

AMSC 607 / CMSC 764 Advanced Numerical Optimization
Fall 2003

UNIT 2: Unconstrained Optimization
PART 2: Alternatives to Newton's Method
Dianne P. O'Leary
©2001,2003

Unconstrained Optimization: Alternatives to Newton's method

Reference: N&S Chapters 11 and 12.

The plan:

Recall that Newton's method is very fast (if it converges) but it requires the gradient and the Hessian matrix to be evaluated at each iteration.

Next we develop some alternatives to Newton's method:

- Some alternatives that avoid calculation of the Hessian:
 - Quasi-Newton methods
 - finite-difference Newton methods
- Algorithms that avoid calculation of the Hessian and storage of any matrix:
 - steepest descent
 - nonlinear conjugate gradient methods
 - limited-memory Quasi-Newton methods
 - truncated Newton methods
- Technology that helps in the calculation of the Hessian: automated differentiation.
- Methods that require no derivatives
 - finite difference methods
 - Nelder&Meade simplex
 - pattern search

Methods that require only first derivatives

If the Hessian is unavailable...

Notation:

- H = Hessian matrix.
- B is its approximation.
- C is the approximation to H^{-1} .

Problem: Solve

$$\min_x f(x)$$

when $g(x)$ can be computed, but not $H(x)$.

Two options:

- Estimate $H(x)$ using finite differences: **discrete Newton**. This can work well.
- Approximate $H(x)$ using **Quasi-Newton** methods. (Also called **variable metric**.)

No Hessian Method 1: Quasi-Newton Method

Quasi-Newton methods could be the basis for a full course; there is a textbook by Dennis and Schnabel. We'll just hit the high points.

Reference: SIAM Rev 21 1979 443-459.

Broyden's chapter in Murray's book.

The idea behind Quasi-Newton methods

The Newton step:

$$p = -H^{-1}g$$

(H and g evaluated at $x^{(k)}$.)

The Quasi-Newton step: accumulate an approximation

$$B^{(k)} \approx H(x^{(k)})$$

using **free** information!

What information comes free?

At step k , we know $g(x^{(k)})$ and we compute $g(x^{(k+1)})$ where $x^{(k+1)} = x^{(k)} + s^{(k)}$.

$H(x^{(k)})$ satisfies

$$H^{(k)} s^{(k)} = \lim_{h \rightarrow 0} \frac{g(x^{(k)} + hs^{(k)}) - g(x^{(k)})}{h},$$

In fact, if f is quadratic, then

$$H^{(k)} s^{(k)} = g(x^{(k)} + s^{(k)}) - g(x^{(k)})$$

We'll ask the same property of our approximation $B^{(k+1)}$ and call this the **secant equation**:

$$B^{(k+1)} s^{(k)} = g^{(k+1)} - g^{(k)}.$$

So we know how we want $B^{(k+1)}$ to behave in the direction $g^{(k+1)} - g^{(k)}$, and we have no new information in any other direction, so we could require

$$B^{(k+1)} v = B^{(k)} v, \text{ if } v^T s^{(k)} = 0.$$

There is a unique matrix $B^{(k+1)}$ that satisfies the secant equation and the no-change conditions. It is called **Broyden's good method**:

$$B^{(k+1)} = B^{(k)} - (B^{(k)} s^{(k)} - y^{(k)}) \frac{s^{(k)T}}{s^{(k)T} s^{(k)}}$$

where

$$\begin{aligned} s^{(k)} &= x^{(k+1)} - x^{(k)}, \\ y^{(k)} &= g^{(k+1)} - g^{(k)}. \end{aligned}$$

Unquiz: Verify that Broyden's good method satisfies the secant equation and the no-change conditions. \square

Notes: For Broyden's good method,

- $B^{(k+1)}$ is formed from $B^{(k)}$ by adding a **rank-one matrix**.
- $B^{(k+1)}$ is not necessarily symmetric, even if $B^{(k)}$ is. This is undesirable since we know H is symmetric.

In order to regain symmetry, we need to sacrifice the no-change conditions. Instead, we formulate the problem in a **least change** sense:

$$\min_{B^{(k+1)}} \|B^{(k+1)} - B^{(k)}\|$$

subject to the secant condition

$$B^{(k+1)} s^{(k)} = y^{(k)}.$$

The solution (again) depends on the choice of norm.

A refinement to this idea: We can impose other constraints, too.

- Perhaps we know that H is sparse, and we want B to have the same structure.
- We might want to keep $B^{(k+1)}$ positive definite.

Some history

An alphabet soup of algorithms:

- DFP: Davidon 1959, Fletcher-Powell 1963
- BFGS: Broyden, Fletcher, Goldfarb, Shanno 1970
- Broyden's good method and Broyden's bad
- ...

An example: DFP

Still one of the most popular because it has many desirable properties.

The choice of norm in the minimization problem:

$$\|W^{1/2}(B^{(k+1)} - B^{(k)})W^{1/2}\|_F$$

where W^{-1} is a symmetric positive definite matrix satisfying the secant condition. (In fact, the solution is independent of the choice of such W .)

The resulting update formula: We accumulate an approximation C to H^{-1} rather than to H .

$$C^{(k+1)} = C^{(k)} - \frac{C^{(k)} y^{(k)} y^{(k)T} C^{(k)}}{y^{(k)T} C^{(k)} y^{(k)}} + \frac{s^{(k)} s^{(k)T}}{y^{(k)T} s^{(k)}}$$

An example: BFGS

The most popular method, and the one discussed in the text.

The choice of norm in the minimization problem is

$$\|W^{1/2}(C^{(k+1)} - C^{(k)})W^{1/2}\|_F$$

where W is a symmetric positive definite matrix satisfying the secant condition. (In fact, the solution is independent of the choice of such W .)

Note that this method is in some sense **dual** to DFP.

The resulting update formula:

$$B^{(k+1)} = B^{(k)} - \frac{B^{(k)} s^{(k)} s^{(k)T} B^{(k)}}{s^{(k)T} B^{(k)} s^{(k)}} + \frac{y^{(k)} y^{(k)T}}{y^{(k)T} s^{(k)}}$$

Use of Quasi-Newton methods

The algorithm looks very similar to Newton's method:

Until $x^{(k)}$ is a **good enough** solution,

Compute a search direction $p^{(k)}$ from $p^{(k)} = -C^{(k)}g^{(k)}$ (or solve $B^{(k)}p^{(k)} = -g^{(k)}$).

Set $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$, where α_k satisfies the Goldstein-Armijo or Wolfe linesearch conditions.

Form the updated matrix $C^{(k+1)}$ (or $B^{(k+1)}$).

Initialization: Now need to initialize $B^{(0)}$ (or $C^{(0)}$) as well as $x^{(0)}$. Take $B^{(0)} = I$ or some better guess.

Unanswered questions

- Near a stationary point, H^{-1} does not exist. How do we keep C from deteriorating?
- What happens if H is indefinite?
- How do we check optimality?

The first two questions concern stability of the algorithm.

Answering concerns about stability

Our dilemma:

- Updating C can be hazardous when H is close to singular.
- Updating B leaves the problem of solving a linear system at each iteration to determine the search direction.

An alternative: Update a factorization of B . This makes it easy to enforce symmetry and positive definiteness.

Example: BFGS This algorithm updates the approximate Hessian B . There are two economical alternatives for solving the necessary linear systems

1. Updating the inverse.

$$B^{(k+1)} = B^{(k)} - \frac{B^{(k)} s^{(k)} s^{(k)T} B^{(k)}}{s^{(k)T} B^{(k)} s^{(k)}} - \frac{y^{(k)} y^{(k)T}}{y^{(k)T} s^{(k)}}$$

so, by the Sherman-Morrison-Woodbury formula (N&S p643) for computing inverses of matrices updated by a rank-2 correction, we can obtain

$$B^{(k+1)^{-1}} = B^{(k)^{-1}} + \frac{y^{(k)T} (B^{-1(k)} y^{(k)} + s^{(k)})}{(y^{(k)T} s^{(k)})^2} s^{(k)} s^{(k)T} - \frac{s^{(k)} y^{(k)T} B^{-1(k)} + B^{-1(k)} y^{(k)} s^{(k)T}}{(y^{(k)T} s^{(k)})}$$

2. Updating a factorization. If we have a Cholesky factorization of $B^{(k)}$ as $B^{(k)} = L^{(k)} D^{(k)} L^{(k)T}$, then we want $B^{(k+1)} = L^{(k+1)} D^{(k+1)} L^{(k+1)T}$. This can be formed by formulas analogous to the Sherman-Morrison-Woodbury formula, and we can use the ideas in the previous set of notes to modify it to preserve positive definiteness. The algorithms are $O(n^2)$. Details are given, for example, in a paper by Gill, Golub, Murray, and Saunders, *Math. Comp.* 28 (1974) pp.505-535, and in some textbooks.

Answering concerns about convergence

It is an unfortunate fact that software packages do not check the second-order optimality conditions. They just return a point at which the gradient is approximately zero and they can find no descent direction or direction of negative curvature.

Desirable properties

1. $\|H^{(k)} - B^{(k)}\|$ should decrease as k increases. At least we would like this for quadratics.

This makes the method more Newton-like as the iteration proceeds and prevents false convergence.

For non-quadratics, we usually settle for **bounded deterioration**:

$$\|H(x^*) - B^{(k+1)}\| \leq \|H(x^*) - B^{(k)}\| + \kappa \sum_{j=1}^k \|p^{(j)}\|$$

where κ is some constant. This property helps in the proof of local convergence.

2. The update should not fail:
- no division by zero.
 - change should be nonnegligible.

Unquiz: Show that the DFP update can be written as

$$C^{(k+1)} = C^{(k)} M^{(k)}$$

for some matrix $M^{(k)}$. []

This property holds for all of the Quasi-Newton formulas, and therefore if $B^{(k)}$ ever becomes singular, all future B s remain singular. This is very bad:

we have a direction a so that $a^T C^{(k)} = 0$, so $a^T C^{(k+1)} = 0$ and we never search in the direction a .

In practice, we reset B to I every $2n$ iterations to avoid problems.

Note: When we have linear equality constraints, this singular behavior can be used to our advantage, to avoid walking off a constraint.

3. The algorithm should behave well on quadratic functions.

Newton on quadratics: terminates in 1 step.
Most Quasi-Newton formulas have n - or $n + 1$ -step termination.

Example: Huang's family of updates, also called the **Broyden class**, is defined near the bottom of p. 355 of N&S. It includes DFP and BFGS. Members of Huang's family have $B^{(n+1)} = H$. (Intermediate B s differ for the different methods, though.)

4. We should have **hereditary positive definiteness**. Many algorithms have this property.

Example: Symmetric rank-1 update (N&S p. 351) Let $B^{(k)} = R^T R$ be nonsingular. Then

$$\begin{aligned} B^{(k+1)} &= B^{(k)} + \frac{(y^{(k)} - B^{(k)}s^{(k)})(y^{(k)} - B^{(k)}s^{(k)})^T}{(y^{(k)} - B^{(k)}s^{(k)})^T s^{(k)}} \\ &= R^T \left(I + \frac{(\bar{y} - \bar{s})(\bar{y} - \bar{s})^T}{(y^{(k)} - B^{(k)}s^{(k)})^T s^{(k)}} \right) R \\ &\equiv R^T W R \end{aligned}$$

where $\bar{y} = R^{-T}y^{(k)}$ and $\bar{s} = Rs^{(k)}$.

How do we know that this is positive definite if $B^{(k)}$ is?

The matrix W has an eigenvalue

$$1 + \frac{(\bar{y} - \bar{s})^T (\bar{y} - \bar{s})}{(y^{(k)} - B^{(k)}s^{(k)})^T s^{(k)}}$$

with eigenvector $\bar{y} - \bar{s}$. Its other $n - 1$ eigenvalues are all ones, with eigenvectors perpendicular to $\bar{y} - \bar{s}$.

Now $B^{(k+1)}$ is positive definite if and only if W is, so we need that plum-colored eigenvalue to be positive. This is true, for example, if the denominator is positive, which means that

$$(y^{(k)} - B^{(k)} s^{(k)})^T s^{(k)} = (y^{(k)} + g^{(k)})^T s^{(k)} = g^{(k+1)T} s^{(k)} > 0.$$

And this is true if we overshoot on each line search. \square

Example: There is a proof of hereditary positive definiteness for BFGS, when the line search is accurate enough, in N&S, p.354. \square

Important theoretical property

Dixon 1971: All algorithms in the Huang family (including DFP and BFGS) produce identical iterates on quadratics when started at the same point with the same H , using exact line search and assuming that H remains nonsingular.

Practical significance....

Convergence rate

All of these methods have an n -, $2n$ -, or $(n + 2)$ -step quadratic convergence rate if the line search is exact. An n -step quadratic convergence rate, for example, means that

$$\lim_{k \rightarrow \infty} \frac{\|x^{(k+n)} - x^*\|}{\|x^{(k)} - x^*\|^2} < \infty.$$

Weakening the line search to a Wolfe or Goldstein-Armijo search generally gives superlinear convergence. See N&S p. 356 for a typical result for the Huang family.

No Hessian Method 2: Finite-difference Newton Method

Finite-difference Newton methods (N&S 11.4.1)

One of our major time-sinks in using Newton's method is to evaluate the Hessian matrix, with entries

$$h_{ij} = \frac{\partial g_i}{\partial x_j}.$$

One way to avoid these evaluations is to **approximate** the entries:

$$h_{ij} \approx \frac{g_i(x + \tau e_j) - g_i(x)}{\tau}$$

where τ is a small number and e_j is the j th unit vector.

Cost: n extra gradient evaluations per iteration. Sometimes this is less than the cost of the Hessian evaluation, but sometimes it is more.

Practical matters:

- How to choose τ .
 - If τ is large, the approximation is poor and we have large **truncation error**.
 - If τ is small, then there is cancellation error in forming the numerator of the approximation, so we have large **round-off error**.

Usually we try to balance the two errors by choosing τ to make them approximately equal.

- If the problem is poorly scaled, we may need a different τ for each j .

Convergence rate

There are theorems that say that if we choose τ carefully enough, we can get superlinear convergence.

Methods that require only first derivatives and store no matrices

Sometimes problems are too big to allow n^2 storage space for the Hessian matrix.

Some alternatives:

- steepest descent

- nonlinear conjugate gradient
- limited memory Quasi-Newton
- truncated Newton

Low-storage Method 1: Steepest Descent N&S 11.2

Back to that foggy mountain

If we walk in the direction of **steepest descent** until we stop going downhill, we clearly are guaranteed to get to a local minimizer.

The trouble is that the algorithm is terribly slow. See the examples in N&S.

If we apply steepest descent to a quadratic function of n variables, then after many steps, the algorithm takes alternate steps approximating two directions: those corresponding to the eigenvectors of the smallest and the largest eigenvalues of the Hessian matrix.

The convergence rate on quadratics is linear:

$$f(x^{(k+1)}) - f(x^*) \leq \left(\frac{\kappa - 1}{\kappa + 1} \right)^2 (f(x^{(k)}) - f(x^*))$$

where κ is the ratio of the largest to the smallest eigenvalue of H .

See N&S p.342 for proof.

If steepest descent is applied to non-quadratic functions, using a good line search, then convergence is local and linear.

Advantages of steepest descent:

- No need to evaluate the 2nd derivative or to solve a linear system.
- Low storage: no matrices.

Disadvantages of steepest descent:

- **Very** slow.
- **Very, very** slow.

Use **nonlinear conjugate gradients** instead. Same advantages but better convergence.

**Low-storage Method 2: Nonlinear conjugate gradient methods
(N&S 12.4)**

Linear conjugate gradients

The conjugate gradient method (Hestenes&Stiefel, 1952) is a method for solving linear systems of equations $Ax = b$ when A is symmetric and positive definite.

There are many ways to understand it, but for us, we think of it as minimizing the function

$$f(x) = \frac{1}{2}x^T Ax - x^T b$$

which has gradient $g(x) = Ax - b$. So a minimizer of f is a solution to our linear system.

We could use steepest descent, but we want something faster.

See the notes on the linear algorithm to understand how conjugate gradient combines the concepts of **descent** and **conjugate directions**.

Summary of the linear conjugate gradient algorithm

Given $x^{(0)}$, form $-g(x^{(0)}) = b - Ax^{(0)} = p^{(0)}$.

For $k = 0, 1, \dots$, until convergence,

Use a line search to determine $x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$.

Set $p^{(k+1)} = -g(x^{(k+1)}) + \beta^{(k+1)} p^{(k)}$.

The nonlinear conjugate gradient algorithm

The parameter β has many definitions that are equivalent for the linear problem but different when we minimize nonlinear functions:

$$\begin{aligned}\beta^{(k+1)} &= \frac{g^{(k+1)T} g^{(k+1)}}{g^{(k)T} g^{(k)}} && \text{Fletcher-Reeves} \\ \beta^{(k+1)} &= \frac{y^{(k)T} g^{(k+1)}}{g^{(k)T} g^{(k)}} && \text{Polak-Ribière} \\ \beta^{(k+1)} &= \frac{y^{(k)T} g^{(k+1)}}{y^{(k)T} p^{(k)}} && \text{Hestenes-Stiefel}\end{aligned}$$

Good theorems have been proven about convergence of Fletcher-Reeves, but Polak-Ribière is generally better performing.

Note that this method stores no matrix. We only need to remember a few vectors at a time, so it can be used for problems in which there are thousands or millions of variables.

The convergence rate is linear, unless the function has special properties, but generally faster than steepest descent: for quadratics, the rate is

$$f(x^{(k+1)}) - f(x^*) \leq \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^2 (f(x^{(k)}) - f(x^*))$$

where κ is the ratio of the largest to the smallest eigenvalue of H .

An important property of conjugate gradients:

If we run cg on a quadratic function, then it generates the same iterates as the Huang family of Quasi-Newton algorithms.

Low-storage Method 3: Limited-memory Quasi-Newton methods (N&S 12.5)

Consider as an example the DFP formula

$$C^{(k+1)} = C^{(k)} - \frac{C^{(k)} y^{(k)} y^{(k)T} C^{(k)}}{y^{(k)T} C^{(k)} y^{(k)}} + \frac{s^{(k)} s^{(k)T}}{y^{(k)T} s^{(k)}}$$

We've thought of this as a matrix stored in memory. Let's develop a different view.

How do we use this matrix? All we need to do is to multiply vectors, such as the gradient $g^{(k)}$, by it.

Suppose we store $C^{(0)}$. Then if we stored $C^{(0)}s^{(0)}$, $y^{(0)}$, and $s^{(0)}$, we would be able to form products of $C^{(1)}$ with any vector:

$$C^{(1)}z = C^{(0)}z - \frac{y^{(0)T}C^{(0)}z}{y^{(0)T}C^{(0)}y^{(0)}}C^{(0)}y^{(0)} - \frac{s^{(0)T}z}{y^{(0)T}s^{(0)}}s^{(0)}.$$

We can play this game again for $C^{(2)}$: if we store $(C^{(0)}y^{(0)}, y^{(0)}, s^{(0)})$ and $(C^{(1)}y^{(1)}, y^{(1)}, s^{(1)})$, then we can form products of $C^{(2)}$ with any vector.

What have we accomplished? Recall that $C^{(0)}$ is usually chosen to be the identity matrix, which requires no storage. Therefore, instead of taking n^2 storage locations for $C^{(2)}$, we only need $6n$!

The idea behind **limited memory quasi-Newton algorithms** is to continue this process ℓ steps, until we don't want to store any more vectors. Then, we start storing the new vectors in place of the oldest ones that we remember, always keeping the ℓ most recent updates.

For various more refined strategies, see reference 4 at the end of these notes.

Low-storage Method 4: Truncated Newton methods (N&S 12.3)

Again we return to the way the Hessian approximation is used.

If we have the Hessian matrix, how do we use it? Newton's method determines the search direction by solving the linear system

$$Hp = -g$$

We usually think of solving this by factoring H and then using forward- and back- substitution.

But if H is large, this might be too expensive, and we might choose to use an iterative method, like **linear conjugate gradients**, to solve the system.

If we do, **How do we use the Hessian?** All we need to do is to multiply a vector by it at each step of the algorithm.

Now Taylor series tells us that, if v is a vector of length 1, then

$$g(x + hv) = g(x) + hH(x)v + O(h^2),$$

so

$$H(x)v = \frac{g(x + hv) - g(x)}{h} + O(h).$$

Therefore, we can get an $O(h)$ approximation of the product of the Hessian with an arbitrary vector by taking a **finite difference approximation** to the change in the gradient in that direction.

This is akin to the finite-difference Newton method, but much neater, because we only evaluate the finite difference in directions in which we need it.

The Truncated Newton strategy

So we'll compute an approximation to the Newton direction $p = -H^{-1}g$ by solving the linear system $Hp = -g$ using the conjugate gradient method, computing **approximate** matrix-vector products by extra evaluations of the gradient.

We hope to obtain a superlinear convergence rate, so we need, by the theorem of the previous set of notes (or N&S p304), that

$$\frac{\|\text{our direction} - \text{Newton direction}\|}{\|\text{our direction}\|} \rightarrow 0$$

as the iteration number $\rightarrow \infty$.

We ensure this by

- taking enough iterations of conjugate gradient to get a small residual to the linear system.
- choosing h in the approximation carefully, so that the matrix- vector products are accurate enough.

Interesting proof in the problems in N&S.

Automated differentiation

Automated differentiation (N&S 11.4.2)

The most tedious and error-prone part of nonlinear optimization:
writing code for derivatives.

An alternative: let the computer do it.

Automatic differentiation is an old idea:

- The **forward** (bottom-up) algorithm was proposed in the 1970s.
- The **backward** (top-down) algorithm was proposed in the 1980s.
- Reliable software (AdiFor, etc., by Bischof, Griewank, ...) was developed in the 1990s.

For clarity, we'll discuss first derivatives.

Forward mode

Example: Let $f(x, y) = (\sin(x/y) + x/y - \exp(y)) * (x/y - \exp(y))$.

Consider how this function would be evaluated on a computer. The operations form a tree:

Now imagine that at each node of that tree we carry not just the function value but also the two partial derivatives. The **chain rule** from calculus gives us the means to evaluate a partial derivative for a cost proportional to that for evaluating the function.

Costs for the gradient:

- Time = $O(n)$ * function evaluation time.
- Space = $O(n)$.

[]

Backward mode

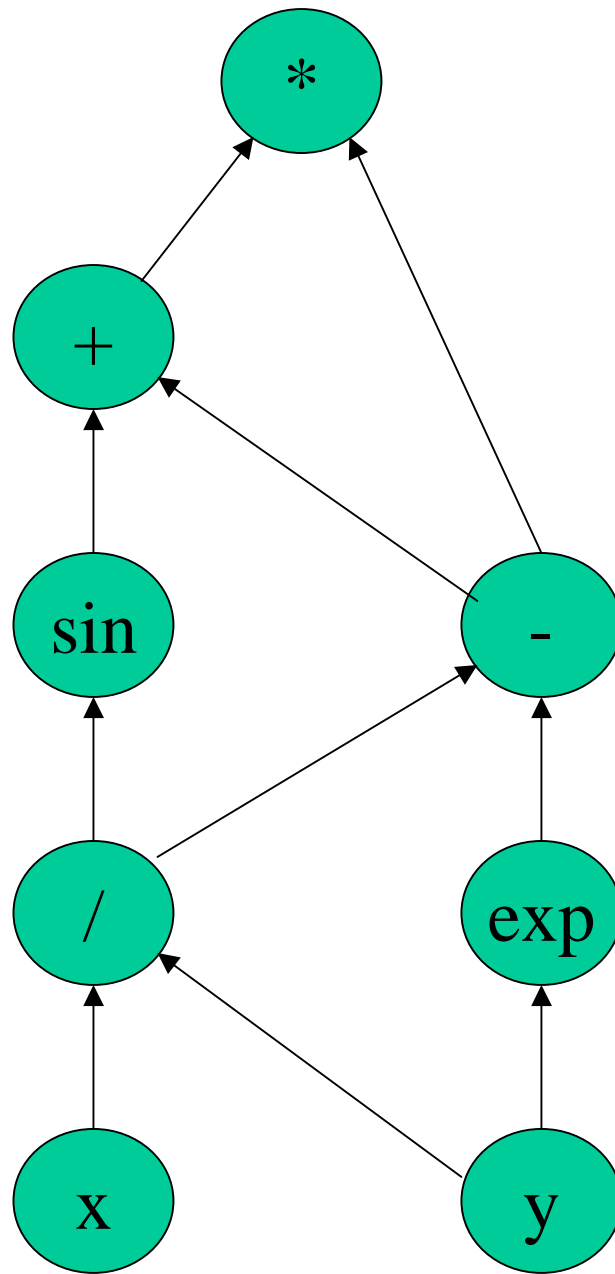


Figure 1: Example: The forward mode of automatic differentiation

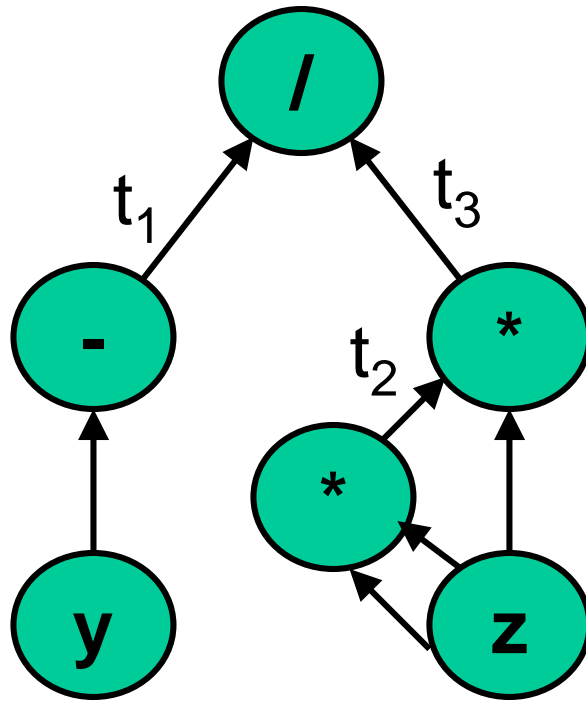


Figure 2: Example: The backward mode of automatic differentiation

Example: (See N&S for a different example.) Let $w(z, y) = -y/(z * z * z)$.

For ease of notation, for a variable q , let $\bar{q} \equiv \partial w / \partial q$. (This is sometimes known as the **adjoint**.)

We want to compute \bar{z} and \bar{y} to obtain the gradient of w .

We accumulate these derivatives recursively, so initially we set all “barred” quantities to zero.
insert figure here.

Rules for the backward mode:

- If $s = f(t)$, then we can add a new contribution to \bar{t} :

$$\begin{aligned} \frac{\partial w}{\partial t} &= \text{currentterms} + \frac{\partial w}{\partial s} \frac{\partial s}{\partial t} \\ \frac{\partial w}{\partial t} &= \text{currentterms} + \frac{\partial w}{\partial s} \frac{\partial f}{\partial t} \end{aligned}$$

so

$$\bar{t}_+ = \bar{s} * \frac{\partial f}{\partial t}.$$

- Similarly, if $s = f(t, u)$, then

$$\begin{aligned} \bar{t}_+ &= \bar{s} * \frac{\partial f}{\partial t}, \\ \bar{u}_+ &= \bar{s} * \frac{\partial f}{\partial u}. \end{aligned}$$

We illustrate this for our example:

1. $t_1 = -y$.
2. $t_2 = z * z$.
3. $t_3 = t_2 * z$.
4. $w = t_1 / t_3$, so
5. $\bar{t}_1 = 1 / t_3$ and
6. $\bar{t}_3 = -t_1 / t_3^2$.
7. (3) $\rightarrow \bar{t}_2 = \bar{t}_3 * z$ and $\bar{z} = \bar{t}_3 * t_2$.

8. $(2) \rightarrow \bar{z}_+ = \bar{t}_2 * 2z.$

9. $(1) \rightarrow \bar{y}_+ = \bar{t}_1 * (-1).$

All operations are binary or unary, so the number of instructions to compute the derivatives is **at most** $2 * \text{the number to compute the function}.$

Thus the backward mode computes all of the partial derivatives in time at most double the time for evaluating the function - independent of the number of variables!

The drawback: The algorithm can use **a lot** of memory.

Practical automatic differentiation

In practice, we use a combination of the forward mode, to keep storage low, and the backward mode, to keep operations counts low.

Good programs make good decisions about when to use each mode.

Hessians

The automatic algorithms to compute second derivatives can take advantage of two extra tricks:

- Exploit any sparsity. Suppose H has many zeros, because components of the gradient do not depend on all of the variables:

Example:

$$H = \begin{bmatrix} x & x & 0 & 0 \\ x & 0 & 0 & x \\ 0 & 0 & 0 & x \\ 0 & x & x & 0 \end{bmatrix}$$

Note that the first and third columns have no nonzeros in common, and similarly for the second and fourth.

Therefore, if we compute the partial derivative of the gradient with respect to $x_1 + x_3$, we find all of the nonzero entries in columns 1 and 3 of the Hessian.

Similarly, we can determine the second and fourth columns by computing the partial with respect to $x_2 + x_4$.

Practical programs for automatic differentiation do this by allowing you to specify that you want to compute HJ rather than H , where

$$J = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

□

- Sometimes we can eliminate interior nodes of the computational graph in order to save storage. The tricks to do this are the same as for doing sparse Gauss elimination, and we won't discuss them here.

No-derivative methods (N&S 11.4.3)

These are **methods of last resort**, generally used when

- derivatives are not available.
- derivatives do not exist.

This is an area of very active research currently. We'll consider three classes of methods.

No-derivative Method 1: Finite difference methods

We could do finite differences on the function to get an approximate gradient, but this is not usually a good idea, given that automatic differentiation methods exist.

No-derivative Method 2: Simplex-based methods

(Not to be confused with the simplex method for linear programming.)

The most popular of these is the **Nelder-Meade algorithm**, and Matlab has an implementation of this.

Idea:

- Suppose we have evaluated the function at the vertices of a **simplex**. (In 2-dimensions, this is a triangle, in 3, it is a tetrahedron, etc.)
- We would like to move one vertex of this simplex, reflecting it around its current position, until we have enclosed the minimizer in the simplex.
- Then we would like to shrink the size of the simplex to hone in on the minimizer.

Simplex-based algorithms have rather elaborate rules for determining when to reflect and when to shrink, and no algorithm that behaves well in practice has a good convergence proof.

For that reason, it looks as if they will fade in popularity, being supplanted by **pattern search methods**.

No-derivative Method 3: Pattern search methods

Idea:

- Suppose we are given an initial guess x at the solution and a set of at least $n + 1$ directions $v_i, i = 1, \dots, N$, that form a **positive basis** for \mathcal{R}^n : this means that any vector can be expressed as a linear combination of these vectors, where the coefficients in the combination are positive numbers.
- At each step, we do a line search in each of the directions to obtain $f(x + \alpha_i v_i)$ and replace x by the point with the smallest function value.

This is a remarkably simple algorithm, but works well in practice and is **provably** convergent!

Another desirable property is that it is easy to parallelize, and this is crucial to making a no-derivative algorithm effective when n is large.

Example: Tamara Kolda software. []

Some personal references

1. Dianne P. O'Leary, "Conjugate gradient algorithms in the solution of optimization problems for nonlinear elliptic partial differential equations," *Computing* 22 (1979) 59-77.

2. Dianne P. O'Leary, "A discrete Newton algorithm for minimizing a function of many variables," *Mathematical Programming* 23 (1982) 20-33. **This algorithm came to be known as "truncated Newton".**
3. Dianne P. O'Leary, "Why Broyden's nonsymmetric method terminates on linear equations," *SIAM Journal on Optimization*, 5 (1995) 231-235.
4. Tamara Kolda, Dianne P. O'Leary, and Larry Nazareth, "BFGS with Update Skipping and Varying Memory," *SIAM Journal on Optimization*, 8 (1998) 1060-1083. <http://epubs.siam.org/sam-bin/dbq/article/30645>