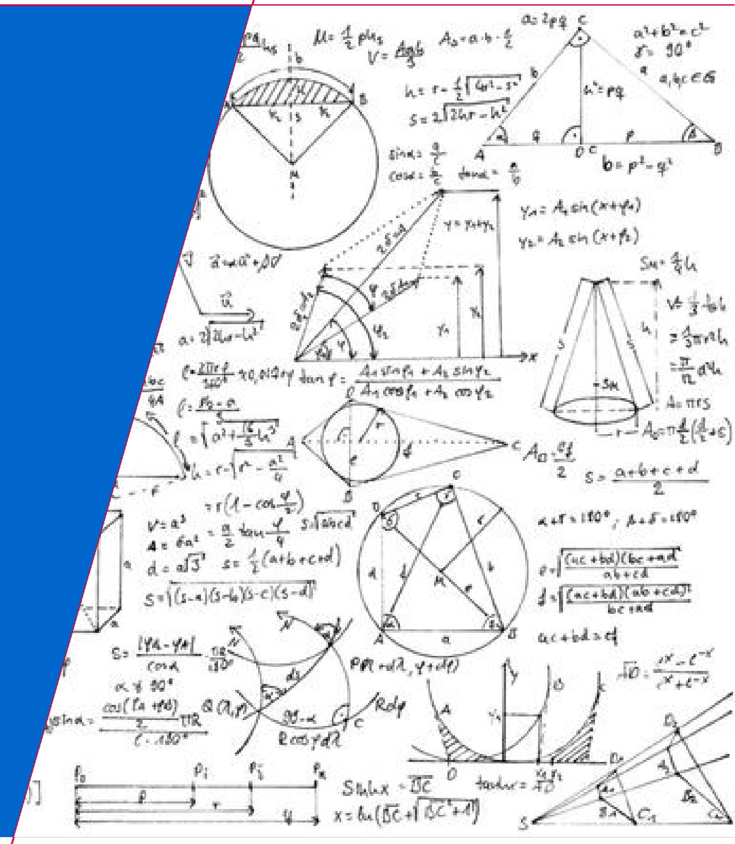
Numerical methods for Chemical Engineers:

Non-linear equations

Prof.dr.ir. Martin van Sint Annaland
Dr.ir. Ivo Roghair



Chemical Process Intensification

TU/e Technische Universiteit **Eindhoven** University of Technology

Where innovation starts

Content

- **How to solve:**

  $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$ for arbitrary functions ***f***

  "Root finding"

  (i.e. move all terms to the left)

  - **One dimensional case:** $f(x) = 0$

    "Bracket or 'trap' a root between bracketing values, then hunt it down like a rabbit."

  - **Multi-dimensional case:** $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$

    - *N* equations in *N* unknowns:
      You can only *hope* to find a solution.
      It may have no (real) solution, or more than one solution!
    - Much more difficult!!
      "You never know whether a root is near, unless you have found it"

Outline

- **One-dimensional case:**
  - Direct iteration method
  - Bisection method
  - Secant and false position method
  - Brent's method
  - Newton-Raphson method

Do not use routines as black boxes without understanding them!!!

- **Multi-dimensional case:**
  - Newton-Raphson method
  - Broyden's method

➢ Introduction to underlying ideas and algorithms
➢ Exercises in how to program the methods in Excel and MATLAB.

General idea

- **Root finding proceeds by iteration:**
  - Start with a good initial guess (crucially important!!)
  - Use an algorithm to improve the solution until some predetermined convergence criterion is satisfied

- **Pitfalls:**
  - Convergence to the wrong root…
  - Fails to converge because there is no root…
  - Fails to converge because your initial estimate was not close enough…

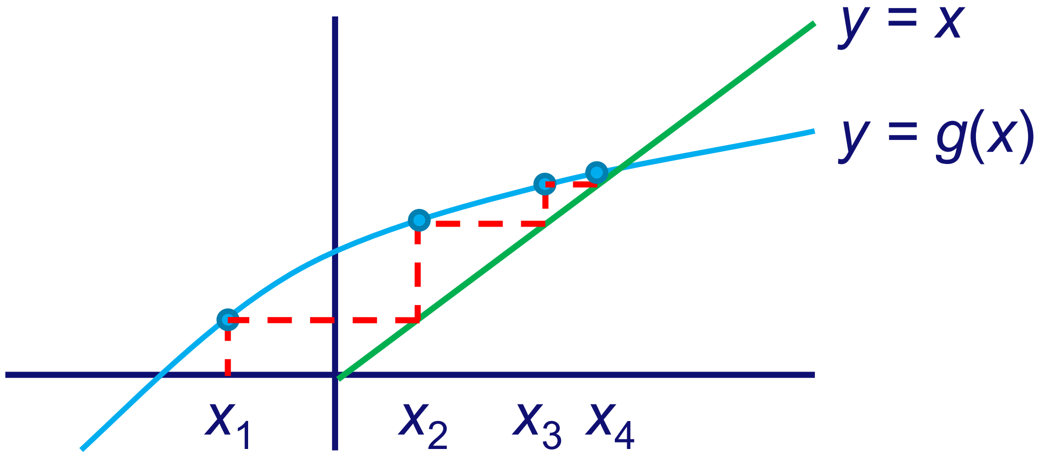
Hamming's motto:
the purpose of computing is insight, not numbers!!

➢ It never hurts to inspect your function graphically
➢ Pay attention to carefully select initial guesses

Direct iteration method/successive substitutions

- **Rewrite** $f(x) = 0 \;\Rightarrow\; x = g(x)$
  - Start with an initial guess: x_1
  - Calculate new estimate with: $x_2 = g(x_1)$
  - Continue iteration with: $x_{i+1} = g(x_i)$
  - Proceed until: $|x_{i+1} - x_i| < \epsilon$

  When the process converges, taking a smaller value for $\in$ results in a more accurate solution, however more iterations need to be performed.



Direct iteration method

Exercise 1: Find the root of $x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method

- **Rewrite as:** $x = \left(3x^2 + 3x + 4\right)^{\frac{1}{3}}$
  - Solve in Excel
  - Solve in Matlab

- **Rewrite as:** $x = \left(x^3 - 3x^2 - 4\right)/3$
  - Solve in Excel
  - Solve in Matlab

Intermezzo: functions revisited

- In MATLAB you can define your own functions, allowing re-use of certain functionalities. We now define the mathematical function in a new file f.m:

$$f(x) = x^2 + \exp(x)$$

```
function y = f(x)
y = x.^2 + exp(x);
end
```

- The first line contains the function keyword
- y is defined as output, x is defined as input
- The computation can use x as a scalar as well as a vector
  - If x is a vector, y is also a vector.

Anonymous functions

- If you do not want to create a file, you can create an anonymous function

```
>> g = @(x) (x.^2 + exp(x))
```

  - g: the name of the function
  - @: indicator of a function handle
  - x: the input argument

```
>> g(0:0.1:1)
```

- A function handle points to a function, but it behaves as a variable. You can pass a function handle as an argument!

Passing functions in Matlab

- For example: to solve $f(x) = x^2 - 4x + 2 = 0$ numerically, we can write a function that returns the value of f:

```
function f = MyFunc(x)
  f = x.^2 – 4*x + 2;
return
```

(Note: case sensitive!!)

- The function handle can be used as an alias:

```
>> f = @MyFunc; a = 4; b = f(a)
```

- We can then call a solving routine (e.g. fzero):

```
>> ans = fzero(@MyFunc,5)
>> fzero(@(x) x.^2-4*x+2,5)
```

Passing functions in Matlab

- We can also make **our own** function, that takes the function handle as an input (save as draw_my_function.m):

```
function [] = draw_my_function(func)
% Draws a function in the range [0 10] using 20 data
% points. 'func' is a function handle that can point to
% any actual function.
x = linspace(0, 10, 20);
y = func(x);
plot(x,y,"-o");
end
```

- Now we can call the function with a function handle, which points to an anonymous function or a common function:

```
>> f = @(x) (x.^2 -  4*x + 2);
>> draw_my_function(f)
>> ezplot(f, [0 10])
```


TU/e Technische Universiteit Eindhoven University of Technology

Direct iteration method

Exercise 1: Find the root of $x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method

- **Rewrite as:** $x = \left(3x^2 + 3x + 4\right)^{\frac{1}{3}}$
  - Solve in Excel
  - Solve in Matlab

- **Rewrite as:** $x = \left(x^3 - 3x^2 - 4\right)/3$
  - Solve in Excel
  - Solve in Matlab

Direct iteration method

Exercise 1: Find the root of $x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method in Excel

$$x = \left(3x^2 + 3x + 4\right)^{\frac{1}{3}}$$

=(3*x1^2+3*x1+4)^(1/3)

1	2.5
2	3.11584
3	3.489024
[illegible]	[illegible]708113
[illegible]	[illegible]834375
6	3.906376
7	3.94719
8	3.970248
9	3.98325
10	3.990573

Converges!

$$x = \left(x^3 - 3x^2 - 4\right)/3$$

=(x1^3-3*x1^2-4)/3

1	2.5
2	-2.375
3	-11.4395
4	-631[illegible]
5	-8.4E[illegible]
6	-2E+23
7	-2.6E+69
8	-6E+207
9	#NUM!
10	#NUM!

Diverges!

Direct iteration method

Exercise 1: Find the root of $x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method in Matlab

With simple script:

```
x = 2.5;
fprintf("i: %d, x: %e\n",0,x);
for i=1:20
  x = (3*x^2+3*x+4)^(1/3);
  fprintf("i = %d: x = %f\n",i,x);
end
```

Not very flexible/reusable ⇒ use functions!

Direct iteration method

Exercise 1: Find the root of $x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method in Matlab

First define the functions

```
function [y] = MyFnc1(x)
   y = (3*x^2 + 3*x + 4)^(1/3);
end
```

```
function [y] = MyFnc2(x)
   y = (x^3 - 3*x^2 - 4)/3;
end
```

Direct iteration method

Exercise 1: Find the root of $x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method in Matlab

Make function to carry out Direct Iteration algorithm:

```
function [y,it] = DirectIterationMethod(g,x,eps)
%Solves x = g(x) with x as initial guess until the
%difference with the next iteration is smaller than eps
%or when the number of iterations exceeds itmax
itmax = 100;
it = 0;
y = g(x);
fprintf("it = %d: x = %f\n",it,y);
while ((abs(y-x)>eps) && (it<itmax))
    it = it + 1;
    x = y;
    y = g(x);
    fprintf("it = %d: x = %f\n",it,y);
end
```

Direct iteration method

Exercise 1: Find the root of $x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method in Matlab

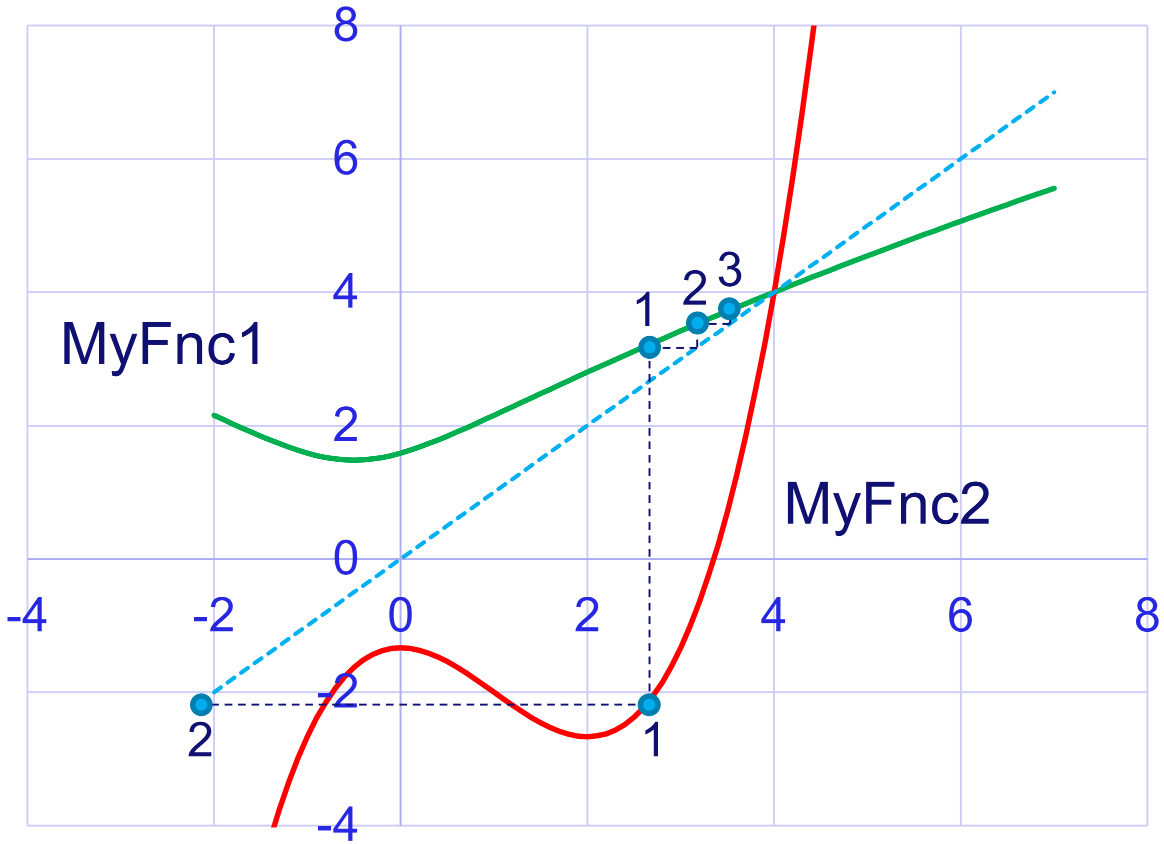
Call Direct Iteration function with:

```
>> DirectIterationMethod(@MyFnc1,2.5,1e-3);

>> DirectIterationMethod(@MyFnc2,2.5,1e-3);
```

Why does it converge with MyFnc1 and diverge with MyFnc2?

Direct iteration method

Exercise 1: Find the root of $f(x) = x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method



Method only works when $|g'(x_i)| < 1$

And even then not very fast ...

$$x = g(x) \approx g(x_i) + g'(x_i)(x - x_i)$$

$$g(x_{i+1}) = g(x_i) + g'(x_i)(x_{i+1} - x_i)$$

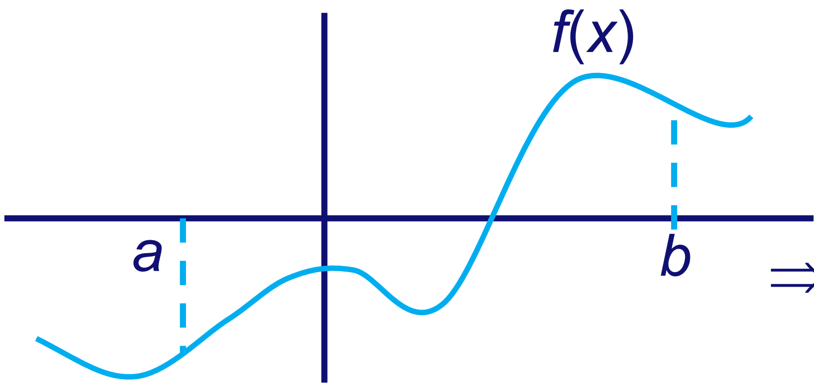
$$x_{i+2} = x_{i+1} + g'(x_i)(x_{i+1} - x_i)$$

$$|x_{i+2} - x_{i+1}| = |g'(x_i)||x_{i+1} - x_i|$$

Convergence $\Rightarrow$ $|g'(x_i)| \leq 1$

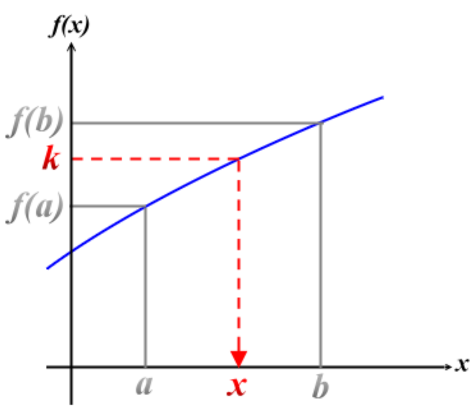
Bracketing

Bracketing a root = knowing that the function changes sign in an identified interval



A root is bracketed in the interval (a,b), if f(a) and f(b) have opposite signs

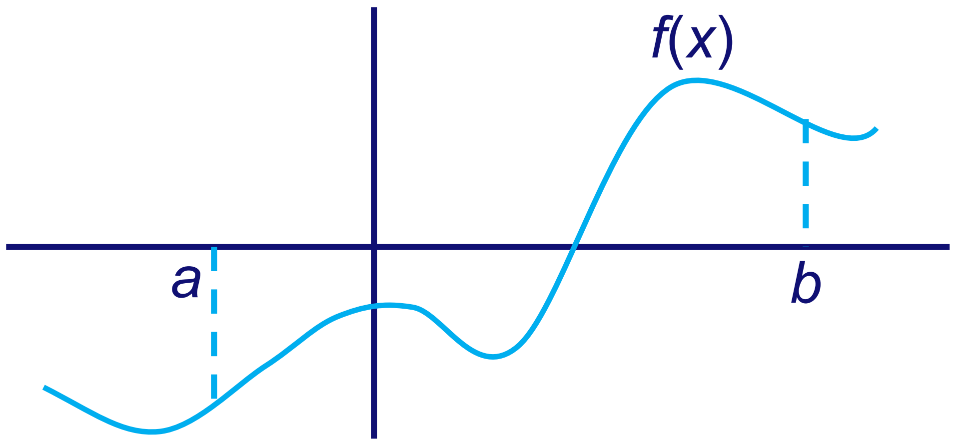
$\Rightarrow$ At least one root must lie in this interval, if the function is continuous



Intermediate Value Theorem
If *f(x)* is *continuous* on [a,b] and k is a constant that lies between *f(a)* and *f(b),* then there is a value $x \in [a,b]$ such that $f(x) = k$

Bracketing

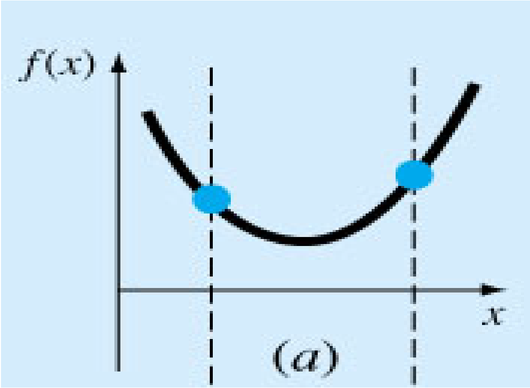
Bracketing a root = knowing that the function changes sign in an identified interval



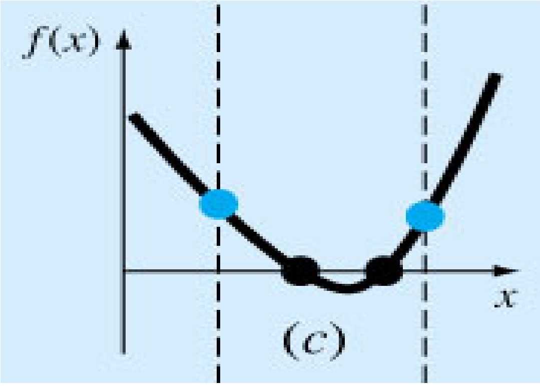
- **General best advise:**
  - Always bracket a root before trying to converge…
  - Never allow your iteration method to get outside the best bracketing bounds…

General idea

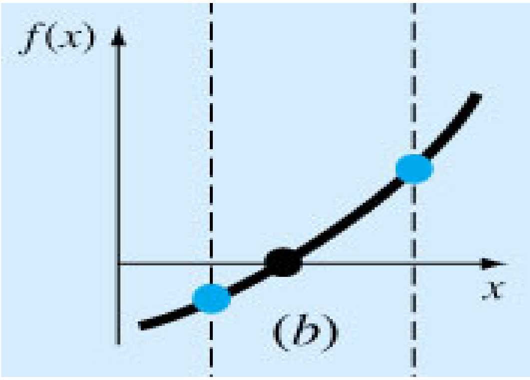
- **Examples of pitfalls of bracketing…**



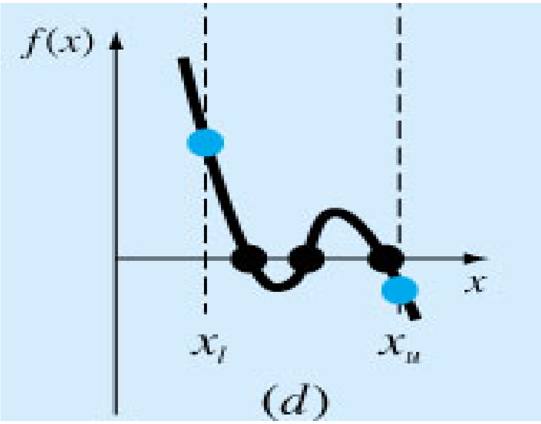
No answer (no root)



Oops!! (two roots!!)



Nice case (one root)



Three roots (might work for a while!)

Bracketing

Exercise 2:

- **Write a function in MATLAB to bracket a function given an initial guessed range x_1 and x_2. (via expansion of the interval)**
- **Write a program to find out how many roots exist (at minimum) in the interval x_1 and x_2.**

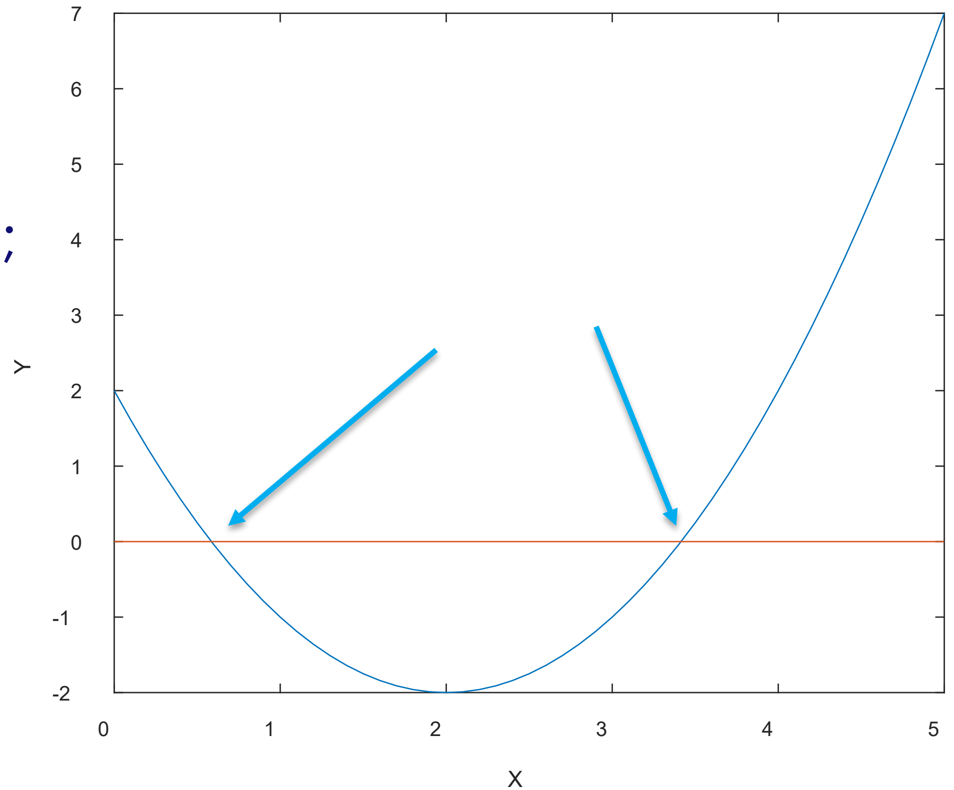
Of course these functions can then be combined to create a function that returns bracketing intervals for different roots.

Bracketing

- **Exercise 2: Function to bracket a function**

If possible, first make a graph: for example via

```
>> x=0:0.1:5;
>> y=x.^2-4*x+2;
>> figure;
>> plot(x,y,x,zeros(size(x));
>> axis tight; box on;
```



Makes immediately clear that there are two roots.

$x_1 = 2 - \sqrt{2} \approx 0.59$

$x_2 = 2 + \sqrt{2} \approx 3.41$

- **Exercise 2: Function to bracket a function**

```
function found = brac(func, x1, x2)
ntry = 50;
factor = 1.6;

found = false;
if (x1~=x2)
  f1 = func(x1);
  f2 = func(x2);
  for i = 1:ntry
    if (f1*f2<0)
      found = true
      break;
    end;
    if (abs(f1)<abs(f2))
      x1 = x1 + factor*(x1-x2);
      f1 = func(x1);
    else
      x2 = x2 + factor*(x2-x1);
      f2 = func(x2);
    end;
  end;
else
  disp('Bad initial range!');
end;

if found
  disp(sprintf('The bracketing interval = [%f, %f]\n', [x1,x2]));
else
  disp('No bracketing interval found!');
end;
return
```

a function to expand the interval (x_1,x_2) maximally $2^{50} \sim 10^{15}$, untill a root is found

returns true when root is found and false otherwise

displays results

Bracketing

• Exercise 2: Function to bracket a function

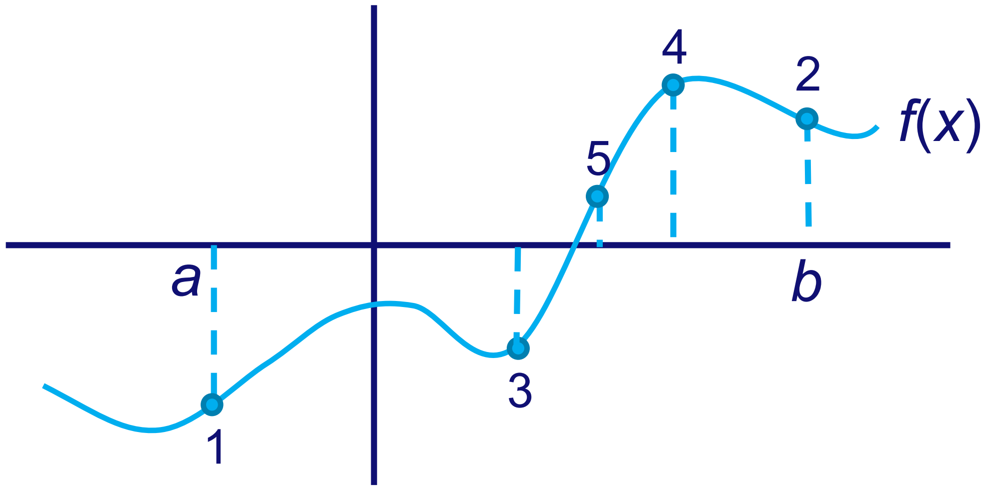
```
function nroot = brak(func, x1, x2, n);
nroot = 0;
dx = (x2 - x1)/n;
x = x1;
fp = func(x1);
for i = 0:n
  x = x + dx;
  fc = func(x);
  if (fc*fp<=0)
    nroot = nroot + 1;
    xb1(nroot) = x - dx;
    xb2(nroot) = x;
  end;
  fp = fc;
end;
if n>0
  for i = 1:nroot
    disp(sprintf('Root %d in bracketing interval [%f, %f]', [i,xb1(i),xb2(i)]));
  end
else
  disp('No roots found!');
end;

return;
```

a function to subdivide the interval (x_1,x_2) in *n* parts and examines whether there is at least one root

Returns the left and right boundaries of the intervals of the roots in xb1, xb2

Bisection method

- **Bisection algorithm:**
  - Over some interval it is known that the function will pass through zero, because the function changes sign
  - Evaluate function value at the interval's midpoint and examine its sign
  - Use the midpoint to replace whichever limit has the same sign



It cannot fail, but relatively slow convergence!

Bisection

Exercise 3:

- **Write a function in Excel to find a root of a function using the bisection method**
  - Assume that an initial bracketing interval (x_1, x_2) is provided
  - Also the required tolerance is specified (which tolerance?)
  - Also output the required number of iterations

- **Do the same in MATLAB**

Bisection method

- **Exercise 3: Bisection method in Excel**

it	x1	x2	f1	f2		xmid	fmid		interval size
0	-2	2	14	-2		0	2		4
1	0	2	2	-2		1	-1		2
[illegible]	0	1	[illegible]	-1		[illegible]	[illegible]		1
3	0.5	1	0.25	-1		[illegible]	[illegible]		0.5
[illegible]	[illegible]	[illegible]	0.25	[illegible]		[illegible]	[illegible]		0.25
[illegible]	[illegible]	[illegible]	0.25	[illegible]		[illegible]	[illegible]		0.125
[illegible]	[illegible]	[illegible]	0.066406	[illegible]		[illegible]	[illegible]		0.0625
7	0.5625	0.59375	0.066406	-0.02246		0.578125	0.021729		0.03125
8	0.578125	0.59375	0.021729	-0.02246		[illegible]	[illegible]		[illegible]
9	0.578125	0.585938	0.021729	-0.00043		[illegible]	[illegible]		[illegible]
10	0.582031	0.585938	0.010635	-0.00043		[illegible]	[illegible]		[illegible]
11	0.583984	0.585938	0.0051	-0.00043		[illegible]	[illegible]		[illegible]
12	0.584961	0.585938	0.002336	-0.00043		0.585449	0.000954		0.000977
13	0.585449	0.585938	0.000954	-0.00043		0.585693	0.000263		0.000488
14	0.585693	0.585938	0.000263	-0.00043		0.585815	-8.2E-05		0.000244
15	0.585693	0.585815	0.000263	-8.2E-05		0.585754	9.06E-05		0.000122
16	0.585754	0.585815	9.06E-05	-8.2E-05		0.585785	4.31E-06		6.1E-05
17	0.585785	0.585815	4.31E-06	-8.2E-05		0.5858	-3.9E-05		3.05E-05
18	0.585785	0.5858	4.31E-06	-3.9E-05		0.585793	-1.7E-05		1.53E-05
19	0.585785	0.585793	4.31E-06	-1.7E-05		0.585789	-6.5E-06		7.63E-06
20	0.585785	0.585789	4.31E-06	-6.5E-06		0.585787	-1.1E-06		3.81E-06
21	0.585785	0.585787	4.31E-06	-1.1E-06		0.585786	1.62E-06		1.91E-06
22	0.585786	0.585787	1.62E-06	-1.1E-06		0.585786	2.69E-07		9.54E-07
23	0.585786	0.585787	2.69E-07	-1.1E-06		0.585787	-4.1E-07		4.77E-07
24	0.585786	0.585787	2.69E-07	-4.1E-07		0.585786	-6.8E-08		2.38E-07
25	0.585786	0.585786	2.69E-07	-6.8E-08		0.585786	1E-07		1.19E-07
26	0.585786	0.585786	1E-07	-6.8E-08		0.585786	1.58E-08		5.96E-08

=IF(f1*fmid<0;x1;xmid)

=IF(f2*fmid<0;x2;xmid)

xmid = 0.5*(x1 + x2)

fmid = f(xmid)

Bisection method

- **Exercise 3: Bisection method in MATLAB**

```matlab
function [p] = bisection(f, x1, x2, tol_step, tol_func)
  f1 = f(x1);
  f2 = f(x2);
  fp = f2;
  if (f1*f2>0)
    error('Root must be bracketed!');
  else
    it = 1;
    while ((abs(fp)>tol_func) && (abs(x2 - x1)>tol_step))
      it = it + 1;
      p = 0.5*(x1 + x2);
      fp = f(p);
      if (f1*fp<0)
        x2 = p;
        f2 = fp;
      else
        x1 = p;
        f1 = fp;
      end
    end
    disp(sprintf('Root found in %d iterations at x = %e\n (function value = %e)', [it,p,fp]));
  end
end
```

Note1: We have used a criterion for the function value and the step size!

Note2: usually while loop needs protection for maximum number of iterations (but here bisection is sure to convergence…)

Root found in 25 iterations required. Can we do better?

```
>> bisection(@(x) x^2-4*x+2,0,2,1e-7,1e-7);
```

Bisection method

- **Required number of iterations?**
  - After each iteration the interval bounds containing the root decrease by a factor of 2:

$$\epsilon_{n+1} = \frac{1}{2}\epsilon_n \quad \Rightarrow \quad \boxed{n = \log_2 \frac{\epsilon_0}{tol}}$$

$\epsilon_0 =$ initial bracketing interval
$tol =$ desired tolerance

i.e. after 50 iterations the interval is decreased by factor $2^{50} = 10^{15}$!
(Mind machine accuracy when setting tolerance!)

  - Order of convergence = 1

$$\boxed{\epsilon_{n+1} = K(\epsilon_n)^m}$$

m = 1: linear convergence
m = 2: quadratic convergence

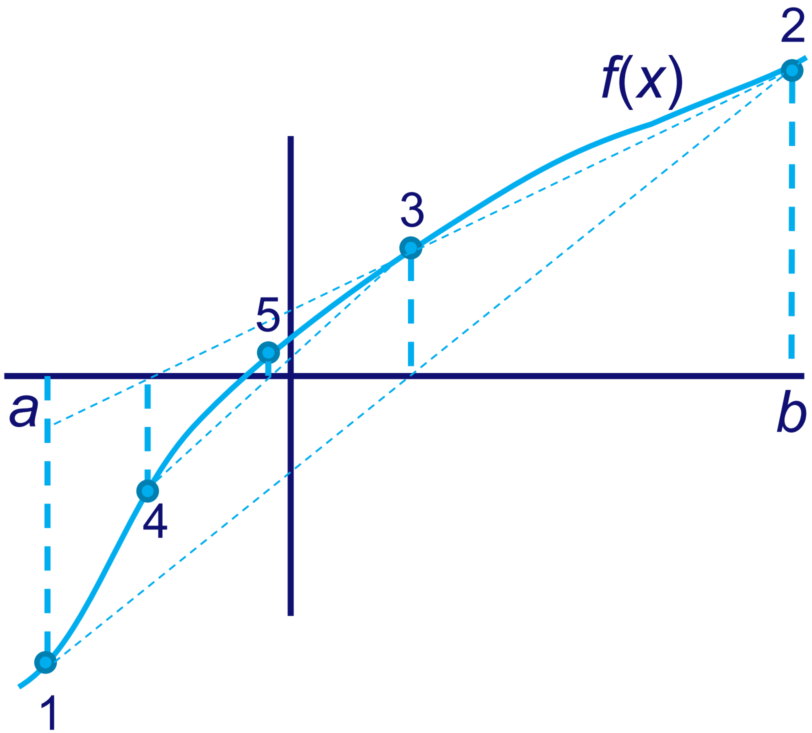
  - Must succeed:
    - More than root ⇒ bisection will find one of them
    - No root, but singularity ⇒ bisection will find singularity

Secant and False position method

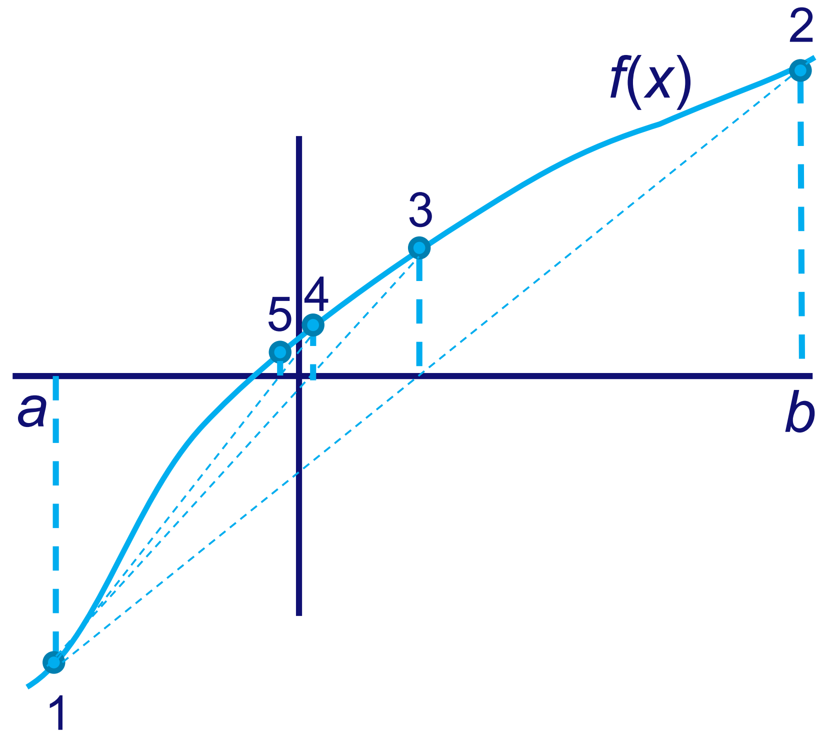
- **Secant/False position (= Regula Falsi) method**
  - Faster convergence (provided sufficiently smooth behaviour)
  - Difference with bisection method in choice of next point:
    - Bisection: mid-point of interval
    - Secant/False position: point where the approximating line crosses the axis
  - One of the boundary points is discarded in favor of the latest estimate of
    - Secant: retains the most recent of the prior estimates
    - False position: retains prior estimate with opposite sign, so that the points continue to bracket the root

Secant and False position method

Secant method



False position method



Secant: slightly faster convergence: $\lim_{n\to\infty} |\epsilon_{n+1}| = K|\epsilon_n|^{1.618}$

False position: guaranteed convergence

Secant and False position method

Exercise 4:

- **Write a function in Excel and MATLAB to find a root of a function using the Secant and the False position methods**
  - Assume that an initial bracketing interval (x_1, x_2) is provided
  - Also the required tolerance is specified
  - Also output the required number of iterations
  - Compare the bisection, false position and secant methods

Secant and False position method

Exercise 4:

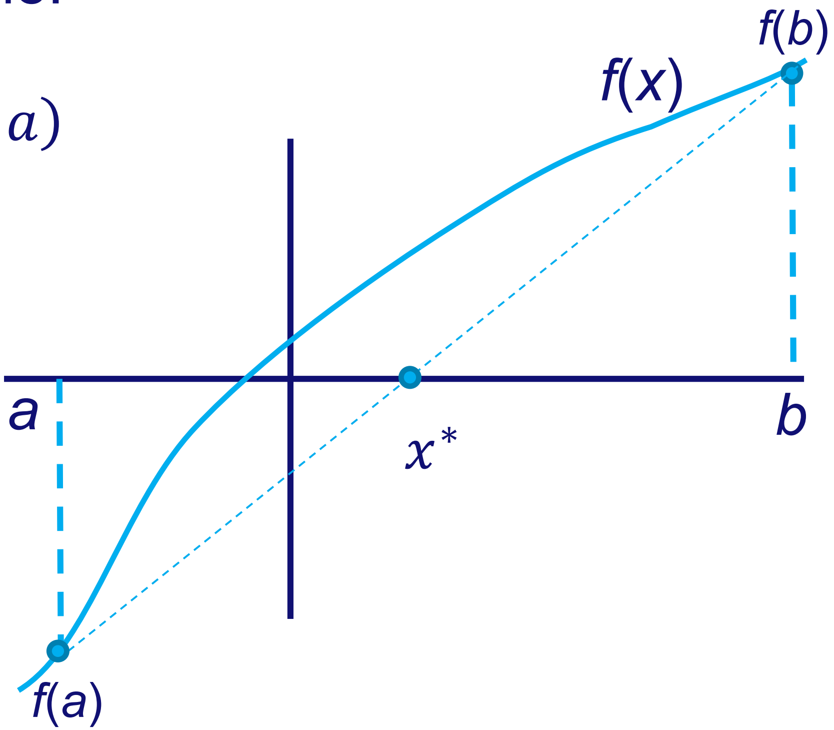
- **Determination of the abscissa of the approximating line:**
  - Determine the approximating line:

$$f(x) \approx f(a) + \frac{f(b) - f(a)}{b - a}(x - a)$$

  - Determine abscissa:

$$f(x^*) = 0$$

$$\Rightarrow \boxed{\begin{aligned} x^* &= a - \frac{f(a)(b-a)}{f(b) - f(a)} \\ &= \frac{af(b) - bf(a)}{f(b) - f(a)} \end{aligned}}$$



Secant and False position method

- **Exercise 4: False position method in Excel**

it	x1	x2	f1	f2		x absc	f absc		interval si
0	-2	2	14	-2		1.5	-1.75		4
1	-2	1.5	14	-1.75		1.111111	-1.20988		0.388889
2	-2	1.111111	14	-1.20988		0.863636	-0.70868		0.247475
3	-2	0.863636	14	-0.70868		0.725664	-0.37607		0.137973
4	-2	0.725664	14	-0.37607		0.654362	-0.18926		0.071301
5	-2	0.654362	14	-0.18926		0.618958	-0.09272		0.035404
6	-2	0.618958	14	-0.09272		0.601727	-0.04483		0.017231
7	-2	0.601727	14	-0.04483		0.593422	-0.02154		0.008305
8	-2	0.593422	14	-0.02154		0.589438	-0.01032		0.003984
9	-2	0.589438	14	-0.01032		0.587532	-0.00493		0.001907
10	-2	0.58[illegible]532	14	-0.00493		0.5[illegible]662	-0.[illegible]0236		0.000911
11	-2	0.586[illegible]2	14	-0.00236		0.58[illegible]185	-0.[illegible]0113		0.000436
12	-2	0.58618[illegible]	14	-0.00113		0.58[illegible]977	-0.[illegible]0054		0.000208
[illegible]	[illegible]	0.5[illegible]	[illegible]	[illegible]00054		[illegible]	[illegible]		[illegible]
[illegible]	[illegible]	0.5[illegible]	[illegible]	[illegible]00026		[illegible]	[illegible]		[illegible]

=IF(f1*fabsc<0; x1;xabsc)

=IF(f2*fabsc<0; x2;xabsc)

x absc = x1 – f1*(x2 – x1)/(f2 – f1)

f absc = f(x absc)

Technische Universiteit Eindhoven University of Technology

Secant and False position method

- **Exercise 4: Secant method in Excel**

it	x1	x2	f1	f2		x absc	f absc		interval size
0	-2	2	14	-2		1.5	-1.75		4
1	-2	1.5	14	-1.75		1.111111	-1.20988		3.111111
2	1.111111	1.5	-1.20988	-1.75		0.24	1.0976		0.388889
3	0.24	1.111111	1.0976	-1.20988		0.654362	-0.18926		0.871111
4	0.24	0.654362	1.0976	-0.18926		0.593422	-0.02154		0.414362
5	0.593422	0.654362	-0.02154	-0.18926		0.585596	0.000538		0.060941
6	0.585596	0.593422	0.000538	-0.02154		0.585787	-1.5E-06		0.007826
7	0.585596	0.585787	0.000538	-1.5E-06		0.585786	-9.8E-11		0.000191
8	0.585786	0.585787	-9.8E-11	-1.5E-06		0.585786	0		5.15E-07
9	0.585786	0.585786	0	-9.8E-11		0.585786	0		3.46E-11

$=\min(x_{it-2},x_{it-1})$

$=\max(x_{it-2},x_{it-1})$

x absc = x1 – f1*(x2 – x1)/(f2 – f1)

f absc = f(x absc)

Secant and False position method

- **Exercise 4: False position method in MATLAB**

```
function [p] = falseposition(f, x1, x2, tol_step, tol_func)
  f1 = f(x1);
  f2 = f(x2);
  fp = f2;
  if (f1*f2>0)
    error('Root must be bracketed!');
  else
    it = 1;
    while ((abs(fp)>tol_func) && (abs(x2 - x1)>tol_step))
      it = it + 1;
      p = (x1*f2 - x2*f1)/(f2 - f1);
      fp = f(p);
      if (f1*fp<0)
        x2 = p;
        f2 = fp;
      else
        x1 = p;
        f1 = fp;
      end
    end
    disp(sprintf('Root found in %d iterations at x = %e\n (function value = %e)', [it,p,fp]));
  end
end
```

The only difference with bisection!

Root found in 12 iterations!
(Bisection needed 25 iterations)

```
>> falseposition(@(x) x^2-4*x+2,0,2,1e-7,1e-7);
```

Secant and False position method

- **Exercise 4: Secant method in MATLAB**

```
function [p] = secant(f, x1, x2, tol_step, tol_func)
  f1 = f(x1);
  f2 = f(x2);
  fp = f2;
  if (f1*f2>0)
    error('Root must be bracketed!');
  else
    it = 1;
    while ((abs(fp)>tol_func) && (abs(x2 - x1)>tol_step))
      it = it + 1;
      p = (x1*f2 - x2*f1)/(f2 - f1);
      fp = f(p);
      x1 = x2;
      f1 = f2;
      x2 = p;
      f2 = fp;
    end
    disp(sprintf('Root found in %d iterations at x = %e\n (function value = %e)', [it,p,fp]));
  end
end
```

The only difference with False position method!

```
>> secant(@(x) x^2-4*x+2,0,2,1e-7,1e-7);
```

Secant method: 8 iterations
False position: 12 iterations
Bisection: 25 iterations

Secant and False position method

- **Comparison of methods**

$$f(x) = x^2 - 4x + 2 = 0$$

tol_eps, tol_func = 1e-15, and $(x_1, x_2) = (0,2)$

Method	Nr. iterations
Bisection	52
False position	22
Secant	9

Compare with:

```
>> fzero(@(x) x^2-4*x+2,2,optimset('TolX',1e-15,'Display','iter'))
```

Note the initial bracketing steps in fzero!

Brent’s method

- **Superlinear convergence + sureness of bisection**
  - Keep track of superlinear convergence, and if not, intersperse with bisection steps (assures at least linear convergence)
  - Brent’s method (is implemented in MATLAB’s fzero): root-bracketing + bisection/secant/inverse quadratic interpolation
  - Inverse quadratic interpolation: uses 3 prior points to fit an inverse quadratic function (i.e. x(y)) with contingency plans, if root falls outside brackets:

    $$x = b + P/Q \qquad R = f(b)/f(c)$$
    $$P = S[T(R-T)(c-b) - (1-R)(b-a)] \qquad S = f(b)/f(a)$$
    $$Q = (T-1)(R-1)(S-1) \qquad T = f(a)/f(c)$$

    b = current best estimate
    P/*Q* = ought to be a ‘small’ correction
  - When *P*/*Q* does not land within the bounds or when bounds are not collapsing fast enough ⇒ take bisection step

Brent's method

```
function [root] = brent(f, x1, x2, tol)
  ITMAX = 100;
  EPS = 3e-8;
  a = x1; b = x2; c = x2;
  fa = f(a);
  fb = f(b);
  fc = fb;
  if (fa*fb>0)
    error('Root must be bracketed!');
  else
    for iter=1:ITMAX
      if (fb*fc>0)
        c = a;  fc = fa;      % Rename a, b, c and
        d = b - a;  e = d;    % adjust bounding interval d
      end;
      if (abs(fc)<abs(fb))
        a = b;  fa = fb;
        b = c;  fb = fc;
        c = a;  fc = fa;
      end;
      tol1 = 2.0*EPS*abs(b) + 0.5*tol; % Convergence check.
      xm = 0.5*(c - b);
      if ((abs(xm)<=tol1) || (fb == 0))
        root = b;
        disp(sprintf('\nRoot found in %d iterations at x = %e (f(x) = %e)', [iter,b,fb]));
        break;
      end;
      if ((abs(e)>=tol1) && (abs(fa)>abs(fb)))
        % Attempt inverse quadratic interpolation.
        s = fb/fa;
        if (a==c)
          p = 2.0*xm*s;
          q = 1.0 - s;
        else
          q = fa/fc;
          r = fb/fc;
          p = s*(2.0*xm*q*(q - r) - (b - a)*(r - 1.0));
          q = (q - 1.0)*(r - 1.0)*(s - 1.0);
        end;
```

Brent’s method

```
        if (p>0.0)
          q = -q; % Check whether in bounds.
        end;
        p = abs(p);
        min1 = 3.0*xm*q - abs(tol1*q);
        min2 = abs(e*q);
        if (2.0*p<min(min1,min2))
          e = d; % Accept interpolation.
          d = p/q;
        else
          d = xm; % Interpolation failed, use bisection.
          e = d;
        end;
      else
         d = xm; % Bounds decreasing too slowly, use bisection.
         e = d;
      end;
      a = b; % Move last best guess to a.
      fa = fb;
      if (abs(d)>tol1) % Evaluate new trial root.
        b = b + d;
      else
        if (xm<0)
          b = b - tol1;
        else
          b = b + tol1;
        end;
      end;
      fb = f(b);
      if (d == xm)
        disp(sprintf('Iteration: %d => x = %e, f(x) = %e (bisection)', [iter,b,fb]));
      else
        disp(sprintf('Iteration: %d => x = %e, f(x) = %e (inverse quadratic interpolation)', [iter,b,fb]));
      end;
    end;
    if (iter==ITMAX)
      disp('Maximum number of iterations exceeded in brent!');
    end;
  end;
end
```

Non-linear equation solving in Excel

- Excel comes with a goal-seek and solver function. Some prerequisites have to be installed. For Excel 2010:
  - Install via Excel →File → Options → Add-Ins → Go (at the bottom) → Select solver add-in. You can now call the solver screen on the 'data' menu ('Oplosser' in Dutch).

- The procedure to solve is then:
  - Select the goal-cell, and whether you want to minimize, maximize or set a certain value
  - Enter the variable cells; Excel is going to change the values in these cells to get to the desired solution
  - Specify the boundary conditions (e.g. to keep certain cells above zero)
  - Click 'solve' (possibly after setting the advanced options).

Excel: goal-seek example

- Goal-Seek can be used to set the goal-cell to a specified value (e.g. zero) by changing another cell:
  - Open Excel and type the following:

	A	B
1	x	3
2	f(x)	=-3*B1^2-5*B1+2
3		

  - Go to tab Data → What-if Analysis → Goal Seek
    - Set cell: B2
    - To Value: 0
    - By changing cell: B1
  - OK: You'll find a solution of 0.3333…

Excel: solver example

- The solver is used to change the value in a goal-cell, by changing the values in 1 or more other cells while keeping boundary conditions:
  - Use the following sheet:

	A	B	C
1		x	f(x)
2	x1	3	=2*B2*B3-B3+2
3	x2	4	=2*B3-4*B2-4

- Go to tab Data → Solver
  - Goalfunction: C2 (value of: 0)
  - Add boundary condition: C3 = 0
  - By changing cells: $B$2:$B$3 (you can just select the cells)
- Solve. You will find B2=0 and B3=2.

Technische Universiteit Eindhoven University of Technology

Non-linear equation solver in Matlab (1 var)

- **Use fzero for <u>single</u> variable non-linear zero finding**

```
>> fzero(@(x) -3*x^2-5*x+2,3)
```

Or with

```
function [F] = TestFuncFZero(x)
F = -3*x^2 - 5*x + 2;
end
>> fzero(@TestFuncFZero,3)
```

```
>> fzero(@(x) -3*x^2-5*x+2,3,optimset('Display','iter'))
```

Search for an interval around 3 containing a sign change:

Func-count	a	f(a)	b	f(b)
1	3	-40	3	-40
3	2.91515	-38.07	3.08485	-41.9732
5	2.88	-37.2832	3.12	-42.8032
7	2.83029	-36.1832	3.16971	-43.9896
[illegible]	[illegible]	[illegible]	3.24	-45.6928
[illegible]	[illegible]	[illegible]	3.33941	-48.1521

Note the initial bracketing steps in fzero!

Non-linear equation solver in Matlab (≥ 2 var)

- **Use fsolve for systems of non-linear equations with <u>multiple</u> variables**

```
function [F] = TestFuncFSolve(x)
F = [ 2*x(1)*x(2) - x(2) + 2
                2*x(2) - 4*x(1) - 4];
end
```

```
>> fsolve(@TestFuncFSolve,[2,2])
```

Newton-Raphson method

- **Requires the evaluation of the function *f*(*x*) and the derivative *f* ′(*x*) at arbitrary points**
  - Algorithm:
    - Extend tangent line at current point x_i till it crosses zero
    - Set next guess x_{i+1} to the abscissa of that zero crossing

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{1}{2}f''\delta^2 + \cdots$$ (Taylor series at *x*)

For small enough values of δ and for well-behaved functions, the non-linear terms become unimportant

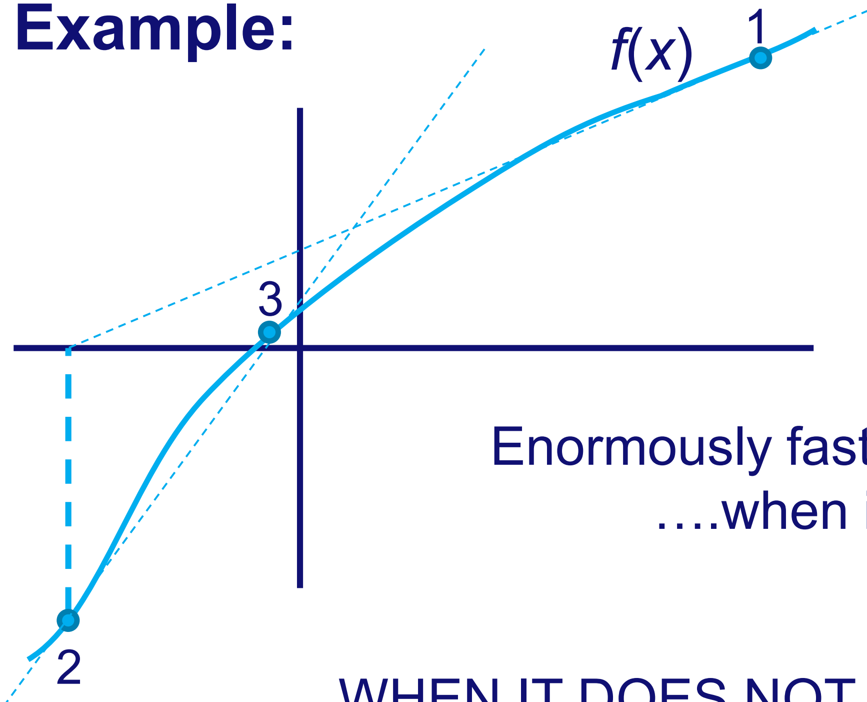
$$\Rightarrow \boxed{\delta = -\frac{f(x)}{f'(x)}}$$

- Can be extended to higher dimensions
- Requires an initial guess sufficiently close to the root! (otherwise even failure!!)

Technische Universiteit Eindhoven University of Technology

Newton-Raphson method

- **Example:**

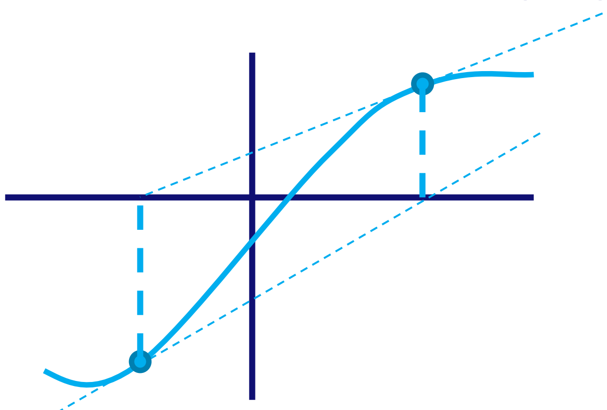


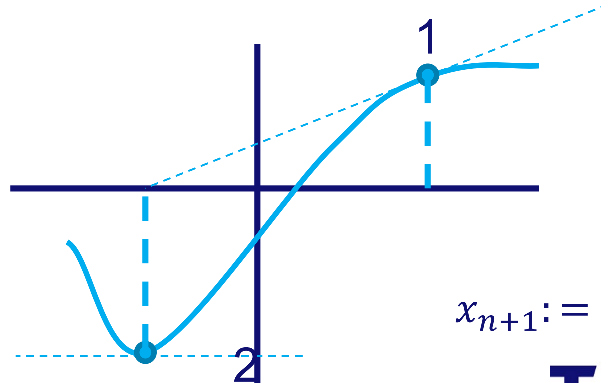
Newton-Raphson method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Enormously fast convergence,
....when it works

WHEN IT DOES NOT WORK...





Sometimes underrelaxation can help...

$$x_{n+1} := (1-\omega)x_n - \omega x_{n+1}$$

Newton-Raphson method

- **Basic algorithm:**

Given initial x, required tolerance $\varepsilon > 0$
Repeat
1. Compute f(x) and f'(x).
2. If $|f(x)| \leq \epsilon$, return x
3. $x := x - f(x)/f'(x)$

until maximum number of iterations is exceeded

Newton-Raphson method

- **Exercise 5: Newton-Raphson method in Excel**

it	x	f	df/dx	dx
0	0	2	-4	0.5
1	0.5	0.25	-3	0.083333
2	0.583333333333333	0.00694444	-2.83333	0.002451
3	0.585784313725490	6.0073E-06	-2.82843	2.12E-06
4	0.585786437625310	4.5108E-12	-2.82843	1.59E-12
5	0.585786437626905	0	-2.82843	0
analytical	0.585786437626905			

$$x_{n+1} = x_n + \delta x_n$$

$$\delta x_n = \frac{-f_n}{\frac{df}{dx_n}}$$

Newton-Raphson method

- **Why is Newton-Raphson so powerful?**
  $\Rightarrow$ High rate of convergence

Newton-Raphson method: $$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Subtracting the solution x^*: $$x_{n+1} - x^* = x_n - x^* - \frac{f(x_n)}{f'(x_n)}$$

Defining the error $\epsilon_n = x_n - x^*$: $$\epsilon_{n+1} = \epsilon_n - \frac{f(x_n)}{f'(x_n)}$$

$$\epsilon_{n+1} = \epsilon_n - \frac{f(x^*) + f'(x^*)\epsilon_n + \frac{1}{2}f''(x^*)\epsilon_n^2 + \cdots}{f'(x^*) + \cdots}$$

$$\epsilon_{n+1} = \epsilon_n - \epsilon_n - \frac{1}{2}\frac{f''(x^*)}{f'(x^*)}\epsilon_n^2 \quad \Rightarrow$$

$$\epsilon_{n+1} \sim K\epsilon_n^2$$

Quadratic convergence!!

Newton-Raphson method

- **Order of convergence**

$$\lim_{n\to\infty} \frac{|\epsilon_{n+1}|}{|\epsilon_n|^m} = K$$

m = order of convergence
K = asymptotic error constant

$\epsilon_n = x_n - x^*$ with x^* the solution

When the solution is not known a priori: $\epsilon_{n+1} \approx x_{n+1} - x_n$

$$\frac{|\epsilon_{n+1}|}{|\epsilon_n|} = \frac{K|\epsilon_n|^m}{K|\epsilon_{n-1}|^m} \Rightarrow \frac{|\epsilon_{n+1}|}{|\epsilon_n|} = \left(\frac{|\epsilon_n|}{|\epsilon_{n-1}|}\right)^m$$

$$\Rightarrow \ln\left(\frac{|\epsilon_{n+1}|}{|\epsilon_n|}\right) = m\, ln\left(\frac{|\epsilon_n|}{|\epsilon_{n-1}|}\right)$$

$$m = \frac{\ln\left(\frac{|\epsilon_{n+1}|}{|\epsilon_n|}\right)}{\ln\left(\frac{|\epsilon_n|}{|\epsilon_{n-1}|}\right)}$$

for $n \rightarrow \infty$

Newton-Raphson method

- **Exercise 5: Newton-Raphson method in Excel**

it	x	f	df/dx	dx		eps	m
0	0	2	-4	0.5		0.585786438	
1	0.5	0.25	-3	0.083333		0.085786438	
2	0.583333333333333	0.00694444	-2.83333	0.002451		0.002453104	1.850
3	0.585784313725490	6.0073E-06	-2.82843	2.12E-06		2.1239E-06	1.984
4	0.585786437625310	4.5108E-12	-2.82843	1.59E-12		1.59472E-12	2.000
5	0.585786437626905	0	-2.82843	0		1.11022E-16	
analytical	0.585786437626905						

$$= x^*$$

$$\epsilon_n = x_n - x^*$$

$$m = \frac{\ln\left(\frac{|\epsilon_{n+1}|}{|\epsilon_n|}\right)}{\ln\left(\frac{|\epsilon_n|}{|\epsilon_{n-1}|}\right)}$$

Newton-Raphson method

Exercise 6:

- **Write a function in MATLAB to find a root of a function using the Newton-Raphson method**
  - Assume that an initial guess x_0 is provided
  - Also the required tolerance is given
  - Output the results for every iteration
  - Verify that at every iteration the number of significant digits double, and compute the order of convergence

Exercise 6: Newton-Raphson in MATLAB

```matlab
function [p] = newton1D(func, grad, x, tol_x, tol_f)
  ITMAX = 100;
  error = 2*tol_f;
  it = 0;
  f = func(x);
  while (((error>tol_f) || (dx>tol_x)) && (it<ITMAX))
    it = it + 1;
    g = grad(x);
    dx = -f/g;
    x = x + dx;
    f = func(x);
    error = abs(f);
  end;
  if it<=ITMAX
    disp(sprintf('Root found in %d iterations at x = %e\n (function value = %e)', [it,x,f]));
  else
    disp(sprintf('No root found after %d iterations!', [it]));
  end;
end
```

```
>> newton1D(@(x) x^2-4*x+2, @(x) 2*x-4,1,1e-12,1e-12)
```

Convergence in 6 iterations.

Why does it not work with an initial guess of $x_0 = 2$??

Newton-Raphson method

- **Modifications to the basic algorithm**
  - If the first derivative $f'(x)$ is not known or cumbersome to compute/program, we can use the local num. approximation:

$$f'(x) \approx \frac{f(x+dx) - f(x)}{dx} \qquad (dx \sim 10^{-8})$$

dx should be small (otherwise the method reduces to first order)

But not too small (otherwise you will be wiped out by roundoff!)

  - Unless you know that the initial guess is close to the solution, the Newton-Raphson method should be combined with:
    - a bracketing method, to reject the solution if it wanders outside of the bounds;
    - Reduced Newton step method (= relaxation) for more robustness. Don’t take the entire step if the error does not decrease (enough)
    - More sophisticated step size control: Local line searches and backtracking using cubic interpolation (for global convergence)

Newton-Raphson method

Exercise 6: Newton-Raphson in MATLAB

```
function [p] = newton1Dnum(func, x, tol_x, tol_f)
  ITMAX = 100;
  h = 1e-8;
  error = 2*tol_f;
  it = 0;
  f = func(x);
  while (((error>tol_f) || (dx>tol_x)) && (it<ITMAX))
    it = it + 1;
    g = (func(x+h) - func(x))/h;
    dx = -f/g;
    x = x + dx;
    f = func(x);
    error = abs(f);
  end;
  if it<=ITMAX
    disp(sprintf('Root found in %d iterations at x = %e\n (function value = %e)', [it,x,f]));
  else
    disp(sprintf('No root found after %d iterations!', [it]));
  end;
end
```

Numerical differentiation

```
>> newton1Dnum(@(x) x^2-4*x+2,1,1e-12,1e-12)
```

Convergence also in 6 iterations!

Newton-Raphson method

- **How to solve:**

  $\boldsymbol{f}(\boldsymbol{x}) = \mathbf{0}$ for arbitrary functions $\boldsymbol{f}$

  "Root finding"

  (i.e. move all terms to the left)

  - **One dimensional case:** $f(x) = 0$

    "Bracket or 'trap' a root between bracketing values, then hunt it down like a rabbit."

  - **Multi-dimensional case:** $\boldsymbol{f}(\boldsymbol{x}) = \mathbf{0}$
    - *N* equations in *N* unknowns:
      You can only *hope* to find a solution.
      It may have no (real) solution, or more than one solution!
    - Much more difficult!!
      "You never know whether a root is near, unless you have found it"

Newton-Raphson method

- **Extensions to multi-dimensional case:**

Let's first consider the two-dimensional case:

$$f(x, y) = 0$$
$$g(x, y) = 0$$

Multi-variate Taylor series expansion:

$$f(x + \delta x, y + \delta y) \approx f(x, y) + \frac{\partial f}{\partial x}\delta x + \frac{\partial f}{\partial y}\delta y + O(\delta x^2, \delta y^2) = 0$$
$$g(x + \delta x, y + \delta y) \approx g(x, y) + \frac{\partial g}{\partial x}\delta x + \frac{\partial g}{\partial y}\delta y + O(\delta x^2, \delta y^2) = 0$$

Neglecting higher order terms:

$$\frac{\partial f}{\partial x}\delta x + \frac{\partial f}{\partial y}\delta y = -f(x, y)$$
$$\frac{\partial g}{\partial x}\delta x + \frac{\partial g}{\partial y}\delta y = -g(x, y)$$

$\Rightarrow$ Two linear equations in the two unknowns δx and δy.

Newton-Raphson method

- **Extensions to multi-dimensional case:**

Newton-Raphson method:

$$\frac{\partial f}{\partial x}\delta x + \frac{\partial f}{\partial y}\delta y = -f(x,y)$$
$$\frac{\partial g}{\partial x}\delta x + \frac{\partial g}{\partial y}\delta y = -g(x,y)$$

Or in matrix notation:

$$\begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} = -\begin{bmatrix} f(x,y) \\ g(x,y) \end{bmatrix}$$

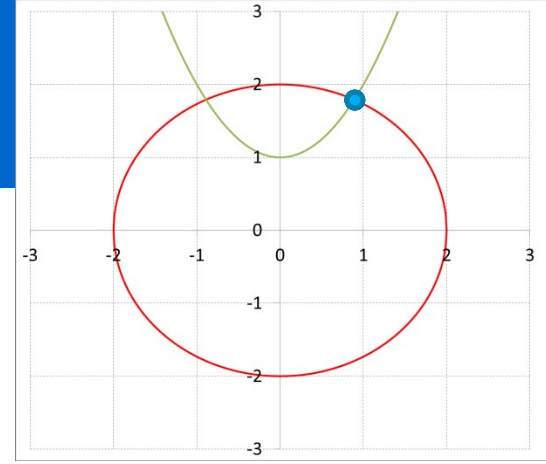
Jacobian matrix

Solution via Cramer's rule:

$$\delta x = \begin{vmatrix} -f & \frac{\partial f}{\partial y} \\ -g & \frac{\partial g}{\partial y} \end{vmatrix} \Bigg/ \begin{vmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{vmatrix} = \frac{-f\frac{\partial g}{\partial y}+g\frac{\partial f}{\partial y}}{\frac{\partial f}{\partial x}\frac{\partial g}{\partial y}-\frac{\partial f}{\partial y}\frac{\partial g}{\partial x}}$$

$$\delta y = \begin{vmatrix} \frac{\partial f}{\partial x} & -f \\ \frac{\partial g}{\partial x} & -g \end{vmatrix} \Bigg/ \begin{vmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{vmatrix} = \frac{-g\frac{\partial f}{\partial x}+f\frac{\partial g}{\partial x}}{\frac{\partial f}{\partial x}\frac{\partial g}{\partial y}-\frac{\partial f}{\partial y}\frac{\partial g}{\partial x}}$$

Newton-Raphson method



- **Extensions to multi-dimensional case:**

  Example: intersection of circle with parabola:

$$x^2 + y^2 = 4 \qquad y = x^2 + 1 = 0 \quad \Rightarrow$$

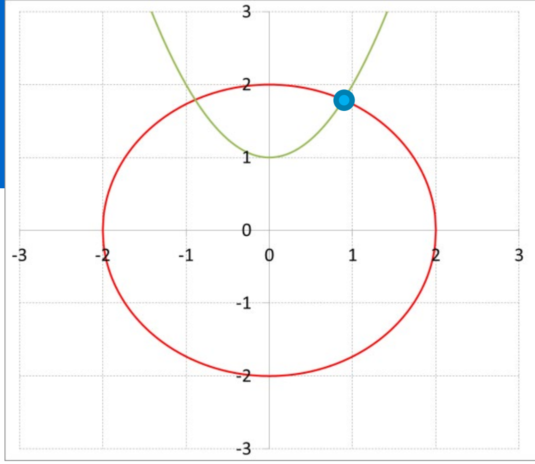
In matrix form:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \boldsymbol{f} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} x_1^2 + x_2^2 - 4 \\ x_1^2 - x_2 + 1 \end{bmatrix} \quad \mathbf{J} = \begin{bmatrix} 2x_1 & 2x_2 \\ 2x_1 & -1 \end{bmatrix}$$

	$\boldsymbol{x}^{(i)}$	$\boldsymbol{f}^{(i)}$	$\mathbf{J}^{(i)}$	$\delta\boldsymbol{x}^{(i)}$
i = 1:	$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 2 & 4 \\ 2 & -1 \end{bmatrix}$	$\begin{bmatrix} -0.1 \\ -0.2 \end{bmatrix}$
i = 2:	$\begin{bmatrix} 0.9 \\ 1.8 \end{bmatrix}$	$\begin{bmatrix} 0.05 \\ 0.01 \end{bmatrix}$	$\begin{bmatrix} 1.8 & 3.6 \\ 1.8 & -1 \end{bmatrix}$	$\begin{bmatrix} -0.01039 \\ -0.0087 \end{bmatrix}$
i = 3:	$\begin{bmatrix} 0.889614 \\ 1.791304 \end{bmatrix}$	$\begin{bmatrix} 0.000183 \\ 0.0000108 \end{bmatrix}$	$\begin{bmatrix} 1.7792 & 3.5826 \\ 1.7792 & -1 \end{bmatrix}$	$\begin{bmatrix} -6.99 \cdot 10^{-5} \\ -1.65 \cdot 10^{-5} \end{bmatrix}$
i = 4:	$\begin{bmatrix} 0.8895436 \\ 1.7912878 \end{bmatrix}$	$\begin{bmatrix} 5.16 \cdot 10^{-9} \\ 4.89 \cdot 10^{-9} \end{bmatrix}$	$\begin{bmatrix} 1.779087 & 3.582576 \\ 1.779087 & -1 \end{bmatrix}$	$\begin{bmatrix} -2.78 \cdot 10^{-9} \\ -5.94 \cdot 10^{-11} \end{bmatrix}$

Technische Universiteit Eindhoven University of Technology

Newton-Raphson method



- **Extensions to multi-dimensional case:**

  Example: intersection of circle with parabola:

  Check order of convergence:

it	x1	x2		eps1	eps2		m1	m2
1	1.0000000000000000	2.0000000000000000						
2	0.9000000000000000	1.8000000000000000		0.1000000000000000	0.2000000000000000			
3	0.8896135265700480	1.7913043478260900		0.0103864734299518	0.0086956521739132		1.983532	2.948192
4	0.8895436203043770	1.7912878475373300		0.0000699062656710	0.0000165002887549		2.094992	2.32082
5	0.8895436175241320	1.7912878474779200		0.0000000027802448	0.0000000000594120		2.058946	2.138235

Quadratic convergence!
= doubling number of significant digits every iteration

$$\epsilon_{n+1} \approx x_{n+1} - x_n$$

$$m = \frac{\ln\left(\frac{|\epsilon_{n+1}|}{|\epsilon_n|}\right)}{\ln\left(\frac{|\epsilon_n|}{|\epsilon_{n-1}|}\right)}$$

Newton-Raphson method

- **Extensions to multi-dimensional case:**

Generalization to the *N*-dimensional case:

$f_i(x_1, x_2, \dots, x_N) = 0$ for *i* = 1, 2, …, *N*

Define: $\boldsymbol{x} = [x_1, x_2, \dots, x_N]$ and $\boldsymbol{f} = [f_1, f_2, \dots, f_N]$ $\Rightarrow$ $\boxed{\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}}$

Multi-variate Taylor series expansion:

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

Define Jacobian matrix: $\boxed{J_{ij} = \frac{\partial f_i}{\partial x_j}}$ $\Rightarrow \boldsymbol{f}(\boldsymbol{x} + \delta\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}) + \mathbf{J} \cdot \delta\boldsymbol{x} + O(\delta\boldsymbol{x}^2)$

Neglect higher order terms:

$$\boxed{\begin{aligned} &\mathbf{J} \cdot \delta\boldsymbol{x} = -\boldsymbol{f}(\boldsymbol{x}) \\ &\boldsymbol{x}_{new} = \boldsymbol{x}_{old} + \delta\boldsymbol{x} \end{aligned}}$$

Multi-variate Newton-Raphson in MATLAB

```
function [f] = MyFunc(x)
  f(1) = x(1)^2 + x(2)^2 - 4;
  f(2) = x(1)^2 - x(2) + 1;
  f = f';
end
```

```
function [jac] = MyJac(x)
  jac(1,1) = 2*x(1);
  jac(1,2) = 2*x(2);
  jac(2,1) = 2*x(1);
  jac(2,2) = -1;
end
```

```
function [p] = newton(func, jac, x, tol_x, tol_f)
  ITMAX = 100;
  error = 2*tol_f;
  it = 0;
  f = feval(func,x);
  while (((error>tol_f) || (max(abs(dx))>tol_x)) && (it<ITMAX))
    it = it + 1;
    j = feval(jac,x);
    dx = j\(-f);
    x = x + dx;
    f = func(x);
    error = max(abs(f));
    disp(sprintf('iteration %d: x[1] = %e, x[2] = %e with f[1] = %e, f[2] = %e', [it,x(1),x(2),f(1),f(2)]));
  end;
  if it<=ITMAX
    disp(sprintf('\nRoot found in %d iterations at x[1] = %e, x[2] = %e with f[1] = %e; f[2] = %e\n', [it,x(1),x(2),f(1),f(2)]));
  else
    disp(sprintf('\nNo root found after %d iterations!\n', [it]));
  end;
end
```

```
>> newton(@MyFunc,@MyJac,[1;2],1e-12,1e-12)
```

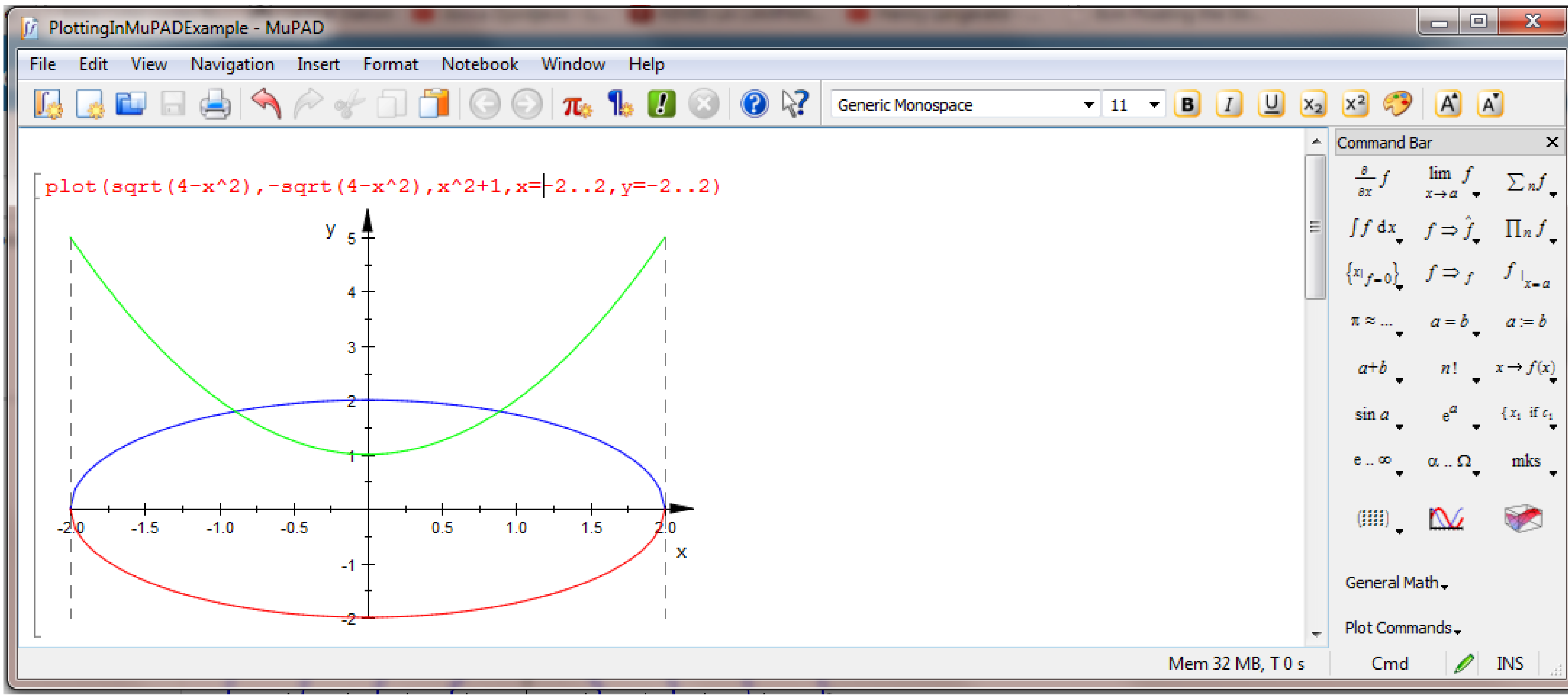
$\Rightarrow$ Only 5 iterations needed!

Newton-Raphson method

Multi-variate Newton-Raphson in MATLAB

Plotting the functions:

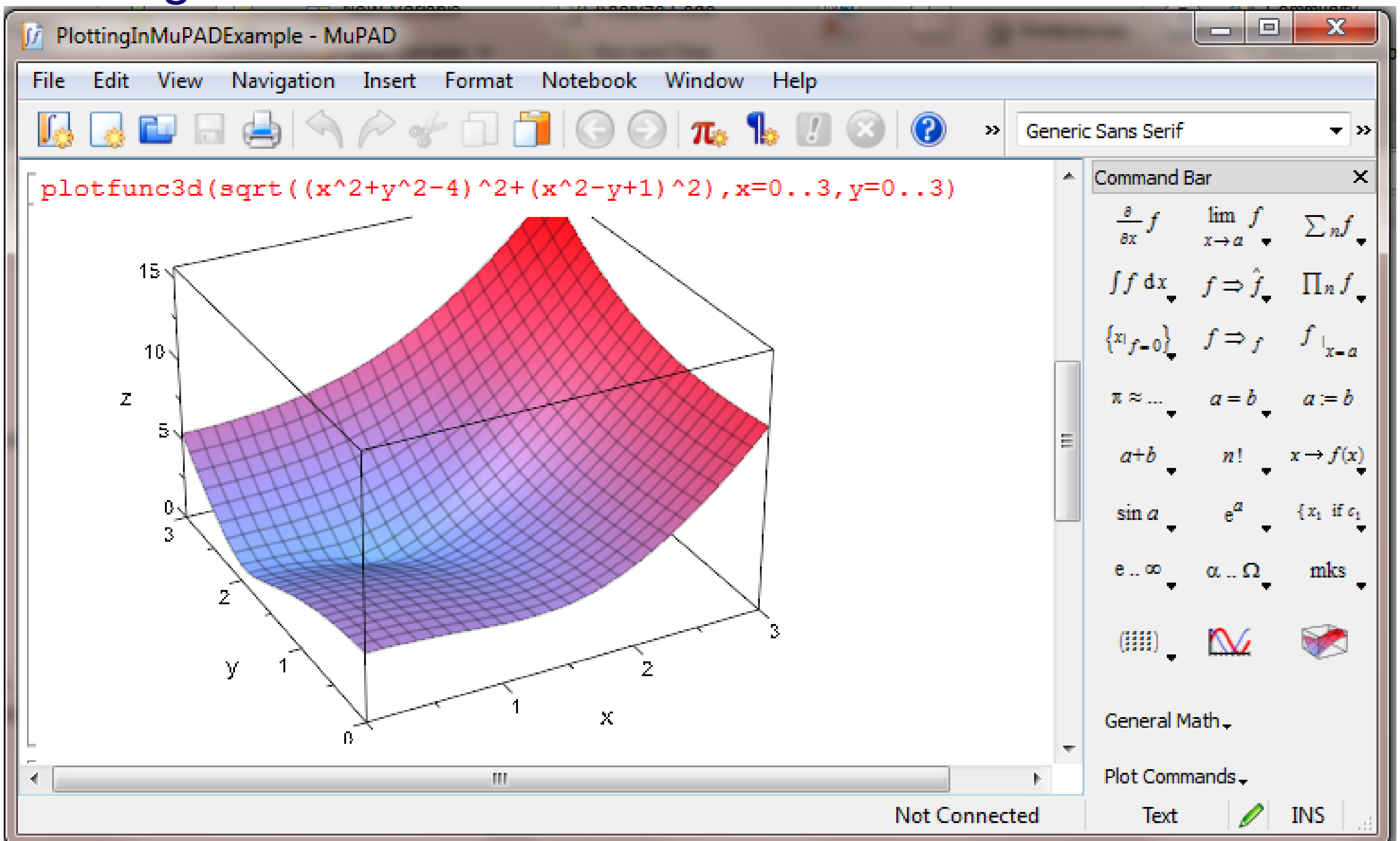
>> mphandle = mupad



Newton-Raphson method

Multi-variate Newton-Raphson in MATLAB

Plotting the norm of the function:

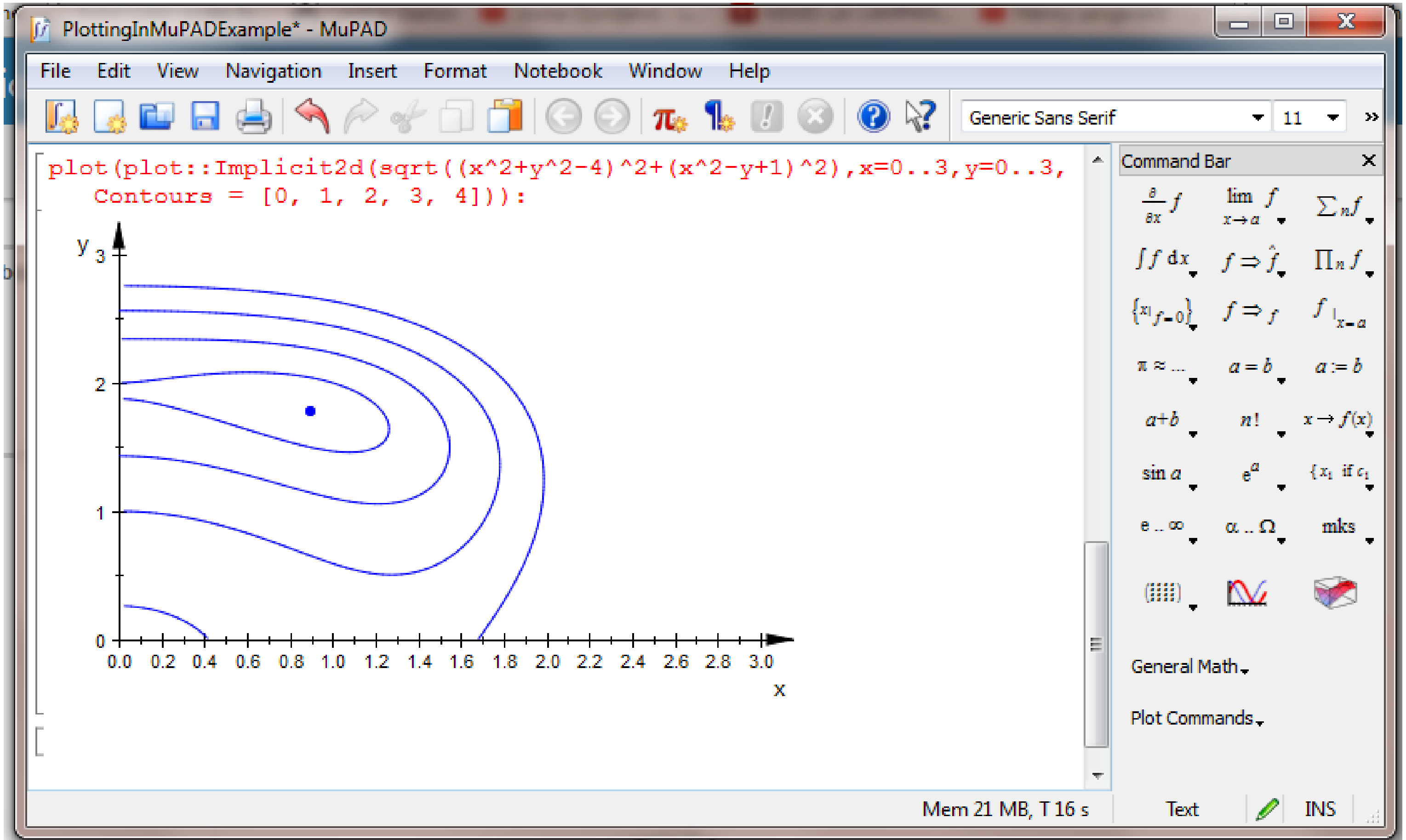


Technische Universiteit Eindhoven University of Technology

Newton-Raphson method

Multi-variate Newton-Raphson in MATLAB

Plotting contours of the norm of the function:



Broyden's method

- **Multi-dimensional secant method ('quasi-Newton'):**

Disadvantage of the Newton-Raphson method:
It requires the Jacobian matrix

- In many problems no analytical Jacobian available
- If the function evaluation is expensive, the numerical approximation using finite differences can be prohibitive!

$\Rightarrow$ $\boxed{\text{use cheap approximation of the Jacobian!}}$

(= secant, or 'quasi-Newton' method)

Newton-Raphson:	Secant method:	
$\mathbf{J}^n \cdot \delta \boldsymbol{x}^n = -\boldsymbol{f}^n(\boldsymbol{x}^n)$	$\mathbf{B}^n \cdot \delta \boldsymbol{x}^n = -\boldsymbol{f}^n(\boldsymbol{x}^n)$	$\mathbf{B}^n$= approximation of the Jacobian
$\boldsymbol{x}^{n+1} = \boldsymbol{x}^n + \delta \boldsymbol{x}^n$	$\boldsymbol{x}^{n+1} = \boldsymbol{x}^n + \delta \boldsymbol{x}^n$	

Broyden's method

- **Multi-dimensional secant method ('quasi-Newton'):**

  Secant equation (generalization of 1D case):

  $$\mathbf{B}^{n+1} \cdot \delta \boldsymbol{x}^n = \delta \boldsymbol{f}^n \qquad \delta \boldsymbol{x}^n = \boldsymbol{x}^{n+1} - \boldsymbol{x}^n \qquad \delta \boldsymbol{f}^n = \boldsymbol{f}^{n+1} - \boldsymbol{f}^n$$

  Underdetermined (i.e. not unique: n equations with n^2 unknowns)
  $\Rightarrow$ we need another condition to pin down $\mathbf{B}^{n+1}$

  Broyden's method: determine $\mathbf{B}^{n+1}$ by making the least change to $\mathbf{B}^n$ that is consistent with the secant condition

  Updating formula:
  $$\boxed{\mathbf{B}^{n+1} = \mathbf{B}^n + \frac{(\delta \boldsymbol{f}^n - \mathbf{B}^n \cdot \delta \boldsymbol{x}^n)}{\delta \boldsymbol{x}^n \cdot \delta \boldsymbol{x}^n} \otimes \delta \boldsymbol{x}^n}$$

  (Note: sometimes $\mathbf{B}^{-1}$ is updated directly)

Broyden's method

- **Multi-dimensional secant method ('quasi-Newton'):**

Background of Broyden's method:

Secant equation: $\mathbf{B}^{n+1} \cdot \delta \boldsymbol{x}^n = \delta \boldsymbol{f}^n$

Broyden's method: Since there is no update on derivative info, why would $\mathbf{B}^n$ change in a direction $\boldsymbol{w}$ orthogonal to $\delta \boldsymbol{x}^n$

$$\Rightarrow \quad (\delta \boldsymbol{x}^n)^{\mathrm{T}} \boldsymbol{w} = \mathbf{0}$$

$$\left.\begin{aligned} \mathbf{B}^{n+1} \cdot \boldsymbol{w} &= \mathbf{B}^n \cdot \boldsymbol{w} \\ \mathbf{B}^{n+1} \cdot \delta \boldsymbol{x}^n &= \delta \boldsymbol{f}^n \end{aligned}\right\} \Rightarrow \boxed{\mathbf{B}^{n+1} = \mathbf{B}^n + \frac{(\delta \boldsymbol{f}^n - \mathbf{B}^n \cdot \delta \boldsymbol{x}^n)}{\delta \boldsymbol{x}^n \cdot \delta \boldsymbol{x}^n} \otimes \delta \boldsymbol{x}^n}$$

Initialize $\mathbf{B}^0$ with identity matrix (or with finite difference approx.)

Broyden's method

- **Same example as before but now with Broyden's method**

```
function [p] = broyden(func, x, tol_x, tol_f)
  ITMAX = 100;
  error = 2*tol_f;
  it = 0;
  f = feval(func,x);
  b = eye(2); % create identity matrix
  while (((error>tol_f) || (max(abs(dx))>tol_x)) && (it<ITMAX))
    it = it + 1;
    dx = b\(-f);
    x = x + dx;
    f0 = f;
    f = func(x);
    df = f - f0;
    b = b + ((df - b*dx)*dx.')/(dx.'*dx); % Broyden's updating
    error = max(abs(f));
    disp(sprintf('iteration %d: x[1] = %e, x[2] = %e with f[1] = %e, f[2] = %e', [it,x(1),x(2),f(1),f(2)]));
  end;
  if it<=ITMAX
    disp(sprintf('\nRoot found in %d iterations at x[1] = %e, x[2] = %e with f[1] = %e; f[2] = %e\n', [it,x(1),x(2),f(1),f(2)]));
  else
    disp(sprintf('\nNo root found after %d iterations!\n', [it]));
  end;
end
```

Slower convergence with Broyden's method should be offset by improved efficiency of each iteration!

>> broyden(@MyFunc,[1;2],1e-12,1e-12)

Requires 11 iterations (compare with Newton: 5 iterations)

But much fewer function evaluations per iteration!

Broyden's method

- **Same example as before but now with Broyden's method**

  Note how the approximate Jacobian (**B**) is updated over subsequent iterations:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 5 \\ 1 & -1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1.0225 & 4.9685 \\ 0.9437 & -0.9212 \end{bmatrix} \Rightarrow \begin{bmatrix} 2.0881 & 4.6442 \\ 1.7312 & -1.1608 \end{bmatrix} \Rightarrow$$

$$\Rightarrow \begin{bmatrix} 1.7932 & 4.6321 \\ 1.8420 & -1.1563 \end{bmatrix} \Rightarrow \begin{bmatrix} 2.0222 & 3.6454 \\ 1.8043 & -0.9940 \end{bmatrix} \Rightarrow \begin{bmatrix} 2.0032 & 3.5423 \\ 1.8024 & -1.0043 \end{bmatrix} \Rightarrow \begin{bmatrix} 1.9295 & 3.4553 \\ 1.7948 & -1.0133 \end{bmatrix}$$

$$\Rightarrow \quad \cdots \quad \Rightarrow \begin{bmatrix} 1.9284 & 3.4539 \\ 1.7945 & -1.0136 \end{bmatrix}$$

Compare with exact Jacobian: $\begin{bmatrix} 1.779087 & 3.582576 \\ 1.779087 & -1 \end{bmatrix}$

$\Rightarrow$ Note that the approximate Jacobian (**B**) is not exact even when the solution has already been found!

Technische Universiteit Eindhoven University of Technology

Conclusions

- **Recommendations for root finding:**
  - **One-dimensional cases:**
    - If it is not easy/cheap to compute the function's derivative
      $\Rightarrow$ use Brent's algorithm
    - If derivative information is available
      $\Rightarrow$ use Newton-Raphson's method + bookkeeping on bounds
      provided you can supply a good enough initial guess!!
    - There are specialized routines for (multiple) root finding of polynomials (but not covered in this course)
  - **Multi-dimensional cases:**
    - Use Newton-Raphson method, but make sure that you provide an initial guess close enough to achieve convergence
    - In case derivative information is expensive
      $\Rightarrow$ use Broyden's method (but slower convergence!)