Nonlinear equations

• In real life problems are not always linear as in the previous chapter.

• Nonlinear equations or groups of equations occur quite frequently in computational science either as independent problems or as a part of a larger problem.

• In general a nonlinear equation can be written as

$$f(x) = 0$$

• With more than one variable we get the group

$$f_1(x_1, x_2, \ldots, x_N) = 0$$
$$f_2(x_1, x_2, \ldots, x_N) = 0$$
$$\ldots$$
$$f_N(x_1, x_2, \ldots, x_N) = 0$$

- By writing $\mathbf{x} = [x_1, x_2, \ldots, x_N]^T$ and $\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_N(\mathbf{x})]^T$ the group can be expressed as

$$\mathbf{f}(\mathbf{x}) = 0.$$

- As opposed to groups of linear equations (with nonsingular matrices) this group of nonlinear equation may not have solution at all ($\sin x + 2 = 0$) or may have infinite solutions ($\sin x = 0$).

Nonlinear equations

• Linear equations need only a finite number of arithmetic operations

- Only seldom the same applies to nonlinear equations
  - Polynomes: upto degree 4 there is an analytical solution, after that no methods with finite predetermined number of operations.

- Finite sequence of arithmetic computer operations $\Leftrightarrow$ analytical formula $\Rightarrow$ no exact solution for nonlinear equations (even with infinite precision of floating point numbers)

• Let's denote the solution of the equation as x^*: $f(x^*) = 0$

- We have found an approximation for solution: $\tilde{x}$:

$$|f(\tilde{x})| \approx 0 \text{ or}$$
$$|\tilde{x} - x^*| \approx 0$$

- We don't know x^* so the last conditions looks useless.
- If f is continuous we need not know the right solution:
  - Let a and b be such that $f(a)f(b) < 0$
  - This implies that $\exists x^*, x \in [a, b], f(x^*) = 0$ and for all points x in this interval $|x - x^*| \leq |a - b|$.
  - If a is near to b then the condition $|\tilde{x} - x^*| \approx 0$ is fulfilled.

Nonlinear equations

- In most cases conditions $|f(\tilde{x})| \approx 0$ and $|\tilde{x} - x^*| \approx 0$ are equivalent.

  - Assume the derivative is continuous and limited: $|f'(x)| \le L$
  - Express the function value in $\tilde{x}$ as a Taylor series around x^*:

$$|f(\tilde{x})| = |f(x^*) + f'(\zeta)(\tilde{x} - x^*)| = |f'(\zeta)| \cdot |\tilde{x} - x^*| \le L|\tilde{x} - x^*|$$

    where $\zeta \in [\tilde{x}, x^*]$

  - So, if $|\tilde{x} - x^*| \approx 0 \rightarrow |f(\tilde{x})| \approx 0$
  - If we further assume that $|f'(x)| \ge c > 0$ we get

$$|\tilde{x} - x^*| \le \frac{|f(\tilde{x})|}{c}.$$

  - I.e. implication to the other direction.

Nonlinear equations

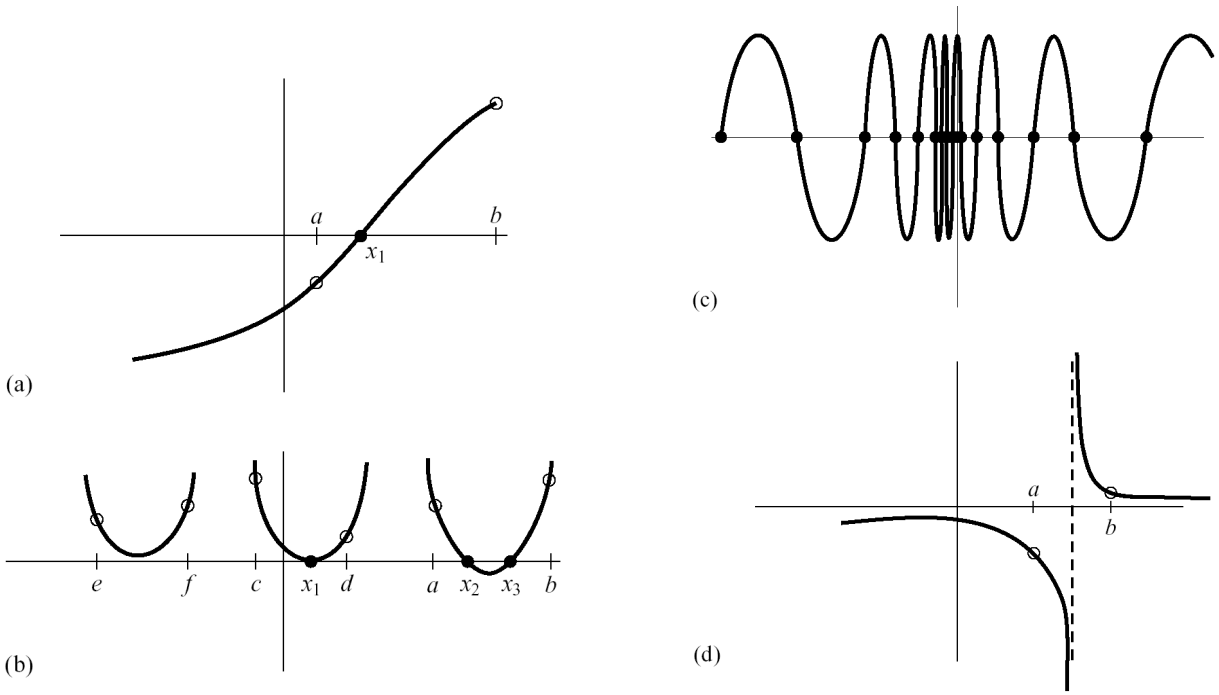
• Solution of nonlinear equations always involves iteration.

  - We get a series of numbers $x_1, x_2, \ldots, x_n, x_{n+1}, \ldots$ which (hopefully) are improving approximations to solution of the equation.
  - A good initial guess is needed.
  - In one-dimensional case an interval $[a, b]$ is given so that $f(a)f(b) < 0$: *bracketing* the root
    - If the interval does not contain the root it must be made larger:

```
#include <math.h>
#define FACTOR 1.6
#define NTRY 50
int zbrac(double (*func)(float), double *x1, double *x2)
{
     int j;
     double f1,f2;
     f1=(*func)(*x1);
     f2=(*func)(*x2);
     for (j=1;j<=NTRY;j++) {
          if (f1*f2 < 0.0) return 0;
          if (fabs(f1) < fabs(f2))
               *x1 += FACTOR*(*x1-*x2)
               f1=(*func)(*x1);
          else
               *x2 += FACTOR*(*x2-*x1)
               f2=(*func)(*x2);
     }
     return -1;
}
```

Nonlinear equations

- Root does not always imply a change in sign of $f(x)$
- A change in sign of $f(x)$ does not always imply a root.



Numerical Recipes, Fig. 9.11

- Conclusion: You should have some kind of feeling how your function behaves.
  - Plot it!

Nonlinear equations: bisection method

• The most simple and realible method to solve $f(x) = 0$

- As input an interval is given: $[a, b]$, $a < b$, $f(a)f(b) < 0$

- At every iteration step the interval is halfed and the new interval is the one that has the solution.

1. If $b - a \le \text{tol}_1$ stop.
2. Set $m = \frac{1}{2}(a + b)$. Calculate $f(m)$. If $|f(m)| \le \text{tol}_2$ stop.
3. If $f(m)f(a) < 0$, set $b \leftarrow m$ else $a \leftarrow m$. Go to 1.

- Here we have two conditions for stopping: we are near the root or that the function value is small enough.

Example: function $f(x) = x^3$ within interval $[a, b] = [-1, 2]$

Initial: $a = -1, b = 2, f(a) = -1, f(b) = 8$.

1. iteration: $m = (-1 + 2)/2 = 1/2, f(m) = 1/8$,
   $b \leftarrow 1/2, [a, b] = [-1, 1/2]$.

2. iteration: $m = (-1 + 1/2)/2 = -1/4, f(m) = -1/64$,
   $a \leftarrow -1/4, [a, b] = [-1/4, 1/2]$.

3. iteration: $m = (-1/4 + 1/2)/2 = 1/8, f(m) = 1/512$,
   $b \leftarrow 1/8, [a, b] = [-1/4, 1/8]$.

etc

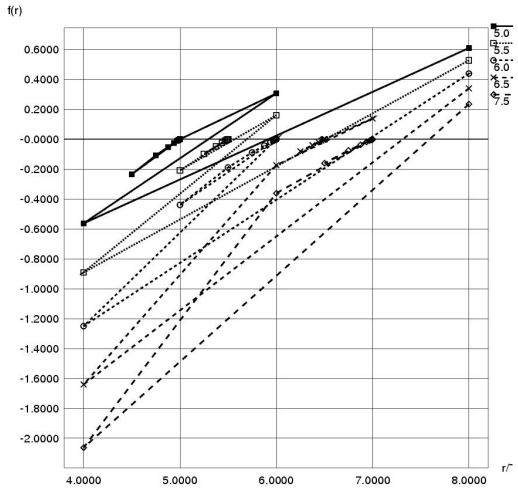
Nonlinear equations: bisection method

- If we don't know the function well it might be necessary to enlarge the interval.

- Example: $f(r) = 0$, $f(r) = 1 - \frac{b^2}{r^2} - \frac{V(r)}{E}$, $V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$

parameter values: $\varepsilon = 3.121\times10^{-3}$ eV, $\sigma = 2.74$ Å.

- Equation $f(r) = 0$ was solved with $E = 5$ eV and $b = 5.0, 5.5, 6.0, 6.5, 7.0$ Å.
- Below is shown the iteration:



Nonlinear equations: bisection method

- Estimating the error of the method is easy:
  - At every iteration step the interval is halfed
  - If we assume that x^* is in the middle of the interval m then the error $|m - x^*|$ is also halfed.
  - I.e. at every step we get one significant bit to the result.
  - If the mantissa has 24 bits the highest possible accuracy is achieved in 24 steps.

- Example: find zeroes of $f(x) = x^2 - 2$

```
program root
  implicit none
  integer, parameter :: sp=4
  real(sp) :: x1,x2,dx,f,fmid,xmid,rtbis
  integer, parameter :: MAXIT=60
  integer :: j
  x1=-1.0_sp
  x2=2.0_sp
  fmid=func(x2)
  f=func(x1)
  if (f < 0.0) then
     rtbis=x1
     dx=x2-x1
  else
   rtbis=x2
   dx=x1-x2
end if
do j=1,MAXIT
   dx=dx*0.5_sp
   xmid=rtbis+dx
   fmid=func(xmid)
 if (fmid <= 0.0) rtbis=xmid
   write(6,'(i5,2g24.14)') j,fmid,xmid
end do
```

```
contains
  function func(x)
    implicit none
    real (sp) :: func,x
    func=x**2-2.0_sp
    return
  end function func
end program root
```

Nonlinear equations: bisection method

- Output:

```
 1  -1.75000      0.50000000
 2 -0.437500      1.25000000
 3  0.640625      1.62500000
 4  0.664063E-01  1.43750000
 5 -0.194336      1.34375000
 6 -0.661621E-01  1.39062500
 7 -0.427246E-03  1.41406250
 8  0.328522E-01  1.42578125
 9  0.161781E-01  1.41992188
10  0.786686E-02  1.41699219
11  0.371766E-02  1.41552734
12  0.164461E-02  1.41479492
13  0.608683E-03  1.41442871
14  0.905991E-04  1.41424561
15 -0.168324E-03  1.41415405
16 -0.388622E-04  1.41419983
17  0.259876E-04  1.41422272
18 -0.643730E-05  1.41421127
19  0.977516E-05  1.41421700
20  0.166893E-05  1.41421413
21 -0.238419E-05  1.41421270
22 -0.357628E-06  1.41421342
23  0.715256E-06  1.41421378
24  0.238419E-06  1.41421366
25 -0.119209E-06  1.41421354
26 -0.119209E-06  1.41421354
27 -0.119209E-06  1.41421354
```

```
 1  -1.75000      0.500000000000000000
 2 -0.437500      1.250000000000000000
 3  0.640625      1.625000000000000000
 4  0.664063E-01 1.437500000000000000
 5 -0.194336      1.343750000000000000
 6 -0.661621E-01 1.390625000000000000
 7 -0.427246E-03 1.414062500000000000
 8  0.328522E-01 1.425781250000000000
 9  0.161781E-01 1.419921875000000000
10  0.786686E-02 1.416992187500000000
11  0.371766E-02 1.415527343750000000
12  0.164467E-02 1.414794921875000000
13  0.608578E-03 1.414428710937500000
14  0.906326E-04 1.414245605468750000
15 -0.168315E-03 1.414154052734375000
16 -0.388434E-04 1.414199829101562500
17  0.258941E-04 1.414222717285156250
18 -0.647477E-05 1.414211273193359375
19  0.970962E-05 1.414216995239257813
20  0.161742E-05 1.414214134216308594
21 -0.242868E-05 1.414212703704833984
22 -0.405632E-06 1.414213418960571289
23  0.605893E-06 1.414213776588439941
24  0.100130E-06 1.414213597774505615
25 -0.152751E-06 1.414213508367538452
...
43 -0.932365E-12 1.414213562372765409
44 -0.450084E-12 1.414213562372935939
45 -0.208944E-12 1.414213562373021205
46 -0.883738E-13 1.414213562373063837
47 -0.279776E-13 1.414213562373085153
48  0.222045E-14 1.414213562373095812
49 -0.128786E-13 1.414213562373090483
50 -0.532907E-14 1.414213562373093147
51 -0.155431E-14 1.414213562373094479
52  0.444089E-15 1.414213562373095145
53 -0.444089E-15 1.414213562373094923
54  0.444089E-15 1.414213562373095145
55 -0.444089E-15 1.414213562373094923
56 -0.444089E-15 1.414213562373094923
57 -0.444089E-15 1.414213562373094923
```


Nonlinear equations: bisection method

- Let $e_i = m - x^*$ be the error of the solution at step i.
- Now we have

$$\frac{|e_{i+1}|}{|e_i|} \approx \frac{1}{2}$$

- In general we say that order of convergence is r is

$$\lim_{i \to \infty} \frac{|e_{i+1}|}{|e_i|^r} = C, \text{ where } C \text{ is a nonzero constant.}$$

- Another way to write this $|e_{i+1}| = O(|e_i|^r)$

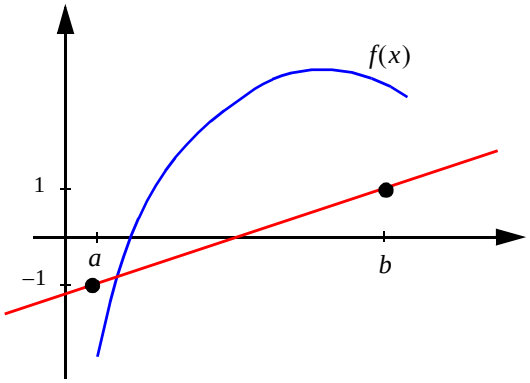
- For bisection method $r = 1$ and $C = \frac{1}{2}$ i.e. linear convergence.

Nonlinear equations: bisection method

- Bisection method can also be seen as a way to appoximate the function f.
  - At every step f is approximated by a line from $(a, \text{sign}(f(a)))$ to $(b, \text{sign}(f(b)))$
  - If we assume $f(a) < 0$ and $f(b) > 0$ the equation of the line is

$$y = -1 + 2\frac{x-a}{b-a}$$

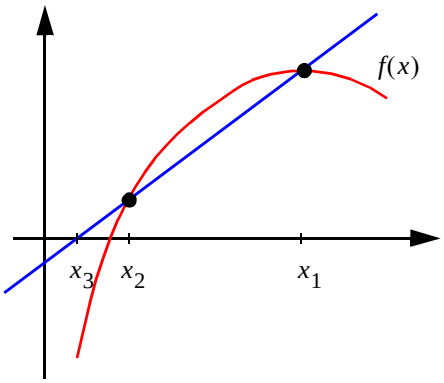
  - Now $y = 0$ if $x = m = \frac{1}{2}(a+b)$



- By using better approxmations methods with higher order convergence can be developed.

Nonlinear equations: secant method

• A line is drawn through two previous iteration points $(x_1, f(x_1))$ and $(x_2, f(x_2))$
  - The next approximation for the root is taken as the zero of this line:



  - Let the previous iteration points be $(x_i, f(x_i))$ and $(x_{i-1}, f(x_{i-1}))$.
  - The line going through them is

$$y = f(x_i) + (x - x_i)\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

  - By setting $y = 0$ we find the next point (x_{i+1}):

$$x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$

Nonlinear equations: secant method

- Example: find the root of $f(x) = x^2 - 2$

Initialization: $x_0 = 1, x_1 = 2$.

1st iteration: $x_2 = x_1 - f(x_1)((x_1 - x_0)/(f(x_1) - f(x_0)))$
$= 2 - (2)((2-1)/(2-(-1))) = 4/3$

2nd iteration: $x_3 = 4/3 - (-2/9)(4/3 - 2)/(-2/9 - 2) = 7/5$

3rd iteration: $x_4 = 7/5 - (-1/25)(7/5 - 4/3)/(-1/25 - (-2/9))$
$= 58/41 \approx 1.414634$

Bisection method: $x_4 = 1.625$

- Exact answer is $x^* = \sqrt{2} \approx 1.414214$

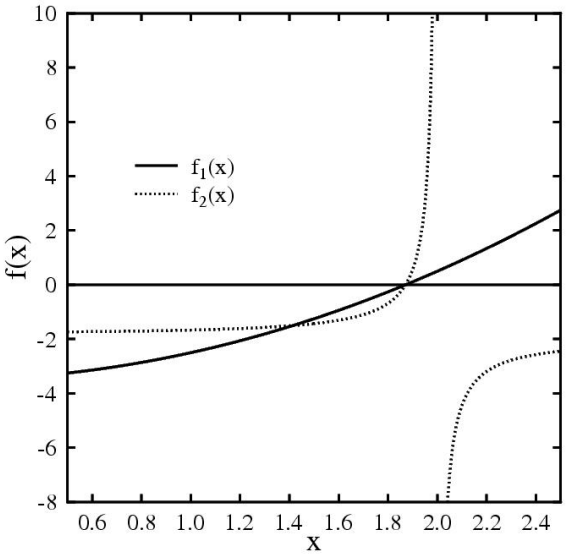
- Convergence of the secant method: $r = \frac{1}{2}(1 + \sqrt{5})$ or

$$|e_{i+1}| = O(|e_i|^{1.6180})$$

Nonlinear equations: secant method

- There are cases where the secant methods does not work well.

- Example: Compare functions $f_1(x) = x^2 - 7/2$, $f_2(x) = -\frac{1}{x^2 - 4} - 2$



- Both have the root $x^* = \sqrt{\frac{7}{2}} \approx 1.8708287$

- The essential parts of the routine (from NR[1]) used in finding roots is below:

```
fl=func(x1);
f=func(x2);
if (fabs(fl) < fabs(f)) {
     rts=x1; xl=x2; swap=fl; fl=f; f=swap;
} else {
     xl=x1; rts=x2;
}
for (j=1;j<=MAXIT;j++) {
     dx=(xl-rts)*f/(f-fl);
     xl=rts;
     fl=f;
     rts += dx;
     f=func(rts);
     if (fabs(dx) < xacc || f == 0.0) return rts;
}
```

1. Numerical Recipes

Nonlinear equations: secant method

- Reults (with double):

f_1

```
i      xi                                                  f(xi)

1   1.85910652920962      -4.372291305015263E-002
2   1.87070686809931      -4.558136460848239E-004
3   1.87082907626297       1.432590948535761E-006
4   1.87082869337450      -4.664535424581118E-011
```

f_2

```
i    xi                     f(xi)

1   1.60463917525773      -1.29831116271794
2   1.78989537239138      -0.744151759464560
3   2.03866625715734       -8.40368409286354
4   1.76572636525828      -0.866483554758109
5   1.73434894249072      -0.991969774865652
6   1.98238777075349       12.2574587791504
7   1.75291932985699      -0.921569901811837
8   1.76896536888348      -0.851579941473249
9   1.96420029842120       5.04636285662813
10   1.79715454527271      -0.701695848164392
11   1.81754670155672      -0.564299266477370
12   1.90129899503933       0.596983495110348
13   1.85824424684140      -0.171606835644274
14   1.86785728619677      -4.347078591337628E-002
15   1.87111855714021       4.348051768997330E-003
16   1.87082201762604      -9.990847146412740E-005
17   1.87082867838917      -2.244664789596840E-007
18   1.87082869338775       1.161382101599884E-011
```

- For f_2 much more iterations must be performed.
  - In this algorithm the two previous points are taken regardless of whether the root is or is not between them.
  - f_2 has a singularity at $x = 2$ and in the beginning the second branch of the function is visited.

Nonlinear equations: secant method

- Secant method can be modified so that the next iteration points always bracket the root (*false position* method)
  - For f_2 this gives:

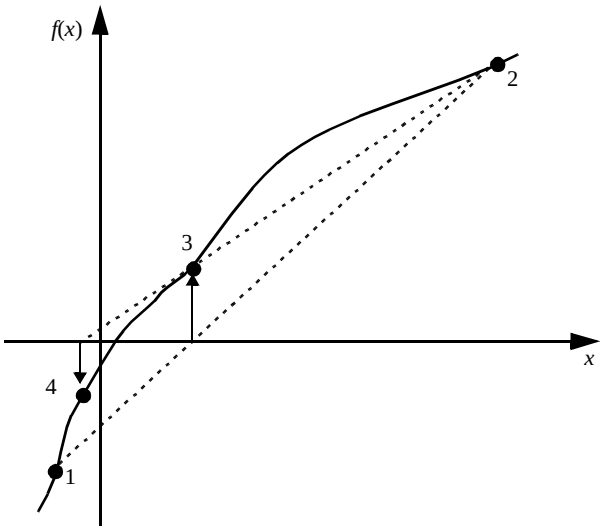
```
i    xi                     f(xi)
 1   1.60463917525773      -1.29831116271794
 2   1.78989537239138      -0.744151759464560
 3   1.84625319104558      -0.308951670432048
 4   1.86336905936041      -0.105543034024736
 5   1.86856464865188      -3.330075657966880E-002
 6   1.87014156465159      -1.022952210069694E-002
 7   1.87062015459057      -3.116086384895134E-003
 8   1.87076540350391      -9.467718107392109E-004
 9   1.87080948542342      -2.874356834887681E-004
10   1.87082286392825      -8.724340726740110E-005
11   1.87082692419442      -2.647848642811645E-005
12   1.87082815645166      -8.036078376294498E-006
13   1.87082853043155      -2.438890370992652E-006
```

- However, convergence of this method may in some cases be slower than the normal secant method.
  - It sometimes keeps the older point instead of the newer one.

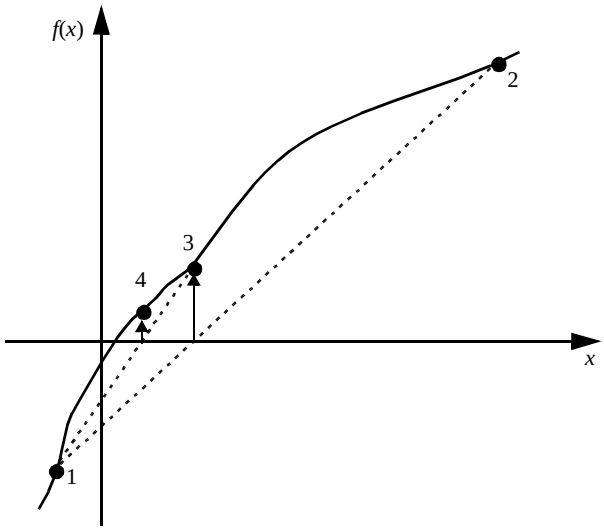
Nonlinear equations: secant method

- Figures below show the difference of these methods:

Secant



False position



Nonlinear equations: secant method

- Essential parts of routines for the secant and false position methods (from NR) for comparison:

```
fl=func(x1);
f=func(x2);
if (fabs(fl) < fabs(f)) {
  rts=x1;
  xl=x2;
  swap=fl;
  fl=f;
  f=swap;
} else {
  xl=x1;
  rts=x2;
}
for (j=1;j<=MAXIT;j++) {
  dx=(xl-rts)*f/(f-fl);
  xl=rts;
  fl=f;
  rts += dx;
  f=func(rts);
  if (fabs(dx)<xacc || f==0.0) return rts;
}

error("MAXIT exceeded.");
}
```

```
fl=func(x1);
fh=func(x2);
if (fl < 0.0) {
  xl=x1;
  xh=x2;
} else {
  xl=x2;
  xh=x1;
  swap=fl;
  fl=fh;
  fh=swap;
}
dx=xh-xl;
for (j=1;j<=MAXIT;j++) {
  rtf=xl+dx*fl/(fl-fh);
  f=func(rtf);
  if (f < 0.0) {
       del=xl-rtf;
       xl=rtf;
       fl=f;
  } else {
       del=xh-rtf;
       xh=rtf;
       fh=f;
  }
  dx=xh-xl;
  if (fabs(del)<xacc || f==0.0) return rtf;
}

error(“MAXIT exceeded.”);
}
```

Nonlinear equations: secant method

- Well, this function $f_2(x) = -1/(x^2-4)-2$ is also a difficult case for the bisection method:

```
i xi                    f(xi)
1 1.455000000000000071  -1.46893
2 1.682500000000000107  -1.14471
3 1.796250000000000124 -0.707152
4 1.853125000000000133 -0.232990
5 1.881562500000000027  0.175225
6 1.867343750000000080 -0.507861E-01
7 1.874453125000000053  0.558133E-01
8 1.870898437500000178  0.104440E-02
9 1.869121093750000018 -0.252232E-01
10 1.870009765625000098 -0.121793E-01
11 1.870454101562500027 -0.559015E-02
12 1.870676269531249991 -0.227858E-02
13 1.870787353515624973 -0.618520E-03
14 1.870842895507812464  0.212581E-03
15 1.870815124511718830 -0.203059E-03
16 1.870829010009765758  0.473879E-05
17 1.870822067260742294 -0.991657E-04
18 1.870825538635253915 -0.472148E-04
19 1.870827274322509837 -0.212384E-04
20 1.870828142166137686 -0.824988E-05
```

- In this case the point $x = 2$ does not pose any problem if it is not in the initial interval $[a, b]$.

- Secant also causes trouble if $f(x_i) \approx f(x_{i-1})$ because the denominator in the iteration formula

$x_{i+1} = x_i - f(x_i)\dfrac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$ may become huge and consequently throw the next point far away from the real root.

Nonlinear equations: secant method

- Secant method can be generalized: instead of line draw e.g. a parabola through previous points:

$$g(y) = x_{i-2}\frac{(y-f_{i-1})(y-f_i)}{(f_{i-2}-f_{i-1})(f_{i-2}-f_i)} + x_{i-1}\frac{(y-f_{i-2})(y-f_i)}{(f_{i-1}-f_{i-2})(f_{i-1}-f_i)} + x_i\frac{(y-f_{i-2})(y-f_{i-1})}{(f_i-f_{i-2})(f_i-f_{i-1})}$$

  - This a parabola $g(y)$ forced to go through points (f_{i-2}, x_{i-2}), (f_{i-1}, x_{i-1}), (f_i, x_i).
  - Next iterate os obtained as $g(y=0)$.
  - Convergence is 1.839

- We have seen that all methods have their good and bad points:

Bisection:	Sure convergence, if the initial interval $[a, b]$ is reasonable. Slow convergenve.
Secant: (and its parabola version)	Problems if the function is not smooth. Fast convergence.

Nonlinear equations: secant method

- Strategy: use both methods

  - Use the faster method but if it seems to fail do a couple of iteration using the slower but more robust method.

- NR routine that combines the parabola and bisection methods gives for function f_2 (with $[a, b] = [1.0, 1.91]$)

```
i    xi                      f(xi)
1   1.91000000000000       0.841716396703608
2   1.91000000000000       0.841716396703608
3   1.78989537239138      -0.744151759464560
4   1.84625319104558      -0.308951670432048
5   1.87551973726703       7.285799378975089E-002
6   1.86993501261844      -1.328338036293353E-002
7   1.87079620182582      -4.861667285616100E-004
8   1.87082870157870       1.226026520306789E-007
9   1.87082870157870       1.226026520306789E-007
```


Nonlinear equations: Newton's method

• If we can calculate the function derivative $f'(x)$ it can be used for locating the roots:

- f is approximated by its tangent at x_i.
- New estimate for the root from the zero of the tangent.
  Expand $f(x)$ into Taylor's series near x_i

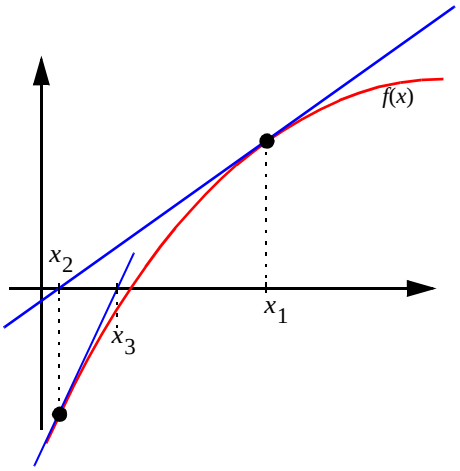
$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \ldots$$

- We only take the two first terms:

$$y = f(x_i) + f'(x_i)(x - x_i)$$

- And by setting $y = 0$ we get the next iterate:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$



- Example: $f(x) = x^2 - 2 \rightarrow f'(x) = 2x$

Initial point: $x_0 = 1$

1. iteration: $x_1 = x_0 - (x_0^2 - 2)/(2x_0) = 1 - (1 - 2)/2 = 3/2$

2. iteration: $x_2 = 3/2 - (9/4 - 2)/3 = 17/12$

3. iteration: $x_3 = 17/12 - (289/144 - 2)/(17/6) = 577/408 \approx 1.414216$

- Exact value $x^* \approx 1.414214$; i.e. after 3 iteration 6 figures right.

Nonlinear equations: Newton's method

- Before electronic calculators the following iterative method to calculate square roots was taught in schools:
  - To calculate square root of a we need to find roots of function $f(x) = x^2 - a$.
  - $f'(x) = 2x$
  - For this we get the iteration formula

$$x_{i+1} = x_i - \frac{x_i^2 - a}{2x_i} = \frac{1}{2}\left(\frac{a}{x_i} + x_i\right)$$

  - So, if you accidentally delete **`libm.so`** you still can calculate square roots with the following program ;-)

```
#include <stdio.h>
#include <stdlib.h>
main (int argc, char **argv)
{
  double a,x0,x,y;
  int n,i;
  if (argc!=4) {
    fprintf(stderr,"Usage: %s x0 a n\n",argv[0]); return (1);
  }
  x0=atof(*++argv); a=atof(*++argv); n=atoi(*++argv);
  x=x0;
  printf(" %5d %25.18g\n",0,x);
  for (i=1;i<=n;i++) {
    x=0.5*(x+a/x);
    printf(" %5d %25.18g\n",i,x);
  }
  return(0);
}
```


Nonlinear equations: Newton's method

- Compilation and run:

```
progs> gcc sqr.c
       progs> a.out 1.0 2.0 10
            0                         1
            1                       1.5
            2       1.41666666666666674
            3       1.41421568627450989
            4       1.41421356237468987
            5       1.41421356237309515
            6       1.41421356237309515
            7       1.41421356237309515
            8       1.41421356237309515
            9       1.41421356237309515
           10       1.41421356237309515
'Exact' value =     1.4142135623730950488
```

```
progs> a.out 1.0 9.0 10
     0                         1
     1                         5
     2       3.39999999999999991
     3        3.0235294117647058
     4       3.00009155413138018
     5       3.00000000139698386
     6                         3
     7                         3
     8                         3
     9                         3
    10                         3
progs> a.out -1.0 9.0 10
     0                        -1
     1                        -5
     2      -3.39999999999999991
     3       -3.0235294117647058
     4      -3.00009155413138018
     5      -3.00000000139698386
     6                        -3
     7                        -3
     8                        -3
     9                        -3
    10                        -3
```

Nonlinear equations: Newton's method

- The order of convergence of Newton's method is 2; it can be shown as below.

  - Taylor's series of $f(x)$ around x_i:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2}f''(\zeta)(x - x_i)^2$$

    where $\zeta \in [x, x_i]$

  - Set $x = x^*$ (i.e. the exact solution of $f(x) = 0$):

$$0 = f(x^*) = f(x_i) + f'(x_i)(x^* - x_i) + \frac{1}{2}f''(\zeta)(x^* - x_i)^2$$

  - Divide by $f'(x_i)$ and we get

$$x^* - \left(x_i - \frac{f(x_i)}{f'(x_i)} \right) = \frac{f''(\zeta)}{2f'(x_i)}(x^* - x_i)^2$$

  - Because $x_{i+1} = x_i - f(x_i)/f'(x_i)$ the left hand size is equal to $x^* - x_{i+1}$
  - By defining the errors in the iterates as

$$e_i = x^* - x_i, \quad e_{i+1} = x^* - x_{i+1}$$

    we get

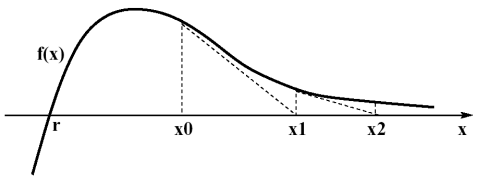
$$e_{i+1} = C_i e_i^2, \qquad \text{where } C_i = \frac{f''(\zeta)}{2f'(x_i)}$$

- If the iteration converges $\lim_{i \to \infty} \frac{|e_{i+1}|}{|e_i^2|} = C$, where $C = \left| \frac{f''(x^*)}{2f'(x^*)} \right|$.

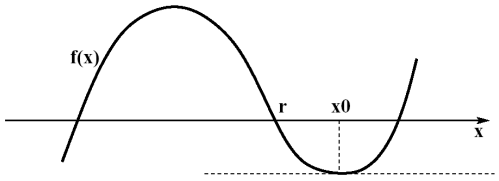
Nonlinear equations: Newton's method

- Problems in the Newton's method (starting point and during the iteration):

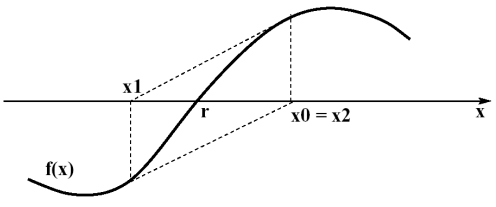
- **Runaway:** If x_0 is not sufficiently close to x^*, the tangent line at x_0 can intersect the x axis at a point remote from the root x^*, and successive points in the Newton's iteration recede from x^* instead of converging to it.



- **Flat spot:** If the tangent line at x_0 is parallel to the x axis (in a local extremum point), the next approximation is $x_1 = \pm\infty$ and Newton's method fails. Here a combination of the bisection method and Newton's method would produce a successful outcome.



- **Cycle:** It can happen that $x_2 = x_0$, which means that the iteration values cycle. This situation is often encountered if the function f is obtained by table interpolation. In practice, roundoff errors or limited precision of the computer may eventually cause this situation to become unbalanced so that the iteration continues either inward and converge or outward and diverge. With a better initial guess, the method would have succeeded.



Nonlinear equations: Newton's method

- As mentioned before the best strategy is to combine the methods.
  - Newton's methods is used but if the iteration takes the point outside predetermined limits or does not converge fast enough a couple of bisection steps are performed.

- Example: finding zero of the familiar function $f_2(x) = -1/(x^2-4)-2$, $[a,b] = [1.0, 1.91]$[1]:

```
i    xi                      f(xi)
1   1.45500000000000       -1.46892550352501
2   1.68250000000000       -1.14470976260350
3   1.79625000000000      -0.707151673329548
4   1.85312500000000      -0.232989939776709
5   1.87325873806264       3.706770509442148E-002
6   1.87087445941745       6.852062755804411E-004
7   1.87082870962077       2.429652523616710E-007
```

1. NR routine **rtsafe**

Nonlinear equations: Newton's method

- Essential parts of a routine combining bisection and Newton's methods (from NR):

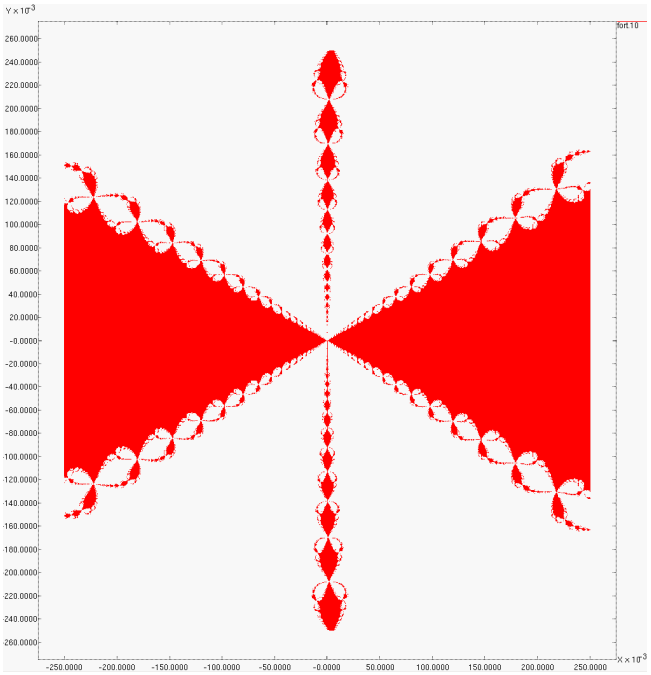
```
#include <math.h>
#define MAXIT 100

double rtsafe(void (*funcd)(double, double *,
       double *), double x1, double x2,
       double xacc)
{
  int j;
  double df,dx,dxold,f,fh,fl;
  double temp,xh,xl,rts;

  (*funcd)(x1,&fl,&df);
  (*funcd)(x2,&fh,&df);
  if ((fl>0.0 && fh>0.0) || (fl<0.0 && fh<0.0))
    error("Root must be bracketed.");
  if (fl==0.0) return x1;
  if (fh==0.0) return x2;
  if (fl<0.0) {
    xl=x1;
    xh=x2;
  } else {
    xh=x1;
    xl=x2;
  }
  rts=0.5*(x1+x2);
  dxold=fabs(x2-x1);
  dx=dxold;
  (*funcd)(rts,&f,&df);
  for (j=1;j<=MAXIT;j++) {
    if ((((rts-xh)*df-f)*((rts-xl)*df-f)>=0.0)
       || (fabs(2.0*f)>fabs(dxold*df))) {
      dxold=dx;
      dx=0.5*(xh-xl);
      rts=xl+dx;
      if (xl==rts) return rts;
    } else {
      dxold=dx;
      dx=f/df;
      temp=rts;
      rts -= dx;
      if (temp==rts) return rts;
    }
    if (fabs(dx)<xacc) return rts;
    (*funcd)(rts,&f,&df);
    if (f<0.0)
      xl=rts;
    else
      xh=rts;
  }
  error("MAXIT exceeded.");
}
```

- Function `(*funcd)(x,&f,&df)` returns $f(x)$ and $f'(x)$; $f(\mathtt{xl}) < 0$, $f(\mathtt{xh}) > 0$

Nonlinear equations: Newton's method

- Newton's method and fractals.
  - Complex equation $z^3 - 1 = 0$ has roots $z = 1$ and $z = \exp(\pm 2\pi i/3)$.
  - Iteration formula for Newton's method $z_{j+1} = z_j - (z_j^3 - 1)/(3z_j^2)$
  - Those initial points for the iteration that end up to root $z = 1$ form a fractal:

```
program iter
  implicit none
  integer, parameter :: rr=4,IMAX=100,&
                & IRMAX=500,IIMAX=500
  real (rr), parameter :: DR=0.001,&
                & DI=0.001,EPSI=1.0E-6
  complex (rr), parameter :: zroot=(1.0,0.0)
  real (rr) :: d,zr,zi
  complex (rr) :: z0,z1
  character (len=80) :: line
  integer :: ir,ii,i
  do ir=-IRMAX,IRMAX
     zr=DR*ir
     do ii=-IIMAX,IIMAX
        if (ir==0 .and. ii==0) exit
        zi=DI*ii
        z0=cmplx(zr,zi)
        do i=1,IMAX
           z1=z0-(z0**3-1.0)/(3.0*z0**2)
           d=abs(z1-zroot)
           if (d<EPSI) exit
           z0=z1
        enddo
        if (i<IMAX) write(10,*) zr,zi
     enddo
  enddo
end program iter
```



Nonlinear equations: multiple roots

• Multiple roots are more difficult to determine than simple roots.

- If the function can be written near the root x^* in the form

$$f(x) = (x - x^*)^m g(x) \qquad \textbf{(1)}$$

where $g(x)$ is continuous near x^* and $g(x^*) \neq 0$.,

then the multiplicity of the root is m.

- m need not be integer: $f(x) = x\sqrt{x-1}$, $x^* = 1$, $m = 1/2$
- If the m first derivatives of f are continuous in an interval containing x^*, and the following applies

$$\begin{cases} f(x^*) = 0 \\ f'(x^*) = f''(x^*) = \ldots = f^{(m-1)}(x^*) = 0 \\ f^{(m)}(x^*) \neq 0 \end{cases}$$

then the multiplicity of x^* is m.

- This is easy to see when we expand f as a Taylor series

$$f(x) = f(x^*) + (x - x^*)f'(x^*) + \frac{(x-x^*)^2}{2}f''(x^*) \quad \text{, where } \zeta_x \in [x, x^*]$$
$$+ \ldots + \frac{(x-x^*)^{m-1}}{(m-1)!}f^{(m-1)}(x^*) + \frac{(x-x^*)^m}{m!}f^{(m)}(\zeta_x)$$

- This can be simplified to $f(x) = \dfrac{(x-x^*)^m}{m!}f^{(m)}(\zeta_x)$ **(2)**

and writing $g(x) = f^{(m)}(\zeta_x)/m!$ we get Eq. **(1)** because $g(x^*) = f^{(m)}(x^*)/m! \neq 0$

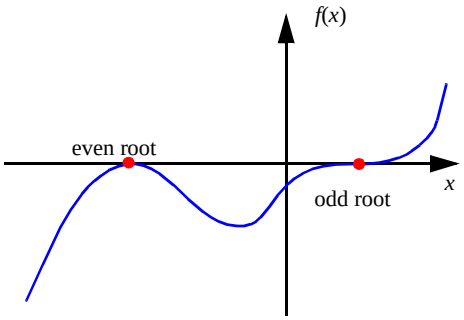
Nonlinear equations: multiple roots

- One can easily see that if m is odd $f(x)$ changes sign at the root and if m is even it does not.

- Two roots near to each other; finite precision $\rightarrow$ roots intepreted as one multiple root:

$$
\begin{aligned}
f(x) &= (x - x_1^*)(x - x_2^*)g(x) \\
&= (x - x_1^*)[x - x_1^* - (x_2^* - x_1^*)]g(x) \\
&\approx (x - x_1^*)^2 g(x)
\end{aligned}
$$

if $|x - x_1^*| \gg |x_2^* - x_1^*|$



- How does root multiplicity affect the numerical accuracy?

  - Let $\tilde{f}$ be the approximation to the function value returned by computer near x^*:

$$|f(x) - \tilde{f}(x)| \leq \varepsilon,$$

where ε is a constant.

  - Assume that computer return zero with argument z:

$$\tilde{f}(z) = 0$$

Nonlinear equations: multiple roots

  - How large is the error in z?

  - If the multiplicity of x^* is m we can use Eq. **(2)**:

$$f(z) \approx (z - x^*)^m \frac{f^{(m)}(x^*)}{m!}$$

  - Because we assume that $|f(x) - \tilde{f}(x)| \leq \varepsilon \rightarrow |f(z) - \tilde{f}(z)| \leq \varepsilon \rightarrow |f(z)| \leq \varepsilon$:

$$\mp\varepsilon \approx (z - x^*)^m \frac{f^{(m)}(x^*)}{m!} \quad \Rightarrow \quad |z - x^*| \approx \varepsilon^{1/m} \left| \frac{m!}{f^{(m)}(x^*)} \right|^{1/m}.$$

  - ε is usually small but if m is large can error increase to large values.
    - Near a multiple root a small error in the function value may cause a large error in root position.
    - Put in a another way: Near a multiple root the function is almost horizontal. Because of this a small shift in the function value causes large variation in point where the function crosses the x axis.

- Simple roots ($m = 1$) can sometimes pose problems if the 1st derivative is small:

$$|z - x^*| \approx \left| \frac{\varepsilon}{f'(x^*)} \right|$$

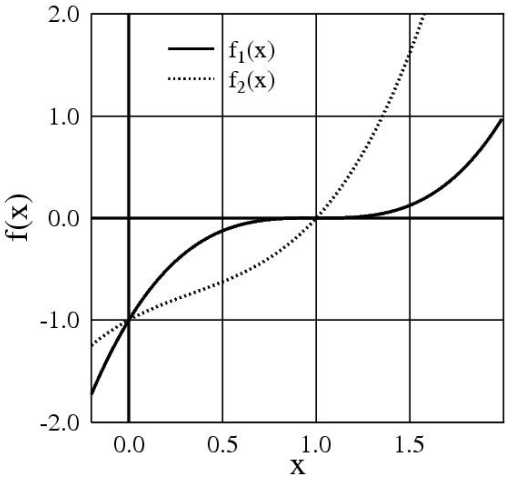
Nonlinear equations: multiple roots

- Example: Finding root for the functions

$$f_1(x) = (x-1)^3$$
$$f_2(x) = x^3 - x^2 + x - 1$$

using the NR routines **`zbrent`** (bisection+inverse quadratic), **`rtsec`** (secant method), **`rtsafe`** (bisection+Newton), **`rtbis`** (bisection) and pure Newton method.

- Both functions have one real root $x^* = 1$.
- Multiplicities are $m = 3$ for f_1 and $m = 1$ for f_2.

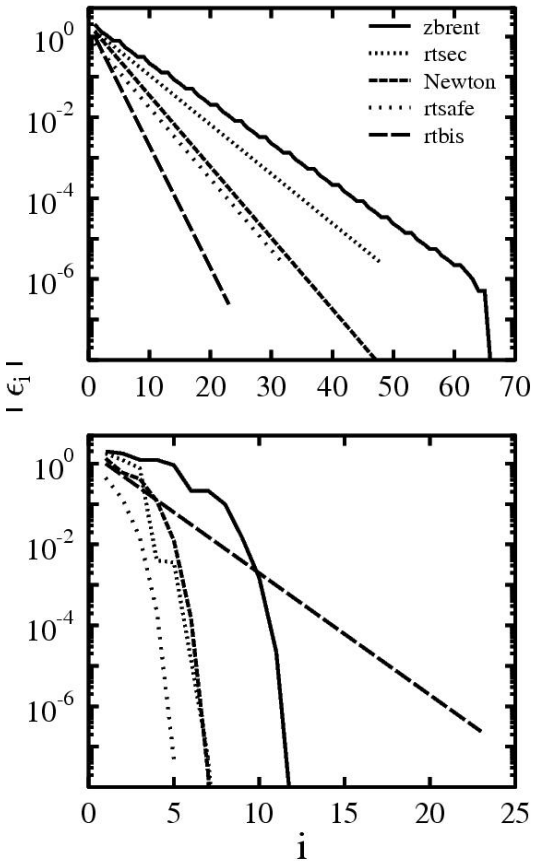


Nonlinear equations: multiple roots

- Error in root position as a function of iteration step.

- Convergence of bisection method does not depend on the odd multiplicity. For even multiplicity bisection method can not be used.

- Other methods slow down when multiplicity is increased.



Upper figure f_1, lower figure f_2.

Nonlinear equations: multiple roots

- If $f'(x)$ can be computed then the function can be expressed in such a way that the bisection method can also be used for even multiple roots.

  - Let the multiplicity of root x^* be m:

$$
\begin{aligned}
f(x) &= (x - x^*)^m g(x) \\
f'(x) &= (x - x^*)^{m-1} G(x) \\
G(x) &= m g(x) + (x - x^*) g'(x) \\
G(x^*) &= m g(x^*) \neq 0 .
\end{aligned}
$$

  - This means that even roots of $f(x)$ are odd roots of $f'(x)$.

  - If we write

$$u(x) = \frac{f(x)}{f'(x)}$$

  then near the root

$$u(x) = (x - x^*) H(x)$$

  where $H(x) = \dfrac{g(x)}{G(x)}, \quad H(x^*) = \dfrac{1}{m}, \quad m \neq 0$

Nonlinear equations: multiple roots

- So the search for multiple root is replaced by the search of simple root. (Well $u(x)$ has problems when $f'(x) = 0$).

- Example: $f(x) = xe^{-x} - e^{-1}$

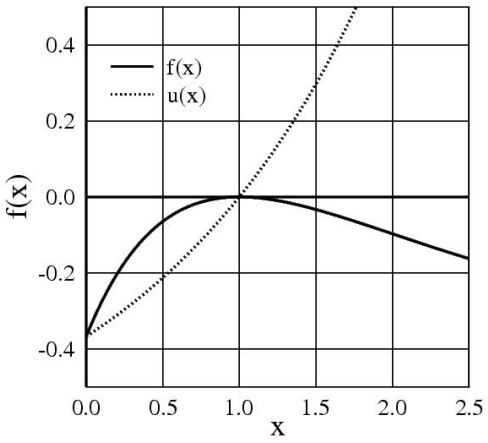
  - $x^* = 1$, $m = 2$ (This is easily seen when setting $x = 1$. Then $f(1) = 0$ and $f^{(m)}(1) \neq 0$ when $m > 1$.

  - Form the function $u(x)$

$$u(x) = \frac{f(x)}{f'(x)} = \frac{xe^{-x} - e^{-1}}{e^{-x}(1 - x)}$$

  and its derivative

$$u'(x) = \frac{1 - e^{x-1}}{1 - x} + \frac{x - e^{x-1}}{(1 - x)^2}$$



Nonlinear equations: multiple roots

- Newton's method with initial guess $x = 2$:

```
i      xi               u(xi)              abs(ei)

1   1.281718172      0.7182818285      0.2817181715
2   1.025236738      0.1550732730      0.2523673838E-01
3   1.000211406      0.1272519113E-01  0.2114062102E-03
4   1.000000015      0.1057105538E-03  0.1489881019E-07
5   1.000000000      0.1490351257E-07  0.4702349621E-11
6   1.000000000       0.000000000      0.4702349621E-11
```

- Also the secant method by using $u(x)$ gives the root after a few iterations:

```
i     xi               u(xi)             abs(ei)

1   1.513440361      0.3069293249      0.5134403609
2   1.150395478      0.7911365175E-01  0.1503954779
3   1.024320717      0.1225954387E-01  0.2432071733E-01
4   1.001201434      0.6009576680E-03  0.1201434043E-02
5   1.000009719      0.4859604479E-05  0.9719171624E-05
6   1.000000004       0.000000000      0.3886029276E-08
```


Nonlinear equations

• What method to use?

- Robustness by using multiple methods.

- If you can't or don't want to compute the derivative of the function:

  Bisection method combined with e.g. parabola method (parabola through three previous points; generalization of the secant method.

- Derivative available:

  Bisection method combined with Newton's method.

- Special attention to multiple roots.

• Things are much easier if you know your function.

  Plot it with different parameter values!

• In Matlab function **`fzero`** finds a root of a function.
• See also GSL documentation for root finding routines.
• A Fortran 77 package MINPACK contains, in addition to minimization, also root finding routines for one or multidimensional problems; see **http://www.netlib.org/minpack/**

Nonlinear equations: groups of equations

• Solving groups of equations is much harder than only one

- Strategy: generalize 1D methods

- Bisection method can not be generalized but Newton's can

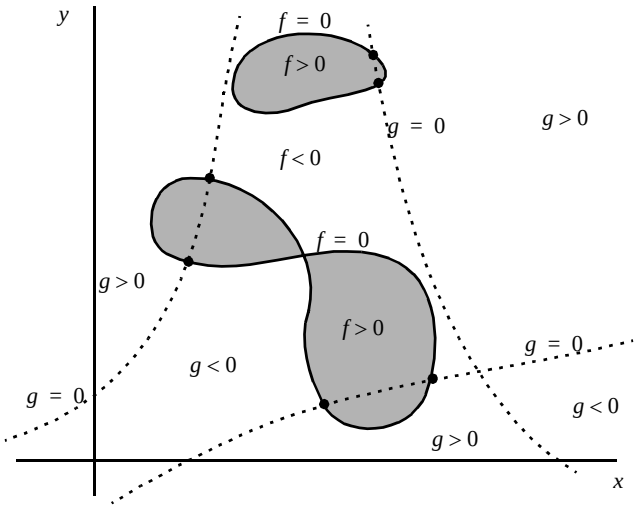
- Let's study a 2D group of equations:

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

- Zeros of f and g form curves on xy plane.

- Intersections of these points form the solutions

- Nothing special in these points → must go through all the curves to find the solutions



Nonlinear equations: groups of equations

• The most simple method: generalization of 1D Newton

- Zeros of N functions of N variables sought:

$$F_i(x_1, x_2, \ldots, x_N) = 0\,, \qquad i = 1, 2, \ldots, N$$

- In vector notation

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}$$

- Taylor's series around point $\mathbf{x}$:

$$F_i(\mathbf{x} + \delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial F_i}{\partial x_j}\delta x_j + O(\delta\mathbf{x}^2)$$

- Define the Jacobian matrix:

$$J_{ij} = \frac{\partial F_i}{\partial x_j}$$

- Now the Taylor's series can be written as

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x} + O(\delta\mathbf{x}^2)$$

Nonlinear equations: groups of equations

- Truncate the series (drop $O(\delta\mathbf{x}^2)$ terms):

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x}$$

- Now the equation reads as

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = 0 \quad \rightarrow \quad \mathbf{F}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x} = 0$$

$$\rightarrow \quad \mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{F}(\mathbf{x})$$

- How to intepret this?

  - By solving $\delta\mathbf{x}$ from the last equation we get a correction that takes us closer to the solution of the equations.
  - During iteration solve $\delta\mathbf{x}$ by using e.g. LU factorization and add the correction to get the new iterate:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \delta\mathbf{x}$$

  - Same stopping criteria can be used as in the 1D case: function values and change in vector $\mathbf{x}$.

Nonlinear equations: groups of equations

- The essential parts of a routine[1] performing Newton's method in meny dimensions

```
void mnewt(int ntrial, float x[], int n, float tolx, float tolf)
    {
            int k,i,*indx;
            float errx,errf,d,*fvec,**fjac,*p;
            // Allocate memory for arrays
            indx=ivector(1,n);
            dx=vector(1,n);
            fvec=vector(1,n);
            fjac=matrix(1,n,1,n);

            for (k=1;k<=ntrial;k++) {
                    usrfun(x,n,fvec,fjac); // Routine calculates function values and the Jacobian
                    errf=0.0;
                    for (i=1;i<=n;i++) errf += fabs(fvec[i]);
                    if (errf <= tolf) return; // Check the function value criterion
                    for (i=1;i<=n;i++) dx[i] = -fvec[i];
                    ludcmp(fjac,n,indx,&d); // Do LU decomposition
                    lubksb(fjac,n,indx,dx);  // Calculate the solution
                    errx=0.0;
                    for (i=1;i<=n;i++) {
                            errx += fabs(dx[i]);
                            x[i] += dx[i];
                    }
                    if (errx <= tolx) return; // Check the dx criterion
            }
            return;
    }
```

1. Modified from the corresponding NR routine

Nonlinear equations: groups of equations

- Example: equations

$$\begin{cases} F_1(x_1, x_2) = (x_1 - 1)^2 + \frac{1}{2}(x_2 - 2)^2 - 1 = 0 \\ F_2(x_1, x_2) = \left(x_1 - \frac{3}{2}\right)^2 + \frac{1}{2}\left(x_2 - \frac{9}{5}\right)^2 - 2 = 0 \end{cases}$$

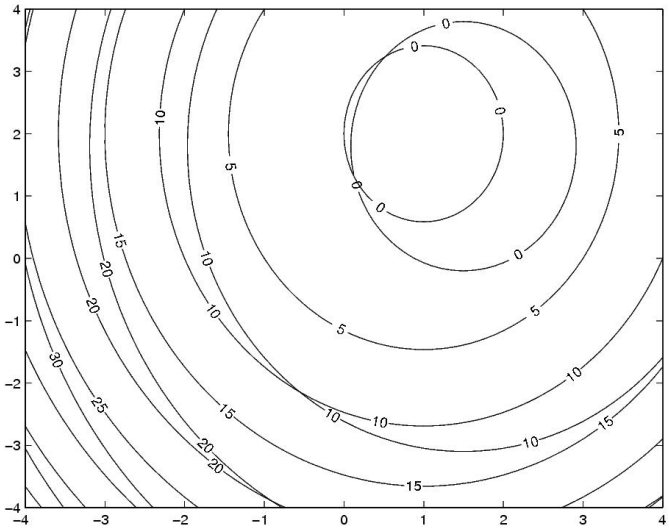
- Jacobian is now

$$J_{11} = \frac{\partial F_1}{\partial x_1} = 2(x_1 - 1) \quad J_{12} = \frac{\partial F_1}{\partial x_2} = x_2 - 1$$

$$J_{21} = \frac{\partial F_2}{\partial x_1} = 2\left(x_1 - \frac{3}{2}\right) \quad J_{22} = \frac{\partial F_2}{\partial 2} = x_2 - \frac{9}{5}$$

- Contour plot of the functions on the right.

- Apparently two solutions: $\mathbf{x}_1^* \approx (0.1, 1.3)$, $\mathbf{x}_2^* \approx (0.5, 3.2)$



Nonlinear equations: groups of equations

- The routine that calculates the function values and Jacobian is

```
void usrfun(float *x,int n,float *fvec,float **fjac)
{
  fvec[1] = (x[1]-1.0)*(x[1]-1.0) + 0.5*(x[2]-2.0)*(x[2]-2.0) - 1.0 ;
  fvec[2] = (x[1]-1.5)*(x[1]-1.5) + 0.5*(x[2]-1.8)*(x[2]-1.8) - 2.0 ;
  fjac[1][1] =  2.0*(x[1]-1.0);
  fjac[1][2] =  (x[2]-2.0);
  fjac[2][1] =  2.0*(x[1]-1.5);
  fjac[2][2] =  (x[2]-1.8);
}
```

- And the essential parts of the main program

```
#define NTRIAL 20
#define TOLX 1.0e-6
#define TOLF 1.0e-6
#define N 2
#define KMAX 3
#define KKMAX 3
#define SCALE 1.0

int main(void)
{
  int i,j,k,kk;
  float *x,*fvec,**fjac;

  fjac=matrix(1,N,1,N);
  fvec=vector(1,N);
  x=vector(1,N);
  for (kk=-KKMAX;kk<=KKMAX;kk++) {
    for (k=-KMAX;k<=KMAX;k++) {
      x[1]=SCALE*(kk+0.1);
      x[2]=SCALE*k;
      printf("%6.2f %6.2f     ",x[1],x[2]);
      mnewt(NTRIAL,x,N,TOLX,TOLF);
      usrfun(x,N,fvec,fjac);
      printf("%10.6f %10.6f  %15.6g %15.6g\n",
             x[1],x[2],fvec[1],fvec[2]);
    }
  }
  printf("\n");
  return 0;
}
```

Nonlinear equations: groups of equations

- Output:

x10	x20	x1*	x2*	F1	F2
-2.90	-3.00	0.130363	1.301815	1.20086e-07	1.18298e-07
-2.90	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-2.90	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-2.90	0.00	0.130363	1.301815	1.20086e-07	1.18298e-07
-2.90	1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-2.90	2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-2.90	3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	-3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	0.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-1.90	3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	-3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	0.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
-0.90	3.00	0.517785	3.238926	1.51653e-07	1.94568e-07
0.10	-3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
0.10	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
0.10	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
0.10	0.00	0.130363	1.301815	1.09384e-08	1.80909e-08
0.10	1.00	0.130363	1.301815	1.46003e-07	1.59116e-07
0.10	2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
0.10	3.00	0.517785	3.238926	3.89552e-07	4.20546e-07
1.10	-3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
1.10	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
1.10	-1.00	0.130363	1.301814	4.99362e-07	5.00555e-07
1.10	0.00	0.130363	1.301815	1.09384e-08	1.80909e-08
1.10	1.00	0.130363	1.301815	1.20086e-07	1.18298e-07
1.10	2.00	0.517785	3.238926	9.41686e-08	7.74793e-08
1.10	3.00	0.517785	3.238926	1.51653e-07	1.94568e-07
2.10	-3.00	0.130363	1.301815	1.20086e-07	1.18298e-07
2.10	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08

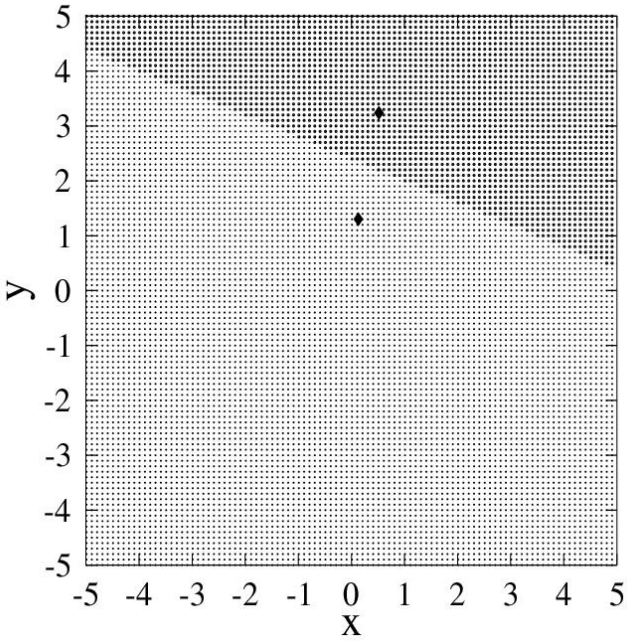
2.10	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
2.10	0.00	0.130363	1.301814	2.5515e-07	2.59323e-07
2.10	1.00	0.130363	1.301814	3.64298e-07	3.59529e-07
2.10	2.00	0.517785	3.238926	1.51653e-07	1.94568e-07
2.10	3.00	0.517785	3.238926	1.51653e-07	1.94568e-07
3.10	-3.00	0.130363	1.301815	1.09384e-08	1.80909e-08
3.10	-2.00	0.130363	1.301815	1.09384e-08	1.80909e-08
3.10	-1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
3.10	0.00	0.130363	1.301815	1.09384e-08	1.80909e-08
3.10	1.00	0.130363	1.301815	1.09384e-08	1.80909e-08
3.10	2.00	0.517785	3.238926	1.51653e-07	1.94568e-07
3.10	3.00	0.517785	3.238926	1.51653e-07	1.94568e-07

Nonlinear equations: groups of equations

- Both solutions are found

- By printing the number of iterations one can see that the iterates converge towards the solution after less than 10 steps.

- By scanning xy plane we can study to which solution each initial point converges:



Nonlinear equations: groups of equations

- We will later learn that there are efficient methods to find the *minimum* of a scalar function.
- Could these be used for solving equations?

- In minimization the zero of the gradient is looked for.
- Difference between equation solving and minimzation:
  In minimization there is a natural direction: *downhill*.
  In equation solving there is no such direction.

- Turn the equation solving into a minimization problem:
  - Solution of the equation $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ minimizes the sum

$$F(\mathbf{x}) = \sum_{i=1}^{N} [F_i(\mathbf{x})]^2$$

  - The problem here is that we may end up to a local minimum that is not the solution.

- More efficient is to combine Newton and the abovementioned minimization:
  Newton's fast convergence near solution+
  robustness of the minimization method to find the minimum almost regardless of the starting point.

  - Newton: iterate the corrections $\mathbf{x}_{i+1} = \mathbf{x}_i + \delta\mathbf{x}$, where $\delta\mathbf{x} = -\mathbf{J}^{-1} \cdot \mathbf{F}$

Nonlinear equations: groups of equations

- First perform a Newton step and accept it if $|\mathbf{F}|^2 = \mathbf{F} \cdot \mathbf{F}$ decreases
  - This is the same as in minimizing $f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F}$
  - Every solution of the equation $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ minimizes f but f may have local minima
    $\rightarrow$ we can't use only the minimization method
  - Newton step $\delta\mathbf{x} = -\mathbf{J}^{-1} \cdot \mathbf{F}$ is downhill from the point of view of f because

$$\nabla f \cdot \delta\mathbf{x} = (\mathbf{F} \cdot \mathbf{J}) \cdot (-\mathbf{J}^{-1} \cdot \mathbf{F}) = -\mathbf{F} \cdot \mathbf{F} < 0$$

  - Now at every iteration step we calculate $\delta\mathbf{x}$ from the Jacobian.
  - If f does not decrease we backtrack the step so that f decreases.
  - So the new step is

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda\delta\mathbf{x}, \quad 0 < \lambda \leq 1$$

    and λ is chosen so that $f(\mathbf{x}_i + \lambda\delta\mathbf{x})$ decreases sufficiently.
  - A suitable criterion might be for example that the average rate of decrease of f is at least some fraction α of the initial decrease of $\nabla f \cdot \delta\mathbf{x}$

$$f(\mathbf{x}_{i+1}) \leq f(\mathbf{x}_i) + \alpha\nabla f \cdot (\mathbf{x}_{i+1} - \mathbf{x}_i)$$

Nonlinear equations: groups of equations

- Example: equations

$$\begin{cases} x_1^2 + x_2^2 - 2 = 0 \\ e^{x_1 - 1} + x_2^3 - 2 = 0 \end{cases}$$

- One solution is $x_1 = x_2 = 1$
- Solution using the NR routine **newt**

x10	x20	x1	x2	f1	f2
0.00000	1.00000	-0.713748	1.220887	3.09046e-07	-1.83345e-08
0.00000	1.50000	-0.713757	1.220893	2.9386e-05	2.70813e-05
0.00000	2.00000	-0.713747	1.220890	6.34566e-06	1.40348e-05
0.50000	1.00000	1.000000	1.000000	2.38419e-07	1.19209e-07
0.50000	1.50000	1.000000	1.000001	8.34466e-07	2.08616e-06
0.50000	2.00000	1.000000	1.000000	2.38419e-07	5.96047e-07
1.00000	1.00000	1.000000	1.000000	0	0
1.00000	1.50000	0.999999	1.000001	1.07289e-06	2.68221e-06
1.00000	2.00000	1.000000	1.000000	3.57628e-07	8.9407e-07
1.50000	1.00000	1.000002	1.000000	4.52996e-06	3.21865e-06
1.50000	1.50000	0.999999	1.000002	1.43051e-06	3.8147e-06
1.50000	2.00000	1.000000	1.000000	4.76837e-07	1.19209e-06
2.00000	1.00000	0.999998	1.000003	3.09945e-06	7.74863e-06
2.00000	1.50000	0.999994	1.000013	1.32324e-05	3.28427e-05
2.00000	2.00000	0.999999	1.000001	1.07289e-06	2.68221e-06

- Newton's method needs the Jacobian.
- If you can't calculate it then you have to resort to e.g. the generalization of the secant method; consult for example Numerical Recipes.

Nonlinear equations: roots of polynomials by matrix eigenvalues

• Remember that eigenvalues of a matrix $\mathbf{A}$ are the roots of its characteristic polynomial $f_{\mathbf{A}}(\lambda) = \det(\mathbf{A} - \lambda\mathbf{1})$:

$$f_{\mathbf{A}}(\lambda) = 0$$

- Assume we have a matrix of the form

$$\mathbf{A} = \begin{bmatrix} -p_{N-1} & -p_{N-2} & \cdots & -p_1 & -p_0 \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix} .$$

- Its characteristic polynomial is written as

$$f(\lambda) = \det \begin{bmatrix} -p_{N-1} - \lambda & -p_{N-2} & \cdots & -p_1 & -p_0 \\ 1 & -\lambda & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & -\lambda \end{bmatrix} = (-1)^N [\lambda^N + p_{N-1}\lambda^{N-1} + \dots + p_0] .$$

- So, roots of any polynomial: $a_N x^N + a_{N-1} x^{N-1} + \dots + a_0 = 0$ can be calculated as eigenvalues of matrix $\mathbf{A}$ when we define: $p_0 = \frac{a_0}{a_N}, p_1 = \frac{a_1}{a_N}, \dots, p_{N-1} = \frac{a_{N-1}}{a_N}$

- Matlab/Octave function **`roots`** computes roots of a polynomial in this way.