Non-linear equations

One dimensional case

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

Multiphase Reactors group
Eindhoven University of Technology

Advanced Numerical Methods for EngD (6PDPPD122), 2023-2024

Outline

- Introduction
- Bracketing
- Bisection method
- Other methods
- Newton-Raphson method
- Python solvers

Outline

Goals

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)

Introduction to underlying ideas and algorithms:

- 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!!

Outline

- Introduction
- Bracketing
- Bisection method
- Other methods
- Newton-Raphson method
- Python solvers

A short 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:

```python
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:

```python
from nonlin_functions import demo_f1
```

Passing Functions in Python

- To solve $f(x) = x^2 - 4x + 2 = 0$ numerically, we first need to 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()
```

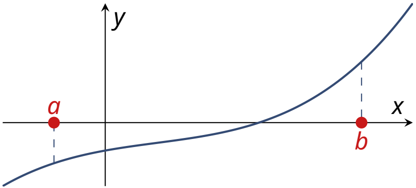
- Now we can call the function with another function, either a lambda function or a common function:

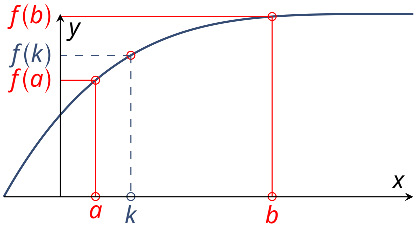
```python
f = lambda x: x**2 - 4*x + 2
draw_my_function(f)
```

Outline

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.

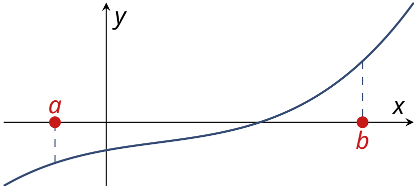
Intermediate value theorem

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.

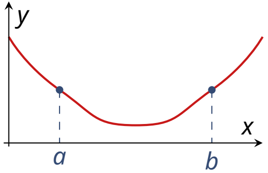


General best advice:

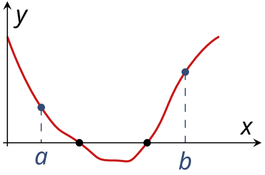
- Always bracket a root before attempting to converge on a solution.
- Never allow your iteration method to get outside the best bracketing bounds...

General Idea

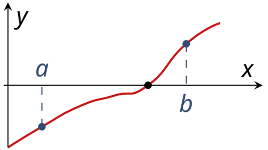
Potential issues to be cautious of while bracketing:



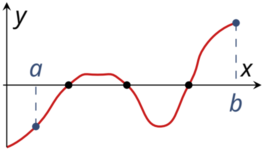
No answer (no root found)



Oops! Encountering two roots



Ideal scenario with one root found



Finding three roots (might work temporarily)

Bracketing exercise

Write a Python function to bracket a function, starting with an initially guessed interval `iv=[x1,x2]` through the expansion of the interval bounds with a growth factor `gr`.

1. Define a function with input `func` and interval `iv`
2. Test whether the interval is currently bracketing a root: `f(x1)*f(x2)<0`
3. If `True`: exit function and return the interval.
4. If `False`:
   - Find the function evaluation (`f(x1)` or `f(x2)`) closest to zero
   - Expand that interval boundary with the growth factor times the interval size
   - Lower interval boundary should decrease, upper interval boundary should increase
   - Re-evaluate the function on the new bounds
5. Repeat from 2

Test the function for $f(x) = x^2 - 4x + 2$

Bracketing exercise

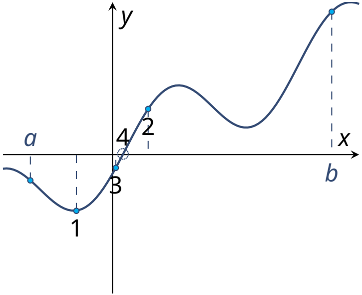
```python
def bracket(func,iv,gr=0.2,max_it=100):
   if not callable(func):
         print("The function func should be a callable function.")
         return
      if len(iv) != 2 or iv[0]==iv[1]:
         print("The interval iv should be a list or array of size 2 with unique values.")
         return

   # Make sure that the interval is ordered and of type float
   iv = np.sort(np.array(iv,dtype=np.float64))
   feval = func(iv)
   it = 0

   while np.prod(feval) > 0:
      interval_size = ...

      # Determine which value is closer to 0

      # Expand interval range

      # Apply new boundaries

      # Safeguard possible divergence
      if (it := it+1) >= max_it:
         print(f"Maximum iterations reached, no bracket found on interval {iv}")
         return False,iv

   print(f"Bracketing interval found: {iv}")
   return True,iv
```

Outline

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 - 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.

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

ε_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.

Outline

- Introduction
- Bracketing
- Bisection method
- **Other methods**
- Newton-Raphson method
- Python solvers

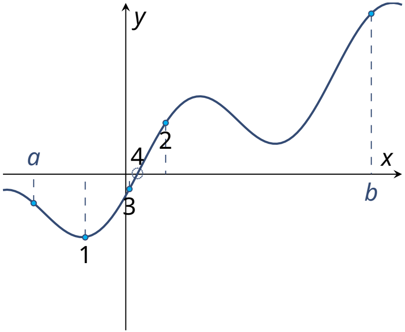
Secant and False Position Method

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

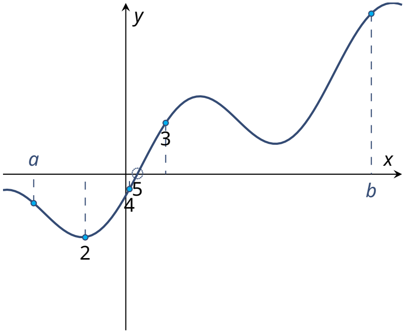
Secant Method



- Slightly faster convergence:

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

False Position Method



- Guaranteed convergence

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 `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

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*B1^2 - 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.

Outline

- Introduction
- Bracketing
- Bisection method
- Other methods
- Newton-Raphson method
- Python solvers

Newton-Raphson Method

Very effective method, often used.

- 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 δx and well-behaved functions, non-linear terms in the Taylor series become unimportant.

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

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

- 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:

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

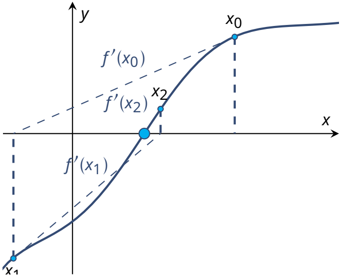
When it works:

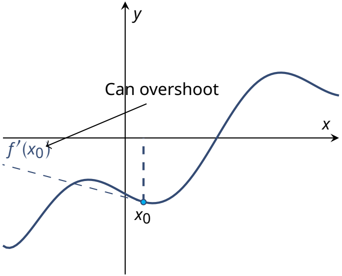
- Converges enormously fast when it functions correctly.

When it does not work:

- Underrelaxation can sometimes be helpful.
- Underrelaxation formula:

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





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

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

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

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

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

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

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

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

Exercise: 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.

Exercise: Newton-Raphson Method in Python

```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 δx 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: 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.

Outline

Non-linear Equation Solving in Python: root$_s$*calar*

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

func = lambda x: -3*x**2 - 5*x + 2

root_scalar(func, method='brentq', bracket=[1, 4], xtol=1e-15)
```

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

Non-linear Equation Solving in Python: fsolve

Single Variable Non-linear Root Finding:

- Use the `fsolve` function from `scipy.optimize` for finding zeros of a single-variable non-linear function.

```python
from scipy.optimize import fsolve

func = lambda x: -3*x**2 - 5*x + 2

ans = fsolve(func, 1, full_output=True)
```

```
(array([-0.66666667]),
 {'nfev': 13,
  'fjac': array([[-1.]]),
  'r': array([1.00000009]),
  'qtf': array([1.32049927e-12]),
  'fvec': array([4.4408921e-16])},
 1,
 'The solution converged.')
```

Non-linear Equation Solving in Python: additional arguments

- Use the help function on `fsolve` or documentation to find out more information about its functionalities
- As an example, we demonstrate the use of a function that takes multiple input parameters:

```python
from scipy.optimize import fsolve

def func(x,a):
   return -3*x**2 - 5*x + a

ans = fsolve(func, 1, args=(4,))
print(ans)
```

```
[0.59066729]
```

Non-linear equation solver in Python (multiple variables)

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

```
from scipy.optimize import fsolve

def equations(x):
   f1 = 2*x[0]*x[1] - x[1] + 2
   f2 = 2*x[1] - 4*x[0] - 4
   return [f1,f2]

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

```
array([-0.5, 1. ])
```