THE UNIVERSITY OF CHICAGO

Lecture 2

Mihai Anitescu STAT 310

PLAN

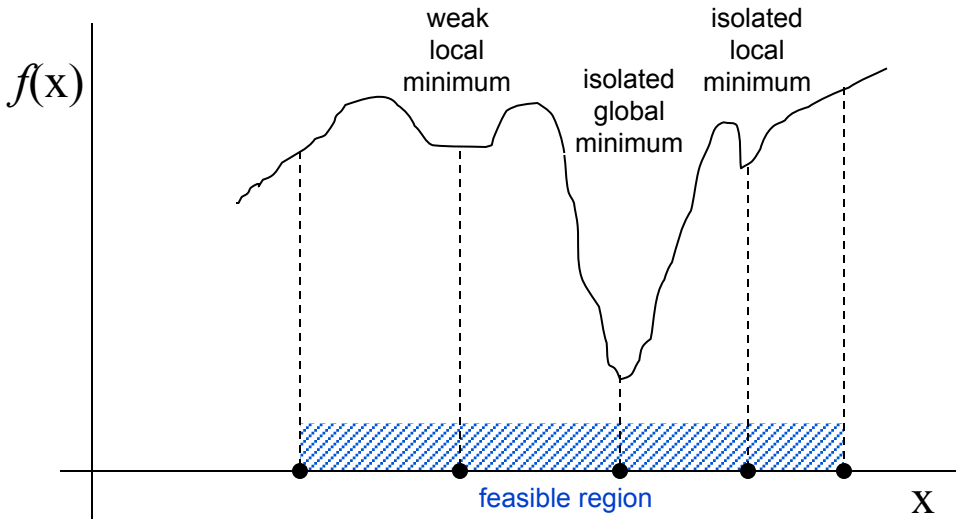
- Announcements.
- 1.4 Course objective.
- 2.1 Newton's method and implications.
- 2.2 Computing Derivatives.
- 2.3 Linear Algebra.
- 2.4 Sparse Linear Algebra

ANNOUNCEMENTS

- Homework.
- 01/11 office hour.
- Stop me at EXPAND and DEMO.
- I will try to post slides; MATLAB diary; homeworks; and software. If you cannot access them contact me.

1.4 COURSE OBJECTIVES

Types of minima



- which of the minima is found depends on the starting point
- such minima often occur in real applications

Summary LOCAL optimality conditions

- Conditions for *local* minimum of unconstrained problem: $\min_x f(x); \quad f \in C^2$

  - First Order Necessary Condition: $\nabla f = \mathbf{0}$
  - Second Order Sufficient Condition: $\nabla^2_{xx} f \geq \mathbf{0}$

  - Second Order Sufficient Condition: $\nabla^2_{xx} f \succ \mathbf{0}$

- EXPAND: Geometry.

How about global optimality?

- There is no simple criterion; extremely hard question (most such problems are NP hard).
- One exception f is convex:

$$\nabla^2_{xx} f \succ \mathbf{0} \quad \text{EVERYWHERE}$$

- But we will consider the general case.

Course objectives

- Derive efficient iterative algorithms to "solve" the problem (and its constrained form).

$$\min_x f(x); \quad f \in C^2$$

- Solve= guarantee convergence to a point that satisfies the NECESSARY conditions.
- Typically, if point also SUFFICIENT, then local convergence should be FAST (e.g. quadratic),
- NOTE: WE WILL DO NO SIMULATION.

2.1 Intro to Methods for Continuous Optimization: Newton' Method

- Focus on continuous numerical optimization methods
  - Virtually ALL of them use the Newton Method idea

Newton's Method

- Idea in 1D:
  - Fit parabola through 3 points, find minimum
  - Compute derivatives as well as positions, fit cubic
  - Use *second* derivatives: Newton by means of Taylor expansion at the current point.

Newton's Method

Interpolating Poly (Taylor)

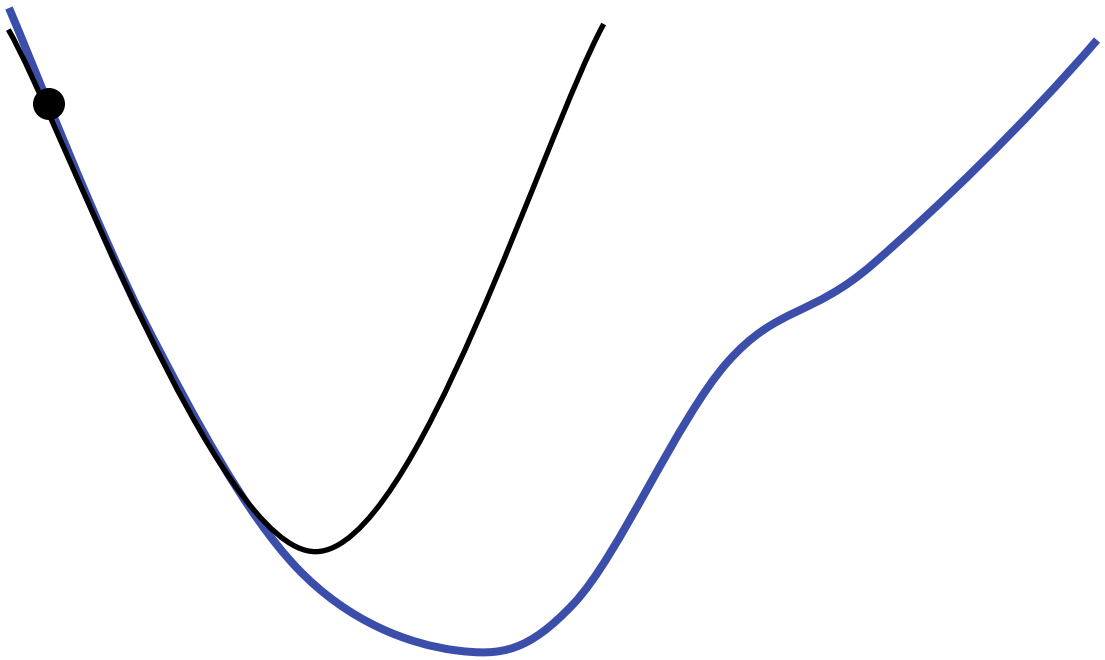
- At each step:

$$\min_x \left[\frac{1}{2}(x - x_k)^2 f''(x_k) + f'(x_k)(x - x_k) + f(x_k) \right]$$

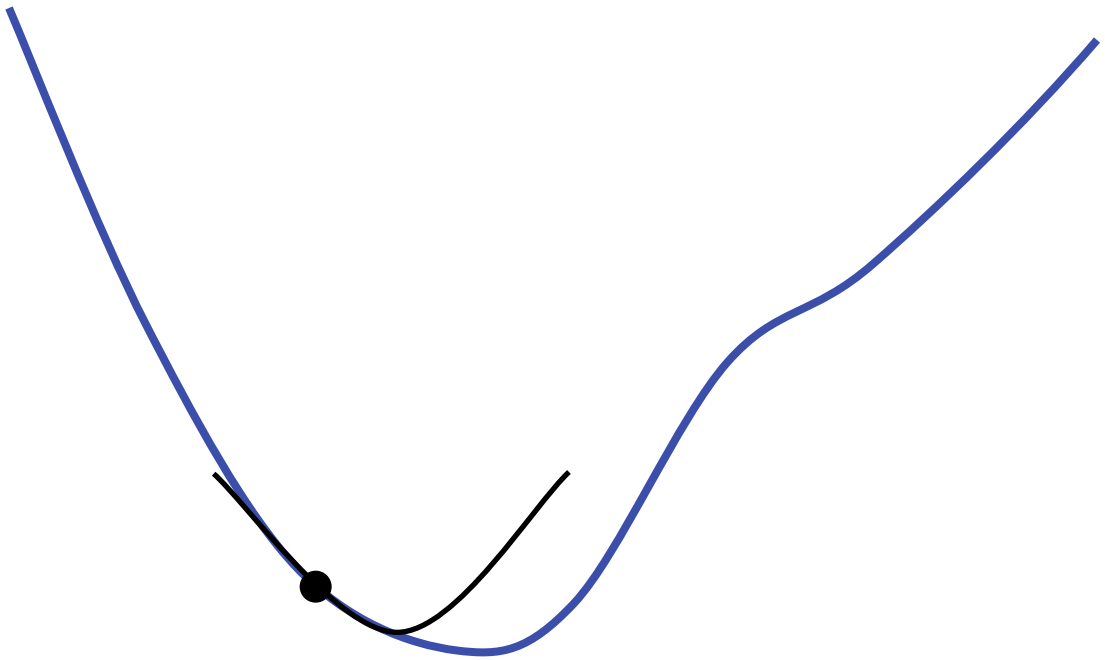
$$\Rightarrow x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

- Requires 1st and 2nd derivatives

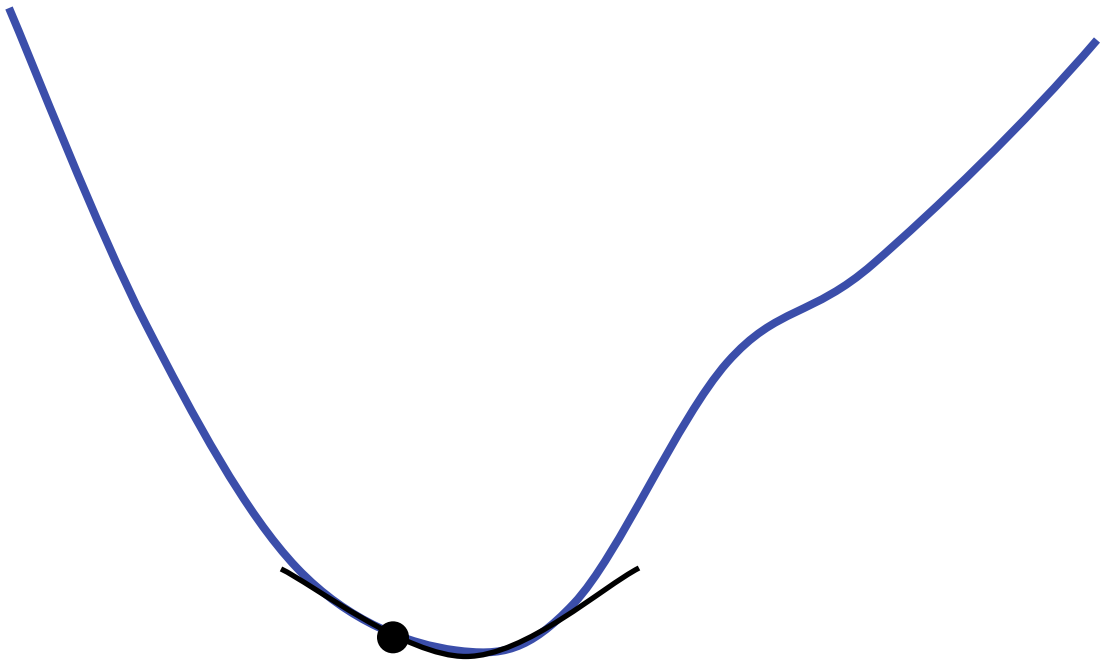
Newton's Method



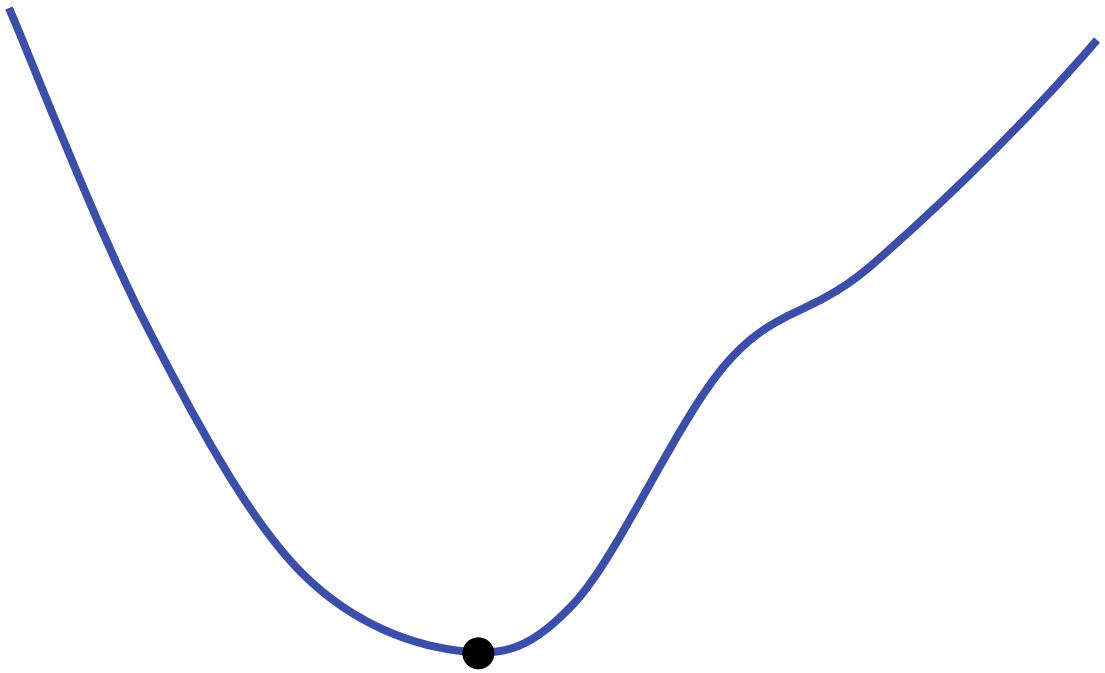
Newton's Method



Newton's Method



Newton's Method



Newton's Method in Multiple Dimensions

- Replace 1st derivative with gradient, 2nd derivative with Hessian

$$f(x, y)$$

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

Newton's Method in Multiple Dimensions

- Replace 1st derivative with gradient, 2nd derivative with Hessian
- So, $\vec{x}_{k+1} = \vec{x}_k - H^{-1}(\vec{x}_k)\nabla f(\vec{x}_k)$

RECAP: Taylor Series

- The *Taylor series* is a representation of a function as an infinite sum of terms calculated from the values of its derivatives at a single point. It may be regarded as the limit of the Taylor polynomials

Taylor series for a polynominal function, the wt. sum of its derivatives

$$f(x) = \frac{f(x_0)}{0!}(x-x_0)^0 + \frac{f'(x_0)}{1!}(x-x_0)^1 + \frac{f''(x_0)}{2!}(x-x_0)^2 + \ldots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n$$

Taylor series for an arbitrary function, any function $\approx$ by the wt. sum of its derivatives

$$f(x) - \frac{f(x_0)}{0!}(x-x_0)^0 = \frac{f'(x_0)}{1!}(x-x_0)^1 + \frac{f''(x_0)}{2!}(x-x_0)^2 + \ldots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n + R$$

Where $R = \frac{f^{(n+1)}(p)}{(n+1)!}(x-x_0)^{n+1}$

CHICAGO

Recap: Multi-dimensional Taylor expansion

A function may be approximated locally by its Taylor series expansion about a point **x***

$$\boxed{f(\mathbf{x}^* + \mathbf{x}) \approx f(\mathbf{x}^*) + \nabla f^T \mathbf{x} + \frac{1}{2}\mathbf{x}^T \mathbf{H} \mathbf{x}}$$

where the gradient $\nabla f(\mathbf{x}^*)$ is the vector

$$\nabla f(\mathbf{x}^*) = \left[\frac{\partial f}{x_1} \cdots \frac{\partial f}{x_N}\right]^T$$

and the Hessian **H(x*)** is the symmetric matrix

$$\mathbf{H}(\mathbf{x}^*) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_N} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_N \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_N^2} \end{bmatrix}$$

Q: What is a residual bound? How would you prove it from 1D?

Recap: Orders of convergence

- R-convergence and Q-convergence.
- EXPAND

- Q: Which order of convergence is desirable? Why?

Newton's Method in Multiple Dimensions

- EXPAND: Justify by Quadratic Approximation, and sketch quadratic convergence.
- Tends to be extremely fragile unless function very smooth and starting close to minimum.
- Nevertheless, this iteration is the basis of most modern numerical optimization.

Newton Method: Abstraction and Extension

- "Minimizing a quadratic model iteratively"
- EXPAND

NM Implementations

- Descent Methods, Secant Methods may be seen as "Newton-Like"
- All "Newton-like" methods need to solve a linear system of equations.
- All "Newton-like" methods need the implementation of derivative information (unless a modeling language provides it for free, such as AMPL). .

2.2 Computing Derivatives

- Three important ways.
- 1. Hand Coding (rarely done and error prone). Typical failure: do the physics, ignore the design till it is too late.
- 2. Divided differences.
- 3. Automatic Differentiation.

CHICAGO

2.2.1. Divided Differences

The formulas developed next can be used to estimate the value of a derivative at a particular value in the domain of a function, they are primarily used in the solution of differential equations in what called **finite difference methods**.

Note: There a several ways to generate the following formulas that approximate f '(x). The text uses interpolation. Here we use Taylor expansions.

A **difference quotient** is a change in function values divided by the corresponding domain values. For example

$$\frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}.$$

For y = f(x) with $x = x_0$ and x_1 we have

$$\frac{\Delta y}{\Delta x} = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

or for y = f(x) with $x = x_0$ and $x_1 = x_0 + h$ we have

$$\frac{\Delta y}{\Delta x} = \frac{f(x_0 + h) - f(x_0)}{x_0 + h - x_0} = \frac{f(x_0 + h) - f(x_0)}{h}.$$

Note that the last formula also applies in multiple dimensions, if I perturb one coordinate at the time. EXPAND

CHICAGO

Forward Difference Approximation

Given $\mathbf{y = f(x)}$ and $y_h = \dfrac{f(x_0 + h) - f(x_0)}{h}$ for h > 0 and some fixed value x_0. Assume also that $|f''(x)|$ is bounded by a constant C. Show that $f'(x_0) = y_h + O(h)$. Here we use Taylor's Theorem.

Proof: Expand $f(x_0 + h)$ using Taylor's Theorem with center of expansion x_0 we get

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(\xi)$$ where ξ is between x_0 and $x_0 + h$.

Subtract $f(x_0)$ from both sides & divide by h.

It follows then that $$y_h = \frac{\cancel{f(x_0)} + hf'(x_0) + \frac{h^2}{2} f''(\xi) - \cancel{f(x_0)}}{h} = f'(x_0) + \frac{h}{2} f''(\xi).$$

So $y_h = f'(x_0) + \frac{h}{2} f''(\xi)$. Using that $|f''(x)| \le C$ we get that $|f'(x_0) - y_h| \le \frac{h}{2} C$. It then follows that $f'(x_0) = y_h + O(h)$.

$\dfrac{f(x_0 + h) - f(x_0)}{h}$ is called the **Forward Difference Approximation** to $f'(x)$ at $x = x_0$.

Finite Differences

- Nevertheless, we use forward differences, particularly in multiple dimensions. (Q: How many function evaluations do I need for gradient?)
- Q: How do we choose the parameter h? EXPAND
- DEMO.
- EXPAND Multiple Dimension Procedure.

2.2.2 Automatic Differentiation

- There exists another way, based upon the chain rule, implemented automatically by a "compiler-like" approach.
- Automatic (or Algorithmic) Differentiation (AD) is a technology for automatically augmenting computer programs, including arbitrarily complex simulations, with statements for the computation of derivatives
- In MATLAB, done through package "intval".

Automatic Differentiation (AD) in a Nutshell

- Technique for computing analytic derivatives of programs (millions of loc)
- Derivatives used in optimization, nonlinear PDEs, sensitivity analysis, inverse problems, etc.

Automatic Differentiation (AD) in a Nutshell

- 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

Automatic Differentiation (AD) in a Nutshell

- Associativity of the chain rule leads to many ways of combining partial derivatives, including two main modes: forward and reverse
- Can be implemented using source transformation or operator overloading

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
- EXPAND: Example 1D case, + reverse.

A Small Example

... lots of code...

a = cos(x)

b = sin(y)*y*y

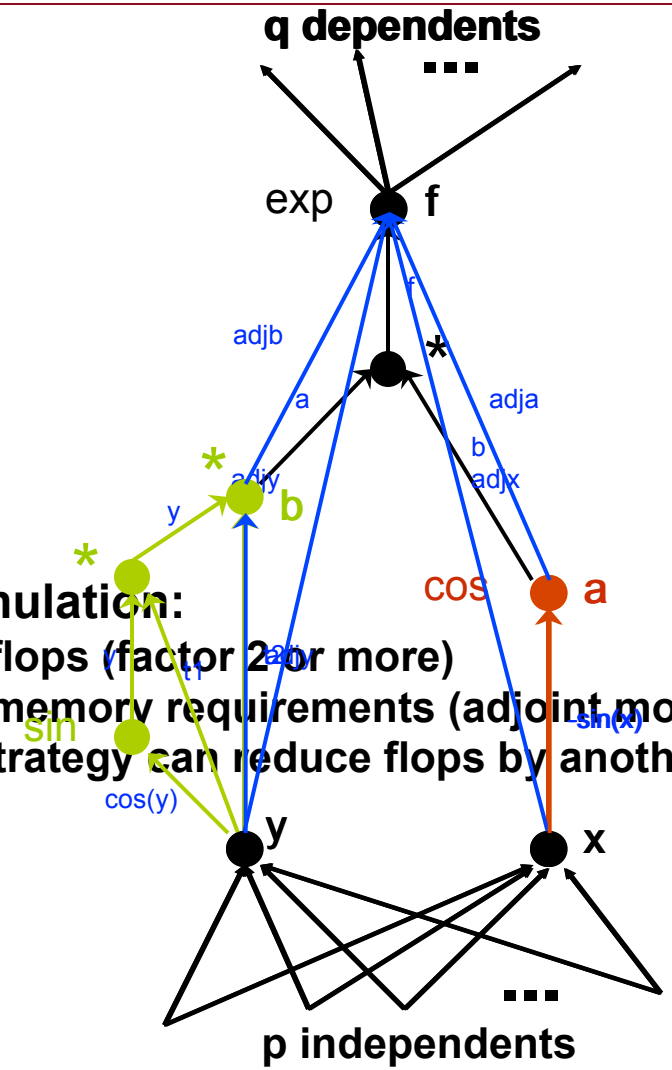
f = exp(a*b)

... lots of code...

Forward mode: 3n+12p

Adjoint mode: 17+12p

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

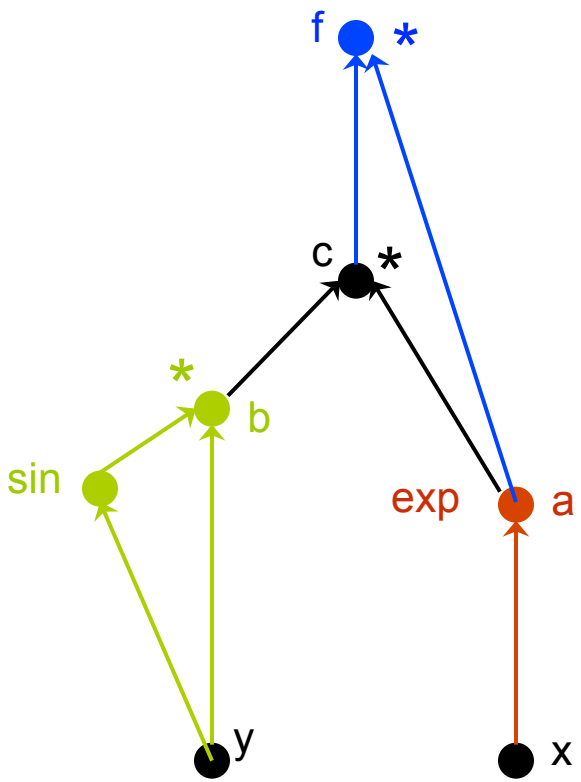


Preaccumulation:

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

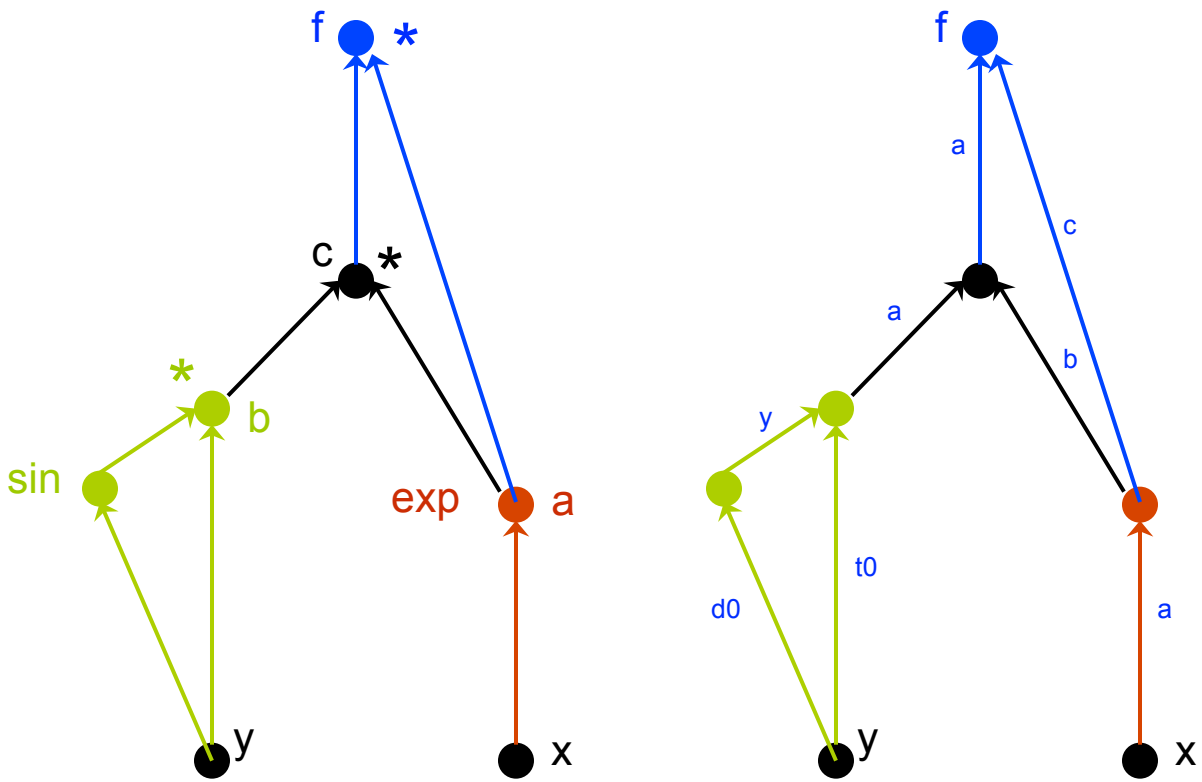
A simple example

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

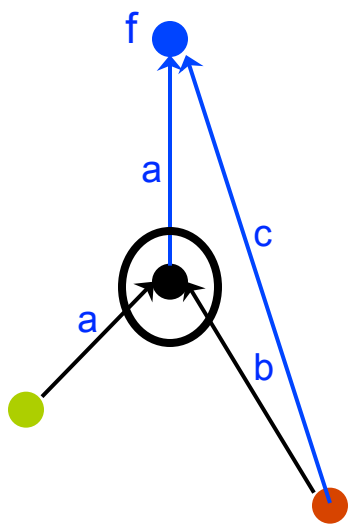


A simple example

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

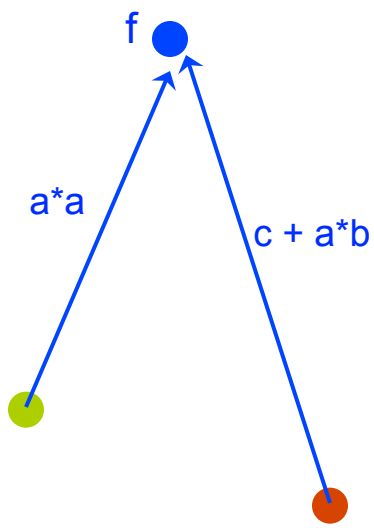


Vertex elimination



- 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

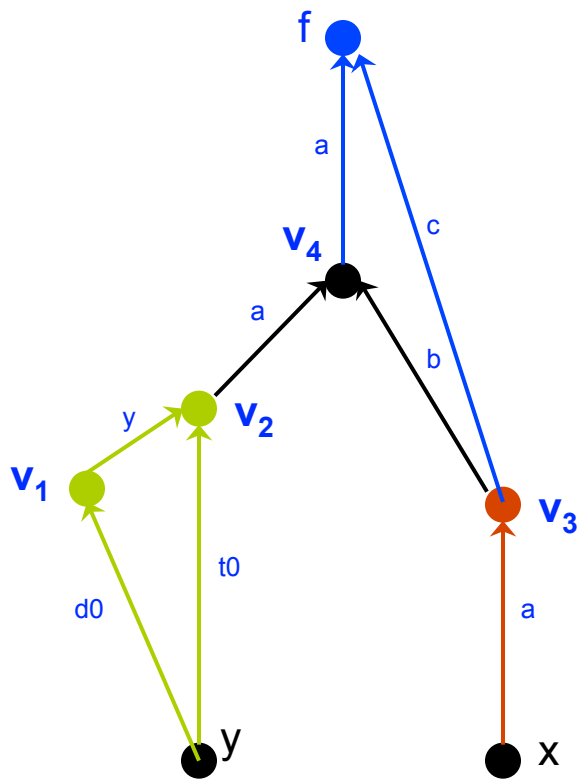
Vertex elimination



- 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

CHICAGO

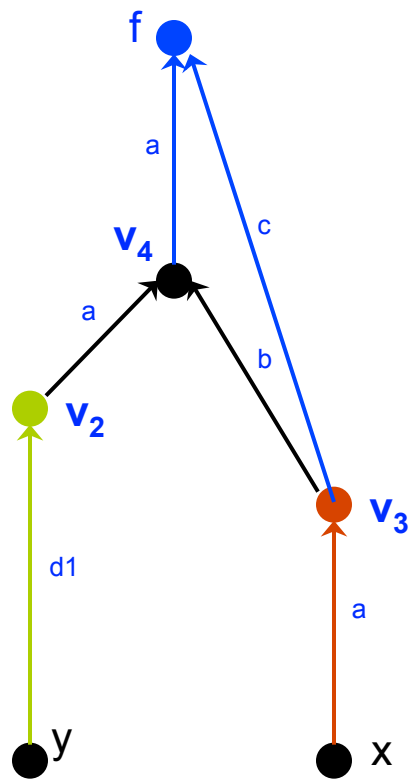
Forward mode: eliminate vertices in topological order



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

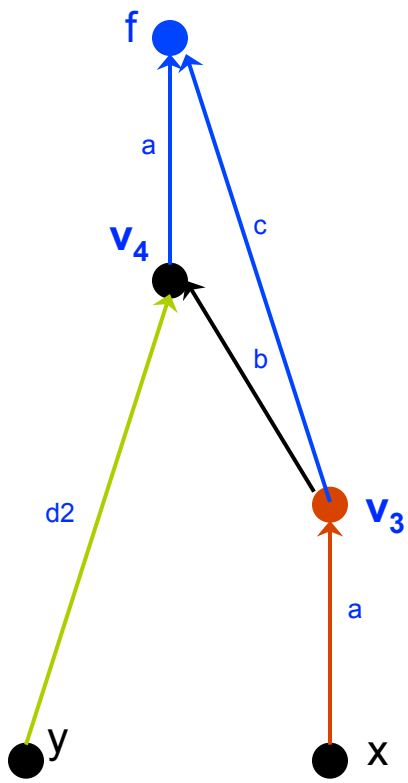
CHICAGO

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

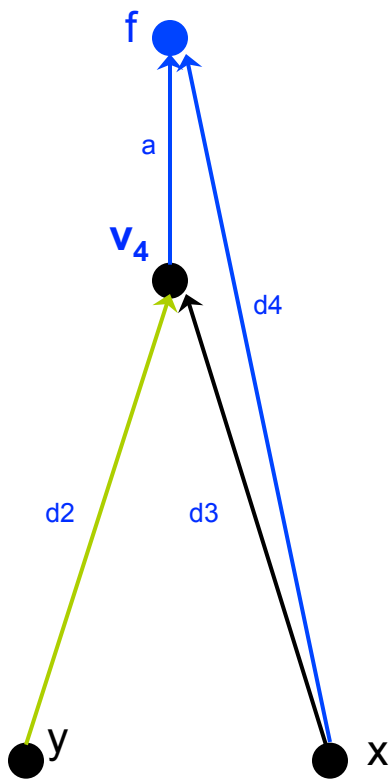
Forward mode: eliminate vertices in topological order



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

CHICAGO

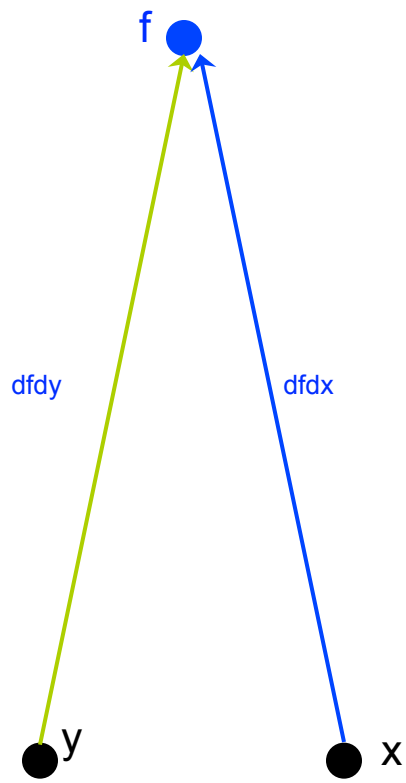
Forward mode: eliminate vertices in topological order



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

CHICAGO

Forward mode: eliminate vertices in topological order

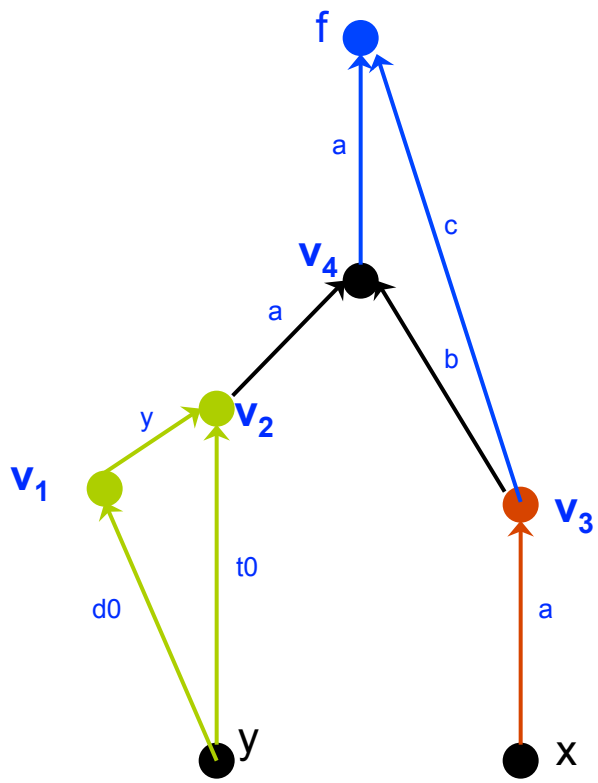



```
t0 = sin(y)
d0 = cos(y)
b = t0*y
a = exp(x)
c = a*b
f = a*c
d1 = t0 + d0*y
d2 = d1*a
d3 = a*b
d4 = a*c
dfdy = d2*a
dfdx = d4 + d3*a
```

6 mults 2 adds

CHICAGO

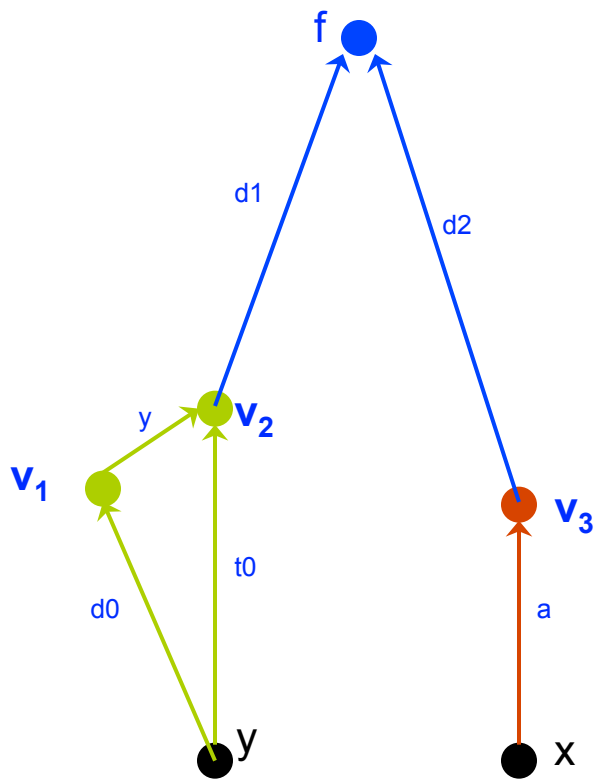
Reverse mode: eliminate in reverse topological order



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

CHICAGO

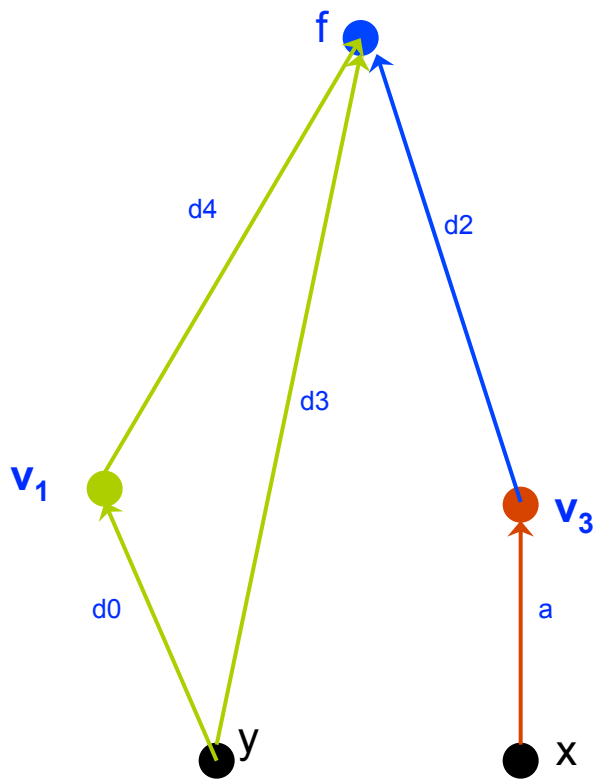
Reverse mode: eliminate in reverse topological order



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

CHICAGO

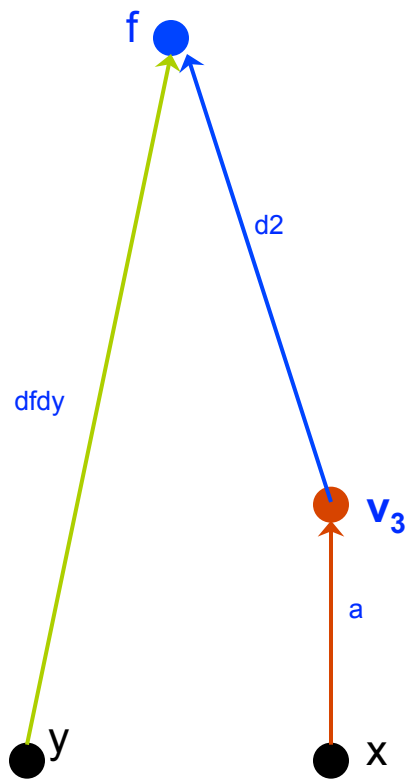
Reverse mode: eliminate in reverse topological order



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

CHICAGO

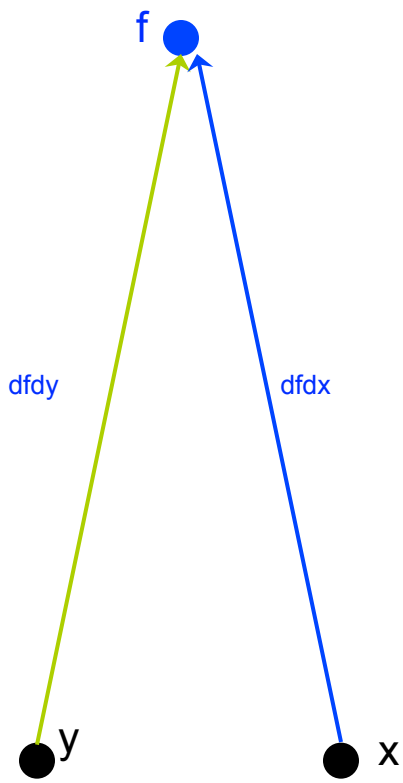
Reverse mode: eliminate in reverse topological order



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

CHICAGO

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

Forward gradient Calculation

- Forward mode computes $\nabla f; \quad f : R^n \rightarrow R^m$
  - At a cost proportional to the number of components of f.
  - Ideal when number of independent variables is small
  - Follows control flow of function computation
  - Cost is comparable to finite differences (can be much less, rarely much more)

Forward versus Reverse

- Reverse mode computes $J = \nabla f; \quad f : R^n \to R^m$
  - At a cost proportional to m
  - Ideal for $J^T v$, or J when number of dependent variables is small
  - Cost can be substantially less than finite differences
- COST IF m=1 IS NO MORE THAN 5* COST OF FEVAL. EXPAND.

AD versus divided differences

- AD is preferable whenever implementable.
- C, Fortran versions exist.
- In Matlab, free package INTVAL (one of the main reasons not doing C). DEMO
- Nevertheless, sometimes, the source code DOES not exist. (e.g max likelihood).
- Then, divided differences.