Computational Foundations of Automatic Differentiation

Paul D. Hovland

Mathematics & Computer Science Division

Argonne National Laboratory

The Automatic Differentiation Team @ Argonne

- Paul Hovland
- Boyana Norris
- Jean Utke (UChicago)
- Sri Hari Krishna Narayanan
- Ilya Safro
- Heather Cole-Mullen (UChicago)
- Azamat Mametjanov




- Alumni: J. Abate, C. Bischof, S. Bhowmick, A. Griewank, P. Khademi, J. Kim, P. Malusare, U. Naumann, L. Roh, J. Shin, M. Strout, B. Winnicka



- Collaborators: P. Heimbach, L. Hascoët

- Funding: US DOE, NSF, NASA

Outline

- Introduction to automatic differentiation (AD)
- Some combinatorial (graph) problems in AD
- Implementation of source transformation AD tools
  - Compiler infrastructure
  - [Infrastructure-independent analysis]
    - Domain-specific dataflow analysis
  - [Language-independent transformation modules]
- Parallelism
- Mathematical challenges

AD in a Nutshell

- Technique for computing analytic derivatives of programs (millions of loc)
- Derivatives used in optimization, nonlinear PDEs, sensitivity analysis, inverse problems, uncertainty quantification, etc.
- 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
- Less work than hand coding, more accurate than finite diffs

Simple Function with Forward Mode Derivatives (differentiated using Tapenade)

```
DO i=3*n*m,1000
   a = a - 2*B(i)
   IF (a.gt.0) THEN
      c = c*a
   ENDIF
ENDDO
```

```
DO i=3*n*m,1000
   ad = ad - 2*Bd(i)
   a = a - 2*B(i)
   IF (a.gt.0) THEN
      cd = cd*a + c*ad
      c = c*a
   ENDIF
ENDDO
```

Reverse/Adjoint Mode Gradient (Tapenade)

Forward Sweep

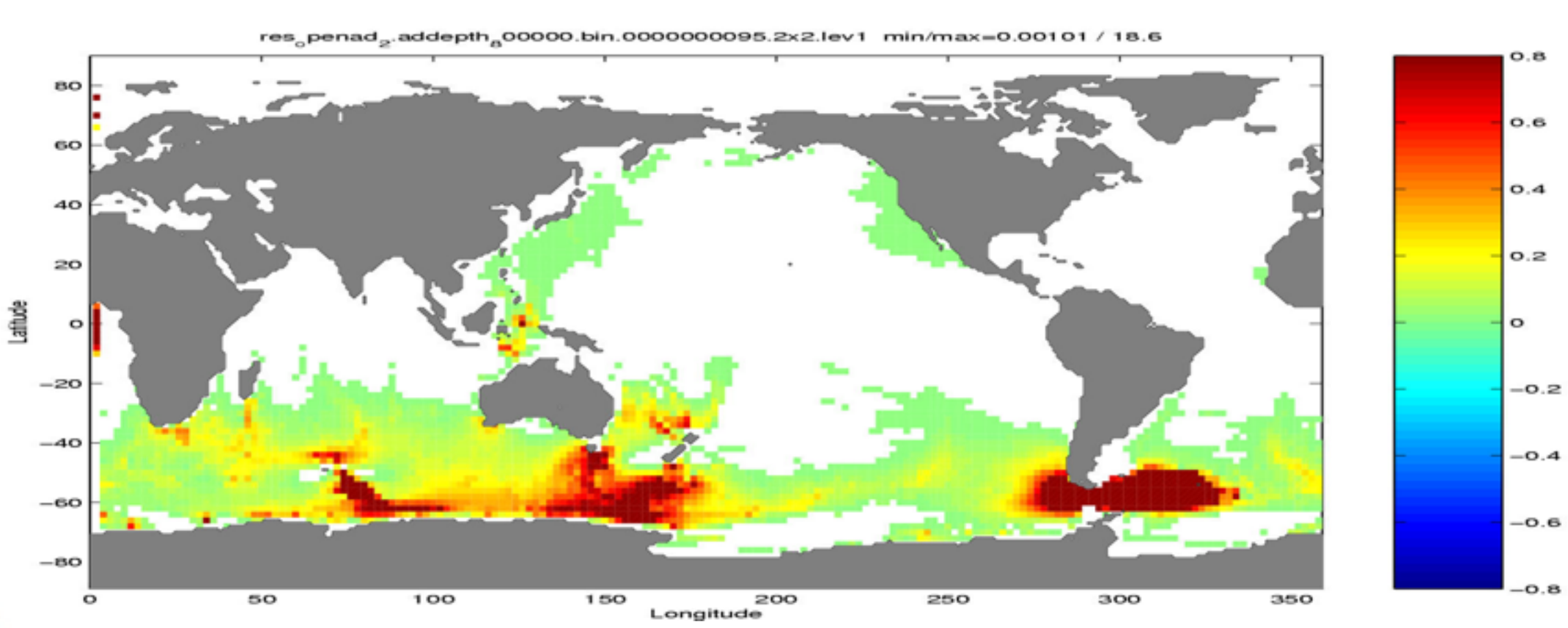
```
tmpF = 3*n*m
DO i=tmpF,1000
   CALL PUSHREAL4(a)
   a = a - 2*B(i)
   IF (a.gt.0) THEN
      CALL PUSHREAL4(c)
      c = c*a
      CALL PUSHCONTROL(0)
   ELSE
      CALL PUSHCONTROL(1)
   ENDIF
ENDDO
CALL PUSHINTEGER4(tmpF)
```

Adjoint Sweep

```
CALL POPINTEGER4(tmpF)
DO i=1000,tmpF,-1
   CALL POPCONTROL(branch)
   IF (branch.eq.0) THEN
      CALL POPREAL4(c)
      ab = ab + c*cb
      cb = a*cb
   ENDIF
   CALL POPREAL4(a)
   Bb(i) = Bb(i) - 2*ab
ENDDO
```

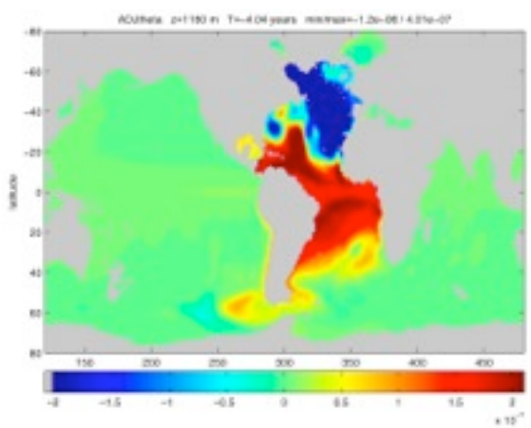
Application: Sensitivity analysis in simplified climate model

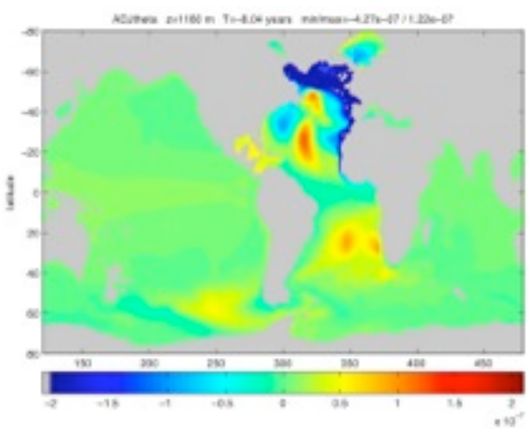
- Sensitivity of flow through Drake Passage to ocean bottom topography
  - Finite difference approximations: 23 days
  - Naïve automatic differentiation: 2 hours 23 minutes
  - Smart automatic differentiation: 22 minutes

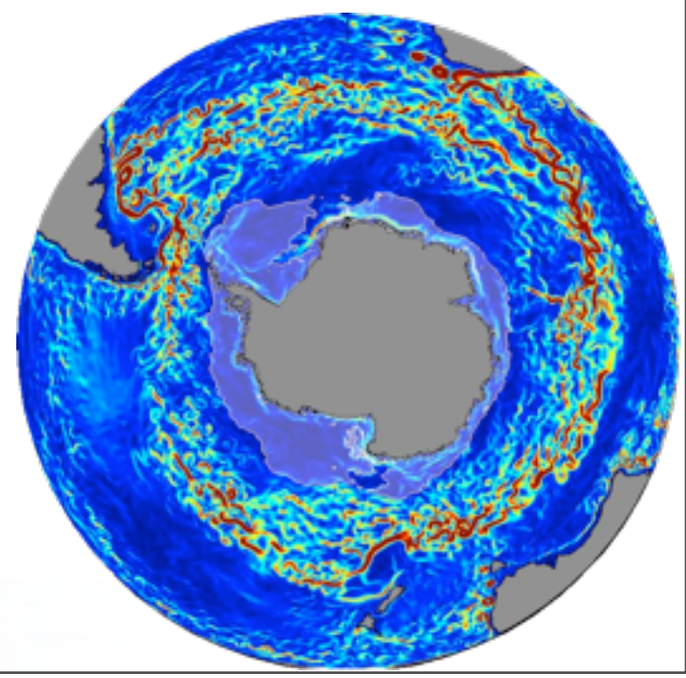


Application: Sensitivities for State Estimation

- Facilitated through the use of reverse mode automatic differentiation tools
- Provide the full gradient vector: first derivatives with respect to potentially millions or billions of independent variables
- Moderate cost : a small multiple of the forward model alone), independent of the number of independent variables
- Example: MITgcm
  - One simulation run (20 yrs at 4º): 52 cpu-hours
  - Gradient using AD: 204 hrs (8.5 cpu-days)
  - Finite-difference gradient approximation: 1.1 million cpu-years
  - Goal: O(10)-O(100) gradient evaluations at 1/2º

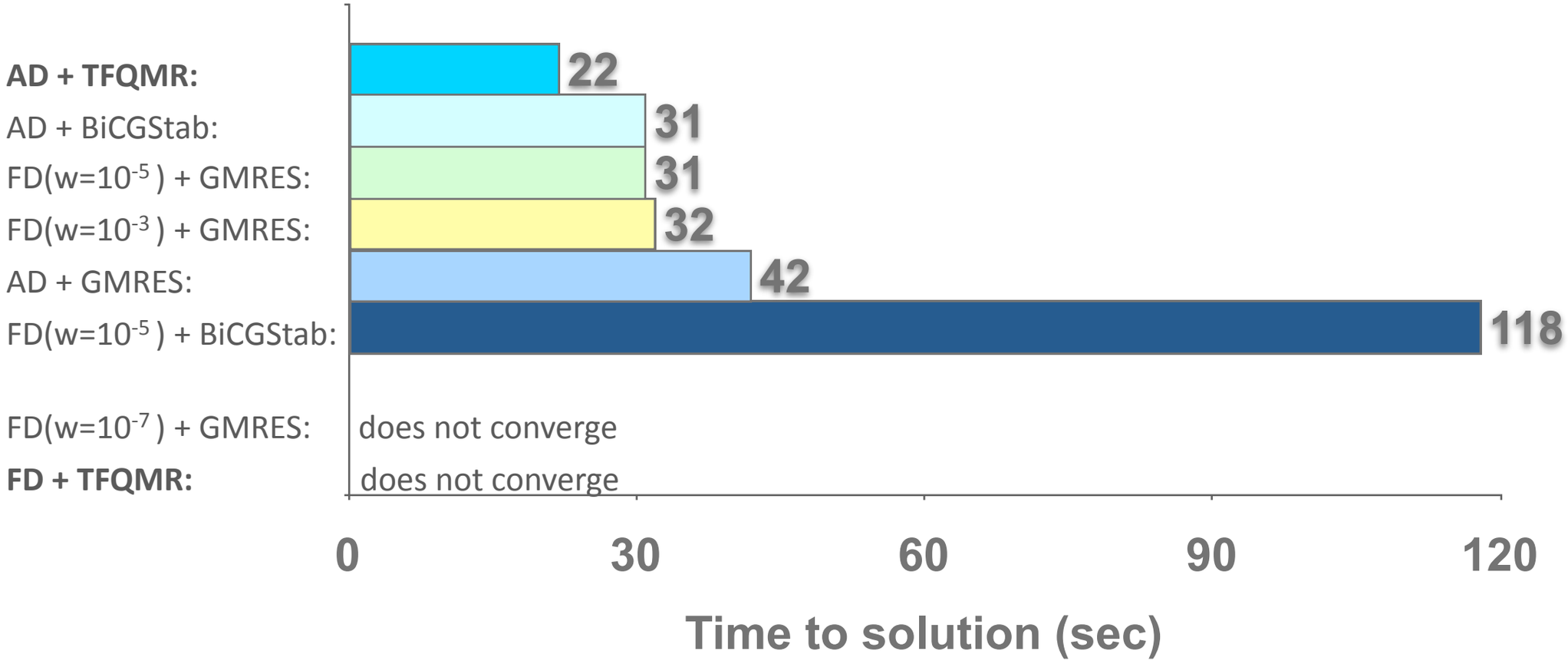






Application: solution of nonlinear PDEs

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



AD = automatic differentiation

FD = finite differences

W = noise estimate for Brown-Saad

Combinatorial problems in AD

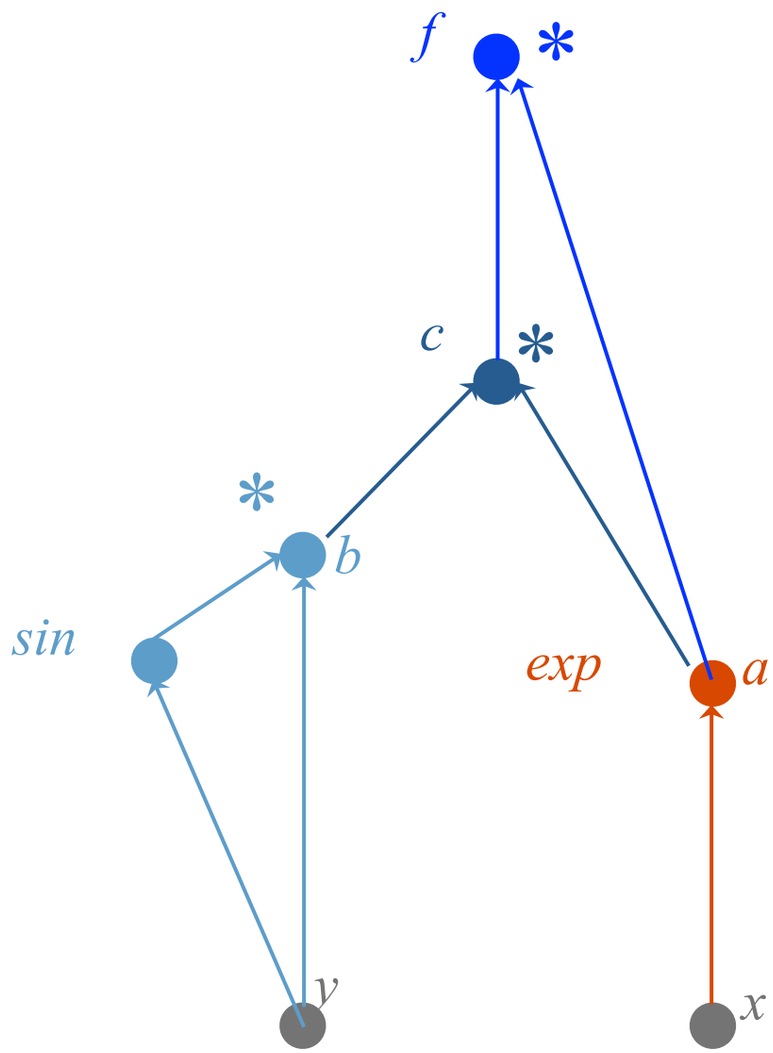
- Derivative accumulation
- Minimal representation

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 (flops): identify common subpaths (=common subexpressions in Jacobian)

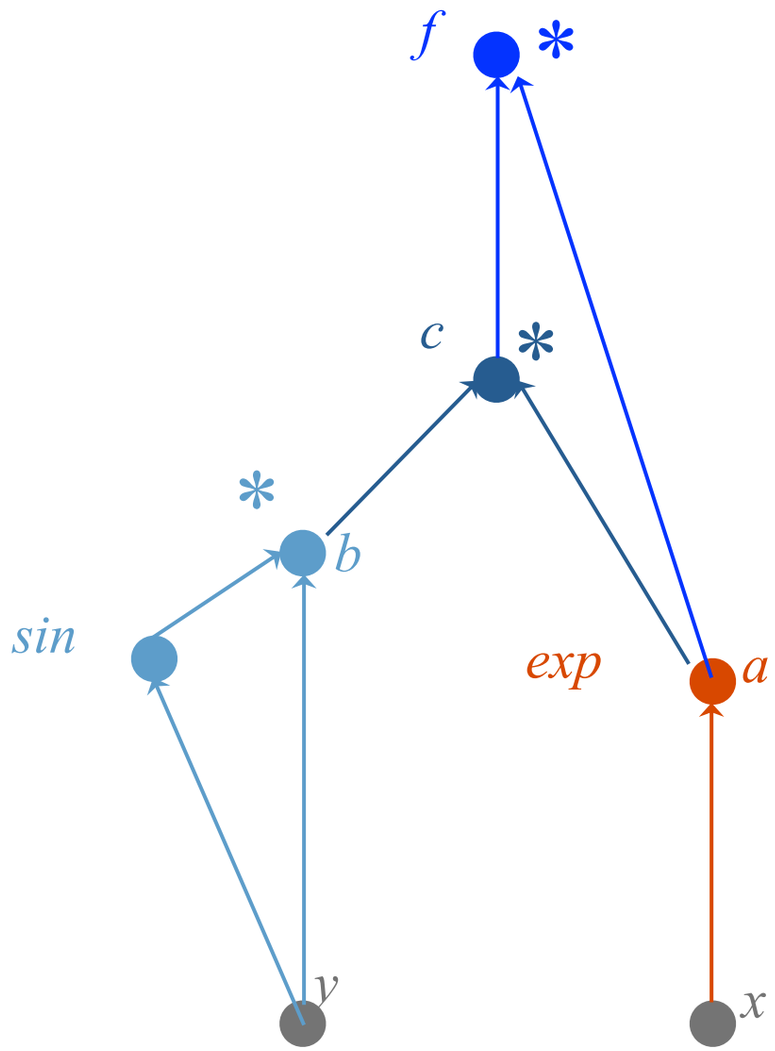
A simple example

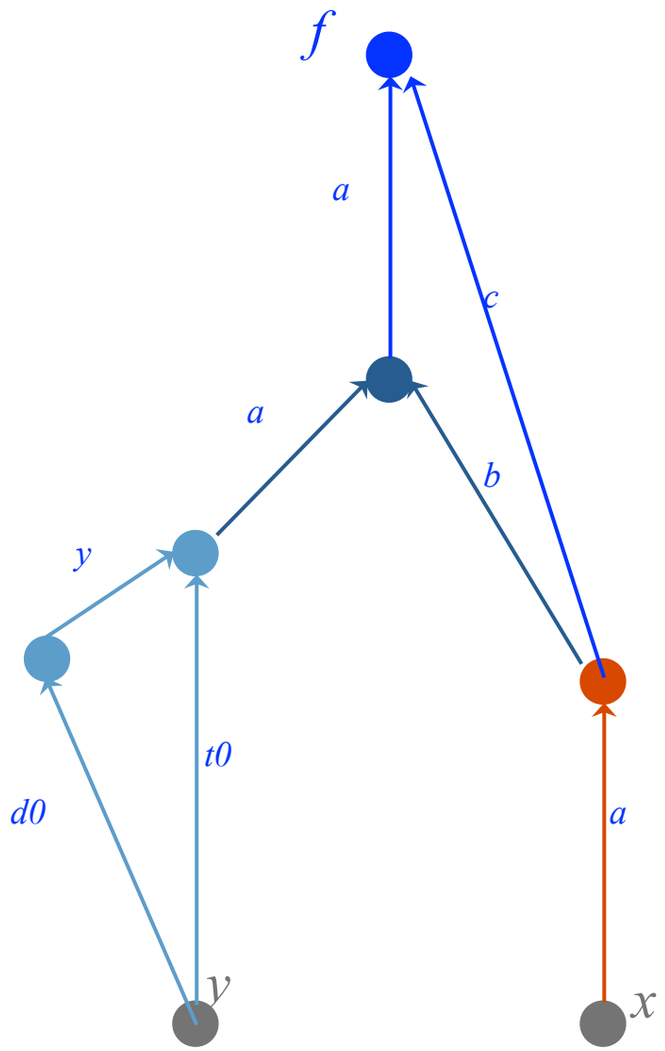
$b = sin(y)*y$
$a = exp(x)$
$c = a*b$
$f = a*c$



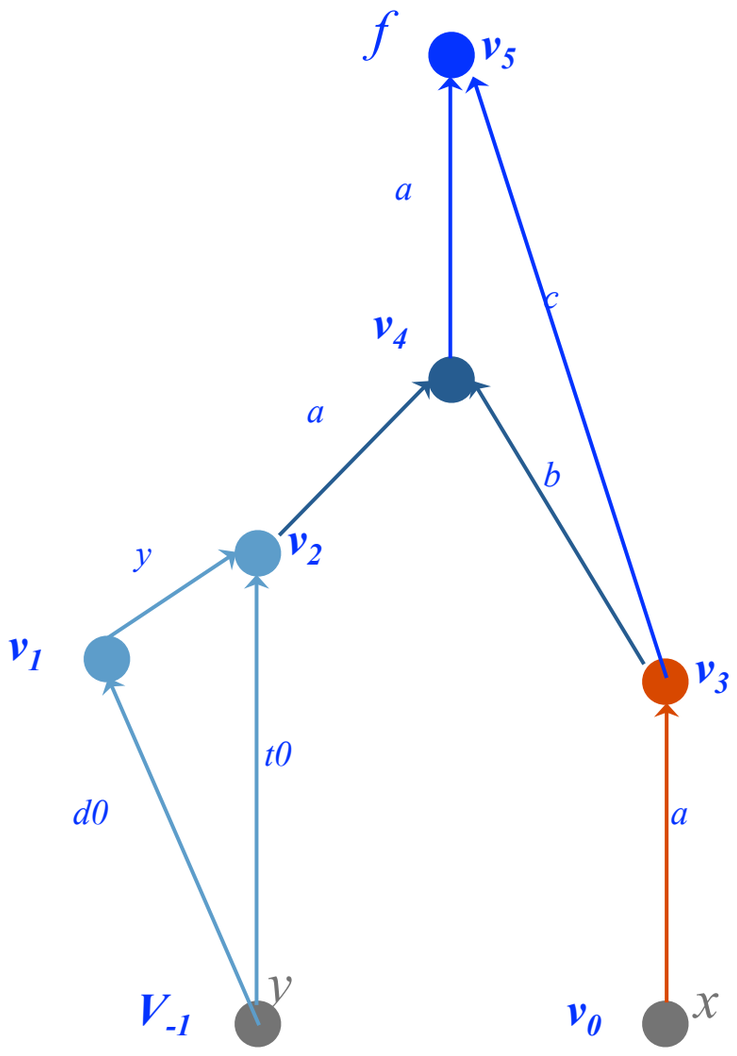
A simple example

$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$





Brute force

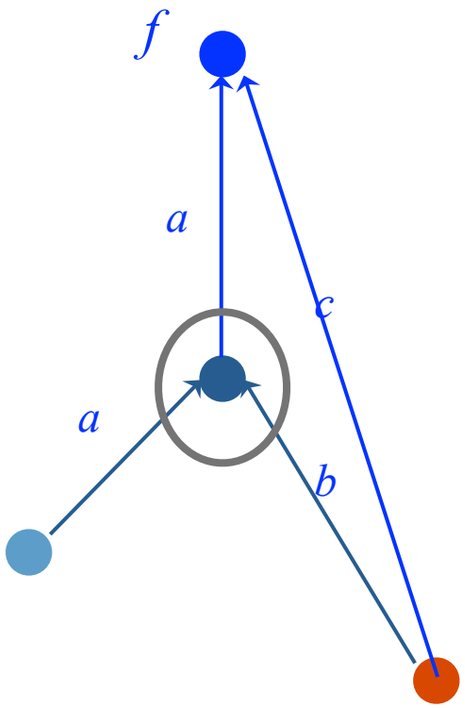


- Compute products of edge weights along all paths
- Sum all paths from same source to same target
- Hope the compiler does a good job recognizing common subexpressions

*dfdy = d0*y*a*a + t0*a*a*
*dfdx = a*b*a + a*c*

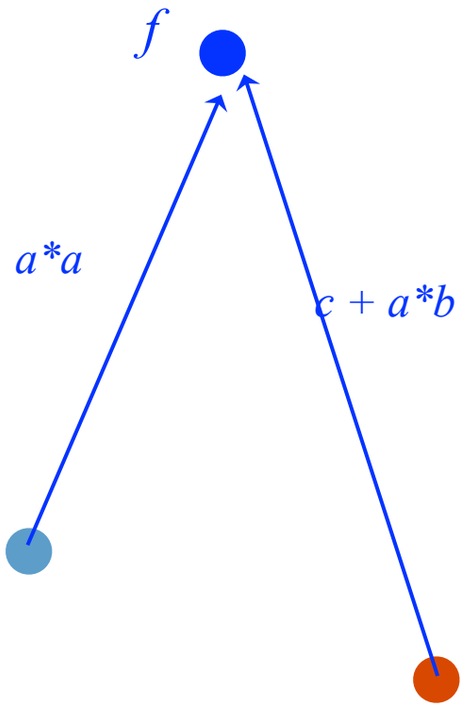
8 mults 2 adds

Vertex elimination



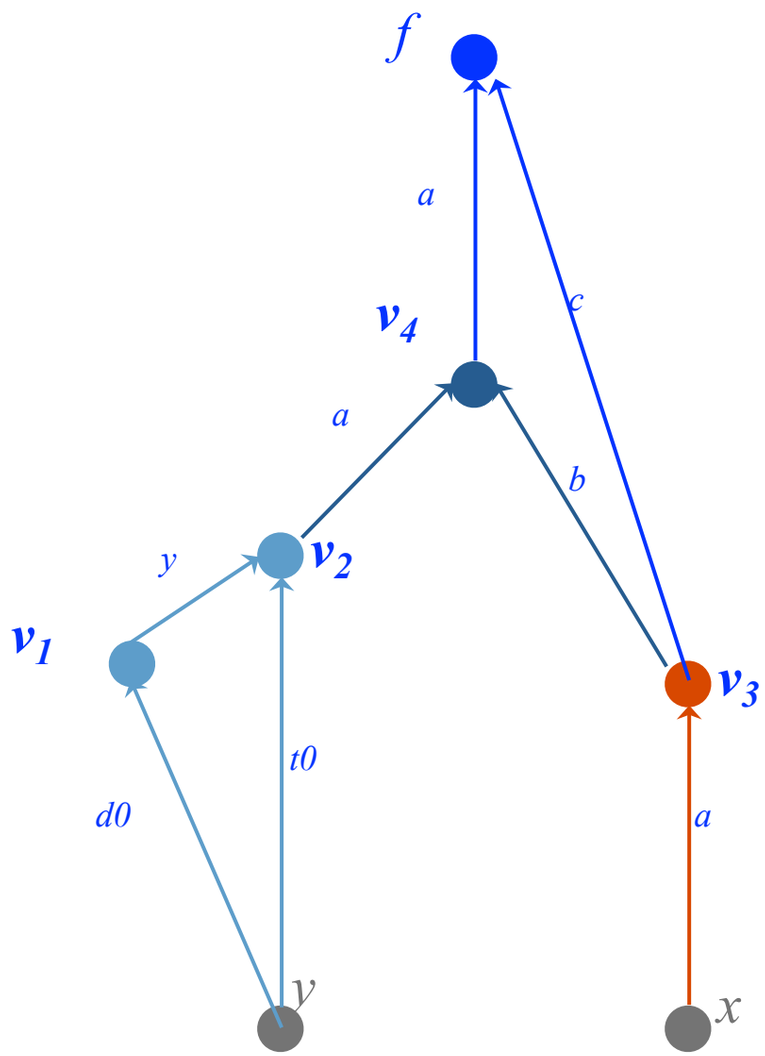
- Multiply each in edge by each out edge, add the product to the edge from the predecessor to the successor
- Conserves path weights
- This procedure always terminates
- The terminal form is a bipartite graph

Vertex elimination



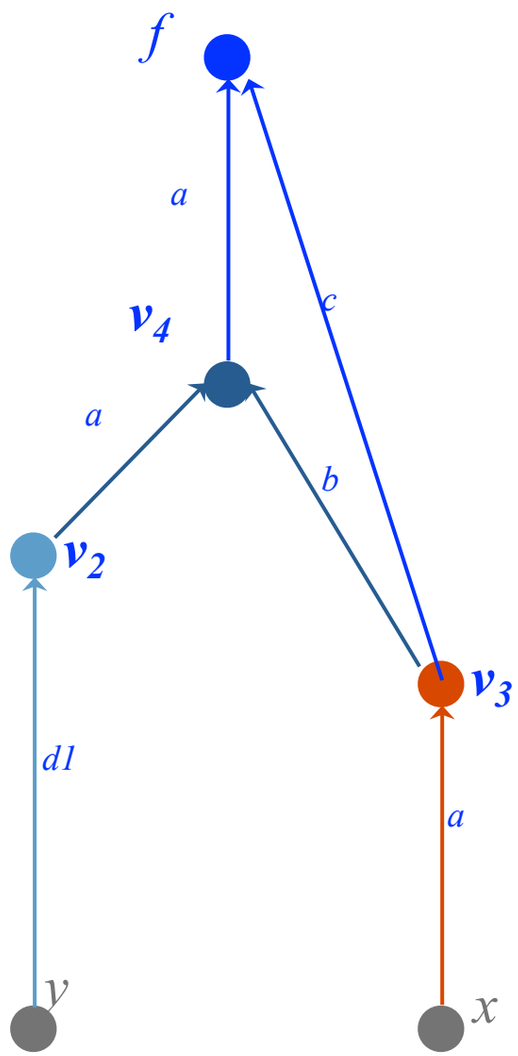
- Multiply each in edge by each out edge, add the product to the edge from the predecessor to the successor
- Conserves path weights
- This procedure always terminates
- The terminal form is a bipartite graph

Forward mode: eliminate vertices in topological order



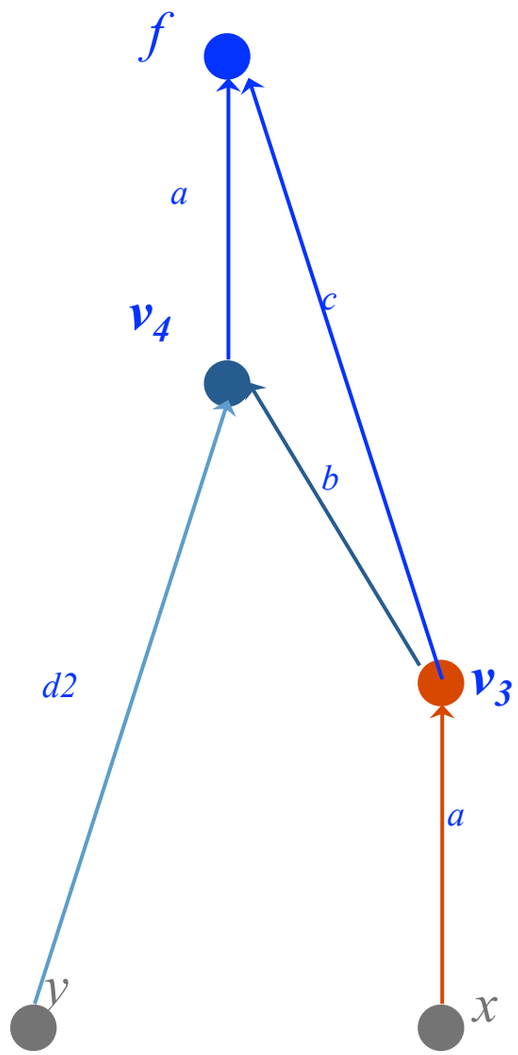
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$

Forward mode: eliminate vertices in topological order



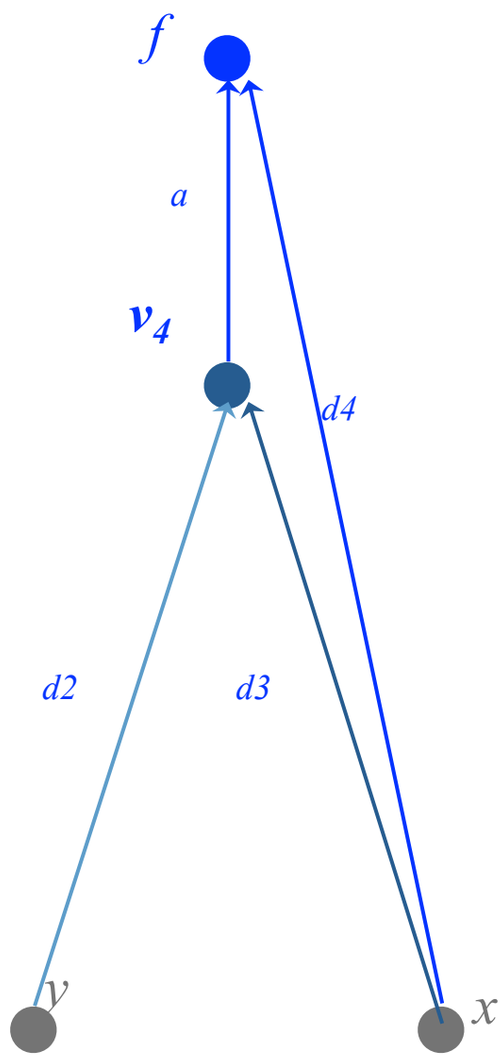
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$

Forward mode: eliminate vertices in topological order



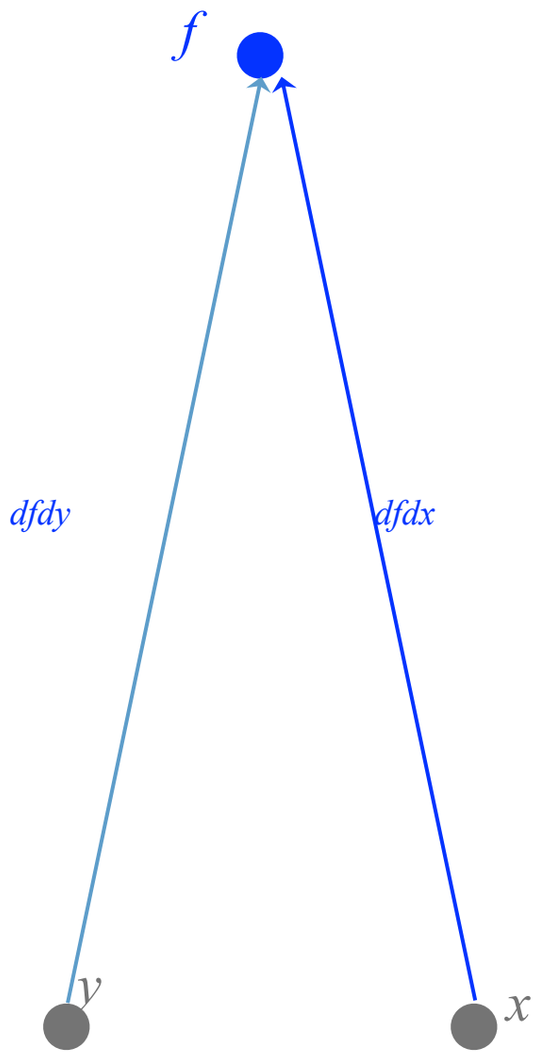
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = d1*a$

Forward mode: eliminate vertices in topological order



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = d1*a$
$d3 = a*b$
$d4 = a*c$

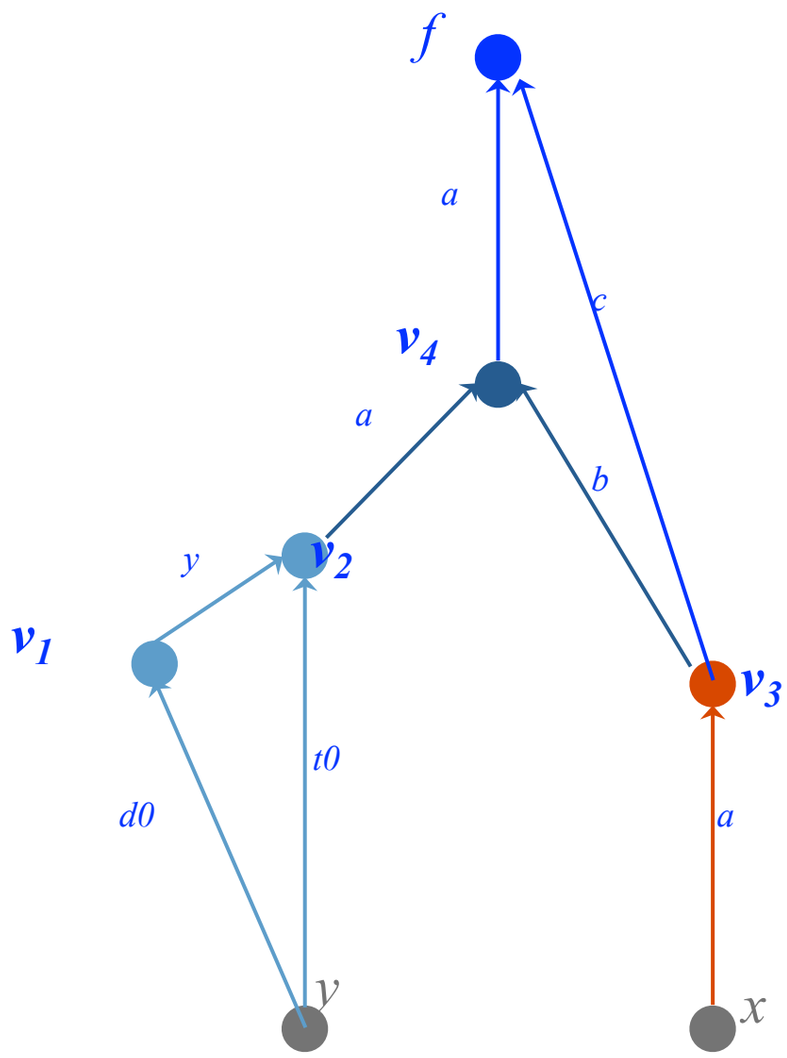
Forward mode: eliminate vertices in topological order



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = d1*a$
$d3 = a*b$
$d4 = a*c$
$dfdy = d2*a$
$dfdx = d4 + d3*a$

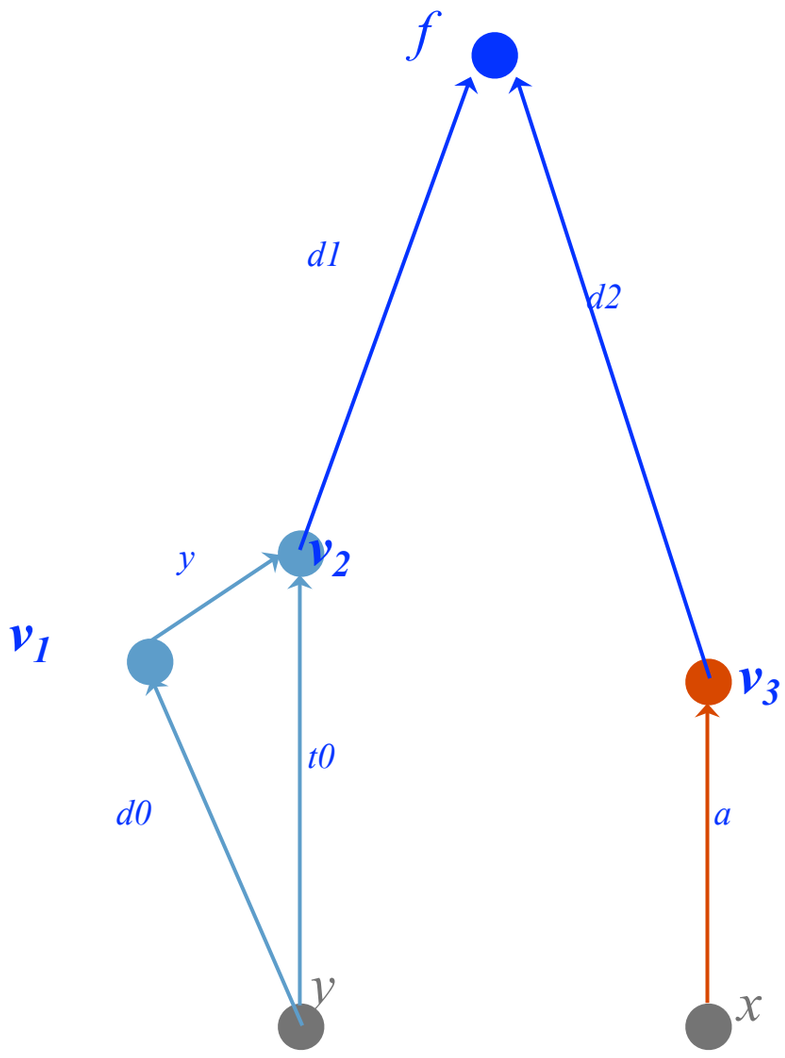
6 mults 2 adds

Reverse mode: eliminate in reverse topological order



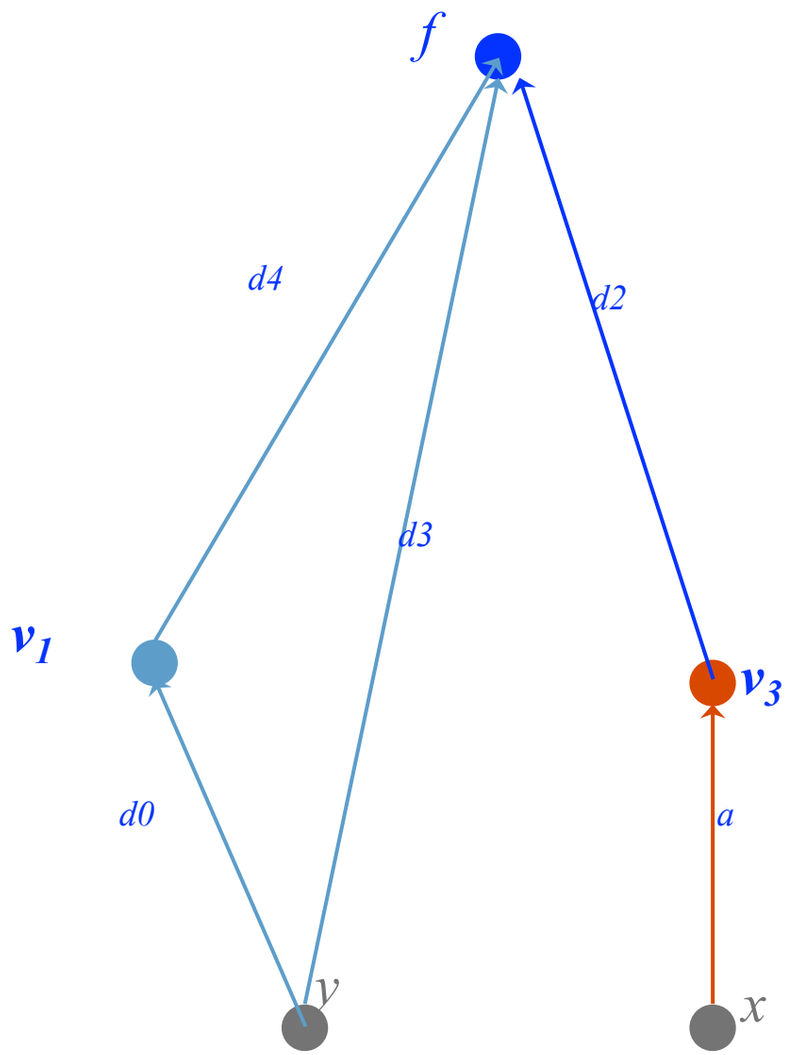
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$

Reverse mode: eliminate in reverse topological order



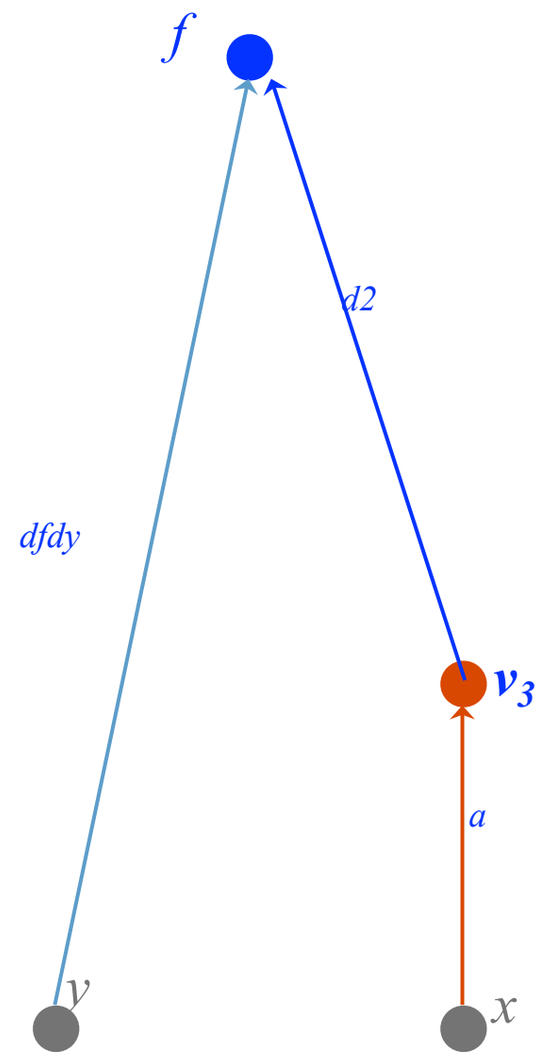
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = a*a$
$d2 = c + b*a$

Reverse mode: eliminate in reverse topological order



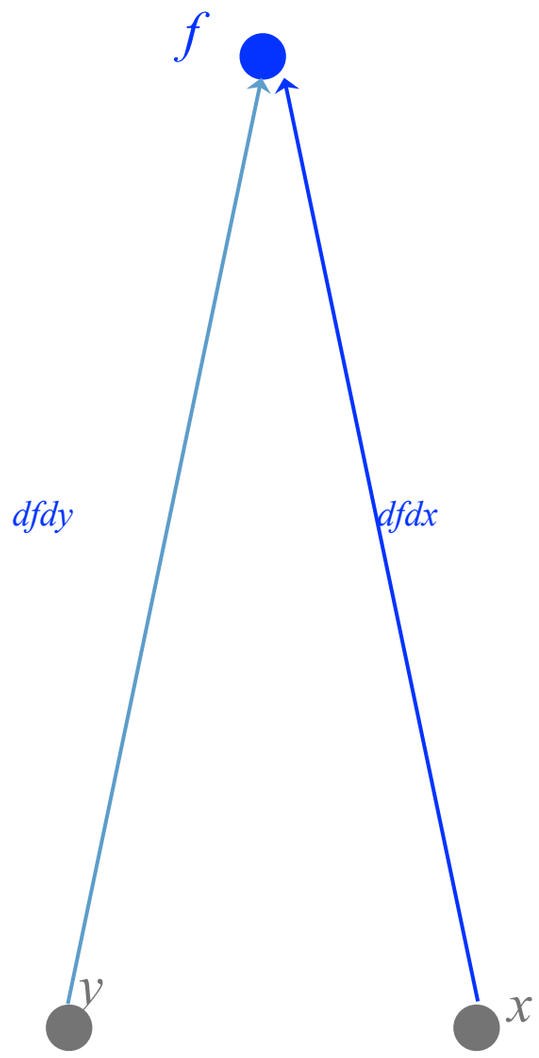
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = a*a$
$d2 = c + b*a$
$d3 = t0*d1$
$d4 = y*d1$

Reverse mode: eliminate in reverse topological order



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = a*a$
$d2 = c + b*a$
$d3 = t0*d1$
$d4 = y*d1$
$dfdy = d3 + d0*d4$

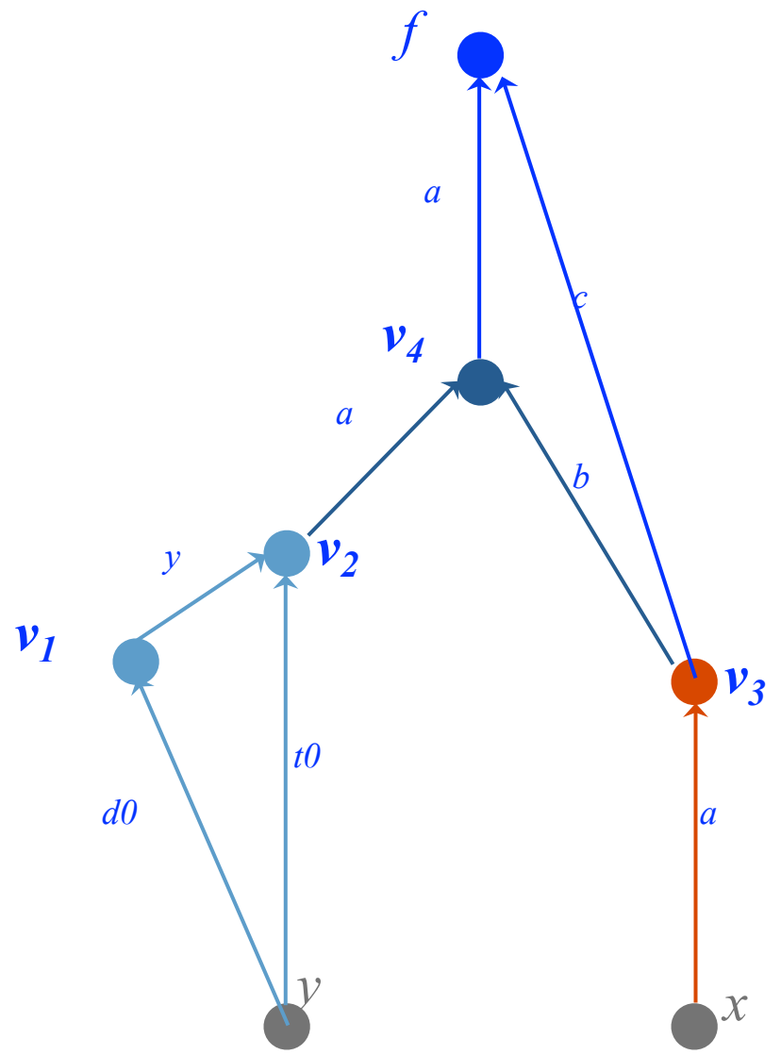
Reverse mode: eliminate in reverse topological order



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = a*a$
$d2 = c + b*a$
$d3 = t0*d1$
$d4 = y*d1$
$dfdy = d3 + d0*d4$
$dfdx = a*d2$

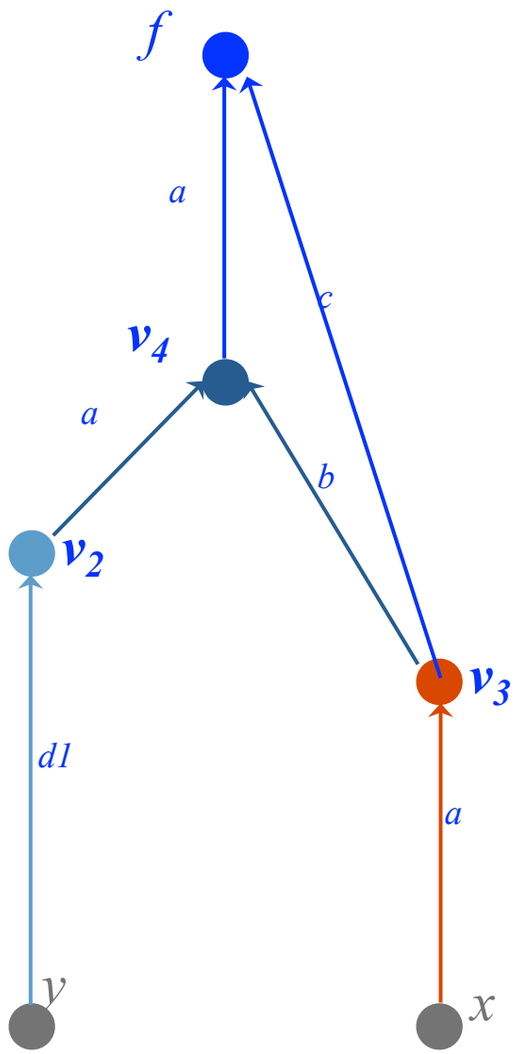
6 mults 2 adds

“Cross-country” mode



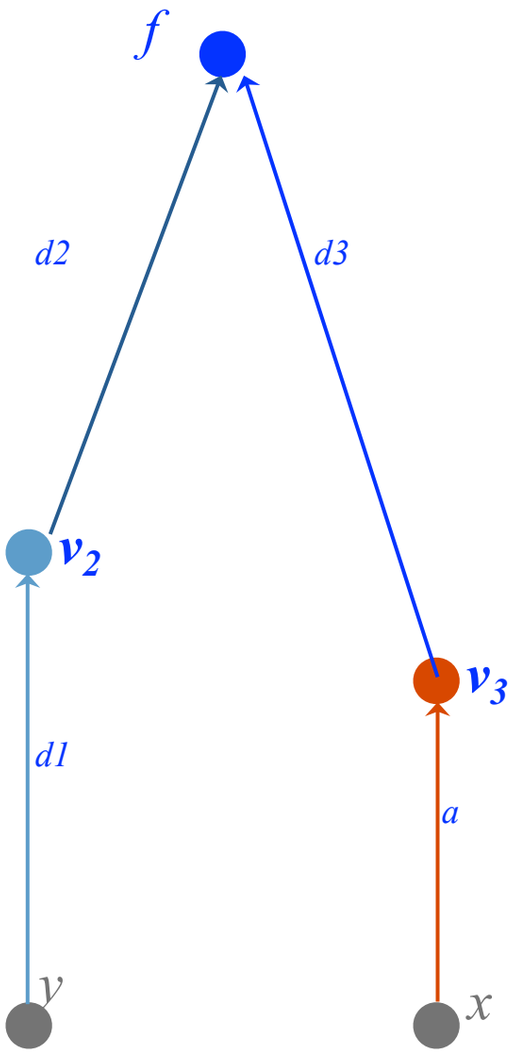
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$

“Cross-country” mode



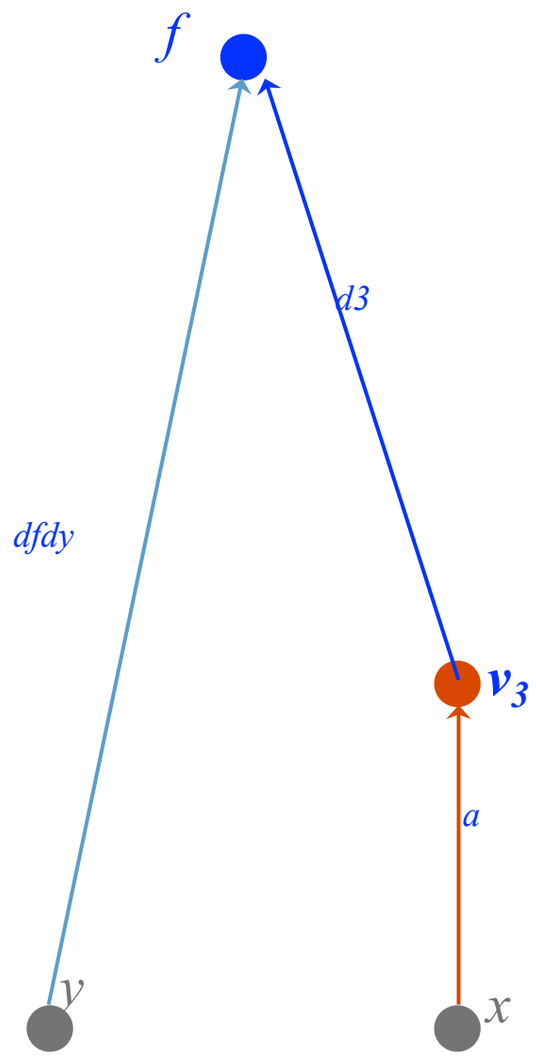
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$

“Cross-country” mode



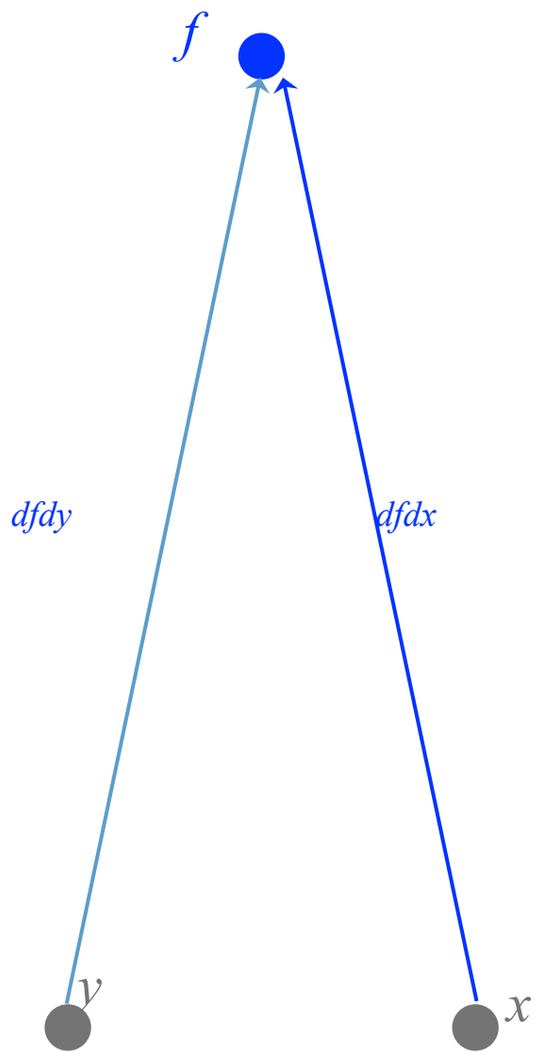
$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = a*a$
$d3 = c + b*a$

“Cross-country” mode



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = a*a$
$d3 = c + b*a$
$dfdy = d1*d2$

“Cross-country” mode



$t0 = sin(y)$
$d0 = cos(y)$
$b = t0*y$
$a = exp(x)$
$c = a*b$
$f = a*c$
$d1 = t0 + d0*y$
$d2 = a*a$
$d3 = c + b*a$
$dfdy = d1*d2$
$dfdx = a*d3$

5 mults 2 adds

What We Know

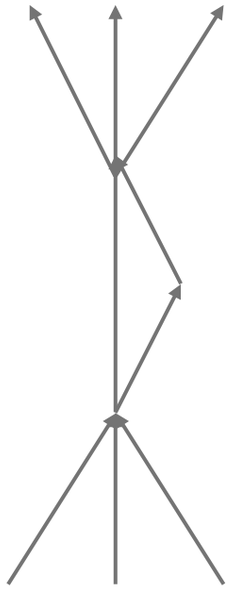
- Reverse mode is within a factor of 2 of optimal for functions with one dependent variable. This bound is sharp.
- Eliminating one edge at a time (edge elimination) can be cheaper than eliminating entire vertices at a time
- Eliminating pairs of edges (face elimination) can be cheaper than edge elimination
- Optimal Jacobian accumulation is NP hard
- Various linear and polynomial time heuristics
- Optimal orderings for certain special cases
  - Polynomial time algorithm for optimal vertex elimination in the case where all intermediate vertices have one out edge

What We Don't Know

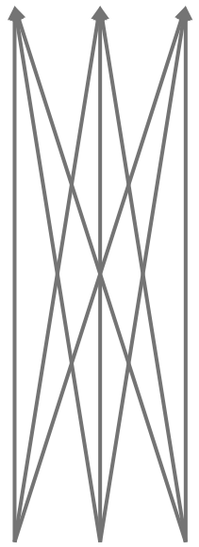
- What is the worst case ratio of optimal vertex elimination to optimal edge elimination? … edge to face?
- When should we stop? (minimal representation problem)
- How to adjust cost metric to account for cache/memory behavior?
- Is O(min(#indeps,#deps)) a sharp bound for the cost of computing a general Jacobian relative to the function?

Minimal graph of a Jacobian (scarcity)

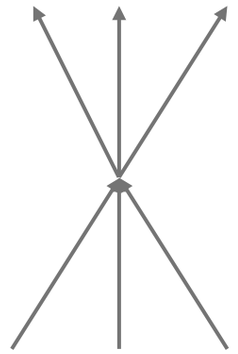
Original DAG



Bipartite DAG



Minimal DAG



Reduce graph to one with minimal number of edges (or smallest number of DOF)

How to find the minimal graph? Relationship to matrix properties?

Avoid "catastrophic fill in" (empirical evidence that this happens in practice)

In essence, represent Jacobian as sum/product of sparse/low-rank matrices

Practical Matters: constructing computational graphs

- At compile time (source transformation)
  - Structure of graph is known, but edge weights are not: in effect, implement inspector (symbolic) phase at compile time (offline), executor (numeric) phase at run time (online)
  - In order to assemble graph from individual statements, must be able to resolve aliases, be able to match variable definitions and uses
  - Scope of computational graph construction is usually limited to statements or basic blocks
  - Computational graph usually has O(10)—O(100) vertices
- At run time (operator overloading)
  - Structure and weights both discovered at runtime
  - Completely online—cannot afford polynomial time algorithms to analyze graph

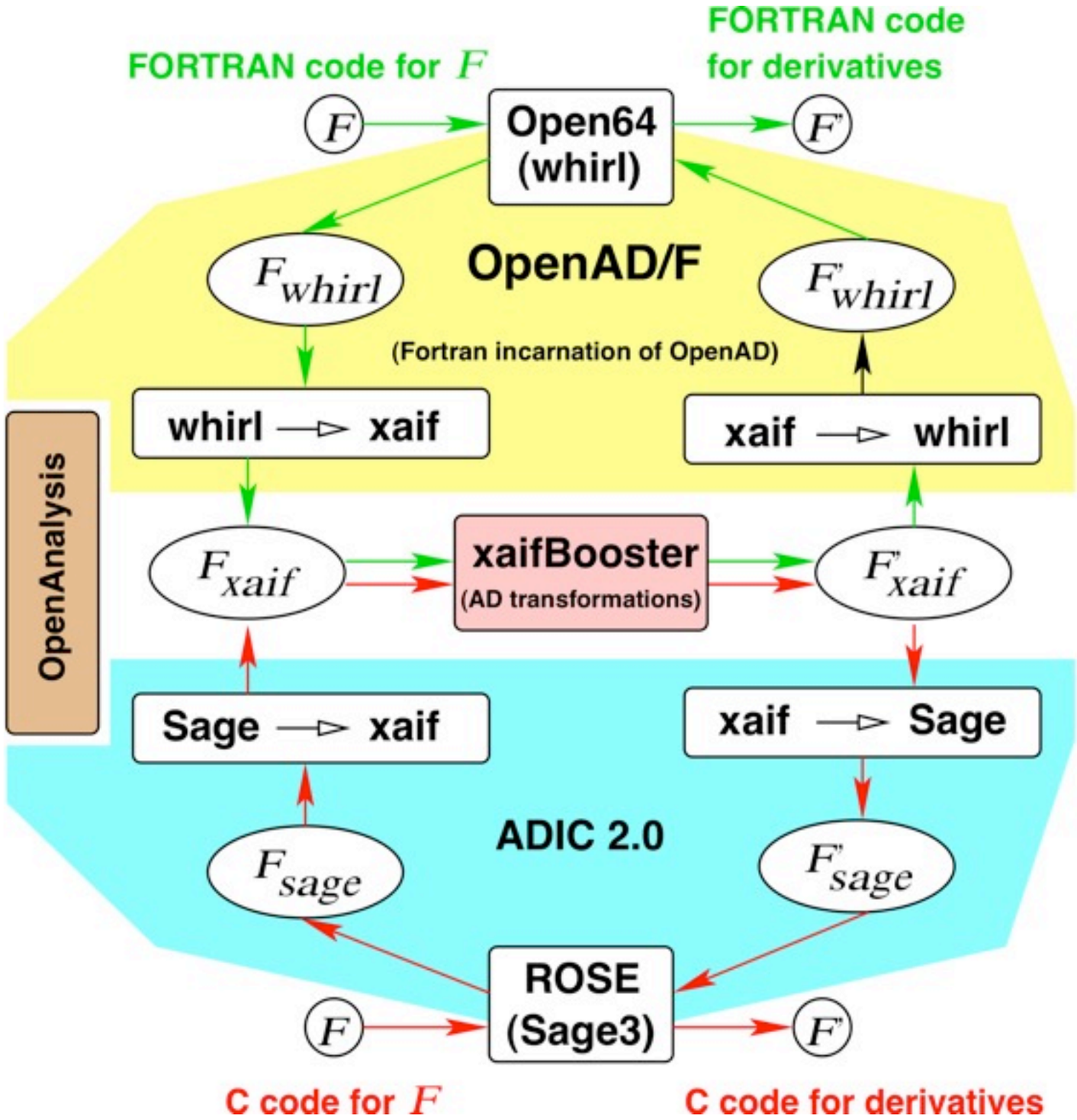
Implementation of source transformation AD tools

- System components
- System architecture
- OpenAnalysis: infrastructure-independent analysis

System components

- Fortran 90/95 parser and C/C++ parser
- Compiler analyses
  - CFG construction
  - Alias analysis
  - Intra- and interprocedural dataflow analysis
- Differentiation (or any mathematically motivated transformation): solve graph problems to improve performance
- Unparser (Fortran 90/95 and C/C++)

OpenAD system architecture



Domain-Specific Data Flow Analysis

- Linearity Analysis
  - Applications in automatic differentiation, optimization, PDEs, ...
  - Identify variables with a linear dependence on input variables [forward data flow] or upon which output variables depend linearly [backward]
  - Data flow analysis formalized by Strout (paper at ICCS2006)
- Activity Analysis
  - Unique to AD (but connections to slicing/chopping)
  - Identify variables that depend on independent variables and upon which the dependent variables depend
  - Data flow analysis formalized by Naumann (Vary (F) & Useful (B))
  - Static analysis: context (in)sensitive, flow (in)sensitive
  - Runtime analysis: simple vary analysis using an active bit
  - Hybrid analysis: restrict dynamic analysis to worthwhile cases (requires may-must staic analysis); Strout & Kreaseck (ICCS2006)

Activity Analysis

```
x: independent
f: dependent

a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end
```

x, f, a, b active

```
x: independent
g: dependent

a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end
```

x, g, c active

```
y: independent
f: dependent

a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end
```

y, f active

Activity Analysis

```
x: independent
f: dependent

a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end

x, f, a, b active
```

```
x: independent
g: dependent

a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end

x, g, c active
```

```
y: independent
f: dependent

a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end

y, f active
```

Activity Analysis

x: independent
f: dependent

```
a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end
```

x, f, a, b active

x: independent
g: dependent

```
a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end
```

x, g, c active

y: independent
f: dependent

```
a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end
```

y, f active

Activity Analysis

```
x: independent
f: dependent

a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end

x, f, a, b active
```

```
x: independent
g: dependent

a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end

x, g, c active
```

```
y: independent
f: dependent

a = 0.0
f = 0.0
g = 0.0
for i = 1, N
  a += x[i]*x[i]
  b = sin(a)
  c = y[i] - x[i]
  f += b
  g += c
end

y, f active
```

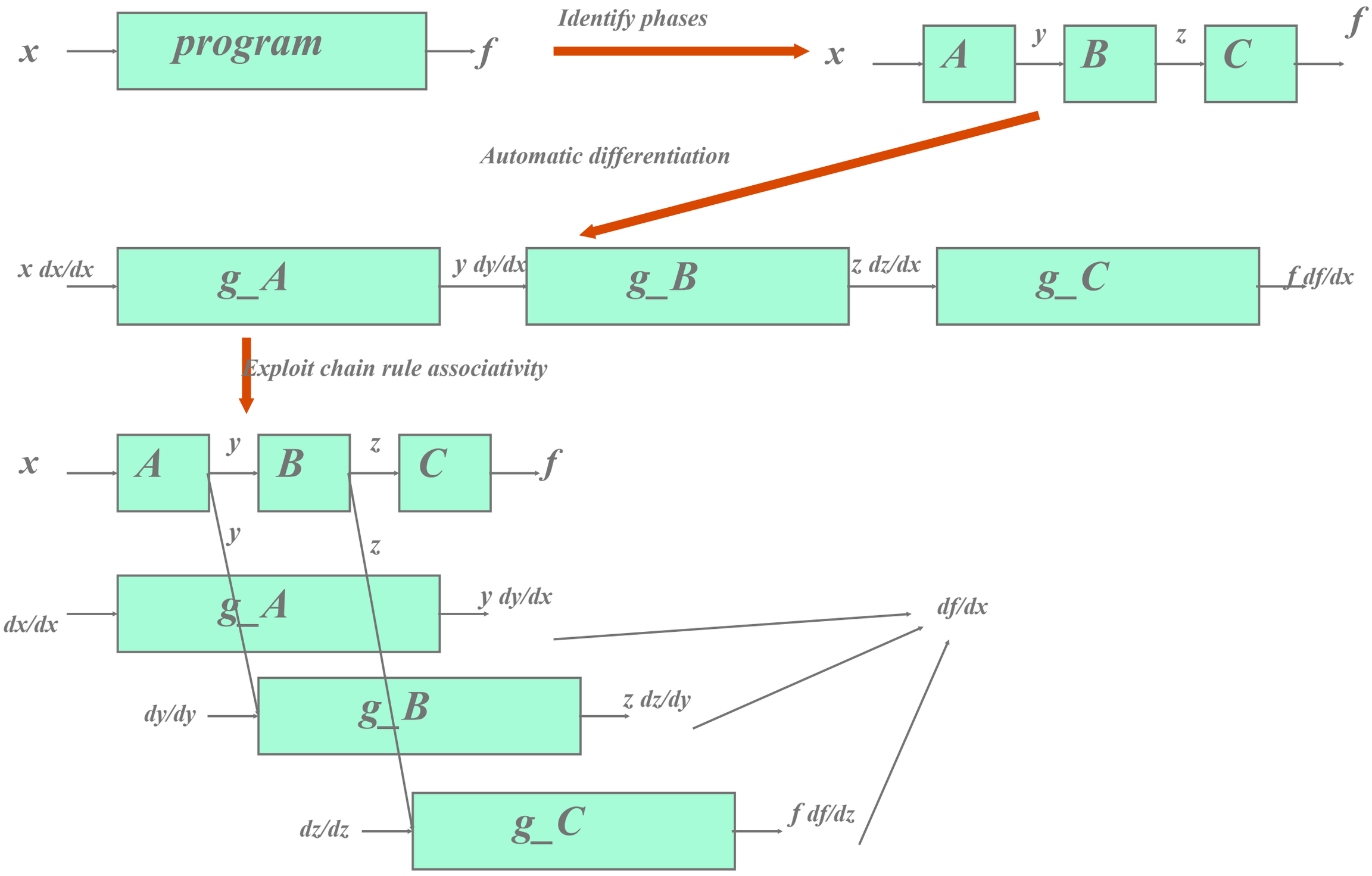
AD and Parallel Computation

- Two aspects
  - AD-enabled parallelism
    - Research began with Bischof in 1991
    - Important advances in 1994-5
  - AD of parallel programs
    - Earliest work by Hinkins (1994)
    - Wider scope, consideration of MPI by H., Carle (1997)

AD-enabled Parallelism

- Data parallelism
  - Computing derivatives with respect to multiple independent variables
  - Can compute in parallel via “strip-mining”
  - Redundant evaluation of function
  - Load balance an issue for second derivatives
- “Time parallelism”
- Multithreading (Roh)

Time Parallelism



AD of Parallel Programs

- AD of data parallel programs not too difficult (especially correctness)
- AD of message passing programs is more difficult
  - Harder to maintain association between variable and derivative object
  - Must track data flow through messages: ongoing research on Dataflow analysis of MPI programs (Strout, Hovland, Kreaseck)
- Must differentiate parallel intrinsics (parallel reductions)
  - Sum is easy
  - Max, Min can be tricky when there are “ties”
  - Product and user-defined reductions challenging
    - Modified version of parallel prefix algorithm possible

Mathematical Challenges in AD

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

Other Issues

- Reverse mode requires that control flow of entire program be reversed
  - Need complete path information
  - Need to store or recompute intermediate variables
  - Need (incremental) checkpointing for full system state
- Reverse mode differentiation of implicitly parallel programs is hard
- Many real applications are written in multiple languages or languages not yet supported by tools

Checkpointing for Resilience vs. Program Reversal

- Need all checkpoints for adjoint computation, only most recent for resilience
- Checkpoints for adjoints can be stored in volatile memory
- Checkpoints for adjoints can be incremental (state that will change in next step)
- Analysis of optimal checkpoint schedule for adjoint computation does not account for time required to store/restore checkpoint
- Different optimal checkpoint schedules

For resilience:

$$\tau \approx \sqrt{2\delta(M+R)}$$

For program reversal in adjoint computation

$$l(c,r) = \binom{c+r}{c}$$

Conclusions

- Automatic differentiation research involves a wide range of computer science and a little mathematics
- AD is a powerful tool for scientific computing
- Future challenges include:
  - New languages (Java, python, bytecode, …)
  - Continued work on dataflow analysis of MPI programs
  - Graph heuristics for data locality, second derivatives (symmetry)
  - Effective and efficient control flow reversal (important for reverse mode)
  - Efficient “user-level” checkpointing via source transformation

For More Information

- Andreas Griewank, *Evaluating Derivatives*, SIAM, 2000.
- 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/
- E-mail: hovland@mcs.anl.gov