Numerical Optimization

General Setup

• Let $f(.)$ be a function such that

$$x \in R^n \rightarrow f(\boldsymbol{b}, x) \in R$$

where $\boldsymbol{b}$ is a vector of unknown parameters. In many cases, $\boldsymbol{b}$ will not have a closed form solution. We will estimate $\boldsymbol{b}$ by minimizing some loss function.

• Popular loss function: A sum of squares.

• Let $\{y_i, x_i\}$ be a set of measurements/constraints. That is, we fit $f(.)$, in general a nice smooth function, to the data by solving:

$$\min_b \{ S(\mathbf{b}) = \sum_{i=1}^{T} e_i^{\,2} = \sum_{i=1}^{T} (y_i - f(x_i, \mathbf{b}))^2 \}$$

Finding a Minimum - Review

• When $f(.)$ is twice differentiable, a local minima is characterized by:

1. $\nabla f(\boldsymbol{b}_{min}) = 0$ (f.o.c.)
2. $\boldsymbol{z}' \boldsymbol{H}(\boldsymbol{b}_{min})\, \boldsymbol{z} \geq 0$ (s.o.c.)

Usual approach: Solve $\nabla f(\boldsymbol{b}_{min}) = 0$ for $\boldsymbol{b}_{min}$
Check $\boldsymbol{H}(\boldsymbol{b}_{min})$ is positive definitive.

• Sometimes, an analytical solution to $\nabla f(\boldsymbol{b}_{min}) = 0$ is not possible or hard to find.

• For these situations, a numerical solution is needed.

Notation: $\nabla f(.)$: gradient (of a scalar field).
∇: Vector differential operator ("*del*").

Finding a Univariate Minimum

• Finding an analytic minimum of a simple univariate sometimes is easy given the usual f.o.c. and s.o.c.

Example: Quadratic Function

$$U(x) = 5x^2 - 4x + 2$$

$$\frac{\partial U(x)}{\partial x} = 10x^* - 4 = 0 \quad \Rightarrow \quad x^* = \frac{2}{5}$$

$$\frac{\partial^2 U(x)}{\partial x^2} = 10 > 0$$

Analytical solution (from f.o.c.): $x^* = 0.4$.

The function is globally convex $\Rightarrow x^*=2/5$ is a global minimum.

Note: To find x^*, we find the zeroes of the first derivative function.

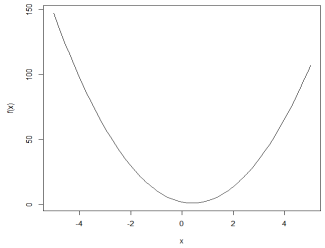
Finding a Univariate Minimum

• Minimization in R:

We can use the popular R optimizer, *optim*, to find the minimum of *U(x)* –in this case, not very interesting application!

Example: Code in R

```
> f <- function(x) (5*x^2-4*x+2)
> optim(10, f, method="Brent", lower=-10, upper=10)$par
[1] 0.4
> curve(f, from=-5, to=5)
```



Finding a Univariate Minimum

• Straightforward transformations add little additional complexity:

$$U(x) = e^{5x^2-4x+2}$$

$$\frac{\partial U(x)}{\partial x} = U(x^*)\left[10x^* - 4\right] = 0 \quad \Rightarrow \quad x^* = \frac{2}{5}$$

• Again, we get an analytical solution from the f.o.c. $\Rightarrow x^*=2/5$.

Since $U(.)$ is globally convex, $x^* = 2/5$ is a global minimum.

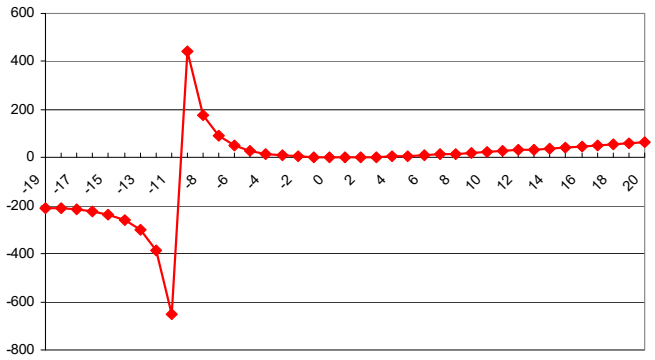
• Usual Problems: Discontinuities, Unboundedness.

Finding a Univariate Minimum - Discontinuity

- Be careful of discontinuities. Need to restrict range for x.

  Example:

$$f(x) = \frac{5x^2 - 4x + 2}{x + 10}$$



Finding a Univariate Minimum - Discontinuity

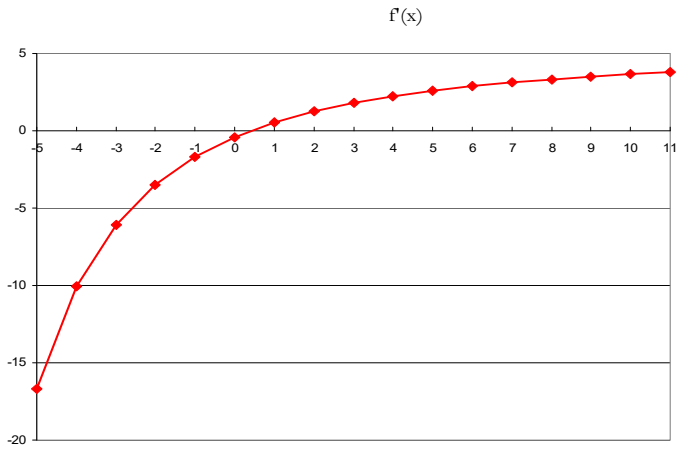
- This function has a discontinuity at some point in its range. If we restrict the search to points where x >-10, then the function is defined at all points

$$\frac{\partial f(x)}{\partial x} = f'(x) = \frac{(10x^* - 4)(x^* + 10) - [5x^{*2} - 4x^* + 2]}{(x^* + 10)^2} = 0$$

$$x^* = \frac{-100 + 2\sqrt{2710}}{10} = 0.411532 \ .$$

- After restricting the range of x, we find an analytical solution as usual –i.e., by finding the zeroes of $f'(x)$.

Finding a Univariate Minimum - Unbounded



Finding a Univariate Minimum – No analytical Solution

• So far, we have presented examples were an analytical solution or close form solution was easy to obtain from the f.o.c.

• In many situations, finding an analytical solution is not possible or impractical. For example:

$$f(x) = x^6 - 4x^5 - 2x^3 + 2x + 40$$

$$f(x) = \sin(x) + (1/6)x^4$$

• In these situations, we rely on numerical methods to find a solutions. Most popular numerical methods are based on iterative methods. These methods provide an approximation to the exact solution, x^*.

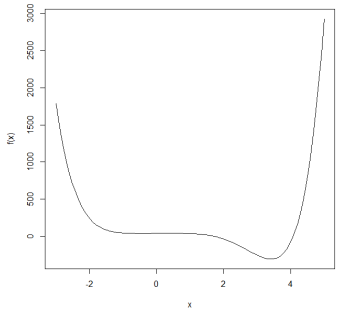
• We will concentrate on the details of these iterative methods.

Numerical solutions: Iterative algorithm

Example: We want to minimize $f(x) = x^6 - 4x^5 - 2x^3 + 2x + 40$.

We use the R flexible minimizer, Optim.

```
> curve(f, from=-3, to=5)
> f <- function(x) (x^6-4*x^5-2*x^3+2*x+40)
> optim(0.5, f, method="Brent", lower=-10,
upper=10)$par
[1] 3.416556
> curve(f, from=-3, to=5)
```



We get the solution $x^* = 3.416556$

• Optim offers many choices to do the iterative optimization. In our example, we used the method Brent, a mixture of a bisection search and a secant method. We will discuss the details of both methods in the next slides.

Numerical solutions: Iterative algorithm

• Old method: There is a *Babylonian method.*

• Iterative algorithms provide a sequence of approximations $\{x_0, x_1, \ldots, x_n\}$ that in the limit converge to the exact solution, x^*.

• Computing x_{k+1} from x_k is called *one iteration* of the algorithm.

• Each iteration typically requires one evaluation of the objective function f (or f and f') evaluated at x_k.

• An algorithm needs a *convergence criterion* (or, more general, a *stopping criterion*). For example: Stop algorithm if

$$|f(x_{k+1}) - f(x_k)| < \text{pre-specified tolerance}$$

Iterative algorithm

• Goal of algorithm:

Produce a sequence $\{\boldsymbol{x}\}=\{x_0, x_1, \ldots, x_n\}$ such that

$$f(x_0) > f(x_1) > \ldots > f(x_n)$$

• Algorithm: Sketch

– Start at an initial position x_0, usually an *initial guess*.

– Generate a sequence $\{\boldsymbol{x}\}$, which hopefully convergence to x^*.

– $\{\boldsymbol{x}\}$ is generated according to an iteration function g, such that $x_{k+1} = g(x_k)$.

• Algorithm: Speed

The speed of the algorithm depends on:

– the cost (in flops) of evaluating $f(x)$ (and, likely, $f'(x)$).

– the number of iterations.

Iterative algorithm

• Characteristics of algorithms:

- We need to provide a subroutine to compute f and, likely, f' at x.

- The evaluation of f and f' can be expensive. For example, it may require a simulation or numerical integration.

• Limitations of algorithms

- There is no algorithm that guarantees finding *all* solutions.

- Most algorithms find at most one (local) solution.

- Need prior information from the user: an initial guess, an interval that contains a zero, etc.

Speed of algorithm

• Suppose $x_k \to x^*$ with $f(x^*) = 0$ –i.e., we find the roots of *f*.
Q: How fast does x_k go to x^*?

• Error after *k* iterations:
- absolute error: $|x_k - x^*|$
- relative error: $|x_k - x^*| / |x^*|$ (defined if $x^* \neq 0$)

• The number of correct digits is given by:

$$-\log_{10} [|x_k - x^*| / |x^*|]$$

(when it is well defined –i.e., $x^* \neq 0$ and $[|x_k - x^*| / |x^*|] \leq 1$).

Speed of algorithm – Rates of Convergence

• Rates of convergence of a sequence x_k with limit x^*.
- *Linear convergence*: There exists a $c \in (0, 1)$ such that

$$|x_{k+1} - x^*| \leq c\ |x_k - x^*| \text{ for sufficiently large } k.$$

- *R-linear convergence*: There exists $c \in (0, 1)$, $M > 0$ such that

$$|x_k - x^*| \leq M c^k \text{ for sufficiently large } k.$$

- *Quadratic convergence*: There exists a $c > 0$ such that

$$|x_{k+1} - x^*| \leq c\ |x_k - x^*|^2 \text{ for sufficiently large } k.$$

- *Superlinear convergence*: There exists a sequence c_k, with $c_k \to 0$ s.t.

$$|x_{k+1} - x^*| \leq c_k\ |x_k - x^*| \text{ for sufficiently large } k.$$

Speed of algorithm – Interpretation

• Assume $x^* \neq 0$.

Let $r_k = -\log_{10} [|x_k - x^*|/|x^*|]$ – $r_k \approx$ the number of correct digits at iteration k.

- *Linear convergence:* We gain roughly $-\log_{10} c$ correct digits per step:

$$r_{k+1} \geq r_k - \log_{10} c$$

- *Quadratic convergence*: For sufficiently large k, the number of correct digits roughly doubles in one step:

$$r_{k+1} \geq -log_{10}(c \mid x^*|) + 2 r_k$$

- *Superlinear convergence:* Number of correct digits gained per step increases with *k:*

$$r_{k+1} - r_k \rightarrow \infty$$

Speed of algorithm – Examples

• Let $x^* = 1$.

The number of correct digits at iteration k, r_k, can be approximated by: $r_k = -\log_{10} [|x_k - x^*|/|x^*|]$

• We define 3 sequences for x_k with different types of convergence:

1. *Linear convergence:* $x_{k+1} = 1 + 0.5^k$
2. *Quadratic convergence*: $x_{k+1} = 1 + (0.5^2)^k$
3. *Superlinear convergence:* $x_{k+1} = 1 + (1/(k+1))^k$

• For each sequence we calculate x_k for $k = 0, 1, ..., 10$.

Speed of algorithm – Examples

k	$1 + 0.5^k$	$1 + 0.5^{2^k}$	$1 + (1/(k+1)^k)$
0	2.00000000000000	1.50000000000000	2.00000000000000
1	1.50000000000000	1.25000000000000	1.50000000000000
2	1.25000000000000	1.06250000000000	1.11111111111111
3	1.12500000000000	1.00390625000000	1.01562500000000
4	1.06250000000000	1.00001525878906	1.00160000000000
5	1.03125000000000	1.00000000023283	1.00012860082305
6	1.01562500000000	1.00000000000000	1.00000849985975
7	1.00781250000000	1.00000000000000	1.00000047683716
8	1.00390625000000	1.00000000000000	1.00000002323057
9	1.00195313125000	1.00000000000000	1.00000000100000
10	1.00097656250000	1.00000000000000	1.00000000003855

• Sequence 1: we gain roughly $-log_{10}(c)$ = 0.3 correct digits per step

$$|x_{k+1} - 1| / |x_k - 1| = 2^k/2^{k+1} = 0.5 \leq c \quad (c = 0.5)$$

• Sequence 2: r_k almost doubles at each step

• Sequence 3: r_k per step increases slowly but it gets up to speed later.

$$|x_{k+1} - 1| / |x_k - 1| = (k+1)^k / (k+2)^{k+1} \rightarrow 0$$

Convergence Criteria

• An iterative algorithm needs a stopping rule. Ideally, it is stopped because the solution is sufficiently accurate.

• Several rules to check for accuracy:

– *X vector criterion:* $|x_{k+1} - x_k|$ < tolerance

– *Function criterion:* $|f(x_{k+1}) - f(x_k)|$ < tolerance

– *Gradient criterion:* $|\nabla f(x_{k+1})|$ < tolerance

• If none of the convergence criteria is met, the algorithm will stop by hitting the stated maximum number of iterations. This is not a convergence!

• If a very large number of iterations is allowed, you may want to stop the algorithm if the solution diverges and or cycles.

Numerical Derivatives

• Many methods rely on the first derivative: $f'(x)$.

Example: Newton's method's requires $f'(x_k)$ and $f''(x_k)$

Newton's method algorithm: $x_{k+1} = x_k - \lambda_k f'(x_k) / f''(x_k)$

• It is best to use the analytical expression for $f'(x)$. But, it may not be easy to calculate and/or expensive to evaluate. In these situations it may be appropriate to approximate $f'(x)$ numerically by using the difference quotient:

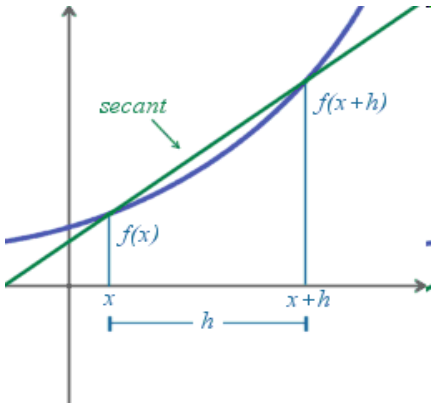
$$f'(x_k) = \frac{f(x_{k,2}) - f(x_{k,1})}{x_{k,2} - x_{k,1}} = \frac{f(x_k + h) - f(x_k)}{h}$$

• Then, pick a small h and compute f' at x_k using:

$$f'(x_k) = \frac{f(x_k + h) - f(x_k)}{h}$$

Numerical Derivatives – Graphical View

• Approximation errors will occur. For example, using the traditional definition, we have that the secant line differs from the slope of the tangent line by an amount that is approximately proportional to h.



Numerical Derivatives – 2-point Approximation

- We can approximate $f'(x_k)$ with another two-point approximation:

$$f'(x_k) = \frac{[f(x_k+h)-f(x_k)]+[f(x_k)-f(x_k-h)]}{2h} = \frac{f(x_k+h)-f(x_k-h)}{2h}$$

• Errors may cancel out on both sides. For small values of h this is a more accurate approximation to the tangent line than the one-sided estimation.

• Q: What is an appropriate h?

Floating point considerations (and cancellation errors) point out to choose an h that's not too small. On the other hand, if h is too large, the first derivative will be a secant line. A popular choice is sqrt(ε)x, where ε is of the order 10^{-16}. If working with returns (in %), h=.000001 is fine.

Source of errors – Computation

• We use a computer to execute iterative algorithms. Recall that iterative algorithms provide a sequence of approximations that in the limit converge to the exact solution, x^*. Errors will occur.

(1) *Round-off errors:* A practical problem because a computer cannot represent all $x \in R$ exactly.

Example: Evaluate two expression for same function at x=0.0002:

$f(x) = [1 - \cos^2(x)]/x^2$ & $g(x) = [\sin^2(x)]/x^2$

$f(x = 0.0002) = 0.99999998$ & $g(x = 0.0002) = 0.99999999$

(2) *Cancelation errors: a* and *b* should be the same (or almost the same), but errors during the calculations makes them different. When we subtract them the difference is no longer zero.

Source of errors – Truncation and Propagation

(3) *Truncation errors*: They occurs when the iterative method is terminated, usually after a convergence criterion is met.

(4) *Propagation of errors*: Once an error is generated, it will propagate. This problem can arise in the calculation of standard errors, CI, etc.

Example: When we numerically calculate a second derivative, we use the numerically calculated first derivate. We potentially have a compounded error: an approximation based on another approximation.

Source of errors – Data

• *Ill-conditioned problem*

This is a data problem. Any small error or change in the data produce a large error in the solution. No clear definition on what "large error" means (absolute or relative, norm used, etc.)

Usually, the *condition number* of a function relative to the data (in general, a matrix $\mathbf{X}$), $\kappa(\mathbf{X})$, is used to evaluate this problem. A small $\kappa(\mathbf{X})$, say close to 1, is good.

Example: A solution to a set of linear equations, $\mathbf{A}\boldsymbol{x} = \mathbf{b}$

Now, we change $\mathbf{b}$ to $(\mathbf{b} + \Delta\mathbf{b})$ $\Rightarrow$ new solution is $(\boldsymbol{x} + \Delta\boldsymbol{x})$, which satisfies $\mathbf{A}(\boldsymbol{x} + \Delta\boldsymbol{x}) = (\mathbf{b} + \Delta\mathbf{b})$

- The change in $\boldsymbol{x}$ is $\Delta\boldsymbol{x} = \mathbf{A}^{-1}\,\Delta\mathbf{b}$

- We say, the equations are *well-conditioned* if small $\Delta\mathbf{b}$ results in small $\Delta\boldsymbol{x}$. (The condition of the solution depends on $\mathbf{A}$.)

Source of errors - Algorithm

• *Numerical stability*

It refers to the accuracy of an algorithm in the presence of small errors, say a round-off error. Again, no clear definition of "accurate" or "small."

Examples: A small error grows during the calculation and significantly affects the solution. A small change in initial values can produce a different solution.

Line Search

- Line search techniques are simple optimization algorithms for one-dimensional minimization problems.
- Considered the backbone of non-linear optimization algorithms.
- Typically, these techniques search a bracketed interval.
- Often, unimodality is assumed.
- Usual steps:
  - Start with *bracket* $[x_L, x_R]$ such that the minimum x^* lies inside.
  - Evaluate $f(x)$ at two points inside the bracket.
  - Reduce the bracket in the *descent direction (f*↓).
  - Repeat the process.

- Note: Line search techniques can be applied to any function and differentiability and continuity is not essential.

Line Search

- Basic Line Search (Exhaustive Search)

  1. Start with a $[x_L, x_R]$ and divide it in *T* intervals.

  2. Evaluate f in each interval (*T* evaluations of f). Choose the interval where f is smaller.

  3. Set $[x_L, x_R]$ as the endpoints of the chosen interval => Back to 1.

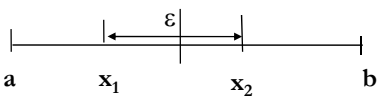
  4. Continue until convergence.

- Key step: Sequential reduction in the brackets. The fewer evaluations of functions, the better for the line search algorithm.

  - There are many techniques to reduce the brackets:
    - Dichotomous –i.e., *divide in 2 parts*- search
    - Fibonacci series
    - Golden section

Line Search: Dichotomous Bracketing

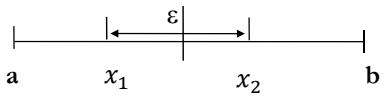
- Very simple idea: Divide the bracket in half in each iteration.
- Start at the middle point of the interval [a, b]: (a+b)/2. Then evaluate $f(.)$ at two points near the middle (x_1 & x_2). "Near": $\pm$ ε/2.
- Then, evaluate the function at x_1 and x_2 . Move the interval in the direction of the lower value of the function.



- The interval size after 1 iteration (& 2 *f* evaluations):

$$|b_1 - a_1| = (1/2)\ |b_0 - a_0 + \varepsilon|$$

Line Search: Dichotomous Bracketing - Steps



• Steps:

0) Assume an interval [a,b]

1) Find $x_1 = a + (b - a)/2 - \varepsilon/2$

$x_2 = a+(b - a)/2 + \varepsilon/2$ (ε: is the resolution)

2) Compare $f(x_1)$ and $f(x_2)$

3) If $f(x_1) < f(x_2)$ then eliminate $x > x_2$ and set b = x_2

If $f(x_1) > f(x_2)$ then eliminate $x < x_1$ and set a = x_1

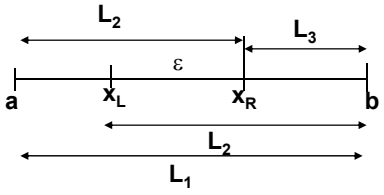
If $f(x_1) = f(x_2)$ then pick another pair of points

4) Continue until interval < 2 ε (tolerance)

Line Search: Fibonacci numbers

• Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, ... That is , the sum of the last 2 numbers: $F_n = F_{n-1} + F_{n-2}$ ($F_0 = F_1 = 1$)

• The Fibonacci sequence becomes the basis for choosing sequentially *N* points such that the discrepancy $x_{k+1} - x_{k-1}$ is minimized.



• Q: How is the interval reduced? Start at final interval, after K iterations and go backwards: $|b_{K-1} - a_{K-1}| = 2|b_K - a_K|$;

$|b_{K-2} - a_{K-2}| = 3|b_K - a_K|$; ... ; $|b_{K-J} - a_{K-J}| = F_{J+1}|b_K - a_K|$

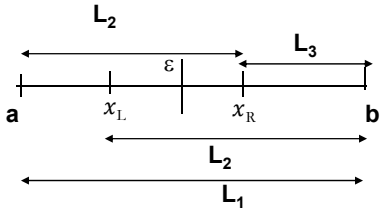
$\Rightarrow |b_k - a_k| = |b_{k-2} - a_{k-2}| - |b_{K-1} - a_{K-1}|$

Line Search: Fibonacci numbers

Let ε = b – a. Then, evaluate *f(x)* at

$x_L = a + F_{k-2}(b-a) / F_k$ and $x_R = a + F_{k-1}(b-a) / F_k$

We keep the smaller functional value and its corresponding opposite end–point.



Example: $f(x) = x/(1+x^2)$, $x \in [-0.6, 0.75]$

$x_L = -.6+(.75+.6)*1/3 = -0.15 \Rightarrow f1 = -0.1467$

$x_R = -.6+(.75+.6)*2/3 = 0.30 \Rightarrow f2 = 0.2752$

New $x \in [-0.6, 0.30]$

$x_L = -.6+(.3+.6)*2/5 = -0.24 \Rightarrow f1 = -0.2267$

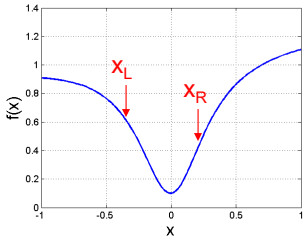
$x_R = -.6+(.3+.6)*3/5 = -0.06 \Rightarrow f2 = -0.0598$

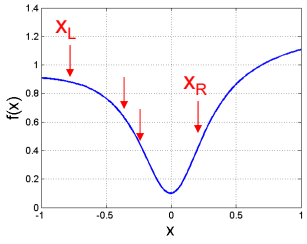
New $x \in [-.6, -0.06]$

Continue until bracket < tolerance.

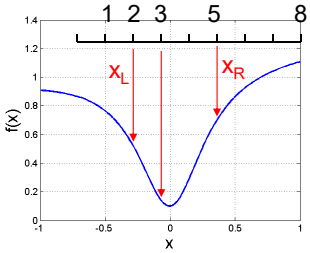
$x^* = -0.59997$.

Search methods - Examples



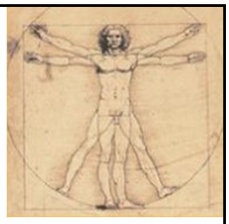


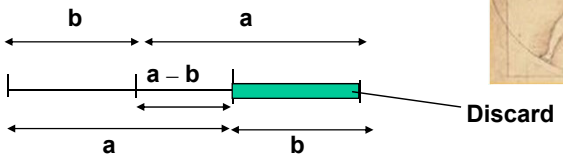
Dichotomous



Fibonacci: 1 1 2 3 5 8 …

Line Search: Golden Section





• In Golden Section, you try to have b/(a – b) = a/b,

$\Rightarrow$ b*b = a*a – ab

Solving gives a = (b ± b* sqrt(5))/2

$\Rightarrow$ a/b = -0.618 or 1.618 (Golden Section ratio)

• Note that 1/1.618 = 0.618

Note: $\lim_{n\to\infty} F_{n-2}/F_n = .382$ and $\lim_{n\to\infty} F_{n-1}/F_n = .618$

• Golden section method uses a *constant* interval reduction ratio: 0.618.

Line Search: Golden Section - Example

Initialize:

x1 = a + (b – a) * 0.382

x2 = a + (b – a) * 0.618

f1 = f(x1)

f2 = f(x2)

Loop:

if f1 > f2 then

  a = x1; x1 = x2; f1 = f2

  x2 = a + (b – a) * 0.618

  f2 = f(x2)

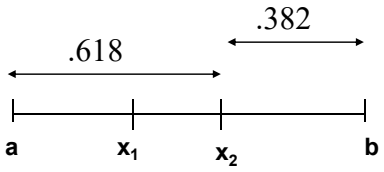
else

  b = x2; x2 = x1; f2 = f1

  x1 = a + (b – a) * 0.382

  f1 = f(x1)

endif



Example: $f(x) = x/(1+x^2)$, $x \in [-0.6, 0.75]$

x_L=-.6+(.75+.6)*.382=-0.0843 $\Rightarrow f1$=-0.0837

x_R=-.6+(.75+.6)*.618=0.2343 $\Rightarrow f2$=0.2221

New $x \in$ [-0.6, 0.234]

x_L=-.6+(.2343+.6)*.382=-0.2813 $\Rightarrow f1$=-0.2607

x_R=-.6+(.2343+.6).*618=-0.0844 $\Rightarrow f2$=-0.0838

New $x \in$ [-.6, -0.0844]

Continue until bracket < tolerance.

x^* = -0.59997.

Pure Line Search - Remarks

- Since only f is evaluated, a line search is also called *0^{th} order method*.
- Line searches work best for unimodal functions.
- Line searches can be applied to any function. They work very well for discontinuous and non-differentiable functions.
- Very easy to program.
- Robust.
- They are less efficient than higher order methods –i.e., methods that evaluate f' and f"
- Golden section is very simple and tends to work well (in general, fewer evaluations than most line search methods).

Line Search: Bisection

• The bisection method involves shrinking an interval $[a, b]$ known to contain the root (zero) of the function, using f' ($\Rightarrow$ a *1st order method*).

• *Bisection:* Interval is replaced by either its left or right half at each *k*.

• Method:

- Start with an interval $[a, b]$ that satisfies $f'(a)\ f'(b) < 0$. Since f is continuous, the interval contains at least one solution of $f(x) = 0$.

- In each iteration, evaluate f' at the midpoint of the interval $[a, b]$, $f'((a+b)/2)$.

- Reduce interval: Depending on the sign f', we replace a or b with the midpoint value, $(a+b)/2$. (The new interval still satisfies $f'(a)\ f'(b) < 0$.)

• Note: After each iteration, the interval $[a, b]$ is halved:

$$|a_k - b_k| = (1/2)^k\ |a_0 - b_0|$$

Line Search: Bisection - Example

Example: $f(x)=\frac{5x^2-4x+2}{x+10}$

- We restrict the range of x to x >-10.

- Steps:

(1) Start with interval $[a=-8,\ b=20]$. The bisection of that interval is 6. At $x=6, f'(x=6)=2.882>0 \quad \Rightarrow f$ increasing.

(1a) Exclude subinterval [6, 20] from search: New interval $[a=-8,\ b=6]$. Bisection at -1.

(2) New interval [-8, 6]. At $f'(x=-1)$=-1.6914 < 0 $\Rightarrow f$ decreasing

(3) New interval becomes $[a=-1,\ b=6]$. Bisection at 2.5.

- Continue until interval < ε (tolerance)

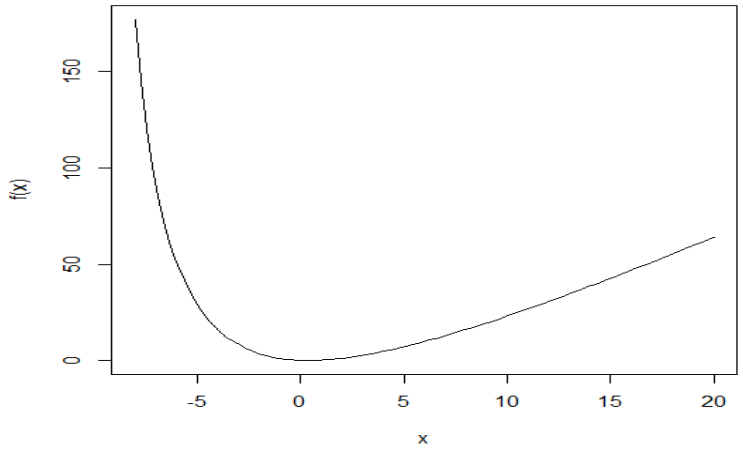
• Since at each iteration, the interval is halved. Then,

$|x_k-x^*| \le (1/2)^k \mid a_0-b_0|$ $\quad \Rightarrow$ R-linear convergence.

Line Search : Bisection – Example

• Graph of following function:

$$f(x)=\frac{5x^2-4x+2}{x+10}$$



Line Search : Bisection – Example

	Lower	Upper	Midpoint	Gradient	Replace
0	-8.0000	20.0000	6.0000	2.8828	Upper
1	-8.0000	6.0000	-1.0000	-1.6914	Lower
2	-1.0000	6.0000	2.5000	1.5312	Upper
3	-1.0000	2.5000	0.7500	0.3099	Upper
4	-1.0000	0.7500	-0.1250	-0.5581	Lower
5	-0.1250	0.7500	0.3125	-0.0965	Lower
6	0.3125	0.7500	0.5313	0.1130	Upper
7	0.3125	0.5313	0.4219	0.0099	Upper
8	0.3125	0.4219	0.3672	-0.0429	Lower
9	0.3672	0.4219	0.3945	-0.0164	Lower
10	0.3945	0.4219	0.4082	-0.0032	Lower
11	0.4082	0.4219	0.4150	0.0034	Upper
12	0.4082	0.4150	0.4116	0.0001	Upper
13	0.4082	0.4116	0.4099	-0.0016	Lower
14	0.4099	0.4116	0.4108	-0.0007	Lower

- We can calculate the required k for convergence, say with $\varepsilon = 0.002$:

$$\log[(a_0 - b_0)\,|\,/\varepsilon)]\;[1/\log(2)] = \log(28/.002)/\log(2) = 13.773 \approx 14$$

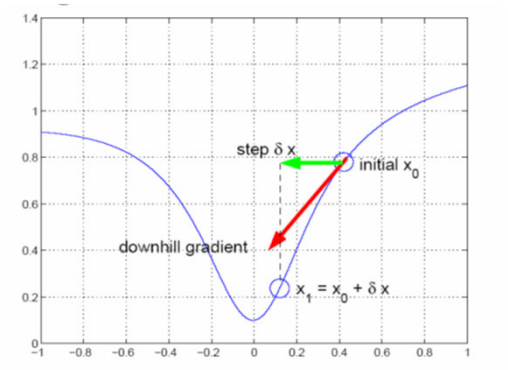
Line Search : Bisection – Example in R

```
bisect <- function(fn, lower, upper, tol=.0002, ...) {
f.lo <- fn(lower, ...)
f.hi <- fn(upper, ...)
feval <- 2
if (f.lo * f.hi > 0) stop("Root is not bracketed in the specified interval\n")
chg <- upper - lower

while (abs(chg) > tol) {
      x.new <- (lower + upper) / 2
      f.new <- fn(x.new, ...)
      if (abs(f.new) <= tol) break
      if (f.lo * f.new < 0) upper <- x.new
      if (f.hi * f.new < 0) lower <- x.new
      chg <- upper - lower
      feval <- feval + 1 }
list(x = x.new, value = f.new, fevals=feval)
}

# Example => function to minimize = (a*x^2-4*x+2)/(x+10)
# first derivative => fn1= (2*a*x-4)/(x+10)-(a*x^2-4*x+2)/(x+10)^2
fn1 <- function(x, a) {
(2*a*x-4)/(x+10)-(a*x^2-4*x+2)/(x+10)^2 }
bisect(fn1, -8, 20, a=5)
```

Descent Methods

• Typically, the line search pays little attention to the direction of change of the function –i.e., $f'(x)$.

• Given a starting location, x_0, examine $f(x)$ and move in the *downhill* direction to generate a new estimate, $x_1 = x_0 + \delta x$.



• Q: How to determine the step size δx? Use the first derivative: $f'(x)$.

Steepest (Gradient) descent

- Basic algorithm:
  1. Start at an initial position x_0
  2. Until convergence
     - Find minimizing step δx_k, which will be a function of $f'(x)$.
     - $x_{k+1} = x_k + \delta x_k$

• In general, it pays to multiply the direction δx_k by a constant, λ, called *step-size*. That is,

$$x_{k+1} = x_k + \lambda\, \delta x_k$$

• To determine the direction δx_k, note that a small change proportional to $f'(x)$multiplied by (-1) decreases $f(x)$. That is,

$$x_{k+1} = x_k - \lambda\, f'(x)$$

Steepest descent – Theorem

• **Theorem**: Given a function f: $R^n \to R$, differentiable at $\boldsymbol{x}_0$, the direction of steepest descent is the vector $-\nabla f(\boldsymbol{x}_0)$.

Proof: Consider the function

$$z(\lambda) = f(\boldsymbol{x}_0 + \lambda\, \boldsymbol{u}) \qquad (\boldsymbol{u}\text{: unit vector } \|\boldsymbol{u}\|=1)$$

By chain rule:

$$z'(\lambda) = \frac{\partial f}{\partial x_1}\frac{dx_1}{\partial \lambda} + \frac{\partial f}{\partial x_2}\frac{dx_2}{\partial \lambda} + \ldots + \frac{\partial f}{\partial x_n}\frac{dx_n}{\partial \lambda}$$

$$= \frac{\partial f}{\partial x_1}u_1 + \frac{\partial f}{\partial x_2}u_2 + \ldots + \frac{\partial f}{\partial x_n}u_n = \nabla f(\mathbf{x_0} + \lambda \mathbf{u}) \cdot \mathbf{u}$$

Thus, $z'(0) = \nabla f(\boldsymbol{x}_0) \cdot \boldsymbol{u} = \|\nabla f(\boldsymbol{x}_0)\|\ \|\boldsymbol{u}\| \cos(\theta)$ (dot product)

$= \|\nabla f(\boldsymbol{x}_0)\| \cos(\theta)$ (θ: angle between $\nabla f(\boldsymbol{x}_0)$ & $\boldsymbol{u}$)

Then, $z'(0)$ is minimized when $\theta = \pi$ $\Rightarrow$ $z'(0) = -\|\nabla f(\boldsymbol{x}_0)\|$.

Since $\boldsymbol{u}$ is a unit vector ($\|\boldsymbol{u}\|=1$) $\Rightarrow \boldsymbol{u} = -\nabla f(\boldsymbol{x}_0)/\|\nabla f(\boldsymbol{x}_0)\|$

Steepest descent – Algorithm

• Since $\boldsymbol{u}$ is a unit vector $\Rightarrow \boldsymbol{u} = -\nabla f(\boldsymbol{x}_0)/\|\nabla f(\boldsymbol{x}_0)\|$

$\Rightarrow$ The method becomes the *steepest descent*, due to Cauchy (1847).

• The problem of minimizing a function of N variables can be reduced to a single variable minimization problem, by finding the minimum of $z(\lambda)$ for this choice of $\mathbf{u}$:

$\Rightarrow$ Find λ that minimizes $z(\lambda) = f(\boldsymbol{x}_k - \lambda\, \nabla f(\boldsymbol{x}_k))$

• Algorithm: 1. Start with $\boldsymbol{x}_0$. Evaluate $\nabla f(\boldsymbol{x}_0)$.

2. Find λ_0 and set $\boldsymbol{x}_1 = \boldsymbol{x}_0 - \lambda_0 \nabla f(\boldsymbol{x}_0)$

3. Continue until convergence.

• Sequence for $\{\boldsymbol{x}\}$: $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \lambda_k \nabla f(\boldsymbol{x}_k)$

Steepest descent – Algorithm

Note: Two ways to determine λ_k, in the general $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \lambda_k \nabla f(\boldsymbol{x}_k)$

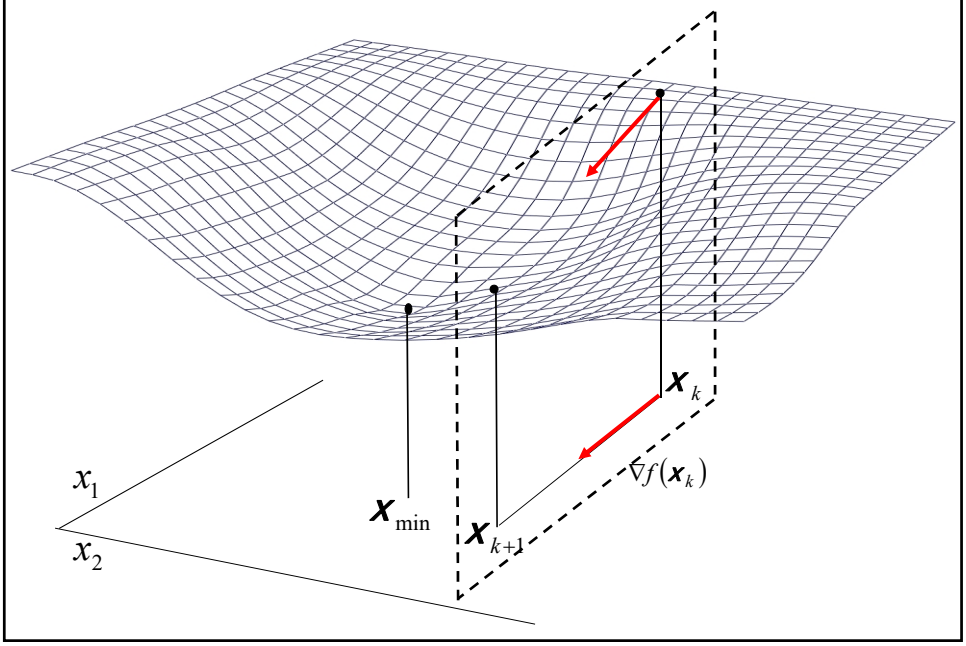
1. Optimal λ_k. Select λ_k that minimizes $g(\lambda_k) = f(\boldsymbol{x}_k - \lambda_k \nabla f(\boldsymbol{x}_k))$.
2. Non-optimal λ_k can be calculated using line search methods.

• Good property: The method of steepest descent is guaranteed to make at least some progress toward x^* during each iteration. (Show that $z'(0) < 0$, which guarantees there is a $\lambda > 0$, such that $z(\lambda) < z(0)$.)

• It can be shown that the steepest descent directions from $\boldsymbol{x}_{k+1}$ and $\boldsymbol{x}_k$ are orthogonal, that is $\nabla f(\boldsymbol{x}_{k+1}) \cdot \nabla f(\boldsymbol{x}_k) = 0$.

For some functions, this property makes the method to "*zig-zag*" (or *ping pong)* from $\boldsymbol{x}_0$ to $\boldsymbol{x}^*$.

Steepest descent – Graphical Example



Steepest descent – Limitations

• Limitations:

- In general, a global minimum is not guaranteed. (This issue is also a fundamental problem for the other methods).

- Steepest descent can be relatively slow close to the minimum: going in the steepest downhill direction is not efficient. Technically, its asymptotic rate of convergence is inferior to many other methods.

- For poorly conditioned convex problems, steepest descent increasingly 'zigzags' as the gradients point nearly orthogonally to the shortest direction to a minimum point.

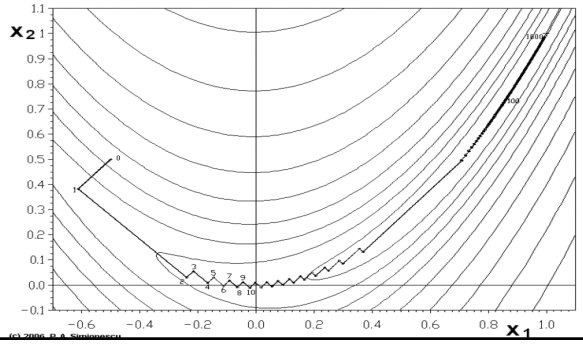
-

Steepest descent – Example (Wikipedia)

• Steepest descent has problems with pathological functions such as the Rosenbrock function.

Example: $f(x_1, x_2) = (1 - x_1)^2 + 100\,(x_2 - x_1^2)^2$

This function has a narrow curved valley, which contains x^*. The bottom of the valley is very flat. Because of the curved flat valley the optimization zig-zags slowly with small step-sizes towards $x^*=(1, 1)$.



Newton-Raphson Method (Newton's method)

• It is a method to iteratively find roots of functions (or a solution to a system of equations, $f(\mathbf{x}) = \mathbf{0}$).

• Idea: Approximate $f(\boldsymbol{x})$ near the current guess $\boldsymbol{x}_k$ by a function $f_k(\mathbf{x})$ for which the system of equations $f_k(\mathbf{x})=0$ is easy to solve. Then use the solution as the next guess $\boldsymbol{x}_{k+1}$.

• A good choice for $f_k(\mathbf{x})$ is the linear approximation of $f(\boldsymbol{x})$ at $\boldsymbol{x}_k$:

$$f_k(\mathbf{x}) \approx f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)\,(\boldsymbol{x} - \boldsymbol{x}_k).$$

• Solve the equation $f_k(\boldsymbol{x}_{k+1})=0$, for the next $\boldsymbol{x}_{k+1}$. Setting $f_k(\mathbf{x})=0$:

$$0 = f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)\,(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k).$$

Or $$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - |\nabla f(\boldsymbol{x}_k)|^{-1} f(\boldsymbol{x}_k)$$ (NR iterative method)

Newton-Raphson Method – History

• History:

- Heron of Alexandria (10–70 AD) described a method (called *Babylonian method*) to iteratively approximate a square root.

- François Viète (1540-1603) developed a method to approximate roots of polynomials.

- Isaac Newton (1643–1727) in 1669 (published in 1711) improved upon Viète's method. A simplified version of Newton's method was published by Joseph Raphson (1648–1715) in 1690. Though, Newton (and Raphson) did not see the connection between his method and calculus.

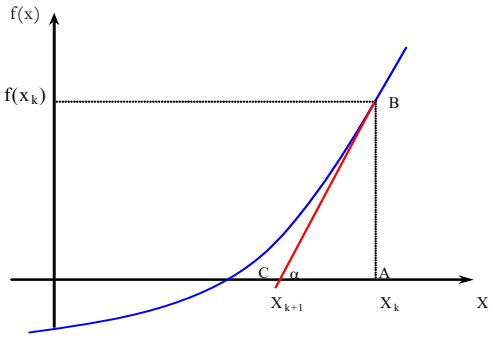
- The modern treatment is due to Thomas Simpson (1710–1761).

Newton-Raphson Method – Univariate Version

• We want to find the roots of a function $f(x)$. We start with:

$$\Delta y \cong \frac{\partial f(x)}{\partial x}\Delta x \quad \Rightarrow \quad \Delta x = \frac{\Delta y}{f'(x)}$$

$$\Rightarrow \quad x_{k+1} = x_k + \frac{f(x_{k+1}) - f(x_k)}{f'(x)} = x_k - \frac{0 - f(x_k)}{f'(x)} = x_k - \frac{f(x)}{f'(x)}$$



• Alternative derivation:

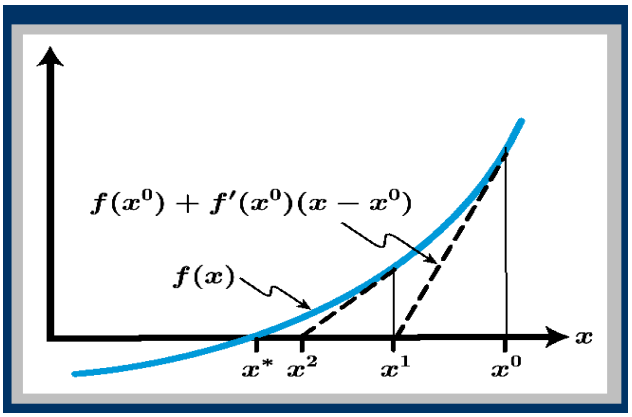
$$\tan(\alpha) = \frac{AB}{AC}$$

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

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

Newton-Raphson Method – Graphical View

• Iterations: Starting at x^0 "inscribe triangles" along $f(x) = 0$.



Note: At each step, we need to evaluate f and f'.

Newton-Raphson Method – Minimization

• We can use NR method to minimize a function.

• Recall that $f'(x^*) = 0$ at a minimum or maximum, thus stationary points can be found by applying NR method to the derivative. The iteration becomes:

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

• We need $f''(x_k) \neq 0$; otherwise the iterations are undefined. Usually, we add a step-size, λ_k, in the updating step of x :

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

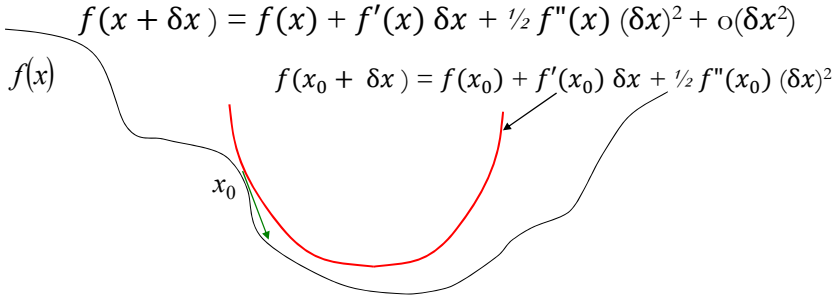
• Note: NR uses information from the second derivative. This information is ignored by the *steepest descent* method. But, it requires more computations.

NR Method – As a Quadratic Approximation

• When used for minimization, the NR method approximates $f(x)$ by its quadratic approximation near x_k.

• Expand $f(x)$ locally using a 2nd-order Taylor series:

$$f(x + \delta x) = f(x) + f'(x)\,\delta x + \tfrac{1}{2} f''(x)\,(\delta x)^2 + o(\delta x^2)$$



• Find the δx which minimizes this local quadratic approximation:

$$\delta x = -f'(x) \,/\, f''(x)$$

• Update x : $\quad x_{k+1} = x_k - \lambda_k \frac{f'(x_k)}{f''(x_k)}$

NR Method – *N*-variate Version

• The *N*-variate version of the NR method is based on the 1st-order Taylor series expansion of $\nabla f(\boldsymbol{x}) = 0$:

$$\nabla_x f(x) = \nabla_x f(x^*) + \nabla^2_{xx} f(x^*)(x - x^*) = 0$$

$$(x - x^*) = \left(\nabla^2_{xx} f(x^*)\right)^{-1} \nabla_x f(x^*)$$

Then, $$x_{t+1} = x_t - \left(\nabla^2_{xx} f(x_t)\right)^{-1} \nabla_x f(x_t) = x_t - H^{-1} \nabla_x f(x_t)$$

• It is always convenient to add a step-size λ_k.

• Algorithm: 1. Start with $\boldsymbol{x}_0$. Evaluate $\nabla f(\boldsymbol{x}_0)$ and $\mathbf{H}(\boldsymbol{x}_0)$.

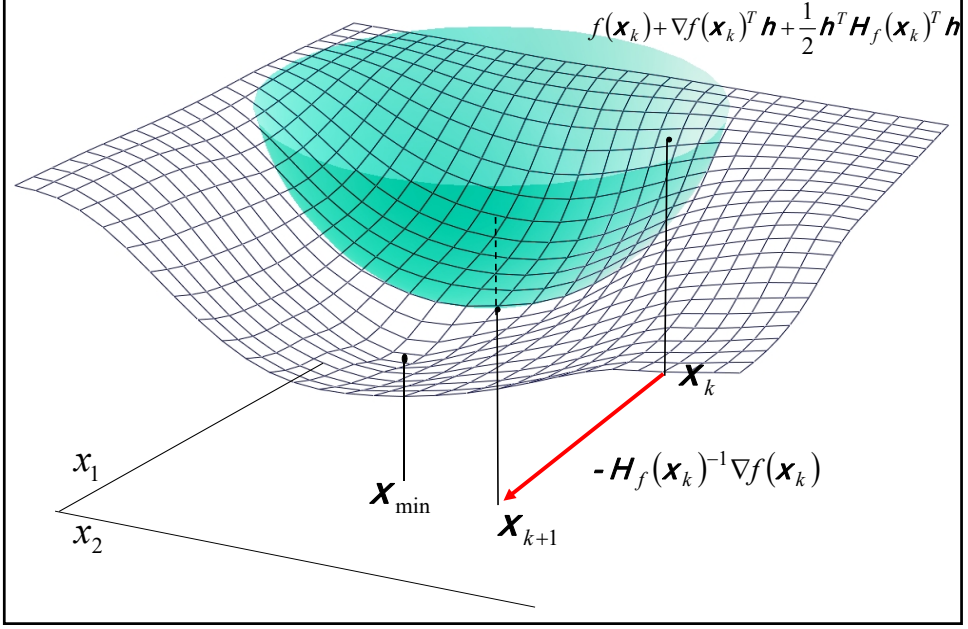
2. Get $\boldsymbol{x}_1 = \boldsymbol{x}_0 - \mathbf{H}(\boldsymbol{x}_0)^{-1} \nabla f(\boldsymbol{x}_0)$ $\Rightarrow f(\boldsymbol{x}_1, \lambda=1)$

3. Add λ, $\boldsymbol{x}_1(\lambda) = \boldsymbol{x}_0 - \lambda \mathbf{H}(\boldsymbol{x}_0)^{-1} \nabla f(\boldsymbol{x}_0)$ $\Rightarrow f(\boldsymbol{x}_1, \lambda)$

Pick λ, say λ_1, that min $f(\boldsymbol{x}_1, \lambda)$. $\Rightarrow \boldsymbol{x}_0 = \boldsymbol{x}_1(\lambda_1)$

4. Continue until convergence.

NR Method – Bivariate Version - Graph



NR Method – Properties

• If $\mathbf{H}(\boldsymbol{x}_k)$ is pd, the critical point is also guaranteed to be the unique strict global minimizer of $f_k(\mathbf{x})$.

• For quadratic functions, NR method applied to $f(\mathbf{x})$ converges to $\mathbf{x}$ in one step; that is, $\boldsymbol{x}_{k=1} = \boldsymbol{x}^*$.

• If $f(\mathbf{x})$ is not a quadratic function, then NR method will generally not compute a minimizer of $f(\mathbf{x})$ in one step, even if its $\mathbf{H}(\boldsymbol{x}_k)$ is pd. But, in this case, NR method is guaranteed to make progress.

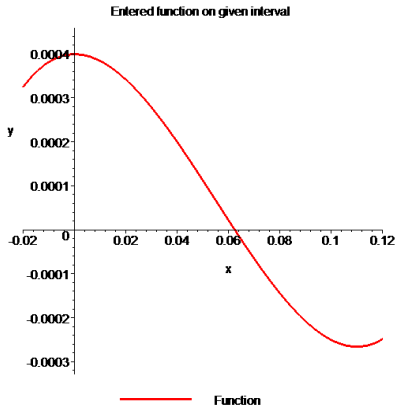
• Under certain conditions, the NR method has quadratic convergence, given a sufficiently close initial guess.

NR Method – Example

Find the roots for the following function:

$$f(x) = x^3 - 0.165\,x^2 + 3.993 \times 10^{-4}$$

Set x_0=.05.
Use the Newton's method of finding roots of equations to find
a) The root.
b) The absolute relative approximate error at the end of each iteration.
c) The number of significant digits at least correct at the end of each iteration.



NR Method – Example

• Calculate $f'(x)$

$$f(x) = x^3 - 0.165\,x^2 + 3.993 \times 10^{-4}$$
$$f'(x) = 3x^2 - 0.33\,x$$

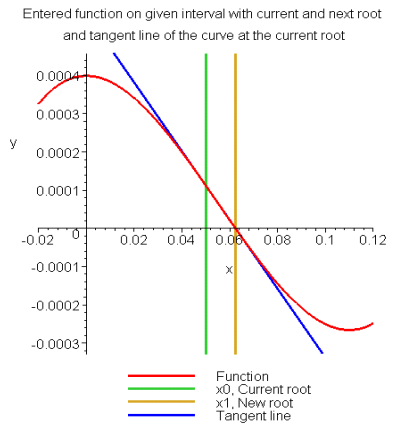
• Iterations: $x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)} = x_k - \dfrac{x_k^3 - 0.165\,x_k^2 + 3.993 \times 10^{-4}}{3x_k^2 - 0.33\,x_k}$

1) Iteration 1 ($x_0 = .05$)

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 0.05 - \frac{(0.05)^3 - 0.165(0.05)^2 + 3.993 \times 10^{-4}}{3(0.05)^2 - 0.33(0.05)}$$
$$= \mathbf{0.05} - \frac{1.118 \times 10^{-4}}{-9 \times 10^{-3}} = \mathbf{0.05} - (-0.01242) = 0.06242$$

NR Method – Example

Estimate of the root and abs relative error for the first iteration.



The absolute relative approximate error $|\varepsilon_a|$ at the end of Iteration 1 is

$$|\epsilon_a| = \left|\frac{x_1 - x_0}{x_1}\right| \times 100$$
$$= \left|\frac{0.06242 - 0.05}{0.06242}\right| \times 100$$
$$= 19.90\,\%$$

Number of significant digits at least correct: 0. (Suppose we use as stopping rule $|\varepsilon_a| < 0.05\%$).

NR Method – Example

2) Iteration 2 ($x_1 = .06242$)

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 0.06242 - \frac{(0.06242)^3 - 0.165(0.06242)^2 + 3.\mathbf{993}\times 10^{-4}}{3(0.06242)^2 - 0.33(0.06242)}$$

$$= 0.06242 - \frac{-3.97781\times 10^{-7}}{-8.90973\times 10^{-3}} = 0.06242 - \left(4.4646\times 10^{-5}\right) = 0.06238$$

Asolute relative approximate error $|\varepsilon_\alpha|$ at the end of Iteration 2 is:

$$|\in_a| = \left|\frac{x_2 - x_1}{x_2}\right| \times 100 = \left|\frac{0.06238 - 0.06242}{0.06238}\right| \times 100 = 0.0716\ \%$$

Number of significant digits at least correct: 2.

NR Method – Example

3) Iteration 3 ($x_2 = .06238$)

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$$

$$= 0.06238 - \frac{(0.06238)^3 - 0.165(0.06238)^2 + 3.993\times 10^{-4}}{3(0.06238)^2 - 0.33(0.06238)}$$

$$= 0.06238 - \frac{4.44\times 10^{-11}}{-8.91171\times 10^{-3}}$$

$$= 0.06238 - \left(-4.9822\times 10^{-9}\right) = 0.06238$$

Absolute relative approximate error $|\varepsilon_\alpha|$ at the end of Iteration 3 is:

$$|\in_a| = \left|\frac{x_2 - x_1}{x_2}\right| \times 100 = \left|\frac{0.06238 - 0.06238}{0.06238}\right| \times 100 = 0\ \%$$

Number of significant digits at least correct: 4. ($|\varepsilon_\alpha| < .05\% \Rightarrow$ stop).

64

NR Method – Example (Code in R)

```
> # Newton Raphson Method
>
> f1<-function(x){
+   return(x^3 - .165*x^2 + .0003993);
+ }
>
> df1<-function(x){
+   return(3*x^2-.33*x+.0003993);
+ }
> NR.method<-function(func,dfunc,x){
+   if (abs(dfunc(x))<.000001){
+     return (x);
+     }else{
+     return(x - func(x)/dfunc(x));
+ }
+ }
> curve(f1, -.05,.15)
>
> iter=20;
> xn <- NULL;
> xn[1] = .05;
> n = 1;
>
```

```
> while(n<iter){
+   n = n+1
+   xn<-c(xn,NR.method(f1,df1,xn[n-1]));
+   if(abs(xn[n]-xn[n-1])<.000001) break;
+ }
> xn
[1] 0.05000000 0.06299894 0.06234727 0.06237900
0.06237751 0.06237758
```

65

NR Method – Convergence

• Under certain conditions, the NR method has quadratic convergence, given a sufficiently close initial guess.

$$0 = f(x^*) = f(x^k) + \frac{df(x^k)}{dx}(x^* - x^k) + \underbrace{\frac{d^2 f(\tilde{x})}{dx^2}(x^* - x^k)^2}$$

Mean Value theorem truncates Taylor series

But $0 = f(x^k) + \frac{df(x^k)}{dx}(x^{k+1} - x^k)$ (*) by Newton definition

Subtracting (*) from above: $\frac{df(x^k)}{dx}(x^{k+1} - x^*) = \frac{d^2 f(\tilde{x})}{dx^2}(x^k - x^*)^2$

Then, solving for $(x^{k+1} - x^*)$:

$$(x^{k+1} - x^*) = [\frac{df}{dx}(x^k)]^{-1} \frac{d^2 f}{d^2 x}(\tilde{x})(x^k - x^*)^2$$

NR Method – Convergence

• From $(x^{k+1} - x^*) = [\frac{df}{dx}(x^k)]^{-1} \frac{d^2 f}{d^2 x}(\tilde{x})(x^k - x^*)^2$

Let $\left|[\frac{df}{dx}(x^k)]^{-1} \frac{d^2 f}{d^2 x}(\tilde{x})\right| = K^k$

then $\left|x^{k+1} - x^*\right| \le K^k \left|x^k - x^*\right|^2$ $\Rightarrow$ Convergence is quadratic.

• Local Convergence Theorem

If *a)* $\frac{df}{dx}$ *bounded away from zero*

b) $\frac{d^2 f}{dx^2}$ *bounded*

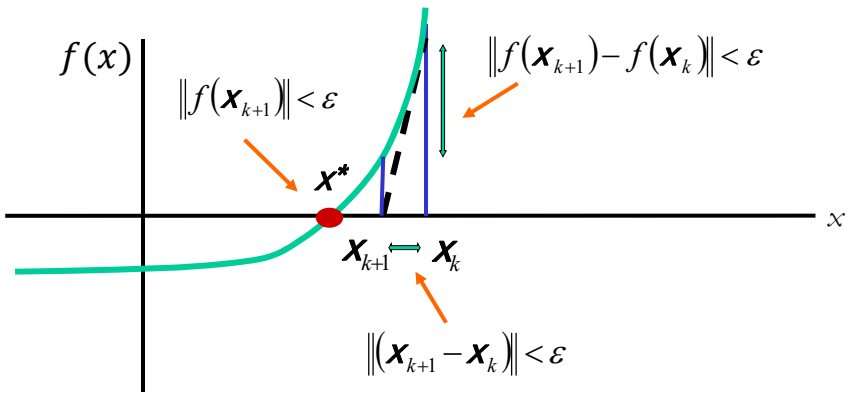
} K *is bounded*

Then, NR method converges given a *sufficiently close* initial guess (and convergence is quadratic).

NR Method – Convergence

• Convergence Check

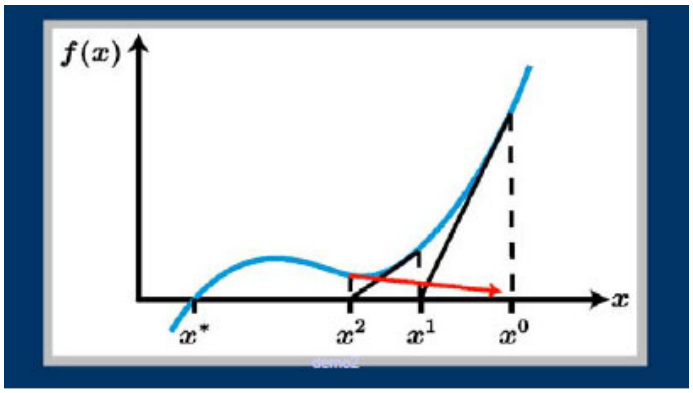
Check $f(x)$ to avoid false convergence. Check more than one criterion.



NR Method – Convergence

• Local Convergence
Convergence depends on initial values. Ideally, select $\boldsymbol{x}_0$ close to $\boldsymbol{x}^*$.

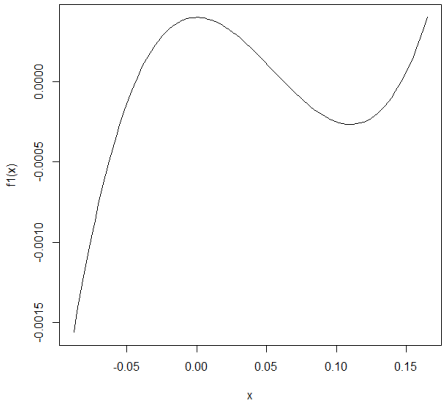
Note: Inflection point or flat regions can create problems.



NR Method: Limitations – Local results

• Multiple roots: Different initial values take us to different roots.

Example: $f(x) = x^3 - 0.165\,x^2 + 3.993 \times 10^{-4}$



• For $x_0 =$ 0.05,
> xn
[1] 0.05000000 0.06299894 0.06234727
0.06237900 0.06237751 0.06237758

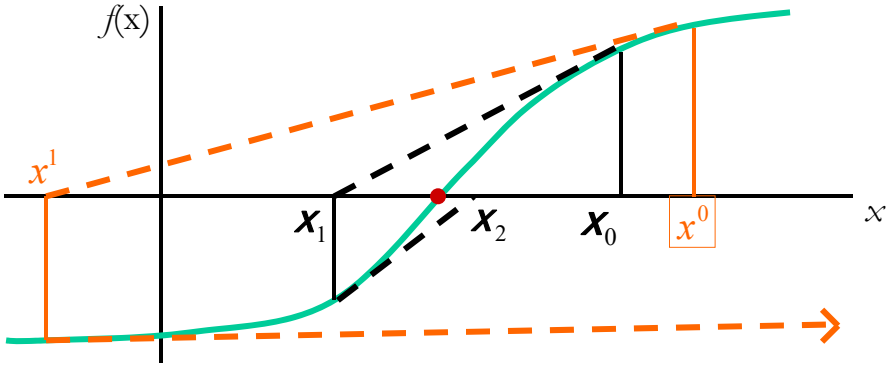
• For $x_0 =$ 0.15,
> xn
[1] 0.1500000 0.1466412 0.1463676
0.1463597 0.1463595.

• For $x_0 =$ -0.05,
> xn
[1] -0.05000000 -0.04433590 -0.04375361 -
0.04373741 -0.04373709

NR Method – Limitations $\nabla f(\boldsymbol{x}_k)=0$ or $\nabla f(\boldsymbol{x}_k)\approx 0$

- Division by zero. During the iteration we get $\boldsymbol{x}_k$ such that $\nabla f(\boldsymbol{x}_k)=0$.
- When $\nabla f(\boldsymbol{x}_k)\approx 0$ (flat region of f), the algorithm diverges.

- Both problems can be solved by using different $\boldsymbol{x}_0$ (popular solution)



courtesy Alessandra Nardi UCB

NR Method: Limitations – Division by zero

- Division by zero

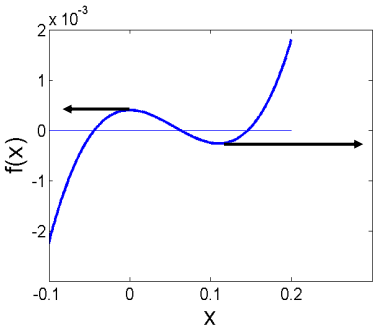
  For the equation

$$f(x)=x^3-0.03x^2+2.4\times10^{-6}=0$$

  the NR method reduces to

$$x_{i+1}=x_i-\frac{x_i^3-0.03x_i^2+2.4\times10^{-6}}{3x_i^2-0.06x_i}$$

  For $x_0=0$, or $x_0=0.2$, the denominator will equal zero.



NR Method: Limitations – Inflection Points

• Divergence at inflection points

Selection of $\boldsymbol{x}_0$ or an iteration value of the root that is close to the inflection point of the function $f(x)$ may start diverging away from the root in the Newton-Raphson method.

Example: Find the root of the equation: $f(x) = (x-1)^3 + 0.512 = 0$

The NR method reduces to $x_{i+1} = x_i - \dfrac{(x_i^3 - 1)^3 + 0.512}{3(x_i - 1)^2}$.

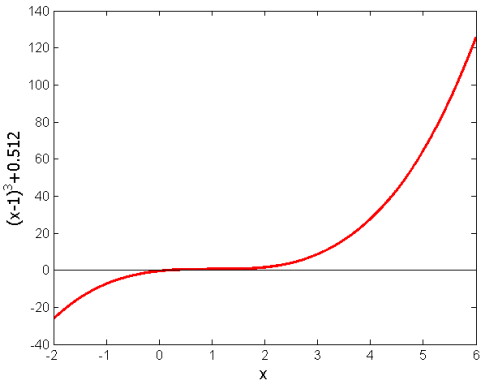
The root starts to diverge at Iteration 6 because the previous estimate of 0.92589 is close to the inflection point of $x = 1$.

Note: After $k > 12$, the root converges to the root of $x^* = 0.2$.

NR Method: Limitations – Inflection Points

• Divergence near inflection point for: $f(x) = (x-1)^3 + 0.512 = 0$

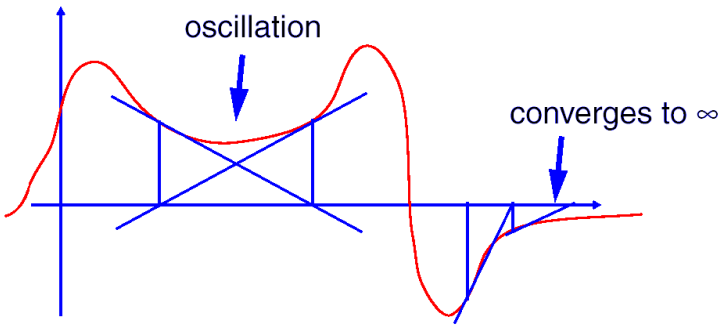
Iteration Number	x_i
0	5.0000
1	3.6560
2	2.7465
3	2.1084
4	1.6000
5	0.92589
6	−30.119
7	−19.746
18	0.2000



NR Method: Limitations – Oscillations

• It is possible that $\nabla f(\boldsymbol{x}_k)$ and $\nabla f(\boldsymbol{x}_{k+1})$ have opposite values (or close enough) and the algorithm becomes lock in an infinite loop. Again, different $\boldsymbol{x}_0$ can help to move the algorithm away from this problem.

Example:



NR Method: Limitations – Oscillations

• Oscillations near local maximum and minimum

Results obtained from the Newton-Raphson method may oscillate about the local maximum or minimum without converging on a root but converging on the local maximum or minimum.

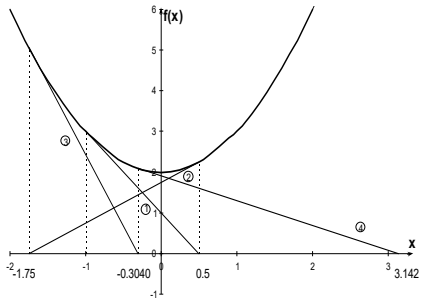
Eventually, it may lead to division by a number close to zero and may diverge.

Example: $f(x) = x^2 + 2 = 0$ $\Rightarrow$ the equation has no real roots.

NR Method: Limitations – Oscillations

• Oscillations near local maxima and minima in NR method.

Iteration Number	x_i (λ_k)	$f(x_k)$	$\lvert\varepsilon_\alpha\rvert$
0	–1.0000	3.00	-
1	0.5	2.25	300.00
2	–1.75	5.063	128.571
3	–0.30357	2.092	476.47
4	3.1423	11.874	109.66
5	1.2529	3.570	150.80
6	–0.17166	2.029	829.88
7	5.7395	34.942	102.99
8	2.6955	9.266	112.93
9	0.97678	2.954	175.96



Oscillations around local minima for $f(x)=x^2+2$

NR Method: Limitations – Oscillations

Note: Let's add a step-size, λ_k, in the updating step of x:

$$x_{k+1} = x_k - \lambda_k \frac{f\prime(x_k)}{f\prime\prime(x_k)}$$

λ_k = (.8, .9, 1, 1.1, 1.2)

Iteration	x_k	x_i (λ_k)	$f(x_k)$	$f(x_i(\lambda_k))$	$\lvert\varepsilon_\alpha\rvert$
1	0.5	0.2	2.25	2.04	300
2	-4.9	-3.88	26.01	17.0544	104.0816
3	-1.68227	-1.2427	4.8300	3.5444	130.641
4	0.183325	0.04072	2.0336	2.0016	777.8805
5	-24.5376	-19.6219	604.0944	387.0207	100.1659
6	-9.76001	-7.787	97.2578	62.6471	101.0443
7	-3.7654	-2.9610	16.17825	10.7673	106.8205
8	-1.14275	-0.7791	3.3058	2.6070	159.108
9	0.893963	0.5593	2.7992	2.3128	187.1523
10	-1.50812	-1.094	4.2744	3.1982	137.0891

$\Rightarrow$ The step-size improves the value of function!

NR Method: Limitations – Root Jumping

• Root Jumping

In some cases where the function $f(x)$ is oscillating and has a number of roots, one may choose $\boldsymbol{x}_0$ close to a root. However, the guesses may jump and converge to some other root.

Example:

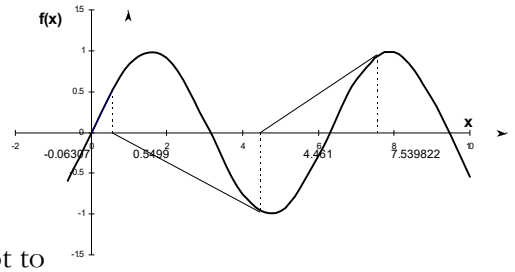
$$f(x) = \sin x = 0$$

Choose

$$x_0 = 2.4\pi = 7.539822$$

It will converge to $x = 0$, not to

$$x = 2\pi = 6.283185$$



Root jumping from intended location of root for $f(x) = \sin x = 0$

NR Method: Advantages and Drawbacks

• Advantages
- Converges fast (quadratic convergence), if it converges.
- Requires only one guess.

• Drawbacks
- Initial values are important.
- If $\nabla f(\boldsymbol{x}_k) = 0$, algorithm fails.
- Inflection points are a problem.
- If $f'(x_k)$ is not continuous the method may fail to converge.
- It requires the calculation of first and second derivatives.
- Not easy to compute $\boldsymbol{H}(\boldsymbol{x}_k)$.
- No guarantee of global minimum.

NR Method: Numerical Derivatives

- NR algorithm: $x_{k+1} = x_k - \lambda_k \frac{f(x_k)}{f\prime(x_k)}$

It requires $f'(x_k)$ and $f"(x_k)$. We can use the quotient ratio as a starting point, which delivers a 1-point approximation:

$$f'(x_k) = \frac{f(x_{k,2}) - f(x_{k,1})}{x_{k,2} - x_{k,1}} = \frac{f(x_k + h) - f(x_k)}{h}$$

- We can use a 2-point approximation of $f'(x_k)$ (errors may cancel):

$$f'(x_k) = \frac{[f(x_k + h) - f(x_k)] + [f(x_k) - f(x_k - h)]}{h} = \frac{f(x_k + h) - f(x_k - h)}{h}$$

- This approximation produces the *secant method formula* for x_{k+1}:

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

with a slower convergence than NR method (1.618 relative to 2).

NR Method: Numerical Derivatives

- We also need $f''(x_k)$. A similar approximation for the second derivative $f''(x_k)$ is also done. Let's suppose we use a one-point approximation:

$$f''(x_k) = \frac{f'(x_{k,2}) - f'(x_{k,1})}{x_{k,2} - x_{k,1}} = \frac{f'(x_k + h) - f'(x_k)}{h}$$

- The approximation problems for the 2nd derivative become more serious: The approximation of the 1st derivative is used to approximate the 2nd derivative. A typical propagation of errors situation.

Example: In the previous R code, we can use:

```
> #   num derivatives
> df1<-function(x,h=0.001){
+    return((f(x+h)-f(x))/h);
+ }
```

Multivariate Case - Example

• N-variate NR-method iteration: $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{H}^{-1}\nabla f(\boldsymbol{x}_k)$

Example:

$$\max_x \; x_1^{.2} x_2^{.3} x_3^{.4} x_4^{.1} \qquad \text{s.t.} \quad x_1+2x_2+3x_3+x_4=100$$

$$\Rightarrow \quad \max_x \quad x_2^{.3} x_3^{.4} (100-2x_2-3x_3-x_4)^{.2} x_4^{.1}$$

• Starting with $\boldsymbol{x}_0=(1,1,1)$

$$\nabla_x f(x) = \begin{pmatrix} .7337 & .9766 & .2428 \end{pmatrix}$$

$$\nabla^2_{xx} f(x) = \begin{pmatrix} -.5275 & .2885 & .0717 \\ .2885 & -.6085 & .0954 \\ .0717 & .0954 & -.2244 \end{pmatrix}$$

$$x_{t+1} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} - \nabla_x f(x)\left(\nabla^2_{xx} f(x)\right)^{-1} = \begin{pmatrix} 5.3559 & 5.3479 & 5.3226 \end{pmatrix}$$

• Solution $\boldsymbol{x}^*$ = (15.000593 13.333244 9.998452)

Multivariate Case – Example in R

```
library(numDeriv)
f <- function(z) {y <- -((100-2*z[1]-3*z[2]-z[3])^.2*z[1]^.3*z[2]^.4*z[3]^.1)
return(y)
}
x <- c(1,1,1)
# numerical gradient & hessian
#  df1 <- grad(f, x, method="Richardson")
#  d2f1 <- hessian(f, x, method="complex")
max_ite = 10; tol=.0001
#  NR
NR_num <- function(f,tol,x,N) {
i <- 1; x.new <- x
p <- matrix(1,nrow=N,ncol=4)
while(i<N) {
  df1 <- grad(f, x, method="Richardson")
  d2f1 <- hessian(f, x, method="complex")
     x.new <- x - solve(d2f1)%*%df1
     p[i,] <- rbind(x.new,f(x.new))
     i <- i + 1
if (abs(f(x.new) - f(x)) < tol) break
      x <- x.new
}
      return(p[1:(i-1),])
}
NR_num(f,tol,x,max_ite)
         [,1]         [,2]         [,3]        [,4]
[1,]  5.355917  5.347583  5.322583  -8.890549
[2,] 16.499515 16.238509 15.464327 -11.441345
[3,] 18.145527 15.038933 12.965833 -12.877825
[4,] 16.732599 15.097816 10.829738 -13.982991
[5,] 15.764754 13.970439 10.555647 -14.473775
[6,] 15.118464 13.442423 10.074552 -14.553002
[7,] 15.002920 13.335869 10.002014 -14.554887
[8,] 15.000002 13.333335 10.000001 -14.554888
```

Multivariate Case – Example in R

Note: $\boldsymbol{H}(\boldsymbol{x}_k)^{-1}$ can create problems. If we change the calculation method to the *Richardson extrapolation*, with $\boldsymbol{x}_0$=(1,1,1), we get a NaN result for x.new after 3 iterations $\Rightarrow \boldsymbol{H}(\boldsymbol{x}_k)^{-1}$ is not pd!

```
> d2f1 <- hessian(f, x, method="Richardson")
```

But, if we use the Richardson extrapolation, with $\boldsymbol{x}_0$=(2,2,2), we get

```
          [,1]     [,2]     [,3]     [,4]
[1,]  9.51400  9.480667  9.380667 -12.83697
[2,] 16.24599 15.461114 13.168466 -13.52119
[3,] 16.42885 14.464135 10.247566 -14.32000
[4,] 15.30370 13.624640 10.268205 -14.54030
[5,] 15.02318 13.353077 10.012483 -14.55482
[6,] 15.00010 13.333421 10.000072 -14.55489
```

• Lots of computational tricks are devoted to deal with these situations.

Multivariate Case – Computational Drawbacks

• Basic N-variate NR-method iteration:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{H}(\boldsymbol{x}_k)^{-1} \nabla f(\boldsymbol{x}_k)$$

As illustrated before, $\boldsymbol{H}(\boldsymbol{x}_k)^{-1}$ can be difficult to compute. In general, the inverse of $\boldsymbol{H}$ is time consuming. (In addition, in the presence of many parameters, evaluating $\boldsymbol{H}$ can be impractical or costly.)

• In the basic algorithm, it is better not to compute $\boldsymbol{H}(\boldsymbol{x}_k)^{-1}$.

Instead, solve $\boldsymbol{H}(\boldsymbol{x}_k)\ (\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = -\nabla f(\boldsymbol{x}_k)$

• Each iteration requires:

- Evaluation of $\nabla f(\boldsymbol{x}_k)$
- Computation of $\boldsymbol{H}(\boldsymbol{x}_k)^{-1}$
- Solution of a linear system of equations, with coefficient matrix $\boldsymbol{H}(\boldsymbol{x}_k)$ and RHS matrix $-\nabla f(\boldsymbol{x}_k)$.

Multivariate Case – H Matrix

• In practice, $\boldsymbol{H}(\boldsymbol{x}_k)$ can be difficult to calculate. Many times, $\boldsymbol{H}(\boldsymbol{x}_k)$is just not pd. There are many tricks to deal with this situation.

• A popular trick is to add a matrix, $\boldsymbol{E}_k$ (usually, $\delta\mathbf{I}$), where δ is a constant, that ensures $\boldsymbol{H}(\boldsymbol{x}_k)$is pd. That is,

$$\boldsymbol{H}(\boldsymbol{x}_k) \approx \nabla^2 f(\boldsymbol{x}_k) + \boldsymbol{E}_k.$$

• The algorithm can be structured to take a different step when $\boldsymbol{H}(\boldsymbol{x}_k)$ is not pd, for example, the steepest descent. That is, $\boldsymbol{H}(\boldsymbol{x}_k) \approx \mathbf{I}$.

Note: Before using the Hessian to calculate standard errors, make sure it is pd. This can be done by computing the eigenvalues and checking they are all positive.

Multivariate Case – H Matrix

• NR method is computationally expensive. The structure of the NR algorithm does not help (there is no re-use of data from one iteration to the other).

• To avoid computing the Hessian –i.e., second derivatives-, we'll approximate. Theory-based approximations:

- Gauss-Newton: $$H(x_k) = \left| E[\frac{\partial^2 L}{\partial x \partial x'}] \right|_{x_k} = \left| [\frac{\partial f(x)'}{\partial x} \frac{\partial f(x)}{\partial x}] \right|_{x_k}$$

- BHHH: $$H(x_k) = -\left| \sum_{t=1}^{T} \frac{\partial L_t}{\partial x} \frac{\partial L_t}{\partial x'}] \right|_{x_k}$$

Note: In the case we are doing MLE, for each algorithm, $-H(x_k)$ can serve as an estimator for the asymptotic covariance matrix for the maximum likelihood estimator of x_k.

Modified Newton Methods

The Modified Newton method for finding an extreme point is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\, \mathbf{S}_k\, \nabla y(\mathbf{x}_k)$$

Note that:

if $\mathbf{S}_k = \mathbf{I}$, then we have the method of steepest descent

if $\mathbf{S}_k = \mathbf{H}^{-1}(\mathbf{x}_k)$ and $\alpha = 1$, then we have the "pure" Newton method

if $y(\mathbf{x}) = 0.5\ \mathbf{x}^T\mathbf{Q}\mathbf{x} - \mathbf{b}^T\mathbf{x}$, then $\mathbf{S}_k = \mathbf{H}^{-1}(\mathbf{x}_k) = \mathbf{Q}$ (quadratic case)

Classical Modified Newton's Method:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\, \mathbf{H}^{-1}(\mathbf{x}_0)\, \nabla y(\mathbf{x}_k)$$

Note that the Hessian is only evaluated at the initial point $\mathbf{x}_0$.

Quasi-Newton Methods

• Central idea underlying *quasi-Newton methods (*a *variable metric method*) is to use an approximation of the inverse Hessian (H^{-1}).

• By using approximate partial derivatives, there is a slightly slower convergence resulting from such an approximation, but there is an improved efficiency in each iteration.

• Idea: Since $\boldsymbol{H}(\boldsymbol{x}_k)$ consists of the partial derivatives evaluated at an element of a convergent sequence, intuitively Hessian matrices from consecutive iterations are " close" to one another.

• Then, it should be possible to cheaply update an approximate $\boldsymbol{H}(\boldsymbol{x}_k)$ from one iteration to the other. With an NxN matrix $\boldsymbol{D}$:

$\boldsymbol{D} = \boldsymbol{A} + \boldsymbol{A}^u$, $\boldsymbol{A}^u$: update, usually of the form $\mathbf{u}\mathbf{v}^T$.

Quasi-Newton Methods

• Or $\boldsymbol{D}_{k+1} = \boldsymbol{A}_k + \boldsymbol{A}_k^u$, $\boldsymbol{A}_k^u$: update, usually of the form $\mathbf{uv}^T$,

where **u** and **v** are Nx1 given vectors in R^n. (This modification of **A** to obtain **D** is called a *rank-one update*, since $\mathbf{uv}^T$ has rank one.)

• In quasi-Newton methods, instead of the true Hessian, an initial matrix $\boldsymbol{H}_0$ is chosen (usually, $\boldsymbol{H}_0 = I$), which is subsequently updated by an update formula:

$$\boldsymbol{H}_{k+1} = \boldsymbol{H}_k + \boldsymbol{H}_k^u, \quad \text{where } \boldsymbol{H}_k^u \text{ is the update matrix.}$$

• Since in the NR method, we really care about $\boldsymbol{H}^{-1}$, not $\boldsymbol{H}$. The updating is done for $\boldsymbol{H}^{-1}$. Let $\mathbf{B} = \boldsymbol{H}^{-1}$; then the updating formula for $\boldsymbol{H}^{-1}$ is also of the form:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \mathbf{B}_k^u$$

Quasi-Newton Methods – Conjugate Gradient

• Conjugate Method for Solving $\mathbf{Ax} = \mathbf{b}$

- Two non-zero vectors **u** and **v** are conjugate (with respect to $\boldsymbol{A}$) if $\mathbf{u}^T\mathbf{Av} = 0$ ($\mathbf{A}=\mathbf{H}$ symmetric and pd $\Rightarrow \langle \mathbf{u},\mathbf{Av}\rangle = \mathbf{u}^T\mathbf{Av}$).

- Suppose we want to solve $\mathbf{Ax}=\mathbf{b}$. We have n mutually conjugate directions, **P** (a basis of R^n). Then, $x^* = \Sigma\, \alpha_i p_i$.

- Thus, $\mathbf{b} = \mathbf{Ax}^* = \Sigma\, \alpha_i \mathbf{A} p_i$.

- For any $\mathbf{p}_K \in \mathbf{P}$,

$$\mathbf{p}_K^T\,\mathbf{b} = \mathbf{p}_K^T\,\mathbf{Ax}^* = \Sigma\, \alpha_i\, \mathbf{p}_K^T\,\mathbf{A}p_i = \boldsymbol{\alpha}_K\, \mathbf{p}_K^T\,\mathbf{Ap}_K$$

Or

$$\boldsymbol{\alpha}_K = \mathbf{p}_K^T\,\mathbf{b} \;/\; \mathbf{p}_K^T\,\mathbf{Ap}_K$$

• Method for solving $\mathbf{Ax} = \mathbf{b}$: Find a sequence of n conjugate directions, and then compute the coefficients $\boldsymbol{\alpha}_K$.

Quasi-Newton Methods – Conjugate Gradient

• Conjugate Gradient Methods

- Conjugate gradient methods "build" up information on H.

- From our standard starting point, we take a Taylor series expansion around the point $x_k + s_k$:

$$\nabla_x f(x_k + s_k) = \nabla_x f(x_k) + \nabla^2_{xx} f(x_k) s_k$$

$$\nabla_x f(x_k + s_k) - \nabla_x f(x_k) = H(x_k) s_k$$

$$s_k'[\nabla_x f(x_k + s_k) - \nabla_x f(x_k)] = s_k' H(x_k) s_k$$

Or

$$\mathbf{s_K}^T \mathbf{Q_K} = \mathbf{s_K}^T \boldsymbol{H}(\boldsymbol{x}_k)\ \mathbf{s_K}$$

Note: $\boldsymbol{H}(\boldsymbol{x}_k)$, scaled by s_k, can be approximated by the change in the gradient:

$$q_k = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)$$

Hessian Matrix Updates

• Define $\boldsymbol{g}_k = \nabla f(\boldsymbol{x}_k)$

$\boldsymbol{p}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$ and $\boldsymbol{q}_k = \boldsymbol{g}_{k+1} - \boldsymbol{g}_k$

Then, $\boldsymbol{q}_k = \boldsymbol{g}_{k+1} - \boldsymbol{g}_k \approx \boldsymbol{H}(\boldsymbol{x}_k)\, \boldsymbol{p}_k$ *(secant condition).*

If the Hessian is constant: $\boldsymbol{H}(\boldsymbol{x}_k) = \boldsymbol{H}$ $\Rightarrow$ $\mathbf{q}_k = \boldsymbol{H}\mathbf{p}_k$

If $\boldsymbol{H}$ is constant, then the following condition would hold as well

$$\boldsymbol{H}_k^{-1} q_i = p_i \qquad 0 \le i \le k$$

This is called the *quasi-Newton condition* (also, *inverse secant condition).*

Let $\mathbf{B} = \boldsymbol{H}^1$, then the quasi-Newton condition becomes:

$$p_i = \boldsymbol{B}_k\, q_i \qquad 0 \le i \le k.$$

Rank One and Rank Two Updates

• Substitute the updating formula $\boldsymbol{B}_{k+1} = \boldsymbol{B}_k + \boldsymbol{B}_k^u$ and we get:

$$\boldsymbol{p}_i = \boldsymbol{B}_k \boldsymbol{q}_i + \boldsymbol{B}_k^u \boldsymbol{q}_i \qquad (1)$$

(remember: $\boldsymbol{p}_i = x_{i+1} - x_i$ & $\boldsymbol{q}_i = g_{i+1} - g_i$)

Note: There is no unique solution to finding the update matrix $\boldsymbol{B}_k^u$.

• A general form is $\boldsymbol{B}_k^u = a\,\mathbf{u}\mathbf{u}^T + b\,\mathbf{v}\mathbf{v}^T$

where a & b are scalars, & $\mathbf{u}$ and $\mathbf{v}$ are vectors satisfying condition (1).

Note: $a\,\mathbf{u}\mathbf{u}^T$ & $b\,\mathbf{v}\mathbf{v}^T$ are symmetric matrices of (at most) rank one.

Quasi-Newton methods that take $b = 0$ are using *rank one* updates.

Quasi-Newton methods that take $b \neq 0$ are using *rank two* updates.

Note that $b \neq 0$ provides more flexibility.

Update Formulas: Rank One

• Simple approach: Add new information to the current $\boldsymbol{B}_k$. For example, using a *rank one* update: $\boldsymbol{B}_k^u = \boldsymbol{B}_{k+1} - \boldsymbol{B}_k = \mathbf{u}\mathbf{v}^T$

$$\Rightarrow \mathbf{p}_i = (\boldsymbol{B}_k + \mathbf{u}\mathbf{v}^T)\, q_i$$

$$\Rightarrow \mathbf{p}_i - \boldsymbol{B}_k\, q_i = \mathbf{u}\mathbf{v}^T q_i$$

$$\Rightarrow \mathbf{u} = [1/(\mathbf{v}^T q_i)]\,(p_i - \boldsymbol{B}_k\, q_i)$$

$$\Rightarrow \mathbf{B}_{k+1} = \boldsymbol{B}_k + [1/(\mathbf{v}^T q_i)]\,(p_i - \boldsymbol{B}_k\, q_i)\mathbf{v}^T$$

Set $\mathbf{v}^T = (\mathbf{p}_i - \boldsymbol{B}_k q_i)$

$$\Rightarrow \boldsymbol{B}_{k+1} = \boldsymbol{B}_k + [1/((\mathbf{p}_i - \boldsymbol{B}_k q_i)^T q_i)]\,(\mathbf{p}_i - \boldsymbol{B}_k q_i)(\mathbf{p}_i - \boldsymbol{B}_k q_i)^T$$

• No systems of linear equations need to be solved during an iteration; only matrix-vector multiplications are required, which are computationally simpler.

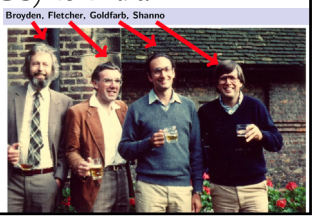
Rank One and Rank Two Updates

• Rank one updates are simple, but have limitations. It is difficult to ensure that the approximate $\boldsymbol{H}$ is pd.

• Rank two updates are the most widely used schemes, since it is easier to get the approximate $\boldsymbol{H}$ to be pd.

• Two popular rank two update formulas:

- Davidon -Fletcher-Powell (DFP) formula
- Broyden-Fletcher-Goldfarb-Shanno (BFGS) formula.



Davidon-Fletcher-Powel (DFP) Formula

• Earliest (and one of the most clever) schemes for constructing the inverse Hessian, $\boldsymbol{H}^1$, was originally proposed by Davidon (1959) and later developed by Fletcher and Powell (1963).

• It has the nice property that, for a quadratic objective, it simultaneously generates the directions of the conjugate gradient method while constructing $\boldsymbol{H}^1$ (or $\mathbf{B}$).

• Sketch of derivation:

- Rank two update for $\mathbf{B}$: $\mathbf{B}_{k+1} = \mathbf{B}_k + a\,\mathbf{u}\mathbf{u}^T + b\,\mathbf{v}\mathbf{v}^T$ (*)
- Recall $\mathbf{B}_{k+1}$ must satisfy the Inverse Secant Condition: $\mathbf{B}_{k+1}\,\mathbf{q}_k = \mathbf{p}_k$
- Post-multiply (*) by $\mathbf{q}_k$: $\mathbf{p}_k - \mathbf{B}_k\,\mathbf{q}_k = a\,\mathbf{u}\mathbf{u}^T\,\mathbf{q}_k + b\,\mathbf{v}\mathbf{v}^T\,\mathbf{q}_k$ (=0!)
- The RHS must be a linear combination of $\mathbf{p}_k$ and $\mathbf{B}_k\,\mathbf{q}_k$, and it is already a linear combination of $\mathbf{u}$ and $\mathbf{v}$. Set $\mathbf{u} = \mathbf{p}_k$ & $\mathbf{v} = \mathbf{B}_k\,\mathbf{q}_k$.
- This makes $a\,\mathbf{u}^T\mathbf{q}_k = 1$, & $b\,\mathbf{v}^T\mathbf{q}_k = -1$
- DFP update formula:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\mathbf{p}_k\mathbf{p}_k^T}{\mathbf{p}_k^T\mathbf{q}_k} - \frac{\mathbf{B}_k\mathbf{q}_k\mathbf{q}_k^T\mathbf{B}_k}{\mathbf{q}_k^T\mathbf{B}_k\mathbf{q}_k}$$

DFP Formula - Remarks

• It can be shown that if $\mathbf{p}_k$ is a descent direction, then each $\mathbf{B}_k$ is pd.

• The DFP Method benefits from picking an arbitrary pd $\mathbf{B_0}$, instead of evaluating $\boldsymbol{H}(\boldsymbol{x}_0)^{-1}$, but in this case the benefit is greater because computing an inverse matrix is very expensive.

• If you select $\mathbf{B_0} = \mathbf{I}$, we use the steepest descent direction.

• Once $\mathbf{B}_k$ is computed, the DFP Method computes

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \lambda_k \mathbf{B}_k \nabla f(\boldsymbol{x}_k)$$

where $\lambda_k > 0$ is chosen to make sure $f(\boldsymbol{x}_{k+1}) < f(\boldsymbol{x}_k)$ (use an optimal search or line search.)

Broyden–Fletcher–Goldfarb–Shanno Formula

• Remember secant condition: $\mathbf{q}_i = \mathbf{H}_{k+1}\,\mathbf{p}_i$ and $\mathbf{B}^{-1}{}_{k+1}\,\mathbf{q}_i = \mathbf{p}_i$ $0 \le i \le k$. Both equations have exactly the same form, except that $\mathbf{q}_i$ and $\mathbf{p}_i$ are interchanged and $\mathbf{H}$ is replaced by $\mathbf{B}$ ($\mathbf{B}_k = \mathbf{H}_k$) (or vice versa).

Observation: Any update formula for $\mathbf{B}$ can be transformed into a corresponding *complimentary formula* for $\mathbf{H}$ by interchanging the roles of $\mathbf{B}$ and $\mathbf{H}$ and of $\mathbf{q}$ and $\mathbf{p}$. The reverse is also true.

• BFGS formula update of $\mathbf{H}_k$: Take complimentary formula of DFP:

$$\mathbf{H}_{k+1} = \mathbf{H}_k + \frac{\mathbf{q}_k\mathbf{q}_k^T}{\mathbf{q}_k^T\mathbf{p}_k} - \frac{\mathbf{H}_k\mathbf{p}_k\mathbf{p}_k^T\mathbf{H}_k}{\mathbf{p}_k^T\mathbf{H}_k\mathbf{p}_k}$$

By taking the inverse, the BFGS update formula for $\mathbf{B}_{k+1}$ is obtained:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \left(\frac{1 + \mathbf{q}_k^T\mathbf{B}_k\mathbf{q}_k}{\mathbf{q}_k^T\mathbf{p}_k}\right)\frac{\mathbf{p}_k\mathbf{p}_k^T}{\mathbf{p}_k^T\mathbf{q}_k} - \frac{\mathbf{p}_k\mathbf{q}_k^T\mathbf{B}_k + \mathbf{B}_k\mathbf{q}_k\mathbf{p}_k^T}{\mathbf{q}_k^T\mathbf{p}_k}$$

Some Comments on Broyden Methods

- BFGS formula is more complicated than DFP, but straightforward to apply.
- Under BFGS, if $\mathbf{B}_k$ is psd, then $\mathbf{B}_{k+1}$ is also psd.
- BFGS update formula can be used exactly like DFP formula.
- Numerical experiments have shown that BFGS formula's performance is superior over DFP formula.
- Both DFP and BFGS updates have symmetric rank two corrections that are constructed from the vectors $\mathbf{p}_k$ and $\mathbf{B}_k\mathbf{q}_k$. Weighted combinations of these formulae will therefore also have the same properties.
- This observation leads to a whole collection of updates, know as the Broyden family, defined by:

$$\mathbf{B}^f = (1 - w)\,\mathbf{B}^{DFP} + w\,\mathbf{B}^{BFGS}$$

where w is a parameter that may take any real value.

Quasi-Newton Algorithm

1. Input x_0, $\mathbf{B}_0$ (say, I), termination criteria.
2. For any k, set $\mathbf{S}_k = -\mathbf{B}_k g_k$.
3. Compute a step size λ (e.g., by line search on $f(x_k + \lambda\,\mathbf{S}_k)$) and set $x_{k+1} = x_k + \lambda\,\mathbf{S}_k$.
4. Compute the update matrix $\mathbf{B}^u_k$ according to a given formula (say, DFP or BFGS) using the values $q_k = g_{k+1} - g_k$, $p_k = x_{k+1} - x_k$, and B_k.
5. Set $\mathbf{B}_{k+1} = \mathbf{B}_k + \mathbf{B}^u_k$.
6. Continue with next k until termination criteria are satisfied.

Note: You do have to calculate the vector of first order derivatives g for each iteration.

Some Closing Remarks

• Both DFP and BFGS methods have theoretical properties that guarantee superlinear (fast) convergence rate and global convergence under certain conditions.

• However, both methods could fail for general nonlinear problems. In particular:

– DFP is highly sensitive to inaccuracies in line searches.

– Both methods can get stuck on a saddle-point. In NR method, a saddle-point can be detected during modifications of the (true) Hessian. Therefore, search around the final point when using quasi-Newton methods.

– Update of Hessian becomes "corrupted" by round-off and other inaccuracies.

• All kind of "tricks" such as scaling and preconditioning exist to boost the performance of the methods.

Iterative approach

• In economics and finance, many maximization problems involve, sums of squares:

$$\arg\min_{\beta} \left\{ S(\beta) = \sum_i \left[y_i - f_i(x;\beta) \right]^2 = \sum_i \varepsilon_i^{\,2} \right\}$$

where x is a known data set and β is a set of unknown parameters.

• The above problem can be solved by many nonlinear optimization algorithms:

- Steepest descent
- Newton-Raphson
- Gauss-Newton

Gauss-Newton Method

• Gauss-Newton takes advantage of the quadratic nature of the problem.

Algorithm:

Step 1: Initialize $\beta = \beta_0$

Step 2: Update the parameter β. Determine optimal update, $\Delta\beta$.

$$\arg\min_{\Delta\beta} \sum_i [y_i - f_i(\beta_k + \Delta\beta)]^2$$

$$\approx \arg\min_{\Delta\beta} \sum_i \left[y_i - (f_i(\beta_k) + \frac{\partial f_i}{\partial \beta_k}\Delta\beta) \right]^2 \quad \text{Taylor series expansion}$$

$$= \arg\min_{\Delta\beta} \left\{ \sum_i \left[(y_i - f_i(\beta_k)) - \frac{\partial f_i}{\partial \beta_k}\Delta\beta \right]^2 = \sum_i \left[\varepsilon_i(\beta_k) - \frac{\partial f_i}{\partial \beta_k}\Delta\beta \right]^2 \right\}$$

Note: This is a quadratic function of $\Delta\beta$. Straightforward solution.

Gauss-Newton Method

$$\arg\min_{\Delta\beta} \left\{ \sum_i \left[\varepsilon_i(\beta_k) - \frac{\partial f_i}{\partial \beta_k}\Delta\beta \right]^2 = (\boldsymbol{\varepsilon} - \mathbf{J}\Delta\boldsymbol{\beta})^T (\boldsymbol{\varepsilon} - \mathbf{J}\Delta\boldsymbol{\beta}) \right\}$$

where $\mathbf{J}$ is the Jacobian of $f(\beta)$. Setting the gradient equal to zero:

$$\Delta\boldsymbol{\beta} = (\mathbf{J}^T \mathbf{J})^{-1} \mathbf{J}^T \boldsymbol{\varepsilon} \qquad \Rightarrow \text{LS solution!}$$

• Notice the setting looks like the familiar linear model:

$$\begin{vmatrix} \frac{\partial f_1}{\partial \beta_1} & \frac{\partial f_1}{\partial \beta_2} & \cdots & \frac{\partial f_1}{\partial \beta_K} \\ \frac{\partial f_2}{\partial \beta_1} & \frac{\partial f_2}{\partial \beta_2} & \cdots & \frac{\partial f_2}{\partial \beta_K} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_T}{\partial \beta_1} & \frac{\partial f_T}{\partial \beta_2} & \cdots & \frac{\partial f_T}{\partial \beta_K} \end{vmatrix} \cdot \begin{vmatrix} \Delta\beta_1 \\ \Delta\beta_2 \\ \vdots \\ \Delta\beta_K \end{vmatrix} = \begin{vmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_T \end{vmatrix}$$

Or, $\mathbf{J}\,\Delta\boldsymbol{\beta} = \boldsymbol{\varepsilon}$ (A linear system of TxN equations)

Gauss-Newton Method

• From the LS solution, the updating step involves an OLS regression:

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k + (\mathbf{J}^T \mathbf{J})^{-1} \mathbf{J}^T \boldsymbol{\varepsilon}$$

A step-size, λ_k, can be easily added:

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k + \lambda_k (\mathbf{J}^T \mathbf{J})^{-1} \mathbf{J}^T \boldsymbol{\varepsilon}$$

Note: The Guass-Newton method can be derived from the Newton-Raphson's method.

N-R's updating step: $\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \lambda_k \boldsymbol{H}^{-1} \nabla f(\boldsymbol{\beta}_k)$

where $H \approx 2(\mathbf{J}^T \mathbf{J})$ –i.e., second derivatives are ignored

$$\nabla f(\boldsymbol{\beta}_k) = 2 \mathbf{J}^T \boldsymbol{\varepsilon}$$

Gauss-Newton Method - Application

• Non-Linear Least Squares (NLLS) framework:

$$y_i = h(x_i; \boldsymbol{\beta}) + \boldsymbol{\varepsilon}_i$$

- Minimization problem: $\arg\min_{\beta} \left\{ S(\beta) = \sum_i [y_i - h(x_i;\beta)]^2 = \sum_i \varepsilon_i^2 \right\}$

- Iteration: $\mathbf{b}_{NLLS,k+1} = \mathbf{b}_{NLLS,k} + \lambda_k (\mathbf{J}^T \mathbf{J})^{-1} \mathbf{J}^T \boldsymbol{\varepsilon}$

where $\mathbf{J}^T \boldsymbol{\varepsilon} = -2 \Sigma_i \, \delta h(x_i; \boldsymbol{\beta}) / \delta \boldsymbol{\beta}_k \; \varepsilon_i$

$(\mathbf{J}^T \mathbf{J})^{-1} = -2 \Sigma_i \, \delta h(x_i; \boldsymbol{\beta}) / \delta \boldsymbol{\beta}_k \; \text{x} \; \delta h(x_i; \boldsymbol{\beta}) / \delta \boldsymbol{\beta}_k$

Note: $(\mathbf{J}^T \mathbf{J})^{-1}$ ignored the term $\{- \delta^2 h(x_i; \boldsymbol{\beta}) / \delta \boldsymbol{\beta}_k \delta \boldsymbol{\beta}_k{}^T \varepsilon_i\}$.

Or, $\mathbf{b}_{NLLS,k+1} = \mathbf{b}_{NLLS,k} + \lambda_k (\mathbf{x}^{0T} \mathbf{x}^0)^{-1} \mathbf{x}^{0T} \boldsymbol{\varepsilon}^0$ $\quad -\mathbf{x}^0 = \mathbf{J}(\mathbf{b}_{NLLS,k})$

General Purpose Optimization Routines in R

- The most popular optimizers are *optim* and *nlm*:

- *optim* gives you a choice of different algorithms including Newton, quasi-Newton, conjugate gradient, Nelder-Mead and simulated annealing. The last two do not need gradient information, but tend to be slower. (The option method="*L-BFGS-B*" allows for parameter constraints.)

- *nlm* uses a Newton algorithm. This can be fast, but if $f(\boldsymbol{x}_k)$ is far from quadratic, it can be slow or take you to a bad solution. (*nlminb* can be used in the presence of parameter constraints).

- Both *optim* and *nlm* have an option to calculate the Hessian, which is needed to calculate standard errors.