Solutions of Non-Linear Equations

Fabio Cannizzo

The Problem

- Finding the roots of an equation is one of the oldest and most common problems
- Given $y=f(x)$, find all values r such that $f(r)=0$
- An example in finance is the resolution of the *implied volatility*:
  - The premium of a Call option under certain assumptions can be computed using the Black & Sholes formula:

    $C = f(S, K, T, \sigma, r)$, where S is the spot price, K is the strike of the option, T is the time to expiry of the option, σ is the volatility of the spot price and r is the interest rate.
  - Sometime we happen the premium of the option and all other parameters, except for the volatility σ. Then we can work it out by solving the non linear equations:

$\bar{x}\sigma : g(\sigma)=0$, where $g(\sigma) = f(S, K, T, \sigma, r)-C$

Classification

- Root search methods can be classified as:
  - General root finding methods
    - Iterative (use repetitive formulas)
    - Efficiency, applicability and reliability depend on the problem
    - Sub-classification:
      - Bracketing vs open root
      - Scalar vs Multi-dimensional
  - Specialized (problem specific) root finding methods
    - Either direct or iterative
    - Typically specializations of general methods taking advantage of information specific to the problem (e.g. Heron)

Bisection

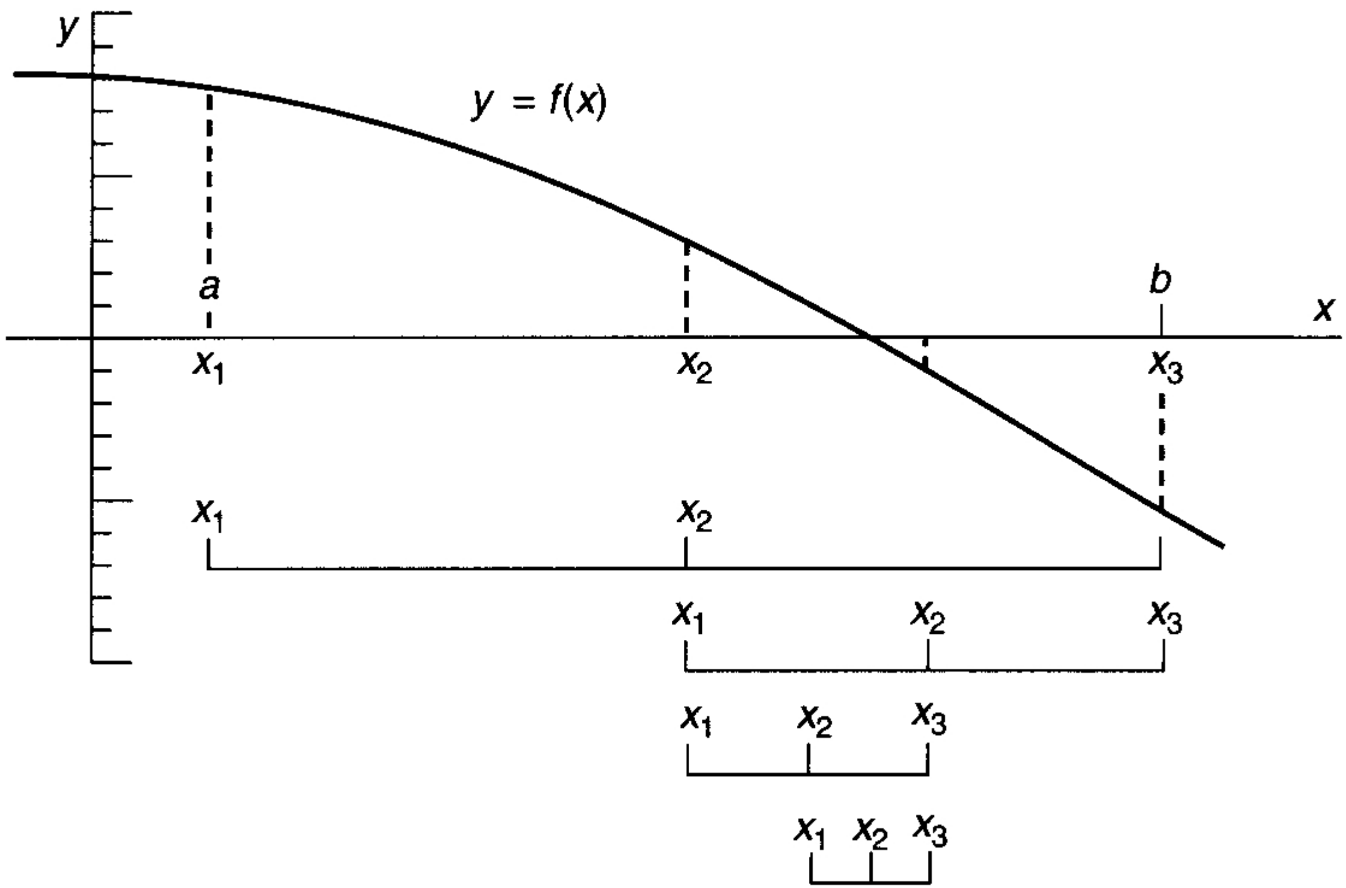
- One of the oldest and geometrically most intuitive methods
- It requires that:
  - $f(x)$ is a continuous function in $[a,b]$
  - $f(a)\,f(b) < 0$
- for the **intermediate value theorem**, there must be at least one point r in $[a,b]$ such that $f(r)=0$

Description

- The method simply values the function at the mid point $c = (a+b)/2$. There are 3 possibilities:

1. $f(c) = 0$ ➔ we have found the zero and the algorithm terminates
2. $f(c)f(b)<0$ ➔ the zero must be in $[c,b]$, we replace a with c and repeat the procedure
3. $f(c)f(a)<0$ ➔ the zero must be in $[a,c]$, we replace b with c and repeat the procedure

Geometrical Derivation



Error Bound

- The typology of this algorithm is called "*bracketing*"
- At each iteration the solution gets "bracketed" in a narrower range
- This provides bounds for the errors:

  at any iteration n the maximum error is

  $|e_n| < (b-a)/2^n$

Rate of Convergence

$$|x_n - X| < cg(n), \quad \text{for all } n > n_0$$

$$|e_n| < \frac{b-a}{2^n}$$

the error e_n decrease like $(1/2^n)$

$$\text{let} \quad E_n = \sup\{|e_n|\}$$

$$E_n = \frac{b-a}{2^n}$$

$$\log E_n = \log(b-a) - n\log 2$$

a straight line

$$\frac{E_{n+1}}{E_n} = \frac{1}{2}$$

maximum error reduces by half at each iteration

- We say convergence is **linear**, because the plot of log(E_n) vs n is a straight line with slope log2, i.e. the number of accuracy digits we gain at every iteration grows linearly

Exit Conditions

- At every step, in addition to check that $f(c)=0$, we need to impose some exit conditions.
- We can chose one or the combination of:
  1. (b-a) < epsX (most commonly used)
  2. |f(c)| < epsY
  3. Maximum number of iterations

Source Code

```
function [c,x] = bisection( f, a, b, eps )
    fa = f(a);
     fb = f(b);
    i  = 1; x=[];
    if (fa*fb > 0 ), error( 'f(a)f(b)>0' ); end
    while (b-a > eps)
        c  = (a+b)/2;
        fc = f(c);
        x(i,:)=[i,c, b-a]; i=i+1; % store iteration and error bound
        if (fc*fa<0), b = c; fb=fc;
        elseif fc == 0, return    % unlikely, so not the first branch
        else          a = c; fa=fc;
        end
    end
```

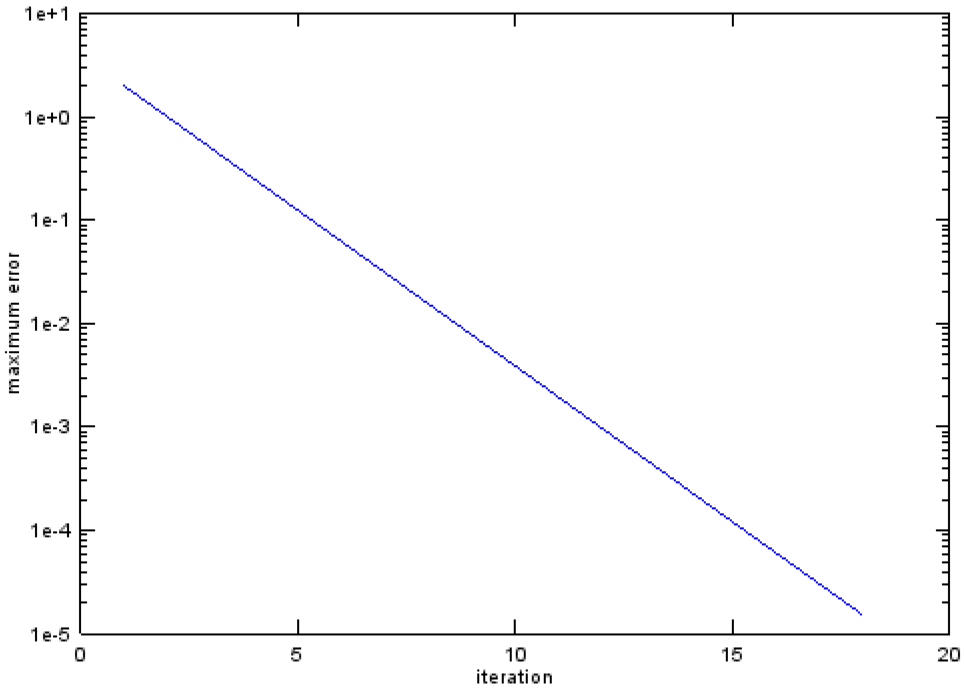
Example

```
>> myfun=@(x)exp(x)-1
>> [r,x]=bisection(@myfun,-1,2,1e-4)
r = 7.6294e-006
x =
```

it	root	a	b	bound
1	5.0000e-001	-1.0000e+000	2.0000e+000	3.0000e+000
2	-2.5000e-001	-1.0000e+000	5.0000e-001	1.5000e+000
3	1.2500e-001	-2.5000e-001	5.0000e-001	7.5000e-001
4	-6.2500e-002	-2.5000e-001	1.2500e-001	3.7500e-001
5	3.1250e-002	-6.2500e-002	1.2500e-001	1.8750e-001
6	-1.5625e-002	-6.2500e-002	3.1250e-002	9.3750e-002
7	7.8125e-003	-1.5625e-002	3.1250e-002	4.6875e-002
8	-3.9062e-003	-1.5625e-002	7.8125e-003	2.3438e-002
9	1.9531e-003	-3.9062e-003	7.8125e-003	1.1719e-002
10	-9.7656e-004	-3.9062e-003	1.9531e-003	5.8594e-003
11	4.8828e-004	-9.7656e-004	1.9531e-003	2.9297e-003
12	-2.4414e-004	-9.7656e-004	4.8828e-004	1.4648e-003
13	1.2207e-004	-2.4414e-004	4.8828e-004	7.3242e-004
14	-6.1035e-005	-2.4414e-004	1.2207e-004	3.6621e-004
15	3.0518e-005	-6.1035e-005	1.2207e-004	1.8311e-004

roughly, we gain one decimal digit every 3-4 iterations

Convergence



Error in logarithmic scale decreases linearly with the number of iterations, i.e. the number of accurate digits gained increase linearly

Summary

- As it is often true in numerical analysis, the simplest methods are the ones which converge more slowly, but also the most robust.
- The bisection method:
  - is very robust (if the hypothesis are satisfied, it is guaranteed to find the solution)
  - converges only linear
- If multiple solutions exist, it will only return one (like most root searching methods)

Regula Falsi

- It also has a nice geometric derivation
- It is also called “the method of false position”
- The assumptions are the same as in the bisection method

Description

- The method simply values the function at the intercept between the abscissa and the secant line passing for the points (*a*, f(*a*)) and (*b*, *f*(*b*)). There are 3 possibilities:

1. f(c) = 0 ➔ we have found the zero and the algorithm terminates
2. f(c)f(b)<0 ➔ the zero must be in [*c*,*b*], we replace *a* with *c* and repeat the procedure
3. f(c)f(a)<0 ➔ the zero must be in [*a*,*c*], we replace *b* with *c* and repeat the procedure

Iteration Step

- The secant line has equation

  $y - f(a) = (x-a)\, [\, f(b) - f(a)\,] \,/\, (b-a)$

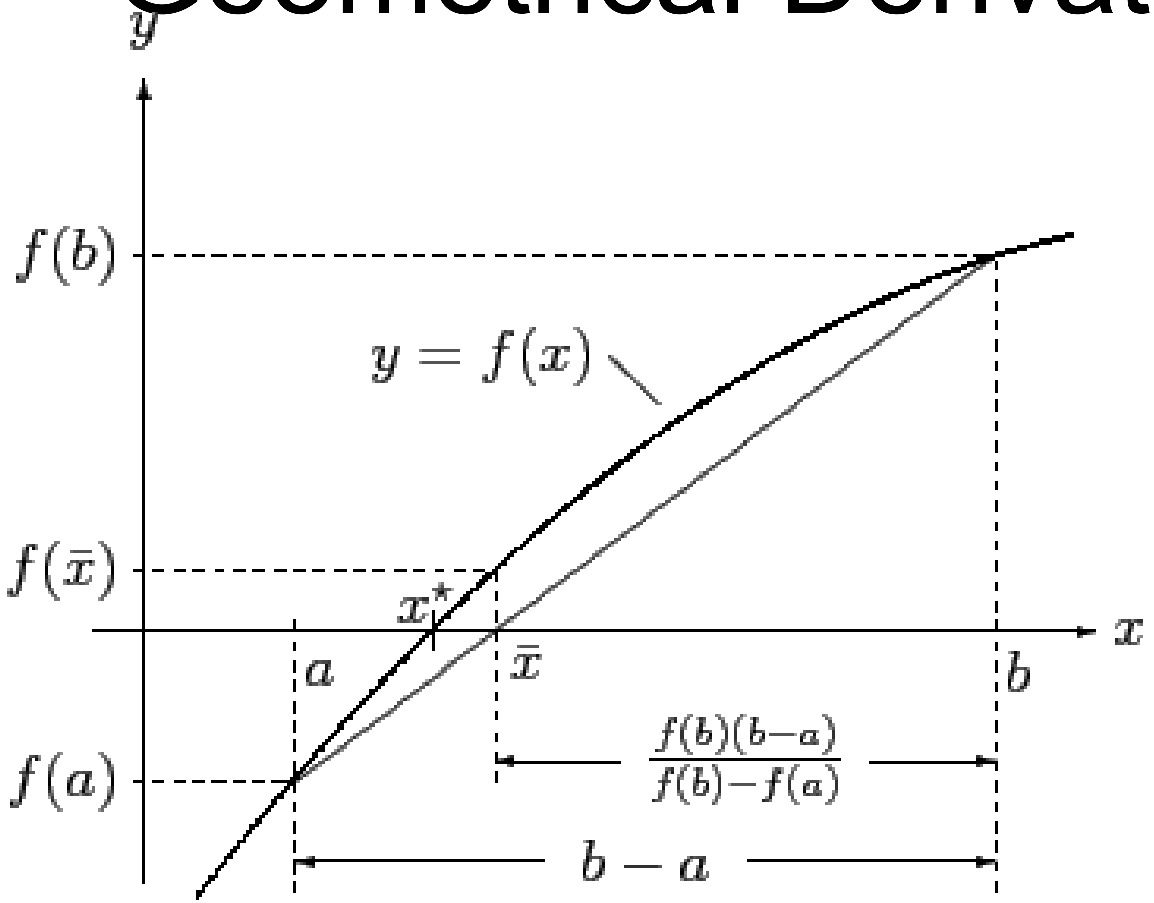
- and intercepts the x axis at:

  $c = [\, a\, f(b) - b\, f(a)\,] \,/\, [\, f(b) - f(a)\,]$

Intuition

- The idea is that, if $|f(a)| > |f(b)|$, then it can be expected that $|r-b| < |r-a|$, i.e. that the root r is closer to b than to a.
- In the bisection method instead, we take blindly the half of the interval
- Intuitively we would expect this method to converge faster

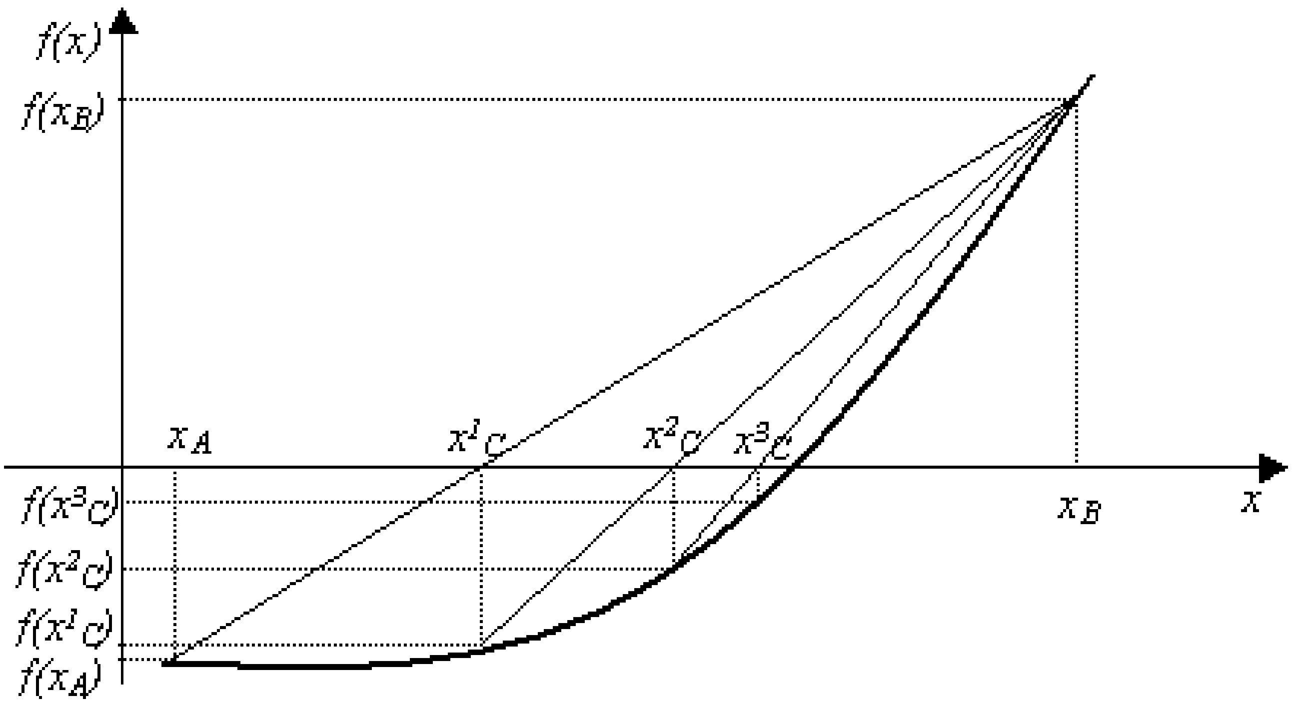
Geometrical Derivation



Bracketing

- This is also a bracketing algorithm
- At each iteration the solution gets "bracketed" in a narrower range
- The bracket size does not necessary converges to zero, as in the bisection method.

Iteration Steps



Rys. Metoda "regula falsi"

In some cases, when it keeps picking the same side of the interval, convergence can become slower close to the root

Source Code

```
function [c, x] = regulaFalsi( f, a, b, yAcc )
   fb = f( b );   fa = f( a );
  if ( fa * fb >= 0.0 ), error('f( a ) * f( b ) >= 0.0' ); end

   c = a;  iter = 1;  x = []; loop=true; % init loop variables
    while(loop)
       cOld = c;      % save previous value of c. Guard for infinite loop
       c  = a - fa * ( b  - a ) / ( fb - fa );  % new point
       fc = f( c );

       if ( fc * fb < 0.0 )  % update interval
          a  = c;
          fa = fc;
       else
          b  = c;
          fb = fc;
       end

       x(iter,:)=[iter, c]; iter=iter+1;  % unnecessary, just for display

       loop =~( abs(fc) < yAcc || cOld == c ); % exit criteria is on yAcc.
    end
```

Example

```
>> myFun =@(x)exp(x)-1
>> [y, res]= regulaFalsi(@myFun,-1,1,1.0e-5)
y = -6.01388435685245e-006
res =
iteration                 root
        1   -0.46211715726001
        2   -0.20303083197927
        3   -0.08681112900584
        4   -0.03664653812504
        5   -0.01538302292341
        6   -0.00644174012773   (roughly, 1 digit every 2-3 iterations)
        7   -0.00269477630035
        8   -0.00112682565296
        9   -0.00047109994300
       10   -0.00019694133745
       11   -0.00008232791683
       12   -0.00003441531027
       13   -0.00001438645784
       14   -0.00000601388436
```

Summary

- It is very robust. Like the bisection method, if a solution exists it is guaranteed to find it.
- Usually (but not always) it converges faster than the bisection method (super-linear)
- When it keeps picking always the same point, convergence slows down. This usually happens in proximity of the root, when $f''(x)$ has constant sign

Illinois Variation of Regula Falsi

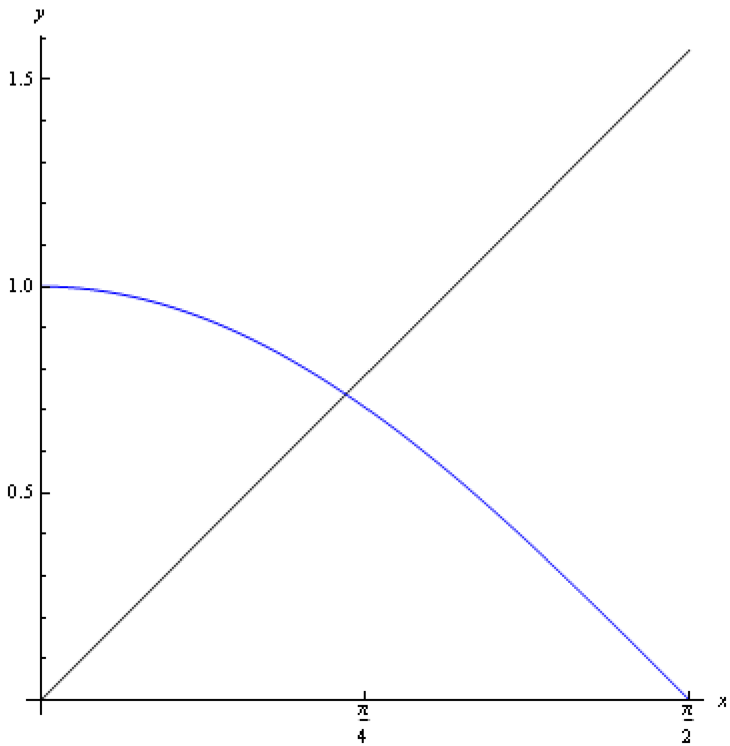
- To overcome the problem that convergence may become linear, a possible variation of the algorithm, which guarantees asymptotically super-linear convergence is:
  - when the same end-point (e.g. b) is picked twice in a row, at the next iteration we use half of $f(b)$ in the iteration step:

    $c = [\ a\ f(b)/2 - b\ f(a)\ ]\ /\ (\ f(b)/2 - f(a)\ )$
  - then at least two normal step follow

Fixed Point Algorithms

- Consider a generic function $g(x)$
- We define a fixed point of $g(x)$ as any point s such that $s = g(s)$
- The fixed point algorithm is a method to find fixed points of $g(x)$, which proceeds iteratively, simply setting $x_{i+1}=g(x_i)$

Fixed Point Graphical Solution

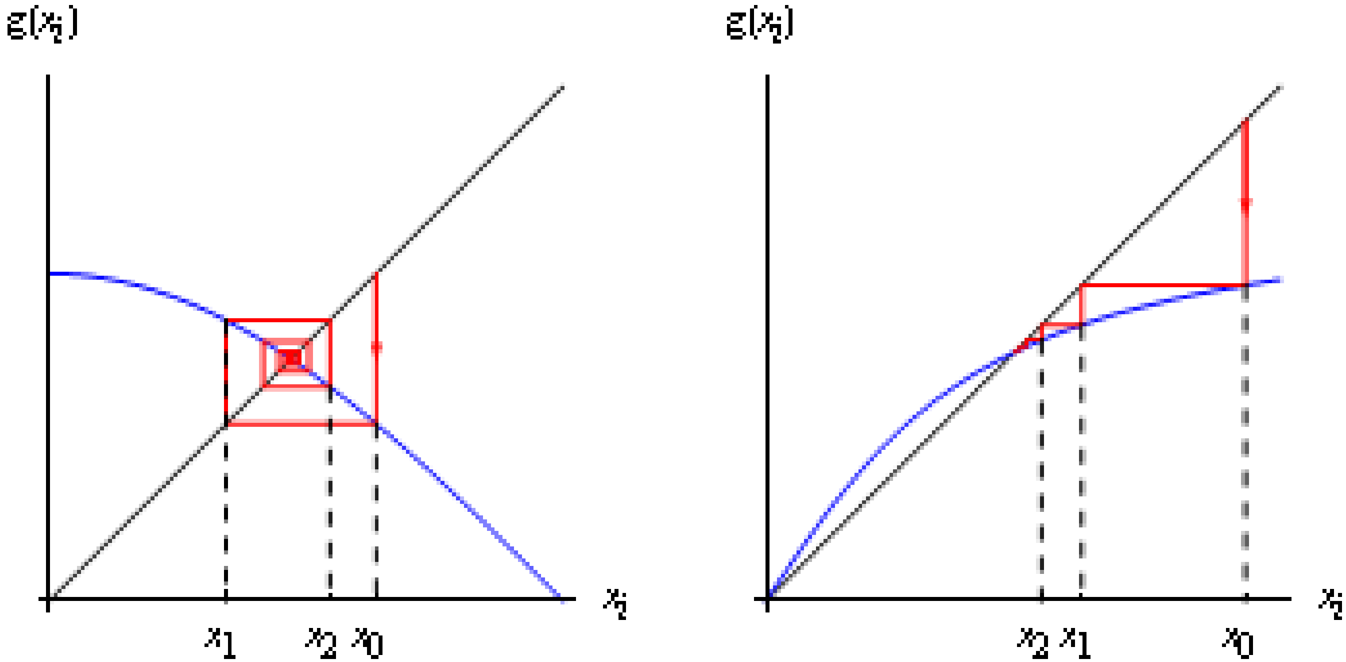


- Graphically, fixed points are just the intersections of the two lines:

  $y = g(x)$

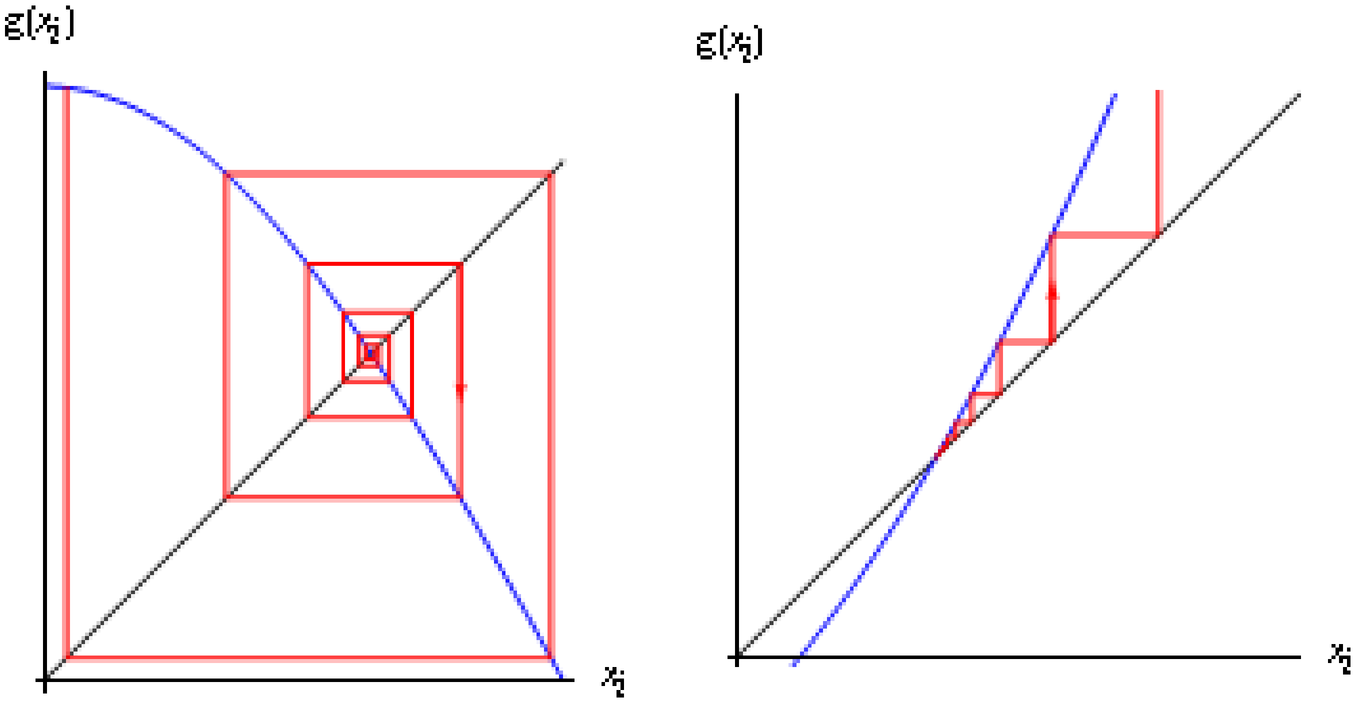
  $y = x$

Fixed Point Iteration



- We can see how the iterative process converges to the fixed point, either spiral-ing or zig-zag-ing

Fixed Point Iteration



- We can see how the iterative process does not converge to the fixed point, either spiral-ing or zig-zag-ing

Fixed Point Convergence

- Let's look at the Taylor expansion truncated at the first term (we neglect high order terms, assuming to be close to the fixed point):
  $g(x) \sim g(s) + g'(s)(x-s)$
  where s is the fixed point, i.e. $s = g(s)$
- The iteration step is $x_{i+1}=g(x_i)$, hence
  $x_{i+1} = g(x_i) \sim g(s) + g'(s)(x_i - s)$
  $x_{i+1} - s \sim g'(s)(x_i - s)$
- The distance between the estimate x_i and the solution s is multiplied by $g'(s)$ at every step
- In order for the algorithm to converge, we need the distance to reduce, i.e. $|g'(s)| < 1$

Attractive vs Repulsive Fixed Points

- If $|g'(s)|<1$, then we say the fixed point is attractive, i.e. the algorithm converges towards it
- If $|g'(s)|>1$, then we say the fixed point is repulsive, i.e. the algorithm diverge away from it

Rate of Convergence

- To analyze rate of convergence, let's use Taylor expansion of the error, truncated at the first non null term

$$e_{n+1} = x_{n+1} - s = g(x_n) - g(s) = g'(s)(x_n - s) + \frac{1}{2} g''(s)(x_n - s)^2 + \frac{1}{3!} g'''(\alpha)(x_n - s)^3$$

$$e_{n+1} = g'(s)\, e_n + \frac{1}{2} g''(s)\, e_n^{\,2} + \frac{1}{3!} g'''(\alpha_n)\, e_n^{\,3} \qquad \alpha_n \in [\min(s, x_n), \max(s, x_n)]$$

if $g'(s) \neq 0$ then the error term is :

$e_{n+1} = g'(\alpha_n)\, e_n$ linear convergence

if $g'(s) = 0$ and $g''(s) \neq 0$ then the error term is :

$e_{n+1} = \frac{1}{2} g''(\alpha_n)\, e_n^{\,2}$ quadratic convergence

and so on...

Transformation to Fixed Point

- How can the fixed point method help us in finding the roots of $f(x)$?
- Let's construct a function $g(x)$ such that $x=g(x)$ when $f(x)=0$.
- We have transformed the problem of finding the roots of $f(x)$ in the problem of finding the fixed points of $g(x)$

Transformation to Fixed Point

- Suppose: $f(x) = x^3 - 10x + 1$, which has a root for $x \sim 0.1$
- We have infinite possible choices for $g(x)$:

  $g_1(x) = [\ x^3 + 1\ ] / 10 \ \rightarrow\ g_1'(x) = 3x^2 / 10 \rightarrow\ g_1'(0.1) < 1$

  $g_2(x) = x^3 - 9x + 1 \ \rightarrow\ g_2'(x) = 3x^2 - 9 \ \rightarrow\ g_2'(0.1) > 1$

  $g_3(x) = [\ 10x - 1\ ]^{1/3} \ \rightarrow\ g_3'(x) = 10/3[\ 10x - 1\ ]^{-2/3} \ \rightarrow\ g_3'(0.1) > 1$

  to list just a few!

- g_2 and g_3 are not good choices
- Unfortunately we do not know s in advance, so how can we choose?

Newton - Raphson (1690)

- Can we chose *g*(*x*) so that we have quadratic convergence when we get close to the root?

Consider:

$g(x) = x - \alpha(x) f(x)$,

fixed points of $g(x)$ are the solutions $f(x) = 0$

$g'(x) = 1 - \alpha'(x) f(x) - \alpha(x) f'(x)$

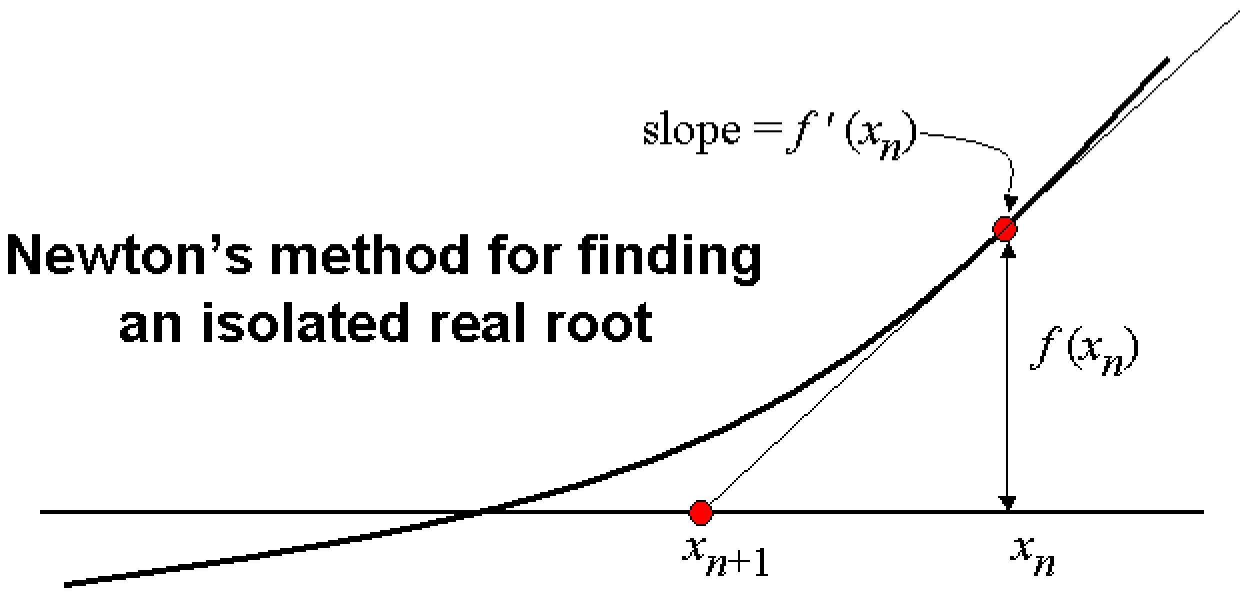
we want that $g'(s) = 0$, therefore:

$$g'(s) = 1 - \underbrace{\alpha'(s) f(s)}_{0} - \alpha(s) f'(s) \quad \Rightarrow \quad \alpha(s) = \frac{1}{f'(s)}$$

hence:

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

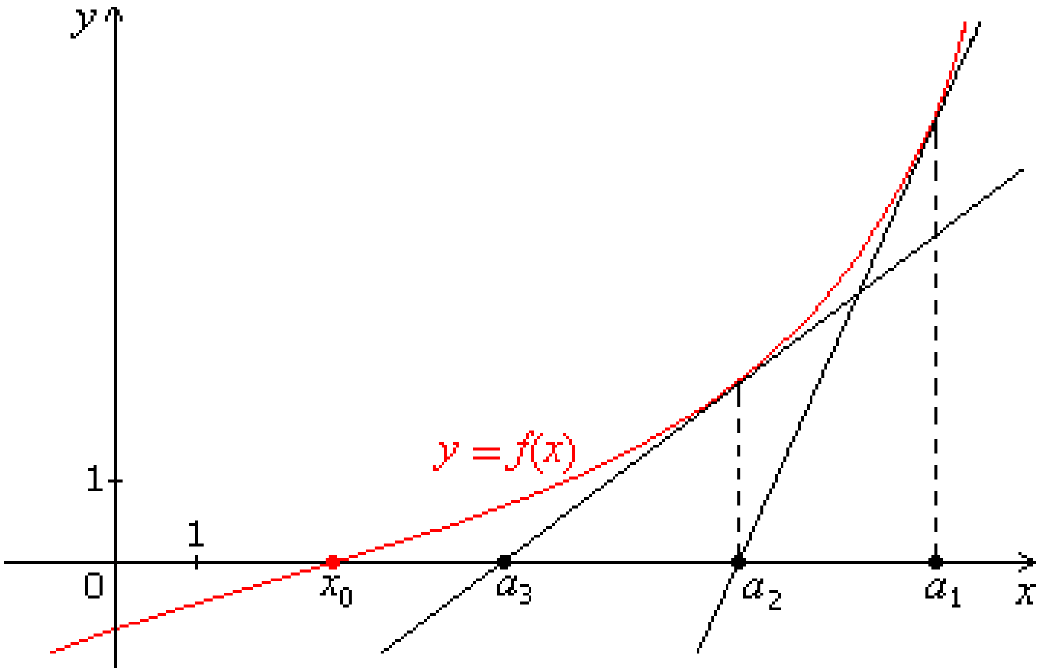
Geometrical Derivation



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

- Newton Raphson has an intuitive geometrical interpretation: at every step we take the intersection of the tangent line with the abscissa

Algorithm Steps



Derivation from Taylor

- The Newton step can also be derived from the Taylor expansion of $f(x)$

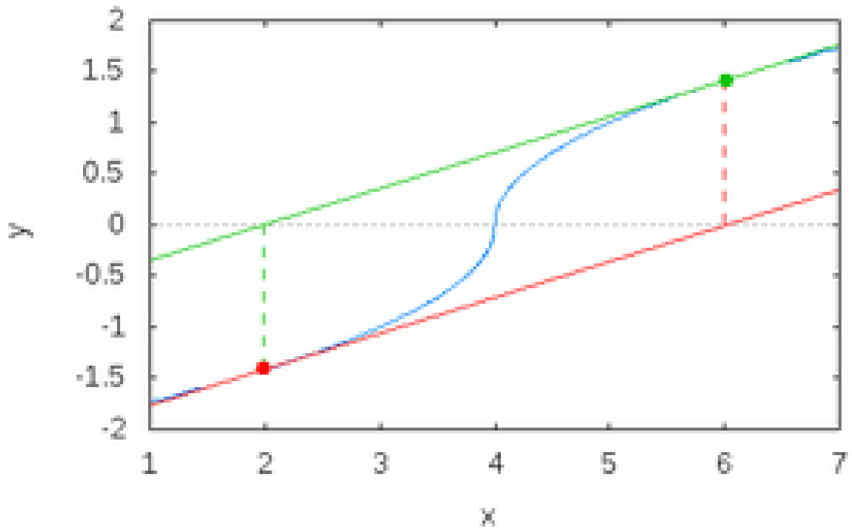
$f(x_{i+1}) \sim f(x_i) + f\,'(x_i)(x_{i+1}-x_i)$

- If the function was truly linear, i.e. if there were not higher order terms, this would be a strict equality
- We could solve the equation for x_{i+1} such that $f(x_{i+1}) = 0$, i.e.

$0 = f(x_i) + f\,'(x_i)(x_{i+1}-x_i) \;\Rightarrow\; x_{i+1} = x_i - f(x_i) \,/\, f\,'(x_i)$

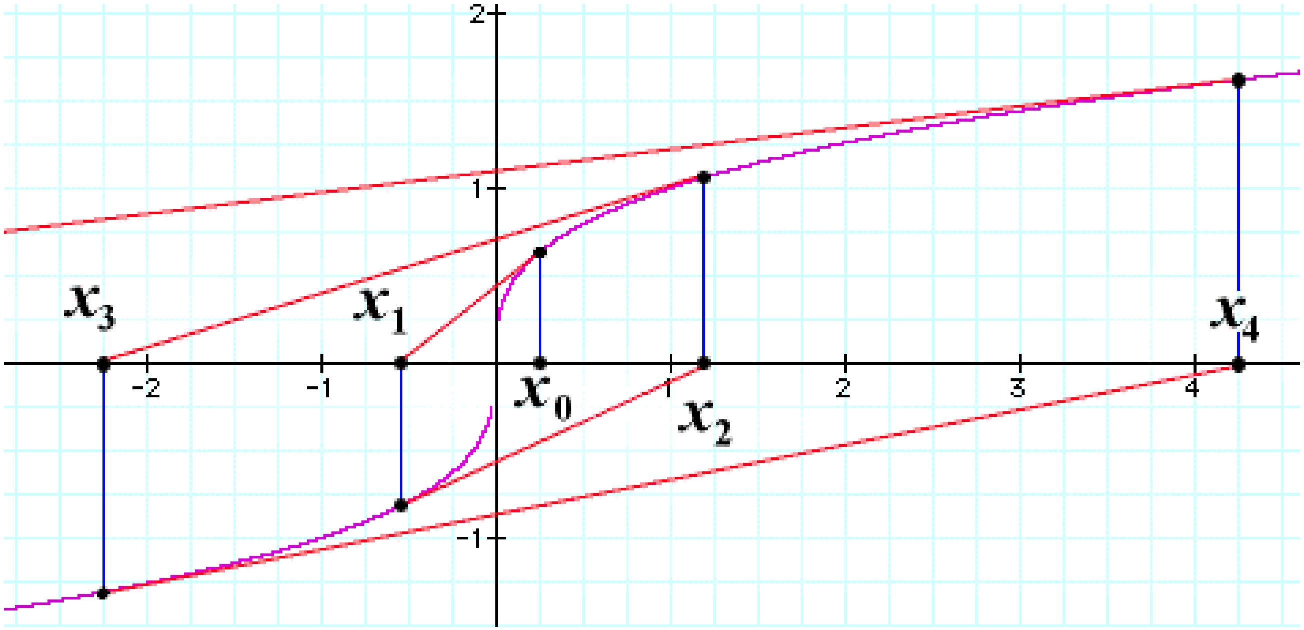
What Can Go Wrong?

- Keep visiting the same points in loop



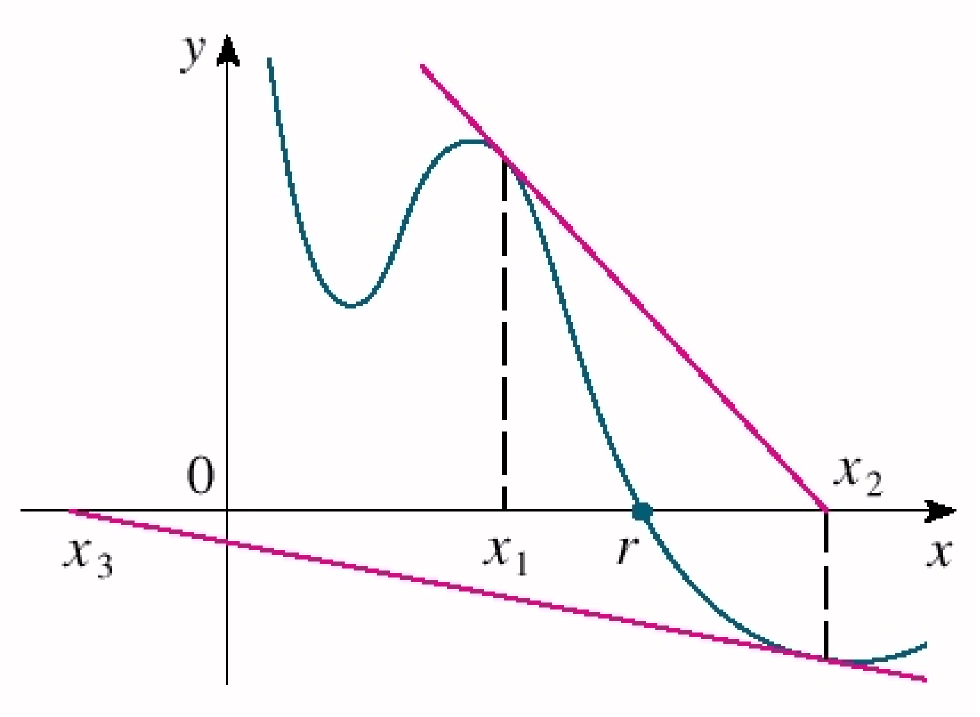
What Can Go Wrong?

- Diverge



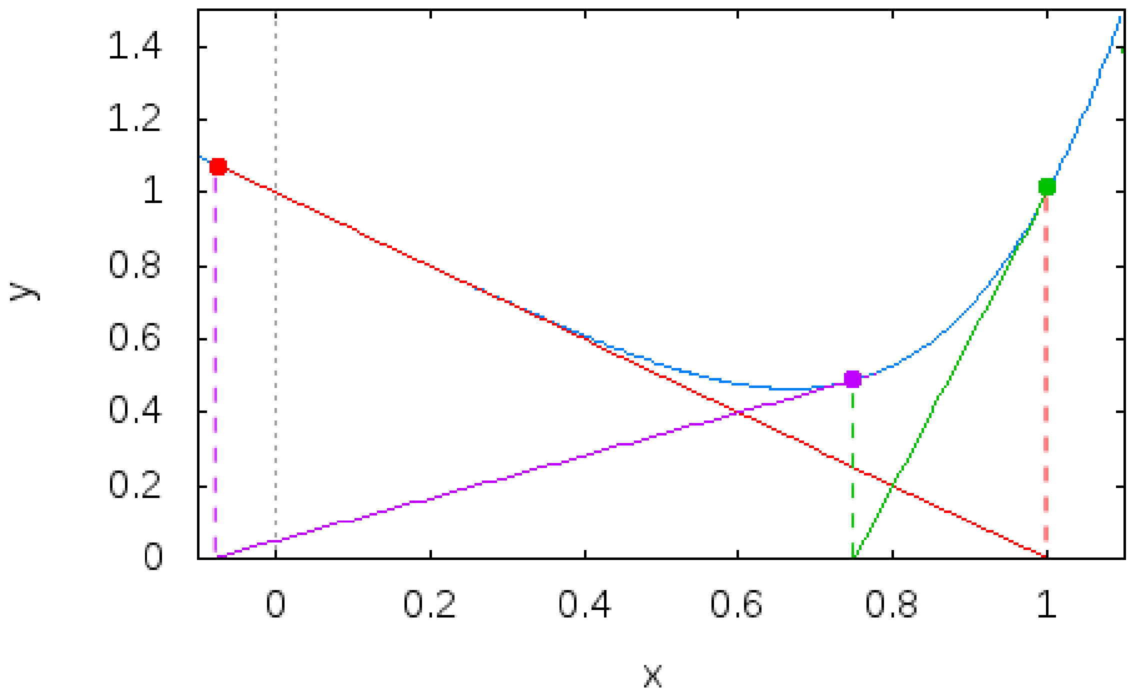
What Can Go Wrong?

- Diverge



What Can Go Wrong?

- Trapped in a local minima



Source Code

```
function [x,h] = newtonRaphson( f, fp, x, xAcc, nIter )

   found = 0; h=[];

   for i= 1:nIter  % the limit on nIter is a guard for infinite loops

         xOld = x;  % save old value of x
         x = x - f( x ) / fp( x ); % update x
        h(i,:)=[i, x]; % unnecessary, just for display

         % exit criteria on xAcc. we could also have on yAcc
        if ( abs( x - xOld ) < xAcc )
                found = 1;
                break;
        end

   end

   if( ~found ), error( 'Maximum number of iterations exceeded' ); end
```

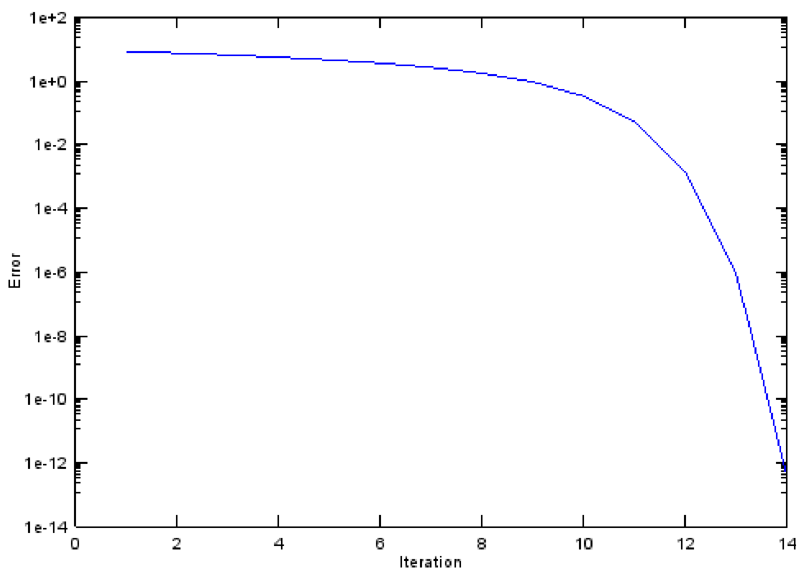
Example 1

```
>> myFun=@(x)y=exp(x)-1;
>> [y, res]= newtonRaphson(@myFun,@exp,-1,1.0e-5,200)
y = 1.0793E-016
res =
iteration root
  1        7.1828E-001
  2        2.0587E-001
  3        1.9809E-002    (the number of digits doubles
  4        1.9491E-004     at every iteration)
  5        1.8994E-008
  6        1.0793E-016
```

Example 2

```
>> myFun=@(x)exp(x)-1
>> [y, res]= newtonRaphson(@myFun,@exp,-2.5,1.0e-5,200)
>> semilogy( res(:,1), res(:,2))
y = 3.7207E-013
res =
iteration root
  1        8.6825E+000
  2        7.6827E+000
  3        6.6831E+000
  4        5.6844E+000
  5        4.6878E+000
  6        3.6970E+000
  7        2.7218E+000
  8        1.7875E+000
  9        9.5491E-001
 10        3.3976E-001
 11        5.1700E-002
 12        1.3137E-003
 13        8.6255E-007
 14        3.7207E-013
```

- The difference with Example 1 is the initial guess: -2.5 instead of -1.0
- Initially converges is very slow (sub linear)
- When it gets close convergence becomes quadratic



Rate of Convergence

- Using the convergence analysis of the fixed point algorithm, we conclude convergence is quadratic, in proximity of the root
- We could have also figured out directly from the Taylor derivation, observing that the first neglected term is of second order
- The closer to the root we start, the better it is

Condition for Convergence

$$x_{i+1} - g(s) = \underbrace{g'(s)}_{0}(x_i - s) + \frac{g''(s)}{2}(x_i - s)^2 + O\left((x_i - s)^3\right)$$

ignoring high order terms:

$$e_{i+1} \sim \underbrace{\frac{g''(s)}{2}}_{\theta} e_i^2 = \theta e_{i+1}^2 = \frac{1}{\theta}(\theta e_i)^2$$

resolving the recursion:

$$e_{i+1} \sim \frac{1}{\theta}(\theta e_o)^{2i}$$

therefore the condition for convergence is:

$$|\theta e_o| < 1$$

Sufficient Condition For Global Convergence

- We state without proving:
  - $f(a)f(b)<0$
  - $f''(x)$ has constant sign in $[a,b]$
  - we select either $x_0=a$ or $x_0=b$ so that $f(x_0)f''(x_0)>0$
- Under these condition the method will convergence even if we start far away from the root
- Note this are "sufficient" condition, but not "necessary"
- Note that second order convergence still happens only when we get *close* to the root

Newton Summary

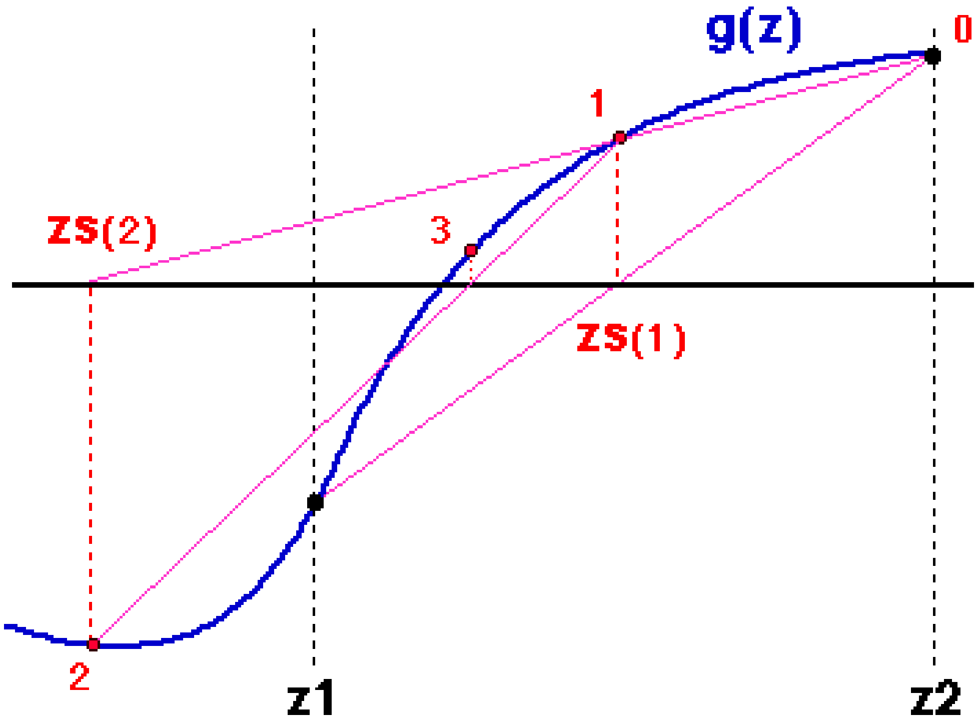
- Quadratic convergence in proximity of the solution
- It requires computation of the derivative at every point, which might be expensive
- It can only find one solution at a time
- It requires more than just continuity of the function
- It is not guaranteed to converge

Secant

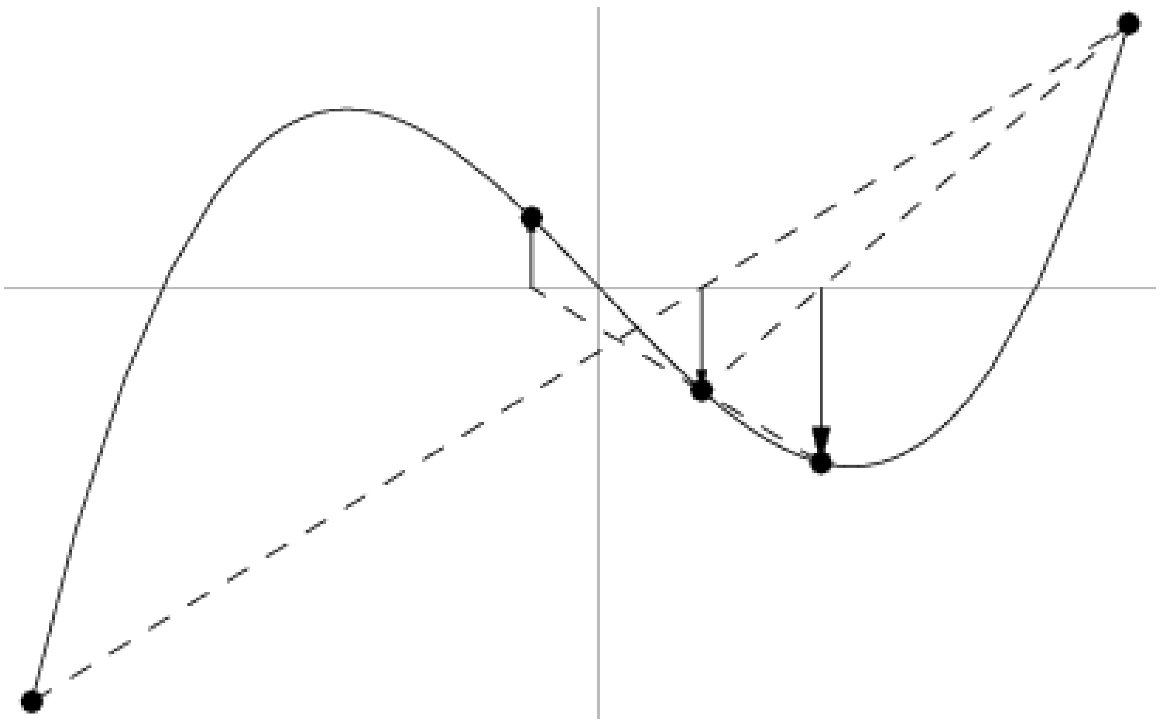
- The secant method is similar to Newton method, but approximate the derivative numerically
- It predates Newton by 3000 years
- The first step is identical to the Regula-Falsi method
- Next iterations are:

$$x_{i+1} = x_i - f(x_i)\,[x_i - x_{i-1}] \,/\, [f(x_i) - f(x_{i-1})]$$

Iteration Steps



Iteration Steps



Alternative implementation: here we do 2 initial steps of Regula Falsi

Source Code

```
function [x1,h] = secant( f, x0, x1, xAcc, nIter )

   fx0 = f( x0 );

   found = 0;

   for i = 1:nIter  % the limit on nIter is a guard for infinite loops
         x1Old = x1;
         fx1   = f( x1 );
         x1 = x1 - fx1 * ( x1  - x0 ) / ( fx1 - fx0 ); % update x1
         h(i,:)=[i, x1]; % unnecessary, just for display

         if ( abs( x1 - x1Old ) < xAcc ) % exit criteria
                 found = 1;
                 break;
         end

         x0  = x1Old;  % update x0
         fx0 = fx1;    % update fx0
   end

   if( ~found ), error( 'Maximum number of iterations exceeded' ); end
```

Example

```
>> myFun=@(x)exp(x)-1
>> [y, res]= secant(@myFun,-1,1,1.0e-5,200)
y = -2.7598E-011
res =
iteration          root
   1        -4.6212E-001
   2        -2.0303E-001
   3         5.2499E-002
   4        -5.4582E-003
   5        -1.4215E-004
   6         3.8830E-007
   7        -2.7598E-011
```

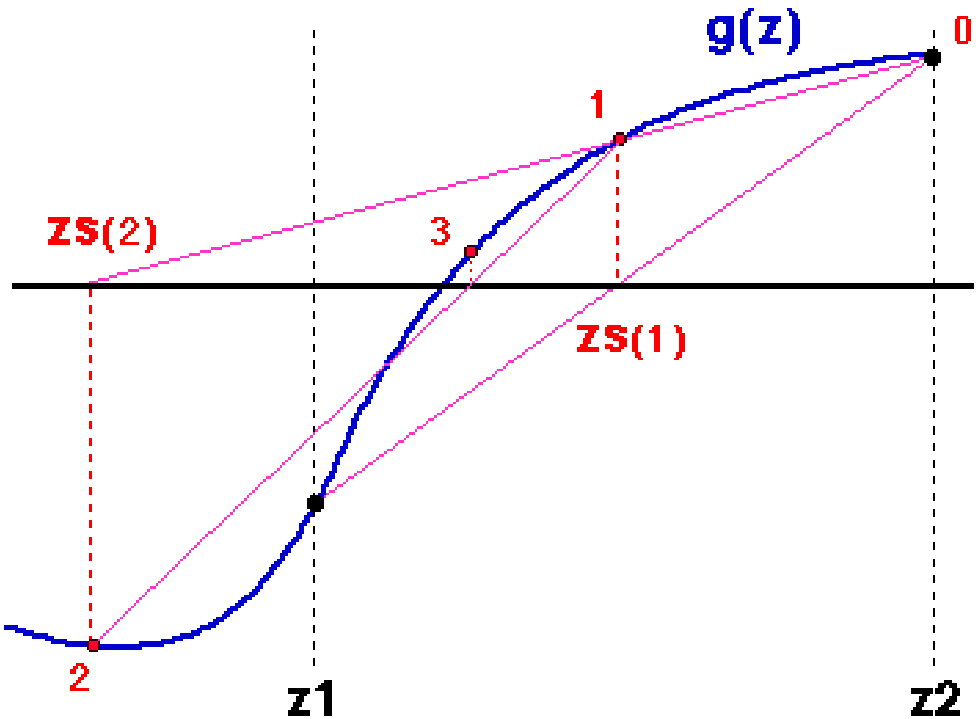
- Convergence pattern similar to Newton
- When we get close enough, convergence becomes very fast

Merging Fixed Point with Bracketing

- Bracketing methods are guaranteed to find the solution, but they are slow
- Newton-Raphson and the bisection method are fast close to the root, but convergence is not guaranteed and they can be slow far from the root.
- Idea: we use a combination of the two methods!

Mixing Methods

- Given the bracket [Z(1), Z(2)], Regula Falsi gives us ZS(1) and the new bracket [Z1,ZS(1)]
- Next, secant gives us ZS(2), which is outside of the bracket. For instance, we could ignore it and do another steps of the secant method



Brent

- Very sophisticated algorithms are based on the idea of merging more simple methods, and chose the best at each step
- Brent is a commonly used one. At every step it proceeds combining the:
  - bisection method
  - the secant method
  - inverse quadratic interpolation (intersection with y=0 of horizontal parabola passing by the last 3 points)

Implied Volatility Problem

- Problem:
  - a European call option with expiry T and settlement at *T+2 days* has price *C*
  - the forward price for time T is *F*
  - A zero coupon bond maturing at T+2 has price Z
- What is the implied volatility which gives the premium *C*?

Implied Volatility Problem (cont.)

- The Black formula can be used to price such an option:

$$Call\left(F,K,\sigma,Z,T\right)=Z\left[FN\left(d_1\right)-KN\left(d_2\right)\right]$$

$$d_1=\frac{\left(\ln F-\ln K\right)}{\sigma\sqrt{T}}+\frac{1}{2}\sigma\sqrt{T},\quad d_2=d_1-\sigma\sqrt{T}$$

- We want to solve the root search problem in the unknown σ

$$Z\left[FN\left(d_1\right)-KN\left(d_2\right)\right]-C=0$$

Implied Volatility Problem (cont.)

- Since for root searching we have to evaluate the formula a few time, let's make it as simple as possible

$$Let \quad \overline{C}=\frac{C}{ZF}, \quad \overline{K}=\frac{K}{F}, \quad L=-\ln\overline{K}, \quad s=\sigma\sqrt{T}$$

we obtain the transformed problem:

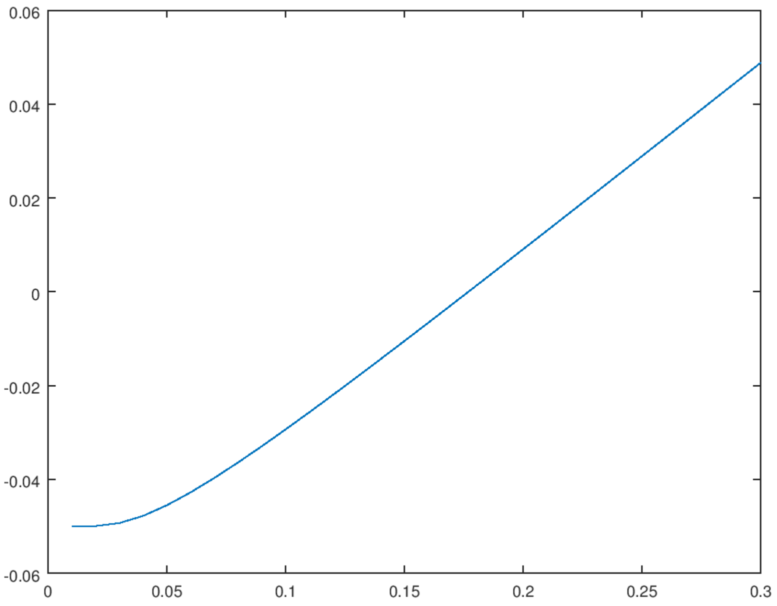
$$N\left(\overline{d}_1\right)-\overline{K}\,N\left(\overline{d}_2\right)-\overline{C}=0, \qquad \overline{d}_1=\frac{L}{s}+\frac{1}{2}s, \quad \overline{d}_2=\overline{d}_1-s$$

- We can solve the problem for *s*, then it is immediate to recover σ

Implied Volatility Problem (cont.)

- Assume F=1, T=1, K=1.05, C=0.05.
- We plot the function, to understand its behaviour: the function is smooth, convex and monotonic

```
function y=call(k,s);
    d1=-log(k)./s+0.5*s;
    d2=d1-s;
    y=normcdf(d1)-k.*normcdf(d2);

# g=@(s)call(1.05,s)-0.05;
# s=0.01:0.01:0.30; plot(s, g(s))
```



Implied Volatility Problem (cont.)

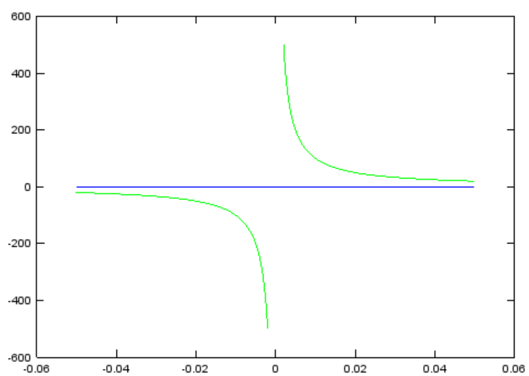
- We use the secant method

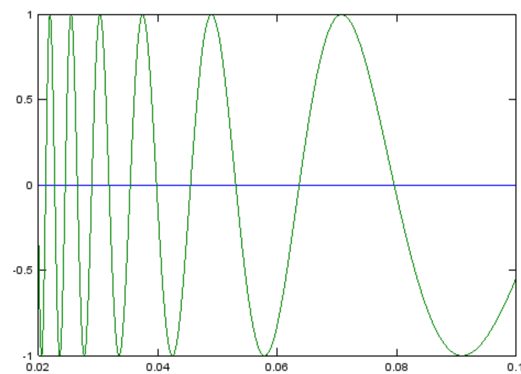
```
# [x1,h]=secant(g,0.01,0.3,0.000001, 100)
x1 =  0.17699
h =

   1.00000   0.15674
   2.00000   0.17670
   3.00000   0.17699
   4.00000   0.17699
   5.00000   0.17699
```

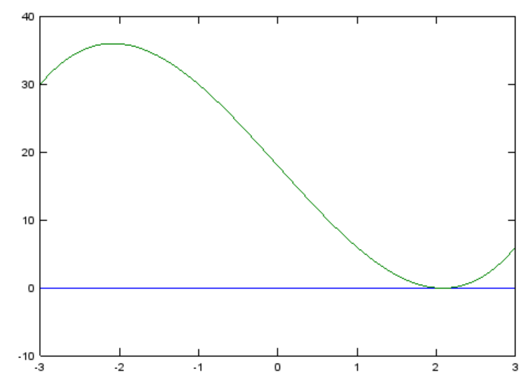
- Matlab provides the function *fzero*. Try and use that.

Difficult Root Search Situations

- We can expect problems with all algorithms presented to fail if the function $f(x)$ is not behaved:
  - Singularities (e.g. $1/x$)
  - An infinite number of roots (e.g. $\sin(1/x)$)
  - An extreme at the root ($13x^3 + 13x - 10$)





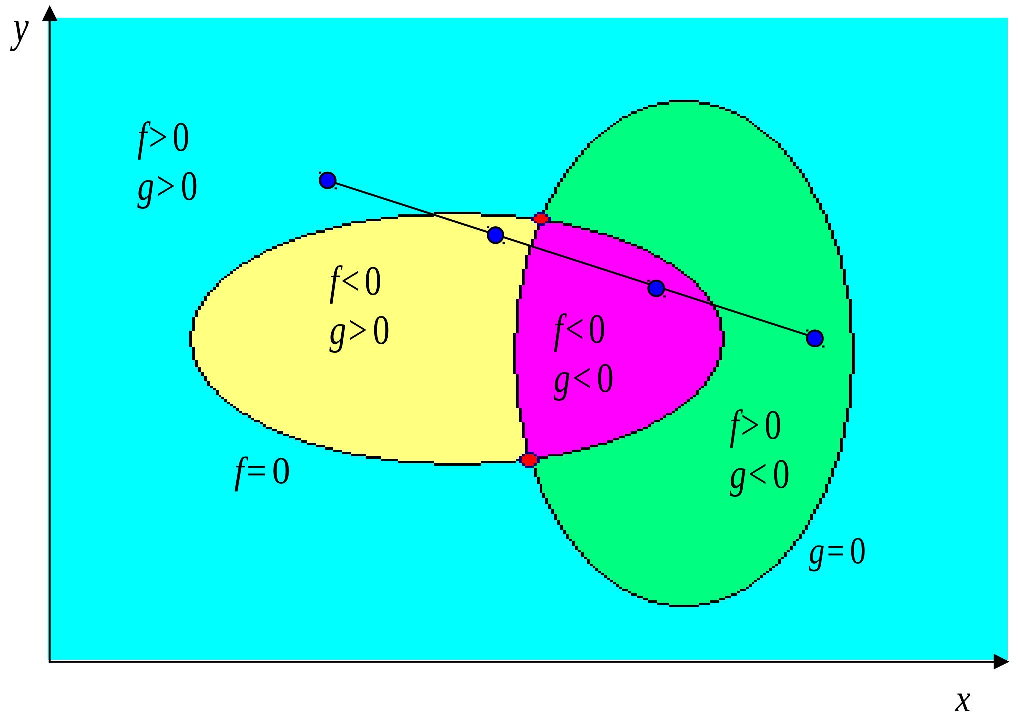


Multiple Dimensions

- In general root-search in multiple dimensions is a hard problem
- It is difficult to extend bracketing methods seen to multi-dimension

Example in Two Variables

$$\begin{cases} f(x,y) = 0 \\ g(x,y) = 0 \end{cases}$$



- How do we know if we have found all the roots?
- Even if we know one point in each region of the plane, there is no obvious way to bracket the root

Newton in Multi-Dimension

- Newton has a natural extension:

Let x be a vector of size N, and $F_i(x)$ a vector of functions ofsize N

we want to find the roots x which solve:

$F(x+h)=0$

Taylor expansion truncated to first term:

$F(x+h)=F(x)+J(x)h$

where is the $J(x)$ Jacobian matrix: $J_{i,j}(x)=\dfrac{\partial F_i(x)}{\partial x_j}$

Imposing $F(x+h)=0$

$h=-J^{-1}(x)F(x)$

Therefore we take:

$x^{(k+1)}=x^{(k)}-J^{-1}\left(x^{(k)}\right)F\left(x^{(k)}\right)$

Newton in Multi-Dimensions

- The method requires the specification of N^2 derivatives
- It is computationally very expensive. At every iteration:
  - we need to compute the N^2 derivatives (or even worse, we could use finite differences)
  - we need to solve a linear system
- It suffers of the same stability issues seen in one dimension
- There are variations of the method which achieve global convergence and reduce computational cost, merging techniques like the bisection method and techniques of multi-dimensional “minimization”
  - Good coverage in *Numerical Recipes in C++*

Polynomial

- Heavily researched area
- Closed formulae exist up to n=4
- General methods introduced can be used, but there are specializations which do better
- Good coverage in "*Numerical Recipes in C++*"

Further Readings

- Bisection Method
  - http://en.wikipedia.org/wiki/Bisection_method
- Brent Method
  - http://en.wikipedia.org/wiki/Brent's_method
- Boris Obsieger, *Numerical Methods II*
  - Good educational book on 1-d root search methods. Quite a few pages available from google books in preview.
- Johnson, Riess, *Numerical Analysis*, 1982
  - Excellent old book covering several numerical analysis topics