# Non-linear equations

## One dimensional case

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

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2023-2024

S

# Today's outline

- Introduction
  - General
- Direct Iteration Method
  - Passing functions
- Bracketing
- Bisection method
- Secant/False Position
- Brent's method

EINDHOVEN UNIVERSITY OF TECHNOLOGY

# Today's outline

- Introduction
  - General
- Direct Iteration Method
  - Passing functions
- Bracketing
- Bisection method
- Secant/False Position
- Brent's method

# Content

## Root finding

How to solve $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ for arbitrary functions $\mathbf{f}$ (i.e., $\mathbf{f}(\mathbf{x})$ move all terms to the left)

- One-dimensional case: 'Bracket' or 'trap' a root between bracketing values, then hunt it down like a rabbit.
- Multi-dimensional case:
  - $N$ 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

**Multi-dimensional case:**

- Newton-Raphson method
- Broyden's method

# Outline

**One-dimensional case:**

- Direct iteration method
- Bisection method
- Secant and false position method
- Brent's method
- Newton-Raphson method

**Multi-dimensional case:**

- Newton-Raphson method
- Broyden's method

**In this course we will:**

- Introduction to underlying ideas and algorithms
- Exercises in how to program the methods in Excel and Python.

# Outline

**One-dimensional case:**

- Direct iteration method
- Bisection method
- Secant and false position method
- Brent's method
- Newton-Raphson method

**Multi-dimensional case:**

- Newton-Raphson method
- Broyden's method

**In this course we will:**

- Introduction to underlying ideas and algorithms
- Exercises in how to program the methods in Excel and Python.

**Warning**

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

# Today's outline

- Introduction
  - General
- Direct Iteration Method
  - Passing functions
- Bracketing
- Bisection method
- Secant/False Position
- Brent's method

# 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...

# 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. . .

Tips:

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

# 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...

Tips:

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

**Hamming's motto**

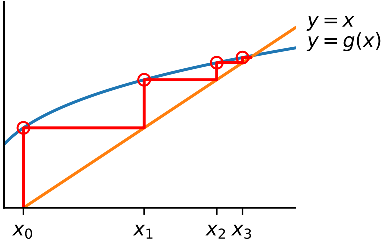
The purpose of computing is insight, not numbers!!

# Direct Iteration Method/Successive Substitutions

Rewrite $f(x) = 0 \Rightarrow x = g(x)$

- Start with an initial guess: $x_0$
- Calculate new estimate with: $x_1 = g(x_0)$
- Continue iteration with: $x_2 = g(x_1)$
- Proceed until: $|x_{i+1} - x_i| < \varepsilon$

When the process converges, taking a smaller value for $x_{i+1} - x_i$ results in a more accurate solution, but more iterations need to be performed.



# Direct Iteration Method - Exercise 1

Find the root of

$$f(x) = x^3 - 3x^2 - 3x - 4$$

# Direct Iteration Method - Exercise 1

Find the root of

$$f(x) = x^3 - 3x^2 - 3x - 4$$

**Attempt 1**

Rewrite as $x = (3x^2 + 3x + 4)^{(1/3)}$

- Solve in Excel
- Solve in Python

# Direct Iteration Method - Exercise 1

Find the root of

$$f(x) = x^3 - 3x^2 - 3x - 4$$

## Attempt 1

Rewrite as $x = (3x^2 + 3x + 4)^{(1/3)}$

- Solve in Excel
- Solve in Python

## Attempt 2

Rewrite as: $x = (x^3 - 3x^2 - 4)/3$

- Solve in Excel
- Solve in Python

# Intermezzo: Functions Revisited

- In Python, you can define your own functions to reuse certain functionalities. We can define a mathematical function at the top of a file, or in a separate file with `.py` extension:

```
def demo_f1(x):
    return x**2 + np.exp(x)
```

- The first line contains the function name, in this case `demo_f1`
- The return statement defines the output, `x` is defined as input
- It can use `x` as a scalar as well as a vector by using NumPy: e.g. `np.exp()`
  - If `x` is a vector, the output is also a vector.
- In case you define your function in a separate file, e.g. `nonlin_functions.py`, you can import the function into another file through:

```
from nonlin_functions import demo_f1
```

# Passing Functions in Python

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

```python
def MyFunc(x): # Note: case sensitive!!
    return x**2 - 4*x + 2
```

- The function can be assigned to a variable as an alias:

```python
f = MyFunc
a = 4
b = f(a)
```

```
2
```

- We can then call a solving routine (e.g., `fsolve` from SciPy):

```python
from scipy.optimize import fsolve
ans = fsolve(MyFunc, 5)
ans = fsolve(lambda x: x**2 - 4*x + 2, 5)
```

```
array([3.41421356])
array([3.41421356])
```

# Passing Functions in Python

- We can also make our own function, that takes another function as an argument:

```python
import matplotlib.pyplot as plt
import numpy as np

def draw_my_function(func):
    # Draws a function in the range [0, 10] using 20 data points.
    # 'func' is a function that can be any actual function.
    x = np.linspace(0, 10, 20)
    y = func(x)
    plt.plot(x, y, "-o")
    plt.show()
```

- Now we can call the function with another function, either a lambda function or a common function:

```python
f = lambda x: x**2 - 4*x + 2
draw_my_function(f)
```

# Direct Iteration Method - Exercise 1

Find the root of

$$f(x) = x^3 - 3x^2 - 3x - 4$$

**Attempt 1**

Rewrite as $x = (3x^2 + 3x + 4)^{(1/3)}$

- Solve in Excel
- Solve in Python

**Attempt 2**

Rewrite as: $x = (x^3 - 3x^2 - 4)/3$

- Solve in Excel
- Solve in Python

# Direct Iteration Method - Exercise 1

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

First attempt:

| Iteration | Formula | Result |
|---|---|---|
| 1 | $(3x^2 + 3x + 4)^{(1/3)}$ | 2 |
| 2 | | 3.115 |
| 3 | | 3.489 |
| $\vdots$ | | $\vdots$ |
| 10 | | 3.990 |

**Converges!**

Second attempt:

| Iteration | Formula | Result |
|---|---|---|
| 1 | $x = (x^3 - 3x^2 - 4)/3$ | $-1$ |
| 2 | | $-2.375$ |
| 3 | | $-11.439$ |
| $\vdots$ | | $\vdots$ |
| 10 | | #NUM! |

**Diverges!**

# Direct Iteration Method - Exercise 1

Find the root of $f(x) = x^3 - 3x^2 - 3x - 4 = 0$ with the direct iteration method in Python:
A simple script:

```python
x = 2.5
print(f"i: {0}, x: {x:.6e}")
for i in range(1, 21):
    x = (3*x**2 + 3*x + 4)**(1/3)
    print(f"i: {i}, x: {x:.6e}")
```

```
i: 0, x: 2.500000e+00
i: 1, x: 3.115840e+00
i: 2, x: 3.489024e+00
...
i: 19, x: 3.999970e+00
i: 20, x: 3.999983e+00
```

**Lesson**

Not very flexible/reusable → use functions

# Direct Iteration Method - Exercise 1

Find the root of the equation $f(x) = x^3 - 3x^2 - 3x - 4 = 0$ using the direct iteration method in Python.

- First, define the functions.

```python
def MyFnc1(x):
    return (3*x**2 + 3*x + 4)**(1/3)

def MyFnc2(x):
    return (x**3 - 3*x**2 - 4) / 3
```

# Direct Iteration Method - Exercise 1

Find the root of the equation $f(x) = x^3 - 3x^2 - 3x - 4 = 0$ using the direct iteration method in Python.

- First, define the functions.

```python
def MyFnc1(x):
    return (3*x**2 + 3*x + 4)**(1/3)

def MyFnc2(x):
    return (x**3 - 3*x**2 - 4) / 3
```

- Then, create a function to carry out the Direct Iteration algorithm.

```python
def DirectIterationMethod(g, x, eps):
    itmax = 100
    it = 0
    y = g(x)
    print(f"i: {0}, x: {x:.6e}")
    while (abs(y - x) > eps) and (it < itmax):
        it += 1
        x = y
        y = g(x)
        print(f"i: {it}, x: {x:.6e}")
```

# Direct Iteration Method - Exercise 1

Find the root of the equation $f(x) = x^3 - 3x^2 - 3x - 4 = 0$ using the direct iteration method in Python.

- Finally, call the Direct Iteration function with the appropriate parameters.

```
DirectIterationMethod(MyFnc1, 2.5, 1e-3)
DirectIterationMethod(MyFnc2, 2.5, 1e-3)
```

# Direct Iteration Method - Exercise 1

Find the root of the equation $f(x) = x^3 - 3x^2 - 3x - 4 = 0$ using the direct iteration method in Python.

- Finally, call the Direct Iteration function with the appropriate parameters.

```
DirectIterationMethod(MyFnc1, 2.5, 1e-3)
DirectIterationMethod(MyFnc2, 2.5, 1e-3)
```

```
i: 0, x: 2.500000e+00
i: 1, x: 3.115840e+00
i: 2, x: 3.489024e+00
i: 3, x: 3.708113e+00
...
i: 9, x: 3.990573e+00
i: 10, x: 3.994696e+00
i: 11, x: 3.997016e+00
i: 12, x: 3.998321e+00
```

# Direct Iteration Method - Exercise 1

Find the root of the equation $f(x) = x^3 - 3x^2 - 3x - 4 = 0$ using the direct iteration method in Python.

- Finally, call the Direct Iteration function with the appropriate parameters.

```python
DirectIterationMethod(MyFnc1, 2.5, 1e-3)
DirectIterationMethod(MyFnc2, 2.5, 1e-3)
```

```
i: 0, x: 2.500000e+00
i: 1, x: 3.115840e+00
i: 2, x: 3.489024e+00
i: 3, x: 3.708113e+00
...
i: 9, x: 3.990573e+00
i: 10, x: 3.994696e+00
i: 11, x: 3.997016e+00
i: 12, x: 3.998321e+00
```

```
i: 0, x: 2.500000e+00
i: 1, x: -2.375000e+00
i: 2, x: -1.143945e+01
i: 3, x: -6.311875e+02
i: 4, x: -8.421961e+07
i: 5, x: -1.991216e+23
i: 6, x: -2.631687e+69
Traceback (most recent
    call last):
```

# Direct Iteration Method - Exercise 1

Find the root of the equation $f(x) = x^3 - 3x^2 - 3x - 4 = 0$ using the direct iteration method in Python.

- Finally, call the Direct Iteration function with the appropriate parameters.

```python
DirectIterationMethod(MyFnc1, 2.5, 1e-3)
DirectIterationMethod(MyFnc2, 2.5, 1e-3)
```

```
i: 0, x: 2.500000e+00
i: 1, x: 3.115840e+00
i: 2, x: 3.489024e+00
i: 3, x: 3.708113e+00
...
i: 9, x: 3.990573e+00
i: 10, x: 3.994696e+00
i: 11, x: 3.997016e+00
i: 12, x: 3.998321e+00
```

```
i: 0, x: 2.500000e+00
i: 1, x: -2.375000e+00
i: 2, x: -1.143945e+01
i: 3, x: -6.311875e+02
i: 4, x: -8.421961e+07
i: 5, x: -1.991216e+23
i: 6, x: -2.631687e+69
Traceback (most recent
    call last):
```

**Thinking**

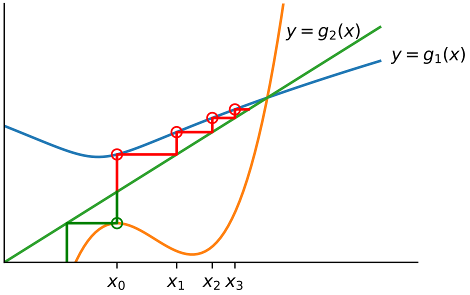
Discuss why it converges with `MyFnc1` and diverges with `MyFnc2`

# Direct Iteration Method

- Exercise 1: Find the root of the equation

$$f(x) = x^3 - 3x^2 - 3x - 4 = 0$$

using the direct iteration method.
- Observe that the method only works effectively when $g'(x_i) < 1$. Even then, it may not converge quickly.




**Point**

The iterations can be represented using the following relations:

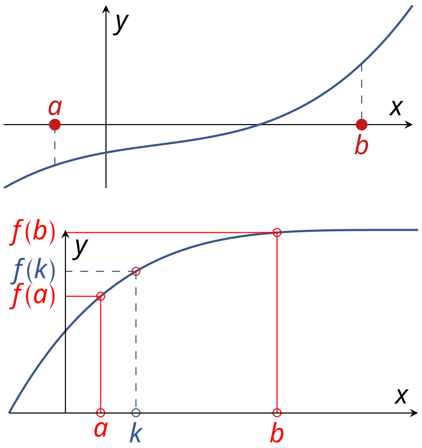
$$x_{i+1} = g(x_i) + g'(x_i)(x - x_i)$$
$$x_{i+2} = g(x_{i+1}) + g'(x_{i+1})(x_{i+1} - x_i)$$
$$|x_{i+2} - x_{i+1}| = |g'(x_i)||x_{i+1} - x_i|$$
Convergence if $|g'(x_i)| \leq 1$

# Today's outline

- Introduction
  - General
- Direct Iteration Method
  - Passing functions
- Bracketing
- Bisection method
- Secant/False Position
- Brent's method

# Bracketing

Bracketing a root involves identifying an interval $(a, b)$ within which the function changes its sign.




- If $f(a)$ and $f(b)$ have opposite signs, it indicates that at least one root lies in the interval $(a, b)$, assuming the function is continuous in the interval.

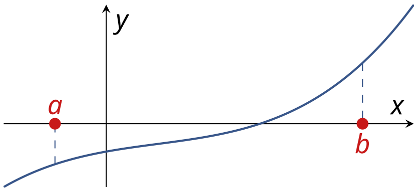
**Intermediate value theorem**

States that if $f(x)$ is continuous on $[a, b]$ and $k$ is a constant lying between $f(a)$ and $f(b)$, then there exists a value $x \in [a, b]$ such that $f(x) = k$.

# Bracketing

**What's the point?**

Bracketing a root = Understanding that the function changes its sign in a specified interval, which is termed as bracketing a root.

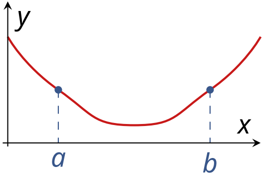



**General best advice**:

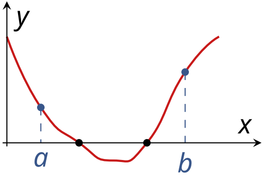
- Always bracket a root before attempting to converge on a solution.
- Never allow your iteration method to get outside the best bracketing bounds...

# General Idea

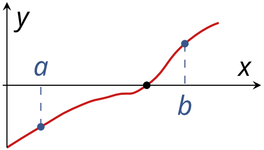
Potential issues to be cautious of while bracketing:



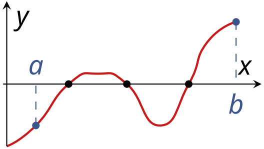

No answer (no root found)




Oops! Encountering two roots




Ideal scenario with one root found




Finding three roots (might work temporarily)

# Bracketing - exercise 2

1. Write a Python function to bracket a function, starting with an initially guessed range $x_1$ and $x_2$ through the expansion of the interval.
2. Develop a program to ascertain the minimum number of roots existing within the $x_1$ and $x_2$ interval.
3. Note: These functions can be integrated to formulate a function that yields bracketing intervals for diverse roots.
4. Test the function for $f(x) = x^2 - 4x + 2$

# Bracketing - exercise 2

- Initially, if feasible, draft a graph using the following Python commands:

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 5, 50)
y = x**2 - 4*x + 2
plt.figure()
plt.plot(x, y, x, np.zeros(len(x)))
plt.axis('tight')
plt.grid(True)
plt.show()
```

- This graphical representation instantly reveals the existence of two roots, evaluated as:

$$x_1 = 2 - \sqrt{2} \approx 0.59 \quad , \qquad x_2 = 2 + \sqrt{2} \approx 3.41$$

# Bracketing - exercise 2

```python
def find_root_by_bracketing(func, x1, x2, tol=1e-6, max_iter=1000):
  # Ensure the bracket is valid
  if func(x1) * func(x2) > 0:
     print('The bracket is invalid. The function must have opposite signs at
           the two endpoints.')
     return False

  # Loop until we find the root or exceed the maximum number of iterations
  for i in range(max_iter):
     # Find the midpoint
     x_mid = (x1 + x2) / 2

     # Check if we found the root
     if abs(func(x_mid)) < tol:
        print(f'Root found: {x_mid}')
        return True

     # Narrow down the bracket
     if func(x_mid) * func(x1) < 0:
        x2 = x_mid
     else:
        x1 = x_mid

  # If we reach here, we did not find the root within the maximum number of
        iterations
  print('Failed to find the root within the maximum number of iterations.')
  return False
```

**Steps:**

- Formulate a function to augment the interval $(x_1, x_2)$ up to a maximum of 250 iterations or until a root is discovered.
- The function should:
  - Return true if a root is found, and false otherwise.
  - Showcase the results.

# Bracketing

## Exercise 2: Function to Bracket a Function

```python
def brak(func, x1, x2, n):
    nroot = 0
    dx = (x2 - x1) / n
    xb1 = []
    xb2 = []

    x = x1
    for i in range(n):
        x += dx
        if func(x) * func(x - dx) <= 0:
            nroot += 1
            xb1.append(x - dx)
            xb2.append(x)

    for i in range(nroot):
        print(f'Root {i+1} in bracketing interval
              [{xb1[i]}, {xb2[i]}]')
    else:
        if nroot == 0:
            print('No roots found!')
```

**Steps:**

- The function subdivides the interval $(x_1, x_2)$ into $n$ parts to check for at least one root.
- It returns the left and right boundaries of the intervals where roots are found in arrays xb1 and xb2.

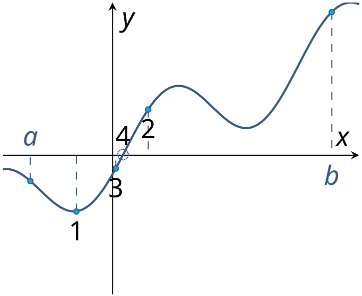
# Today's outline

- Introduction
  - General
- Direct Iteration Method
  - Passing functions
- Bracketing
- Bisection method
- Secant/False Position
- Brent's method

# Bisection Method

Bisection Algorithm:

- Within a certain interval, the function crosses zero, indicated by a change in sign.
- Evaluate the function value at the midpoint of the interval and examine its sign.
- The midpoint then supersedes the limit sharing its sign.




**Properties**

- Pros: The method is infallible.
- Cons: Convergence is relatively slow.

# Bisection Method

**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.
- Specify the required tolerance.
- Output the required number of iterations.
- Implement the same in Python.

# Exercise 3

**Bisection Method in Excel:**

| it | $x_1$ | $x_2$ | $f_1$ | $f_2$ | xmid | fmid | Interval Size |
|---|---|---|---|---|---|---|---|
| 0 | -2 | 2 | 14 | -2 | 0 | 2 | 4 |
| 1 | 0 | 2 | 2 | -2 | 1 | -1 | 2 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| 25 | 0.585786 | 0.585786 | $1 \times 10^{-7}$ | $-6.8 \times 10^{-8}$ | 0.585786 | $1.58 \times 10^{-8}$ | $5.96 \times 10^{-8}$ |

Note: The table represents a sequence of iterations showing how the bisection method converges to a root with each step, demonstrating variable updates and interval size reduction.

# Bisection Method

## Exercise 3: Python Implementation

```python
def bisection(func, a, b, tol, maxIter):
    if func(a) * func(b) > 0:
        print('Error: f(a) and f(b) must have different signs.'
              )
        return None

    iter = 0
    while (b - a) / 2 > tol:
        iter += 1
        if iter >= maxIter:
            print('Maximum iterations reached')
            return None

        c = (a + b) / 2
        print(f'Iteration {iter}: Current estimate: {c}')

        if func(c) == 0:
            return c

        if np.sign(func(c)) != np.sign(func(a)):
            b = c
        else:
            a = c

    return (a + b) / 2
```

- Criterion used for both the function value and the step size.
- While loop usually requires protection for a maximum number of iterations.
- Bisection is sure to converge.
- Root found in 25 iterations. Can we optimize it further?

# Bisection Method

**Required Number of Iterations:**

- Interval bounds containing the root decrease by a factor of 2 after each iteration.

$$\varepsilon_{n+1} = \frac{1}{2}\varepsilon_n \quad \Rightarrow \quad \boxed{n = \log_2 \frac{\varepsilon_0}{tol}}$$

$\varepsilon_0$ = initial bracketing interval,
$tol$ = desired tolerance.

- After 50 iterations, the interval is decreased by a factor of $2^{50} = 10^{15}$.
- Consider machine accuracy when setting tolerance.
- Order of convergence is 1:

$$\boxed{\varepsilon_{n+1} = K\varepsilon_n^m}$$

- $m = 1$: linear convergence.
- $m = 2$: quadratic convergence.

- Bisection method will:
  - Find one of the roots if there is more than one.
  - Find the singularity if there is no root but a singularity exists.

# Today's outline

- Introduction
  - General
- Direct Iteration Method
  - Passing functions
- Bracketing
- Bisection method
- Secant/False Position
- Brent's method

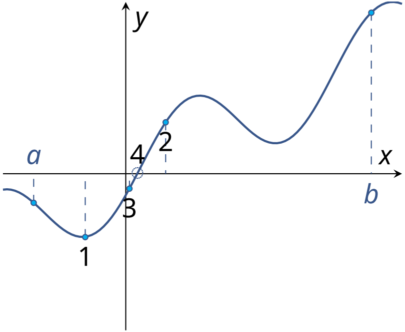
# Secant and False Position Method

**Secant/False Position (Regula Falsi) Method**

- Provides faster convergence given sufficiently smooth behavior.
- Differs from the bisection method in the choice of the next point:
  - **Bisection**: selects the mid-point of the interval.
  - **Secant/False position**: chooses the point where the approximating line intersects the axis.
- Adopts a new estimate by discarding one of the boundary points:
  - **Secant**: retains the most recent of the previous estimates.
  - **False position**: maintains the prior estimate with the opposite sign to ensure the points continue to bracket the root.

# Secant and False Position Method: Comparison

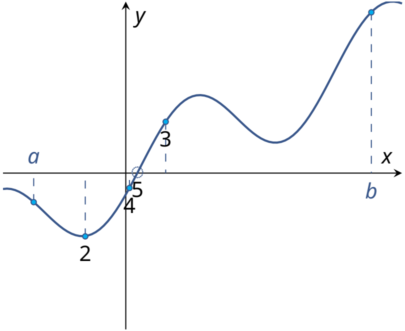
## Secant Method




- Slightly faster convergence:

$$\lim_{n\to\infty} |\varepsilon_{n+1}| = K|\varepsilon_n|^{1.618}$$

## False Position Method




- Guaranteed convergence

# Secant and False Position Method

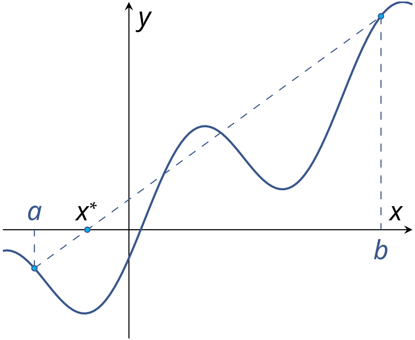
**Exercise 4:**

- Write a function in Excel and Python to find a root of a function using the Secant and False position methods.
- Assume that an initial bracketing interval $(x_1, x_2)$ is provided.
- Specify the required tolerance.
- 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 using the expression:

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

- Determine the abscissa where $f(x^*) = 0$:

$$x^* = a - \frac{f(a)(b-a)}{f(b) - f(a)}$$
$$= \frac{af(b) - bf(a)}{f(b) - f(a)}$$

Note: In the above equations, $a$ and $b$ are the initial guesses/boundaries where the root is suspected to be, and $f(x)$ is the function for which we are finding the root.

# Secant and False Position Method

**Exercise 4:**

- Write a function in Excel and Python 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.
- Specify the required tolerance.
- Output the required number of iterations.
- Compare the bisection, false position, and secant methods.

# Secant and False Position Method

### Exercise 4: False Position Method in Excel

| iteration | xa | xb | fa | fb | x absc | fabsc | interval |
|---|---|---|---|---|---|---|---|
| 0 | -1.5000 | 4.0000 | -0.3895 | 2.1628 | -0.6606 | -0.8455 | 5.5000 |
| 1 | -0.6606 | 4.0000 | -0.8455 | 2.1628 | 0.6493 | 0.6896 | 4.6606 |
| 2 | -0.6606 | 0.6493 | -0.8455 | 0.6896 | 0.0609 | -0.1972 | 1.3099 |
| 3 | 0.0609 | 0.6493 | -0.1972 | 0.6896 | 0.1917 | 0.0070 | 0.5884 |
| 4 | 0.0609 | 0.1917 | -0.1972 | 0.0070 | 0.1873 | -0.0001 | 0.1308 |
| 5 | 0.1873 | 0.1917 | -0.0001 | 0.0070 | 0.1874 | 0.0000 | 0.0045 |
| 6 | 0.1874 | 0.1917 | 0.0000 | 0.0070 | 0.1874 | 0.0000 | 0.0044 |
| 7 | 0.1874 | 0.1917 | 0.0000 | 0.0070 | 0.1874 | 0.0000 | 0.0044 |

Relevant expressions:

- `a=IF((a*fa)<0,a,xbsc)`
- `b=IF((b*fb)<0,b,xbsc)`
- `xbsc=a − fa*(xb−xa)/(fb−fa)`

# Secant and False Position Method

**Exercise 4:**

- Write a function in Excel and Python 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: Secant method in excel

| iteration | x | f |
|---|---|---|
| -1 | 2.0000 | 0.5895 |
| 0 | -1.0000 | -0.7591 |
| 1 | 0.6886 | 0.7368 |
| 2 | -0.1431 | -0.4819 |
| 3 | 0.1857 | -0.0026 |
| 4 | 0.1875 | 0.0002 |
| 5 | 0.1874 | 0.0000 |

## Relevant expressions:

$$x_n = x_{n-1} - f(x_{n-1}) \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$

$$f_n = f(x_n)$$

# Secant and False Position Method

## Exercise 4: False position method in Python

```python
def false_position(f, x0, x1, tol, max_iter):
    if f(x0) * f(x1) > 0:
        raise ValueError('f(x0) and f(x1) must have different signs.')

    history = []

    for i in range(max_iter):
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        history.append(x2)

        if abs(f(x2)) < tol:
            break

        if f(x2) * f(x0) < 0:
            x1 = x2
        else:
            x0 = x2

    root = x2
    return root, history
```

Calling the function:

```python
secant_method(lambda x: x**2 - 4*x + 2, 0, 2, 1e-7, 100)
```

# Secant and False Position Method

## Exercise 4: Secant method in Python

```python
def secant_method(f, x0, x1, tol, max_iter):
    history = [x0, x1]

    for i in range(1, max_iter):
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        history.append(x2)

        if abs(x2 - x1) < tol:
            break

        x0 = x1
        x1 = x2

    root = x1
    return root, history
```

Calling the function:

```python
false_position(lambda x: x**2 — 4*x + 2, 0, 2, 1e—7, 100)
```

# Comparison of Methods

**Exercise 4:**

- $\text{tol}_{\text{eps}}, \text{tol}_{\text{func2}} = 1e-15$, and $(x_1, x_2) = (0, 2)$
- $f(x) = x^2 - 4x + 2 = 0$

| Method | Nr. of iterations |
|---|---|
| Bisection | 52 |
| False position | 22 |
| Secant | 9 |

```python
from scipy.optimize import root_scalar

root_scalar(lambda x: x**2 - 4*x + 2, method='brentq', bracket=[0, 2], xtol=1e-15)
```

Note the initial bracketing steps in root_scalar!

# Today's outline

- Introduction
  - General
- Direct Iteration Method
  - Passing functions
- Bracketing
- Bisection method
- Secant/False Position
- Brent's method

# Brent's Method

**Features of Brent's method:**

- Superlinear convergence with the sureness of bisection
- Keeps track of superlinear convergence, and if not achieved, alternates with bisection steps, ensuring at least linear convergence
- Implemented in MATLAB's `scipy.optimize.fzero` function:
  - Utilizes root-bracketing
  - Bisection/secant/inverse quadratic interpolation
- Inverse quadratic interpolation:
  - Uses three prior points to fit an inverse quadratic function ($x(y)$)
  - Involves contingency plans for roots falling outside the brackets

# Brent's method

**Formulas:**

$$x = b + \frac{P}{Q},$$

$$P = S\left[T(R-T)(c-b) - (1-R)(b-a)\right],$$

$$Q = (T-1)(R-1)(S-1),$$

$$R = \frac{f(b)}{f(c)}$$

$$S = \frac{f(b)}{f(a)}$$

$$T = \frac{f(a)}{f(c)}$$

- b = current best estimate
- $P/Q$ = a 'small' correction

Note: If P/Q does not land within the bounds or if bounds are not collapsing quickly enough, a bisection step is taken.

# Brent's method script

```python
def brent_method(f, a, b, tol=1e-6, max_iter=100):
   if f(a) * f(b) >= 0:
      raise ValueError("f(a) and f(b) must have different signs.")
   # Initialize variables
   c = a
   fa = f(a)
   fb = f(b)
   fc = fa
   history = [a, b]
   d = e = b - a
   for _ in range(max_iter):
      if fa * fc > 0:
         c = a
         fc = fa
         d = e = b - a
      if abs(fc) < abs(fb):
         a, b, c = b, c, a
         fa, fb, fc = fb, fc, fa
      tol1 = 2 * 1.0e-16 * abs(b) + 0.5 * tol
      xm = 0.5 * (c - b)
      if abs(xm) <= tol1 or fb == 0:
         return b, history
      if abs(e) >= tol1 and abs(fa) > abs(fb):
         s = fb / fa
         if a == c:
            # Linear interpolation (Secant method)
            p = 2 * xm * s
            q = 1 - s
         else:
            # Inverse quadratic interpolation
            q = fa / fc
            r = fb / fc
            p = s * (2 * xm * q * (q - r) - (b - a) * (r - 1))
            q = (q - 1) * (r - 1) * (s - 1)
         if p > 0:
            q = -q
         p = abs(p)

         if 2 * p < min(3 * xm * q - abs(tol1 * q), abs(e * q)):
            e = d
            d = p / q
         else:
            d = xm
            e = d
      else:
         d = xm
         e = d
      a = b
      fa = fb
      if abs(d) > tol1:
         b += d
      else:
         b += tol1 if xm > 0 else -tol1

      fb = f(b)
      history.append(b)
   raise ValueError("Maximum number of iterations reached.")
```

# Using Excel for Solving Non-linear Equations: Goal-Seek and Solver

**Setting up Goal-Seek and Solver in Excel:**

- Available in Excel with some prerequisites installation.
- For Excel 2010:
  - Install via `Excel → File → Options → Add-Ins → Go` (at the bottom) → `Select solver add-in`.
  - Accessible through the 'data' menu ('Oplosser' in Dutch).

**Procedure for solving:**

- Select the goal-cell.
- Specify whether you want to minimize, maximize, or set a certain value.
- Define the variable cells for Excel to adjust to find the solution.
- Set the boundary conditions (if any).
- Click 'solve', possibly after setting advanced options.

# Excel: Goal-Seek Example

**Using Goal-Seek to find a solution:**

- The Goal-Seek function can set the goal-cell to a desired value by adjusting another cell.
- Steps:
  1. Open Excel and input the following data:

| A | x | B |
|---|---|---|
| 1 | x | 3 |
| 2 | f(x) | f(x) = -3*B1ˆ2 - 5*B1 + 2 |
| 3 | | |

  2. Navigate to `Data → What-if Analysis → Goal Seek` and input:
     - Set cell: B2
     - To value: 0
     - By changing cell: B1
  3. Press OK to find a solution of approximately 0.3333.

# Excel: Solver Example

**Using Solver to Find Solutions with Boundary Conditions:**

- Solver can adjust values in one or more cells to reach a desired goal-cell value, respecting specified boundary conditions.
- Example sheet setup:

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

- Procedure:
  1. Navigate to `Data → Solver`.
  2. Set the goal function to C2 with a target value of 0.
  3. Add a boundary condition: C3 = 0.
  4. Specify the cells to change as `$B$2:$B$3`.
  5. Click "Solve" to find B2 = 0 and B3 = 2 as solutions.

# Non-linear equations

## Towards the multi-dimensional case

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

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2023-2024

# Today's outline

- Python solvers
- Newton-Raphson method
- Multi-dimensional Newton-Raphson

# Today's outline

- Python solvers
- Newton-Raphson method
- Multi-dimensional Newton-Raphson

# Non-linear Equation Solving in Python (1 var)

**Single Variable Non-linear Zero Finding:**

- Use the `root_scalar` function from `scipy.optimize` for finding zeros of a single-variable non-linear function.
- Be aware of the initial bracketing steps in `root_scalar`.

```python
from scipy.optimize import root_scalar

root_scalar(lambda x: -3*x**2 - 5*x + 2, method='brentq', bracket=[1, 4], xtol=1e-15)
```

```
     converged: True
          flag: converged
function_calls: 10
    iterations: 9
          root: 0.3333333333333333
```

# Non-linear equation solver in Python ($\geq 2$ var)

**Solving Systems of Non-linear Equations (Multiple Variables):**

- Use `fsolve` from `scipy.optimize` for systems involving multiple variables.
- Suitable for non-linear equations with two or more variables.

```python
from scipy.optimize import fsolve

def equations(x):
    return [2*x[0]*x[1] - x[1] + 2, 2*x[1] - 4*x[0] - 4]

fsolve(equations, [1, 1], xtol=1e-15)
```

# Newton-Raphson Method

**Algorithm:**

- Requires evaluating both the function $f(x)$ and its derivative $f'(x)$ at arbitrary points.
- Extend the tangent line at the current point $x_i$ until it intersects with zero.
- Set the next guess $x_{i+1}$ as the abscissa of that zero crossing.
- For small enough $\delta x$ and well-behaved functions, non-linear terms in the Taylor series become unimportant.

$$
\begin{aligned}
f(x) &\approx f(x_i) + f'(x_i)\delta x + \mathcal{O}(\delta x^2) + \ldots \\
0 &\approx f(x_i) + f'(x_i)\delta x \\
\delta x &\approx -\frac{f(x_i)}{f'(x_i)}
\end{aligned}
$$

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

- Can be extended to higher dimensions.
- Requires an initial guess close enough to the root to avoid failure.

# Newton-Raphson Method

**Example with the Formula:**

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

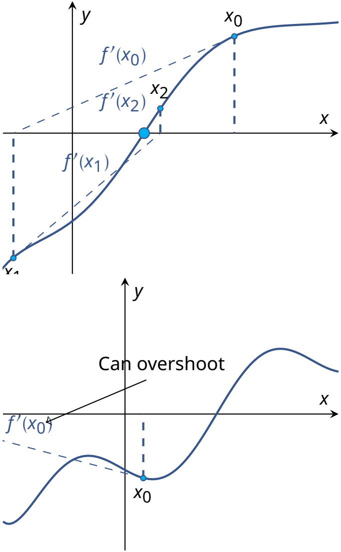
**When it works:**

- Converges enormously fast when it functions correctly.

**When it does not work:**

- Underrelaxation can sometimes be helpful.
- Underrelaxation formula:

$$x_{n+1} = (1-\lambda)x_n + \lambda x_{n+1}$$
$$\lambda \in [0,1]$$




EINDHOVEN UNIVERSITY OF TECHNOLOGY

# Newton-Raphson Method

**Basic Algorithm:**
Given initial $x$ and a required tolerance $\varepsilon > 0$,

1. Compute $f(x)$ and $f'(x)$.
2. If $|f(x)| \leq \varepsilon$, return $x$.
3. Update $x$ using the formula:

$$x \leftarrow x - \frac{f(x)}{f'(x)}$$

Repeat the above steps until a solution is found within the tolerance or the maximum number of iterations is exceeded.

# Newton-Raphson Method

## Exercise 5: Newton-Raphson Method in Excel

| iteration | x | f | f' |
|---|---|---|---|
| 0 | -2 | 14 | -8 |
| 1 | -0.25 | 3.0625 | -4.5 |
| 2 | 0.430556 | 0.463156 | -3.13889 |
| 3 | 0.57811 | 0.021772 | -2.84378 |
| 4 | 0.585766 | 5.86E-05 | -2.82847 |
| 5 | 0.585786 | 4.29E-10 | -2.82843 |
| 6 | 0.585786 | 0 | -2.82843 |

Used formulas:

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

$$f' = 2x - 4$$

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

# Newton-Raphson Method

**Why is the Newton-Raphson so powerful?**

- High rate of convergence
- Can achieve quadratic convergence!

**Derivation of quadratic convergence:**

1. Subtract solution
2. Define error
3. Express in terms of error
4. Use taylor expansion around solution
5. Rewrite in terms of error
6. Ignore higher order terms

$$x_{n+1} - x^* = x_n - x^* - f(x_n)/f'(x_n)$$

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

$$\varepsilon_{n+1} = \varepsilon_n - f(x_n)/f'(x_n)$$

$$\varepsilon_{n+1} \approx \varepsilon_n - \frac{f(x^*) + f'(x^*)\varepsilon_n + f''(x^*)\varepsilon_n^2}{f'(x^*) + \mathcal{O}(\varepsilon_n^2)}$$

$$\varepsilon_{n+1} \approx -\frac{f''(x^*)\varepsilon_n^2 + \mathcal{O}(\varepsilon_n^3)}{f'(x^*) + \mathcal{O}(\varepsilon_n^2)}$$

$$\boxed{\varepsilon_{n+1} \approx -K\varepsilon_n^2}$$

EINDHOVEN UNIVERSITY OF TECHNOLOGY

# Newton-Raphson Method

**Deriving the order of convergence**

- The main issue with determining the order of convergence is that the solution is not known a priori
- To get around this issue it is possible to rewrite the problem in terms of known quantities.
- In the coming derivation, the following steps are taken to derive the order of convergence:
    1. The formal definition of $K$ is given in terms of $\varepsilon$ and the order of convergence $m$
    2. This formal definition is used to rewrite the fraction of successive errors
    3. Logarithms are used to isolate $m$
- Since the $\varepsilon$ can't be computed without knowing the solution, the following approximation is made before plugging the final result:

$$\varepsilon_{n+1} \approx |x_{n+1} - x_n|$$

# Newton-Raphson Method

1. Formal definition of $K$ and $m$:

$$\lim_{n\to\infty} |\varepsilon_{n+1}| = K|\varepsilon_n|^m$$

2. Fraction of successive errors:

$$\frac{|\varepsilon_{n+1}|}{|\varepsilon_n|} = \frac{K|\varepsilon_n|^m}{K|\varepsilon_{n-1}|^m} \Rightarrow \left|\frac{\varepsilon_n}{\varepsilon_{n-1}}\right|^m$$

3. Extracting $m$:

$$\ln\left|\frac{\varepsilon_{n+1}}{\varepsilon_n}\right| = m\ln\left|\frac{\varepsilon_n}{\varepsilon_{n-1}}\right| \Rightarrow \boxed{m = \frac{\ln\left|\frac{\varepsilon_{n+1}}{\varepsilon_n}\right|}{\ln\left|\frac{\varepsilon_n}{\varepsilon_{n-1}}\right|}}$$

# Newton-Raphson Method

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

- In this exercise, you will be working with the Newton-Raphson method implemented in Excel.
- The order of convergence ($m$) can be estimated using the relation:

$$m = \frac{\ln\left(\frac{\varepsilon_{n+1}}{\varepsilon_n}\right)}{\ln\left(\frac{\varepsilon_n}{\varepsilon_{n-1}}\right)}$$

  Where it is assumed that $\varepsilon$ can be approximated by:

$$\varepsilon_{n+1} = |x_{n+1} - x_n|$$

- Solve a problem using the Newton-Raphson method in Excel and verify the order of convergence using the formulas above.

EINDHOVEN UNIVERSITY OF TECHNOLOGY

# Newton-Raphson Method

## Exercise 5: Newton-Raphson Method in Excel solution

| iteration | x | f | f' | eps | m |
|---|---|---|---|---|---|
| 0 | -2.000 | 14.000 | -8.000 | 1.750 | |
| 1 | -0.250 | 3.063 | -4.500 | 0.681 | 1.619 |
| 2 | 0.431 | 0.463 | -3.139 | 0.148 | 1.935 |
| 3 | 0.578 | 0.022 | -2.844 | 0.008 | 1.998 |
| 4 | 0.586 | 0.000 | -2.828 | 0.000 | 2.000 |
| 5 | 0.586 | 0.000 | -2.828 | 0.000 | |
| 6 | 0.586 | 0.000 | -2.828 | | |

Used formulas:

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

$$m = \frac{\ln\left(\frac{\varepsilon_{n+1}}{\varepsilon_n}\right)}{\ln\left(\frac{\varepsilon_n}{\varepsilon_{n-1}}\right)}$$

$$\varepsilon_{n+1} = |x_{n+1} - x_n|$$

# Newton-Raphson Method

**Exercise 6: Newton-Raphson Method in Python**

- Write a Python function to find the root of a function using the Newton-Raphson method.
- Assume that an initial guess $x_0$ is provided.
- The required tolerance for the solution should also be provided.
- Output the results of each iteration.
- Compute the order of convergence.

# Newton-Raphson Method

## Exercise 6: Newton-Raphson in Python solution

```python
def newton1D(f, df, x0, tol, max_iter):
  x = x0
  e = [0] * max_iter
  p = float('nan')
  for i in range(max_iter):
      x_new = x - f(x) / df(x)
      e[i] = abs(x_new - x)
      if i >= 2:
          p = (log(e[i]) - log(e[i - 1])) / (log(e[i - 1]) - log(e[i - 2]))
      print(f'x: {x_new:.10f}, e: {e[i]:.10f}, p: {p:.10f}')
      if e[i] < tol:
          break
      x = x_new
  return x
```

- Running the following command in Python yielded convergence in 6 iterations:

```python
newton1D(lambda x: x**2 - 4*x + 2, lambda x: 2*x - 4, 1, 1e-12, 100)
```

- Question: Why does it not work with an initial guess of $x_0 = 2$?
- This exercise encourages you to think about the influence of the initial guess on the convergence of the Newton-Raphson method.

# Newton-Raphson Method

## Modifications to the Basic Algorithm

- If $f'(x)$ is not known or is difficult to compute/program, a local numerical approximation can be used:

$$f'(x) \approx \frac{f(x+\delta x) - f(x)}{\delta x} \quad (\text{with } \delta x \sim 10^{-8})$$

- The chosen $\delta x$ should be small but not too small to avoid round-off errors.
- The method should be combined with:
  - A bracketing method to prevent the solution from wandering outside of the bounds.
  - A reduced Newton step method for more robustness; don't take the full step if the error doesn't decrease sufficiently.
  - Sophisticated step size controls like local line searches and backtracking using cubic interpolation for global convergence.

# Newton-Raphson Method in Python

## Exercise 6: Numerical Differentiation

```python
from math import log
def newton1Dnum(f, h, x0, tol, max_iter):
  x = x0
  e = [0] * max_iter
  p = float('nan')
  for i in range(max_iter):
      x_new = x - f(x) / ((f(x + h) - f(x)) / h) # NUMERICAL DIFFERENTIATION
      e[i] = abs(x_new - x)
      if i >= 2:
          p = (log(e[i]) - log(e[i - 1])) / (log(e[i - 1]) - log(e[i - 2]))
      print(f'x: {x_new:.10f}, e: {e[i]:.10f}, p: {p:.10f}')
      if e[i] < tol:
          break
      x = x_new
  return x
```

- A command involving numerical differentiation in Python:

```python
newton1Dnum(lambda x: x**2 - 4*x + 2, 1e-7, 1, 1e-12, 100)
```

- This demonstrates that numerical differentiation can be utilized in the Newton-Raphson method to find the roots with the same efficiency in this specific case.

# Newton-Raphson Method

**How to Solve for Arbitrary Functions $f$: “Root Finding”**

- **One-dimensional case**:
  - Move all terms to the left to have $f(x) = 0$.
  - Bracket or ’trap’ a root between bracketing values, then hunt it down "like a rabbit."
- **Multi-dimensional case**:
  - Involving $N$ equations in $N$ unknowns.
  - It is not guaranteed to find a solution; it might not have a real solution or might have more than one solution.
  - Much more challenging compared to the one-dimensional case.
  - It is unpredictable to know if a root is nearby unless it has been found.

# Newton-Raphson Method: Multi-dimensional Case (1)

- **Two-dimensional case:**

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

- **Multivariate Taylor series expansion:**

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

- **Neglecting higher order terms:**

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

- Leads to two linear equations in the unknowns $\delta x$ and $\delta y$:

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

# Newton-Raphson Method: Multi-dimensional Case (2)

**In matrix notation:**

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

**Elements of this equation:**

- Jacobian matrix:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix}$$

- The small displacement vector and **f**:

$$\delta\mathbf{x} = \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \qquad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f(x,y) \\ g(x,y) \end{bmatrix}$$

**Solving equation by matrix inversion:**

- Expressing the stepping equation in matrix notation:

$$\mathbf{J}(\mathbf{x}) \cdot \delta\mathbf{x} = -\mathbf{f}(\mathbf{x})$$

- Multiplying both sides by the inverse of **J**:

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

- Writing in terms of iteration number:

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}(\mathbf{x}_n) \cdot \mathbf{f}(\mathbf{x}_n)}$$

# Newton-Raphson Method: Multi-dimensional Case (2)

**In matrix notation:**

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

**Elements of this equation:**

- Jacobian matrix:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix}$$

- The small displacement vector and **f**:

$$\delta\mathbf{x} = \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \qquad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f(x,y) \\ g(x,y) \end{bmatrix}$$

**Solution via Cramer's rule:**

- Determinant of the Jacobian $\mathtt{det}(\mathbf{J})$:

$$J = \mathtt{det}(\mathbf{J}) = \frac{\partial f}{\partial x}\frac{\partial g}{\partial y} - \frac{\partial f}{\partial y}\frac{\partial g}{\partial x}$$

- Solutions for $\delta x$ and $\delta y$:

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

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

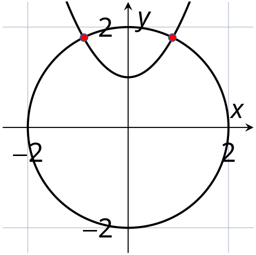
# Newton-Raphson Method: multi-dimensional case

**Example:** *intersection of circle with parabola in matrix form*

$$\begin{aligned} x^2 + y^2 &= 4 \\ y &= x^2 + 1 \end{aligned} \quad \text{can be represented as} \quad \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ f(x) \end{bmatrix} = \begin{bmatrix} x - f(x) \\ x^2 + f(x^2) - 4 \end{bmatrix}$$

Iterations for solving:

| $i$ | $\mathbf{x}$ | $f$ | $\mathbf{J}$ | $\delta\mathbf{x}$ |
|---|---|---|---|---|
| 1 | $\begin{bmatrix} 1.00 \\ 2.00 \end{bmatrix}$ | $\begin{bmatrix} 1.00 \\ 0.00 \end{bmatrix}$ | $\begin{bmatrix} 2.00 & 4.00 \\ 2.00 & -1.00 \end{bmatrix}$ | $\begin{bmatrix} -0.1 \\ -0.2 \end{bmatrix}$ |
| 2 | $\begin{bmatrix} 0.90 \\ 1.80 \end{bmatrix}$ | $\begin{bmatrix} 5.00 \\ 1.00 \end{bmatrix} \times 10^{-2}$ | $\begin{bmatrix} 1.80 & 3.60 \\ 1.80 & -1.00 \end{bmatrix}$ | $\begin{bmatrix} -0.01 \\ -8.7 \times 10^{-3} \end{bmatrix}$ |
| 3 | $\begin{bmatrix} 0.89 \\ 1.79 \end{bmatrix}$ | $\begin{bmatrix} 1.83 \\ 0.11 \end{bmatrix} \times 10^{-4}$ | $\begin{bmatrix} 1.78 & 3.58 \\ 1.78 & -1.00 \end{bmatrix}$ | $\begin{bmatrix} -6.99 \times 10^{-5} \\ -1.65 \times 10^{-5} \end{bmatrix}$ |
| 4 | $\begin{bmatrix} 0.88 \\ 1.79 \end{bmatrix}$ | $\begin{bmatrix} 5.16 \\ 4.89 \end{bmatrix} \times 10^{-9}$ | $\begin{bmatrix} 1.78 & 3.58 \\ 1.78 & -1.00 \end{bmatrix}$ | $\begin{bmatrix} -2.78 \times 10^{-9} \\ 5.94 \times 10^{-11} \end{bmatrix}$ |



# Newton-Raphson Method: multi-dimensional case

**Extensions to multi-dimensional case:**
**Check order of convergence:**

| it | $x_1$ | $x_2$ | eps1 | eps2 | $m_1$ | $m_2$ |
|---|---|---|---|---|---|---|
| 1 | 1.0000 | 2.0000 | | | | |
| 2 | 0.9000 | 1.8000 | 0.1000 | 0.2000 | | |
| 3 | 0.8896 | 1.7913 | 0.0104 | 0.0087 | 1.9835 | 2.9482 |
| 4 | 0.8895 | 1.7913 | 0.0000699 | 0.0000165 | 2.0949 | 2.3208 |
| 5 | 0.8895 | 1.7913 | 0.0000000278 | 0.0000000059 | 2.0589 | 2.1382 |

**Quadratic convergence**

Doubling number of significant digits every iteration

$$m = \frac{\ln(\varepsilon_{n+1}) - \ln(\varepsilon_n)}{\ln(\varepsilon_n) - \ln(\varepsilon_{n-1})}$$

# Newton-Raphson Method

**Deriving the extension to more than two variables:**

1. Generalization to the N-dimensional case
2. Define variables
3. Multi-variate Taylor series expansion
4. Define Jacobian matrix
5. Neglect higher-order terms
6. Express in terms of iterations

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}(\mathbf{x}_n) \cdot \mathbf{f}(\mathbf{x}_n)}$$

1. $f_i(x_1, x_2, \ldots, x_N) = 0$
2. $\mathbf{x} = [x_1, x_2, \ldots, x_N]$ $\mathbf{f} = [f_1, f_2, \ldots, f_N]$
3. $f_i(\mathbf{x} + \delta\mathbf{x}) = f_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial f_i}{\partial x_j} \delta x_j + O(\delta\mathbf{x}^2)$
4. $J_{ij} = \frac{\partial f_i}{\partial x_j}$ $f(\mathbf{x} + \delta\mathbf{x}) = f(\mathbf{x}) + \mathbf{J}\delta\mathbf{x} + O(\delta\mathbf{x}^2)$
5. $\mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{f}(\mathbf{x})$

# Newton-Raphson Method

## Multi-variate Newton-Raphson in Python:

```python
def my_equations(X):
    F = np.zeros(2)
    F[0] = X[0]**2 + X[1]**2 - 4
    F[1] = X[0]**2 - X[1] + 1
    return F
```

```python
def my_jac(x):
    jac = np.zeros((2, 2))
    jac[0, 0] = 2 * x[0]
    jac[0, 1] = 2 * x[1]
    jac[1, 0] = 2 * x[0]
    jac[1, 1] = -1
    return jac
```

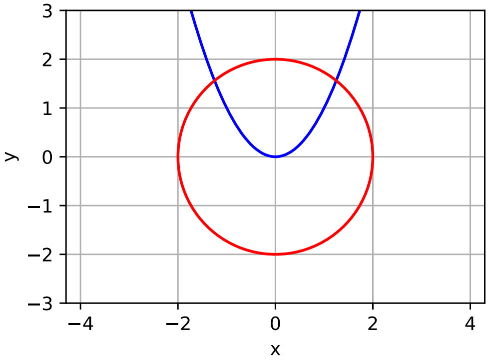
```python
import numpy as np
def newton_nd(f, J, x0, tol, max_iter):
    x = np.array(x0)
    err = np.zeros(max_iter)
    p = np.zeros(max_iter)
    for i in range(max_iter):
        delta_x = -np.linalg.solve(J(x), f(x))
        x += delta_x
        err[i] = np.linalg.norm(delta_x)
        if i > 0:
            p[i] = np.log(err[i]) / np.log(err[i-1])
        else:
            p[i] = float('nan')
        print(f'i = {i}: x = {x}, err = {err[i]:.6e}, p = {p[i]:.6f}')
        if err[i] < tol:
            break
    return x
```

```python
newton_nd(my_equations, my_jac, [1, 2], 1e-12, 100)
```

# Newton-Raphson Method

**Multi-variate Newton-Raphson in Python:**
Plotting the functions:

```
plot_implicit_function(lambda x,y: y-x**2, resolution=100, colors="blue")
plot_implicit_function(lambda x,y: y**2+x**2-4, resolution=100, colors="red")
```

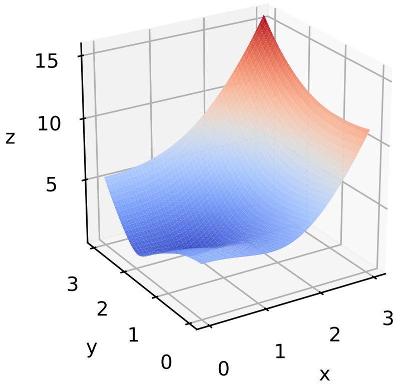



- Code can be found in `plot_implicit.py`
- Uses contour plot at $f(x,y) = 0$

# Newton-Raphson Method

**Multi-variate Newton-Raphson in Python:**
Plotting the norm of the function:

```
plot_surface_function(lambda x,y: np.sqrt((x**2 + y**2 -4)**2+(x**2-y+1)**2),
                      (0,3),(0,3))
```

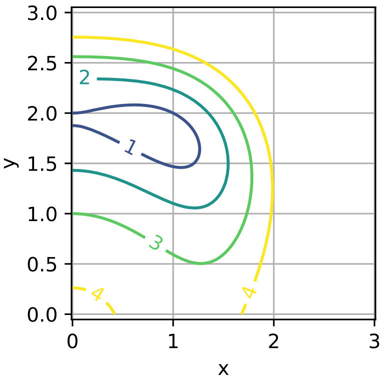



- Code can be found in `plot_implicit.py`
- Uses contour plot at $f(x,y) = 0$

# Newton-Raphson Method

**Multi-variate Newton-Raphson in Python:**
Plotting the norm of the function:

```
plot_contours(lambda x,y: np.sqrt((x**2 + y**2 -4)**2+(x**2-y+1)**2),
              (0, 3), (0, 3), resolution = 100, levels=[0, 1, 2, 3, 4])
```




- Code can be found in `plot_implicit.py`
- Uses contour plot at $f(x,y) = 0$

# 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 is available.
  - If the function evaluation is expensive, the numerical approximation using finite differences can be prohibitive.
- Solution: Use a cheap approximation of the Jacobian! (Secant or 'quasi-Newton' method)
- Comparison:

Newton-Raphson: $J_{ij}(\mathbf{x}) = \dfrac{\partial f_i}{\partial x_j}(\mathbf{x})$ (Analytical)

Secant method: $\mathbf{J}(\mathbf{x})$ approximated by $\mathbf{B}$ (Numerical)

# Broyden's Method

**Approximating $\mathbf{B}^{n+1}$:**

- Multi-dimensional secant method ('quasi-Newton'):
- Secant equation (generalization of 1D case):

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

- Underdetermined (not unique: -n equations with n unknowns), need another condition to pin down $B^{n+1}$.

**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:

$$\mathbf{B}^{n+1} = \mathbf{B}^n + \frac{\left(\delta\mathbf{f}^n - \mathbf{B}^n \cdot \delta\mathbf{x}^n\right)}{\delta\mathbf{x}^n \cdot \delta\mathbf{x}^n} \otimes \delta\mathbf{x}^n$$

- Note: Sometimes $\delta B_{n-1}$ is updated directly.

# Broyden's Method

**Background of Broyden's method:**

- Secant equation:

$$\mathbf{B}^{n+1} \cdot \delta \mathbf{x}^n = \delta f_n$$

- Since there is no update on derivative info, why would $\mathbf{B}^n$ change in a direction orthogonal to $\delta \mathbf{x}^n$?

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

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

- Initialize $\delta \mathbf{x}^n$ and $\mathbf{B}_0$ with the identity matrix (or with finite difference approx.).

# Broyden's Method

**Python implementation of Broyden's method:**

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

```python
import numpy as np
from numpy.linalg import inv

def broyden(F, x0, tol=1e-6, max_iter=100):
    x = np.array(x0)
    B = np.eye(x.size)
    for i in range(max_iter):
        Fx = F(x)
        if np.linalg.norm(Fx) < tol:
            print(f"Converged after {i} iterations.")
            return x
        x_new = x - inv(B)@Fx
        delta_x = x_new - x
        delta_Fx = F(x_new) - Fx
        B += np.outer((delta_Fx - B@delta_x)/(delta_x@delta_x), delta_x)
        x = x_new
    print("Max iterations reached.")
    return x
```

# Broyden's Method

- Same example as before but now with Broyden's method.
- Note how the approximate Jacobian (**B**) is updated over subsequent iterations:

$$\begin{bmatrix}1. & 0.\\0. & 1.\end{bmatrix}\to\begin{bmatrix}3. & -1.\\4. & -1.\end{bmatrix}\to\begin{bmatrix}-1.0 & -9.0\\3.4 & -2.2\end{bmatrix}\to\begin{bmatrix}-1.062 & -9.260\\3.411 & -2.154\end{bmatrix}\to$$

$$\begin{bmatrix}5.290 & -3.864\\2.493 & -2.934\end{bmatrix}\to\begin{bmatrix}7.363 & -1.931\\3.556 & -1.943\end{bmatrix}\to\begin{bmatrix}2.349 & -0.773\\3.547 & -1.941\end{bmatrix}\to\begin{bmatrix}-0.934 & -6.772\\2.351 & -4.124\end{bmatrix}\to$$

$$\begin{bmatrix}-0.384 & -5.879\\2.500 & -3.884\end{bmatrix}\to\begin{bmatrix}10.416 & 6.344\\5.947 & 0.018\end{bmatrix}\to\begin{bmatrix}9.781 & 5.515\\5.641 & -0.382\end{bmatrix}\to\begin{bmatrix}3.577 & 3.630\\3.362 & -1.074\end{bmatrix}\to$$

$$\begin{bmatrix}3.116 & 3.238\\2.912 & -1.458\end{bmatrix}\to\begin{bmatrix}1.992 & 3.272\\1.989 & -1.430\end{bmatrix}\to\ \dots\to\ \dots\to$$

- Compare with analytical jacobian: $\mathbf{B}=\begin{bmatrix}1.748 & 3.261\\1.736 & -1.439\end{bmatrix}$ $\mathbf{J}=\begin{bmatrix}1.779 & 3.583\\1.779 & -1\end{bmatrix}$
- Note that the approximate Jacobian (**B**) is not exact even when the solution has already been found!

# 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!).