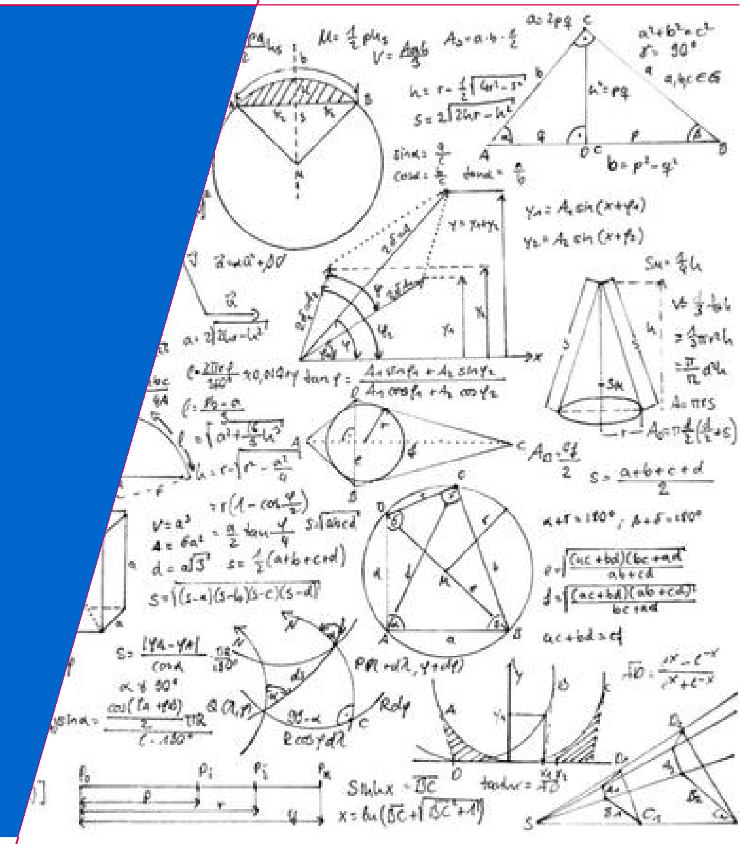
Numerical methods for Chemical Engineers:

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:**

  $\boldsymbol{f}(\boldsymbol{x}) = \mathbf{0}$ 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:** $\boldsymbol{f}(\boldsymbol{x}) = \mathbf{0}$
    - *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"

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…

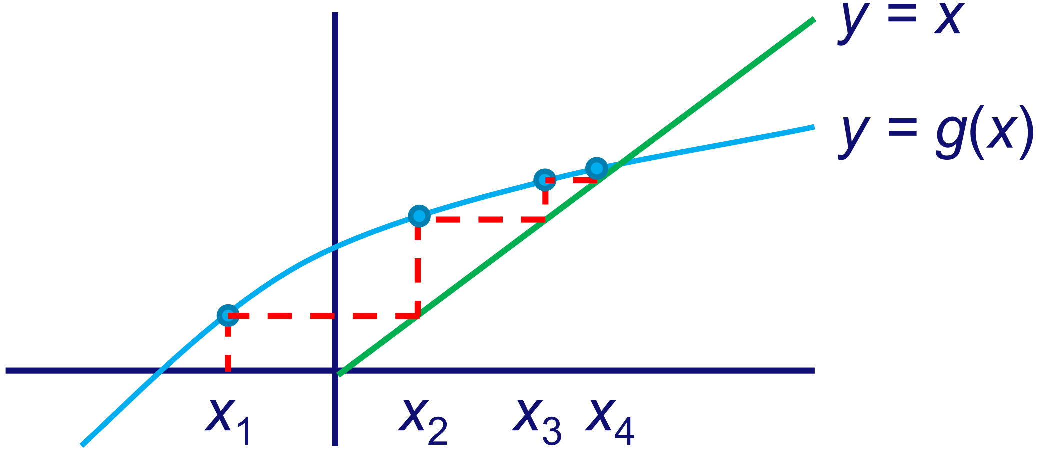
Hamming’s motto:
the purpose of computing is insight, not numbers!!

➢ 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)
```


Technische Universiteit Eindhoven University of Technology

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])
```

Direct iteration method

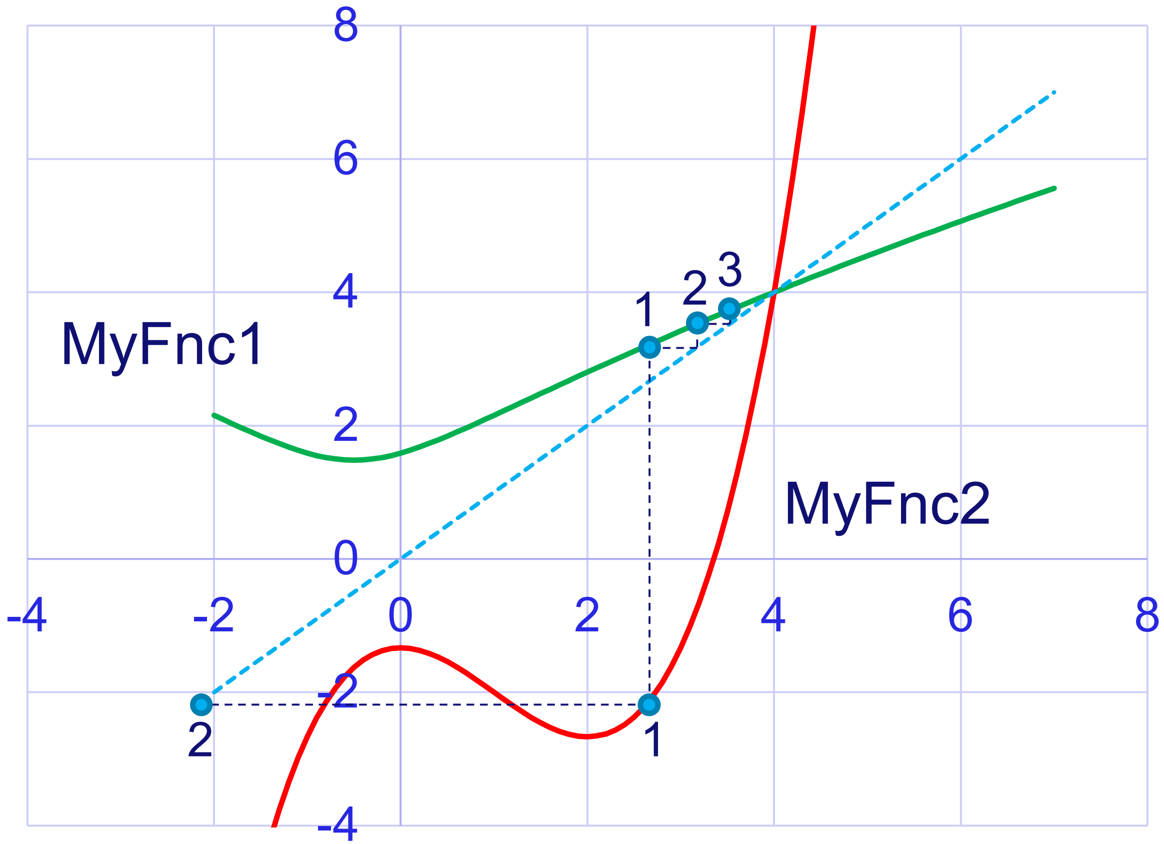
Exercise 1: Find the root of $x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method

- **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 $f(x) = x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method



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

And even then not very fast ...

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

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

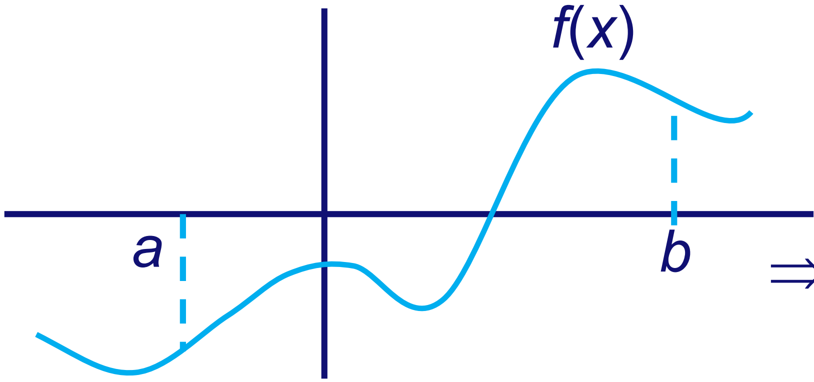
$$x_{i+2} = x_{i+1} + g'(x_i)(x_{i+1} - x_i)$$

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

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

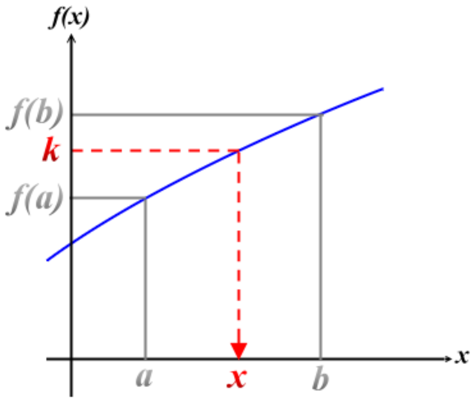
Bracketing

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

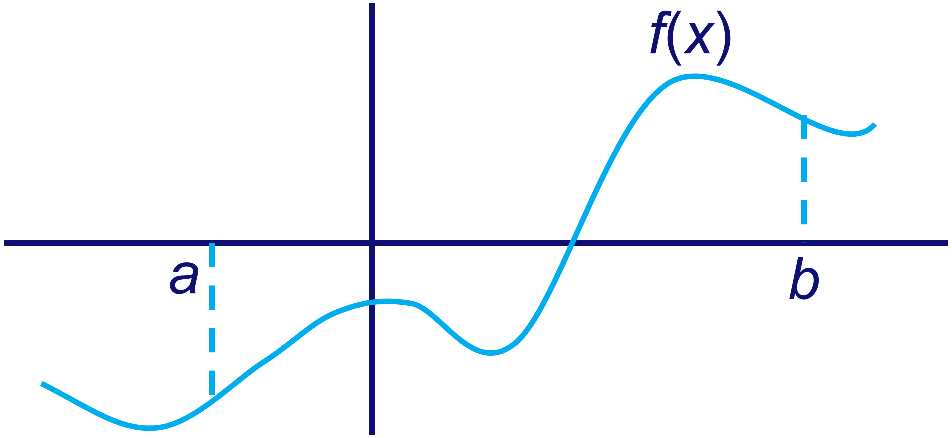


Intermediate Value Theorem

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

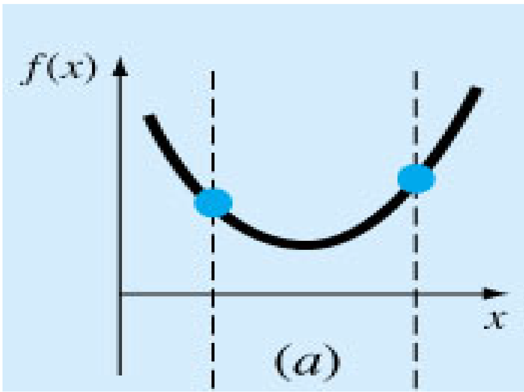
Bracketing a root = knowing that the function changes sign in an identified interval



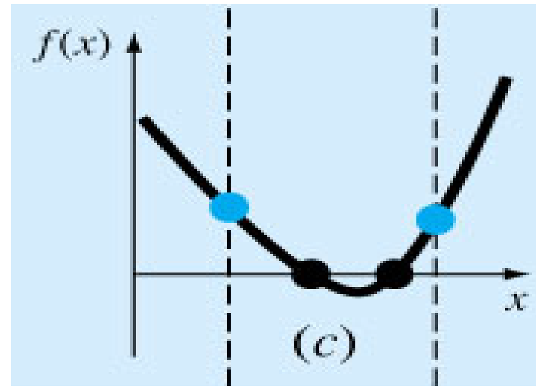
- **General best advise:**
  - Always bracket a root before trying to converge…
  - Never allow your iteration method to get outside the best bracketing bounds…

General idea

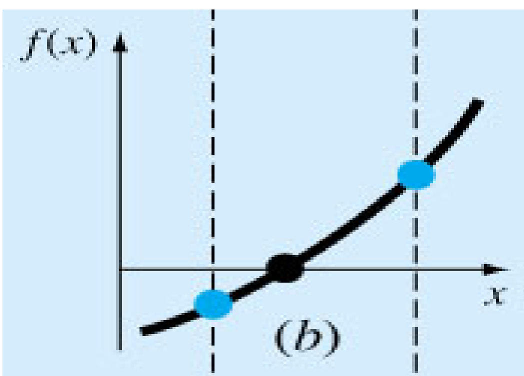
- **Examples of pitfalls of bracketing…**



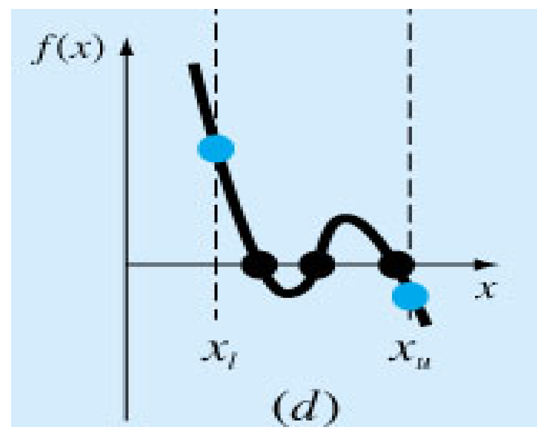
No answer (no root)



Oops!! (two roots!!)



Nice case (one root)



Three roots (might work for a while!)

Technische Universiteit Eindhoven University of Technology

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.

Technische Universiteit Eindhoven University of Technology

Bracketing

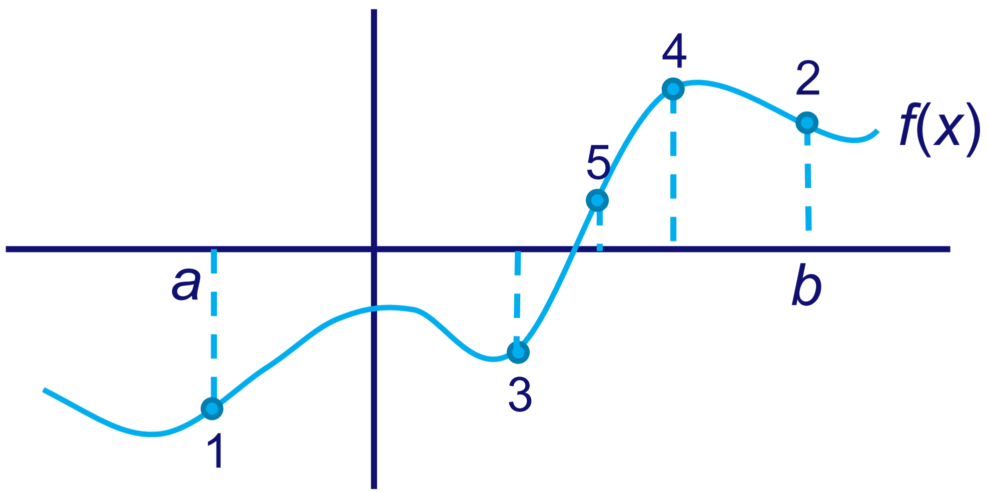
Exercise 2:

- **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.

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

- **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

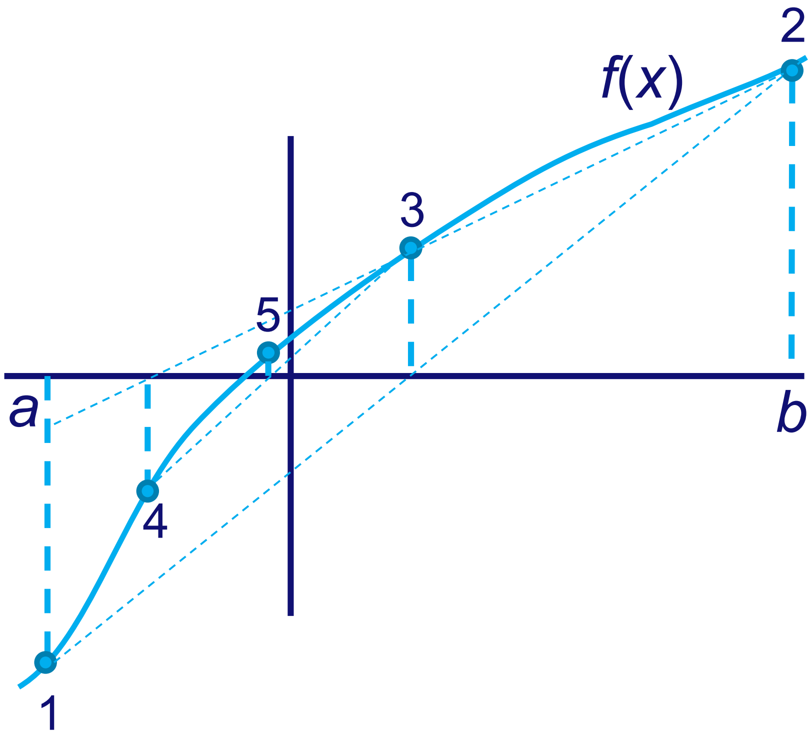
  - Must succeed:
    - More than one root ⇒ bisection will find one of them
    - No root, but singularity ⇒ bisection will find singularity

Secant and False position method

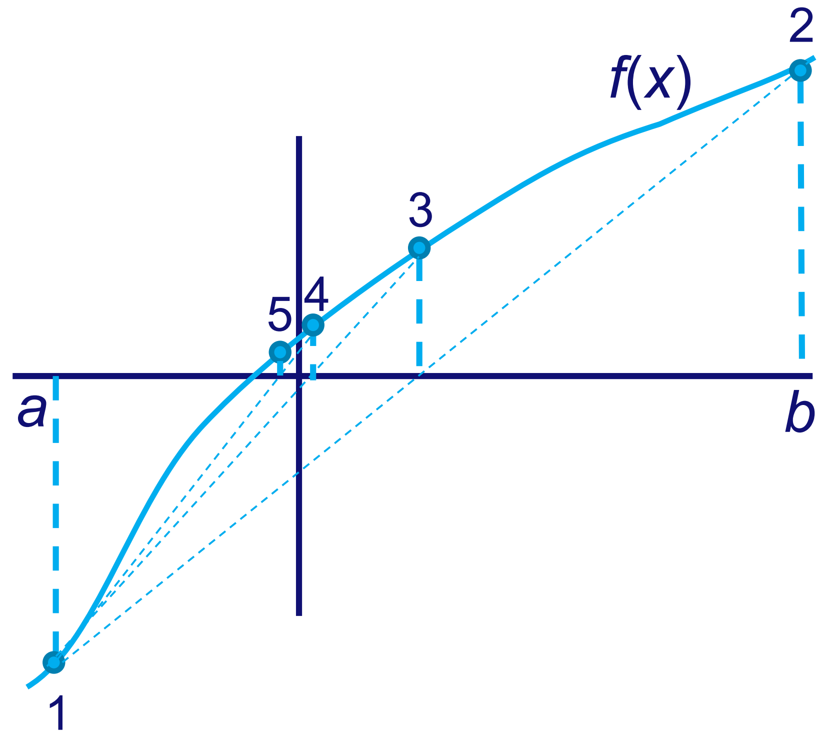
- **Secant/False position (= Regula Falsi) method**
  - Faster convergence (provided sufficiently smooth behaviour)
  - Difference with bisection method in choice of next point:
    - Bisection: mid-point of interval
    - Secant/False position: point where the approximating line crosses the axis
  - 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

Technische Universiteit Eindhoven University of Technology

Secant and False position method

Exercise 4:

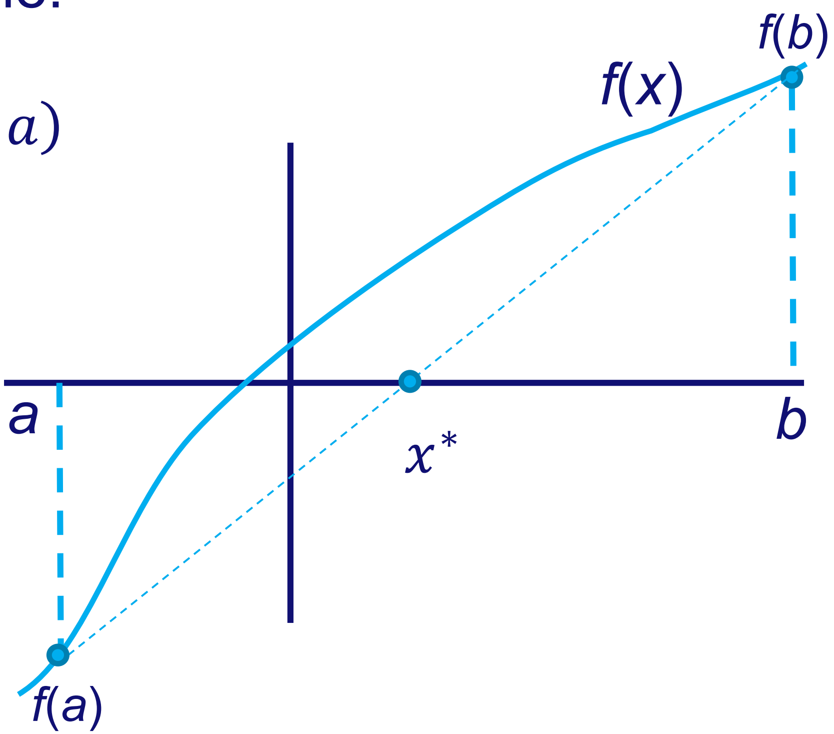
- **Determination of the abscissa of the approximating line:**
  - Determine the approximating line:

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

  - 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:

- **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

- **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 $\Rightarrow$ take bisection step

Technische Universiteit Eindhoven University of Technology

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…

TU/e Technische Universiteit Eindhoven University of Technology

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
			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

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

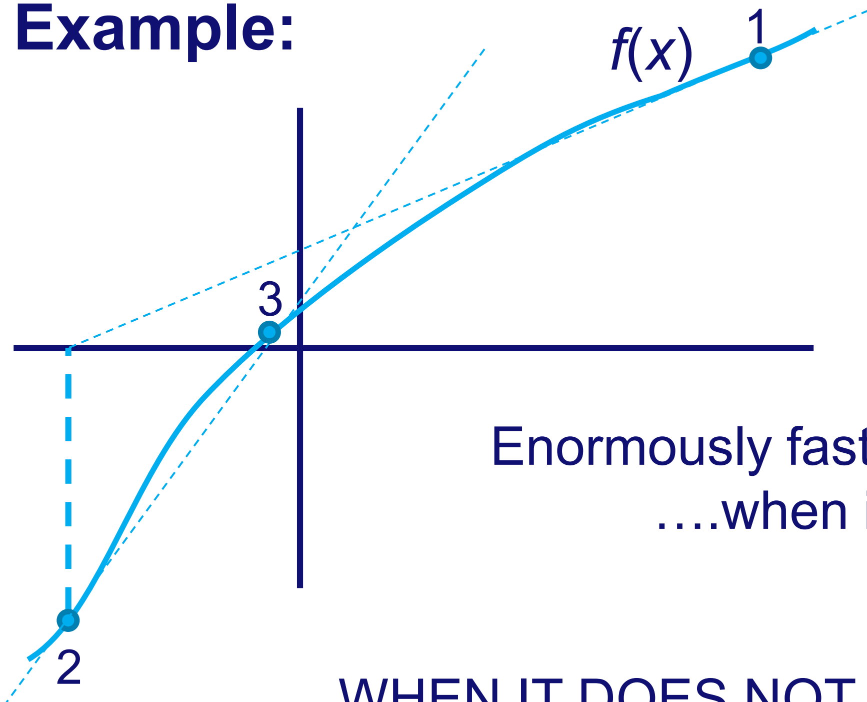
For small enough values of δ and for well-behaved functions, the non-linear terms become unimportant

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

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

Newton-Raphson method

- **Example:**

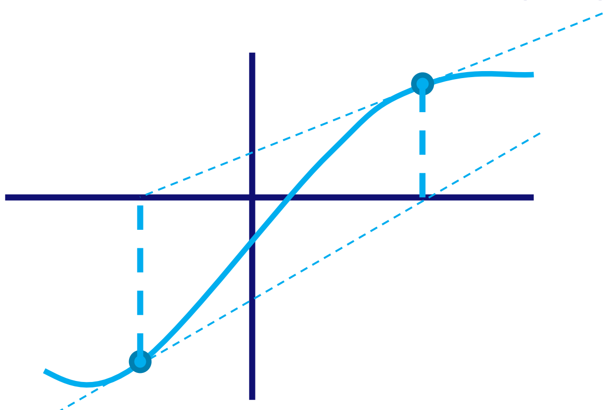


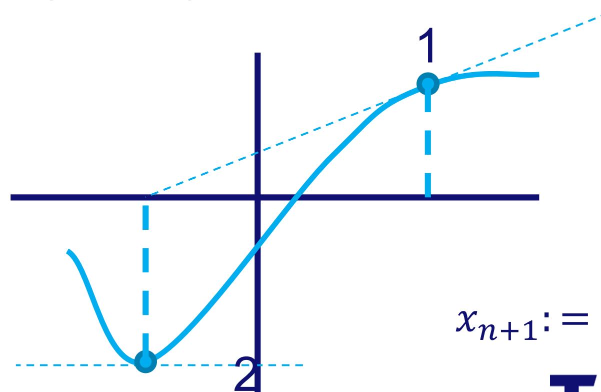
Newton-Raphson method:

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

Enormously fast convergence,
….when it works

WHEN IT DOES NOT WORK…





Sometimes underrelaxation can help...

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

Technische Universiteit Eindhoven University of Technology

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

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

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

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

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

Quadratic convergence!!

Technische Universiteit Eindhoven University of Technology

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 $n \to \infty$

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 doubles, and compute the order of convergence

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

- **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:

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

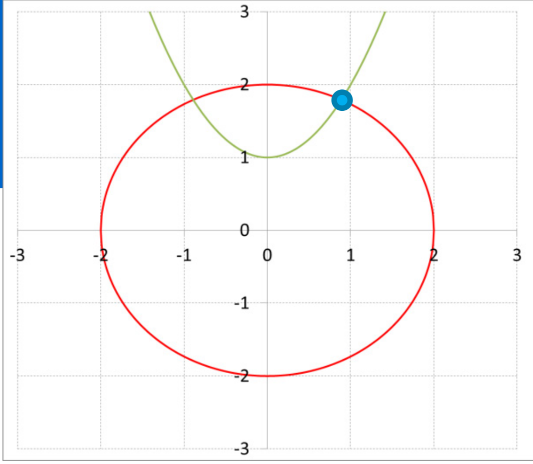
Jacobian matrix

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:

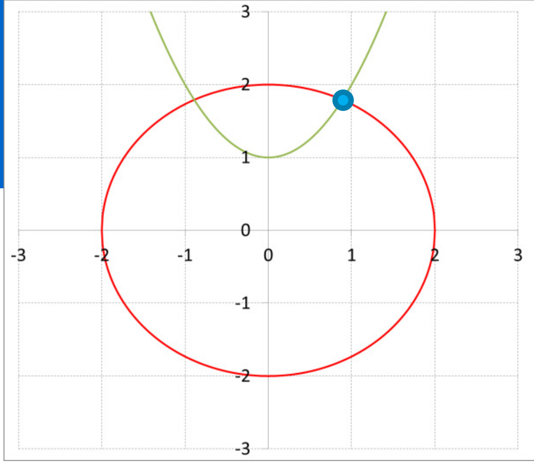
$$\begin{aligned} x^2 + y^2 &= 4 \\ y = x^2 + 1 &= 0 \end{aligned} \quad \Rightarrow$$

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:	$\begin{bmatrix} 0.8895436 \\ 1.7912878 \end{bmatrix}$	$\begin{bmatrix} 5.16 \cdot 10^{-9} \\ 4.89 \cdot 10^{-9} \end{bmatrix}$	$\begin{bmatrix} 1.779087 & 3.582576 \\ 1.779087 & -1 \end{bmatrix}$	$\begin{bmatrix} -2.78 \cdot 10^{-9} \\ -5.94 \cdot 10^{-11} \end{bmatrix}$

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

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

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

Technische Universiteit Eindhoven University of Technology

Newton-Raphson method

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

Generalization to the *N*-dimensional case:

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

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

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:

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

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)

  $$\left(\boldsymbol{a} \otimes \boldsymbol{b} = \boldsymbol{a}\boldsymbol{b}^{\mathrm{T}}\right)$$

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.)

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!)