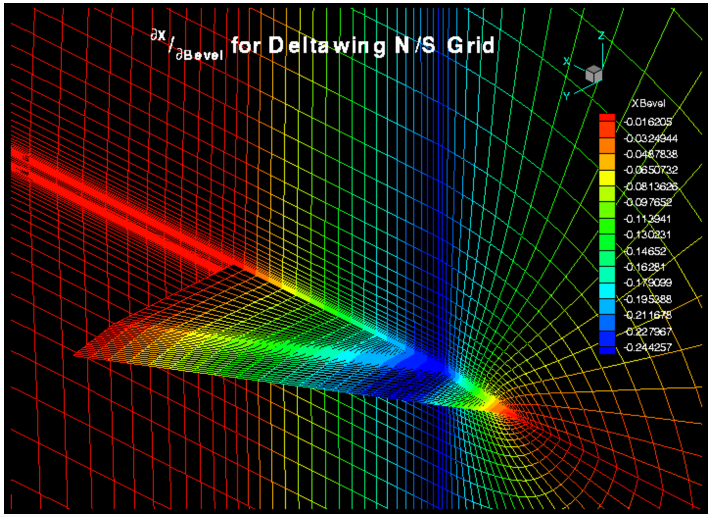
- AD is used in applications for computing
  - Gradients
  - Jacobian projections
  - Hessian projections
  - Higher-order derivatives (full or partial tensors, univariate tensor series)
- Derivatives are used for
  - Measuring the sensitivity of a simulation to unknown or poorly known parameters (e.g., *how does ocean bottom topography affect flow?*)
  - Assessing the role of algorithm parameters in a numerical solution (e.g., *how does the filter radius impact a large eddy simulation?*)
  - Computing a descent direction in numerical optimization (e.g., *compute gradients and Hessians for use in aircraft design*)
  - Solving discretized nonlinear PDEs (e.g., *compute Jacobians or Jacobian-vector products for combustion simulations*)

Application highlights

- ❑ Atmospheric chemistry
- ❑ Breast cancer biostatistical analysis
- ❑ CFD: CFL3D, NSC2KE, (Fluent 4.52: Aachen) ...
- ❑ Chemical kinetics
- ❑ Climate and weather: MITgcm, MM5, CICE
- ❑ Semiconductor device simulation
- ❑ Water reservoir simulation
- ❑ Mechanical engineering (design optimization)



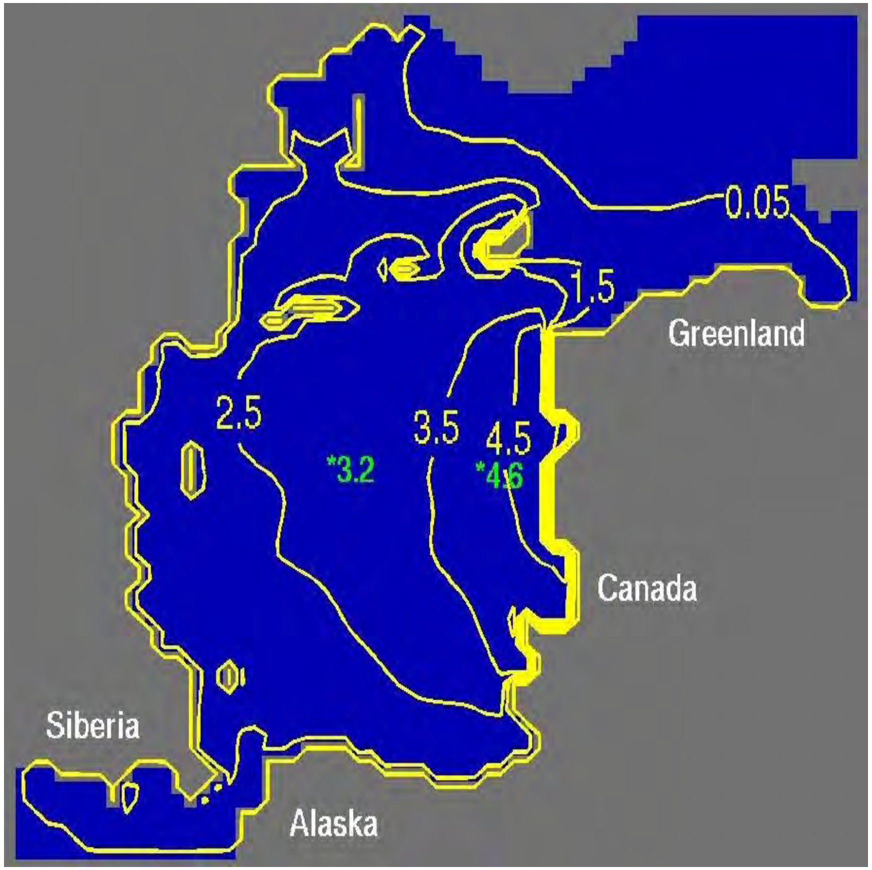




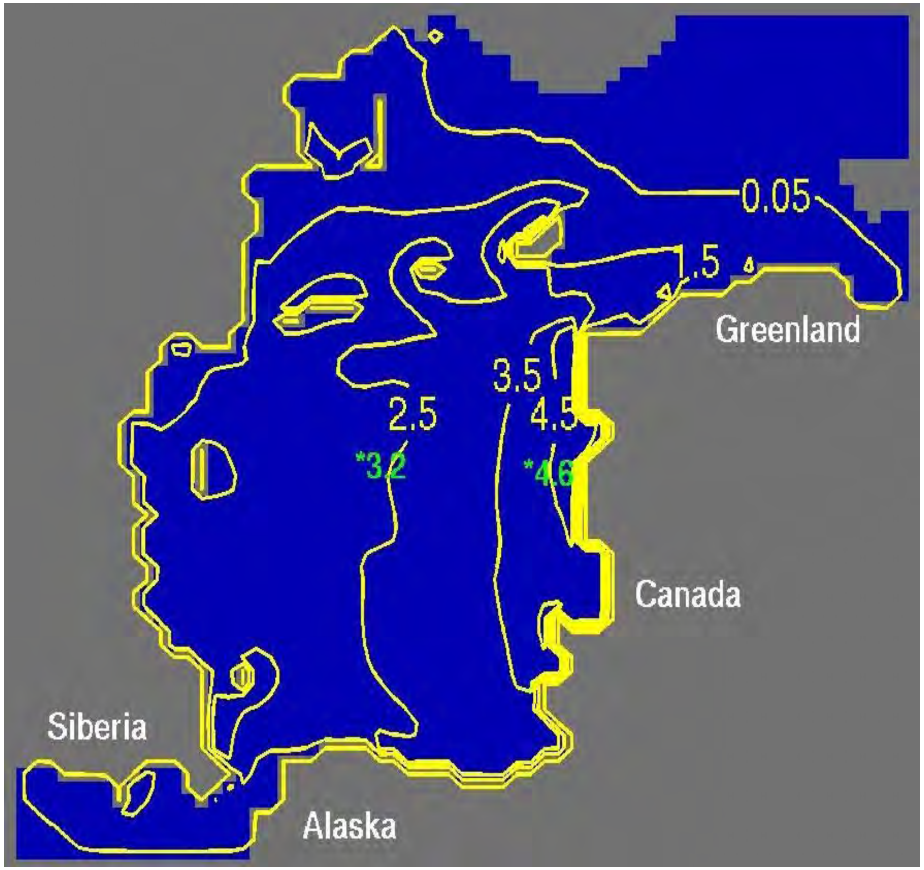
Parameter tuning: Sea ice model

- Simulated (yellow) and observed (green) March ice thickness (m)

Tuned parameters

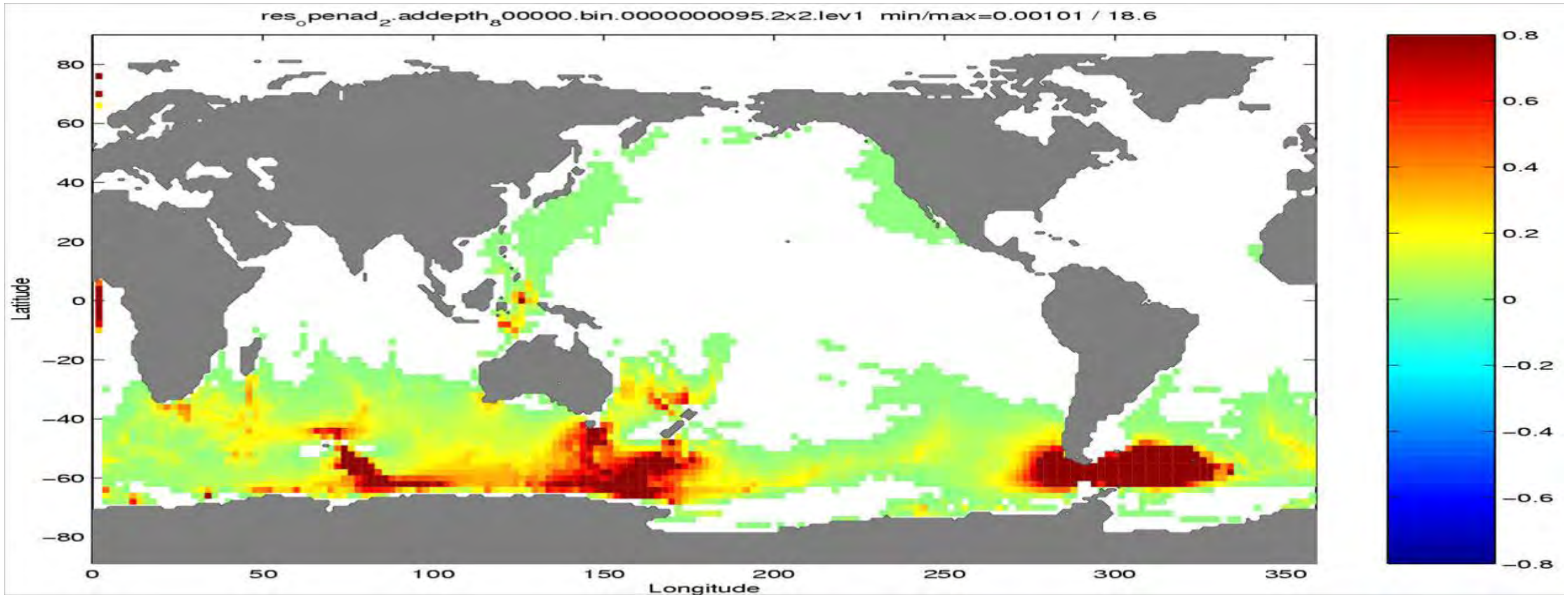


Standard parameters



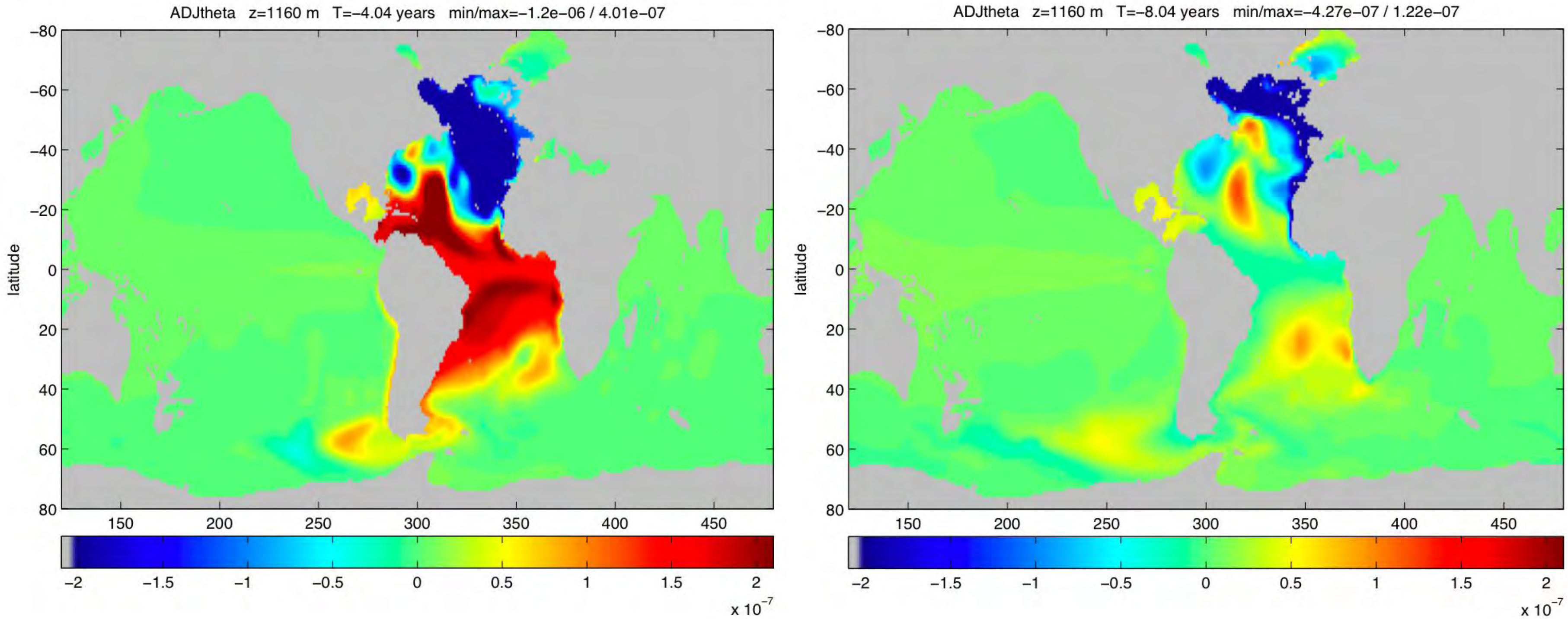
Application: Sensitivity analysis in simplified climate model

- Sensitivity of flow through Drake Passage to ocean bottom topography
  - Finite difference approximations: 23 days
  - Naïve automatic differentiation: 2 hours 23 minutes
  - Smart automatic differentiation: 22 minutes



Application: Preliminary results for MITgcm

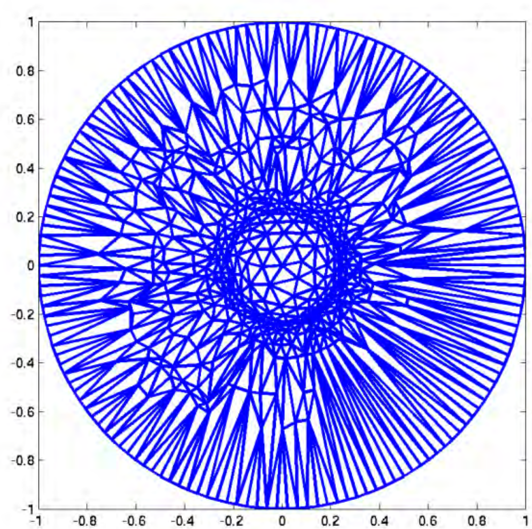
- Time for one simulation run (20 years at 4 degree resolution): 51.75 hrs
- Time for one gradient computation using AD: 204.2 hrs (8.5 days)



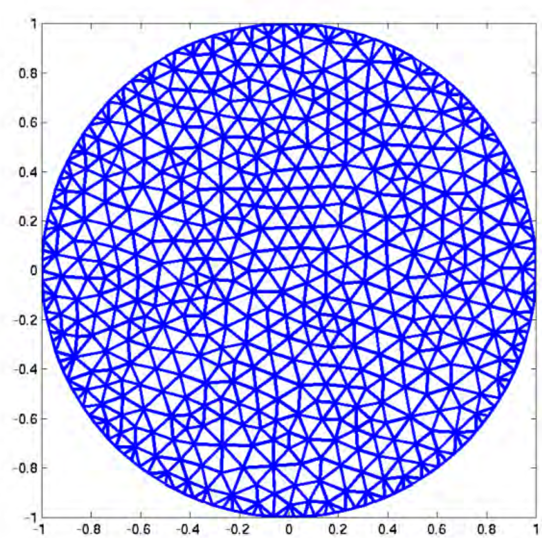
Application: mesh quality optimization

- Optimization used to move mesh vertices to create elements as close to equilateral triangles/tetrahedra as possible
- Semi-automatic differentiation is 10-25% faster than hand-coding for gradient and 5-10% faster than hand-coding for Hessian
- Automatic differentiation is a factor 2-5 times faster than finite differences

Before

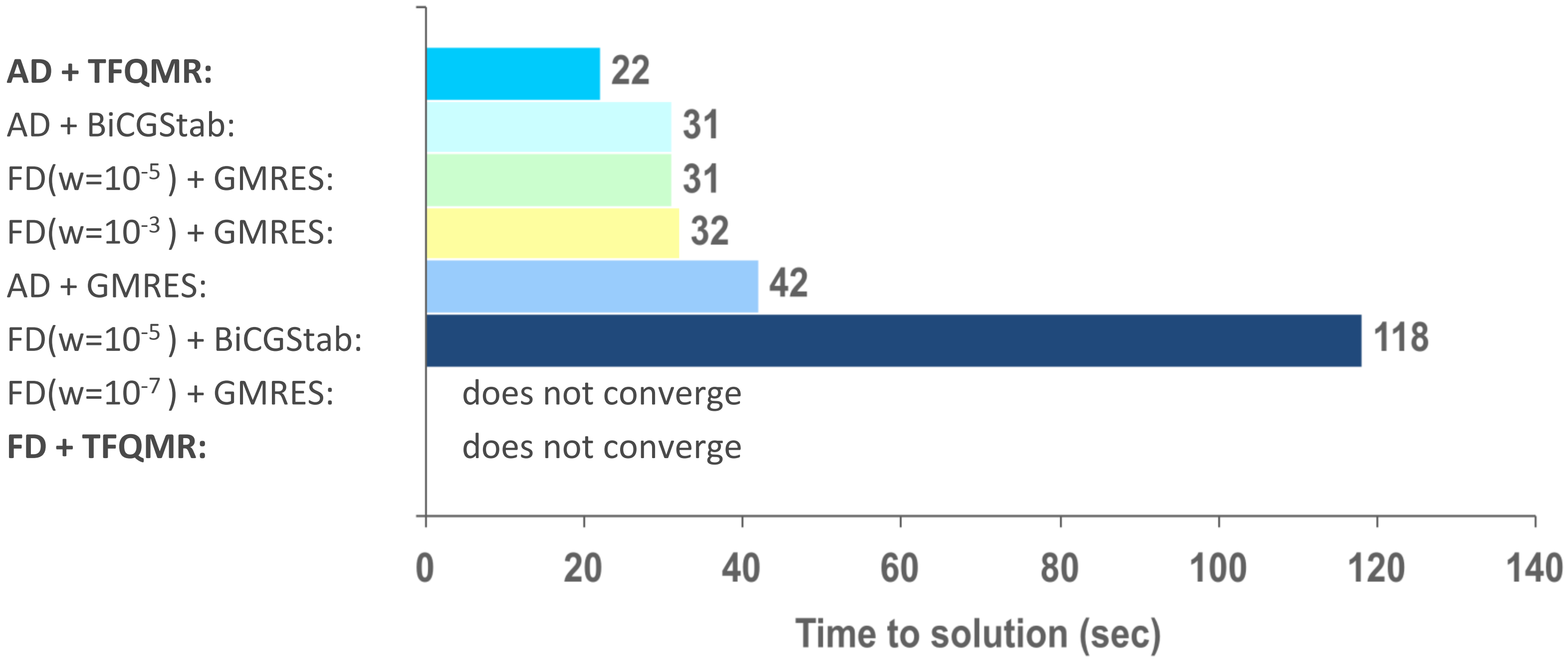


After



Application: solution of nonlinear PDEs

- Jacobian-free Newton-Krylov solution of model problem (driven cavity)



AD = automatic differentiation

FD = finite differences

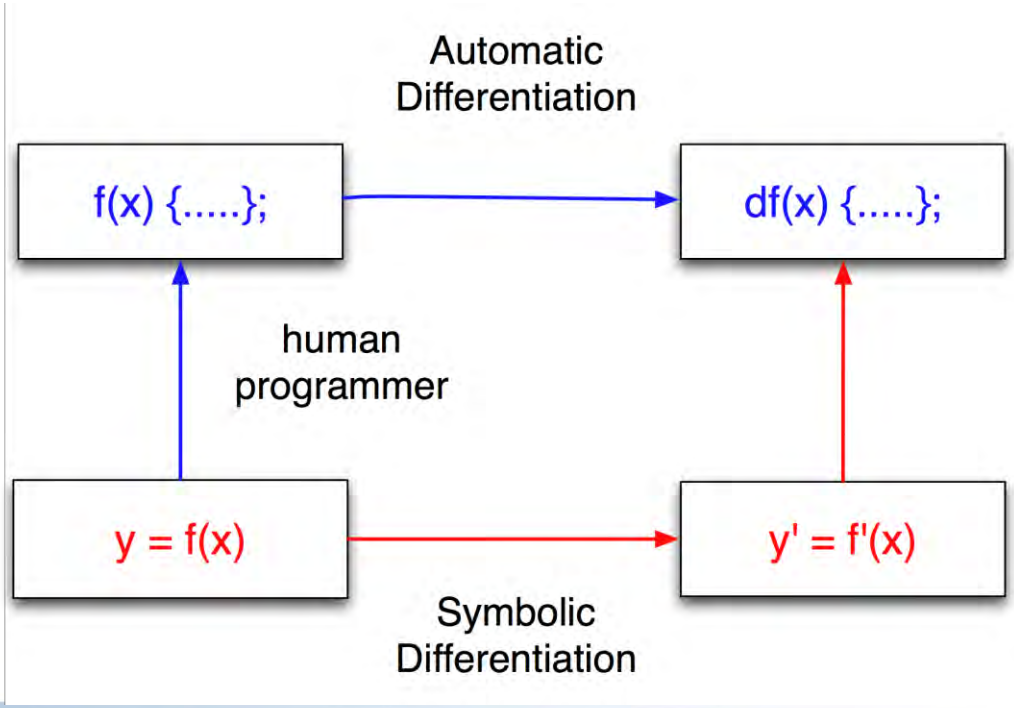
W = noise estimate for Brown-Saad

Automatic differentiation (AD) in a nutshell

- Technique for computing analytic derivatives of programs
- Derivatives are used in a many numerical algorithms, including nonlinear equation solvers, optimization algorithms, and uncertainty quantification
- AD = analytic differentiation of elementary functions + propagation by chain rule
  - Every programming language provides a limited number of elementary mathematical functions
  - Thus, every function computed by a program may be viewed as the composition of these so-called intrinsic functions
  - Derivatives for the intrinsic functions are known and can be combined using the chain rule of differential calculus

AD in a nutshell (cont.)

- Associativity of the chain rule leads to two main modes: forward and reverse
- Can be implemented using source transformation or operator overloading



Modes of AD

- Forward Mode
  - Propagates derivative vectors, often denoted ∇u or g_u
  - Derivative vector ∇u contains derivatives of u with respect to independent variables
  - Time and storage proportional to vector length (# indeps)
- Reverse Mode
  - Propagates adjoints, denoted $\bar{u}$ or u_bar
  - Adjoint $\bar{u}$ contains derivatives of dependent variables with respect to u
  - Propagation starts with dependent variables—must reverse flow of computation
  - Time proportional to adjoint vector length (# dependents)
  - Storage proportional to number of operations
  - Because of this limitation, often applied to subprograms

16

Accumulating derivatives

- Represent function using a directed acyclic graph (DAG)
- Computational graph
  - Vertices are intermediate variables, annotated with function/operator
  - Edges are unweighted
- Linearized computational graph
  - Edge weights are partial derivatives
  - Vertex labels are not needed
- Compute sum of weights over all paths from independent to dependent variable(s), where the path weight is the product of the weights of all edges along the path [Baur & Strassen]
- Find an order in which to compute path weights that minimizes cost (e.g., FLOPS): identify common subpaths (=common subexpressions in Jacobian)

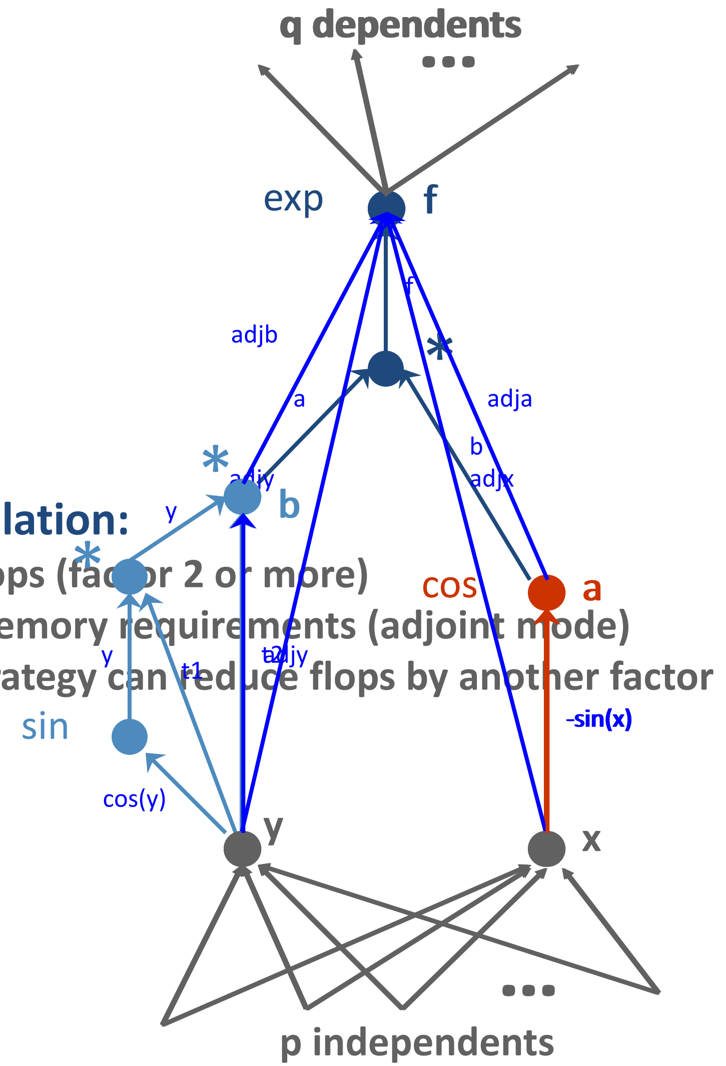
A small example

... lots of code...
a = cos(x)
b = sin(y)*y*y
f = exp(a*b)
... lots of code...

ADIC forward mode / New algorithm 17+5p

```
a = cos(x)
  dadx = -sin(x)
  g_a(1:p) = dadx*g_x(1:p)
tmp1 = sin(y)
  d1dy = cos(y)
tmp2 = tmp1*y
  g_1(1:p) = d1dy*g_y(1:p)
b = tmp2*y
  g_2(1:p) = y*g_1(1:p)+tmp1*g_y(1:p)
  adjx = f*b*dadx
  adjy = f*a*(tmp2 + y*(tmp1 + d1dy*y))
  g_b(1:p) = y*g_2(1:p)+tmp2*g_y(1:p)
f = exp(a*b)
  g_f(1:p) = adjx*g_x(1:p)+adjy*g_y(1:p)
tmp1 = a*b
  adja = f*b
  g_1(1:p) = b*g_a(1:p)+a*g_b(1:p)
  adjb = f*a
f = exp(tmp1)
  g_f(1:p) = adja*g_a(1:p)+adjb*g_b(1:p)
  g_f(1:p) = f*g_1(1:p)
```

Preaccumulation:
- **Reduces flops (factor 2 or more)**
- **Reduces memory requirements (adjoint mode)**
- **Optimal strategy can reduce flops by another factor of 2**



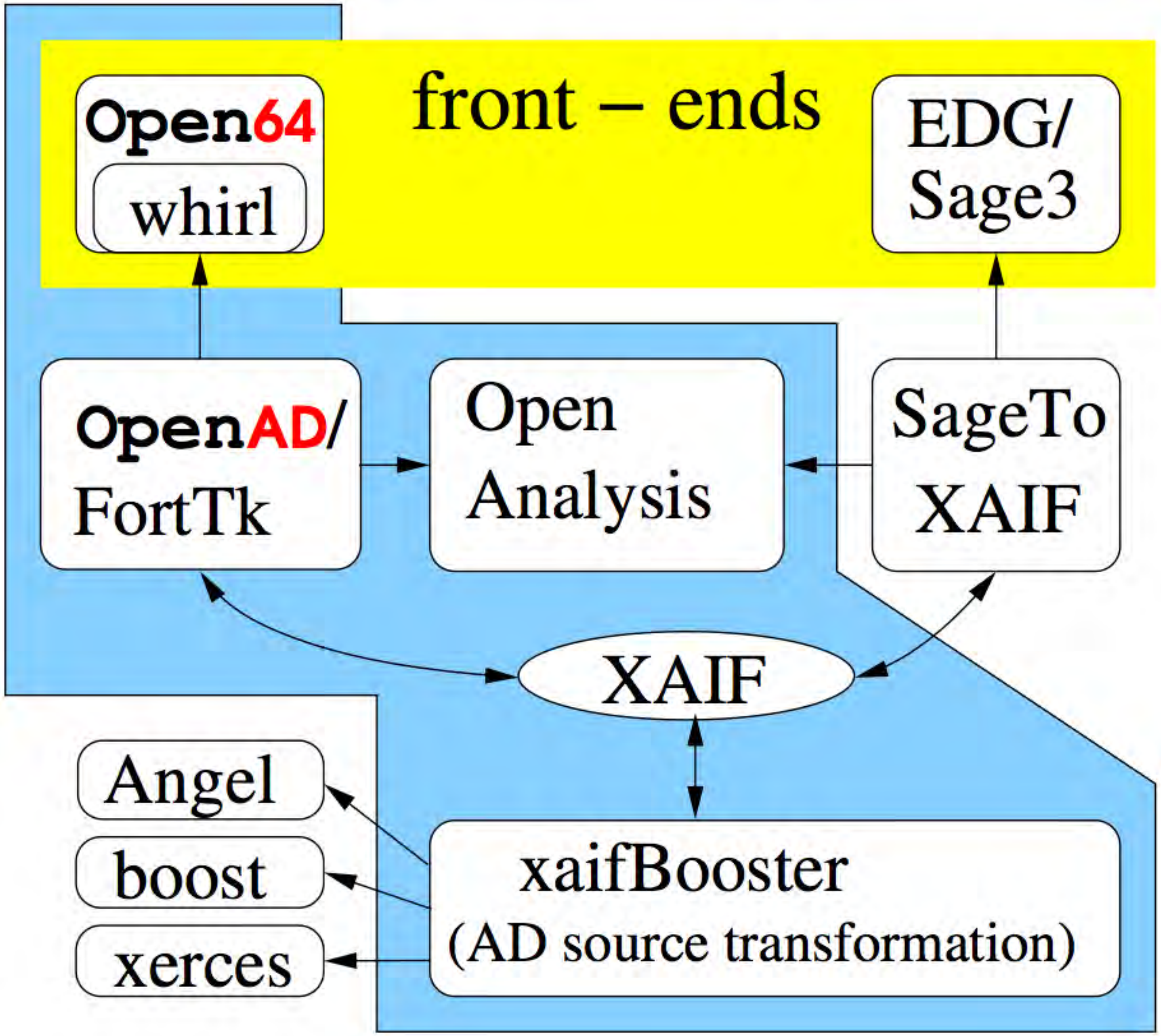
Automatic generation of derivative code

- Automatic differentiation (AD) tools automate the creation of derivative code
- Automatic generation of derivative code from function code offers several benefits relative to hand-coded derivatives
  - Higher productivity
  - Improved quality: hand-coding is tedious and hence error-prone
  - Higher performance: tools explore combinatorial search space
  - Improved software maintenance: easier to maintain consistency
- AD tools require:
  - Robust compiler infrastructure (Open64/SL, ROSE)
  - Traditional and domain-specific compiler analyses (OpenAnalysis)
  - Combinatorial algorithms to identify effective strategies for combining partial derivatives (XAIFBooster – CSCAPES)

Argonne-developed AD tools

- OpenAD/F (Argonne/UChicago/Rice)
  - Support for many Fortran 95 features
  - Developed by a team with expertise in combinatorial algorithms, compilers, software engineering, and numerical analysis
  - Forward and reverse; source transformation
- ADIC (Argonne/UChicago)
  - Support for all of C, some C++
  - Source transformation; forward and reverse mode
  - New version (2.0) based on industrial strength compiler infrastructure
  - Shares some infrastructure with OpenAD/F
- ADIFOR (Rice/Argonne)
  - Mature and very robust tool
  - Support for all of Fortran 77
  - Forward and (adequate) reverse modes

OpenAD system architecture



http://www.mcs.anl.gov/OpenAD

Impact*

- ADIFOR
  - Cited in 232 journal articles (ISI)
  - Cited in ~750 online articles (Google Scholar)
  - 678 registered users (does not include users registered at Rice)
  - 484 subscribers to adifor-users mailing list
- ADIC
  - Cited in 53 journal articles
  - Cited in ~160 online articles
  - 861 registered users
  - 564 subscribers to adic-users mailing list
- Direct collaboration with several applications groups; funded collaborations with:
  - MIT: Ocean Modeling and State Estimation
  - NASA Langley: Multidisciplinary Design Optimization
  - PNNL: Atmospheric Chemistry

*2008 data

Research challenges and opportunities

- Producing more efficient derivative computations
  - Identifying and exploiting structure (e.g., sparsity, low rank)
  - Numerical algorithms that exploit cheap derivative quantities, e.g., Jacobian-vector and vector-Jacobian products, univariate Taylor series coefficients, etc.
  - Elimination strategies
  - Compiler analysis
    - Context sensitive, flow sensitive analysis
    - Linearity analysis
    - Parallel, object-oriented programs

Research challenges and opportunities (cont.)

- Mathematical challenges
- Language feature coverage (e.g., C++ templates)
- Multi-language applications
- Different parallel programming models
  - MPI, OpenMP, hybrid, GPGPU, etc.
- Exploiting parallelism in derivative computation
- Efficient checkpointing strategies
- Derivative propagation
  - Hardware acceleration (cell processor, GeForce 8800GTX)
  - Sparse linear combinations (SparsLinC)

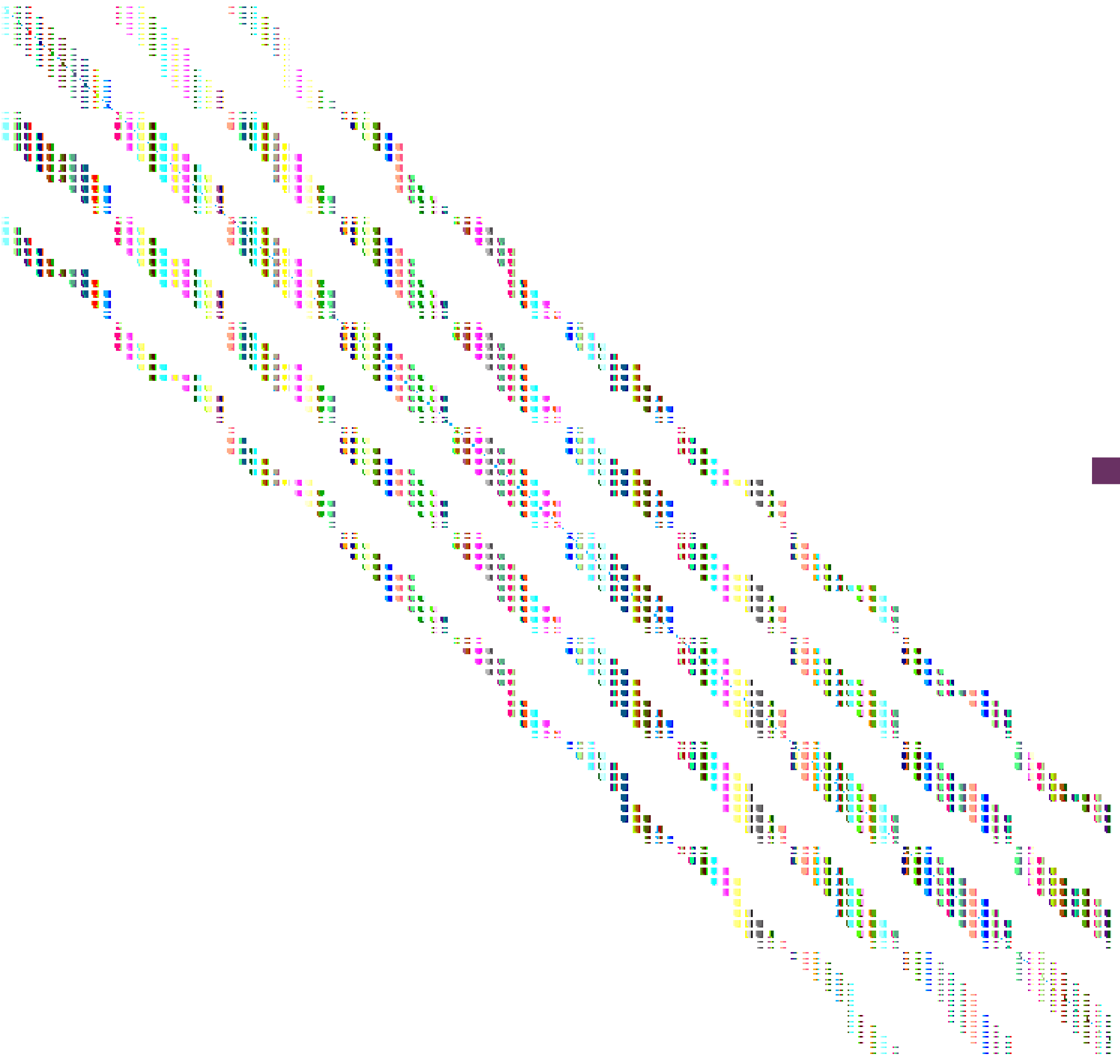
Research examples

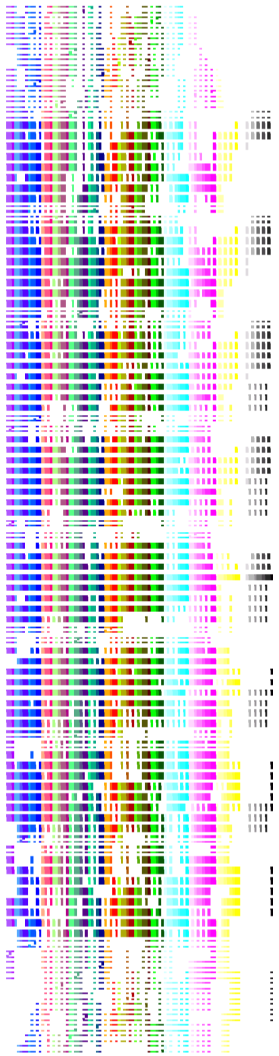
- Exploiting scarcity reduces both the number of flops to preaccumulate local partials and the number of flops to propagate global derivatives
  - Scarcity: the Jacobian $\boldsymbol{J}$ for a given function $f: \Re^n$ a $\Re^m$ may have fewer than $n * m$ degrees of freedom. A scarse $\boldsymbol{J}$ can be represented by a graph with a minimal edge count.
- Matrix coloring for problems with nested sparsity structure can reduce the cost of Jacobian computations for nonlinear PDEs discretized on regular grids (e.g., PETSc DA or DMMG)
- Polynomial-time algorithms for detecting structural properties (e.g., symmetry) of DAGs
- Fully automated derivatives when standard interfaces are available (e.g., NEOS, PETSc)

Matrix Coloring

- Jacobian matrices are often sparse
- The forward mode of AD computes $J \times S$, where S is usually an identity matrix or a vector
- Can "compress" Jacobian by choosing S such that structurally orthogonal columns are combined
- A set of columns are structurally orthogonal if no two of them have nonzeros in the same row
- Equivalent problem: color the graph whose adjacency matrix is $J^T J$
- Equivalent problem: distance-2 color the bipartite graph of J

Compressed Jacobian





Example of a challenge: Chain rule (non-)differentiability

```
if (x .eq. 1.0) then
   a = y
else if ((x .eq. 0.0) then
   a = 0
else
   a = x*y
endif

b = sqrt(x**4 + y**4)
```

Mathematical Challenges

- Derivatives of intrinsic functions at points of non-differentiability
- Derivatives of implicitly defined functions
- Derivatives of functions computed using numerical methods

Points of nondifferentiability

- Due to intrinsic functions
  - Several intrinsic functions are defined at points where their derivatives are not, e.g.:
    - abs(x), sqrt(x) at x=0
    - max(x,y) at x=y
  - Requirements:
    - Record/report exceptions
    - Optionally, continue computation using some generalized gradient
  - ADIFOR/ADIC approach
    - User-selected reporting mechanism
    - User-defined generalized gradients, e.g.:
      - [1.0,0.0] for max(x,0)
      - [0.5,0.5] for max(x,y)
    - Various ways of handling
      - Verbose reports (file, line, type of exception)
      - Terse summary (like IEEE flags)
      - Ignore
- Due to conditional branches
  - May be able to handle using trust regions

Implicitly Defined Functions

- Implicitly defined functions often computed using iterative methods
- Function and derivatives may converge at different rates
- Derivative may not be "accurate" if iteration halted upon function convergence
- Solutions:
  - Tighten function convergence criteria
  - Add derivative convergence to stopping criteria
  - Compute derivatives directly, e.g. A □x = □b

Derivatives of Functions Computed Using Numerical Methods

- Differentiation and approximation may not commute
- Need to be careful about how derivatives of numerical approximations are used
- For example, differentiating through an ODE integrator can provide unexpected results due to feedback induced by adaptive stepsize control:

$$\nabla z^1 = \frac{\partial z^1}{\partial t^1} \nabla t^1 + \frac{\partial z^1}{\partial p}$$

Addressing limitations in black box AD

- Detect points of nondifferentiability
  - proceed with a subgradient
  - currently supported for intrinsic functions, but not conditional statements
- Exploit mathematics to avoid differentiating through an adaptive algorithm
- Modify termination criterion for implicitly defined functions
  - Tighten tolerance
  - Add derivatives to termination test (preferred)

Automatic differentiation and parallelism

- Data-flow analysis framework must become MPI-aware: requires identifying potential send-receive pairs
- Reverse mode dramatically reduces derivative cost for scalar functions
  (1 cpu-week versus 1 million cpu-years for a climate model)
  but requires control and data flow reversal relative to function evaluation
  - In message-passing codes, send becomes receive and receive becomes send; situation significantly more complicated in case of nonblocking communication (EuroPVM/MPI2008, PDSEC2009)
  - Requirement to restore state in reverse order leads to full state and incremental checkpointing strategies; restarts can be done in parallel
- New prefix-like algorithms for derivatives of parallel reduction operations

Exascale challenges

- AD is a *semantic* transformation and the resulting code may exhibit different concurrency characteristics than the original computation
- Differentiation of some existing (e.g., PGAS) and future programming models
- Checkpointing for reverse mode

Summary

- Automatic differentiation provides a (semi-)automated way for generating accurate derivatives
- AD research spans multiple areas: applied mathematics, combinatorial algorithms, compilers
- AD algorithms and tools must keep pace with
  - Increasingly complex applications
  - Evolving hardware, increasing levels of parallelism
  - Changing programming models and languages

For More Information

- Andreas Griewank and Andrea Walther, *Evaluating Derivatives*, 2nd edition, SIAM, 2008.
- Griewank, "On Automatic Differentiation"; this and other technical reports available online at: http://www.mcs.anl.gov/autodiff/tech_reports.html
- AD in general: http://www.mcs.anl.gov/autodiff/, http://www.autodiff.org/
- ADIFOR: http://www.mcs.anl.gov/adifor/
- ADIC: http://www.mcs.anl.gov/adic/
- OpenAD: http://www.mcs.anl.gov/openad/
- Other tools: http://www.autodiff.org/