

Python and Programming 1

Programming basics and algorithms

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

● Introduction

- General programming
- First steps
- Further reading

● Data structures

- Data types
- Lists
- Strings
- Tuples
- Dictionaries

● Control flow

- Loops
- Branching

● Functions

- Defining functions
- Recursion
- Scope
- Lambda functions

● Modules

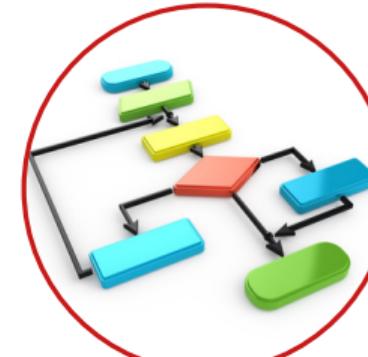
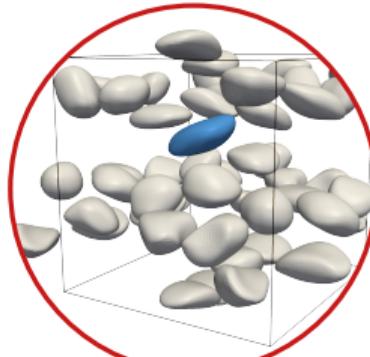
- Using modules
- Math module
- The random module

● Conclusions

● Exercises

Why should you learn something about programming?

- Scientific analyses depend more than ever on computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks
- Ability to understand algorithms by inspection of the code
- Learn to think by dissecting a problem into smaller, easier to solve, parts



Introduction to programming

What is a program?

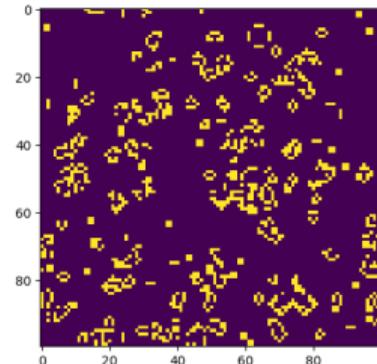
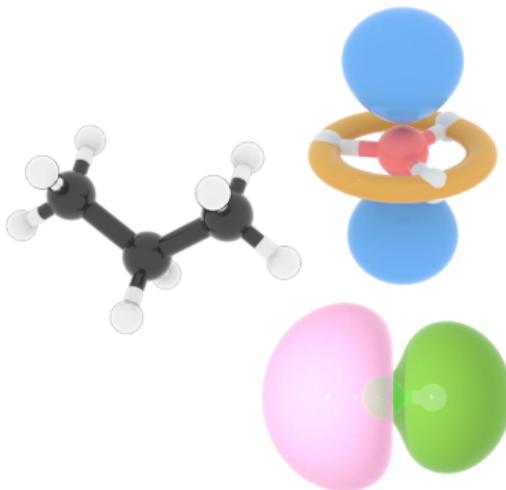
A program is a sequence of instructions that is written to perform a certain task on a computer.

- The computation might be something mathematical, a symbolic operation, image analysis, etc.

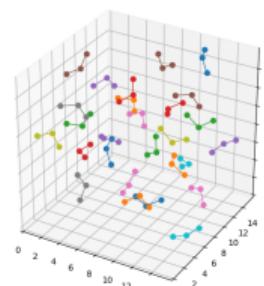
Program layout

- ① Input (Get the radius of a circle)
 - ② Operations (Compute and store the area of the circle)
 - ③ Output (Print the area to the screen)

Versatility of Python



$$\frac{\partial^2 c_{ijk}}{\partial z^2} \approx \text{Div}_{ii} \text{Grad}_{ij} c_{ijk} + \text{Div}_{ii}$$



Versatility of Python: ODE solver

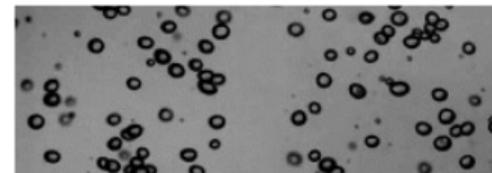
The screenshot shows a Jupyter Notebook environment with the following components:

- Code Editor:** A code cell containing `temp.py` with the following Python code:

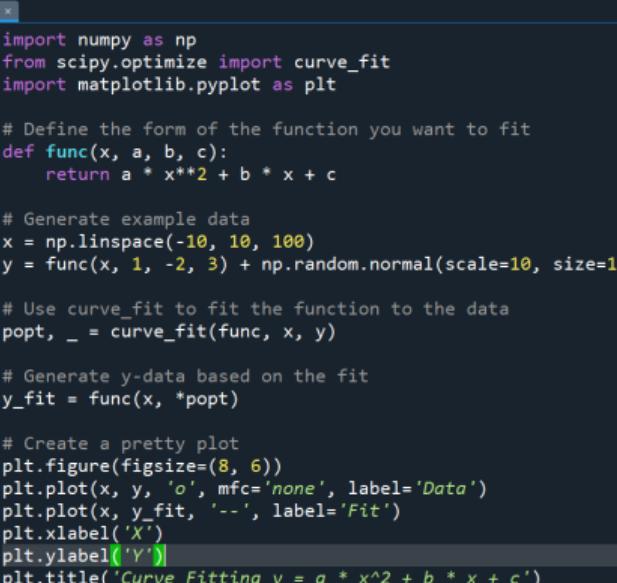
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import odeint
4
5 def ode(y, t):
6     dydt = np.zeros(y.shape, float)
7     dydt[0] = 0 - y[0]
8     dydt[1:] = y[:-1] - y[1:]
9     return dydt
10
11 t = np.linspace(0, 10, 100)
12 y0 = [1, 0, 0, 0, 0]
13 y = odeint(ode, y0, t)
14
15 plt.plot(t, y)
16 plt.grid()
17 plt.xlabel("t [s]")
18 plt.ylabel("c [M]")
19 plt.title("CSTR in series")
20 plt.show()
21
```
- Plot:** A line plot titled "CSTR in series" showing concentration c [M] versus time t [s]. The plot displays five curves starting at $t=0$ with initial concentrations $y_0 = [1, 0, 0, 0, 0]$. The curves represent different species over time.
- Console:** An IPython 8.15.0 session showing the command `In [1]: runfile('C:/Users/phfer/Downloads/temp.py', wdir='C:/Users/phfer/Downloads')`.
- Message Bar:** An orange bar stating "Important" with the note: "Figures are displayed in the Plots pane by default. To make them also appear inline in the console, you need to uncheck 'Mute inline plotting' under the options menu of Plots."
- Page Footer:** Includes the TU Darmstadt logo, page number 7/614, and system status like Python version 3.10.9, CPU usage, and memory usage.

Versatility of Python: Image analysis

```
1 # Importing necessary libraries
2 import numpy as np
3 from scipy import ndimage
4 from PIL.Image import fromarray
5 from skimage import io, color, feature, measure
6
7 # Loading and processing image
8 I = io.imread('bub0.png')
9 BW = color.rgb2gray(I)
10 E = feature.canny(BW)
11 F = ndimage.binary_fill_holes(E)
12
13 # Show final image
14 fromarray(F).show()
```



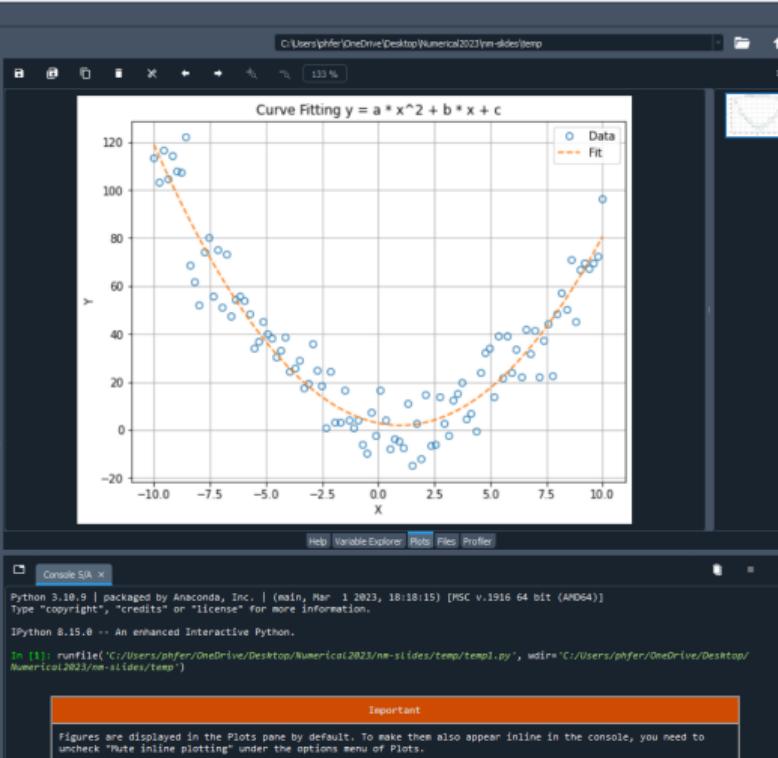
Versatility of Python: Curve fitting



The screenshot shows a Jupyter Notebook interface with the following details:

- Toolbar:** File, Edit, Search, Source, Run, Debug, Consoles, Projects, Tools, View, Help.
- File Explorer:** C:\Users\phifer\OneDrive\Desktop\Numerical2023\nn-slides\temp\temp1.py
- Code Cell:** A code cell titled "temp1.py" containing Python code for curve fitting. The code includes imports for numpy, scipy.optimize, and matplotlib.pyplot, defines a quadratic function, generates example data with noise, fits the data, generates y-data based on the fit, and creates a plot showing the data points and the fitted curve.

```
1 import numpy as np
2 from scipy.optimize import curve_fit
3 import matplotlib.pyplot as plt
4
5 # Define the form of the function you want to fit
6 def func(x, a, b, c):
7     return a * x**2 + b * x + c
8
9 # Generate example data
10 x = np.linspace(-10, 10, 100)
11 y = func(x, 1, -2, 3) + np.random.normal(scale=10, size=100)
12
13 # Use curve_fit to fit the function to the data
14 popt, _ = curve_fit(func, x, y)
15
16 # Generate y-data based on the fit
17 y_fit = func(x, *popt)
18
19 # Create a pretty plot
20 plt.figure(figsize=(8, 6))
21 plt.plot(x, y, 'o', mfc='none', label='Data')
22 plt.plot(x, y_fit, '--', label='Fit')
23 plt.xlabel('X')
24 plt.ylabel('Y')
25 plt.title('Curve Fitting y = a * x^2 + b * x + c')
26 plt.grid(True)
27 plt.legend()
28 plt.show()
```



Getting started

- Start the Python REPL (read–eval–print loop) by running `python` OR `ipython`
- Enter the following commands on the command line. Evaluate the output.

```
>>> 2 + 3 # Some simple calculations
>>> 2 * 3
>>> 2 * 3**2 # Powers are done using **
>>> a = 2 # Storing values into the workspace
>>> b = 3
>>> c = (2 * 3)**2 # Parentheses set priority
>>> 10_000_000 / b
>>> print(a)
>>> print(a,b)
>>> print(a,b,c,sep='--')
>>> print("Numerical methods")
```

```
5
6
18

3333333.3333333335
2
2, 3
2--3--36
Numerical methods
```

Printing and formatting results

You can control the formatting of variables in string literals using various methods - we recommend f-strings. Note that formatting only changes how numbers are *displayed*, not the underlying representation.

```
>>> a = 19/4
>>> print("Few digits {:.2f}".format(a)) # 2 decimal places
>>> print("Many digits {:.10f}".format(a)) # 10 decimal places
>>>
>>> b = 22/7
>>> i = 13
>>> print("Almost pi: %1.4f" % b)
>>> print("i = %d, a = %1.4f and b = %1.8f" % (i,a,b))
>>>
>>> # Using f-strings (Python 3.6+)
>>> c = (21)**0.5 # sqrt of 21
>>> print(f"{c:.10f}") # Float with 10 decimal places
>>> mystr = f"{c:.2e}" # Scientific notation with 2 decimal places in a string object
>>> print(mystr) # Print the string object
>>> print(f"{b=}") # Use = to print variable name and value
>>> print(f"b={:_^15.2}") # Adjust spacing and spacer character
```

A few helpful things

- Using the and keys, you can cycle through recent commands
- Typing part of a command and pressing completes the command and lists the possibilities
- If a computation takes too long, you can press + to stop the program and return to the command line. Stored variables may contain incomplete results.
- Sequences of commands (programs, scripts) are contained as py-files, plain text files with the .py extension.
- Python scripts (and notes/markdown) can also be contained in jupyter notebooks, which have extension .ipynb.
- Such py-files must be in the *current working directory* or in the Python *path*, the locations where Python searches for a command. If you try to run a script that is not in the path, Python will throw an Exception/Error.
- Anything following a # symbol is regarded as a comment
- There are several keyboard shortcuts (vary with text editor) that will make coding much more efficient.

Scripts, notebooks and REPL

- The REPL, indicated by the `>>>` prompt, has the advantage of immediate result after typing a command.
- Larger programs are better written in separate files; either Jupyter notebooks (`.ipynb` files) or plain script files (`.py` files).
- Defining functions in such files will put them in the *scope*, but will not run them until they are actually called.
- The snippets in these slides will continue to use the REPL for single-line commands, and move towards scripts when larger functions are being constructed.

Python help, documentation, resources

- Refer to the Python documentation at [Official documentation](#).
 - Try for instance: `help(print)` or `help(help)`.
- Other packages that we will use:
 - NumPy documentation
 - Matplotlib documentation
 - SciPy documentation
- We supply a number of basic practice/reference modules: Python Crash Course.
- [Python Crash Course, 3rd Edition ↗](#) by Eric Matthes
- [A Whirlwind Tour of Python ↗](#) by Jake Vanderplas
- [Introduction to Scientific Programming with Python ↗](#) by Joakim Sundnes
- [Python Programming And Numerical Methods: A Guide For Engineers And Scientists ↗](#) by Kong, Siauw and Bayen
- Search the web, Reddit, YouTube, etc.

Today's outline

● Introduction

- General programming
- First steps
- Further reading

● Data structures

- Data types
- Lists
- Strings
- Tuples
- Dictionaries

● Control flow

- Loops
- Branching

● Functions

- Defining functions
- Recursion
- Scope
- Lambda functions

● Modules

- Using modules
- Math module
- The random module

● Conclusions

● Exercises

Terminology

Variable Piece of data stored in the computer memory, to be referenced and/or manipulated

Function Piece of code that performs a certain operation/sequence of operations on given input

Operators Mathematical operators (e.g. + - * or /), relational (e.g. < > or ==, and logical operators (**and**, **or**)

Script Piece of code that performs a certain sequence of operations without specified input/output

Expression A command that combines variables, functions, operators and/or values to produce a result.

Variables in Python

- Python stores variables in the *namespace*
- You should recognize the difference between the *identifier* of a variable (its name, e.g. `x`, `setpoint_p`), and the data that it actually stores (e.g. 0.5)
- Python also defines a number of functions by default, e.g. `min`, `max` or `sum`.
 - A list of built-in methods is given by `dir(__builtins__)`
- You can assign a variable by the = sign:

```
>>> x = 4*3  
>>> x  
12
```

- If you don't assign a variable, it will be stored in `_`
- In most text editors, all variables are cleared automatically before the next execution.

Datatypes and variables

Python uses different types of variables:

Datatype	Example
<code>str</code>	'Wednesday'
<code>int</code>	15
<code>float</code>	0.15
<code>list</code>	[0.0, 0.1, 0.2, 'Hello', ['Another', 'List']]
<code>dict</code>	{'name': 'word', "n": 2}
<code>bool</code>	<code>False</code>
<code>tuple</code>	(<code>True</code> , <code>False</code>)

Everything in Python is an object. You can use the `dir()` function to query the possible methods on an object of a datatype (e.g. (`dir(list)`), `dir(28)` or `dir("Yes!")`).

Lists in Python (1)

- Lists are containers of collections of objects
- A list is initialized using square brackets with comma-separated elements

```
>>> brands = ['Audi', 'Toyota', 'Honda', 'Ford', 'Tesla']
```

- Lists can contain and mix any object type, even other lists:

```
>>> another_list = [0.0, 0.1, 0.2, 'Hello', brands]  
>>> print(another_list)
```

```
[0.0, 0.1, 0.2, 'Hello', ['Audi', 'Toyota', 'Honda', 'Ford', 'Tesla']]
```

- Access (i.e., read) an entry in a list. Note that indexing starts at 0:

```
>>> print(another_list[0],another_list[3])
```

```
0.0 Hello
```

Lists in Python (2)

- Manipulate the value of an entry goes likewise:

```
>>> another_list[3] = 'Bye' # Becomes: [0.0, 0.1, 0.2, 'Bye', ['Audi', ...]]
```

- Slicing is used to retrieve multiple elements:

```
>>> another_list[1:4] # This will give the elements from index 1 to index 3
```

```
[0.1, 0.2, 'Bye']
```

- Lists can be unpacked into individual variables:

```
>>> a,b,c,d,e = brands
>>> print(f"The first list element was {a}, then {b}, {c}, {d} and finally {e}.")
```

```
The first list element was Audi, then Toyota, Honda, Ford and finally Tesla.
```

- From here onwards, we will omit the `print` statements from the slides

Lists in Python (3)

- Lists can be concatenated or repeated by the addition and multiplication operators respectively:

```
>>> more_brands = ['Nissan', 'Kia'] + brands
```

```
['Nissan', 'Kia', 'Audi', 'Toyota', 'Honda', 'Ford', 'Tesla']
```

```
>>> zeros = 10*[0]
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Find out which methods can be performed on a list by using `dir(more_brands)`:

```
1 more_brands.append('Volvo') # Append object (here: string literal) at the end of the list
2 more_brands.insert(1,'BMW') # Insert object at index 1
3 more_brands.sort() # Sorts the list in-place
4 item = more_brands.pop(3) # Removes element at index 3 from the list, stores it as item
```

Lists in Python (4)

Ranges of numbers are set using the `range(start=0, stop, step=1)` command:

- Create a list with a range of numbers:

```
>>> a = list(range(1, 11)) # Creates a list from 1 to 10
```

- List comprehensions can be used to create lists with more complex patterns:

```
>>> x = [i/10 for i in range(-10, 11)] # Creates a list from -1 to 1 with a step of 0.1
```

- Manipulating multiple components using slicing and a loop:

```
>>> y = list(range(11)) # Creates a list from 0 to 10
>>> for i in [0, 3, 4, 5, 6]:
>>>     y[i] = 1
```

- Or (by supplying a list instead of a scalar):

```
>>> y[0:2] = [16, 19] # Sets y[0] to 16 and y[1] to 19
```

Assignment of variables; value or reference

Consider the following code snippets:

```
1 >>> i = 3
2 >>> j = i
3 >>> j = i + 3
4 >>> print(i,j)
5 >>> print(id(i), id(j)) # Print memory
   address of data
```

```
3, 6
8885416 8885544
```

```
1 >>> list_a = ['aa', 1, 'bb', 12, True, 1.618]
2 >>> list_b = list_a
3 >>> list_b[2] = 'cc'
4 >>> print(list_a, list_b, sep='\n')
5 >>> print(id(list_a), id(list_b))
```

```
['aa', 1, 'cc', 12, True, 1.618]
['aa', 1, 'cc', 12, True, 1.618]
140285003056512 140285003056512
```

- Primitive or immutable data types (e.g. `int`, `float`, `str`, `tuple`) are *assigned by value*; the value is copied and changes do not affect the original variables.
- Mutable data types (e.g. `list`, `set`, `dict`) are *assigned by reference*; they are two names pointing to the same data, changing one affects the values of the other.

Practice

Given a vector

$$x = [2 \ 4 \ 6 \ 8 \ 10 \ 12 \ 14 \ 16 \ 18 \ 20 \ 30 \ 40 \ 50 \ 60 \ 70 \ 80]$$

- Define the vector using `range`'s, without typing all individual elements
- Investigate the meaning of the following commands:

```
>>> x[2]
>>> x[0:5]
>>> x[:-1]
>>> y = x[4:]
>>> y[3]
>>> y.pop(3)
>>> sum(x)
>>> max(x)
>>> min(x)
>>> x[::-1]
```

Strings in Python (1)

Creating a string:

```
>>> s = "Hello, world!"  
>>> len(s)  
13
```

Accessing a character in a string:

```
>>> s[7]  
'w'
```

Getting a substring:

```
>>> s[7:12]  
'world'
```

Or separate by whitespace using a string method (see `dir(s)`):

```
>>> s.split()  
['Hello', 'world!']
```

Strings in Python (2)

Replacing a substring with another string:

```
>>> s.replace('world', 'Python')
'Hello, Python!'
```

Converting to upper and lower case:

```
>>> s.upper()
'HELLO, WORLD!'
>>> s.lower()
'hello, world!'
```

You can combine methods with string literals too:

```
>>> s.replace('WoRlD'.lower(), 'Python')
'Hello, Python!'
>>> s.startswith('hello'.title())
True
```

Finding the starting index of a substring:

```
>>> s.index("world")
7
```

Practice

Given a string

```
1 >>> s = "Python programming is fun!"
```

- Find and print the index of the word "is".
- Create a new string where "fun" is replaced with "awesome".
- Print the string in uppercase.

Tuples in Python

A tuple is a built-in data type that contains an immutable sequence of values. Creating a tuple:

```
>>> t = (1, 2, 3)
```

Accessing an element of a tuple:

```
>>> t[1]
2
```

Tuples are immutable, so we can't change their elements. However, we can create a new tuple based on the old one:

```
>>> t = t + (4, )
```

Finding the length of a tuple:

```
>>> len(t)
4
```

Practice

Given a tuple

```
1 >>> t = (1, 2, 3, 4, 5, 6)
```

- Access and print the third element of the tuple.
- Try to change the value of the second element of the tuple.
- Create a new tuple by concatenating a second tuple (7,8,9) to the original tuple.

Dictionaries in Python (1)

Creating a dictionary:

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
```

Accessing a value by its key:

```
>>> d['b']
2
```

Modifying a value associated with a key:

```
>>> d['b'] = 47
```

Adding a new key-value pair:

```
>>> d['d'] = 4
```

Removing a key-value pair using pop:

```
>>> d.pop('d')
4
```

Dictionaries in Python (2)

Get all keys as a list:

```
>>> list(d.keys())
['a', 'b', 'c']
```

Get all values as a list:

```
>>> list(d.values())
[1, 47, 3]
```

Get all key-value pairs as a list of tuples:

```
>>> list(d.items())
[('a', 1), ('b', 47), ('c', 3)]
```

Practice

Given a dictionary

```
>>> d = { 'Alice': 24, 'Bob': 27, 'Charlie': 22, 'Dave': 30}
```

- Access and print the age of 'Charlie'.
- Update 'Alice' age to 25.
- Add a new entry for 'Eve' with age 29.
- Print all the keys in the dictionary.

Today's outline

● Introduction

- General programming
- First steps
- Further reading

● Data structures

- Data types
- Lists
- Strings
- Tuples
- Dictionaries

● Control flow

- Loops
- Branching

● Functions

- Defining functions
- Recursion
- Scope
- Lambda functions

● Modules

- Using modules
- Math module
- The random module

● Conclusions

● Exercises

Loops in Python (1)

The `for` loop is used to iterate over a sequence (e.g. lists, sets, tuples, dictionaries, strings). Any *iterable* object can be listed over:

```
>>> for i in range(5):
...     print(i)
0
1
2
3
4
```

You can iterate over a list directly:

```
>>> my_list = [1, 2, 3, 4, 5]
>>> for num in my_list:
...     print(num)
1
2
3
4
5
```

Loops in Python (2)

The **enumerate** keyword returns both the *index* as well as the *list element*:

```
>>> my_list = ['aa', 1, 'bb', 12, True, 1.618034, []]
>>> for idx,elm in enumerate(my_list):
...     print(f'Element {elm} of type {type(elm)} at index {idx}')
```

```
Element aa of type <class 'str'> at index 0
Element 1 of type <class 'int'> at index 1
Element bb of type <class 'str'> at index 2
Element 12 of type <class 'int'> at index 3
Element True of type <class 'bool'> at index 4
Element 1.618034 of type <class 'float'> at index 5
Element [] of type <class 'list'> at index 6
```

Loops in Python (3)

The ‘while’ loop keeps going as long as a condition is **True**:

```
>>> i = 0
>>> while i < 3:
...     print(i)
...     i += 1
0
1
2
```

Use **break** to exit a loop prematurely, and **continue** to skip to the next iteration:

```
>>> for i in range(5):
...     if i == 3:
...         break
...     print(i)
0
1
2
```

Practice

Given a list

```
>>> my_list = [1, 3, 7, 8, 9]
```

- Use a for loop to print each element of the list.
- Use a while loop to print the values at indices 0 to 4.
- Use a loop to find and print the index of the number 7 in the list.

Conditional Statements in Python

The Boolean type `bool` has only 2 possible values: `True` or `False`

The `if` statement is used to execute a block of code only if a condition is evaluated to `True`:

```
>>> x = 5
>>> if x > 0:
...     print("x is positive")
x is positive
```

Use `elif` to specify additional conditions, and `else` to define what to do if no conditions are met:

```
>>> if x > 10:
...     print("x is greater than 10")
... elif x == 10:
...     print("x is exactly 10")
... else:
...     print("x is less than 10")
x is less than 10
```

Nested conditionals

Nesting conditions allows for more complex conditionals:

```
>>> if x > 0:  
... if x % 2 == 0: # The modulo operator % yields the remainder of a division  
... print("x is positive and even")  
... else:  
... print("x is positive but odd")  
... else:  
... print("x is non-positive")  
x is positive but odd
```

The `in` keyword can be used to check membership in a sequence:

```
>>> my_list = [1, 2, 3, 4, 5]  
>>> if 3 in my_list:  
... print("3 is a member of the list")  
3 is a member of the list
```

Combined conditionals

Leap year determination

To be a leap year, the year number must be divisible by four, except for end-of-century years, which must be divisible by 400.

We can create combined conditions using **not**, **and** and **or** to determine whether we have a leap year:

```
>>> year = 2024
>>> if year % 4 == 0 and (not year % 100 == 0 or year % 400 == 0):
...     print(f"{year} is a leap year.")
... else:
...     print(f"{year} is not a leap year.")
```

Conditionals in list comprehensions

List comprehensions can be extended with conditionals too:

```
>>> x = [i for i in range(0,31) if i%3 == 0]
>>> print(x)
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

Conditions are not restricted to modulo's. Here we select artists who have an 'r' or 'R' in their name:

```
artists = ["Adele", "Harry Styles", "Stef Ekkel", "Ed Sheeran", "Nicki Minaj", "Ariana Grande", "Robbie Williams"]

my_artists = [artist for artist in artists if artist.lower().count('r') > 0]
print(my_artists)
['Harry Styles', 'Ed Sheeran', 'Ariana Grande', 'Robbie Williams']
```

Practice

Consider the list:

```
>>> my_list = [x**3 for x in range(1,25,2)] # Cubes of odd numbers
```

- Use an `if` statement to check if 125 is in `my_list`, and print a message indicating the result.
- Write a statement that checks and prints whether `my_list[3]` is divisible by 3, and if not, print the remainder.
- Use a list comprehension to create a list of all numbers in `my_list` that are not divisible by 5.

Today's outline

● Introduction

- General programming
- First steps
- Further reading

● Data structures

- Data types
- Lists
- Strings
- Tuples
- Dictionaries

● Control flow

- Loops
- Branching

● Functions

- Defining functions
- Recursion
- Scope
- Lambda functions

● Modules

- Using modules
- Math module
- The random module

● Conclusions

● Exercises

Functions in Python (1)

Functions are defined using the `def` keyword followed by the function name and a list of parameters in parentheses. The function body starts after the colon:

```
>>> def greet(name):
...     print(f"Hello, {name}!")
```

Call the function with the necessary arguments

```
>>> greet("Alice")
Hello, Alice!
```

Functions can return values using the `return` keyword:

```
>>> def add(a, b):  
...     return a + b
```

Capture the return value in a variable, e.g. `result`

```
>>> result = add(2, 3)
>>> print(result)
5
```

Functions in Python (2)

Default argument values can be specified, making the argument optional:

```
>>> def greet(name, greeting="Hello"):
...     print(f"{greeting}, {name}!")
```

Call the function with or without the optional argument:

```
>>> greet("Bob")
Hello, Bob!
>>> greet("Bob", "Buzz off")
Buzz off, Bob!
```

Python supports functions with a variable number of arguments:

```
>>> def my_function(*args):  
...     print(args)
```

Call the function with a varying number of arguments:

```
>>> my_function(1, 2, 3, "Hello")
(1, 2, 3, "Hello")
```

Functions in Python (3)

Functions can also return multiple values (also >2)

```
>>> def statistics(numbers):
...     return max(numbers), min(numbers)
```

Let's call the function with some list:

```
>>> numlist = [94,12,6,19,33,14,81,56,43,22]
>>> print(statistics(numlist))
(94, 6)
```

Store the elements in separate variables:

```
>>> a,b = statistics(numlist)
>>> print(f'{a=}, {b=}')
a=94, b=6
```

Functions in Python (4)

- Functions are very useful for *abstraction*:
 - You can compartmentalize and 'hide' complex pieces of code
 - Retain flexibility through arguments
 - You can reuse often used pieces of code, limiting copy-paste of code
- Extending functionality or fixing bugs is done in 1 place
- **Documentation is crucial!**

```
>>> def statistics(numbers):
...     """Return the maximum and minimum of a list of numbers
...     Function arguments:
...     numbers: list of numbers"""
...     return max(numbers), min(numbers)
```

```
>>> help(statistics)
```

```
Help on function statistics in module __main__:
```

```
statistics(numbers)
Return the maximum and minimum of a list of numbers
Function arguments:
numbers: list of numbers
```

Passing arguments by value or reference?

Recall that certain Python variables are assigned by value, and others reference; the same goes for passing arguments¹:

- Primitive or immutable data types are *passed by value*; the value is copied and changes made inside the function will not affect the values stored in the variables passed to the function.
- Mutable data types (e.g. list, set, dict) are passed to a function *by reference*; changes that are made inside the function do affect the values outside of the function.

Consider the function:

```
1 def func(x, y):
2     x = x - 1 # Subtract 1
3     y.pop() # Remove last item
```

```
1 i = 1
2 l = ['a', 'b']
3
4 func(i, l)
5
6 print(i, l)
```

```
1, ['a']
```

¹ <https://k0nze.dev/posts/python-copy-reference-none/>

Practice

Define a function that computes the factorial of a number, $n!$:

$$n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n$$

Compute $\exp(x)$ using the Taylor series, iterate until the change is smaller than $1 \cdot 10^{-6}$:

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

Advanced topic: Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function includes a call to itself (a function within a function)
 - This could lead to infinite calls;
 - A base case is required so that recursion is stopped;
 - Base case does not call itself, simply returns.



Advanced topic: Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function includes a call to itself (a function within a function)
 - This could lead to infinite calls;
 - A base case is required so that recursion is stopped;
 - Base case does not call itself, simply returns.



Recursion: example

```
1 def mystery(a, b):
2     if b == 1:
3         # Base case
4         return a
5     else:
6         # Recursive function call
7         return a + mystery(a, b-1)
```

- What does this function do?
- Can you spot the error?
- How deep can you go? Which values of b don't work anymore?

Practice

Define a function that computes the factorial of a number, $n!$, using recursion:

$$n! = 1 \times 2 \times 3 \times \dots \times (n - 1) \times n$$

Scope of functions and variables in Python

In Python, the scope of a variable refers to the regions of a program where that variable is accessible. Understanding the scope of variables helps to avoid bugs and maintain a clean codebase. The scopes in Python are categorized as follows:

Local Scope Variables defined inside a function are in the local scope of that function. They can only be accessed within that function.

Enclosing Scope In the case of nested functions, a function will have access to the variables of the functions it is nested within.

Global Scope Variables defined at the top-level of a script are global and can be accessed by all functions in the script, unless overridden within a function.

Built-in Scope Python has a number of built-in identifiers that should not be used as variable names as they have special significance. Examples include *print*, *list*, *dict*, etc.

Examples Variable Scope

1. Local Scope:

```
1 def my_func():
2     local_var = 100 # Local scope
3     print(local_var)
```

2. Enclosing Scope:

```
1 def outer_func():
2     outer_var = 200 # Enclosing scope
3
4     def inner_func():
5         print(outer_var)
6
7     inner_func()
```

3. Global Scope:

```
1 global_var = 300 # Global scope
2
3 def another_func():
4     print(global_var)
```

4. Built-in Scope:

```
1 print(max, min, len, str, int, list)
```

Exercise Variable Scope

Investigate the behavior of the following nested functions and variables with the same name:

```
1 def outer_func():
2     outer_var = 200
3
4     def inner_func():
5         outer_var = 500
6         print(outer_var)
7
8     inner_func()
```

Lambda functions (1)

Consider the mathematical function $f(x) = x^2 + e^x$ defined as a Python function block:

```
1 def f(x):
2     from math import exp
3     return x**2 + np.exp(x)
```

Note:

- The function is defined using the `def` keyword.
- The variables and `exp` function used are defined *locally*. They will not be available globally unless defined as such.
- The function is defined in a Python script, not in a separate file.

Lambda functions (2)

If you do not want to create a new function block, you can create an *lambda function*: Lambda functions are small, anonymous functions that can be instantiated in a single line, or even as an argument to a function.

```
1 from math import exp
2 f = lambda x: x**2 + exp(x)
```

- `f`: the name of the function
- `lambda`: used to define the inline function
- `x`: the input argument (can be multiple, comma separated)
- `::`: colon indicating the function definition will start
- `x**2 + exp(x)`: the actual function

```
1 xsqr_exp = [f(x) for x in range(5)]
2 print(xsqr_exp)
```

```
[1.0, 3.718281828459045, 11.38905609893065, 29.085536923187668, 70.59815003314424]
```

Today's outline

● Introduction

- General programming
- First steps
- Further reading

● Data structures

- Data types
- Lists
- Strings
- Tuples
- Dictionaries

● Control flow

- Loops
- Branching

● Functions

- Defining functions
- Recursion
- Scope
- Lambda functions

● Modules

- Using modules
- Math module
- The random module

● Conclusions

● Exercises

Using Modules in Python (1)

Modules are files containing Python code, used to organize functionalities and reuse code across projects. To use a module, it must first be imported using the `import` keyword. Here, we import the entire `math` module:

```
>>> import math
```

Once imported, use the dot notation to access functions and variables defined in the module:

```
>>> math.sqrt(16)
4.0
```

You can import specific attributes from a module using the `from ... import ...` syntax:

```
>>> from math import sqrt
>>> sqrt(16)
4.0
```

Using Modules in Python (2)

Alias module names using the `as` keyword to shorten module names and avoid naming conflicts:

```
>>> import numpy as np
```

To view the list of all functions and variables in a module, use the `dir()` function:

```
>>> import math  
>>> dir(math)
```

Get help on how to use a module or a function using the `help()` function:

```
>>> help(math.sqrt)
```

The math module

Many mathematical operations and concepts are available in the **math module**:

```
1 from math import pi,sin,sqrt,log10\
2 ,exp,floor,ceil,factorial,inf,log
3 print(pi)
4 print(sin(0.2*pi))
5 print(sqrt(2))
6 print(log10(10_000))
7 print(exp(1))
8 print(log(exp(2)))
9 print(floor(2.57))
10 print(ceil(2.57))
11 print(floor(-2.57))
12 print(round(2.4))
13 print(round(4.5))
14 print(factorial(5))
15 print(f"1 divided by infinity equals {1/inf}")
```

```
3.141592653589793
0.5877852522924731
1.4142135623730951
4.0
2.718281828459045
2.0
2
3
-3
2
4
120
1 divided by infinity equals 0.0
```

Practice

Use the math module to compute $y = \sin(x)$ for 9 equidistant points $x \in [0, 2\pi]$, including end-points.

- Use *list comprehensions* to generate the lists x and y

The random module (1)

Random number generators and sampling tools are available through the **random module**. A few examples for **integers** and **sequences**:

```
1 import random as rnd
2
3 # Random integers
4 random_integers = [rnd.randint(0,10) for i in range(10)]
5 print(f'{random_integers = }')
6
7 # Sample from given population
8 my_range = range(12) # [0, 1, 2, ..., 11]
9 select_from_range = rnd.sample(my_range,8)
10 print(f'{select_from_range = }') # Selected 8 elements
11
12 # Choose 1 element from list
13 days_of_week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
14 day = rnd.choice(days_of_week)
15 print(f"I've chosen {day} as my lucky day!")
```

```
random_integers = [6, 6, 2, 5, 5, 8, 3, 7, 3, 4]
select_from_range = [2, 3, 10, 1, 6, 4, 8, 5]
I've chosen Wednesday as my lucky day!
```

The random module (2)

Examples for real valued distributions:

```
1 import random as rnd
2
3 # Random number (uniform distribution) 0 < x < 1
4 x = [rnd.random() for i in range(5)]
5 print(x)
6
7 # Random number (uniform distribution) between given bounds
8 x = [rnd.uniform(1,3) for i in range(5)]
9 print(x)
10
11 # Random number from a Gauss distribution (mu=0, sigma=2)
12 x = [rnd.gauss(0,2) for i in range(5)]
13 print(x)
```

```
[0.6697878114597362, 0.4136014290205997, 0.5108247513505662, 0.44260043089156076, 0.9902269207988261]
[2.4749381508841077, 2.943448233960596, 2.516639180020423, 1.0481550073898795, 1.961356325508141]
[1.8199856149229392, 2.000097897306016, -2.4604868187736026, -0.46836605162997846, -2.5069012642608803]
```

Practice

- Create a *function* that returns a list of N dice throws (cubic dice, values 1-6)
- Throw the dice many times
- Print for each value how often it has been thrown

```
[5, 1, 1, 4, 6, 3, 3, 1, 5, 6, 1, 1, 1, 1, 1, 5, 5, 2, 1, 1, 2, 2, 5, 5, 4, 6, 1, 3, 5, 6, 3, 1, 5, 6, 2, 3, 1, 6, 3, 2, 1]
Value 1 was thrown 13 times
Value 2 was thrown 5 times
Value 3 was thrown 6 times
Value 4 was thrown 2 times
Value 5 was thrown 8 times
Value 6 was thrown 6 times
```

Today's outline

● Introduction

- General programming
- First steps
- Further reading

● Data structures

- Data types
- Lists
- Strings
- Tuples
- Dictionaries

● Control flow

- Loops
- Branching

● Functions

- Defining functions
- Recursion
- Scope
- Lambda functions

● Modules

- Using modules
- Math module
- The random module

● Conclusions

● Exercises

In conclusion...

- Python: A versatile development language. Easy to use libraries makes this language multi-purpose and easy to use.
- Programming basics: variables, operators and functions, locality of variables, modules and recursive operations
- For now: exercises on slide deck and Python modules

Practice vectors and arrays

① Create a list x with the elements:

- [2, 4, 6, 8, ..., 16]
- [0, 0.5, 2/3, 3/4, ..., 99/100]

② Create a list x with the elements: $x_n = \frac{(-1)^n}{2n-1}$ for $n = 1, 2, 3, \dots, 200$. Find the sum of the first 50 elements x_1, \dots, x_{50} .

③ Let $x = \text{list}(\text{range}(20, 201, 10))$. Create a list y of the same length as x such that:

- $y[i] = x[i] - 3$
- $y[i] = x[i]$ for every even index i and $y[i] = x[i] + 11$ for every odd index i .

④ Let $T = \text{np.array}([[3, 4, 6], [1, 8, 6], [-4, 3, 6], [5, 6, 6]])$. Perform the following operations on T :

- Retrieve a list consisting of the 2nd and 4th elements of the 3rd row.
- Find the minimum of the 3rd column.
- Find the maximum of the 2nd row.
- Compute the sum of the 2nd column
- Compute the mean of the row 1 and the mean of row 3

Practice plotting

- ① Plot the functions $f(x) = x$, $g(x) = x^3$, $h(x) = e^x$ and $z(x) = e^{x^2}$ over the interval $[0, 4]$ on the normal scale and on the log-log scale. Use an appropriate sampling to get smooth curves. Describe your plots by using the functions: `plt.xlabel`, `plt.ylabel`, `plt.title` and `plt.legend`.
- ② Make a plot of the functions: $f(x) = \sin(1/x)$ and $g(x) = \cos(1/x)$ over the interval $[0.01, 0.1]$. How do you create x so that the plots look sufficiently smooth?

Practice control flow and loops (1)

- ① Write a function that uses two logical input arguments with the following behaviour:

$f(\text{true}, \text{true}) \mapsto \text{false}$

$f(\text{false}, \text{true}) \mapsto \text{true}$

$f(\text{true}, \text{false}) \mapsto \text{true}$

$f(\text{false}, \text{false}) \mapsto \text{false}$

- ② Write a function that computes the factorial of x:

$$f(x) = x! = 1 \times 2 \times 3 \times 4 \times \dots \times x$$

- Using a loop-construction
- Using recursion

Practice control flow and loops (2)

- ① Write a function that computes the exponential function using the Taylor series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

until the last term is smaller than 10^{-6} .

- ② Use a script to compute the result of the following series:

$$f_n = \sum_{n=1}^{\infty} \frac{1}{\pi^2 n^2}$$

This should give you an indication of the fraction this series converges to.

- Now plot in two vertically aligned subplots i) The result as a function of n , and ii) the difference with the earlier mentioned fraction as a function of n . For the latter, consider carefully the axis scale!

Practice logical indexing

- ① Let $x = \text{np.linspace}(-4, 4, 1000)$, $y_1 = 3x^2 - 4x - 6$ and $y_2 = 1.5x - 1$. Use logical indexing to determine function $y_3 = \max(\max(y_1, y_2), 0)$. Plot the function.
- ② Consider these data concerning the age (in years), length (in cm) and weight (in kg) of twelve adult men: $A = [41 \ 25 \ 33 \ 29 \ 64 \ 34 \ 47 \ 38 \ 49 \ 32 \ 26 \ 26]$; $H = [165 \ 186 \ 177 \ 190 \ 156 \ 174 \ 164 \ 205 \ 184 \ 190 \ 165 \ 171]$; $W = [75 \ 90 \ 97 \ 60 \ 74 \ 65 \ 101 \ 85 \ 91 \ 75 \ 87 \ 70]$;
 - Calculate the average of all vectors (age, weight and length).
 - Combine the command `length` with logical indexing to determine how many men in the group are taller than 182 cm.
 - What is the average age of men with a body-mass index ($B \equiv \frac{W}{L^2}$ with W in kg and L in m) larger than 25? And for men with a $B < 25$?
 - How many men are older than the average and at the same time have a BMI below 25?

Practice algorithm: Fourier series for heat equation

The unsteady 1D heat equation in 1D in a slab of material is given as:

$$\frac{\partial T}{\partial t} = k \frac{\partial^2 T}{\partial x^2}$$

We can express the temperature profile $T(x, t)$ in the slab using a Fourier sine series. For an initial profile $T(x, 0) = 20$ and fixed boundary values $T(0, t) = T(L, t) = 0$, the solution is given as:

$$T(x, t) = \sum_{n=1}^{n=\infty} \frac{40(1 - (-1)^n)}{n\pi} \sin\left(\frac{n\pi x}{L}\right) \exp\left(-kt\frac{n\pi^2}{L}\right)$$

- Create a script to solve this equation using loops and/or conditional statements

Python and Programming 2

Programming workflow and advanced features

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

● Scientific computing

- Introduction
- Introduction to NumPy
- Math with NumPy
- Array operations

● Plotting with Matplotlib

- Line plots
- Different plot styles

● IO

● Coding style

- Program design

● Debugging and profiling

● Concluding remarks

Introduction to NumPy (1)

NumPy is a powerful library for numerical computing in Python. It introduces the multidimensional array (`ndarray`) data structure which is central to numerical computing tasks. To start using NumPy, you need to import it first:

```
>>> import numpy as np
```

Creating arrays from Python lists and accessing elements:

```
>>> arr = np.array([1, 2, 3, 4, 5])
>>> arr[0]
1
```

Create arrays with predetermined values:

```
>>> np.zeros(5) # Creates an array of five zeros  
>>> np.ones((3, 3)) # Creates a 3x3 array of ones - note that the input argument is a tuple
```

Introduction to NumPy (2)

Useful array creation functions:

```
>>> arr = np.arange(0, 10, 2) # Creates an array with values from 0 to 10, step 2
>>> arr2 = np.linspace(0, 1, 5) # Creates an array with 5 values evenly spaced between 0 and 1
>>> arr3 = np.logspace(-3, 2, 6) # Creates an array spaced evenly on a logscale
```

Array operations:

```
>>> arr * 2 # Multiplies every element by 2  
>>> arr + np.array([5, 4, 3, 2, 1]) # Element-wise addition
```

Inspecting your array:

```
>>> arr.shape # Returns the dimensions of the array (tuple)
>>> arr.shape[0] # Returns the number of rows ([1] for columns)
>>> len(arr) # Returns the size of the first dimension
>>> arr.size # Returns the total number of elements in arr
```

Introduction to NumPy (3)

Zeros- or ones-filled arrays:

```
>>> many_zeros = np.zeros_like(arr) # Array with zeros of size like arr, also: np.zeros  
>>> just_ones = np.ones((2,4)) # Create array with ones of size (2,4), also: np.ones_like  
>>> identity = np.eye((3,3)) # Identity matrix (zeros with ones on the main diagonal)
```

Or random numbers, useful for e.g. stochastic simulations. Note that these are NumPy intrinsic methods, and differ from the ones in the `random` module.

```
>>> np.random.rand() # Single random floating point number [0,1)  
0.11879317099184983  
  
>>> np.random.rand(2,3)  
array([[0.36482694, 0.47204049, 0.20908193],  
       [0.76626339, 0.64075302, 0.82440279]])  
  
>>> np.random.randint(1,10,5) # 5 random integers from 1 (inclusive) until 10 (exclusive)  
array([9, 1, 6, 4, 6])
```

Math with NumPy (1)

NumPy provides a rich set of functions to perform mathematical operations on arrays. Let's explore some categories of these operations.

- Linear Algebra:

```
>>> np.dot(arr, arr) # Dot product (alternative: arr @ arr)
>>> np.linalg.norm(arr) # Euclidean norm (L2 norm)
```

- Trigonometric Functions:

```
>>> np.sin(arr) # Sine of each element  
>>> np.cos(arr) # Cosine of each element
```

- Statistical functions to analyze data:

```
>>> np.mean(arr) # Mean value of the array elements  
>>> np.std(arr) # Standard deviation of the array elements
```

Math with NumPy (2)

Table: Useful NumPy Functions for Beginners

Function	Description
np.array	Create a NumPy array
np.zeros	Create an array filled with zeros
np.ones	Create an array filled with ones
np.arange	Create an array with evenly spaced values
np.linspace	Create an array with a specified number of evenly spaced values
np.sin	Sine function
np.cos	Cosine function
np.tan	Tangent function
np.exp	Exponential function
np.log	Natural logarithm
np.sqrt	Square root function
np.dot	Dot product of two arrays
np.linalg.norm	Calculate the norm of an array
np.mean	Calculate the mean of an array
np.std	Calculate the standard deviation of an array

Array operations (1)

In Python, NumPy ndarrays enable efficient operations on arrays, particularly mathematical operations associated with linear algebra. Let's look at how we can create and manipulate matrices using NumPy ndarrays:

- Manually creating matrices (2D ndarrays):

```
>>> import numpy as np
>>> matrix = np.array([[1, 2],
                     [3, 4]]) # Matrices are arrays of arrays(rows)
```

- Reshaping, slicing and modifying matrices (try for yourself and print each new result):

```
>>> mat = np.arange(25).reshape(5,5)
>>> col = mat[:,2] # Get column index 2
>>> row = mat[3,:] # Get row index 3
>>> sub = mat[1:4,1:4] # Save submatrix
>>> mat[1,:] = row # Replace row index 1 by 'row'
>>> mat[mat>5] = -1 # Conditional slicing/update of values
```

Array operations (2)

The real power of NumPy lies in the vectorized computations:

- Omit loops, write computations in natural language
- Shorter code
- Much, much faster

```
1 import numpy as np
2 import time
3
4 start = time.time()
5 x = np.linspace(0,2*np.pi,10000000)
6 y = np.exp(-x) * (2+np.sin(2*np.pi*x))
7 total_time = time.time() - start
8
9 print(f'{total_time = }')
```

```
total_time = 0.35589122772216797
```

Array operations (3)

If you do need to iterate over the elements:

- Think twice if you *really* have to
- If the answer is still yes, you can use `np.nditer` OR `np.ndenumerate`:

```
1 import numpy as np
2
3 # A 2D array
4 data = np.array([[11,12,13], [14,15,16]])
5
6 # Loop over each element using nditer
7 for val in np.nditer(data):
8     print(f"Value: {val}")
9
10 # Loop using ndenumerate (gives index + value)
11 for i,val in np.ndenumerate(data):
12     print(f"Enumerate index {i} has value {val};"
13          f" also {data[i]}")
```

```
Value: 11
Value: 12
Value: 13
Value: 14
Value: 15
Value: 16
Enumerate index (0, 0) has value 11; also 11
Enumerate index (0, 1) has value 12; also 12
Enumerate index (0, 2) has value 13; also 13
Enumerate index (1, 0) has value 14; also 14
Enumerate index (1, 1) has value 15; also 15
Enumerate index (1, 2) has value 16; also 16
```

Practice

Given a function $f(x) = x^2 + 2x - 4$

- Create a large `np.linspace` of $x \in [0, 20]$ e.g. with 1M numbers
- Compute $f(x)$ iteratively (i.e. with a `.loop`)
- Compute $f(x)$ vectorized (i.e. using NumPy operations)
- Compare timings

Assignment of arrays by reference

Careful with assignment, default is a reference to the same object as observed with lists:

```
>>> arr = np.arange(0,10,1)
>>> print(arr)
[0 1 2 3 4 5 6 7 8 9]
>>> new_arr = arr
>>> new_arr2 = arr.copy()
>>> new_arr[4] = 11
>>> print(new_arr)
[ 0 1 2 3 11 5 6 7 8 9]
>>> print(arr) # :mind_blown: we did not change this, did we?
[ 0 1 2 3 11 5 6 7 8 9]
```

Saving and loading matrices:

```
>>> np.save('my_matrix.npy', matrix) # Save the matrix to a file
>>> loaded_matrix = np.load('my_matrix.npy') # Load the matrix from a file
```

Today's outline

● Scientific computing

- Introduction
- Introduction to NumPy
- Math with NumPy
- Array operations

● Plotting with Matplotlib

- Line plots
- Different plot styles

● IO

● Coding style

- Program design

● Debugging and profiling

● Concluding remarks

Introduction to Matplotlib

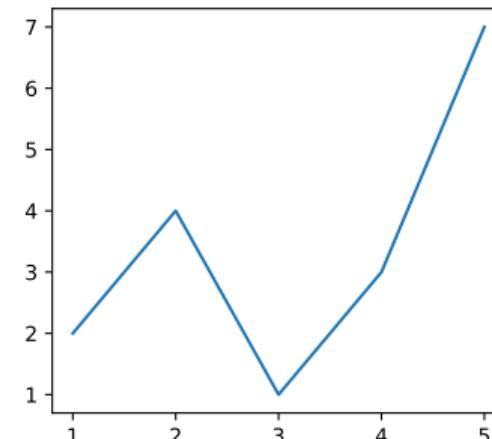
Matplotlib is a Python library used widely for creating static, animated, and interactive visualizations. To start, we first import the `pyplot` module from `matplotlib`.

```
1 import matplotlib.pyplot as plt
```

Creating a simple line plot:

```
2 x = [1, 2, 3, 4, 5]
3 y = [2, 4, 1, 3, 7]
4 plt.plot(x, y)
5 plt.show()
```

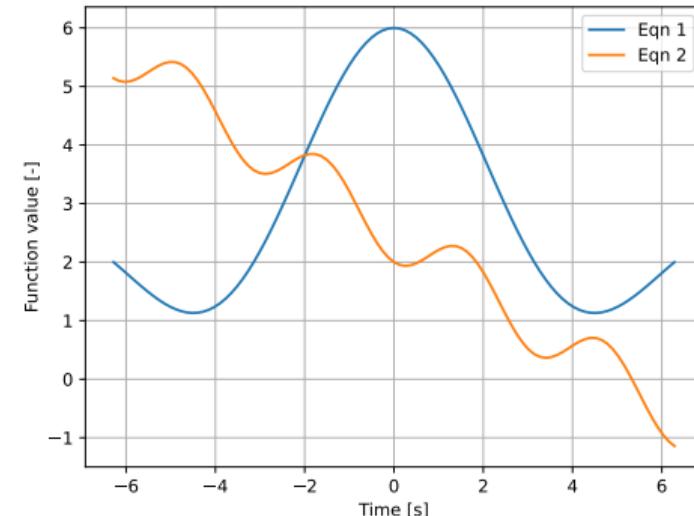
In Jupyter Notebooks, use `%matplotlib` to display figures in a separate window, or `%matplotlib inline` to display in the notebook.



Brushing up the graph

Adding multiple plots, axis labels and a legend:

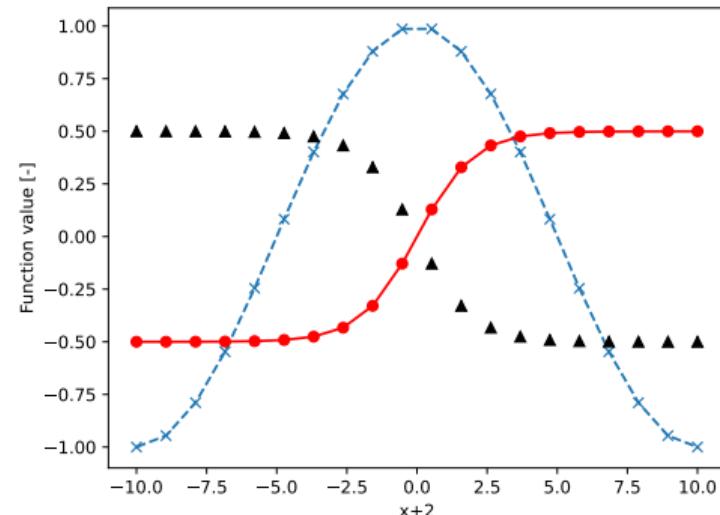
```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 t = np.linspace(-2*np.pi, 2*np.pi, 101)
5 y1 = 4*np.sin(t)/t + 2
6 y2 = np.sin(t)**2 - t/2 + 2
7
8 plt.plot(t,y1,label='Eqn 1')
9 plt.plot(t,y2,label='Eqn 2')
10 plt.xlabel('Time [s]')
11 plt.ylabel('Function value [-]')
12 plt.legend()
13 plt.grid()
14 plt.show()
```



Line plot styles

Changing markers, line styles:

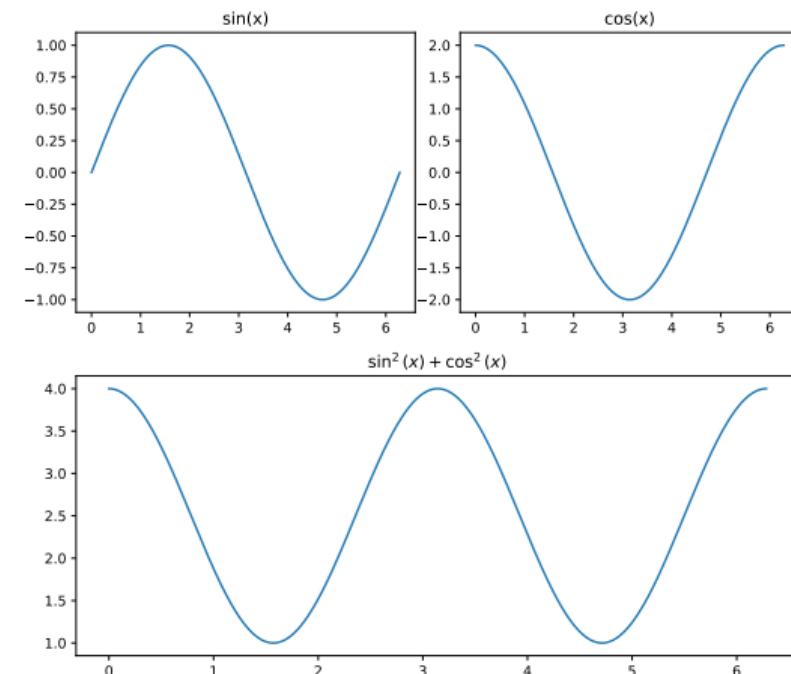
```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 t = np.linspace(-10.0, 10.0, 20)
5 p = np.cos(np.pi * t/10)
6 q = 1/(1+np.exp(-t)) - 0.5
7
8 plt.plot(t,p,'--x') # Dashed line, cross
9 plt.plot(t,q,'r-o') # Red, line, circles
10 plt.plot(t,-q,'k^') # Black triangle marker
11 plt.xlabel('x+2')
12 plt.ylabel('Function value [-]')
13 plt.show()
```



Subplots

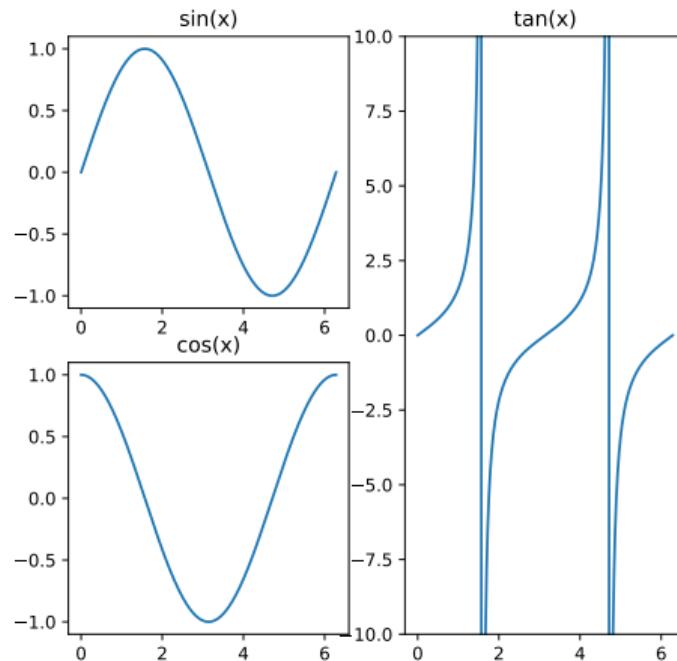
Multiple plots in a single figure are made by specifying a grid using `subplot`:

```
1 x = np.linspace(0,2*np.pi,1000)
2 y1,y2 = np.sin(x), 2*np.cos(x)
3
4 # Specify figure size
5 fig = plt.figure(figsize=(8, 7))
6
7 # First index of 2x2 grid (top-left)
8 ax1 = plt.subplot(2, 2, 1)
9 ax1.plot(x,y1)
10 ax1.set_title('sin(x)')
11
12 # Second index of 2x2 grid (top-right)
13 ax2 = plt.subplot(2, 2, 2)
14 ax2.plot(x,y2)
15 ax2.set_title('cos(x)')
16
17 # Second index of 2x1 grid (whole bottom)
18 ax3 = plt.subplot(2, 1, 2)
19 ax3.plot(x,y1**2+y2**2)
20 ax3.set_title(r'$\sin^2(x)+\cos^2(x)$')
21
22 plt.tight_layout()
23 plt.show()
```



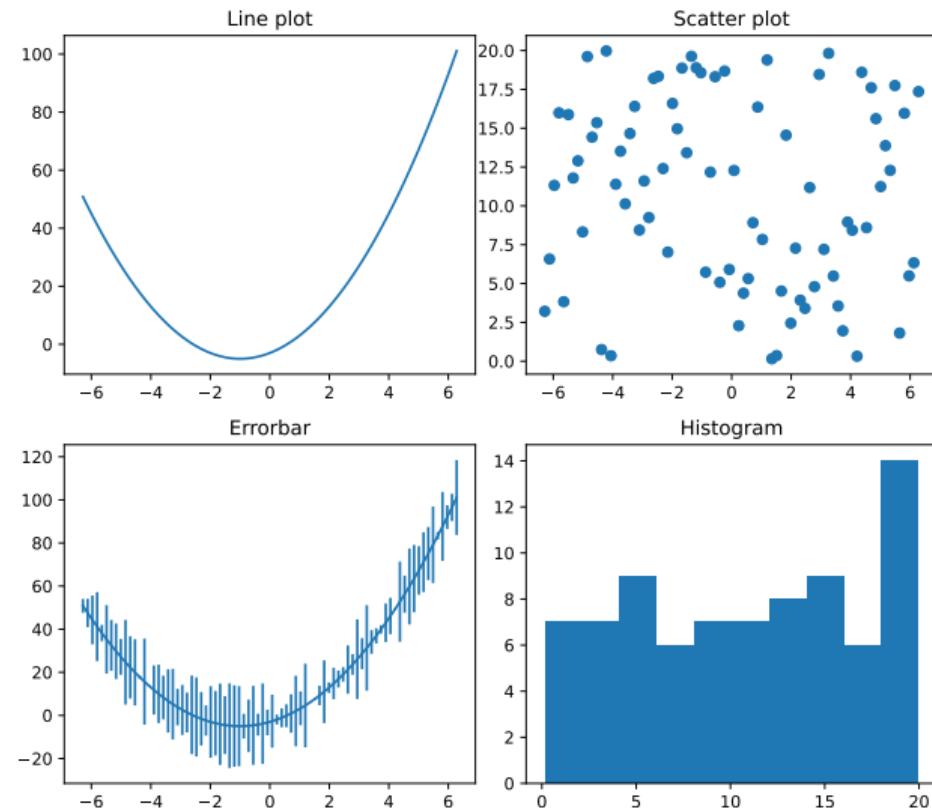
Practice

Try to create the following figure:



2D plot styles

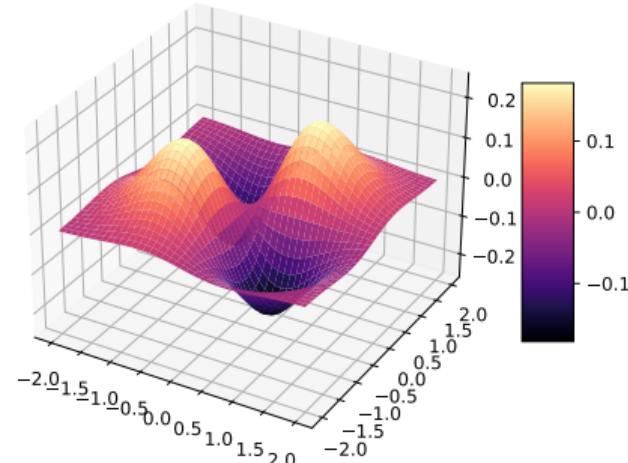
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(-2*np.pi,2*np.pi,80)
5 y2 = 2*x**2 + 4*x - 3
6 y3 = 20*np.random.rand(y2.size)
7
8 ax1 = plt.subplot(2, 2, 1)
9 ax1.plot(x,y2)
10 ax1.set_title('Line plot')
11
12 ax2 = plt.subplot(2, 2, 2)
13 ax2.scatter(x,y3)
14 ax2.set_title('Scatter plot')
15
16 ax3 = plt.subplot(2, 2, 3)
17 ax3.errorbar(x,y2,yerr=y3)
18 ax3.set_title('Errorbar')
19
20 ax4 = plt.subplot(2,2,4)
21 ax4.hist(y3)
22 ax4.set_title('Histogram')
23
24 plt.show()
```



Surface plot

We draw a surface plot of $f(x,y) = xye^{(-x^2-y^2)}$:

```
1 from mpl_toolkits.mplot3d import Axes3D
2 from matplotlib import cm
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 fig = plt.figure()
7 ax = fig.add_subplot(111, projection='3d')
8
9 # Make data.
10 x = np.arange(-2, 2, 0.025)
11 y = np.arange(-2, 2, 0.025)
12 x,y = np.meshgrid(x, y)
13 z = x * y * np.exp(-x**2 - y**2)
14
15 # Plot the surface.
16 surf = ax.plot_surface(x,y,z,
17     cmap=cm.magma, linewidth=0, antialiased=False)
18
19 # Customize the z axis.
20 ax.set_zlim(-0.25, 0.25)
21
22 # Add a color bar which maps values to colors.
23 fig.colorbar(surf, shrink=0.5, aspect=5)
24
25 plt.show()
```



Advanced plotting: Animating plots

The `fig.canvas.draw()` and `fig.canvas.flush_events()` methods hold the execution of your program until the graph is updated, facilitating a live view of the simulation result.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(0, 4*np.pi, 100)
5 y = np.sin(x)
6
7 fig, ax = plt.subplots()
8 line, = ax.plot(x, y, '-')
9
10 ax.set_xlabel('X')
11 ax.set_ylabel('Y')
12 ax.set_title('Animating a Sine Wave')
13 plt.show(block=False)
14
15 for i in range(len(x)):
16     line.set_data(x[:i+1], y[:i+1])
17     fig.canvas.draw()
18     fig.canvas.flush_events()
19     plt.pause(0.01)
```

Today's outline

● Scientific computing

- Introduction
- Introduction to NumPy
- Math with NumPy
- Array operations

● Plotting with Matplotlib

- Line plots
- Different plot styles

● IO

● Coding style

- Program design

● Debugging and profiling

● Concluding remarks

Input and Output in Python (1)

Many programs require some input (data) to function correctly. A combination of the following is common:

- Input may be given in a parameters file ("hard-coded")
- Input may be entered via the keyboard using the 'input' function:

```
>>> a = input('Please enter a number: ')
```

- Input may be read from a file, for instance using Python's built-in open function or libraries like 'numpy' for more complex data structures:

```
with open('myData.txt', 'r') as file:  
    data = file.read() # Alt: data.readlines() gives a list of lines  
  
import numpy as np  
data = np.loadtxt("my_file.csv")
```

- There are many other libraries and functions for more advanced input operations, such as json, xml, etc.

Input and Output in Python (2)

Output of results can be done in several ways, including:

- Displaying results to the console - simply omitting a print statement will automatically show expression results in most Python IDEs.
- Using the 'print' function to show results in the console:

```
>>> print("The result is:", result)
```

- Saving data to a file can be done using various methods including writing to a file or using libraries like pandas for structured data:

```
>>> with open('output.txt', 'w') as file:  
...     file.write(str(data))  
  
>>> data.save("data.npy")
```

- More advanced output methods can utilize libraries such as NumPy, pandas, etc. to save data in various formats including JSON, Excel, etc.

Today's outline

● Scientific computing

- Introduction
 - Introduction to NumPy
 - Math with NumPy
 - Array operations

● Plotting with Matplotlib

- Line plots
 - Different plot styles

• 10

● Coding style

- Program design

● Debugging and profiling

● Concluding remarks

Make a habit of the following adage

MAKE IT WORK

MAKE IT RIGHT

MAKE IT FAST

Make it work

Use the building blocks of previous lecture to create an algorithm:

- ① *Problem analysis*
Contextual understanding of the nature of the problem to be solved
 - ② *Problem statement*
Develop a detailed statement of the mathematical problem to be solved with the program
 - ③ *Processing scheme*
Define the inputs and outputs of the program
 - ④ *Algorithm*
A step-by-step procedure of all actions to be taken by the program (*pseudo-code*)
 - ⑤ *Program the algorithm*
Convert the algorithm into a computer language, and debug until it runs
 - ⑥ *Evaluation*
Test all of the options and conduct a validation study

Now it's time to make it right!

Interpret the following code

```
1 s = checksc()
2 if s:
3     a = cb()
4     b = cfrsp()
5     if a < 5:
6         if b > 5:
7             a = gtbs()
8             if a > b:
9                 ubx()
10 else:
11     brn()
12     gtbs()
```

WAT



Enhancing Readability with Proper Indentation

Proper indentation is not just about syntax in Python; it significantly impacts the readability of the code. Python recommends 4 spaces for indentation. Let's see the difference:

```
1 s = checksc()
2 if s:
3     a = cb()
4     b = cfrsp()
5     if a < 5:
6         if b > 5:
7             a = gtbs()
8             if a > b:
9                 ubx()
10            else:
11                brn()
12                gtbs()
```

```
1 s = checksc()
2 if s:
3     a = cb()
4     b = cfrsp()
5     if a < 5:
6         if b > 5:
7             a = gtbs()
8             if a > b:
9                 ubx()
10            else:
11                brn()
12                gtbs()
```

Readable Variables and Function Names

Making use of descriptive variable and function names enhances the readability and understanding of the code.

```
1 s = checksc()
2 if s:
3     a = cb()
4     b = cfrsp()
5     if a < 5:
6         if b > 5:
7             a = gtbs()
8         if a > b:
9             ubx()
10 else:
11     brn()
12     gtbs()
```

```
1 isAvailable = checkSchedule()
2 if isAvailable:
3     bookCount = countBooks()
4     freeShelfSpace = checkFreeShelfSpace()
5     if bookCount < 5:
6         if freeShelfSpace > 5:
7             bookCount = visitBookStore()
8         if bookCount > freeShelfSpace:
9             useStorageBox()
10 else:
11     burnBooks()
12     visitBookStore()
```

Avoiding Magic Numbers in the Code

Magic numbers are constant values without a name, which can reduce code readability.
Replacing them with named constants can enhance the understanding of the code.

```
1 isAvailable = checkSchedule()
2 if isAvailable:
3     bookCount = countBooks()
4     freeShelfSpace = checkFreeShelfSpace()
5     if bookCount < 5:
6         if freeShelfSpace > 5:
7             bookCount = visitBookStore()
8             if bookCount > freeShelfSpace:
9                 useStorageBox()
10 else:
11     burnBooks()
12     visitBookStore()
```

```
1 MAX_SHELF_SPACE = 5
2 MIN_BOOKS_REQUIRED = 5
3
4 isAvailable = checkSchedule()
5 if isAvailable:
6     bookCount = countBooks()
7     freeShelfSpace = checkFreeShelfSpace()
8     if bookCount < MAX_SHELF_SPACE:
9         if freeShelfSpace > MIN_BOOKS_REQUIRED:
10            bookCount = visitBookStore()
11            if bookCount > freeShelfSpace:
12                useStorageBox()
13 else:
14     burnBooks()
15     visitBookStore()
```

Now, That's More Like It!

Demonstrating the evolution of the script to a more readable and maintainable version.

```
1 s = checksc()
2 if s:
3     a = cb()
4     b = cfrsp()
5     if a < 5:
6         if b > 5:
7             a = gtbs()
8         if a > b:
9             ubx()
10 else:
11     brn()
12     gtbs()
```

```
1 MAX_SHELF_SPACE = 5
2 MIN_BOOKS_REQUIRED = 5
3
4 isAvailable = checkSchedule()
5 if isAvailable:
6     bookCount = countBooks()
7     freeShelfSpace = checkFreeShelfSpace()
8     if bookCount < MAX_SHELF_SPACE:
9         if freeShelfSpace > MIN_BOOKS_REQUIRED:
10            bookCount = visitBookStore()
11            if bookCount > freeShelfSpace:
12                useStorageBox()
13 else:
14     burnBooks()
15     visitBookStore()
```

Writing readable code

Good code reads like a book.

- When it doesn't, make sure to use comments. In Python, everything following `#` is a comment
- Prevent "smart constructions" in the code
- Re-use working code (i.e. create functions for well-defined tasks).
- Document functions using docstrings
- External documentation is also useful, but harder to maintain.

How not to comment

- Useless:

```
1 # Start program
```

- Obvious:

```
1 if (a > 5) # Check if a is greater than 5
2 ...
```

- Too much about the life:

```
1 # Well... I do not know how to explain what is going on
2 # in the snippet below. I tried to code in the night
3 # with some booze and it worked then, but now I have a
4 # strong hangover and some parameters still need to be
5 # worked out...
```

- ...

```
1 # You may think that this function is obsolete, and doesn't seem to
2 # do anything. And you would be correct. But when we remove this
3 # function for some reason the whole program crashes and we can't
4 # figure out why, so here it will stay.
```

Adding comments to our Python program

Use comments to document design and purpose
(functionality), not mechanics (implementation).

```
1 IAMFree = checkSchedule()
2 if IAMFree:
3     # Count books and amount of free space on a shelf.
4     # If minimum number of books I need is less than a
5     # shelf capacity, go shopping and buy additional
6     # literature. If the amount of books after the
7     # shopping is too big, use boxes to store them.
8     books = countBooks()
9     shelfSize = countFreeSpaceShelf()
10
11 ...
12
13 else:
14     burnBooks()
15     goToBookStore()
```

What else makes a good program?

- Portability (guaranteed in Python)
- Readability
- Efficiency
- Structural
- Flexibility
- Generality
- Documentation

Funny thing is: This list does not mention that the program should be actually working for its intended purposes!

Readability

Don't use meaningless variable or function names. Rule of thumb: use verbs for functions and nouns for variables.

```
1 # stupid names
2 x = 5;
3 xx = myfunction(x);
4
5 # proper names
6 number_dams = 6;
7 beaver_workforce = allocate_beavers();
8 dams = build_dams(beaver_workforce, number_dams);
```

Efficiency

This one is difficult. Not much you can do without truly understanding how Python is utilizing your processor and memory. A couple of guidelines though:

- Avoid loops
- Especially avoid nested loops
- Use inherent matrix operations when possible
- Reduce IO (i.e. reading / writing to and from files)
- Don't run scripts from network disks
- Pre-allocate your arrays, that means, making it as large as the maximum required size for your particular problem
- Use Python's built-in modules for performance profiling, such as cProfile or timeit, to test the execution times

Efficiency: Measuring execution time example

```
1 import numpy as np
2 import time
3
4 x = np.linspace(0, 10, 1000001)
5 start_time = time.time()
6
7 y = []
8 for cr in range(len(x)):
9     y.append(np.sin(2 * np.pi * x[cr]))
10
11 end_time = time.time()
12 print(f'Execution time: {end_time - start_time}
13         seconds')
```

```
1 import numpy as np
2 import time
3
4 x = np.linspace(0, 10, 1000001)
5 start_time = time.time()
6
7 y = np.sin(2 * np.pi * x)
8
9 end_time = time.time()
10 print(f'Execution time: {end_time - start_time}
11         seconds')
```

Structural

- Compartmentalize your code.
- Write functions whenever possible.
 - If you have > 15 lines of code, you can probably replace it by one or more functions.
 - In principle, it should not even matter how function works, as long as it gives the expected output.



- Put critical variables at the beginning of your program.

Structural

Write code as though it were paragraphs in a story.

```
1 from dependencies import * # importing all dependencies
2
3 # Step 0: Define variables
4 n_steps = 10000 # number of steps
5 n_walks = 1000 # number of random walk samples
6
7 # Step 1: Generate n_walks number of random walks
8 de = np.zeros((n_walks, 2))
9 for i in range(n_walks):
10     angles = get_random_angles(n_steps)
11     coord = transform_angles_to_coordinates(angles)
12     de[i, 0] = calculate_de(coord) # store de
13     de[i, 1] = de[i, 0]**2 # store de^2
14
15 # Step 2: Plot the histogram
16 plt.hist(de[:, 0], density=True)
17 D, P = calculate_pdf(n_steps, 1000)
18 plt.plot(D, P)
19 plt.show()
```

Flexibility

If you want to add a feature or change something inside the program, it should not require rewriting the whole program. (jargon: non-linear propagation of change).

Solution: Encapsulate your code and "Don't Repeat Yourself"

- Use functions for specific tasks (can you verbalize it? Then it is probably a function)
- Use variables, even for constants (it sounds like a oxymoron, but it's not! In fact, constant variables are a real thing)
- Use abstraction whenever possible

Generalization

If your code works for one problem, it should also work for a similar problem, whether it's in another company or on another planet.

Pro-tip: Separate data from algorithms.

To be honest, I rarely see students making this mistake.

```
1 # Stupid code
2 A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Hardcode data in the program
3
4 # Smart code
5 with open('some_random_dataset.dat') as file:
6     data = file.read() # or use appropriate data loading functions from libraries like pandas
```

Documentation

Properly document your code. Write your comments in a clear and concise fashion. Help your future self: Write clear and concise documentation.

```
1 def f(a, b=None):
2     if b is not None:
3         c = a + b
4     elif b is None and a is not None:
5         c = a + a
6     else:
7         c = 0
8     return c
```



Documentation

Properly document your code. Write your comments in a clear and concise fashion. Help your future self: Write clear and concise documentation.

```
1 def f(a, b=None):
2     """
3         Add two values together.
4
5     Parameters:
6         a: The first number.
7         b: The second number. Optional.
8
9     Returns:
10        The sum of a and b, or 2*a if b is not given, or 0 if a
11        is not given.
12
13    See also: sum, operator.add
14    """
15    if b is not None:
16        c = a + b # sum two different numbers
17    elif b is None and a is not None:
18        c = a + a # double single number
19    else:
20        c = 0 # input not valid
21    return c
```



Make a habit of the following adage

MAKE IT WORK
MAKE IT RIGHT
MAKE IT FAST

Make a habit of the following adage

① *Make it work*

Create an algorithm that does the intended job. Make sure it works, and works repeatedly. Test and verify frequently. Add *todo* comments when you're not sure about a certain decision.

② *Make it right*

Refactor the code to improve the code design. Insert functions, comments, compartmentalize it. Get rid of magic numbers, use sensible variable names. Check input. Test and verify. Align with the team!

③ *Make it fast*

Measure and tune the performance of your code (profiling tool). In Python, vectorized calculations are much (!) faster than for-loops. Use sensible numerical techniques (e.g. higher-order integration).

Program by iterating over these aspects multiple times, starting at the fine-grained level, working your way up.

Today's outline

● Scientific computing

- Introduction
- Introduction to NumPy
- Math with NumPy
- Array operations

● Plotting with Matplotlib

- Line plots
- Different plot styles

● IO

● Coding style

- Program design

● Debugging and profiling

● Concluding remarks

Errors in computer programs

The following symptoms can be distinguished:

- Unable to execute the program
- Program crashes, warnings or error messages
- Never-ending loops
- Wrong (unexpected) result

Three error categories:

Syntax errors You did not obey the language rules. These errors prevent running or compilation of the program.

Runtime errors Something goes wrong during the execution of the program resulting in an error message
(problem with input, division by zero, loading of non-existent files, memory problems, etc.)

Semantic errors The program does not do what you expect, but does what have told it to do.

Validation

- Testcases: run the program with parameters such that a known result is (should be) produced.
- Testcases: what happens when unforeseen input is encountered?
 - More or fewer arguments than anticipated? (Python uses `*args` and `**kwargs` to create a varying number of input arguments, and to check the number of given input arguments)
 - Other data types than anticipated? How does the program handle this? Warnings, error messages (crash), NaN or worse: a program that silently continues?
- For physical modeling, we typically look for analytical solutions
 - Sometimes somewhat stylized cases
 - Possible solutions include Fourier-series
 - Experimental data

But: validation can only tell you *if* something is wrong, not *where* it went wrong.

The debugger (1)

- No-one can write a 1000-line code without making errors
 - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.
- Actually, you are the detective, the murderer and the victim at the same time.

"When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

— A. Conan Doyle, The Sign of Four

The debugger (2)

A debugger can help you to:

- Pause a program at a certain line: set a *breakpoint*
- Check the values of variables during the program
- Controlled execution of the program:
 - One line at a time
 - Run until a certain line
 - Run until a certain condition is met (conditional breakpoint)
 - Run until the current function exits
- Note: You may end up in the source code of Python functions!
- Check Canvas (Python Crash Course section) for a demonstration of the debugger.

Recursive Fibonacci

- Create a program that computes the n -th Fibonacci number using recursion:
$$F_n = F_{n-1} + F_{n-2}$$
 with $F_1 = 1$ and $F_2 = 1$

```
1 def fibonacci_recursive(N):
2     """
3         Prints out the Nth Fibonacci number to the screen.
4         SYNTAX: fibonacci_recursive(N)
5     """
6     if N > 2:
7         Nminus1 = fibonacci_recursive(N-1)
8         Nminus2 = fibonacci_recursive(N-2)
9         out = Nminus1 + Nminus2
10    elif N == 1 or N == 2:
11        out = 1
12    else:
13        raise ValueError('Input argument was invalid')
14    return out
```

- Place a breakpoint line 5 (click on dash or press **F12**), run `fibonacci_recursive(5)`
- Explore the function of step **F10**, step into **F11**, and how the local workspace changes
- Stop the debugger (red stop button on top, or **Shift** + **F5**)
- Right-click the breakpoint, select *Set/modify condition*, enter `N==2`, run again.

Today's outline

● Scientific computing

- Introduction
- Introduction to NumPy
- Math with NumPy
- Array operations

● Plotting with Matplotlib

- Line plots
- Different plot styles

● IO

● Coding style

- Program design

● Debugging and profiling

● Concluding remarks

Advanced concepts

- Object oriented programming: classes and objects
- Memory management: some programming languages require you to allocate computer memory yourself (e.g. for arrays)
- External libraries: in many cases, someone already built the general functionality you are looking for
- Compiling and scripting (“interpreted”); compiling means converting a program to computer-language before execution. Interpreted languages do this on the fly.
- Parallelization: Distributing expensive calculations over multiple processors or GPUs.

Make a habit of the following adage

MAKE IT WORK
MAKE IT RIGHT
MAKE IT FAST

Make a habit of the following adage

① *Make it work*

Create an algorithm that does the intended job. Make sure it works, and works repeatedly. Test and verify frequently. Add *todo* comments when you're not sure about a certain decision.

② *Make it right*

Refactor the code to improve the code design. Insert functions, comments, compartmentalize it. Get rid of magic numbers, use sensible variable names. Check input. Test and verify. Align with the team!

③ *Make it fast*

Measure and tune the performance of your code (profiling tool). In Python, vectorized calculations are much (!) faster than for-loops. Use sensible numerical techniques (e.g. higher-order integration).

Program by iterating over these aspects multiple times, starting at the fine-grained level, working your way up.

Numerical errors in computer simulations

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

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	0.1
A2	= $(A1*10)-0.9$
A3	= $(A2*10)-0.9$
A4	= $(A3*10)-0.9$

(repeat until A30)

What's happening?

Enter:

Cell	Value
A1	2
A2	= $(A1*10)-18$
A3	= $(A2*10)-18$
A4	= $(A3*10)-18$

(repeat until A30)

Example 2

Start Python

Investigate the result of `np.sin(1e40 * np.pi)`

Create a vector `v` containing the powers of 10, e.g., from 10^0 up to 10^{40} , and solve `np.sin(v * np.pi)`:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 v = np.logspace(0, 40, 41)
5 y = np.sin(v * np.pi)
6 plt.loglog(v, np.abs(y)) # Double log plot, values on y-axis must be positive.
7 plt.show()
```

Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters
- Roundoff- and truncation errors
- Break errors

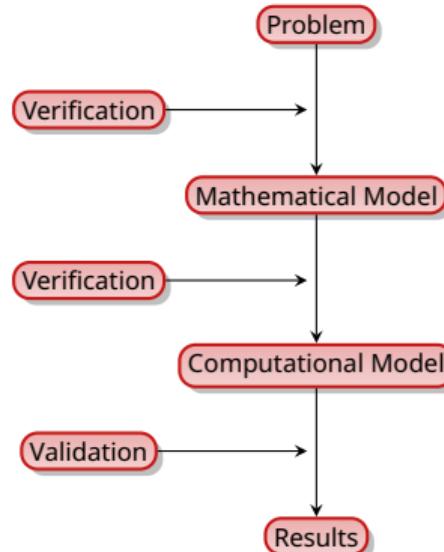
Verification and validation

Verification

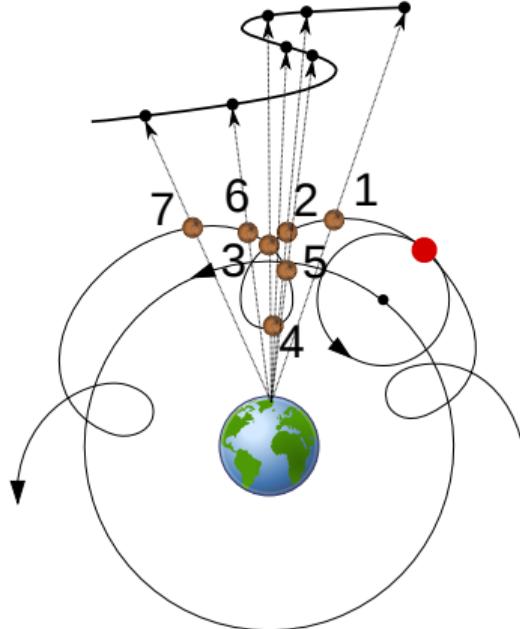
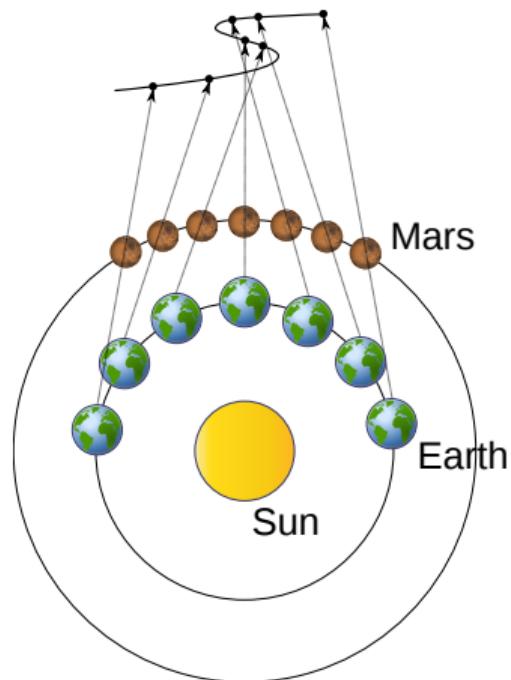
Verification is the process of mathematically and computationally assuring that the model computes the equations you intended to implement.

Validation

Validation is the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model



Verification of the physical model



- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)
- Even though they were not 'right', Earth-centered models (Ptolemy) were still valid

Be aware of your uncertainties

Aleatory uncertainty

Uncertainty that arises due to inherent randomness of the system, features that are too complex to measure and take into account

Epistemic uncertainty

Uncertainty that arises due to lack of knowledge of the system, but could in principle be known

Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters
- Roundoff- and truncation errors
- Break errors

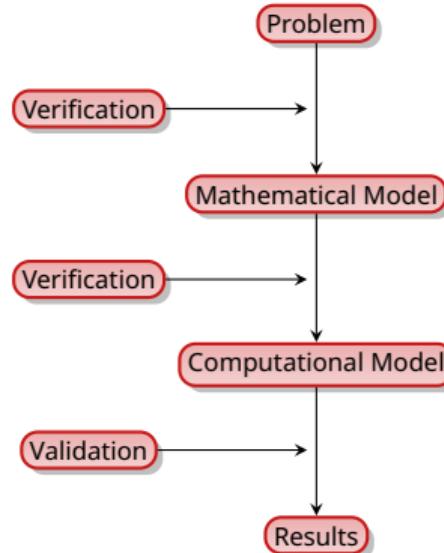
Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes the equations you intended to implement.

Validation

Validation is the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model



Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters
- Roundoff- and truncation errors
- Break errors

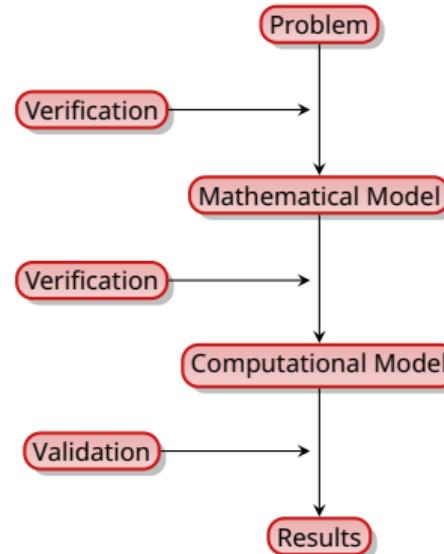
Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes the equations you intended to implement.

Validation

Validation is the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model



Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters
- Roundoff- and truncation errors
- Break errors

Significant digits

A numerical result \tilde{x} is an approximation of the real value x .

- Absolute error

$$\delta = |\tilde{x} - x|, x \neq 0$$

- Relative error

$$\frac{\delta}{\tilde{x}} = \left| \frac{\tilde{x} - x}{\tilde{x}} \right|$$

- Error margin

$$\tilde{x} - \delta \leq x \leq \tilde{x} + \delta$$

$$x = \tilde{x} \pm \delta$$

Significant digits

- \tilde{x} has m significant digits if the absolute error in x is smaller or equal to 5 at the $(m + 1)$ -th position:

$$10^{q-1} \leq |\tilde{x}| \leq 10^q$$

$$|x - \tilde{x}| = 0.5 \times 10^{q-m}$$

- For example:

$$x = \frac{1}{3}, \tilde{x} = 0.333 \Rightarrow \delta = 0.0003333\dots$$

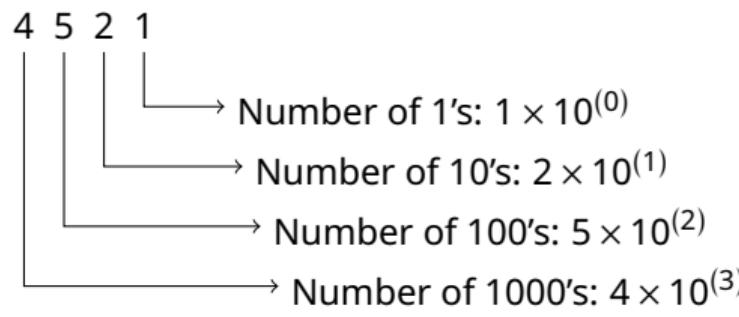
3 significant digits

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.
- In the decimal system, a digit c at position n has a value of $c \times 10^{n-1}$



$$(4521)_{10} = 4 \times 10^3 + 5 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$\begin{aligned}(4521)_{10} &= 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^9 + 1 \times 2^8 + \dots \\ &\quad \dots 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + \dots \\ &\quad \dots 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= (1000110101001)_2\end{aligned}$$

- In general:

$$(c_m \dots c_1 c_0)_q = c_0 q^0 + c_1 q^1 + \dots + c_m q^m, c \in \{0, 1, 2, \dots, q-1\}$$

Representation of numbers

- Numbers are stored in binary in the memory of a computer, in segments of a specific length (called a *word*).
- We distinguish multiple types of numbers:
 - Integers: $-301, -1, 0, 1, 96, 2293, \dots$
 - Floating points: $-301.01, 0.01, 3.14159265, 14498.2$
- A binary integer representation looks like the following bit sequence:

$$z = \sigma(c_0 2^0 + c_1 2^1 + \dots + c_{\lambda-1} 2^{\lambda-1})$$

σ is the sign of z (+ or -), and λ is the length of the word

- Endianness: the order of bits stored by a computer

Excercise

- Convert the following decimal number to base-2: 214

$$214_{10} = 11010110_2$$

- Excel:
 - Decimal: `=DEC2BIN(214)`
 - Octal: `=DEC2OCT(214)`
 - Hexadecimal: `=DEC2HEX(214)`
- Python:
 - Decimal: `bin(214)`
 - Octal: `oct(214)`
 - Hexadecimal: `hex(214)`

Arithmetic operations with binary numbers

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0 \text{ (carry one)}$$

$$\begin{array}{r} 1 & 4 & 5 \\ + & 2 & 3 \\ \hline 1 & 6 & 8 \end{array}$$

$$\begin{array}{r} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Addition:

$$0 - 0 = 0$$

$$1 - 0 = 1$$

$$1 - 1 = 0$$

$$0 - 1 = 1 \text{ (borrow one)}$$

$$\begin{array}{r} 1 & 4 & 5 \\ - & 2 & 3 \\ \hline 1 & 2 & 2 \end{array}$$

$$\begin{array}{r} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ - & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ \hline 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{array}$$

Subtraction:

- Multiplication and division are more expensive, and more elaborate

Exercise

Try the following commands in Python:

Command	Result
<code>np.iinfo(np.int32).min</code>	<code>-2147483648</code>
<code>np.iinfo(np.int32).max</code>	<code>2147483647</code>
<code>i = np.int16(np.iinfo(np.int32).max)</code>	<code>i = 32767</code>
<code>type(i)</code>	<code><class 'numpy.int16'></code>
<code>i = i + 100</code>	<code>i = 32767</code>
<code>np.finfo(np.float64).max</code>	<code>1.7976931348623157e+308</code>
<code>f = 0.1</code>	
<code>type(f)</code>	<code><class 'float'></code>
<code>print(":16f".format(f))</code>	<code>0.1000000000000000</code>
<code>print(":20f".format(f))</code>	<code>0.1000000000000000555</code>

Representation of integer numbers

- In Python, integers are typically represented with at least 32 bits (though this can be larger depending on the Python implementation and system architecture). For a typical 64-bit Python installation, the integer type would have 64 bits.
- The set of numbers that a 32-bit integer 'int32' can represent in numpy is:

$$-2^{31} \leq z \leq 2^{31} - 1 \approx 2 \times 10^9$$

where z is a variable representing an `np.int32` type value.

- If, during a calculation, an integer number in Python exceeds its maximum limit, Python automatically promotes it to a larger type if possible. In the case of numpy, an overflow error will be thrown if a number exceeds the maximum value representable by the '`np.int32`' type.
- How can a computer identify an overflow? Usually through error reporting mechanisms within the language. Python's numpy library, for instance, will throw an `OverflowError` when an operation results in a number larger than what can be represented by the current data type.

Representation of real (floating point) numbers

- Formally, a real number is represented by the following bit sequence

$$x = \sigma \left(2^{-1} + c_2 2^{-2} + \dots + c_m 2^{-m} \right) 2^{e-1023}$$

Here, σ is the sign of x and e is an integer value.

- A floating point number hence contains sections that contain the sign, the exponent and the mantissa

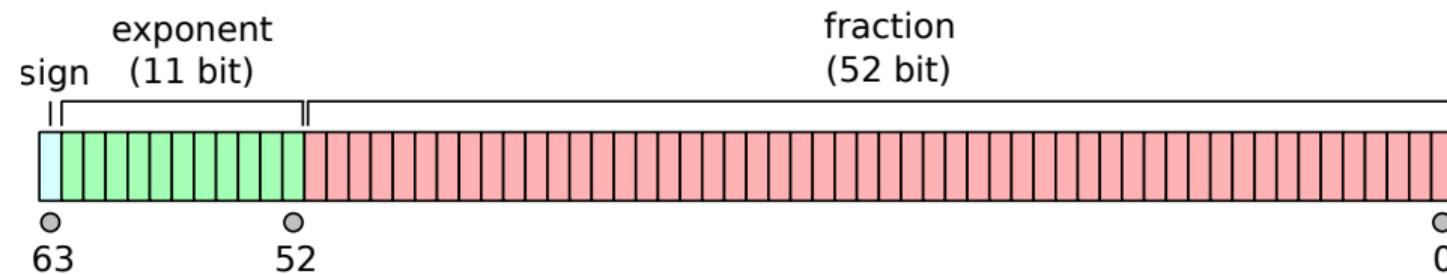


Image: Wikimedia Commons CC by-SA

Representation of real (floating point) numbers

- Example: $\lambda = 3, m = 2, x = \frac{2}{3}$

$$x = \pm(2^{-1} + c_2 2^{-2})2^e$$

- $c_0 \in \{0, 1\}$
- $e = \pm a_0 2^0$
- $a_0 \in \{0, 1\}$
- Truncation: $f_l(x) = 2^{-1} = 0.5$
- Round off: $f_l(x) = 2^{-1} + 2^{-2} = 0.75$

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Trigonometric, Logarithmic, and Exponential computations

- Processors can do logic and arithmetic instructions
- Trigonometric, logarithmic and exponential calculations are “higher-level” functions:
 \exp , \sin , \cos , \tan , \sec , \arcsin , \arccos , \arctan , \log , \ln , ...
- Such functions can be performed using these “low level” instructions, for instance using a Taylor series:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Trigonometric, Logarithmic, and Exponential computations

- These operations involve many multiplications and additions, and are therefore *expensive*
- Computations can only take finite time, for infinite series, calculations are interrupted at N

$$\sin(x) = \sum_{n=0}^N \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + \frac{(-1)^N}{(2N+1)!} x^{2N+1}$$

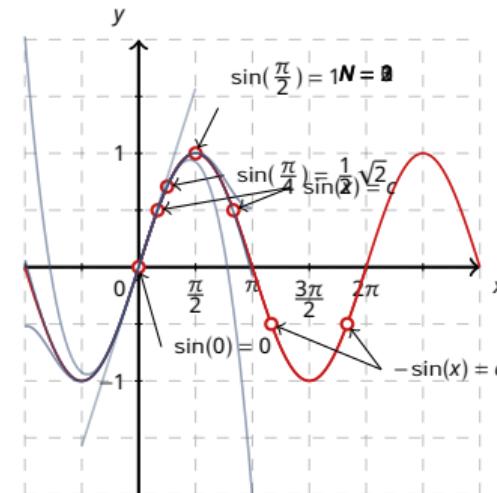
$$e^x = \sum_{n=0}^N \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots + \frac{x^N}{N!}$$

- This results in a *break error*

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

- ① Use periodicity so that $0 \leq x \leq 2\pi$
- ② Use symmetry ($0 \leq x \leq \frac{\pi}{2}$)
- ③ Use lookup tables for known values
- ④ Perform taylor expansion



Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Loss of digits

- During operations such as $+$, $-$, \times , \div , an error can add up
- Consider the summation of x and y

$$\tilde{x} - \delta \leq x \leq \tilde{x} + \delta \quad \text{and} \quad \tilde{y} - \varepsilon \leq y \leq \tilde{y} + \varepsilon$$

$$(\tilde{x} + \tilde{y}) - (\delta + \varepsilon) \leq x + y \leq (\tilde{x} + \tilde{y}) + (\delta + \varepsilon)$$

Loss of digits: Example 1

$$\left. \begin{array}{l} x = \pi, \tilde{x} = 3.1416 \\ y = 22/7, \tilde{y} = 3.1429 \end{array} \right\} \Rightarrow \left. \begin{array}{l} \delta = \tilde{x} - x = 7.35 \times 10^{-6} \\ \varepsilon = \tilde{y} - y = 4.29 \times 10^{-5} \end{array} \right\}$$

$$x + y = \tilde{x} + \tilde{y} \pm (\delta + \varepsilon) \approx 6.2845 - 5.025 \times 10^{-5}$$

$$x - y = \tilde{x} - \tilde{y} \pm (\delta + \varepsilon) \approx -0.0013 + 3.55 \times 10^{-5}$$

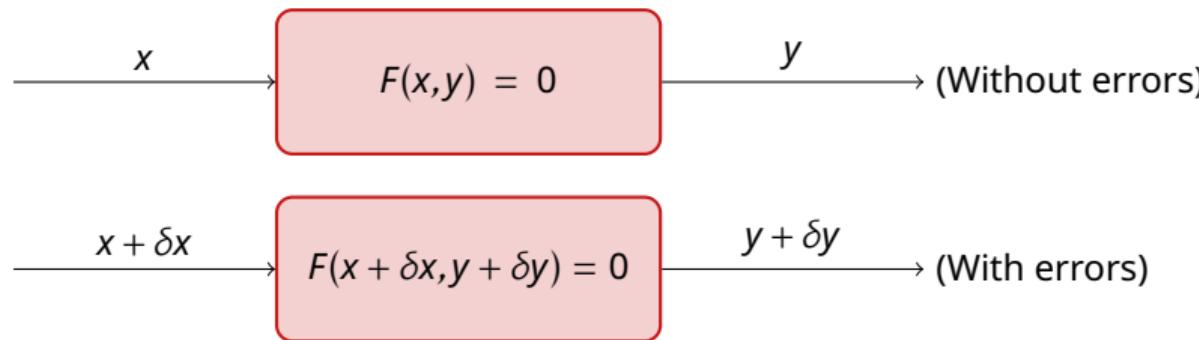
- The absolute error is small ($\approx 10^{-5}$), but the relative error is much bigger (0.028).
- Adding up the errors results in a loss of significant digits!

Loss of digits: Example 2

- Calculate e^{-5}
 - Use the Taylor series
 - Calculate the first 26 terms ($N = 26$)
- Now repeat the calculation, but use for each calculation only 4 digits. What do you find?
Use: `round(term, 4)`
- Without errors you would find: $e^{-5} = 0.006738$
- If you only use 4 digits in the calculations, you'll find 0.00998

Badly (ill) conditioned problems

We consider a system $F(x, y)$ that computes a solution from input data. The input data may have errors:



$$y(x + \delta x) - y(x) \approx y'(x)\delta x$$

Propagated error on the basis of Taylor expansion

$$C = \max_{\delta x} \left(\left| \frac{\delta y / y}{\delta x / x} \right| \right)$$

Condition criterion, $C < 10$ error development small

Badly (ill) conditioned problems: Example

Solve the following linear system in Python using double and single precision:

$$A = \begin{bmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{bmatrix}, \quad x = \begin{bmatrix} 0.8642 \\ 0.1440 \end{bmatrix}, \quad y = \begin{bmatrix} 2.0 \\ -2.0 \end{bmatrix}$$

Double precision

```

1 import numpy as np
2
3 # Double precision
4 A = np.array([[1.2969, 0.8648], [0.2161,
   0.1441]], dtype=np.float64)
5 x = np.array([0.8642, 0.1440], dtype=np.float64)
6 y = np.linalg.solve(A, x)
7 print("y =")
8 print(y)

```

Single precision

```

1 import numpy as np
2
3 # Single precision
4 A = np.array([[1.2969, 0.8648], [0.2161,
   0.1441]], dtype=np.float32)
5 x = np.array([0.8642, 0.1440], dtype=np.float32)
6 y = np.linalg.solve(A, x)
7 print("y =")
8 print(y)

```

Badly (ill) conditioned problems: Example

- Python does not automatically warn about bad condition number. You need to compute and check it manually using NumPy:
`if np.linalg.cond(A)> threshold: raise("Ill conditioned error")`
- The `cond` number is the condition number computed using NumPy.
- A small error in A (the matrix) results in a big error in the solution. This is called an ill-conditioned problem.

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

(Un)stable methods

- The condition criterion does not tell you anything about the quality of a numerical solution method!
- It is very well possible that a certain solution method is more sensitive for one problem than another
- If the method propagates the error, we call it an *unstable method*. Let's look at an example.

The Golden mean

The Golden Mean is a well known ratio and one of the solutions of $\phi^2 = \phi + 1$, such that $\phi = \frac{1+\sqrt{5}}{2}$, and $\phi - 1 = \phi^{-1}$. Many relations to compute ϕ have been established (which pose often an interesting source for creating small test programs).

- Let's evaluate the following recurrent relationship:

$$y_{n+1} = y_{n-1} - y_n$$

$$y_0 = 1, \quad y_1 = 1 - \phi = \phi^{-1} = \frac{2}{1 + \sqrt{5}}$$

- Alternatively, a closed form power law relation exists, which again computes y_n :

$$y_n = x^{-n}, \quad n = 0, 1, 2, \dots, \quad x = \frac{1 + \sqrt{5}}{2}$$

The Golden mean

Recurrent version

```

1 import numpy as np
2
3 def golden_mean_recurrent(Ntot):
4     # Initialize the series with the given
5         # initial conditions
6     y = np.zeros(Ntot)
7     y[0] = 1
8     y[1] = 2 / (1 + np.sqrt(5))
9
10    # Perform the recurrence to fill in the rest
11        # of the series
12    for n in range(2, Ntot):
13        y[n] = y[n-1] - y[n-2]
14    return y

```

Power law version

```

1 import numpy as np
2
3 def golden_mean_powerlaw(Ntot):
4     # Initialize the constant value
5     x = (1 + np.sqrt(5)) / 2
6
7     # Generate a range of values from 0 to Ntot
8         # and apply the power law
9     y = x ** -np.arange(0, Ntot + 1)
10
11    return y

```

- Compare the outcomes: `plot(goldenMeanPowerlaw(40)- goldenMeanRecurrent(40))`
- See what happens if you use single precision (uncomment the second line of both functions).

The Golden mean

n	Recurrent	Power law
0	1.0000	1.0000
1	0.6180	0.6180
2	0.3820	0.3820
3	0.2361	0.2361
...
37	$1.714 \cdot 10^{-8}$	$1.851 \cdot 10^{-8}$
38	$1.366 \cdot 10^{-8}$	$1.144 \cdot 10^{-8}$
39	$3.485 \cdot 10^{-8}$	$7.071 \cdot 10^{-9}$
40	$1.017 \cdot 10^{-8}$	$4.370 \cdot 10^{-9}$

- The recurrent approach enlarges errors from earlier calculations!

Example 1: Explanation

Recall example 1, where the errors blew up our computation of 0.1, whereas they did not for 2. Why did we see these results?

- The number 0.1 is not exactly represented in binary
 - A tiny error can accumulate up to catastrophic proportions!
- The number 2 does have an exact binary representation

Example 2 (large sine series)

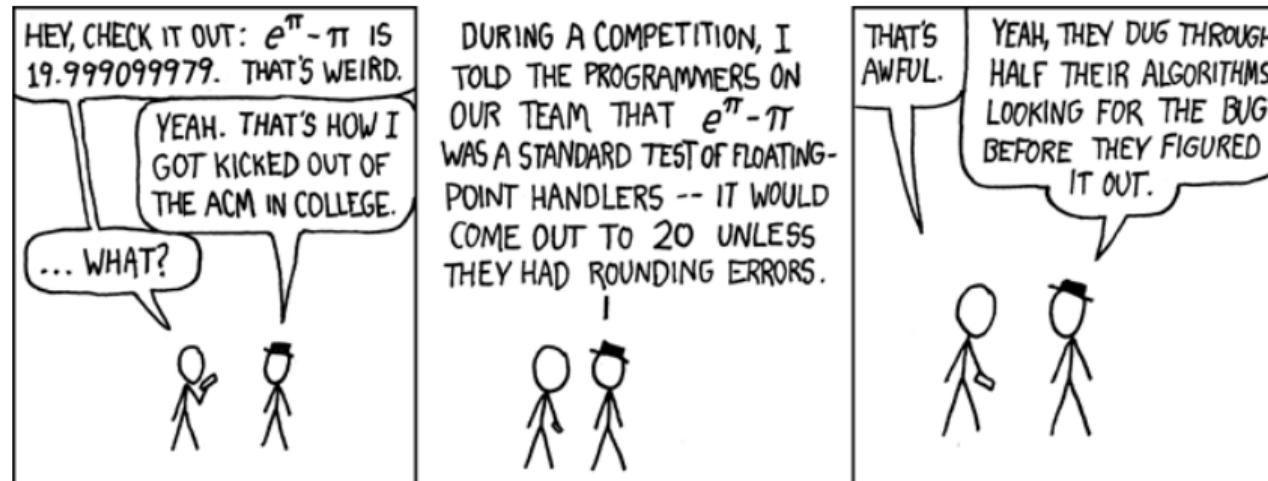
The `np.sin(1e40*np.pi)` result gives poor results because $1e40$ has an error margin on the order of floating-point machine epsilon, which is roughly 1×10^{-16} in Python (double-precision floating-point format). In Python, as in many computing environments, the number of $2 \cdot \pi$ cycles is still much larger than $10^{40} \cdot 10^{-16}$. Also, π is not stored with an infinite number of digits, which further contributes to the imprecision.

Example 3

Start your calculation program of choice (Excel, Python, ...)

Calculate the result of y :

$$y = e^\pi - \pi = 19.999099979 \neq 20$$



Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Symbolic math packages

Definition

The use of computers to manipulate mathematical equations and expressions in symbolic form, as opposed to manipulating the numerical quantities represented by those symbols.

- Symbolic integration or differentiation, substitution of one expression into another
- Simplification of an expression, change of subject etc.
- Packages and toolboxes:

Symbolic math packages

Mathematica Well known software package, license available via [TU/e](#)

Maple Well known, license available via [TU/e](#)

Wolfram|Alpha Web-based interface by Mathematica developer. Less powerful in mathematical respect, but more accessible and has a broad application range (unit conversion, semantic commands).

Sage Open-source alternative to Maple, Mathematica, Magma, and MATLAB.

Matlab Symbolic math toolbox

Python SymPy

Symbolic math: simplify

$$f(x) = (x - 1)(x + 1)(x^2 + 1) + 1$$

```
1 from sympy import symbols, simplify
2
3 x = symbols('x') # Alt: from sympy.abc import x,y,z
4 f = (x - 1)*(x + 1)*(x**2 + 1) + 1
5 f_simplified = simplify(f)
6 print(f_simplified)
```

`f_simplified = x**4`

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

```
1 from sympy import symbols, simplify, integrate, diff
2
3 x = symbols('x')
4 f = 1/(x**3+1)
5 my_f_int = integrate(f, x)
6 my_f_diff = diff(my_f_int, x)
7 my_f_diff_simplified = simplify(my_f_diff)
8 print(my_f_diff_simplified)
```

```
my_f_diff_simplified = 1/(x**3 + 1)
```

Symbolic math: exercises

Exercise 1

Simplify the following expression:

$$f(x) = \frac{2 \tan x}{(1 + \tan^2 x)} = \sin 2x$$

```
1 from sympy import symbols, trigsimp, tan
2
3 x = symbols('x')
4 expr = 2*tan(x)/(1 + tan(x)**2)
5 simplified_expr = trigsimp(expr)
```

Symbolic math: exercises

Exercise 2

Calculate the *value* of p :

$$p = \int_0^{10} \frac{e^x - e^{-x}}{\sinh x} dx$$

```
1 from sympy import exp, sinh, integrate, symbols
2
3 x = symbols("x")
4 f = ((exp(x)- exp(-x))/sinh(x)).simplify()
5 p = integrate(f, (x, 0, 10))
6 print(p)
```

$p = 20$

Symbolic math: root finding

A root finding method searches for the values where a function reaches zero. We will cover the numerical methods later, here we show how to use root finding with symbolic math in Python.

Symbolic math function

$$f(x) = \frac{3}{x^2 + 3x} - 2$$

```
1 from sympy import solve, symbols
2
3 x = symbols("x")
4 f = 3 / (x**2 + 3*x) - 2
5 solutions = solve(f, x)
6 print(solutions)
```

solutions = [-3/2 + sqrt(15)/2, -sqrt(15)/2 - 3/2]

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Summary

- Numerical errors may arise due to truncation, roundoff and break errors, which may seriously affect the accuracy of your solution
- Errors may propagate and accumulate, leading to smaller accuracy
- Ill-conditioned problems and unstable methods have to be identified so that proper measures can be taken
- Symbolic math computations may be performed to solve certain equations algebraically, bypassing numerical errors, but this is not always possible.

Linear equations 1

Linear algebra basics

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

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

Overview

Goals

- Different ways of looking at a system of linear equations
- Determination of the inverse, determinant and the rank of a matrix
- The existence of a solution to a set of linear equations

Different views of linear systems

- Separate equations:

$$x + y + z = 4$$

$$2x + y + 3z = 7$$

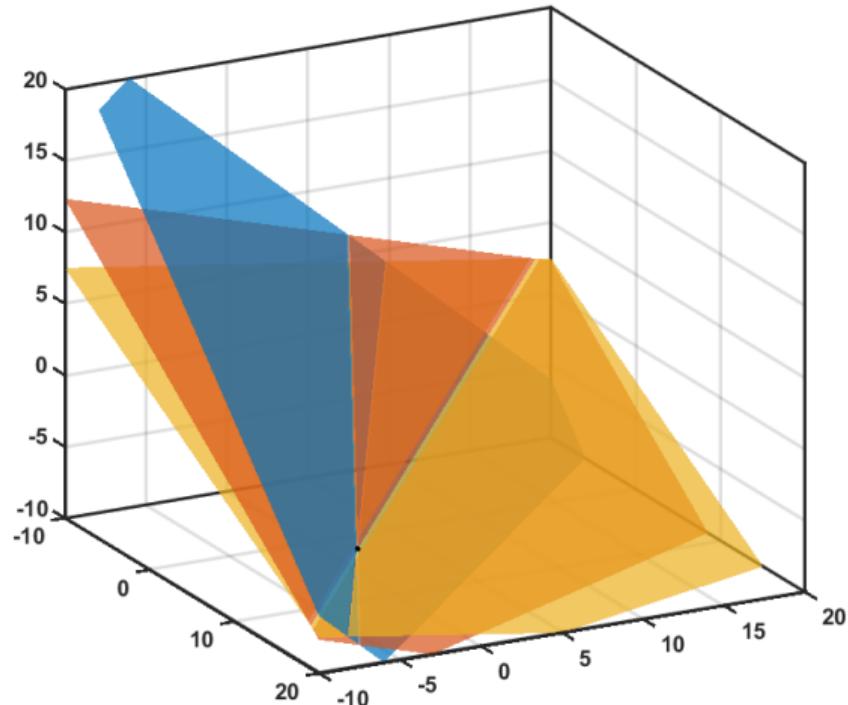
$$3x + y + 6z = 5$$

- Matrix mapping $Mx = b$:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- Linear combination:

$$x \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + y \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + z \begin{bmatrix} 3 \\ 6 \\ 6 \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$



Today's outline

- Introduction
- Matrix inversion

- Solving a linear system
- Towards larger systems
- Summary

Inverse of a matrix

- The inverse M^{-1} is defined such that:

$$MM^{-1} = I \quad \text{and} \quad M^{-1}M = I$$

- Use the inverse to solve a set of linear equations:

$$M\mathbf{x} = \mathbf{b}$$

$$M^{-1}M\mathbf{x} = M^{-1}\mathbf{b}$$

$$I\mathbf{x} = M^{-1}\mathbf{b}$$

$$\mathbf{x} = M^{-1}\mathbf{b}$$

How to calculate the inverse?

- The inverse of an $N \times N$ matrix can be calculated using the co-factors of each element of the matrix:

$$M^{-1} = \frac{1}{\det|M|} \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix}^T$$

- $\det|M|$ is the *determinant* of matrix M .
- C_{ij} is the *co-factor* of the ij^{th} element in M .

Computing the co-factors

Consider the following example matrix: $M = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}$

A co-factor (e.g. C_{11}) is the determinant of the elements left over when you cover up the row and column of the element in question, multiplied by ± 1 , depending on the position.

$$\begin{bmatrix} 1 & \times & \times \\ \times & 1 & 3 \\ \times & 1 & 6 \end{bmatrix}$$

$$\begin{bmatrix} + & - & + \\ - & + & - \\ + & - & + \end{bmatrix}$$

$$\begin{aligned} C_{11} &= +1 \cdot \det \begin{vmatrix} 1 & 3 \\ 1 & 6 \end{vmatrix} \\ &= 6 \times 1 - 3 \times 1 = 3 \end{aligned}$$

Computing the co-factors

Back to our example:

$$M^{-1} = \frac{1}{\det|M|} \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}^{-1} = \frac{1}{\det|M|} \begin{bmatrix} 3 & -3 & -1 \\ -5 & 3 & 2 \\ 2 & -1 & -1 \end{bmatrix}^T$$

- The determinant is very important
- If $\det|M| = 0$, the inverse does not exist (singular matrix)

Calculating the determinant

Compute the determinant by multiplication of each element on a row (or column) by its cofactor and adding the results:

$$\det \begin{vmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{vmatrix} = +\det \begin{vmatrix} 1 & 3 \\ 1 & 6 \end{vmatrix} - \det \begin{vmatrix} 2 & 3 \\ 3 & 6 \end{vmatrix} + \det \begin{vmatrix} 2 & 1 \\ 3 & 1 \end{vmatrix} = -1$$

$$\det \begin{vmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{vmatrix} = +\det \begin{vmatrix} 2 & 1 \\ 3 & 1 \end{vmatrix} - 3\det \begin{vmatrix} 1 & 1 \\ 3 & 1 \end{vmatrix} + 6\det \begin{vmatrix} 1 & 1 \\ 2 & 1 \end{vmatrix} = -1$$

Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

Solving a linear system

- Our example:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- The solution is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M^{-1}b = \frac{1}{-1} \begin{bmatrix} 3 & -5 & 2 \\ -3 & 3 & -1 \\ -1 & 2 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix} = \frac{1}{-1} \begin{bmatrix} -13 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 13 \\ -4 \\ -5 \end{bmatrix}$$

- The inverse exists, because $\det|M| = -1$.

Solving a linear system in Python using the inverse

- Create the matrix:

```
1 >>> A = np.array([[1, 1, 1], [2, 1, 3], [3, 1, 6]])
```

- Create solution vector:

```
1 >>> b = np.array([4, 7, 5])
```

- Get the matrix inverse:

```
1 >>> Ainv = np.linalg.inv(A)
```

- Compute the solution:

```
1 >>> x = np.dot(Ainv, b)
```

- Python's internal direct solver:

```
1 >>> x = np.linalg.solve(A, b)
```

- These are black boxes! We are going over some methods later!

Exercise: performance of inverse computation

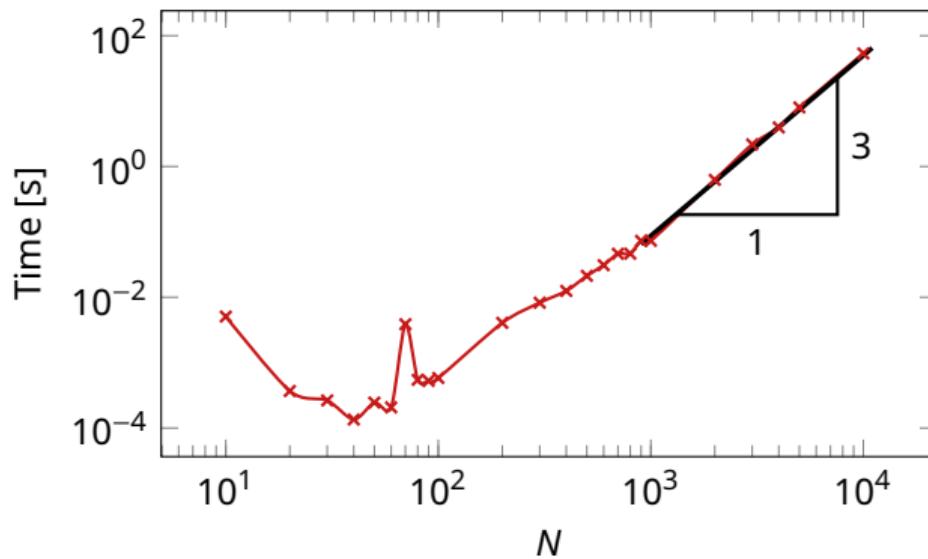
Create a script that generates matrices with random elements of various sizes $N \times N$ (e.g. values of $N \in \{10, 20, 50, 100, 200, \dots, 5000, 10000\}$). Compute the inverse of each matrix, and use `tic` and `toc` to see the computing time for each inversion. Plot the time as a function of the matrix size N .

Hints:

- Create an array that contains the sizes of the systems n
- Loop over the array elements to:
 - Create a random matrix of size $n \times n$
 - Perform the matrix inversion
 - Record the time required
- Plot the time required for inversion vs size of the system on a double-log scale

Exercise: sample results

Each computer produces slightly different results because of background tasks, different matrices, etc. This is especially noticeable for small systems.



The time increases by 3 orders of magnitude, for every magnitude in N . The *computational complexity* of matrix inversion scales with $\mathcal{O}(N^3)$!

Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

Towards larger systems

Computation of determinants and inverses of large matrices in this way is too difficult (slow), so we need other methods to solve large linear systems!

Towards larger systems

- Determinant of upper triangular matrix:

$$\det|M_{\text{tri}}| = \prod_{i=1}^n a_{ii} \quad M = \begin{bmatrix} 5 & 3 & 2 \\ 0 & 9 & 1 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow \det|M| = 5 \times 9 \times 1 = 45$$

- Matrix multiplication:

$$\det|AM| = \det|A| \times \det|M|$$

- When A is an identity matrix ($\det|A| = 1$):

$$\det|AM| = \det|A| \times \det|M| = 1 \times \det|M|$$

- With rules like this, we can use row-operations so that we can compute the determinant more cheaply.

Solutions of linear systems

Rank of a matrix: the number of linearly independent columns (columns that can not be expressed as a linear combination of the other columns) of a matrix.

$$M = \begin{bmatrix} 5 & 3 & 2 \\ 0 & 9 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- 3 independent columns
- In Python:

```
1 >>> numpy.linalg.matrix_rank(M)
```

- col 2 = 2× col 1
- col 4 = col 3 – col 1
- 2 independent columns: rank = 2

Solutions of linear systems

The solution of a system of linear equations may or may not exist, and it may or may not be unique. Existence of solutions can be determined by comparing the rank of the Matrix M with the rank of the augmented matrix M_a :

```
1 >>> numpy.linalg.matrix_rank(A)
2 >>> numpy.linalg.matrix_rank(np.column_stack((A,b))) # Concatenated matrices
```

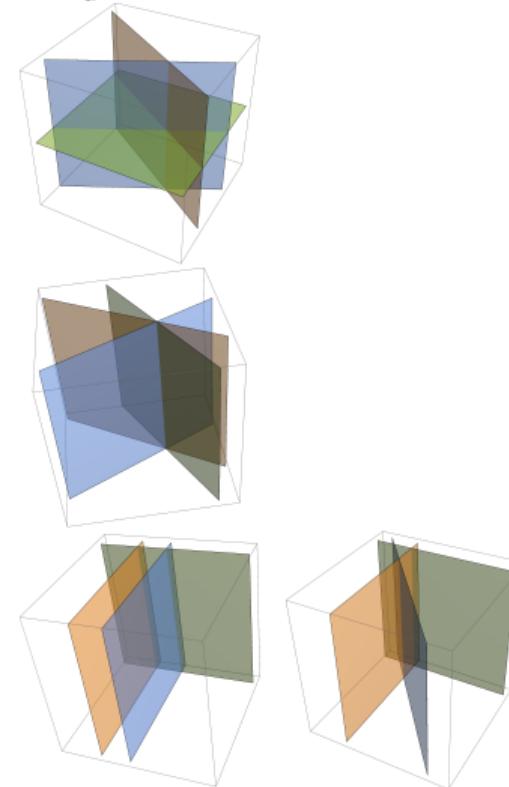
Our system: $Mx = b$

$$M = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} M_{11} & M_{12} & M_{13} & b_1 \\ M_{21} & M_{22} & M_{23} & b_2 \\ M_{31} & M_{32} & M_{33} & b_3 \end{bmatrix}$$

Existence of solutions for linear systems

For a matrix M of size $n \times n$, and augmented matrix M_a :

- $\text{Rank}(M) = n$:
Unique solution
- $\text{Rank}(M) = \text{Rank}(M_a) < n$:
Infinite number of solutions
- $\text{Rank}(M) < n$, $\text{Rank}(M) < \text{Rank}(M_a)$:
No solutions



Two examples

$$M = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 17 \\ 11 \\ 4 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} 1 & 1 & 2 & 17 \\ 0 & 3 & 1 & 11 \\ 0 & 0 & 2 & 4 \end{bmatrix}$$

$\text{rank}(M) = 3 = n \Rightarrow$ Unique solution

$$M = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad b = \begin{bmatrix} 17 \\ 11 \\ 0 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} 1 & 1 & 2 & 17 \\ 0 & 3 & 1 & 11 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$\text{rank}(M) = \text{rank}(M_a) = 2 < n \Rightarrow$ Infinite number of solutions

Today's outline

● Introduction

● Matrix inversion

● Solving a linear system

● Towards larger systems

● Summary

Summary

- Linear equations can be written as matrices
- Using the inverse, the solution can be determined
 - Inverse via cofactors
 - Inverse and solution in Python
- Introduced the concept of computational complexity: matrix inversion scales with N^3
- A solution depends on the rank of a matrix

Linear equations 2

Direct methods

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

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

Overview

Goals

Today we are going to write a program, which can solve a set of linear equations

- The first method is called Gaussian elimination
- We will encounter some problems with Gaussian elimination
- Then LU decomposition will be introduced

Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

Define the linear system

Consider the system:

$$Ax = b$$

In general:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

Desired solution:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \end{bmatrix}$$

Using row operations

- Use row operations to simplify the system. Eliminate element A_{10} by subtracting $A_{10}/A_{00} = d_{10}$ times row 1 from row 2.
- In this case, Row 1 is the pivot row, and A_{00} is the pivot element.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ A_{10} & A_{11} & A_{12} & b_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array} \right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array} \right]$$

Using row operations

Eliminate element A_{10} using $d_{10} = A_{10}/A_{00}$.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ A_{10} & A_{11} & A_{12} & b_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array} \right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array} \right]$$

- $d_{10} \rightarrow A_{10}/A_{00}$
- $A_{10} \rightarrow A_{10} - A_{00}d_{10}$
- $A_{11} \rightarrow A_{11} - A_{01}d_{10}$
- $A_{12} \rightarrow A_{12} - A_{02}d_{10}$
- $b_1 \rightarrow b_1 - b_0d_{10}$

```
1 d10 = A[1,0] / A[0,0]
2
3 A[1,0] = A[1,0] - A[0,0] * d10
4 A[1,1] = A[1,1] - A[0,1] * d10
5 A[1,2] = A[1,2] - A[0,2] * d10
6
7 b[1] = b[1] - b[0] * d10
```

Using row operations

Eliminate element A_{20} using $d_{20} = A_{20}/A_{00}$.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ A_{20} & A_{21} & A_{22} & b_2 \end{array} \right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & A'_{21} & A'_{22} & b'_2 \end{array} \right]$$

- $d_{20} \rightarrow A_{20}/A_{00}$
- $A_{20} \rightarrow A_{20} - A_{00}d_{20}$
- $A_{21} \rightarrow A_{21} - A_{01}d_{20}$
- $A_{22} \rightarrow A_{22} - A_{02}d_{20}$
- $b_2 \rightarrow b_2 - b_0d_{20}$

```
1 d20 = A[2, 0] / A[0, 0]
2
3 A[2, 0] = A[2, 0] - A[0, 0] * d20
4 A[2, 1] = A[2, 1] - A[0, 1] * d20
5 A[2, 2] = A[2, 2] - A[0, 2] * d20
6 b[2] = b[2] - b[0] * d20
```

Using row operations

Eliminate element A'_{21} using $d_{21} = A'_{21}/A'_{11}$. Note that now the second row has become the pivot row.

$$\left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & A'_{21} & A'_{22} & b'_2 \end{array} \right] \longrightarrow \left[\begin{array}{ccc|c} A_{00} & A_{01} & A_{02} & b_0 \\ 0 & A'_{11} & A'_{12} & b'_1 \\ 0 & 0 & A''_{22} & b''_2 \end{array} \right]$$

- $d_{21} \rightarrow A_{21}/A'_{11}$
- $A_{21} \rightarrow A_{21} - A'_{11}d_{21}$
- $A_{22} \rightarrow A_{22} - A'_{12}d_{21}$
- $b_2 \rightarrow b_2 - b'_2d_{21}$

```
1 d21 = A[2, 1] / A[1, 1]
2 A[2, 1] = A[2, 1] - A[1, 1] * d21
3 A[2, 2] = A[2, 2] - A[1, 2] * d21
4 b[2] = b[2] - b[1] * d21
```

The matrix is now a triangular matrix — the solution can be obtained by back-substitution.

Backsubstitution

The system now reads:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ 0 & A'_{11} & A'_{12} \\ 0 & 0 & A''_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b'_1 \\ b''_2 \end{bmatrix}$$

Start at the last row N , and work upward until row 1.

$$x_2 = b''_2 / A''_{22}$$

$$x_1 = (b'_1 - A'_{12}x_2) / A'_{11}$$

$$x_0 = (b_0 - A_{01}x_1 - A_{02}x_2) / A_{00}$$

```
1 x = np.empty_like(b)
2 x[2] = b[2] / A[2,2]
3 x[1] = (b[1] - A[1,2] * x[2]) / A[1,1]
4 x[0] = (b[0] - A[0,1] * x[1] - A[0,2] * x[2]) / A[0,0]
```

In general:

$$x_N = \frac{b_N}{A_{NN}} \quad x_i = \frac{b_i - \sum_{j=i+1}^N A_{ij}x_j}{A_{ii}}$$

Writing the program

- Create a function that provides the framework: take matrix A and vector b as an input, and return the solution x as output:

```
1 def gaussian_eliminate(A, b):
2     pass # Your implementation here
```

- We will use *for-loops* instead of typing out each command line.
- Useful Python (with NumPy) shortcuts:
 - $A[0, :] = [A_{00}, A_{01}, A_{02}]$
 - $A[:, 1] = [A_{01}, A_{11}, A_{21}]$
 - $A[0, 1:] = [A_{01}, A_{02}]$
- A row operation could look like:

```
1 A[i, :] = A[i, :] - d * A[0, :]
```

The program: elimination step

An initial draft could look like:

```
1 def gaussian_eliminate_draft(A,b):
2     """Perform elimination to obtain an upper triangular matrix"""
3     A = np.array(A,dtype=np.float64)
4     b = np.array(b,dtype=np.float64)
5
6     assert A.shape[0] == A.shape[1], "Coefficient matrix should be square"
7
8     N = len(b)
9     for col in range(N-1): # Select pivot
10        for row in range(col+1,N): # Loop over rows below pivot
11            d = A[row,col] / A[col,col] # Choose elimination factor
12            for element in range(row,N): # Elements from diagonal to right
13                A[row,element] = A[row,element] - d * A[col,element]
14            b[row] = b[row] - d * b[col]
15
16    return A,b
```

The program: elimination step

Employing some of the row operations to create gaussian_eliminate_v1:

```
1 for element in range(row,N):  
2     A[row,element] = A[row,element] - d * A[col,element]
```

```
1 A[row,:] = A[row,:] - d * A[col,:]
```

```
1 def gaussian_eliminate_v1(A,b):  
2     A = np.array(A,dtype=np.float64)  
3     b = np.array(b,dtype=np.float64)  
4  
5     assert A.shape[0] == A.shape[1], "Coefficient matrix should be square"  
6  
7     N = len(b)  
8     for col in range(N-1):  
9         for row in range(col+1,N):  
10            d = A[row,col] / A[col,col]  
11            A[row,:] = A[row,:] - d * A[col,:]  
12            b[row] = b[row] - d * b[col]  
13  
14     return A,b
```

Testing

Let's try to eliminate our linear system! If you create/downloaded our file `gaussjordan.py`, you can access the functions by importing them. The file should be stored where your own code/notebook is:

```
1 from gaussjordan import gaussian_eliminate_draft,gaussian_eliminate_v1
2 import numpy as np
3
4 A = np.array([[1, 1, 1], [2, 1, 3], [3, 1, 6]])
5 b = np.array([4, 7, 5])
6
7 Aprime,bprime = gaussian_eliminate_draft(A,b)
8 print(Aprime)
9 print(bprime)
```

The program: Backsubstitution

Now we have elimination working, let's create a back substitution algorithm too. Recall:

$$x_N = \frac{b_N}{A_{NN}} \quad x_i = \frac{b_i - \sum_{j=i+1}^N A_{ij}x_j}{A_{ii}}$$

```
1 def backsubstitution_draft(A, b):
2     """Back substitutes an upper triangular matrix to
3         find x in Ax=b"""
4     x = np.copy(b)
5     N = len(b)
6
7     for row in range(N-1, -1, -1):
8         for i in range(row+1, N):
9             x[row] = x[row] - A[row, i] * x[i]
10            x[row] = x[row] / A[row, row]
11
12    return x
```

```
1 def backsubstitution_v1(A,b):
2     """Back substitutes an upper triangular matrix to find x in Ax=b"""
3     x = np.empty_like(b)
4     N = len(b)
5
6     for row in range(N)[::-1]:
7         x[row] = (b[row] - np.sum(A[row, row+1:] * x[row+1:])) / A[row, row]
8
9     return x
```

A full Gauss Elimination solver

- The functions we just built are distributed via Canvas too
- Use `help GaussianEliminate` to find out how it works
- Solve the following system of equations:

$$\begin{bmatrix} 9 & 9 & 5 & 2 \\ 6 & 7 & 1 & 3 \\ 6 & 4 & 3 & 5 \\ 2 & 6 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \\ 10 \\ 1 \end{bmatrix}$$

- Compare your solution with `np.linalg.solve(A, b)`

Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

Partial pivoting

- Now try to run the algorithm with the following system:

$$\begin{bmatrix} 0 & 2 & 1 \\ 3 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 10 \end{bmatrix}$$

- It does not work! Division by zero, due to $A_{11} = 0$.
- Solution: Swap rows to move largest element to the diagonal.

Partial pivoting: implementing row swaps

- Find maximum element row below pivot in current column

```
index = np.argmax(np.abs(A[:, col])) + col
```

- Store current row

```
temp = A[column, :]
```

- Swap pivot row and desired row in A

```
A[column, :] = A[index, :]
A[index, :] = temp
```

- Do the same for b — store and swap

```
temp = b[column]
b[column] = b[index]
b[index] = temp
```

Adding the partial pivoting rules

```
1 def gaussian_eliminate_partial_pivot(A,b):
2     A = np.array(A,dtype=np.float64)
3     b = np.array(b,dtype=np.float64)
4
5     assert A.shape[0] == A.shape[1], "Coefficient matrix should be square"
6
7     N = len(b)
8     for col in range(N-1):
9         index = np.argmax(np.abs(A[col:, col])) + col
10        temp = A[col,:]
11        A[col,:] = A[index,:]
12        A[index,:] = temp
13
14        temp = b[col]
15        b[col] = b[index]
16        b[index] = temp
17        for row in range(col+1,N):
18            d = A[row,col] / A[col,col]
19            A[row,:] = A[row,:] - d * A[col,:]
20            b[row] = b[row] - d * b[col]
21
22    return A,b
```

Improve the program by using re-usable functions

```
1 def swap_rows(mat,i1,i2):
2     """Swap two rows in a matrix/vector"""
3     temp = mat[i1,...].copy()
4     mat[i1,...] = mat[i2,...]
5     mat[i2,...] = temp
```

```
1 def gaussian_eliminate_v2(A,b):
2     A = np.array(A,dtype=np.float64)
3     b = np.array(b,dtype=np.float64)
4
5     assert A.shape[0] == A.shape[1], "Coefficient matrix should be square"
6
7     N = len(b)
8     for col in range(N-1):
9         index = np.argmax(np.abs(A[:, col])) + col
10        swap_rows(A,col,index)
11        swap_rows(b,col,index)
12        for row in range(col+1,N):
13            d = A[row,col] / A[col,col]
14            A[row,:] = A[row,:] - d * A[col,:]
15            b[row] = b[row] - d * b[col]
16
17    return A,b
```

Alternatives to this program

- Python can compute the solution to $Ax=b$ with `scipy.linalg.solve` OR `numpy.linalg.solve` solvers (more efficient)
- Too many loops. Loops make Python slow.
- There are fundamental problems with Gaussian elimination
 - You can add a counter to the algorithm to see how many subtraction and multiplication operations it performs for a given size of matrix A.
 - The number of operations to perform Gaussian elimination is $\mathcal{O}(2N^3)$ (where N is the number of equations)
 - Exercise: verify this for our script
 - LU decomposition takes $\mathcal{O}(2N^3/3)$ flops, 3 times less!
 - Forward and backward substitution each take $\mathcal{O}(N^2)$ flops (both cases)

Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

LU Decomposition

Suppose we want to solve the previous set of equations, but with several right hand sides:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots \\ x_1 & x_2 & x_3 \\ \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ b_1 & b_2 & b_3 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

Factor the matrix A into two matrices, L and U, such that $A = LU$:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \times & 1 & 0 \\ \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix}$$

Now we can solve for each right hand side, using only a forward followed by a backward substitution!

Substitutions

- Define a lower and upper matrix L and U so that $A = LU$
- Therefore $LUX = b$
- Define a new vector $y = UX$ so that $Ly = b$
- Solve for y , use L and forward substitution
- Then we have y , solve for x , use $UX = y$
- Solve for x , use U and backward substitution
- But how to get L and U?

Decomposing the matrix (1)

When we eliminate the element A_{21} we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to A .

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A_{22} - d_{21}A_{12} & A_{23} - d_{21}A_{13} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Decomposing the matrix (2)

When we eliminate the element A_{31} we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to A .

$$A = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} = A_{22} - d_{21}A_{12} & A'_{23} = A_{23} - d_{21}A_{13} \\ 0 & A'_{32} = A_{32} - d_{31}A_{12} & A'_{33} = A_{33} - d_{31}A_{11} \end{bmatrix}$$

Decomposing the matrix (3)

When we eliminate the element A_{32} we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to A .

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & d_{32} & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A''_{33} = A'_{33} - d_{32}A'_{23} \end{bmatrix}$$

We now have a lower matrix L and an upper matrix U . This finishes the LU decomposition!

Pivoting during decomposition

Suppose we have arrived at the situation below, where $A'_{32} > A'_{22}$:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A'_{22} & A'_{23} \\ A'_{32} & A'_{33} \end{bmatrix}$$

Exchange rows 2 and 3 to get the largest value on the main diagonal. Use a permutation matrix to store the swapped rows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{31} & 0 & 1 \\ d_{21} & 1 & 0 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A'_{33} \end{bmatrix}$$

Multiplying with a permutation matrix will swap the rows of a matrix. The permutation matrix is just an identity matrix, whose rows have been interchanged.

Recipe for LU decomposition

When decomposing matrix A into $A = LU$, it may be beneficial to swap rows to get the largest values on the diagonal of U (pivoting). A permutation matrix P is used to store row swapping such that:

$$PA = LU$$

- Write down a permutation matrix and the linear system
- Promote the largest value in the column diagonal
- Eliminate all elements below diagonal
- Move on to the next column and move largest elements to diagonal
- Eliminate elements below diagonal
- Repeat 5 and 6
- Write down L,U and P

LU decomposition example (1)

Write down a permutation matrix, which starts as the identity matrix, and the linear system:

$$PA = LU$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

Promote the largest value into the diagonal of column 1 — swap row 1 and 2:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

LU decomposition example (2)

Eliminate all elements below the diagonal — row 2 already contains a zero in column 1, row 3 = row 3 - 0.5 row 1. Record the multiplier 0.5 in L :

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1.5 & -0.5 \end{bmatrix}$$

Elimination of column 1 is done. Now step to the next column, and move the largest value before the pivot element to the top of the column. This is called pivoting. This will result in the lower triangle of L accordingly:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1.5 & -0.5 \\ 0 & 1 & 1 \end{bmatrix}$$

LU decomposition example (3)

Eliminate all elements below the diagonal —

row 3 = row 3 - $\frac{2}{3}$ row 2. Record the multiplier $\frac{2}{3}$ in L:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & \frac{2}{3} & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1.5 & -0.5 \\ 0 & 0 & \frac{4}{3} \end{bmatrix}$$

We have obtained the matrices from $PA = LU$:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & \frac{2}{3} & 1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1.5 & -0.5 \\ 0 & 0 & \frac{4}{3} \end{bmatrix}$$

Proceed with solving for x.

Substitutions

$$Ax = b \Rightarrow PAx = Pb \equiv d$$

$$PA = LU \Rightarrow LUx = d$$

- Define a new vector $y = Ux$
 - $Ly = b \Rightarrow Ly = d$
 - Solve for y , forward substitution:

$$y_0 = \frac{d_0}{L_{00}}$$

$$y_i = \frac{d_i - \sum_{j=0}^{i-1} L_{ij}y_j}{L_{ii}}$$

- Then solve $Ux = y$:
 - Solve for x , backward substitution:

$$x_N = \frac{y_N}{U_{NN}}$$

$$x_i = \frac{y_i - \sum_{j=i+1}^N U_{ij}x_j}{U_{ii}}$$

How to use the solver in Python

```
1 import numpy as np
2 from scipy.linalg import lu
3 from gaussjordan import backsubstitution_v1 as backwardSub
4 from gaussjordan import forwardsubstitution as forwardSub
5
6 # Example usage
7 A = np.random.rand(5, 5) # Get random matrix
8 P, L, U = lu(A) # Get L, U and P
9 b = np.random.rand(5) # Random b vector
10 d = P @ b # Permute b vector
11 y = forwardSub(L, d) # Can also do y=L\d
12 x = backwardSub(U, y) # Can also do x=U\y
13 rnorm = np.linalg.norm(A @ x - b) # Residual
```

- Use this as a basis to create a function that takes A and b , and returns x .
- Use the function to check the performance for various matrix sizes and inspect the performance.

Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

Summary

- This lecture covered direct methods using elimination techniques.
- Gaussian elimination can be slow ($\mathcal{O}(N^3)$)
- Back substitution is often faster ($\mathcal{O}(N^2)$)
- LU decomposition means that we don't have to do Gaussian elimination every time (saves time and effort), but the matrix has to stay the same.
- Python's libraries have built in routines for solving linear equations and LU decomposition.
- Advanced techniques such as (preconditioned) conjugate gradient or biconjugate gradient solvers are also available.
- Next part covers iterative approaches

Linear equations 3

Iterative methods

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

- Introduction
 - Sparse matrices
 - Laplace's equation
 - Creating a sparse system
 - Iterative methods
 - Summary

Sparse matrices

- In many engineering cases, we deal with sparse matrices (as opposed to dense matrices)
- A matrix is sparse when it mostly consists of zeros
- Linear systems where equations depend on a limited number of variables (e.g. spatial discretization)
- Storing zeros is not very efficient:

```
1 import numpy as np
2 from scipy.sparse import csr_matrix
3
4 A = np.eye(10000)
5 print(A.nbytes)
6
7 S = csr_matrix(A)
8 print(S.data nbytes)
```

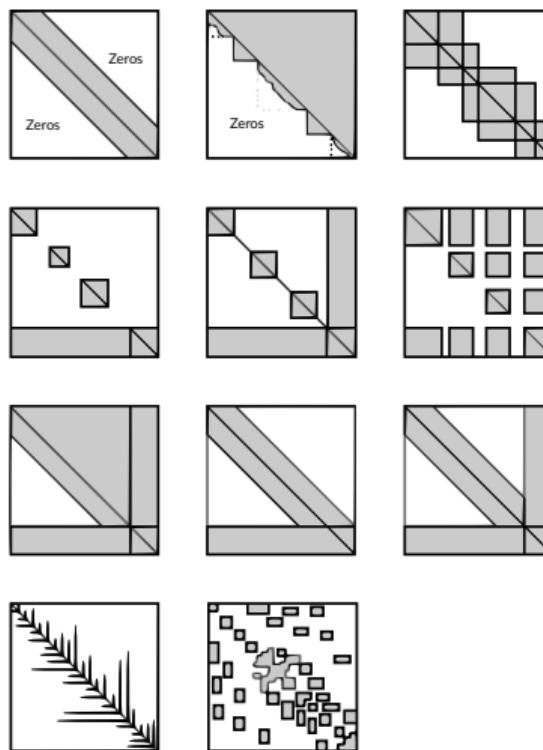
- Can you think of a way to achieve this?
- Sparse matrix formats: Yale, CRS, CCS

Sparse matrix storage format

- Example: Yale storage format, storing 3 vectors:
 - $A = [5 \ 8 \ 3 \ 6]$
 - $IA = [0 \ 1 \ 2 \ 3 \ 4]$
 - $JA = [0 \ 1 \ 2 \ 1]$
- A stores the non-zero values
- IA stores the index in A of the first non-zero in row i
- JA stores the column index

$$A = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{bmatrix}$$

Sparse matrix layout examples



Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

Laplace's equation

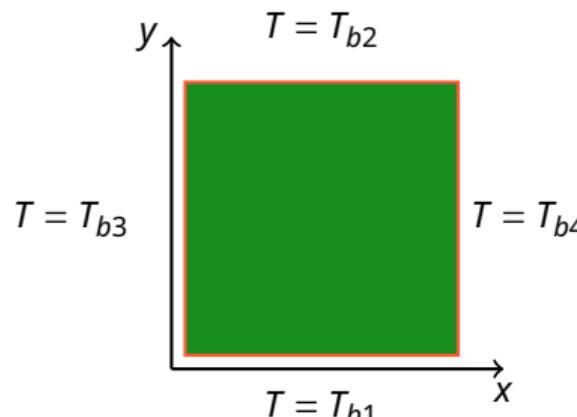
$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

T = Temperature

α = Thermal diffusivity

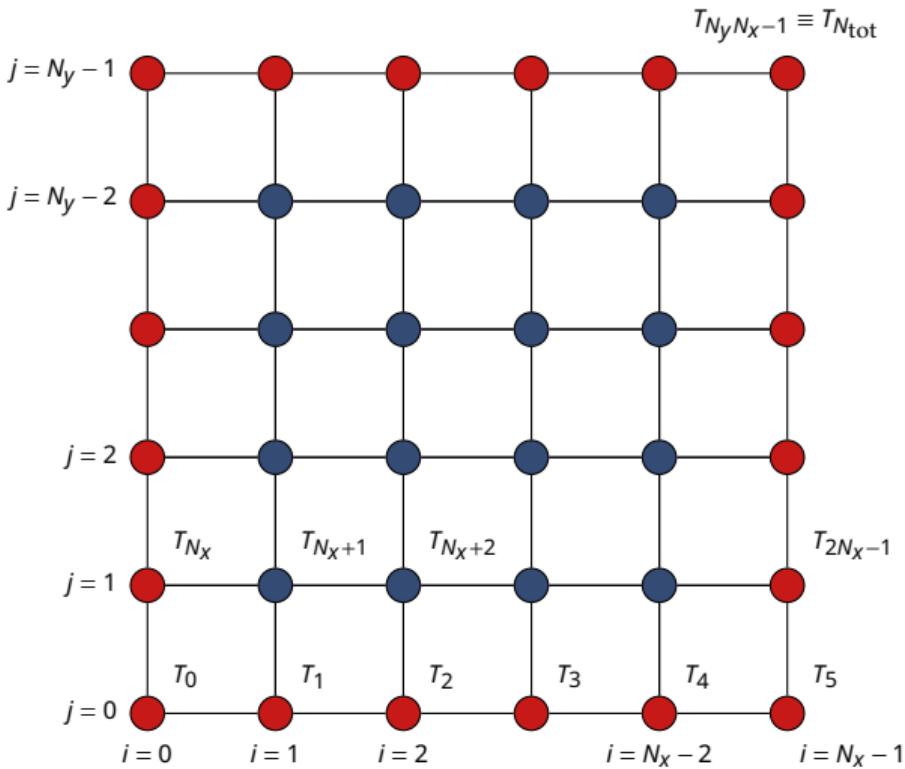
In steady state:

$$\nabla^2 T = 0$$



$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

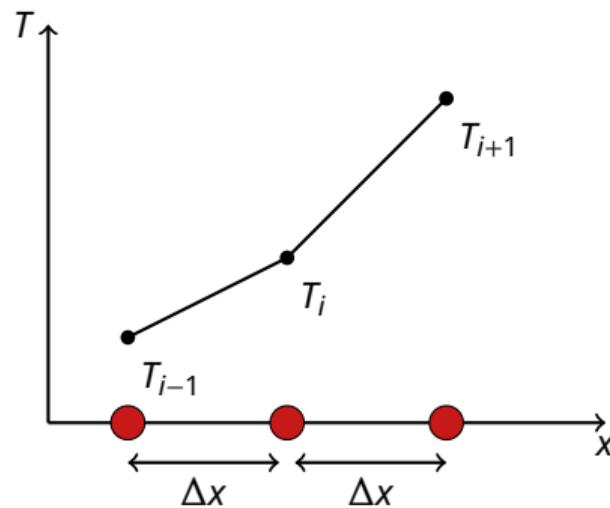
Discretization of Laplace's equation (I)



- Define a grid of points in x and y
- Index of the grid points using 2D coordinates i and j
- Set up the equations using a 1D index system:
 $T_{i,j} \rightarrow T_{i+jN_x}$

Discretization of Laplace's equation (II)

Estimate the second-order differentials: assume a piece-wise linear profile in the temperature:



$$\begin{aligned}\frac{\partial^2 T}{\partial x^2} &\approx \frac{\frac{\partial T}{\partial x}\Big|_{i+\frac{1}{2}} - \frac{\partial T}{\partial x}\Big|_{i-\frac{1}{2}}}{\Delta x} \\ &\approx \frac{\frac{(T_{i+1,j}-T_{i,j})}{\Delta x} - \frac{(T_{i,j}-T_{i-1,j})}{\Delta x}}{\Delta x} \\ &= \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2}\end{aligned}$$

Discretization of Laplace's equation (III)

The y -direction is derived analogously, so that the 2D Laplace's equation is discretized as:

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

Use a single index counter $k = i + N_x(j - 1)$, so that the equation becomes:

$$\frac{T_{k+1} - 2T_k + T_{k-1}}{(\Delta x)^2} + \frac{T_{k+N_x} - 2T_k + T_{k-N_x}}{(\Delta y)^2} = 0$$

For an equal spaced grid $\Delta x = \Delta y = 1$:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

$$\Rightarrow AT = b$$

Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

Creating the linear system

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

Create a *banded* matrix A : the main diagonal k contains -4 , whereas the bands at $k - 1, k + 1, k - N_x$ and $k + N_x$ contain a 1 . Boundary cells just contain a 1 on the main diagonal so that the temperature is equal to T_b (e.g. $T_1 = 1T_b$).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ \cdots & 1 & \cdots & 1 & -4 & 1 & \cdots & 1 & \ddots & 0 \\ 0 & \cdots & 1 & \cdots & 1 & -4 & 1 & \cdots & 1 & \vdots \\ \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ \vdots \\ T_k \\ T_{k+1} \\ \vdots \\ T_{N_y N_x - 2} \\ T_{N_y N_x - 1} \end{bmatrix} = \begin{bmatrix} T_b \\ T_b \\ \vdots \\ 0 \\ 0 \\ \vdots \\ T_b \\ T_b \end{bmatrix}$$

Creating the linear system

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

Create a *banded* matrix A in Python, by setting the coefficients for the internal cells:

```
1 import numpy as np
2 from scipy.sparse import diags
3
4 Nx,Ny = 50,50 # Number of grid points along x,y direction
5 Nc = Nx*Ny # Total number of points
6
7 e = np.ones(Nc)
8 A = diags([e, e, -4*e, e, e], [-Nx, -1, 0, 1, Nx], shape=(Nc,Nc))
9 b = np.zeros(Nc)
```

The function `diags` uses the following arguments:

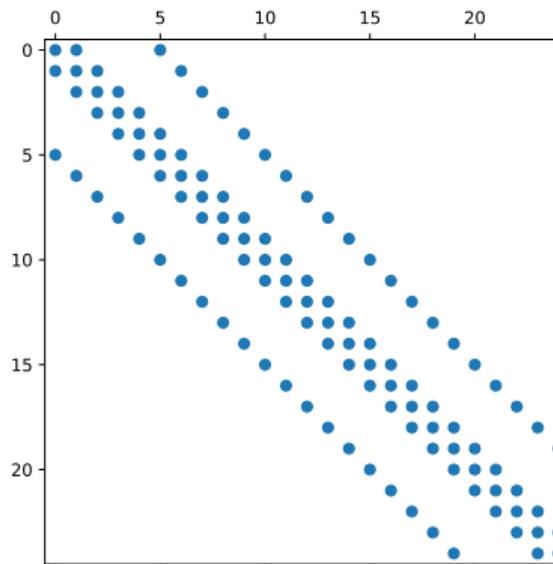
- The coefficients that have to be put on the diagonals arranged as columns in a matrix
- The position of the bands with respect to the main diagonal
- Size of the resulting matrix (in our case square $N_xN_y \times N_xN_y$)

Matrix sparsity

- Let's check the matrix layout by adding:

```
1 print(A)
2 plt.spy(A, marker='o', markersize=6)
```

- The *sparse* structure stores/prints only the nonzero elements
- `spy` shows the location of the nonzero values in the matrix
- Apart from the main diagonal, there are offset bands!



About boundary conditions

- For the nodes on the boundary, we have a simple equation:

$$T_{k,\text{boundary}} = \text{Some fixed value}$$

- However, we have set all nodes to be a function of their neighbors
- Solution: Determine the boundary node indices k and set the coefficients accordingly

```
1 bnd_bottom = np.arange(Nx)
2 bnd_left = np.arange(Ny) * Nx
3 bnd_right = bnd_left + Nx - 1
4 bnd_top = bnd_bottom + Nx*(Ny-1)
```

- Reset each row k in A to zeros, then set element $A_{kk} = 1$
- Set values in rhs: $b_k = T_{\text{boundary}}$
- Boundary conditions are often more elaborate to implement!

Implementation of the boundary conditions

A (shortened) version of the `set_boundary_conditions(A, b, Tb, Nx, Ny)` function:

```
1 def set_boundary_conditions(A, b, Tb, Nx, Ny):
2
3     A = lil_matrix(A) # Required for efficient modification of the sparsity
4
5     # Select nodes that lie at the boundaries
6     bnd_bottom = np.arange(Nx)
7     bnd_left = np.arange(Ny) * Nx
8     bnd_right = bnd_left + Nx - 1
9     bnd_top = bnd_bottom + Nx*(Ny-1)
10
11    bnd_all = np.unique(np.concatenate((bnd_bottom,bnd_left,bnd_right,bnd_top)))
12
13    # Reset the coefficient row to zero, add a 1 only on the main diagonal
14    A[bnd_all,:] = 0
15    A[bnd_all,bnd_all] = 1
16
17    b[bnd_bottom] = Tb['bottom']
18    b[bnd_left] = Tb['left']
19    b[bnd_right] = Tb['right']
20    b[bnd_top] = Tb['top']
21
22    return A.tocsr(), b
```

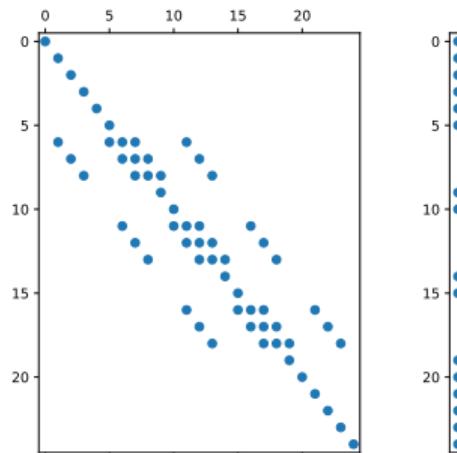
How applying boundary conditions affects the linear system

Using the functions provided in `laplace_demo.py`:

```
1 Nx = Ny = 5 # number of internal grid cells over x/y-direction
2
3 T_boundary = {'bottom': 300, 'left': 1000, 'right': 1000, 'top': 500}
4
5 A,b = create_laplace_coefficient_matrix(Nx,Ny)
6 A,b = set_boundary_conditions(A, b, T_boundary, Nx, Ny)
```

Check the new structure of the matrix and the right hand side:

```
1 plt.subplot(121); plt.spy(A2);
2 plt.subplot(122); plt.spy(b[:,None]);
```

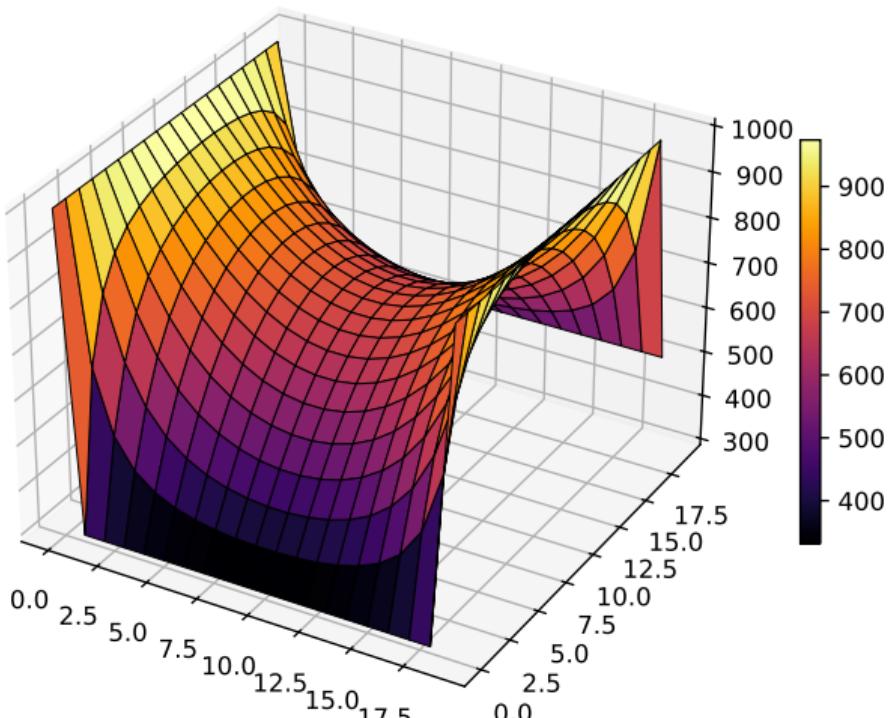


A full program, including solver

The program and auxiliary functions are on Canvas (`laplace_demo.py`)

```
1 import numpy as np
2 from scipy.sparse.linalg import spsolve
3 from matplotlib import cm
4 import matplotlib.pyplot as plt
5
6 Nx = Ny = 20
7
8 T_boundary = {'bottom': 300, 'left': 1000, 'right': 1000, 'top': 500}
9
10 A,b = create_laplace_coefficient_matrix(Nx,Ny)
11 A,b = set_boundary_conditions(A, b, T_boundary, Nx, Ny)
12
13 T = spsolve(A,b).reshape((Nx,Ny))
14
15 fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
16 x,y = np.meshgrid(np.arange(Nx),np.arange(Ny))
17 surf = ax.plot_surface(x,y,T,cmap=cm.inferno)
18 fig.colorbar(surf, shrink=0.5)
19 plt.show()
```

Sample results



Exercise: Verify the numerical solution using Fourier-series

A Fourier-series expansion for the steady-state heat conduction in a flat plate is given for a domain: $x, y \in [0, 1]$, with fixed-temperature boundaries $T|_{x=0} = T|_{x=1} = T|_{y=0} = 0$ and $T|_{y=1} = 1$:

$$T = \frac{4}{\pi} \sum_{n=1}^{\infty} \frac{\sin(m\pi x) \sinh(m\pi y)}{m \sinh(m\pi)} \quad \text{with } m = 2n - 1$$

Compute and plot the exact temperature profile in the 2D plate, and compare it with the numerical solution:

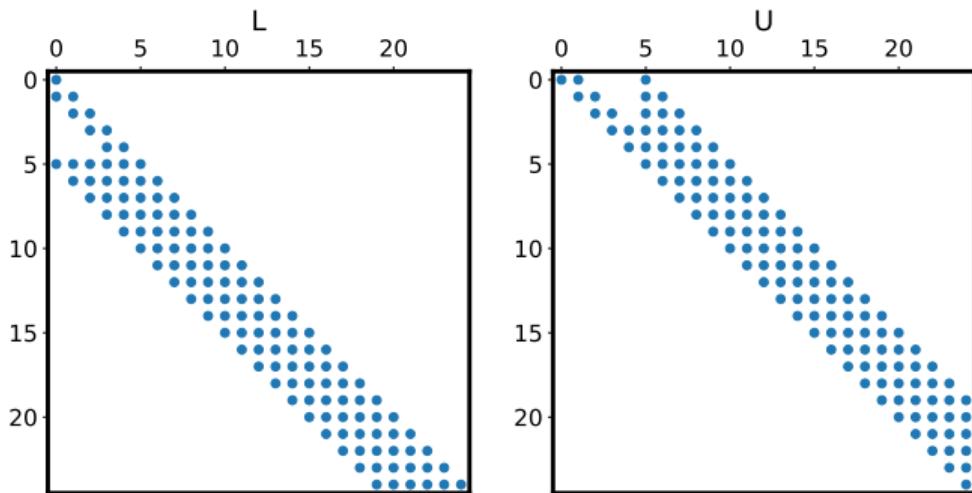
Hints:

- Use meshgrid to create a mesh in x and y
 - Compute the temperature using the Fourier series, use vectorised computations over x and y so that only 1 loop (over n) is required.
 - Solve the numerics for the same problem (note the boundary conditions)
 - Compare the numerical and exact solutions (e.g. a plot).
-

LU decomposition of a sparse matrix

```
1 import numpy as np
2 from scipy.linalg import lu
3 import matplotlib.pyplot as plt
4 from laplace_demo import
5     create_laplace_coefficient_matrix
6
7 A,b = create_laplace_coefficient_matrix(5,5)
8
9 # Perform LU decomposition
10 # Note: lu does not work on sparse arrays,
11 # so we map to a full array
12 P,L,U = lu(A.toarray())
13
14 # Plot the sparsity patterns of L and U
15 plt.subplot(121)
16 plt.spy(L)
17 plt.title('L')
18 plt.subplot(122)
19 plt.spy(U)
20 plt.title('U')
21 plt.tight_layout()
```

- With LU decomposition we produce matrices that are less sparse than the original matrix.
- Sparse storage often required, and also numerical techniques that fully utilizes this!



LU decomposition

- LU decomposition and Gaussian elimination on a matrix like A requires more memory (with 3D problems, the offset in the diagonal would even be bigger!)
- In general extra memory allocation will not be a problem for Python
- Python is clever, in that sense that it attempts to reorder equations, to move elements closer to the diagonal)

Alternatives for elimination methods

- Use iterative methods when systems are large and sparse.
- Often such systems are encountered when we want to solve PDE's of higher dimensions

Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

Examples of iterative methods

- Jacobi method
- Gauss-Seidel method
- Successive over relaxation
- `bicg` — Bi-conjugate gradient method
- `pcg` — preconditioned conjugate gradient method
- `gmres` — generalized minimum residuals method
- `bicgstab` — Bi-conjugate gradient method

The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

- In the Jacobi scheme the iteration proceeds as follows:
 - Start with an initial guess for the values of T at each node
 - Compute updated values and store a new vector:

$$T_k^{\text{new}} = \frac{T_{k-N_x}^{\text{old}} + T_{k-1}^{\text{old}} + T_{k+1}^{\text{old}} + T_{k+N_x}^{\text{old}}}{4}$$

- Do this for all nodes
- Repeat the procedure until converged

Jacobi method for Laplace's equation

See `laplace_jacobi.py` for animation included (from Canvas)

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Set grid resolution
5 nx = 40
6 ny = 40
7
8 # Set old solution array
9 T = np.zeros((nx,ny))
10
11 # Set boundary conditions
12 T[0,:] = 40 # Left
13 T[nx-1,:] = 60 # Right
14 T[:,0] = 20 # Bottom
15 T[:,ny-1] = 30 # Top
16
17 # Set new solution array (inc bnd
18 # conditions)
Tnew = T.copy()
```

```
1 # Create grid for plotting
2 x,y = np.meshgrid(np.arange(1,nx+1), np.arange(1,ny+1))
3
4 # Perform iterations
5 for iter in range(1,1001):
6     for i in range(1,nx-1):
7         for j in range(1,ny-1):
8             # Calculate new solution
9             Tnew[i,j] = \
10                 (T[i-1,j]+T[i+1,j]+T[i,j-1]+T[i,j+1])/4.0
11 T = Tnew.copy()
```

→ Try to modify this script so that 1 cell/block of cells in the center is kept at 100 degrees

About the straightforward implementation

- The method as implemented works fine for a simple Laplace equation
- For generic systems of linear equations, the implementation cannot be used.

We will now introduce the Jacobi method so it can be used for generic systems of linear equations.

The Jacobi method with matrices

We can split our (banded) matrix A into a diagonal matrix D and a remainder R :

$$A = D + R$$

$$\begin{bmatrix} \times & \times & & & \times & & \\ \times & \times & \times & & \times & & \\ & \times & \times & \times & & & \\ & \times & \times & \times & & & \\ & & \times & \times & \times & & \\ & & & \times & \times & \times & \\ & & & & \times & \times & \\ & & & & & \times & \end{bmatrix} = \begin{bmatrix} \times & & & & & & \\ & \times & & & & & \\ & & \times & & & & \\ & & & \times & & & \\ & & & & \times & & \\ & & & & & \times & \\ & & & & & & \times \end{bmatrix} + \begin{bmatrix} \times & & & & & & \\ & \times & & & & & \\ & & \times & & & & \\ & & & \times & & & \\ & & & & \times & & \\ & & & & & \times & \\ & & & & & & \times \end{bmatrix}$$

Jacobi method: solving a system

- We can solve $AT = b$, now written generally as $Ax = b$, by:

$$Ax = b$$

$$(D + R)x = b$$

$$Dx = b - Rx$$

$$Dx^{\text{new}} = b - Rx^{\text{old}}$$

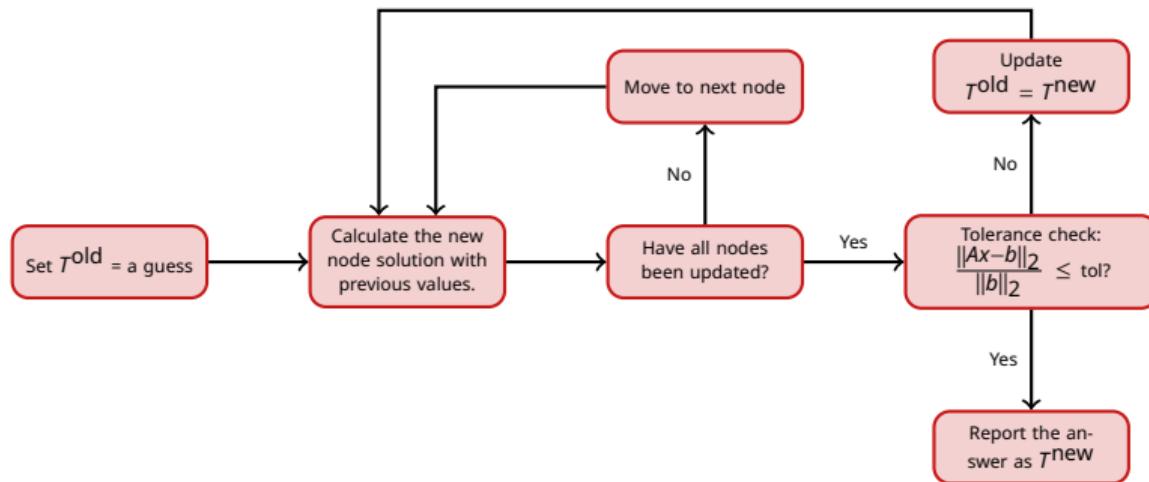
$$x^{\text{new}} = D^{-1}(b - Rx^{\text{old}})$$

- Using the n and $n + 1$ notation for old and new time steps, we find in general:

$$x^{n+1} = D^{-1}(b - Rx^n)$$

$$x_i^{n+1} = \frac{1}{A_{ii}} \left(b_i - \sum_{j \neq i} A_{ij} x_j^n \right)$$

Diagram of the Jacobi method



The core of the solver

The full function `jacobi(A, b, tol=1e-2)` is on Canvas, see `it_methods.py`. The gist is:

```
1 # While not converged or max_it not reached
2 while (x_diff > tol and it_jac < 1000):
3     x_old = x.copy()
4     for i in range(N):
5         s = 0
6         for j in range(N):
7             if j != i:
8                 # Sum off-diagonal*x_old
9                 s += A[i,j] * x_old[j]
10            # Compute new x value
11            x[i] = (b[i] - s) / A[i,i]
12
13    # Increase number of iterations
14    it_jac += 1
15    x_diff = norm(A@x - b)/norm(b)
```

Try to call it from the `laplace_demo.py` file, instead of using `spsolve`.

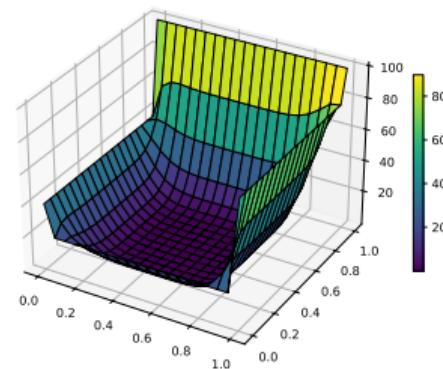
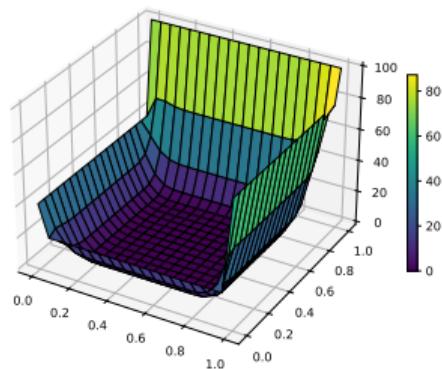
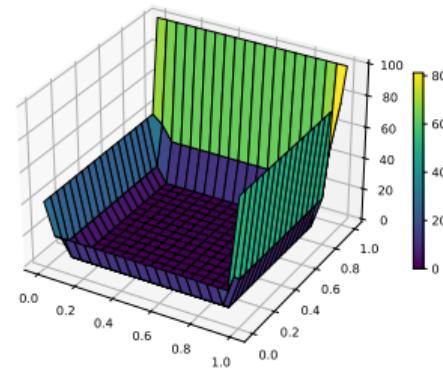
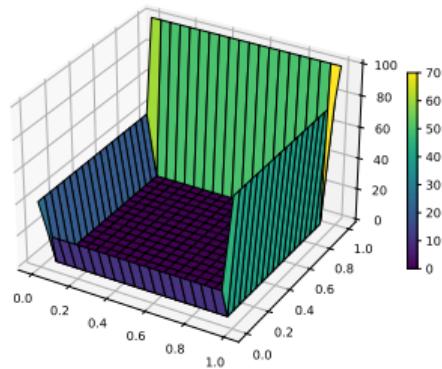
A few details on this algorithm

- The while loop holds two aspects
 - A convergence criterion (`norm(A@x - b)/norm(b) > tol`). Some considerations are:
 - L_1 -norm (sum)
 - L_2 -norm (Euclidian distance)
 - L_∞ -norm (max)
 - Protection against infinite loops (no convergence)
- Reset the sum for each row, before summing for the new unknown node
- Start vector x is not shown in the example, but should be there!
- It can have huge impact on performance!
- The for-loops also have a large performance penalty!

The solver using array indices

Make a copy of the Jacobian solver, and replace the for-loop on j by a vector-operation in a new function `jacobi_vec(A, b, tol=1e-2)`:

Iterations 1, 2, 5 and 10



Gauss-Seidel method

The Gauss-Seidel method is quite similar to Jacobi method

- The only difference is that the new estimate x^{new} is returned to the solution x^{old} as soon as it is completed
- For following nodes, the updated solution is used immediately
- Our straightforward script (from the Jacobi method) is therefore changed easily:
 - Do not create a T_{new} array (save memory!)
 - Do not store the solution in T_{new} , but simply in T
 - Do not perform the update step $T=T_{\text{new}}$
 - See `gaussseidel(A, b, tol=1e-2)` for the algorithm.
- The straightforward script works well for the current Laplace equation, but we define the generic Gauss-Seidel algorithm on the following slides.

Gauss-Seidel method

- Define a lower and strictly upper triangular matrix, such that $A = L + U$
- Now we can solve $AT=b$ by:

$$(L + U)T = b$$

$$LT = b - UT$$

$$LT^{\text{new}} = b - UT^{\text{old}}$$

$$T^{\text{new}} = L^{-1}(b - UT^{\text{old}})$$

- Using the n and $n+1$ notation for old and new time steps, we find in for the general Gauss-Seidel method:

$$x^{n+1} = L^{-1}(b - Ux^n)$$

$$x_i^{n+1} = \frac{1}{A_{ii}} \left(b_i - \sum_{j < i} A_{ij}x_j^{n+1} - \sum_{j > i} A_{ij}x_j^n \right)$$

Create yourself: Gauss-Seidel method

- Create a copy of the `jacobi` method and rename it to `gaussseidel`
- Rework the inner algorithm to reflect the changes for the Gauss-Seidel method
- Test! Perform a timing check and check if the solution is correct.
- Next, create a new copy of the just created method and vectorize it, analogous to our vectorized Jacobi method

Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

Summary

- Partial differential equations can be discretized into sparse systems of linear equations
- Sparse matrices can be stored in memory efficiently using specialised formats (e.g. compressed row storage)
- The Jacobi and Gauss-Seidel methods were introduced as iterative methods; other methods are based on the same principle (successive over-relaxation method, for example)
- Various implementation issues were discussed, e.g. vectorised computing, convergence tolerances

Direct methods vs. Iterative methods

- Iterative methods converge *gradually* to a solution while direct methods (possibly with partial pivoting) factorise a (set of) matrix(es) which allow to compute the solution by *substitution*.
- Direct methods generally use more memory, since they need to store also the result matrices.
- A strictly (or irreducibly) diagonally dominant matrix is a prerequisite for convergence of the Jacobi and Gauss-Seidel method.
- For real-life situations; 1D problems are generally solved with direct methods (LU decomposition). If you have systems of more than 1 dimension, a direct method still can be used, if there are no memory issues, otherwise an iterative method would be more attractive.

Numerical interpolation

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

- Introduction
 - Piecewise constant
 - Linear
 - Polynomial
 - Splines
 - Tutorials

Interpolation problem

Definition

Given a set of points x_k , $k = 0, \dots, n$, $x_i \neq x_j$ with associated function values f_k , $k = 0, \dots, n$, or simply: $\{x_k, f_k\}_{k=0}^n$. The interpolation problem is defined as: find a polynomial p_n such that this interpolates the values of f_k on the points x_k :

$$p_n(x_k) = f_k, \quad k = 0, \dots, n$$

Theorem

The interpolation problem for $\{x_k, f_k\}_{k=0}^n$ has a unique solution when $x_i \neq x_j$ for $i \neq j$. Note that we cannot allow multiple function values f_k for the same value of x_k .

What is interpolation?

Interpolation means constructing additional data points within the range of, and using, a discrete set of known data points.

It is typically performed on a uniformly spread data set, but this is not strictly necessary for all methods

Is interpolation the same as curve fitting?

NO

- Curve-fitting requires additionally some way of computing the error between function (curve) and data
- Curve-fitting does not strictly enforce the function to match the data exactly
- Curve-fitting may be done on multiple datapoints at one position
- Curve-fitting is much more expensive to do, requires optimisation

Why do chemical engineers need interpolation?

- Comparison of two data sets which are given at different positions
 - An experimental data set may have been recorded at a constant rate, but the numerical solution is computed at irregular intervals
- Reconstruction of field values distant of computing nodes
 - A CFD simulation on a regular grid containing structures that are not grid-conformant requires interpolation to the structures
- Calculation of a physical property at a condition between those of a lookup table
 - The viscosity of a substance may have been measured at 20°C and 30°C, but not at the desired 28.5°C

General

Several important numerical interpolation methods are discussed today:

- Piecewise constant interpolation
- Linear interpolation
 - Bilinear interpolation
- Polynomial interpolation (Newton's method)
- Spline interpolation

Today's outline

● Introduction

● Piecewise constant

● Linear

● Polynomial

● Splines

● Tutorials

Today's data set

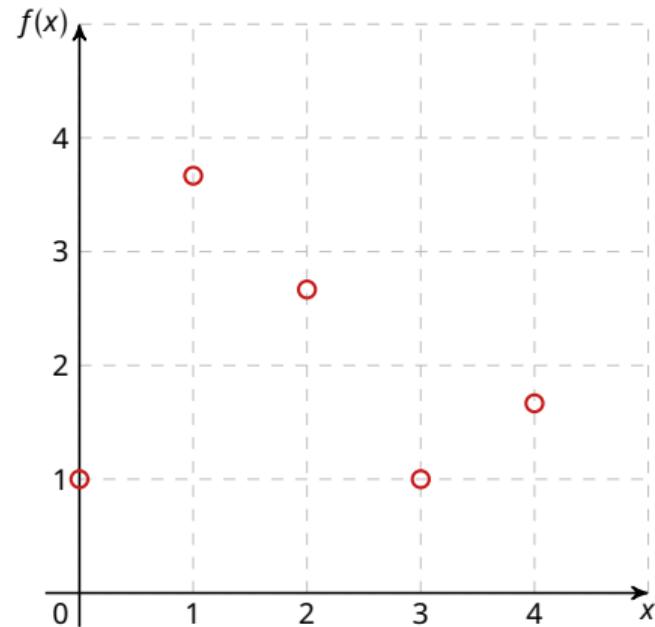
Generate the following data set:

```
1 import numpy as np
2 xdata = np.arange(0,6)
3 fun = lambda x: x**3/2 - (10*x**2)/3 + 11*x
4 /2 + 1
ydata = fun(xdata)
```

This yields some sample points on which we base our examples:

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$
3	1.00
4	$\frac{5}{3} = 1.67$
5	$\frac{23}{3} = 7.67$

Data set $f_h(x_n)$ represented by o at discrete intervals $x_n \in \{0, 5\}$



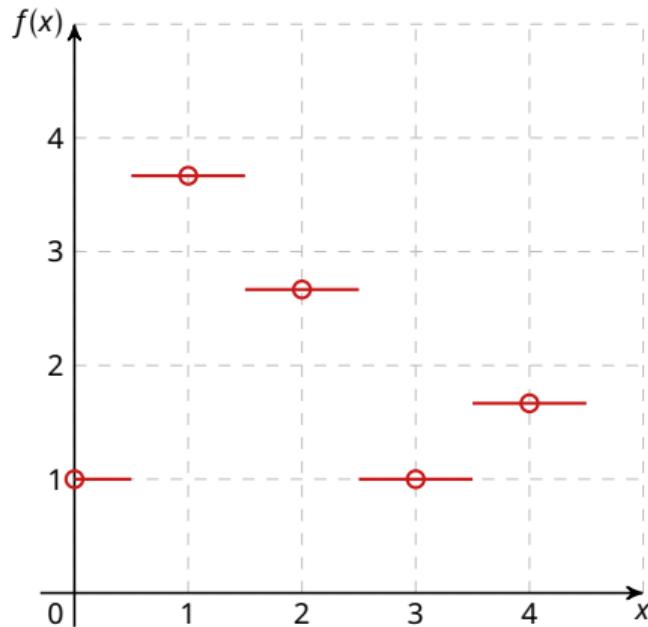
Piecewise constant interpolation

Data set $f_h(x_n)$ represented by \circ at discrete intervals $x_n \in \{0, 5\}$

- Nearest-neighbor interpolation in the continuous range $x \in [0, 5]$
- How to treat the point halfway (e.g. at $x = 2.5$)?

$$\begin{array}{ll} x \in [0, 0.5] & \rightarrow f(x) = f(0) \\ x \in [0.5, 1.5] & \rightarrow f(x) = f(1) \\ x \in [1.5, 2.5] & \rightarrow f(x) = f(2) \\ x \in [2.5, 3.5] & \rightarrow f(x) = f(3) \\ x \in [3.5, 4.5] & \rightarrow f(x) = f(4) \end{array}$$

- Not often used for simple problems, but e.g. for 2D (Voronoi)

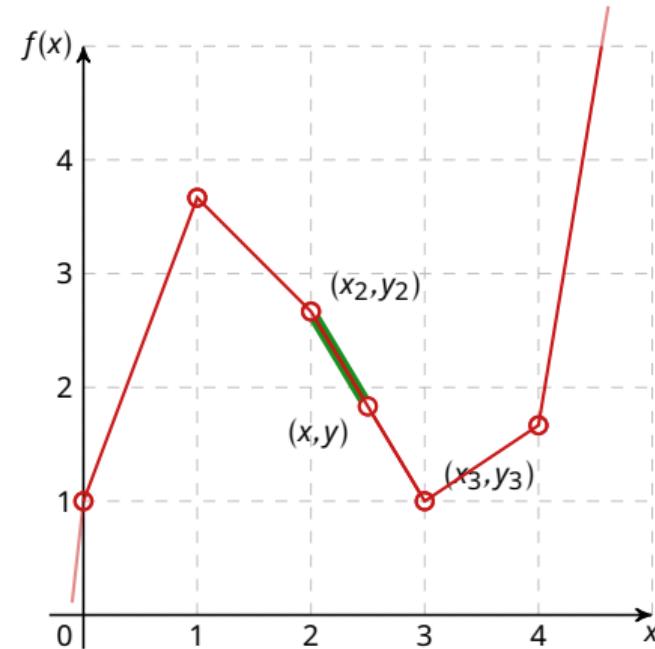


Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- Splines
- Tutorials

Linear interpolation

Data set $f_h(x_n)$ represented by ○ at discrete intervals $x_n \in \{0, 5\}$



- Linear interpolation to (x, y) between 2 data points (x_2, y_2) and (x_3, y_3) :

$$\frac{y - y_2}{x - x_2} = \frac{y_3 - y_2}{x_3 - x_2}$$

- Reordered, and more formally:

$$y = y_n + (y_{n+1} - y_n) \frac{x - x_n}{x_{n+1} - x_n}$$

Linear interpolation

- While linear interpolation is fast, and relatively easy to program, it is not very accurate
- At the nodes, the derivatives are discontinuous i.e. not differentiable
- Error is proportional to the square of the distance between nodes

Interpolation in Python

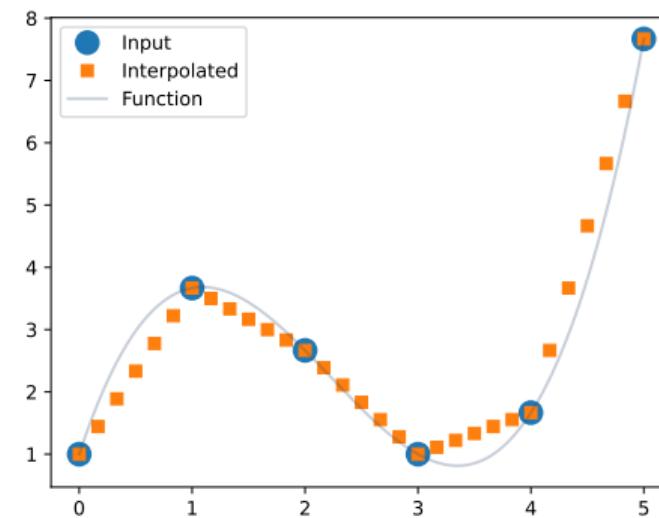
Interpolation can be done using the SciPy interpolation submodule, e.g.:

```
1 from scipy.interpolate import interp1d
2 f = interp1d(xdata, ydata, kind='linear')
```

This creates a function object `f` based on the given data.

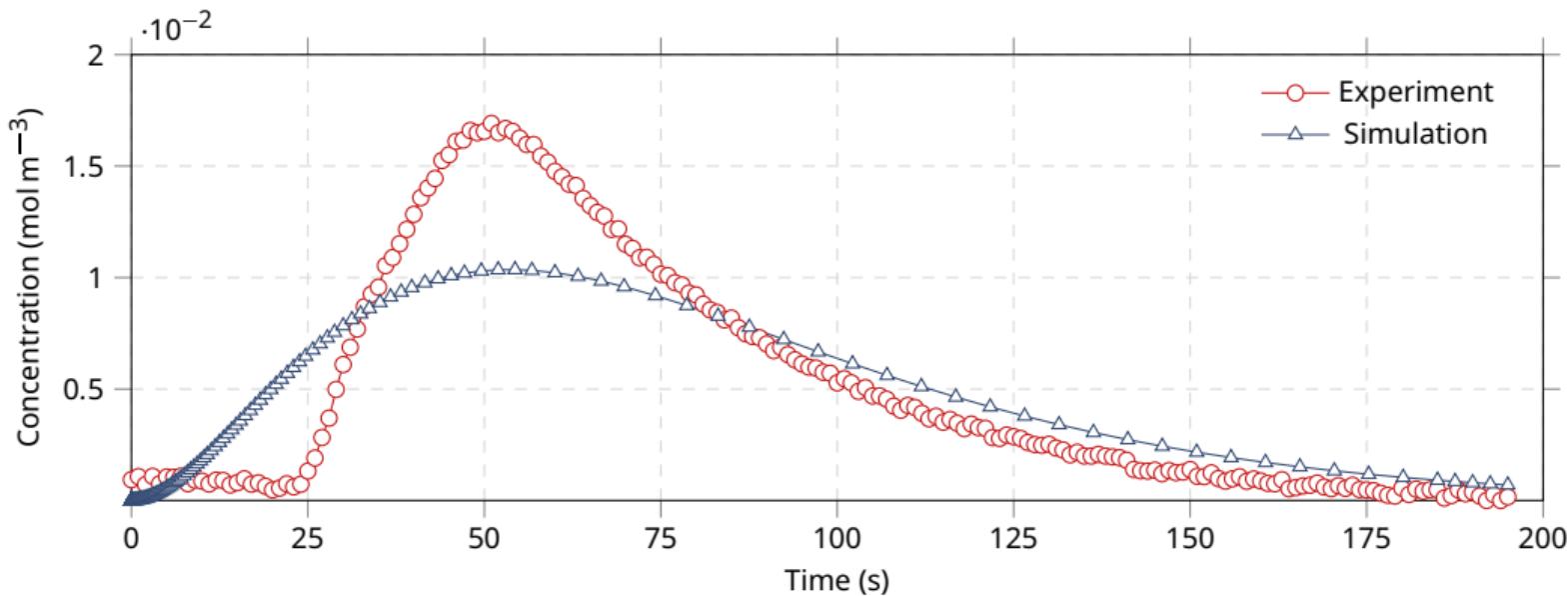
```
1 from scipy.interpolate import interp1d
2 import numpy as np
3
4 fun = lambda x: x**3/2 - (10*x**2)/3 + 11*x/2 + 1
5 xdata = np.arange(0,6)
6 ydata = fun(xdata)
7
8 f = interp1d(xdata,ydata)
9 xint = np.linspace(0,5,31)
10 yint = f(xint)
```

```
1 import matplotlib.pyplot as plt
2 plt.plot(xdata,ydata,'o',markersize=12,label='Input')
3 plt.plot(xint,yint,'s',label='Interpolated')
```



Example: Linear interpolation in Python

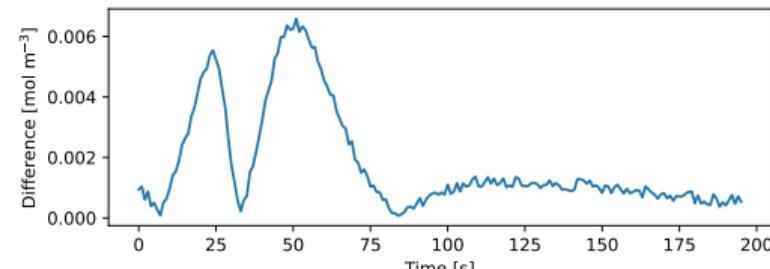
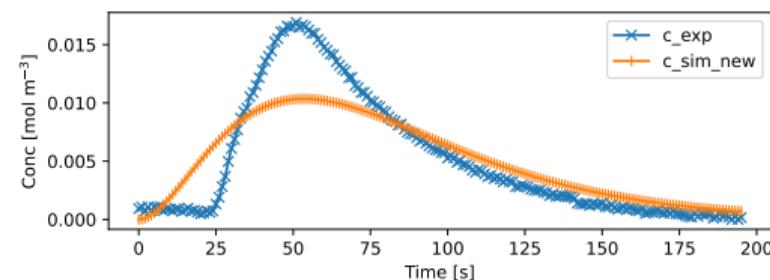
Consider the data sets in `exp_data.txt` and `sim_data.txt`, containing a normalized concentration and time vector for an experiment and a simulation. The simulation was performed with adaptive node distance to save computation time, thus the concentration is not known at the same times. We are not able to compare yet.



Example: Linear interpolation in Python

Consider the data sets in `exp_data.txt` and `sim_data.txt`, containing a normalized concentration and time vector for an experiment and a simulation. The simulation was performed with adaptive node distance to save computation time, thus the concentration is not known at the same times. We are not able to compare yet.

```
1 import numpy as np
2 from scipy.interpolate import interp1d
3 import matplotlib.pyplot as plt
4
5 t_sim, c_sim = np.loadtxt("scripts/interpolation/sim_data.txt").T
6 t_exp, c_exp = np.loadtxt("scripts/interpolation/exp_data.txt").T
7
8 # Linear interpolation
9 f = interp1d(t_sim, c_sim)
10 diff = np.abs(c_exp - f(t_exp))
11
12 # Plot the solution
13 plt.subplot(2, 1, 1)
14 plt.plot(t_exp, c_exp, '-x', label='c_exp')
15 plt.plot(t_exp, f(t_exp), '-|', label='c_sim_new')
16 plt.xlabel('Time [s]'); plt.ylabel('Conc [mol m$^{-3}$]')
17 plt.legend()
18
19 plt.subplot(2, 1, 2)
20 plt.plot(t_exp, diff)
21 plt.xlabel('Time [s]'); plt.ylabel('Difference [mol m$^{-3}$]')
22 plt.tight_layout()
23 # plt.show()
24 plt.savefig('figures/sim_exp_data_interp.pdf')
```



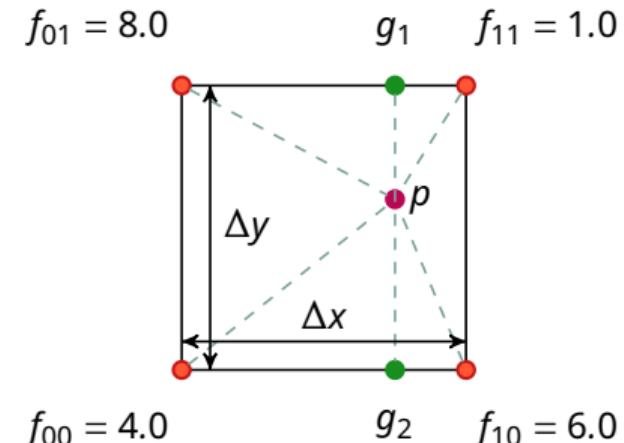
Bi-linear interpolation

When a 2D field of some quantity is known, we can interpolate the solution to an arbitrary position in the 2D domain $p(x, y)$ using 4 field values f_{00}, f_{10}, f_{01} and f_{11} .

$$\begin{aligned}g_1 &= f_{01} \frac{x_1 - x}{x_1 - x_0} + f_{11} \frac{x - x_0}{x_1 - x_0} \\&= f_{01} \frac{x_1 - x}{\Delta x} + f_{11} \frac{x - x_0}{\Delta x}\end{aligned}$$

$$g_2 = f_{00} \frac{x_1 - x}{\Delta x} + f_{10} \frac{x - x_0}{\Delta x}$$

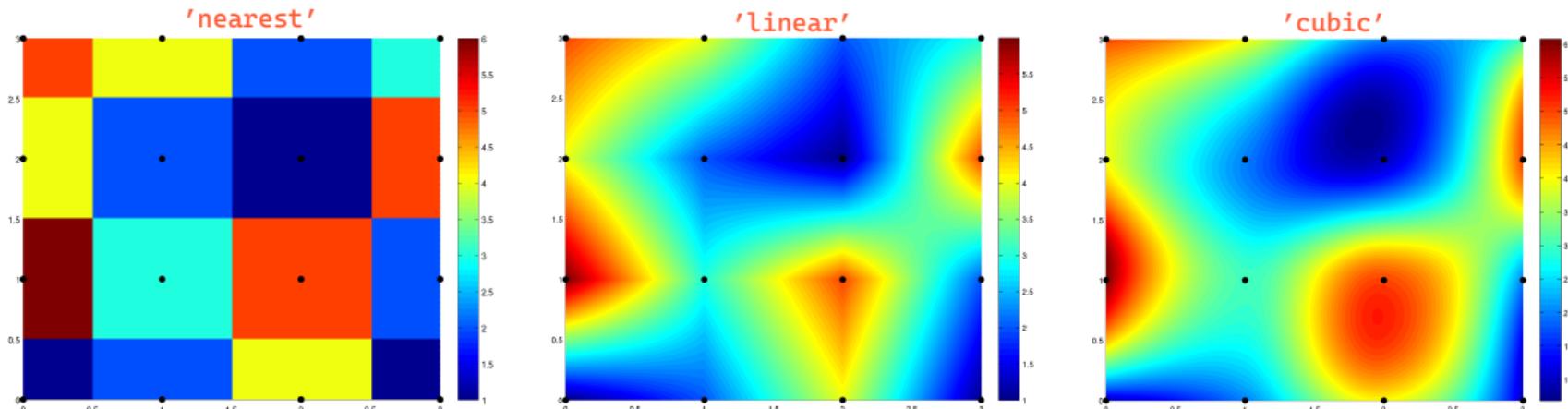
$$p = g_2 \frac{y_1 - y}{\Delta y} + g_1 \frac{y - y_0}{\Delta y}$$



- The order of interpolation (x or y direction first) does not matter; the results are equal

Higher-dimensional field interpolation in Python

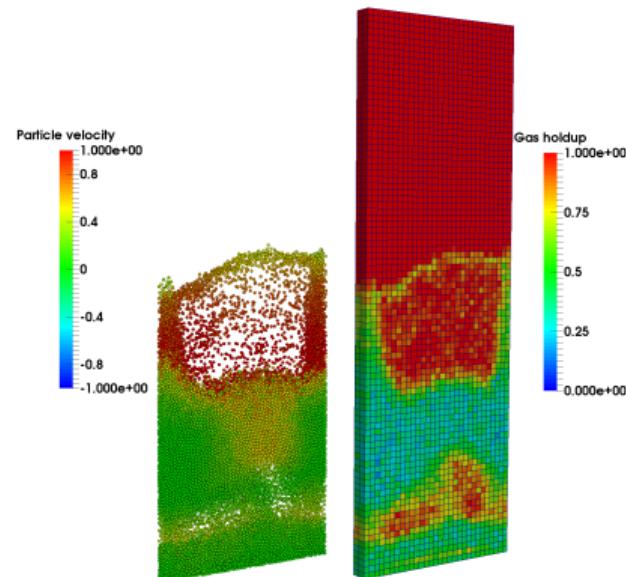
2D or higher-dimensional fields of data can be interpolated in Python using the `scipy.interpolate.interp2d`, `scipy.interpolate.interp3d`, or even `scipy.interpolate.RegularGridInterpolator` functions. The method can be adjusted:



- Also consider tri-linear interpolation (for 3D fields) with `scipy.interpolate.LinearNDInterpolator`, or bicubic interpolation (2D, but third order) with `scipy.interpolate.interp2d`.

A practical example

Field interpolation is used in e.g. CFD simulations, e.g. a fluidized bed simulation using a *discrete particle model*, where particles are found in between the grid nodes used for velocity computation.



Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- Splines
- Tutorials

Polynomial interpolation

The examples that we have seen, are simplified forms of *Newton polynomials*. We can interpolate our data with a polynomial of degree n :

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$$

Polynomial interpolation via Vandermonde matrix

Consider the data points $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$, the Vandermonde matrix V , coefficient vector a and function value vector y :

$$V_{m,n} = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_m^0 & x_m^1 & x_m^2 & \cdots & x_m^{n-1} \end{pmatrix} \quad a = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

The coefficients of a polynomial through the data are obtained by solving the linear system $Va = y$.

```
1 import numpy as np
2 x = np.array([0, 1, 2])
3 y = np.array([1.0000, 3.6667, 2.6667])
4 V = np.vander(x, increasing=True)
5 print(V)
```

```
[ 1.  4.50005 -1.83335]
```

So we found the equation:

$$p_2(x) = -1.8333x^2 + 4.5x - 1$$

These Vandermonde-systems are often *ill-conditioned*,
so we need another, more stable, method!

Construction of Newton polynomials

Formally, the polynomials $p_n(x)$ are described using prefactors $f[x_0, \dots, x_k]$ and polynomial terms $w_m(x)$:

$$p_n(x) = \sum_{k=0}^n f[x_0, \dots, x_k] w_k(x)$$

The polynomial terms are computed via:

$$w_0(x) = 1, \quad w_1(x) = (x - x_0), \quad w_2(x) = (x - x_0) \cdot (x - x_1),$$

$$w_m(x) = (x - x_0) \cdot (x - x_1) \cdots (x - x_{m-1}) = w_{m-1} \cdot (x - x_{m-1})$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j), \quad m = 0, \dots, n$$

The prefactors are *forward divided differences*, which can be computed as:

$$f[x_{r-k}, \dots, x_r] \equiv \frac{f[x_{r-k+1}, \dots, x_r] - f[x_{r-k}, \dots, x_{r-1}]}{x_r - x_{r-k}}$$

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^n f[x_0, \dots, x_k] w_k(x)$$

$$f[x_{r-k}, \dots, x_r] \equiv \frac{f[x_{r-k+1}, \dots, x_r] - f[x_{r-k}, \dots, x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

x_k	f_k
x_0	$f[x_0] = f_0$
x_1	$f[x_1] = f_1$
x_2	$f[x_2] = f_2$

$$f[x_0, x_1] = \frac{f_1 - f_0}{x_1 - x_0}$$

$$f[x_1, x_2] = \frac{f_2 - f_1}{x_2 - x_1}$$

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

x_k	f_k
0	1
1	3.67
2	2.67

$$\frac{\frac{11}{3} - 1}{1 - 0} = \frac{8}{3}$$

$$\frac{\frac{8}{3} - \frac{11}{3}}{2 - 1} = \frac{-1}{1} = -1$$

$$\frac{(-1) - \frac{8}{3}}{2 - 0} = -\frac{11}{6}$$

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^n f[x_0, \dots, x_k] w_k(x)$$

$$f[x_{r-k}, \dots, x_r] \equiv \frac{f[x_{r-k+1}, \dots, x_r] - f[x_{r-k}, \dots, x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

x_k	f_k
0	1
1	3.67
2	2.67

$$\begin{aligned}
 p_2(x) &= 1 \cdot w_m(0) + \frac{8}{3} \cdot w_m(1) + \left(-\frac{11}{6}\right) \cdot w_m(2) \\
 &= 1 \cdot 1 + \frac{8}{3} \cdot (x - 0) + \left(-\frac{11}{6}\right) \cdot (x - 0)(x - 1) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1
 \end{aligned}$$

Construction of Newton polynomials: example

For each three points, a new polynomial interpolant can be derived:

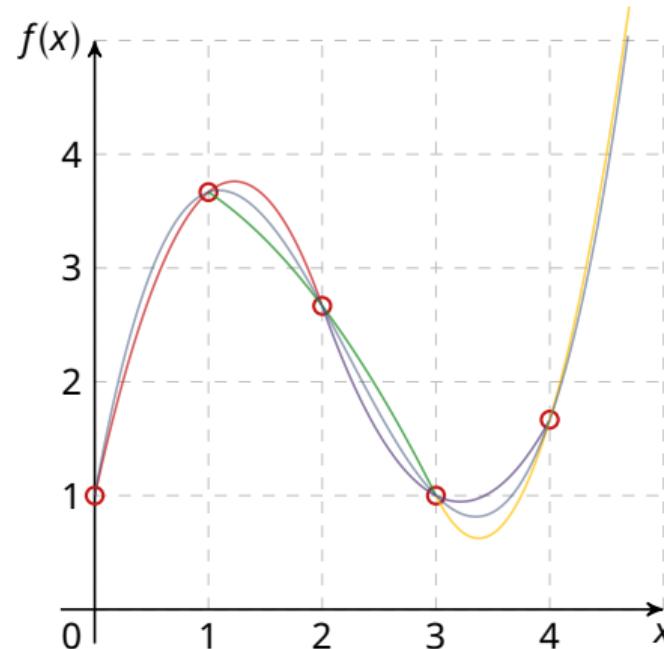
$$p_2(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$$

$$p_2(x) = 4 - \frac{x^2}{3}$$

$$p_2(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$$

$$p_2(x) = \frac{8}{3}x^2 - 18x + 31$$

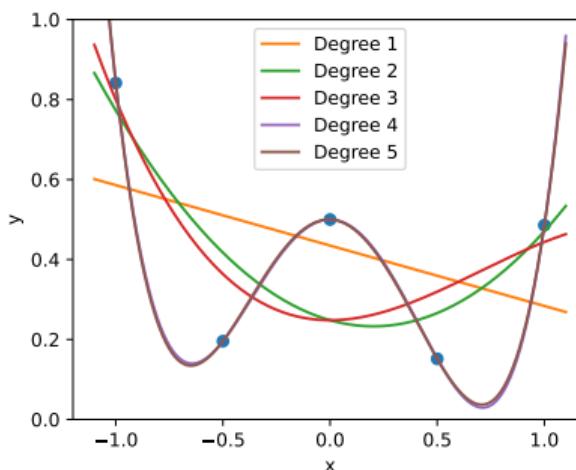
$$f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$$



Polynomial fitting in Python: example

Develop the polynomials $p_1(x)$ through $p_5(x)$ using the following data set:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 xdata = np.arange(-1,1.5,0.5)
4 ydata = [x * np.sin(x)/np.sqrt(x+2) if x != 0 else 0.5 for x in xdata]
5 plt.plot(xdata,ydata,'o')
```



```
1 xc = np.linspace(-1.1,1.1,1001,endpoint=True)
2 for deg in range(1,6):
3     # Fit coefficients
4     p_coeffs = np.polyfit(xdata,ydata,deg)
5     # Compute function values
6     y = np.polyval(p_coeffs,xc)
7     # Plot
8     plt.plot(xc,y,label=f'Degree {deg}')
```

RankWarning: Polyfit may be poorly conditioned

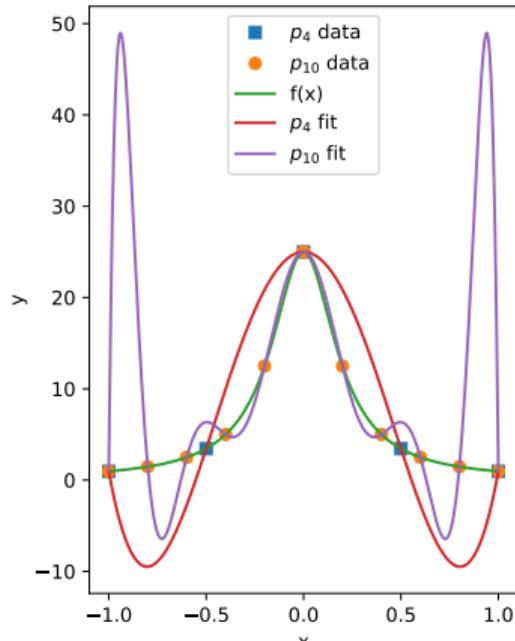
Exercise

Develop the $p_4(x)$ and $p_{10}(x)$ interpolants from the following data sets:

$$f(x) = \frac{1}{x^2 + \frac{1}{25}} \quad x \in [-1, 1]$$

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 f = lambda x: 1/(x**2 + 1/25)
4 x4,x10,xinf = [np.linspace(-1, 1, n) for n in [5,11,1001]]
5 y4,y10,yinf = f(x4), f(x10), f(xinf)
```

```
6 # Get coefficients for 4th and 10th order polynomial
7 p4 = np.polyfit(x4, y4, 4)
8 p10 = np.polyfit(x10, y10, 10)
9 # Compute function values using fitted coeffs
10 yinf4 = np.polyval(p4, xinf)
11 yinf10 = np.polyval(p10, xinf)
```



Final thoughts on polynomial interpolation

- An polynomial interpolant of order n requires $n + 1$ data points
 - More data points: interpolant does *not always* cross the points
 - Fewer data points: interpolant is not unique
- Higher-degree polynomials at equidistant points may cause strong oscillatory behaviour (Runge's phenomenon)
 - Mitigation of the problem on Chebyshev (i.e. non uniform grid)...
 - ... or by performing piecewise interpolation (next topic)
- Python functions `np.polyfit(x,y,n)` and `np.polyval(p,x_new)` were demonstrated.

Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- Splines
- Tutorials

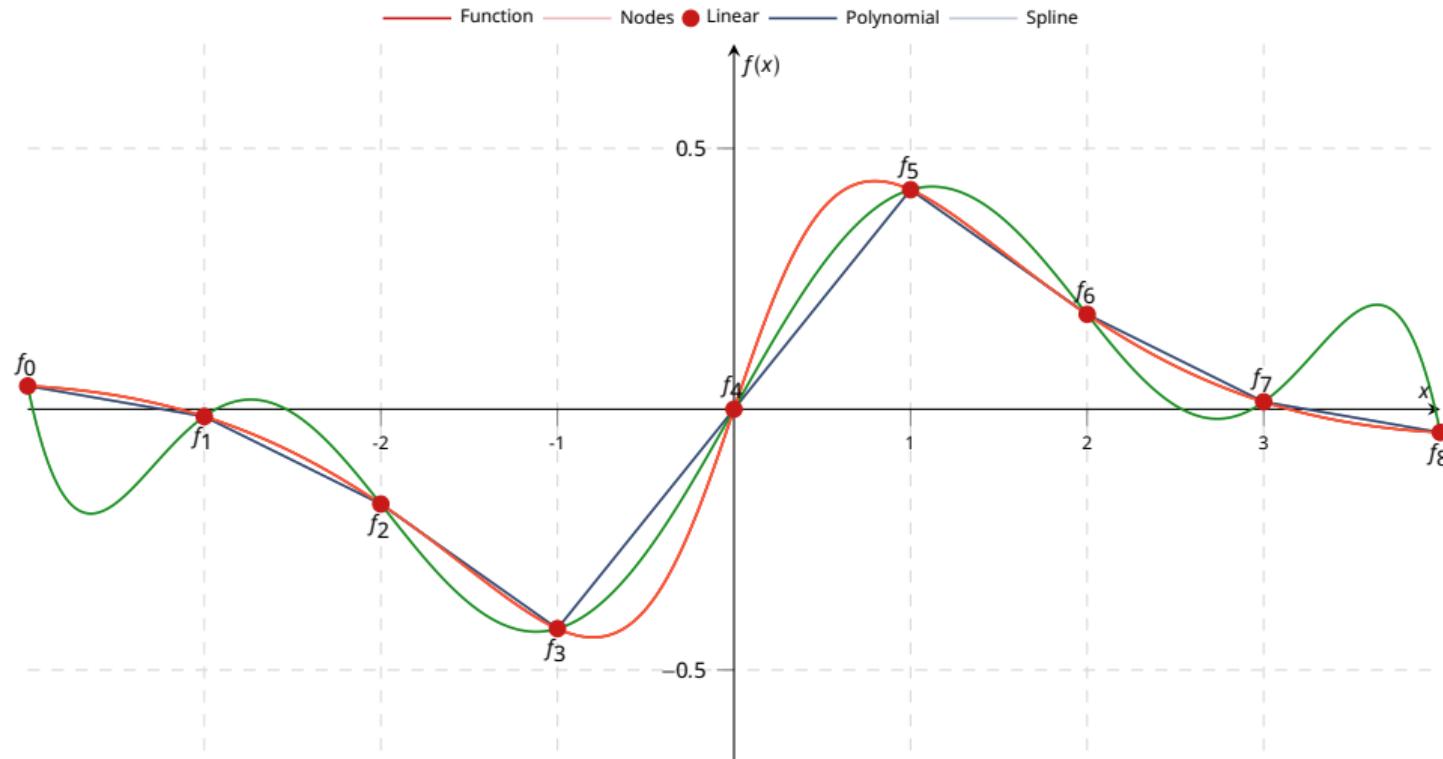
Spline interpolation

A spline is a numerical function that represents a **smooth, higher order, piecewise polynomial** interpolants of a data set.

- Smooth: the interpolant is continuous in the first and second derivatives
- Higher order: The most common type of splines uses third-order polynomials (cubic splines)
- Piecewise polynomial: The interpolant is constructed between each two consecutive tabulated points

Splines: comparison to other interpolation techniques

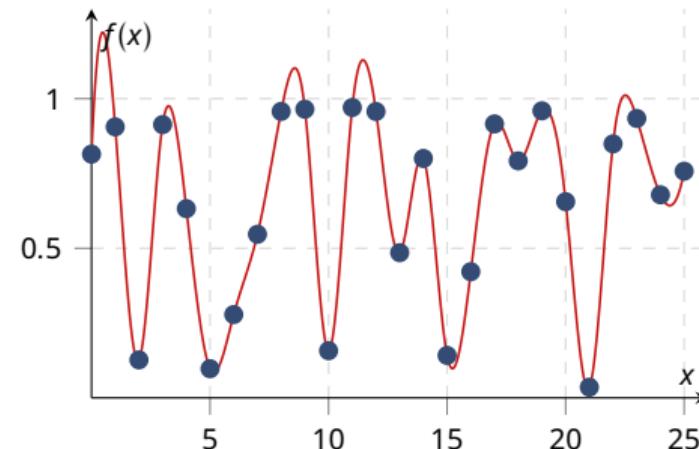
$$\text{Interpolation of } f(x) = \frac{\sin x}{1+x^2}$$



Spline interpolation in Python

We can generate a random data set, and interpolate using `scipy.interpolate.interp1d`:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.interpolate import make_interp_spline
4
5 # Generate random data set
6 xdata = np.arange(0, 26)
7 ydata = np.random.rand(len(xdata))
8
9 # Interpolant on a fine mesh
10 xc = np.linspace(0, 25, 1001)
11 ifun = make_interp_spline(xdata, ydata)
12 yc = ifun(xc)
13
14 # Plot the data
15 plt.plot(xdata, ydata, 'o')
16 plt.plot(xc, yc, '-r')
17 plt.show()
```



Note: The `SciPy Optimize` module contains various interpolation methods with a similar interface.

Summary

- Interpolation is used to obtain data between existing data points
 - (Bi-)Linear, polynomial and spline interpolation methods
 - Construction of Newton polynomials
 - Oscillations of high-order polynomials
- Legendre polynomials: alternative way of performing the polynomial interpolation (not discussed here)

Interpolation tutorials

- ① In Python, generate the data:

```
1 x = np.arange(-4, 6, 1)
2 y = [0, 0, 0, 1, 1, 1, 0, 0, 0]
```

Interpolate the data using polynomial interpolation (which order do you use?) and a spline. Plot the results together with the original data in a graph.

- ② Do the same exercise for the following data. Can you explain your observations?

```
1 t = [0, 0.1, 0.499, 0.5, 0.6, 1.0, 1.4, 1.5, 1.899, 1.9, 2.0]
2 y = [0, 0.06, 0.17, 0.19, 0.21, 0.26, 0.29, 0.29, 0.30, 0.31, 0.31]
```

Hint: Use `scipy.interpolate.interp1d(..., kind="...")` to use different splines.

Numerical integration

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

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion
- Tutorials

What is numerical integration?

To determine the integral $I(x)$ of an integrand $f(x)$, which can be used to compute the area underneath the integrand between $x = a$ and $x = b$.

$$I(x) = \int_a^b f(x)dx$$

Today we will outline different numerical integration methods.

- Riemann integrals
- Trapezoidal rule
- Simpson's rule

Why do chemical engineers need integration?

- Obtaining the cumulative particle size distribution from a particle size distribution
- The concentration outflow over time may be integrated to yield the residence time distribution
- Integration of a varying product outflow yields the total product outflow
- Quantitative analysis of mixture components via e.g. GC/MS
- Not all function have an explicit antiderivative, e.g. $\int e^{x^2} dx$ or $\int \frac{1}{\ln x} dx$

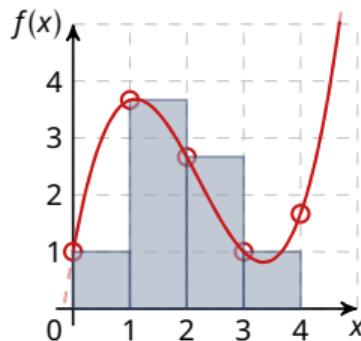
Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion
- Tutorials

Riemann integrals

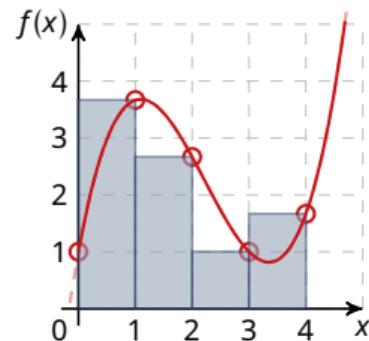
Basic idea: Subdivide the interval $[a, b]$ into n subintervals of equal length $\Delta x = \frac{b-a}{n}$ and use the sum of area to approximate the integral.

Left endpoint rule



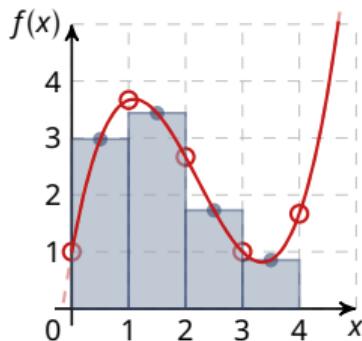
$$L_n = \sum_{i=1}^n f(x_{i-1}) \Delta x_i$$

Right endpoint rule



$$R_n = \sum_{i=1}^n f(x_i) \Delta x_i$$

Midpoint rule



$$M_n = \sum_{i=1}^n f(\bar{x}_i) \Delta x_i$$

$$\text{with } \bar{x}_i = \frac{x_{i-1} + x_i}{2}$$

Errors in Riemann integrals

We define the exact integral as $I = \int_a^b f(x)dx$, and L_n , R_n and M_n represent the left, right and midpoint rule approximations of I based on n intervals.

Writing $f_{\max}^{(k)}$ for the maximum value of the k -th derivative, the upper-bounds of the errors by Riemann integrals are:

- $|I - L_n| \leq \frac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - R_n| \leq \frac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - M_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{24n^2}$

Note that while $|I - L_n|$ and $|I - R_n|$ give the same *upper-bounds* of the error, this does not mean the same error. Rather, the error is of opposite sign!

Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion
- Tutorials

Trapezoid rule

Since the sign of the approximation error of the left and right endpoint rules is opposite, we can take the average of these approximations:

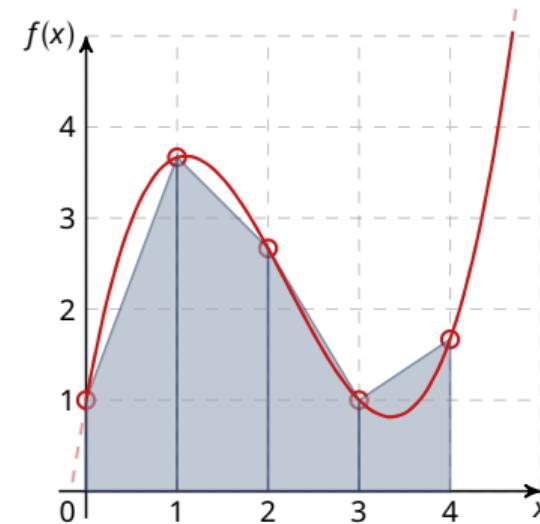
$$T_n = \frac{L_n + R_n}{2}$$

The total area is obtained by geometric reconstruction of trapezoids:

$$T_n = \sum_{i=1}^n \frac{f(x_{i+1}) + f(x_i)}{2} \Delta x_i$$

Note that this can be rewritten for equidistant intervals:

$$T_n = \frac{b-a}{2n} (f(x_0) + 2f(x_1) + \dots + 2f(x_{n-1}) + f(x_n))$$



Error in trapezoid integration

The trapezoid rule result over n intervals T_n approximates the exact integral $I = \int_a^b f(x)dx$. The upper-bounds of the error is given as:

$$|I - T_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{12n^2}$$

Recall that the midpoint rule approximates with an upper-bound error of

$$|I - M_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{24n^2}$$

The midpoint rule approximation has lower error bounds than the trapezoid rule. A linear function is, however, better approximated by the trapezoid rule.

Today's outline

● Introduction

● Riemann integrals

● Trapezoid rule

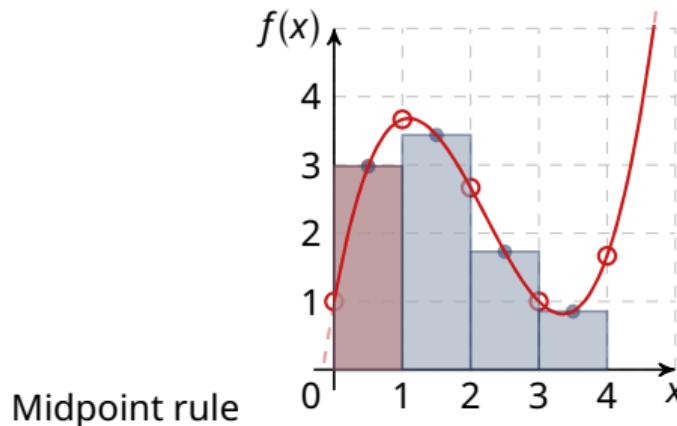
● Simpson's rule

● Conclusion

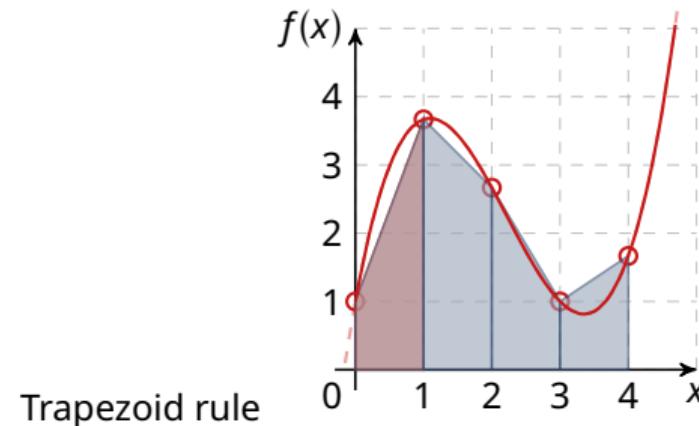
● Tutorials

Towards higher-order integration

Compare how the midpoint and trapezoid functions behave on convex and concave parts of a graph.



Midpoint rule



Trapezoid rule

In convex parts (bending down), the midpoint rule tends to overestimate the integral (trapezoid underestimates). In concave parts (bending up), the midpoint rule tends to underestimate the integral (trapezoid overestimates).

Towards higher-order integration

The errors of the midpoint rule and trapezoid rule behave in a similar way, but have opposite signs.

- Midpoint: $|I - M_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{24n^2}$
- Trapezoid: $|I - T_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{12n^2}$

For a quadratic function, the errors relate as:

$$|I - M_n| = \frac{1}{2}|I - T_n|$$

Taking the weighted average of these two yields the Simpson's rule:

$$S_{2n} = \frac{2}{3}M_n + \frac{1}{3}T_n$$

The $2n$ means we have $2n$ subintervals: the n trapezoid intervals are subdivided by the midpoint rule.

Simpson's rule

Consider the interval $i \in [x_0, x_2]$, subdivided in three equidistant interpolation points: x_0, x_1, x_2 .

- Midpoint: $M_i = f\left(\frac{x_0 + x_2}{2}\right)2\Delta x = f(x_1)2\Delta x$
- Trapezoid: $T_i = \frac{f(x_0) + f(x_2)}{2}2\Delta x$
- Simpson: $S_i = \frac{2}{3}M_i + \frac{1}{3}T_i$

Note that M_i and T_i were computed on interval $x_2 - x_0 = 2\Delta x$.

Now we have:

$$\begin{aligned} S_i &= \frac{2}{3}[f(x_1)2\Delta x] + \frac{1}{3}\left[\frac{f(x_0) + f(x_2)}{2}2\Delta x\right] \\ &= \frac{4\Delta x}{3}f(x_1) + \frac{\Delta x}{3}f(x_0) + f(x_2) = \frac{\Delta x}{3}(f(x_0) + 4f(x_1) + f(x_2)) \end{aligned}$$

Simpson's rule

We write $f(x_k) = f_k$. The integral of an interval $i \in [x_0, x_2]$ is approximated as:

$$S_i = \frac{\Delta x}{3} (f_0 + 4f_1 + f_2)$$

The next interval, S_j with $j \in [x_2, x_4]$ with midpoint $x_3 = \frac{x_2+x_4}{2}$ is approximated as:

$$S_j = \frac{\Delta x}{3} (f_2 + 4f_3 + f_4)$$

If we sum these two intervals we obtain:

$$\begin{aligned} I \approx S_i + S_j &= \left[\frac{\Delta x}{3} (f_0 + 4f_1 + f_2) \right] + \left[\frac{\Delta x}{3} (f_2 + 4f_3 + f_4) \right] \\ &= \frac{\Delta x}{3} (f_0 + 4f_1 + 2f_2 + 4f_3 + f_4) \end{aligned}$$

Simpson's rule

In general, Simpson's rule can be written as:

$$\int_a^b f(x)dx \approx \sum_{\substack{k=2 \\ k \text{ even}}}^n \frac{\Delta x}{3} (f_{k-2} + 4f_{k-1} + f_k)$$
$$= \frac{\Delta x}{3} (f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \dots + 2f_{n-2} + 4f_{n-1} + f_n)$$

The error is given by:

$$|I - S_n| \leq \frac{f_{\max}^{(4)}(b-a)^5}{180n^4}$$

if integrand f is differentiable on $[a, b]$.

Simpson's rule: example

Recall our example data, described by $f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$

$$I = \int_0^4 \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1 = \frac{80}{9} \approx 8.888\dots$$

- Interpolating x_0, x_1 and x_2 : $p_{2a}(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$

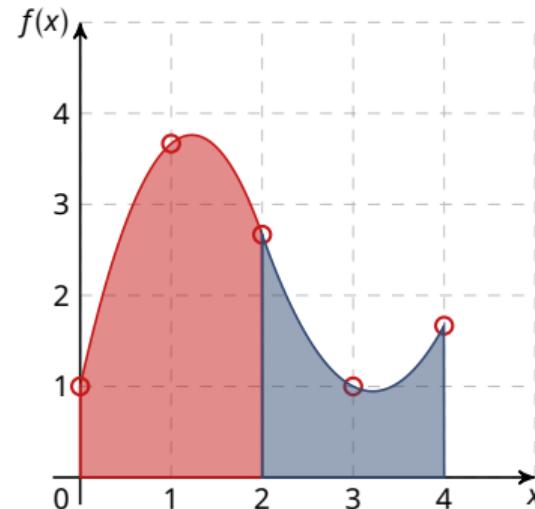
$$\int_0^2 p_{2a} = \frac{55}{9} \approx 6.1111$$

- Interpolating x_2, x_3 and x_4 : $p_{2b}(x) = \frac{7}{6}x^2 - 7\frac{1}{2}x + 13$

$$\int_2^4 p_{2b} = \frac{25}{9} \approx 2.777\dots$$

- Adding the separate integrals:

$$\int_0^2 p_{2a} + \int_2^4 p_{2b} = \frac{80}{9}$$



Using Simpson's rule:

$$I \approx \frac{\Delta x}{3} (f_0 + 4f_1 + 2f_2 + 4f_3 + f_4) = \frac{1}{3} (1 + 4 \cdot 3.6667 + 2 \cdot 2.6667 + 4 \cdot 1.0000 + 1.6667) = 8.88888 = \frac{80}{9}$$

Simpson's method is of fourth order, and it gives exact approximations of third order polynomials!

Integration in Python

Integration can be done numerically in Python.

- `np.trapz(y, x)` uses the trapezoid rule to integrate the data. Make sure you use the `x` variable if your data is not spaced with $\Delta x = 1$. Can handle non-equidistant data.

```
1 import numpy as np
2 x = np.linspace(-2, 2, 2001)
3 y = 1 / (x**2 + 1)
4 I = np.trapz(y, x) # Or: scipy.integrate.trapezoid
5 print(I)
```

```
2.214297328921525
```

- Integration of functions can be done using the `quad(func, a, b)` function:

```
1 import numpy as np
2 from scipy.integrate import quad
3 f = lambda x: np.exp(-x**2)
4 I, err = quad(f, 0, 10)
5 print(I, err)
```

```
0.886226925452758 1.8483380528941764e-13
```

Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion
- Tutorials

What hasn't been discussed?

This course is by no means complete, and further reading is possible.

- Gaussian quadrature: A third-order integration method that requires only two base points (in contrast to the third order Simpson's method, which requires three points)
- Adaptive techniques: Parts of a function that are relatively steady (no wild oscillations) and differentiable can be integrated with much larger step sizes than other parts of the function.
- Simpson's 3/8-rule: Yet another integration technique, requiring an additional data point

Summary

- Several techniques for numerical integration were discussed:
 - Riemann sums, trapezoid rule, Simpson's rule
 - Upper-bound errors were given for each technique
 - Built-in Python functions were illustrated
- Continue with characterization of convergence of the integration methods in the tutorials!

Integration tutorials

- ① Implement a function to integrate a mathematical function for a specific number of integration intervals. Implement it as a function, which can be called with arguments:
 - Function (handle) to integrate
 - Integration boundaries (as separate arguments or as a 2×1 numpy array)
 - Number of integration intervals

For instance: `def leftrule(func, x0, x1, N):..`

- ② Set up a function to integrate:

```
1 def myfunction(x):  
2     return x**2 - 4*x + 6 + np.sin(5*x)
```

- ③ Integrate the function, e.g. `int_left = leftrule(myfunction, 0, 10, 25)`
- ④ Assess how the number of intervals affects the deviation from the true integral value.
- ⑤ Create a log-log plot of the deviation vs. number of intervals used.
- ⑥ Do this for all methods discussed² and compare their performance in a graph

²Riemann left, right, midpoint, trapezoid, and Simpson

Ordinary differential equations 1

Explicit techniques for ODEs

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

● Introduction

● Euler's method

- Forward Euler

● Rates of convergence

● Runge-Kutta methods

- RK2 methods
 - RK4 method

● Step size control

● Solving ODEs in Python

Overview

Ordinary differential equations

An equation containing a function of one independent variable and its derivatives, in contrast to a *partial differential equation*, which contains derivatives with respect to more independent variables.

Main question

How to solve

$$\frac{dy}{dx} = f(y(x), x) \quad \text{with} \quad y(x=0) = y_0$$

accurately and efficiently?

What is an ODE?

- Algebraic equation:

$$f(y(x), x) = 0 \quad \text{e.g. } -\ln(K_{eq}) = (1 - \zeta)$$

- First order ODE:

$$f\left(\frac{dy}{dx}(x), y(x), x\right) = 0 \quad \text{e.g. } \frac{dc}{dt} = -kc^n$$

- Second order ODE:

$$f\left(\frac{d^2y}{dx^2}(x), \frac{dy}{dx}(x), y(x), x\right) = 0 \quad \text{e.g. } \mathcal{D}\frac{d^2c}{dx^2} = -\frac{kc}{1 + Kc}$$

About second order ODEs

Very often a second order ODE can be rewritten into a system of first order ODEs (whether it is handy depends on the boundary conditions!)

More general

Consider the second order ODE:

$$\frac{d^2y}{dx^2} + q(x) \frac{dy}{dx} = r(x)$$

Now define and solve using z as a new variable.

$$\frac{dy}{dx} = z(x)$$

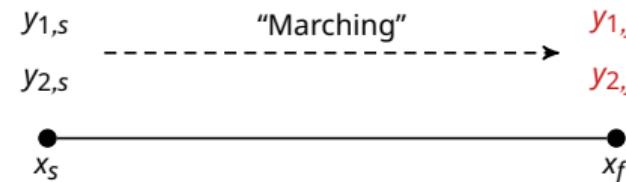
$$\frac{dz}{dx} = r(x) - q(x)z(x)$$

Importance of boundary conditions

The nature of boundary conditions determines the appropriate numerical method. Classification into 2 main categories:

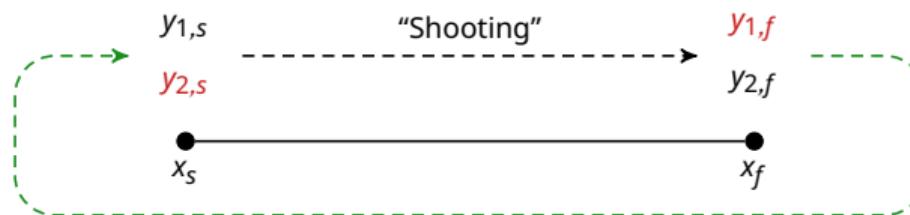
- *Initial value problems (IVP)*

We know the values of all y_i at some starting position x_s , and it is desired to find the values of y_i at some final point x_f .



- *Boundary value problems (BVP)*

Boundary conditions are specified at more than one x . Typically, some of the BC are specified at x_s and the remainder at x_f .



Overview

Initial value problems:

- Explicit methods
 - First order: forward Euler
 - Second order: improved Euler (RK2)
 - Fourth order: Runge-Kutta 4 (RK4)
 - Step size control
- Implicit methods
 - First order: backward Euler
 - Second order: midpoint rule

Boundary value problems

- Shooting method

Today's outline

- Introduction
- Euler's method
 - Forward Euler
- Rates of convergence
- Runge-Kutta methods
 - RK2 methods
 - RK4 method
- Step size control
- Solving ODEs in Python

Euler's method

Consider the following single initial value problem:

$$\frac{dc}{dt} = f(c(t), t) \quad \text{with} \quad c(t=0) = c_0 \quad (\text{initial value problem})$$

Easiest solution algorithm: Euler's method, derived here via Taylor series expansion:

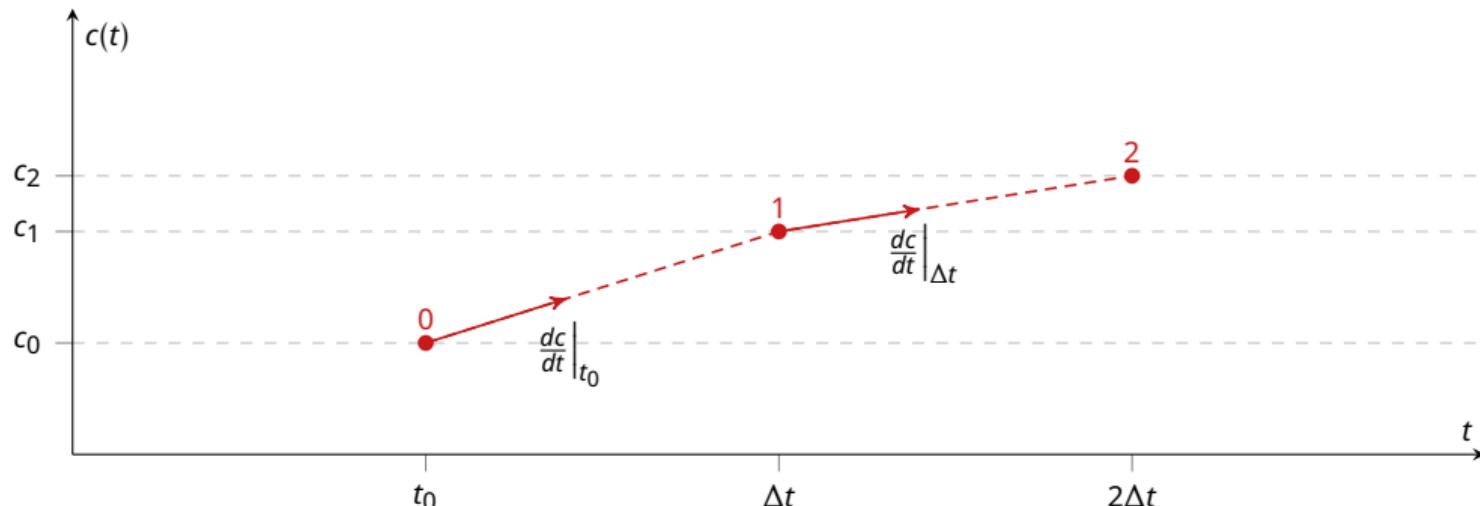
$$c(t_0 + \Delta t) \approx c(t_0) + \left. \frac{dc}{dt} \right|_{t_0} \Delta t + \frac{1}{2} \left. \frac{d^2 c}{dt^2} \right|_{t_0} (\Delta t)^2 + \mathcal{O}(\Delta t^3)$$

Neglect terms with higher order than two: $\left. \frac{dc}{dt} \right|_{t_0} = \frac{c(t_0 + \Delta t) - c(t_0)}{\Delta t}$ Substitution:

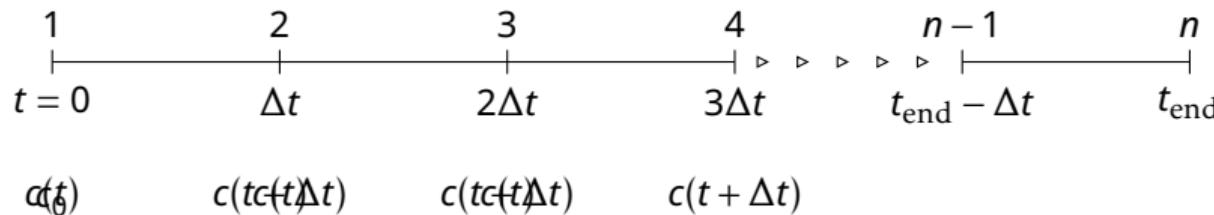
$$\frac{c(t_0 + \Delta t) - c(t_0)}{\Delta t} = f(c_0, t_0) \Rightarrow c(t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$$

Euler's method: graphical example

$$\frac{c(t_0 + \Delta t) - c(t_0)}{\Delta t} = f(c_0, t_0) \Rightarrow c(t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$$



Euler's method - solution method



Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
 $c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Pseudo-code Euler's method: $\frac{dy}{dx} = f(x,y)$ and $y(x_0) = y_0$.

- ① Initialize variables, functions; set $h = \frac{x_1 - x_0}{N}$
- ② Set $x = x_0$, $y = y_0$
- ③ While $x < x_{\text{end}}$ do
 $x_{i+1} = x_i + h$; $y_{i+1} = y_i + hf(x_i, y_i)$

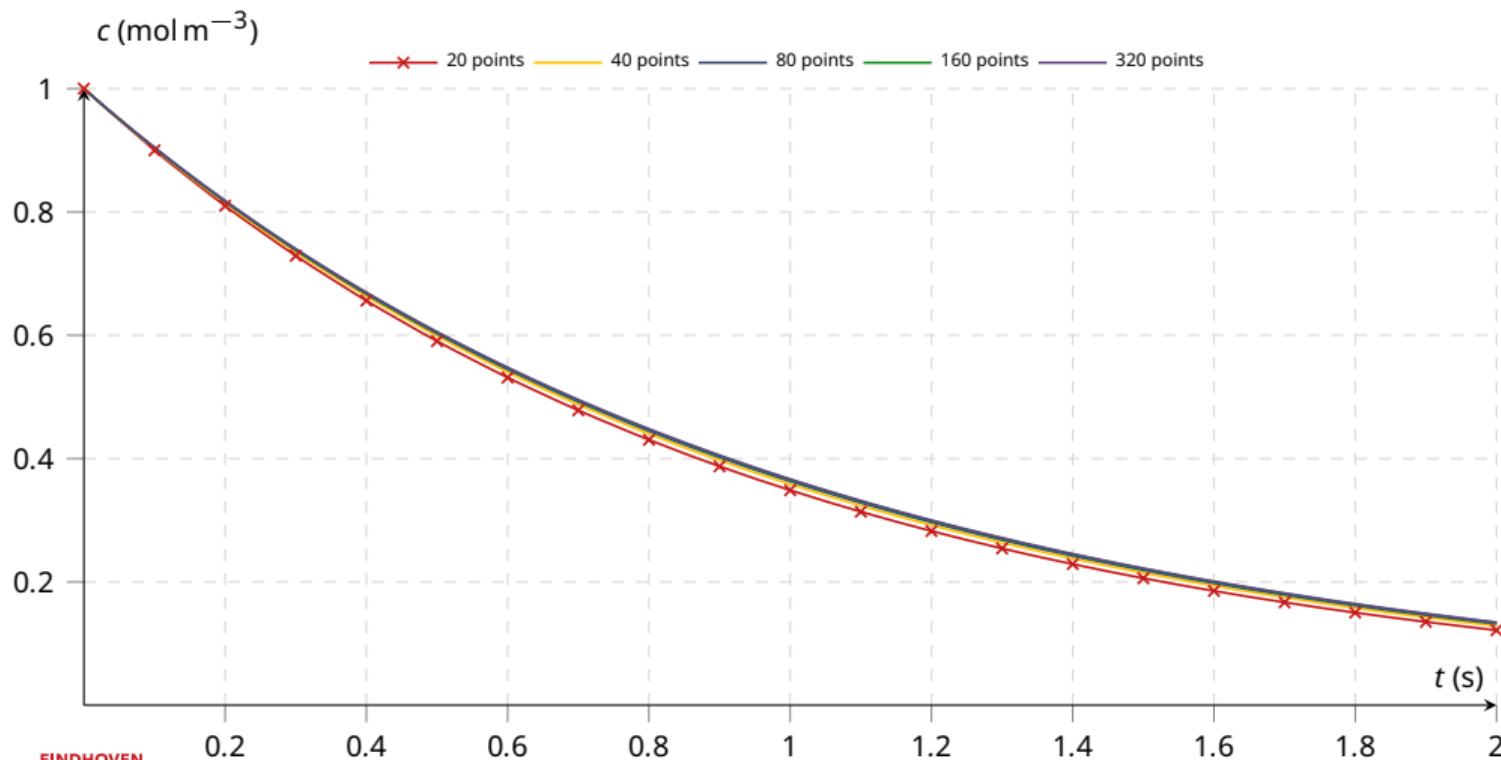
Euler's method - example

First order reaction in a batch reactor:

$$\frac{dc}{dt} = -kc \quad \text{with} \quad c(t=0) = 1 \text{ mol m}^{-3}, \quad k = 1 \text{ s}^{-1}, \quad t_{\text{end}} = 2 \text{ s}$$

Time [s]	Concentration [mol m ⁻³]
$t_0 = 0$	$c_0 = 1.00$
$t_1 = t_0 + \Delta t$ $= 0 + 0.1 = 0.1$	$c_1 = c_0 + \Delta t \cdot (-kc_0)$ $= 1 + 0.1 \cdot (-1 \cdot 1) = 0.9$
$t_2 = t_1 + \Delta t$ $= 0.1 + 0.1 = 0.2$	$c_2 = c_1 + \Delta t \cdot (-kc_1)$ $= 0.9 + 0.1 \cdot (-1 \cdot 0.9) = 0.81$
$t_3 = t_2 + \Delta t$ $= 0.2 + 0.1 = 0.3$	$c_3 = c_2 + \Delta t \cdot (-kc_2)$ $= 0.81 + 0.1 \cdot (-1 \cdot 0.81) = 0.729$
...	...
$t_{i+1} = t_i + \Delta t$... $t_{20} = 2.0$	$c_{i+1} = c_i + \Delta t \cdot (-kc_i)$... $c_{20} = c_{19} + \Delta t \cdot (-kc_{19}) = 0.121577$

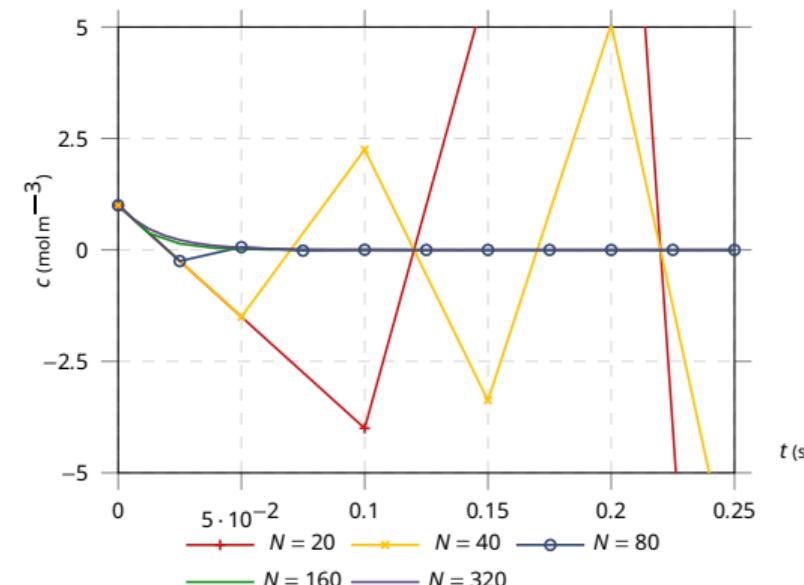
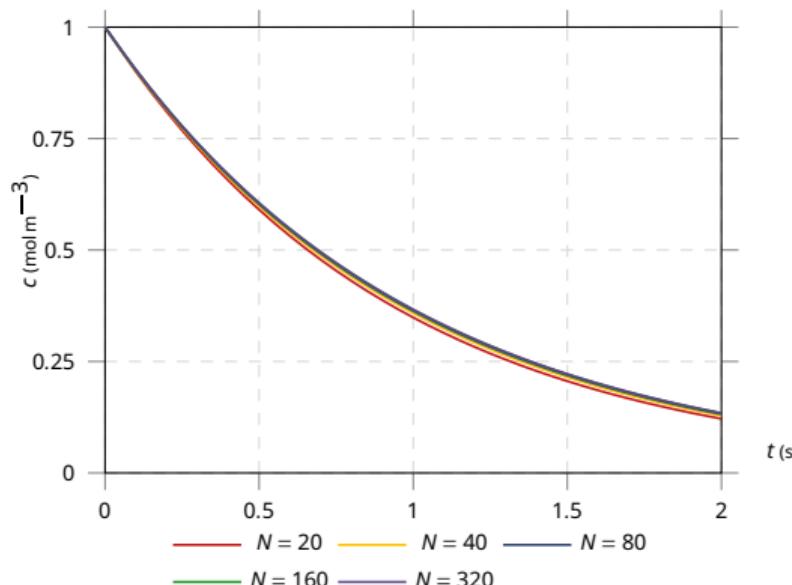
Euler's method - example



Problems with Euler's method

The question is: What step size, or how many steps to use?

- ① **Accuracy** \Rightarrow need information on numerical error!
- ② **Stability** \Rightarrow need information on stability limits!

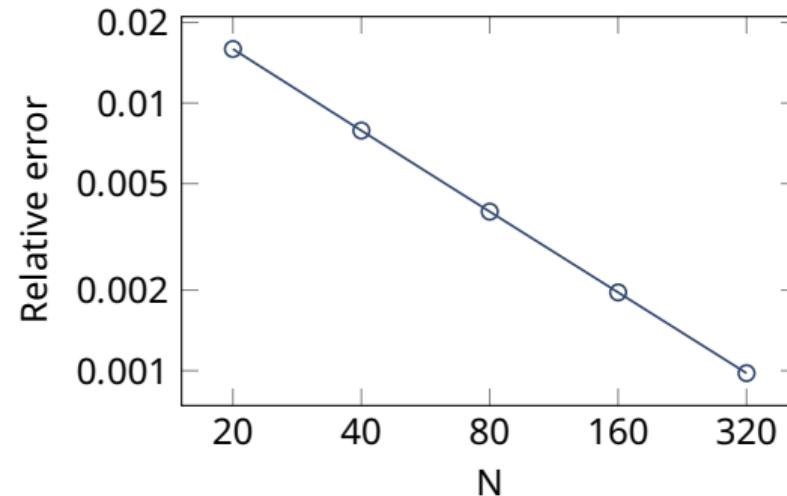


Accuracy

Comparison with analytical solution for $k = 1 \text{ s}^{-1}$:

$$c(t) = c_0 \exp(-kt) \Rightarrow \zeta = 1 - \exp(-kt) \Rightarrow \zeta_{\text{analytical}} = 0.864665$$

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$
20	0.878423	0.015912
40	0.871488	0.007891
80	0.868062	0.003929
160	0.866360	0.001961
320	0.865511	0.000979



Accuracy

For Euler's method: Error halves when the number of grid points is doubled, i.e. error is proportional to Δt : first order method.

Error estimate:

$$\left. \frac{dx}{dt} \right|_{t_0} = \frac{x(t_0 + \Delta t) - x(t_0)}{\Delta t} + \frac{1}{2} \left. \frac{d^2x}{dt^2} \right|_{t_0} (\Delta t) + \mathcal{O}(\Delta t)^2$$

$$\frac{x(t_0 + \Delta t) - x(t_0)}{\Delta t} = f(x_0, t_0) - \frac{1}{2} \left. \frac{d^2x}{dt^2} \right|_{t_0} (\Delta t) + \mathcal{O}(\Delta t)^2$$

Errors and convergence rate

Convergence rate (or: order of convergence) r

$$\epsilon = \lim_{\Delta x \rightarrow 0} c(\Delta x)^r$$

- A first order method reduces the error by a factor 2 when increasing the number of steps by a factor 2
- A second order method reduces the error by a factor 4 when increasing the number of steps by a factor 2

What to do when there is no analytical solution available? Compare to calculations with different number of steps:
 $\epsilon_1 = c(\Delta x_1)^r$ and $\epsilon_2 = c(\Delta x_2)^r$ and solve for r :

$$\frac{\epsilon_2}{\epsilon_1} = \frac{c(\Delta x_2)^r}{c(\Delta x_1)^r} = \left(\frac{\Delta x_2}{\Delta x_1} \right)^r \Rightarrow \log \left(\frac{\epsilon_2}{\epsilon_1} \right) = \log \left(\frac{\Delta x_2}{\Delta x_1} \right)^r$$

$$\Rightarrow r = \frac{\log \left(\frac{\epsilon_2}{\epsilon_1} \right)}{\log \left(\frac{\Delta x_2}{\Delta x_1} \right)} = \frac{\log \left(\frac{\epsilon_2}{\epsilon_1} \right)}{\log \left(\frac{N_1}{N_2} \right)} \text{ in the limit of } \Delta x \rightarrow 0 \quad \text{or} \quad N \rightarrow \infty$$

Today's outline

- Introduction
- Euler's method
 - Forward Euler
- Rates of convergence
- Runge-Kutta methods
 - RK2 methods
 - RK4 method
- Step size control
- Solving ODEs in Python

Errors and convergence rate

L_2 norm (Euclidean norm)

$$\|\mathbf{v}\|_2 = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2} = \sqrt{\sum_{i=1}^n v_i^2}$$

L_∞ norm (maximum norm)

$$\|\mathbf{v}\|_\infty = \max(|v_1|, \dots, |v_n|)$$

Absolute difference

$$\epsilon_{\text{abs}} = \left\| \mathbf{y}_{\text{numerical}} - \mathbf{y}_{\text{analytical}} \right\|_{2,\infty}$$

Relative difference

$$\epsilon_{\text{rel}} = \left\| \frac{\mathbf{y}_{\text{numerical}} - \mathbf{y}_{\text{analytical}}}{\mathbf{y}_{\text{analytical}}} \right\|_{2,\infty}$$

Errors and convergence rate

Convergence rate (or: order of convergence) r

$$\epsilon = \lim_{\Delta x \rightarrow 0} c(\Delta x)^r$$

- A first order method reduces the error by a factor 2 when increasing the number of steps by a factor 2
- A second order method reduces the error by a factor 4 when increasing the number of steps by a factor 2

Computing the rate of convergence

When the analytical solution is available, choose ① or ② for a particular number of grid points N :

- ① Compute the relative or absolute error vector $\bar{\varepsilon}$. Take the norm to compute a single error value ϵ following:

- Based on L_1 -norm: $\epsilon = \frac{\|\bar{\varepsilon}\|_1}{N}$
- Based on L_2 -norm: $\epsilon = \frac{\|\bar{\varepsilon}\|_2}{\sqrt{N}}$
- Based on L_∞ -norm: $\epsilon = \|\bar{\varepsilon}\|_\infty$

- ② Compute the relative or absolute error at a single indicative points (e.g. middle of domain, outlet).

Compare to calculations with different number of steps: $\epsilon_1 = c(\Delta x_1)^r$ and $\epsilon_2 = c(\Delta x_2)^r$ and solve for r :

$$\frac{\epsilon_2}{\epsilon_1} = \frac{c(\Delta x_2)^r}{c(\Delta x_1)^r} = \left(\frac{\Delta x_2}{\Delta x_1} \right)^r \Rightarrow \log \left(\frac{\epsilon_2}{\epsilon_1} \right) = \log \left(\frac{\Delta x_2}{\Delta x_1} \right)^r$$

$$\Rightarrow r = \frac{\log \left(\frac{\epsilon_2}{\epsilon_1} \right)}{\log \left(\frac{\Delta x_2}{\Delta x_1} \right)} = \frac{\log \left(\frac{\epsilon_2}{\epsilon_1} \right)}{\log \left(\frac{N_1}{N_2} \right)} \quad \text{in the limit of } \Delta x \rightarrow 0 \text{ or } N \rightarrow \infty$$

Computing the rate of convergence

When the analytical solution is **not** available:

- ① Compute the solution with $N+1$, N , $N-1$ and $N-2$ grid points
- ② Select a single indicative grid point (e.g. middle of domain, outlet) that lies at exactly the same position in each computation
- ③ Use the solution c at this grid point for various grid sizes to compute:

$$r = \frac{\log \frac{c_{N+1} - c_N}{c_N - c_{N-1}}}{\log \frac{c_N - c_{N-1}}{c_{N-1} - c_{N-2}}}$$

- ④ Alternative for simulations with $2N$, N and $\frac{N}{2}$ grid points:

$$r = \frac{\log \left| \frac{c_{2N} - c_N}{c_N - c_{\frac{N}{2}}} \right|}{\log \left| \frac{N}{2N} \right|}$$

Example: Euler's method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_j}{\epsilon_{j-1}}\right)}{\log\left(\frac{N_{j-1}}{N_j}\right)}$
20	0.878423	0.015912	—
40	0.871488	0.007891	1.011832
80	0.868062	0.003929	1.005969
160	0.866360	0.001961	1.002996
320	0.865511	0.000979	1.001500

⇒ Euler's method is a first order method (as we already knew from the truncation error analysis)

Wouldn't it be great to have a method that can give the answer using much less steps? ⇒ Higher order methods

Today's outline

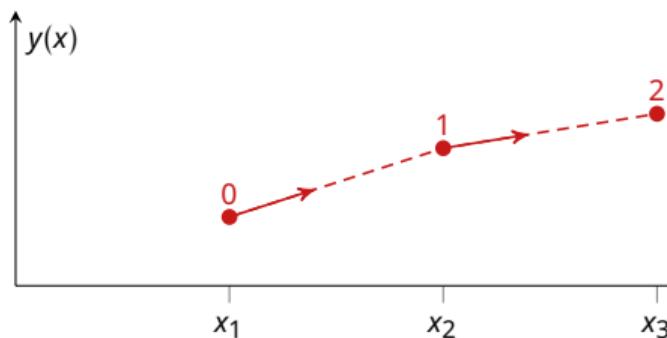
- Introduction
- Euler's method
 - Forward Euler
- Rates of convergence
- Runge-Kutta methods
 - RK2 methods
 - RK4 method
- Step size control
- Solving ODEs in Python

Runge-Kutta methods

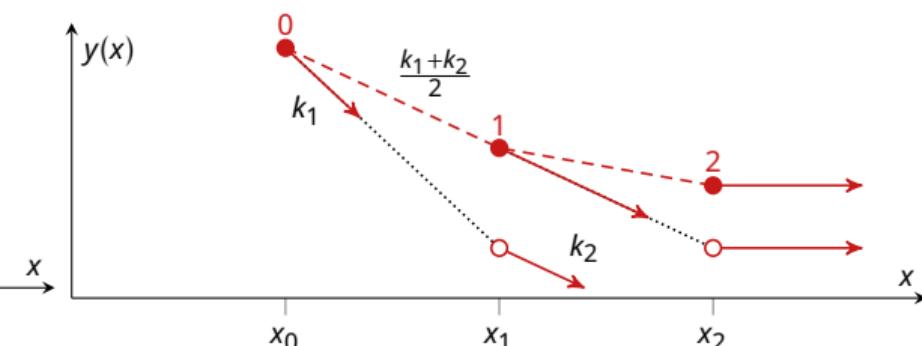
Propagate a solution by combining the information of several Euler-style steps (each involving one function evaluation) to match a Taylor series expansion up to some higher order.

Euler: $y_{i+1} = y_i + hf(x_i, y_i)$ with $h = \Delta x$, i.e. slope = $k_1 = f(x_i, y_i)$.

Euler's method



RK2 method



Classical second order Runge-Kutta (RK2) method

This method is also called Heun's method, or improved Euler method:

- ① Approximate the slope at x_i : $k_1 = f(x_i, y_i)$
- ② Approximate the slope at x_{i+1} : $k_2 = f(x_{i+1}, y_{i+1})$ where we use Euler's method to approximate $y_{i+1} = y_i + hf(x_i, y_i) = y_i + hk_1$
- ③ Perform an Euler step with the average of the slopes: $y_{i+1} = y_i + h \frac{1}{2}(k_1 + k_2)$

In pseudocode:

```
x = x0, y = y0
while x < x_end do
    xi+1 = xi + h
    k1 = f(xi, yi)
    k2 = f(xi + h, yi + hk1)
    yi+1 = yi + h  $\frac{1}{2}(k_1 + k_2)$ 
end while
```

Runge-Kutta methods — derivation

$$\frac{dy}{dx} = f(x, y(x))$$

Using Taylor series expansion: $y_{i+1} = y_i + h \frac{dy}{dx}\Big|_i + \frac{h^2}{2} \frac{d^2y}{dx^2}\Big|_i + \mathcal{O}(h^3)$

$$\left. \frac{dy}{dx} \right|_i = f(x_i, y_i) \equiv f_i$$

$$\frac{d^2y}{dx^2} \Big|_i = \frac{d}{dx} f(x, y(x)) \Big|_i = \frac{\partial f}{\partial x} \Big|_i + \frac{\partial f}{\partial y} \Big|_i \frac{\partial y}{\partial x} \Big|_i = \frac{\partial f}{\partial x} \Big|_i + \frac{\partial f}{\partial y} \Big|_i f_i \quad (\text{chain rule})$$

Substitution gives:

$$y_{i+1} = y_i + h f_i + \frac{h^2}{2} \left(\frac{\partial f}{\partial x} \Big|_i + \frac{\partial f}{\partial y} \Big|_i f_i \right) + \mathcal{O}(h^3)$$

$$y_{i+1} = y_i + \frac{h}{2} f_i + \frac{h}{2} \left(f_i + h \left. \frac{\partial f}{\partial x} \right|_i + h f_i \left. \frac{\partial f}{\partial y} \right|_i \right) + \mathcal{O}(h^3)$$

Runge-Kutta methods — derivation

Note multivariate Taylor expansion:

$$\begin{aligned} f(x_i + h, y_i + k) &= f_i + h \left. \frac{\partial f}{\partial x} \right|_i + k \left. \frac{\partial f}{\partial y} \right|_i + \mathcal{O}(h^2) \\ &\Rightarrow \frac{h}{2} \left(f_i + h \left. \frac{\partial f}{\partial x} \right|_i + h f_i \left. \frac{\partial f}{\partial y} \right|_i \right) = \frac{h}{2} f(x_i + h, y_i + hf_i) + \mathcal{O}(h^3) \end{aligned}$$

Concluding:

$$y_{i+1} = y_i + \frac{h}{2} f_i + \frac{h}{2} f(x_i + h, y_i + hf_i) + \mathcal{O}(h^3)$$

Rewriting:

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + h, y_i + hk_1) \\ \Rightarrow y_{i+1} &= y_i + \frac{h}{2} (k_1 + k_2) \end{aligned}$$

Runge-Kutta methods — derivation

Generalization: $y_{i+1} = y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3)$

with $k_1 = f_i$, $k_2 = f(x_i + c_2 h, y_i + a_{2,1} h k_1)$

(Note that classical RK2: $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$.)

Bivariate Taylor expansion:

$$f(x_i + c_2 h, y_i + a_{2,1} h k_1) = f_i + c_2 h \left. \frac{\partial f}{\partial x} \right|_i + a_{2,1} h k_1 \left. \frac{\partial f}{\partial y} \right|_i + \mathcal{O}(h^2)$$

$$y_{i+1} = y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3)$$

$$= y_i + h \left[b_1 f_i + b_2 f(x_i + c_2 h, y_i + a_{2,1} h k_1) \right] + \mathcal{O}(h^3)$$

$$= y_i + h \left[b_1 f_i + b_2 \left\{ f_i + c_2 h \left. \frac{\partial f}{\partial x} \right|_i + a_{2,1} h k_1 \left. \frac{\partial f}{\partial y} \right|_i + \mathcal{O}(h^2) \right\} \right] + \mathcal{O}(h^3)$$

$$= y_i + h(b_1 + b_2)f_i + h^2 b_2 \left(c_2 \left. \frac{\partial f}{\partial x} \right|_i + a_{2,1} f_i \left. \frac{\partial f}{\partial y} \right|_i \right) + \mathcal{O}(h^3)$$

Comparison with Taylor:

$$y_{i+1} = y_i + h f_i + \frac{h^2}{2} \left(\left. \frac{\partial f}{\partial x} \right|_i + \left. \frac{\partial f}{\partial y} \right|_i f_i \right) + \mathcal{O}(h^3)$$

Using $b_1 + b_2 = 1$, $c_2 b_2 = \frac{1}{2}$, $a_{2,1} b_2 = \frac{1}{2} \Rightarrow 3$ eqns and 4 unknowns \Rightarrow multiple possibilities!

Runge-Kutta methods — derivation

$$y_{i+1} = y_i + h(b_1 + b_2)f_i + h^2 b_2 \left(c_2 \left. \frac{\partial f}{\partial x} \right|_i + a_{2,1} f_i \left. \frac{\partial f}{\partial y} \right|_i \right) + \mathcal{O}(h^3)$$

$$y_{i+1} = y_i + hf_i + \frac{h^2}{2} \left(\left. \frac{\partial f}{\partial x} \right|_i + \left. \frac{\partial f}{\partial y} \right|_i f_i \right) + \mathcal{O}(h^3)$$

⇒ 3 eqns and 4 unknowns ⇒ multiple possibilities!

- ① Classical RK2:

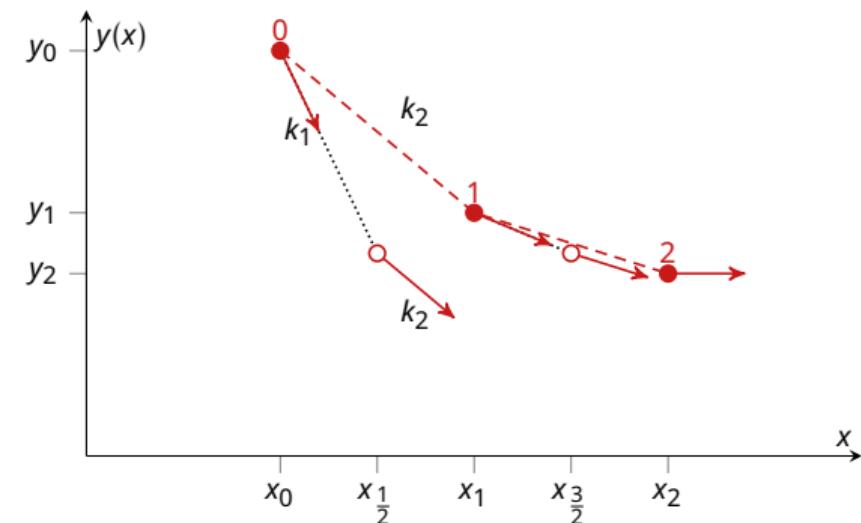
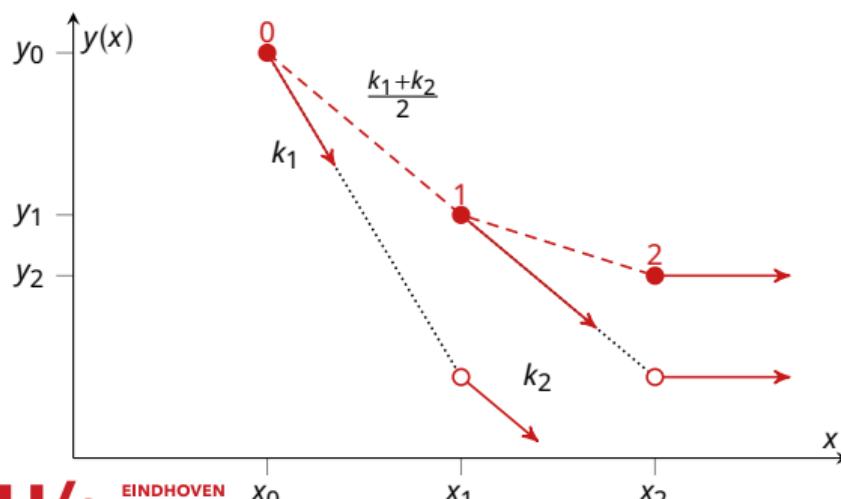
$$b_1 = b_2 = \frac{1}{2} \text{ and } c_2 = a_{2,1} = 1$$

- ② Midpoint rule (modified Euler):

$$b_1 = 0, b_2 = 1, c_2 = a_{2,1} = \frac{1}{2}$$

Second order Runge-Kutta methods

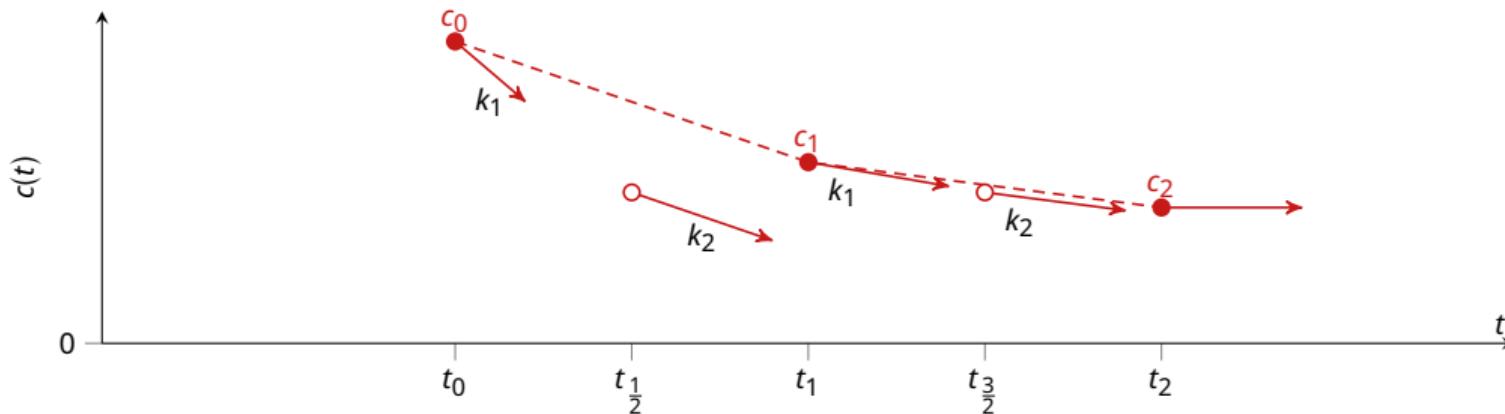
Classical RK2 method (= Heun's method, improved Euler method)	Explicit midpoint rule (modified Euler method)
$k_1 = f_i$	$k_1 = f_i$
$k_2 = f(x_i + h, y_i + hk_1)$	$k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$
$y_{i+1} = y_i + \frac{1}{2}h(k_1 + k_2)$	$y_{i+1} = y_i + hk_2$



Second order Runge-Kutta method — Example

First order reaction in a batch reactor: $\frac{dc}{dt} = -kc$ with $c(t=0) = 1 \text{ mol m}^{-3}$, $k = 1 \text{ s}^{-1}$, $t_{\text{end}} = 2 \text{ s}$.

Time [s]	$C [\text{mol m}^{-3}]$	$k_1 = hf(x_i, y_i)$	$k_2 = hf(x_i + \frac{1}{2}h, y_n + \frac{1}{2}k_1)$
0	1.00	$0.1 \cdot (-1 \cdot 1) = -0.1$	$0.1 \cdot (-1 \cdot (1 - 0.5 \cdot 0.1)) = -0.095$
0.1	$1 - 0.095 = 0.905$	$0.1 \cdot (-1 \cdot 0.905) = -0.0905$	$0.1 \cdot (-1 \cdot (0.905 - 0.5 \cdot 0.0905)) = -0.085975$
...
2	0.1358225	-0.0135822	-0.0129031



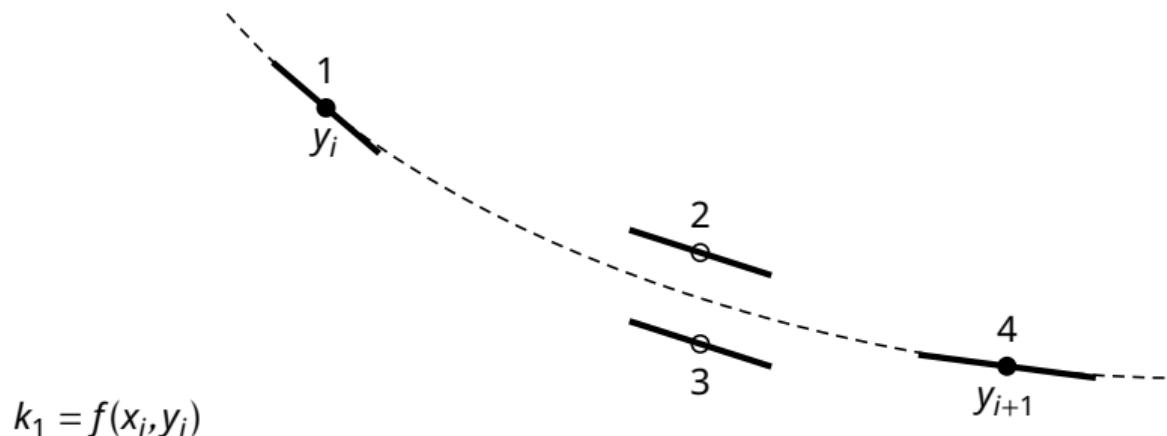
RK2 method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_j}{\epsilon_{j-1}}\right)}{\log\left(\frac{N_{j-1}}{N_j}\right)}$
20	0.864178	5.634×10^{-4}	—
40	0.864548	1.355×10^{-4}	2.056
80	0.864636	3.323×10^{-5}	2.028
160	0.864658	8.229×10^{-6}	2.014
320	0.864663	2.048×10^{-6}	2.007

⇒ RK2 is a second order method. Doubling the number of cells reduces the error by a factor 4!

Can we do even better?

RK4 method (classical fourth order Runge-Kutta method)



$$k_2 = f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1\right)$$

$$k_3 = f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_2\right)$$

$$k_4 = f(x_i + h, y_i + hk_3)$$

$$y_{i+1} = y_i + h \left(\frac{1}{6}k_1 + \frac{1}{3}(k_2 + k_3) + \frac{1}{6}k_4 \right)$$

RK4 method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.864664472	2.836×10^{-7}	—
40	0.864664702	1.700×10^{-8}	4.060
80	0.864664716	1.040×10^{-9}	4.030
160	0.864664717	6.435×10^{-11}	4.015
320	0.864664717	4.001×10^{-12}	4.007

⇒ RK4 is a fourth order method: Doubling the number of cells reduces the error by a factor 16!

Can we do even better?

Today's outline

- Introduction

- Euler's method

- Forward Euler

- Rates of convergence

- Runge-Kutta methods

- RK2 methods
 - RK4 method

- Step size control

- Solving ODEs in Python

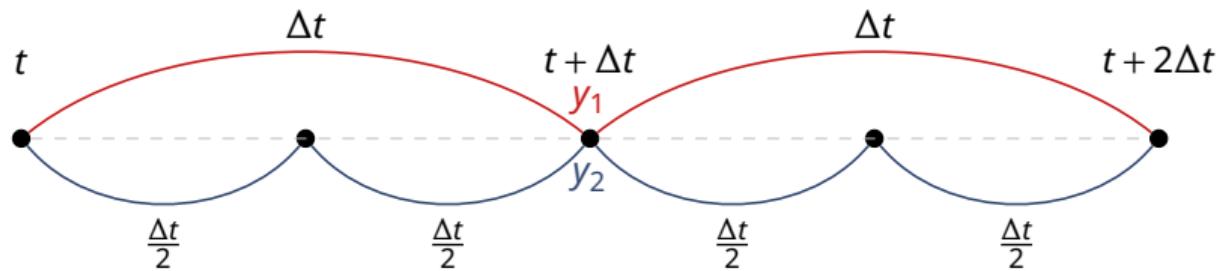
Adaptive step size control

The step size (be it either position, time or both (PDEs)) cannot be decreased indefinitely to favour a higher accuracy, since each additional grid point causes additional computation time. It may be wise to adapt the step size according to the computation requirements.

Globally two different approaches can be used:

- ① Step doubling: compare solutions when taking one full step or two consecutive halve steps
- ② Embedded methods: Compare solutions when using two approximations of different order

Adaptive step size control: step doubling



- RK4 with one large step of h : $y_{i+1} = y_1 + ch^5 + \mathcal{O}(h^6)$
 - RK4 with two steps of $\frac{1}{2}h$: $y_{i+1} = y_2 + 2c(\frac{1}{2}h)^5 + \mathcal{O}(h^6)$

Adaptive step size control: step doubling

- Estimation of truncation error by comparing y_1 and y_2 :

$$\Delta = y_2 - y_1$$

- If Δ too large, reduce step size for accuracy
- If Δ too small, increase step size for efficiency.

- Ignoring higher order terms and solving for c :

$$\Delta = \frac{15}{16}ch^5 \Rightarrow ch^5 = \frac{16}{15}\Delta \Rightarrow y_{i+1} = y_2 + \frac{\Delta}{15} + \mathcal{O}(h^6)$$

(local Richardson extrapolation)

Note that when we specify a tolerance tol , we can estimate the maximum allowable step size as: $h_{\text{new}} = \alpha h_{\text{old}} \left| \frac{\text{tol}}{\Delta} \right|^{\frac{1}{5}}$ with α a safety factor (typically $\alpha = 0.9$).

Adaptive step size control: embedded methods

Use a special fourth and a fifth order Runge Kutta method to approximate y_{i+1}

- The fourth order method is special because we want to use the same positions for the evaluation for computational efficiency.
 - RK45 is the preferred method (minimum number of function evaluations) (this is the default method in `scipy.integrate.solve_ivp`).

Today's outline

- Introduction
- Euler's method
 - Forward Euler
- Rates of convergence
- Runge-Kutta methods
 - RK2 methods
 - RK4 method
- Step size control
- Solving ODEs in Python

Solving ODEs in Python

SciPy provides convenient procedures to solve (systems of) ODEs automatically.

The procedure is as follows:

- ① Create a function that specifies the ODE(s). Specifically, this function returns the $\frac{dy}{dx}$ value (vector).
- ② Initialise solver variables and settings (e.g. step size, initial conditions, tolerance)
- ③ Call the ODE solver function, passing the ODE function as argument
 - The ODE solver will return a solution object (e.g. `sol`), with attribute `sol.t` as the independent variable vector, and a `sol.y` the solution vector (or matrix for systems of ODEs).

Solving ODEs in Python: example 1

We solve the system: $\frac{dx}{dt} = -k_1x + k_2, k_1 = 0.2, k_2 = 2.5$

- Create a lambda function:

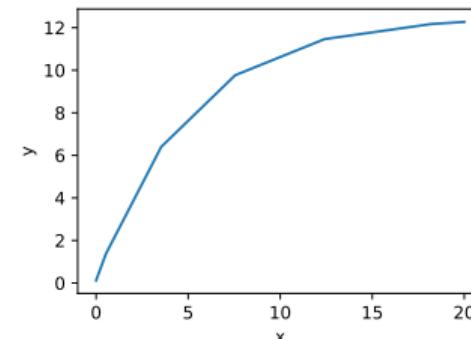
```
1 dydx = lambda x,y: (-0.2*y + 2.5)
```

- Solve with a call to `solve_ivp(function, timespan, initial_condition)`:

```
1 from scipy.integrate import solve_ivp
2 sol = solve_ivp(dydx, tspan, y0)
```

- Draw the results by calling the relevant Matplotlib commands:

```
3 import matplotlib.pyplot as plt
4 plt.plot(sol.t, sol.y[0,:])
5 plt.show()
```



Solving ODEs in Python: example 2

We solve the system: $\frac{dx}{dt} = \begin{cases} -\frac{k_1}{x^2} & t \leq 10 \\ \frac{k_2}{x} - \frac{k_1}{x^2} & t > 10 \end{cases}$ with $k_1 = 0.5$, $k_2 = 1$, $x(0) = 2$

Create an ODE function

```
1 def myEqnFunction(t,x):
2     k1 = 0.5;
3     k2 = 1;
4     dxdt = int(t>10)*k2/x - k1/x**2;
5     return dxdt
```

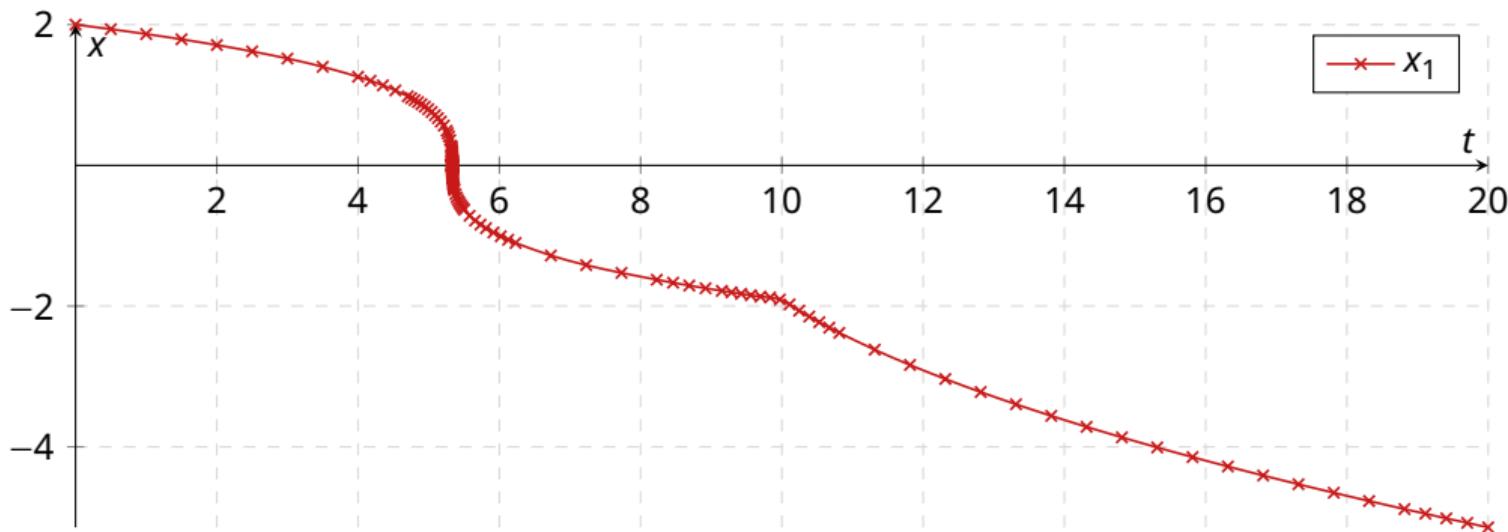
Create a solution script

```
1 tspan = [0, 20]
2 x_init = [2]
3 sol = solve_ivp(myEqnFunction, tspan, x_init, rtol=1e-8, atol=1e-6)
```

Solving ODEs in Python: example 2

Plot the solution:

```
1 plt.plot(sol.t, sol.y[0,:], 'r-x')
2 plt.grid()
3 plt.show()
```



Note the refinement in regions where large changes occur.

Solving ODEs in Python: example

A few notes on working with `scipy.integrate.solve_ivp` and other ODE solvers. If we want to give additional arguments (e.g. `k1` and `k2`) to our ODE function, we can list them in the function line:

```
1 func = lambda t,x,k1,k2: k1*x+k2
2 # or
3 def func(t,x,k1,k2):
4     return k1*x+k2
```

The additional arguments can now be set in the solver script by *adding them as args list*:

```
1 sol = solve_ivp(func,[0,5],[1],args=(k1, k2))
```

Of course, in the solver script, the variables do not have to be called `k1` and `k2`:

```
1 sol = solve_ivp(func,[0,5],[1],args=(q, u))
```

These variables may be of any type (scalar, vector, dictionary, list). For carrying over many variables, a dictionary is useful and descriptive.

Solving systems of ODEs in Python: example

You have noticed that the step size in t varies. This is because we have given just the begin and end times of our time span:

```
1 tspan = [0, 5];
```

You can also obtain the solution at specific points, by supplying a list `t_eval`:

```
1 sol = solve_ivp(func,tspan,[1],args=(some_k1, some_k2),t_eval=np.linspace(tspan[0],tspan[1],31))
```

This example provides 31 explicit time steps between 0 and 5 seconds. Note that the results are interpolated to these data points afterwards; you do not influence the efficiency and accuracy of the solver algorithm this way!

Ordinary differential equations 2

Implicit methods, systems of ODEs and boundary value problems

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

● Introduction

- Backward Euler
 - Implicit midpoint method

● Systems of ODEs

- Solution methods for systems of ODEs
 - Solving systems of ODEs in Python
 - Stiff systems of ODEs

● Boundary value problems

- Shooting method

● Conclusion

Problems with Euler's method: instability

Consider the ODE:

$$\frac{dy}{dx} = f(x, y(x)) \quad \text{with} \quad y(x=0) = y_0$$

First order approximation of derivative: $\frac{dy}{dx} = \frac{y_{i+1} - y_i}{\Delta x}$

Where to evaluate the function f ?

- ① Evaluation at x_i : Explicit Euler method (forward Euler)
 - ② Evaluation at x_{i+1} : Implicit Euler method (backward Euler)

Problems with Euler's method: instability – forward Euler

Explicit Euler method (forward Euler)

- Use values at x_i :

$$\frac{y_{i+1} - y_i}{\Delta x} = f(x_i, y_i) \Rightarrow y_{i+1} = y_i + h f(x_i, y_i).$$
 - This is an explicit equation for y_{i+1} in terms of y_i .
 - It can give instabilities with large function values

Consider the first order batch reactor

$$\frac{dc}{dt} = -kc \Rightarrow c_{i+1} = c_i - k\textcolor{red}{c}_i \Delta t \Rightarrow \frac{c_{i+1}}{c_i} = 1 - k\Delta t$$

It follows that unphysical results are obtained for $k\Delta t > 1$!

Stability requirement

$$k\Delta t <$$

(but probably accuracy requirements are more stringent here!)

Problems with Euler's method: instability – backward Euler

Implicit Euler method (backward Euler)

- Use values at x_{i+1} : $\frac{y_{i+1} - y_i}{\Delta x} = f(x_{i+1}, y_{i+1}) \Rightarrow y_{i+1} = y_i + h f(x_{i+1}, y_{i+1})$.
 - This is an implicit equation for y_{i+1} , because it also depends on terms of y_{i+1} .

Consider the first order batch reactor

$$\frac{dc}{dt} = -kc \Rightarrow c_{i+1} = c_i - k\textcolor{red}{c}_{i+1}\Delta t \Rightarrow \frac{c_{i+1}}{c_i} = \frac{1}{1+k\Delta t}$$

This equation does never give unphysical results

The implicit Euler method is *unconditionally stable* (but maybe not very accurate or efficient).

Semi-implicit Euler method

Usually f is a non-linear function of y , so that linearization is required (recall Newton's method)

$$\frac{dy}{dx} = f(y) \Rightarrow y_{i+1} = y_i + h f(y_{i+1}) \quad \text{using} \quad f(y_{i+1}) = f(y_i) + \left. \frac{df}{dy} \right|_i (y_{i+1} - y_i) + \dots$$

$$\Rightarrow y_{i+1} = y_i + h \left[f(y_i) + \frac{df}{dy} \Big|_i (y_{i+1} - y_i) \right]$$

$$\Rightarrow \left(1 - h \frac{df}{dy} \Big|_j\right) y_{i+1} = \left(1 - h \frac{df}{dy} \Big|_j\right) y_i + hf(y_i)$$

$$\Rightarrow y_{i+1} = y_i + h \left(1 - h \frac{df}{dy} \Big|_i \right)^{-1} f(y_i)$$

For the case that $f(x, y(x))$ we could add the variable x as an additional variable $y_{n+1} = x$. Or add one fully implicit Euler step (which avoids the computation of $\frac{\partial f}{\partial x}$):

$$y_{i+1} = y_i + h f(x_{i+1}, y_{i+1}) \Rightarrow y_{i+1} = y_i + h \left(1 - h \left. \frac{df}{dy} \right|_i \right)^{-1} f(x_{i+1}, y_i)$$

Implicit Euler's method - implementation

A basic function of the implicit Euler method is given in `ode_scalar_implicit.py`:

```
1 def implicit_euler(func, c0, t0, tend, n):
2     h = 1e-8
3     dt = (tend - t0)/n
4     times = np.linspace(t0,tend,n+1)
5     c = np.zeros(n+1)
6     c[0] = c0
7     for i,t in enumerate(times[:-1]):
8         f = func(c[i],t)
9         fh = func(c[i]+h,t)
10        dfdc = (fh - f)/h
11        c[i+1] = c[i] + dt*f/(1 - dt*dfdc)
12        print(f"t={:0.4f}, c: {:.8f}")
13    print(f"t={times[-1]:0.4f}, c: {:.8f}")
14    return times, c
```

```
1 from ode_scalar_implicit import implicit_euler
2 t,c = implicit_euler(lambda c,t: -1.0*c**2, 1, 0, 2,
3                      10)
4 plt.plot(t,c,'-o',label='Implicit Euler')
5 print(f"Conversion = {conv_e}")
```

```
t=0.0000, c: 0.85714286
t=0.2000, c: 0.74772036
t=0.4000, c: 0.66164680
t=0.6000, c: 0.59241445
t=0.8000, c: 0.53566997
t=1.0000, c: 0.48840819
t=1.2000, c: 0.44849689
t=1.4000, c: 0.41438638
t=1.6000, c: 0.38492630
t=1.8000, c: 0.35924657
t=2.0000, c: 0.35924657
Conversion = 0.64075343
```

Semi-implicit Euler method - example

Second order reaction in a batch reactor

$$\frac{dc}{dt} = -kc^2 \text{ with } c_0 = 1 \text{ mol m}^{-3}, k = 1 \text{ m}^3 \text{ mol}^{-1} \text{ s}^{-1}, t_{\text{end}} = 2 \text{ s}$$

Analytical solution: $c(t) = \frac{c_0}{1+kc_0t}$

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc \Rightarrow c_{i+1} = c_i - \frac{hkc_i^2}{1+2hkc_i}$

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.654066262	1.89×10^{-2}	—
40	0.660462687	9.31×10^{-3}	1.02220
80	0.663589561	4.62×10^{-3}	1.01162
160	0.665134433	2.30×10^{-3}	1.00594
320	0.665902142	1.15×10^{-3}	1.00300

Second order implicit method: Implicit midpoint method

Implicit midpoint rule (second order)	Explicit midpoint rule (modified Euler method)
$y_{i+1} = y_i + hf\left(x_i + \frac{1}{2}h, \frac{1}{2}(y_i + y_{i+1})\right)$	$y_{i+1} = y_i + hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$

in case $f(y)$ then:

$$f\left(\frac{1}{2}(y_i + y_{i+1})\right) = f_i + \frac{df}{dy}\Bigg|_i \left(\frac{1}{2}(y_i + y_{i+1}) - y_i\right) = f_i + \frac{1}{2} \frac{df}{dy}\Bigg|_i (y_{i+1} - y_i)$$

Implicit midpoint rule reduces to:

$$\begin{aligned} y_{i+1} &= y_i + hf_i + \frac{h}{2} \frac{df}{dy}\Bigg|_i (y_{i+1} - y_i) \\ &\Rightarrow \left(1 - \frac{h}{2} \frac{df}{dy}\Bigg|_i\right) y_{i+1} = \left(1 - \frac{h}{2} \frac{df}{dy}\Bigg|_i\right) y_i + hf_i \end{aligned}$$

$$\Rightarrow y_{i+1} = y_i + h \left(1 - \frac{h}{2} \frac{df}{dy}\Bigg|_i\right)^{-1} f_i$$

Implicit midpoint method — example

Second order reaction in a batch reactor:

$\frac{dc}{dt} = -kc^2$ with $c_0 = 1 \text{ mol m}^{-3}$, $k = 1 \text{ m}^3 \text{ mol}^{-1} \text{ s}^{-1}$, $t_{\text{end}} = 2 \text{ s}$ (Analytical solution: $c(t) = \frac{c_0}{1+kc_0t}$).

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc$.

Substitution:

$$\begin{aligned} c_{i+1} &= c_i + h \left(1 - \frac{h}{2} \cdot (-2kc_i) \right)^{-1} \cdot (-kc_i^2) \\ &= c_i - \frac{hkc_i^2}{1 + hkc_i} = \frac{c_i + hkc_i^2 - hkc_i^2}{1 + hkc_i} \Rightarrow c_{i+1} = \frac{c_i}{1 + hkc_i} \end{aligned}$$

You will find that this method is exact for all step sizes h because of the quadratic source term!

Implicit midpoint method — example

Second order reaction in a batch reactor:

$$\frac{dc}{dt} = -kc^2 \text{ with } c_0 = 1 \text{ mol m}^{-3}, k = 1 \text{ m}^3 \text{ mol}^{-1} \text{ s}^{-1}, t_{\text{end}} = 2 \text{ s}$$

Analytical solution: $c(t) = \frac{c_0}{1+kc_0t}$

$$c_{i+1} = \frac{c_i}{1+hkc_i}$$

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_j}{\epsilon_{j-1}}\right)}{\log\left(\frac{N_{j-1}}{N_j}\right)}$
20	0.6666666667	1.665×10^{-16}	—
40	0.6666666667	0	—
80	0.6666666667	0	—
160	0.6666666667	0	—
320	0.6666666667	0	—

Implicit midpoint method — example

Third order reaction in a batch reactor: $\frac{dc}{dt} = -kc^3$

Analytical solution: $c(t) = \frac{c_0}{\sqrt{1+2kc_0^2 t}}$

$$c_{i+1} = c_i - \frac{h k c_i^3}{1 + \frac{3}{2} h k c_i^2}$$

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_j}{\epsilon_{j-1}}\right)}{\log\left(\frac{N_{j-1}}{N_j}\right)}$
20	0.5526916174	1.71×10^{-4}	—
40	0.5527633731	4.17×10^{-5}	2.041
80	0.5527807304	1.03×10^{-5}	2.021
160	0.5527849965	2.55×10^{-6}	2.011
320	0.5527860538	6.34×10^{-7}	2.005

Today's outline

● Introduction

- Backward Euler
- Implicit midpoint method

● Systems of ODEs

- Solution methods for systems of ODEs
- Solving systems of ODEs in Python
- Stiff systems of ODEs

● Boundary value problems

- Shooting method

● Conclusion

Systems of ODEs

A system of ODEs is specified using vector notation:

$$\frac{d\mathbf{y}}{dx} = \mathbf{f}(x, \mathbf{y}(x))$$

for

$$\frac{dy_1}{dx} = f_1(x, y_1(x), y_2(x)) \quad \text{or} \quad f_1(x, y_1, y_2)$$

$$\frac{dy_2}{dx} = f_2(x, y_1(x), y_2(x)) \quad \text{or} \quad f_2(x, y_1, y_2)$$

The solution techniques discussed before can also be used to solve systems of equations.

Systems of ODEs: Explicit methods

Forward Euler method

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{f}(x_i, \mathbf{y}_i)$$

Improved Euler method (classical RK2)

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2) \quad \text{using} \quad \begin{aligned} \mathbf{k}_1 &= \mathbf{f}(x_i, \mathbf{y}_i) \\ \mathbf{k}_2 &= \mathbf{f}(x_i + h, \mathbf{y}_i + h\mathbf{k}_1) \end{aligned}$$

Modified Euler method (midpoint rule)

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{k}_2 \quad \text{using} \quad \begin{aligned} \mathbf{k}_1 &= \mathbf{f}(x_i, \mathbf{y}_i) \\ \mathbf{k}_2 &= \mathbf{f}\left(x_i + \frac{h}{2}, \mathbf{y}_i + \frac{h}{2}\mathbf{k}_1\right) \end{aligned}$$

Systems of ODEs: Explicit methods

Classical fourth order Runge-Kutta method (RK4)

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h \left(\frac{\mathbf{k}_1}{6} + \frac{1}{3} (\mathbf{k}_2 + \mathbf{k}_3) + \frac{\mathbf{k}_4}{6} \right)$$

$$\mathbf{k}_1 = \mathbf{f}(x_i, \mathbf{y}_i)$$

$$\mathbf{k}_2 = \mathbf{f}\left(x_i + \frac{h}{2}, \mathbf{y}_i + \frac{h}{2} \mathbf{k}_1\right)$$

using

$$\mathbf{k}_3 = \mathbf{f}\left(x_i + \frac{h}{2}, \mathbf{y}_i + \frac{h}{2} \mathbf{k}_2\right)$$

$$\mathbf{k}_4 = \mathbf{f}(x_i + h, \mathbf{y}_i + h \mathbf{k}_3)$$

Solving systems of ODEs in Python

Solving systems of ODEs in Python is completely analogous to solving a single ODE:

- ① Create a function that specifies the ODEs. This function returns the $\frac{dy}{dx}$ vector.
- ② Initialise solver variables and settings (e.g. step size, initial conditions, tolerance), in a separate script. Initial conditions and tolerances should be given per-equation, i.e. as a vector.
- ③ Call the ODE solver function, using a function argument to the ODE function described in point 1.
 - The ODE solver will return the vector for the independent variable (e.g. time), and a solution matrix, with a column as the solution for each equation in the system.

Solving systems of ODEs in Python: example

We solve the system $\frac{dx_0}{dt} = ax_0 - x_1$, $\frac{dx_1}{dt} = bx_1 + x_0$, with $a = -1$ and $b = -2$:

- Create an ODE function:

```
1 # Example scipy solve_ivp/Example scipy solve_ivp vector.py
2 def func(t, x, a, b):
3     #output can be of list or np.array type:
4     dxdt = np.zeros(2)
5
6     dxdt[0] = a*x[0] - x[1]
7     dxdt[1] = b*x[1] + x[0]
8     return dxdt
```

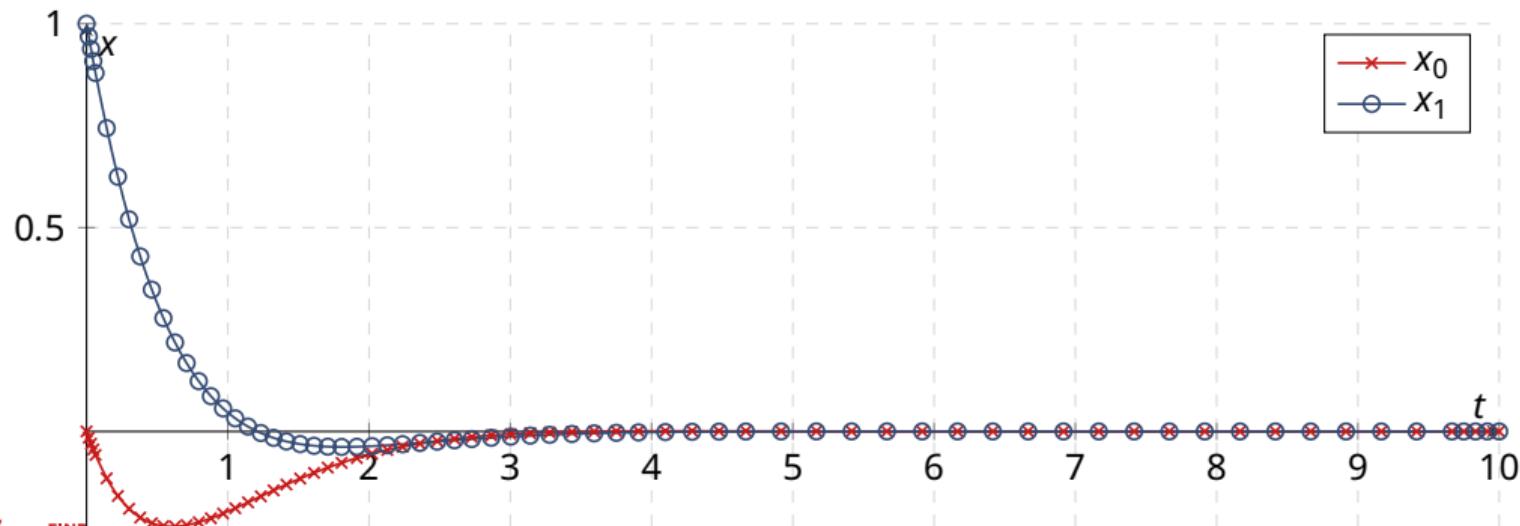
- Solve by calling `solve_ivp`

```
1 from scipy.integrate import solve_ivp
2 x_init = [0,1]; % Initial conditions
3 tspan = [0,10]; % Time span
4 sol = solve_ivp(func, tspan, x_init, args=(-1,-2), rtol=1e-12)
```

Solving systems of ODEs in Python: example

Plot the solution (note: the solution is attribute `sol.y`):

```
1 import matplotlib.pyplot as plt
2 plt.plot(sol.t, sol.y[0], 'r-x', linewidth=2)
3 plt.plot(sol.t, sol.y[1], 'b-o', linewidth=2)
```



Solving ODEs in Python: example

A few notes on working with `scipy.integrate.solve_ivp` and other ODE solvers. If we want to give additional arguments (e.g. `k1` and `k2`) to our ODE function, we can list them in the function line:

```
1 func = lambda t,x,k1,k2: k1*x+k2
2 # or
3 def func(t,x,k1,k2):
4     return k1*x+k2
```

The additional arguments can now be set in the solver script by *adding them as args list*:

```
1 sol = solve_ivp(func,[0,5],[1],args=(k1, k2))
```

Of course, in the solver script, the variables do not have to be called `k1` and `k2`:

```
1 sol = solve_ivp(func,[0,5],[1],args=(q, u))
```

These variables may be of any type (scalar, vector, dictionary, list). For carrying over many variables, a dictionary is useful and descriptive.

Solving systems of ODEs in Python: example

You may have noticed that the step size in t varies. This happens when only the begin and end times of the time span are defined, and `scipy.integrate.solve_ivp` uses adaptive step size for efficiency:

```
1  tspan = [0, 10]
```

You can also retrieve the solution at specific steps, by supplying all steps explicitly as an additional argument to `solve_ivp`, e.g.:

```
1  sol = solve_ivp(func, tspan, x_init, args=(-1,-2), t_eval=np.linspace(0, 10, 101), rtol=1e-12)
```

This example provides 101 explicit time steps between 0 and 10 seconds. It can be useful if you need a direct comparison with e.g. measurements at specific times.

Note that this is an interpolated result. The solver uses, in the background, still the adaptive step size functionality!

Systems of ODEs: Implicit methods

Backward Euler method

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h \left(\mathbf{I} - h \frac{d\mathbf{f}}{d\mathbf{y}} \Big|_i \right)^{-1} \mathbf{f}(\mathbf{y}_i)$$

Implicit midpoint method

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h \left(\mathbf{I} - \frac{h}{2} \frac{d\mathbf{f}}{d\mathbf{y}} \Big|_i \right)^{-1} \mathbf{f}(\mathbf{y}_i)$$

Stiff systems of ODEs

A system of ODEs can be stiff and require a different solution method. For example:

$$\frac{dc_1}{dt} = 998c_1 + 1998c_2 \quad \frac{dc_2}{dt} = -999c_1 - 1999c_2$$

with boundary conditions $c_1(t = 0) = 1$ and $c_2(t = 0) = 0$.

The analytical solution is:

$$c_1 = 2e^{-t} - e^{-1000t} \quad c_2 = -e^{-t} + e^{-1000t}$$

For the explicit method we require $\Delta t < 10^{-3}$ despite the fact that the term is completely negligible, but essential to keep stability.

The “disease” of stiff equations: we need to follow the solution on the shortest length scale to maintain stability of the integration, although accuracy requirements would allow a much larger time step.

Demonstration with example

Forward Euler (explicit)

$$\frac{c_{1,i+1} - c_{1,i}}{dt} = 998c_{1,i} + 1998c_{2,i}$$

$$\frac{c_{2,i+1} - c_{2,i}}{dt} = -999c_{1,i} - 1999c_{2,i}$$

$$\Rightarrow \begin{aligned} c_{1,i+1} &= (1 + 998\Delta t)c_{1,i} + 1998\Delta t c_{2,i} \\ c_{2,i+1} &= -999\Delta t c_{1,i} + (1 - 1999\Delta t)c_{2,i} \end{aligned}$$

Demonstration with example

Backward Euler (implicit)

$$\frac{c_{1,i+1} - c_{1,i}}{\Delta t} = 998c_{1,i+1} + 1998c_{2,i+1}$$

$$\frac{c_{2,i+1} - c_{2,i}}{\Delta t} = -999c_{1,i+1} - 1999c_{2,i+1}$$

$$\Rightarrow (1 - 998\Delta t)c_{1,i+1} - 1998\Delta t c_{2,i} = c_{1,i}$$
$$999\Delta t c_{1,i+1} + (1 + 999\Delta t)c_{2,i+1} = c_{2,i}$$

$$A\mathbf{c}_{i+1} = \mathbf{c}_i \text{ with } A = \begin{pmatrix} 1 - 998\Delta t & -1998\Delta t \\ 999\Delta t & 1 + 1999\Delta t \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} c_{1,i} \\ c_{2,i} \end{pmatrix}$$

Demonstration with example

Backward Euler (implicit) $A\mathbf{c}_{i+1} = \mathbf{c}_i$ with $A = \begin{pmatrix} 1 - 998\Delta t & -1998\Delta t \\ 999\Delta t & 1 + 1999\Delta t \end{pmatrix}$ and $\mathbf{b} = \begin{pmatrix} c_{1,i} \\ c_{2,i} \end{pmatrix}$

Cramers rule:

$$c_{1,i+1} = \frac{\begin{vmatrix} c_{1,i} & -1998\Delta t \\ c_{2,i} & 1 + 1999\Delta t \end{vmatrix}}{\det |A|} = \frac{(1+1999\Delta t)c_{1,i} + 1998\Delta t c_{2,i}}{(1-998\Delta t)(1+1999\Delta t) + 1998 \cdot 999 \Delta t^2}$$

$$c_{2,i+1} = \frac{\begin{vmatrix} 1 - 998\Delta t & c_{1,i} \\ 999\Delta t & c_{2,i} \end{vmatrix}}{\det |A|} = \frac{-999\Delta t c_{1,i} + (1-998\Delta t)c_{2,i}}{(1-998\Delta t)(1+1999\Delta t) + 1998 \cdot 999 \Delta t^2}$$

Forward Euler: $\Delta t \leq 0.001$ for stability

Backward Euler: always stable, even for $\Delta t > 100$ (but then not very accurate!)

Demonstration with example

Cure for stiff problems: use implicit methods! To find out whether your system is stiff: check whether one of the eigenvalues have an imaginary part

Implicit methods in Python

SciPy offers a solver that detects stiff systems, using `method='LSODA'`.

$$\frac{dc_1}{dt} = 998c_1 + 1998c_2 \quad \frac{dc_2}{dt} = -999c_1 - 1999c_2, \quad c_1(0) = 1, \quad c_2(0) = 0$$

- Create the ode function (see `slide_example_solve_ivp_implicit.py`)

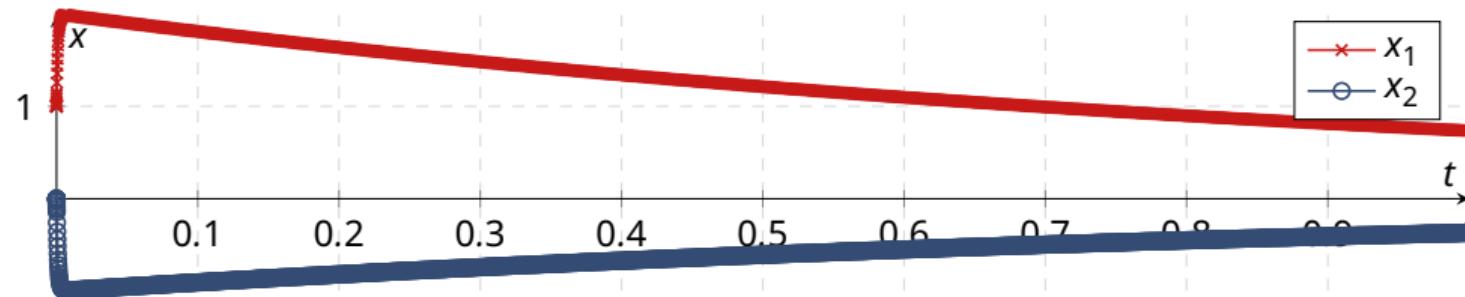
```
1 function [ddcdt] = stiff_ode(t,c)
2 dcdt = zeros(2,1); % Pre-allocation
3 dcdt(1) = 998 * c(1) + 1998*c(2);
4 dcdt(2) = -999 * c(1) - 1999*c(2);
5 return
```

- Compare the resolution of the solutions (see next slide)

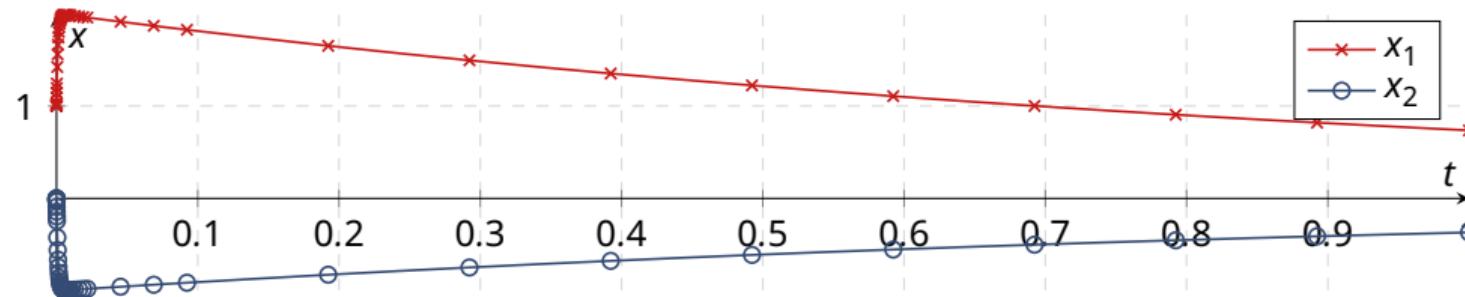
```
1 sol1 = solve_ivp(stiff_ode, [0, 1], [1, 0])
2 # plot sol1
3 sol2 = solve_ivp(stiff_ode, [0, 1], [1, 0], method = 'LSODA')
4 # plot sol2
```

Implicit methods in Python

Default settings



Method: LSODA



The explicit solver requires 1245 data points (default settings), the implicit solver just 48!

Implicit methods in Python: Generic backward Euler

How to make a generic Backward Euler implementation? Recall the update formula:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h \left(\mathbf{I} - h \frac{df}{dy} \Big|_i \right)^{-1} \mathbf{f}(\mathbf{y}_i)$$

- Set up input: Number of steps, end time, initial conditions
- Preallocate and calculate: create a full time vector, calculate the step size h , preallocate \mathbf{y} with zeros and store the initial condition as the first y .
- Loop over the number of iterations:
 - Compute the Jacobian: calculate the function both for y_i as well as for $y_i + s$, where s is a very small number. Recall:

$$\frac{df}{dy} = \frac{f(y+s) - f(y)}{s}$$

- Compute the update formula for y_{i+1} . Use `eye`, `inv`.

Today's outline

● Introduction

- Backward Euler
- Implicit midpoint method

● Systems of ODEs

- Solution methods for systems of ODEs
- Solving systems of ODEs in Python
- Stiff systems of ODEs

● Boundary value problems

- Shooting method

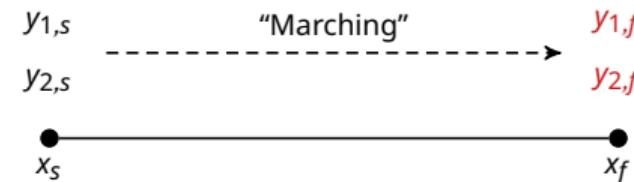
● Conclusion

Importance of boundary conditions

The nature of boundary conditions determines the appropriate numerical method. Classification into 2 main categories:

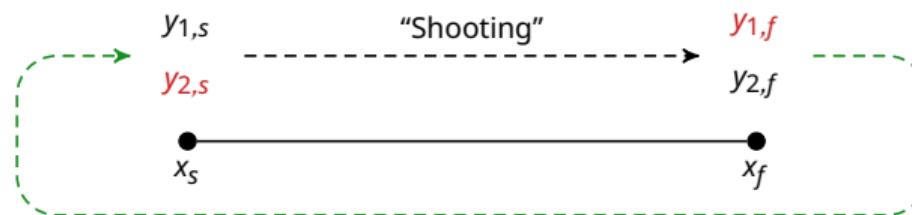
- *Initial value problems (IVP)*

We know the values of all y_i at some starting position x_s , and it is desired to find the values of y_i at some final point x_f .



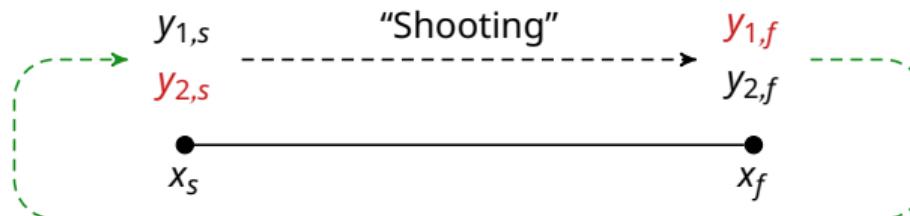
- *Boundary value problems (BVP)*

Boundary conditions are specified at more than one x . Typically, some of the BC are specified at x_s and the remainder at x_f .



Shooting method

How to solve a BVP using the shooting method:



- Define the system of ODEs
- Provide an initial guess for the unknown boundary condition
- Solve the system and compare the resulting boundary condition to the expected value
- Adjust the guessed boundary value, and solve again. Repeat until convergence.
 - Of course, you can subtract the expected value from the computed value at the boundary, and use a non-linear root finding method

BVP: example in Excel

Consider a chemical reaction in a liquid film layer of thickness δ :

$$\mathcal{D} \frac{d^2 c}{dx^2} = k_R c \text{ with } \begin{aligned} c(x=0) &= C_{A,i,L} = 1 && \text{(interface concentration)} \\ c(x=\delta) &= 0 && \text{(bulk concentration)} \end{aligned}$$

Question: compute the concentration profile in the film layer.

Step 1: Define the system of ODEs

This second-order ODE can be rewritten as a system of first-order ODEs, if we define the flux q as:

$$q = -\mathcal{D} \frac{dc}{dx}$$

Now, we find:

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}} q$$

$$\frac{dq}{dx} = -k_R c$$

BVP: example in Excel

Consider a chemical reaction in a liquid film layer of thickness δ :

$$D \frac{d^2c}{dx^2} = k_R c \text{ with } \begin{aligned} c(x=0) &= C_{A,i,L} = 1 && \text{(interface concentration)} \\ c(x=\delta) &= 0 && \text{(bulk concentration)} \end{aligned}$$

Question: compute the concentration profile in the film layer.

Step 2: Set the boundary conditions

The boundary conditions for the concentrations at $x = 0$ and $x = \delta$ are known.

The flux at the interface, however, is not known, and should be solved for.

$$\frac{dc}{dx} = -\frac{1}{D} q$$

$$\frac{dq}{dx} = -k_R c$$

BVP: example in Excel

Solving the two first-order ODEs in Excel. First, the cells with constants:

	A	B	C
1	CAiL	1	mol/m3
2	D	1e-8	m2/s
3	kR	10	1/s
4	delta	1e-4	m
5	N	100	
6	dx	=B4/B5	

$$\frac{dc}{dx} = -\frac{1}{D} q$$
$$\frac{dq}{dx} = -k_R c$$

Now, we program the forward Euler (explicit) schemes for c and q below:

	A	B	C
10	x	c	q
11	0	=B1	10
12	=A11+\$B\$6	=B11+\$B\$6*(-1/\$B\$2*C11)	=C11+\$B\$6*(-\$B\$3*B11)
13	=A12+\$B\$6	=B12+\$B\$6*(-1/\$B\$2*C12)	=C12+\$B\$6*(-\$B\$3*B12)
...
111	=A110+\$B\$6	=B110+\$B\$6*(-1/\$B\$2*C110)	=C110+\$B\$6*(-\$B\$3*B110)

BVP: example in Excel

- We now have profiles for c and q as a function of position x .
- The concentration $c(x = \delta)$ depends (eventually) on the boundary condition at the interface $q(x = 0)$
- We can use the solver to change $q(x = 0)$ such that the concentration at the bulk meets our requirement: $c(x = \delta) = 0$

BVP: example in Python

We first program the system of ODEs in a separate function:

$$\frac{dc}{dx} = -\frac{1}{D}q$$

$$\frac{dq}{dx} = -k_R c$$

```
1 # slides_example_bvp_1.py
2 def diffReactSystem(x, y, param):
3     c, q = y
4     f = np.zeros_like(y)
5     f[0] = -q/param['Diff']
6     f[1] = -param['kR']*c
7     return f
```

Note that we pass a variable (type: dictionary) that contains required parameters: `param`.

BVP: example in Python

Let's first try to solve the ODE system using `scipy.integrate.solve_ivp`:

```
1 # slides_example_bvp_1.py
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from scipy.integrate import solve_ivp
5
6 ### Definition of diffReactSystem here (see slide 433 )
7
8 # Set up parameters
9 q0 = 1e-3 # Initial guess flux@t=0
10 param = {'cAiL': 1.0,'Diff':1e-8,'kR': 10,'delta': 1e-4,'N': 100}
11
12 # Solve ODE system
13 sol = solve_ivp(lambda x, y: diffReactSystem(x, y, param), # ODE func with params
14                  [0, param['delta']], # Time span
15                  [param['cAiL'], q0]) # Initial conditions
16
17 fig,ax1 = plt.subplots()
18 ax1.plot(sol.t,sol.y[0,:],'-b',label='Concentration $mol/m^3$')
19 ax2 = ax1.twinx() # Create y-y axis
20 ax2.plot(sol.t,sol.y[1,:],'-r',label='Flux $mol/m^2/s$')
21 fig.legend(bbox_to_anchor=(0.5, 0.5))
22 plt.show()
```

BVP: example in Python

That seems to work! Now we want to fit the value for q at $x = 0$ (defined below as `bcq`), such that the concentration at $x = \delta$ equals zero. We create a function with the output defined as the deviation from the target value:

```
1 # slides_example_bvp_2.py
2 def diffReactFitCriterium(bcq, param):
3     # Solve the ODE system using changeable parameter bcq
4     # (boundary condition for q), other parameters are defined in param
5     sol = solve_ivp(lambda x, y: diffReactSystem(x, y, param), [0, param['delta']], [param['cAiL']
6         ], bcq)
7     # Return the last value of the concentration (column 0 in y) at x=delta (hence [-1])
8     return sol.y[0,-1] - 0
```

Note the following:

- We use the interval $0 \leq x \leq \delta$
- Boundary conditions are given as: $c(x = 0) = 1$ and $q(x = 0) = bcq$, which is given as a separate argument to the function (i.e. changeable from 'outside'!)
- The function returns the concentration at $x = \delta$

BVP: example in Python

Finally, we should solve the system so that we obtain the right boundary condition $q = bcq$ such that $c(x = \delta) = 0$. We can use the `scipy.optimize.root_scalar` function to do this. Extend the script from slide 434 by:

```
1 # slides_example_bvp_2.py
2 from scipy.optimize import root_scalar
3
4 ### Define diffReactSystem, diffReactFitCriterium, parameters
5
6 # Solve such that c(delta)=0:
7 sol = root_scalar(lambda x: diffReactFitCriterium(x, param),
8                     method='brentq', bracket=[0,1], xtol=1e-15, rtol=1e-15)
9 q0 = sol.root
10 print(f"q0 = {q0}")
11
12 # Solve ODE once more such that we can plot the final data
13 sol = solve_ivp(lambda x, y: diffReactSystem(x, y, param),
14                  [0, param['delta']], [param['cAiL'], q0],
15                  t_eval = np.linspace(0, param['delta'], 101))
```

Postprocessing of the data can be done similar to the example in slide 434.

BVP example: analytical solution

Compare with the analytical solution:

$$q = k_L E_A C_{A,i,L} \quad \text{with}$$

$$E_A = \frac{\text{Ha}}{\tanh \text{Ha}} \quad (\text{Enhancement factor})$$

$$\text{Ha} = \frac{\sqrt{k_R D}}{k_L} \quad (\text{Hatta number})$$

$$k_L = \frac{D}{\delta} \quad (\text{mass transfer coefficient})$$

Today's outline

● Introduction

- Backward Euler
- Implicit midpoint method

● Systems of ODEs

- Solution methods for systems of ODEs
- Solving systems of ODEs in Python
- Stiff systems of ODEs

● Boundary value problems

- Shooting method

● Conclusion

Other methods

Other explicit methods:

- Bulirsch-Stoer method (Richardson extrapolation + modified midpoint method)

Other implicit methods:

- Rosenbrock methods (higher order implicit Runge-Kutta methods)
- Predictor-corrector methods

Summary

- Several solution methods and their derivation were discussed:
 - Explicit solution methods: Euler, Improved Euler, Midpoint method, RK45
 - Implicit methods: Implicit Euler and Implicit midpoint method
 - A few examples of their spreadsheet implementation were shown
- We have paid attention to accuracy and instability, rate of convergence and step size
- Systems of ODEs can be solved by the same algorithms. Stiff problems should be treated with care.
- An example of solving ODEs with Python was demonstrated.

Partial differential equations

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

● Introduction

● Instationary diffusion equation

- Discretization
- Solving the diffusion equation
- Non-linear source terms

● Convection

- Discretization
- Central difference scheme
- Upwind scheme

● Conclusions

- Other methods
- Summary

Overview

Main question

How to solve parabolic PDEs like:

$$\frac{\partial c}{\partial t} = \mathcal{D} \frac{\partial^2 c}{\partial x^2} - u \frac{\partial c}{\partial x} + R$$

$$t = 0; 0 \leq x \leq \ell \Rightarrow c = c_0$$

with

$$t > 0; x = 0 \quad \Rightarrow -D \frac{\partial c}{\partial x} + uc = u_{in}c_{in}$$

$$t > 0; x = \ell \quad \Rightarrow \frac{\partial c}{\partial x} = 0$$

accurately and efficiently?

What is a PDE?

Partial differential equation

An equation containing a function and their derivatives to multiple independent variables.

Order of PDE

The highest derivative appearing in the PDE

General second order PDE:

$$A \frac{\partial^2 f}{\partial x^2} + B \frac{\partial^2 f}{\partial x \partial y} + C \frac{\partial^2 f}{\partial y^2} + D \frac{\partial f}{\partial x} + E \frac{\partial f}{\partial y} + Ff = G$$

- Linear equation: Coefficients A, B, \dots, G do not depend on x and y .
 - Non-linear equation: Coefficients A, B, \dots, G are a function of x and y .

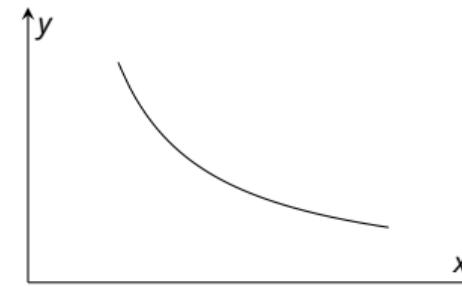
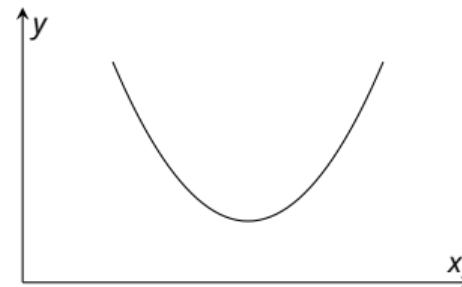
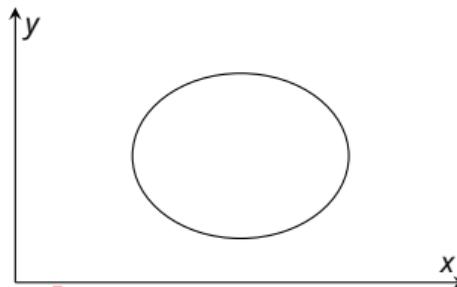
Classification of PDE's

$$A \frac{\partial^2 f}{\partial x^2} + B \frac{\partial^2 f}{\partial x \partial y} + C \frac{\partial^2 f}{\partial y^2} + D \frac{\partial f}{\partial x} + E \frac{\partial f}{\partial y} + Ff = G$$

The discriminant Δ of a quadratic polynomial is computed as (note: only the higher order coefficients are important):

$$\Delta = B^2 - 4AC$$

- $\Delta < 0 \Rightarrow$ Elliptic equation
(e.g. Laplace equation for stationary diffusion in 2D)
 - $\Delta = 0 \Rightarrow$ Parabolic equation
(e.g. instationary heat penetration in 1D)
 - $\Delta > 0 \Rightarrow$ Hyperbolic equation
(e.g. wave equation)

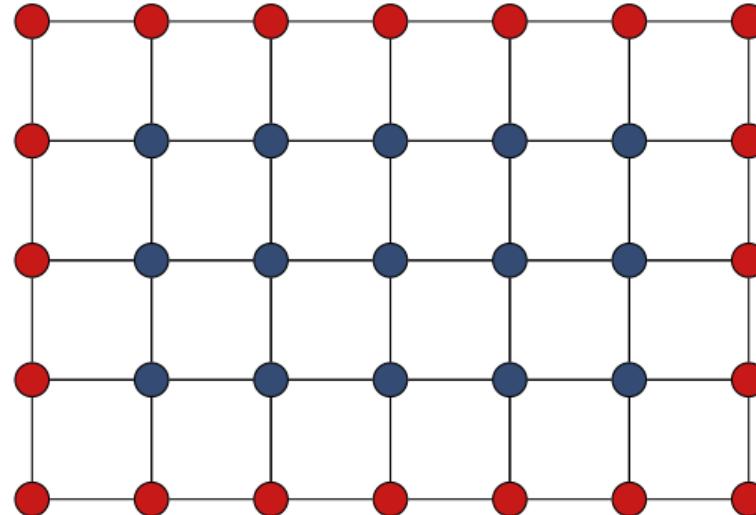


Importance of classification

Different PDE types require different solution techniques because of the difference in range of influence:

- *Characteristics*
Curves in xy -domain along with signal propagation takes place
- *Domain of dependence of point P*
points in xy -domain which influence the value of f in point P
- *Range of influence of point P*
points in xy -domain which are influenced by the value of f in point P

Example elliptic PDE (boundary value problems: BVP)



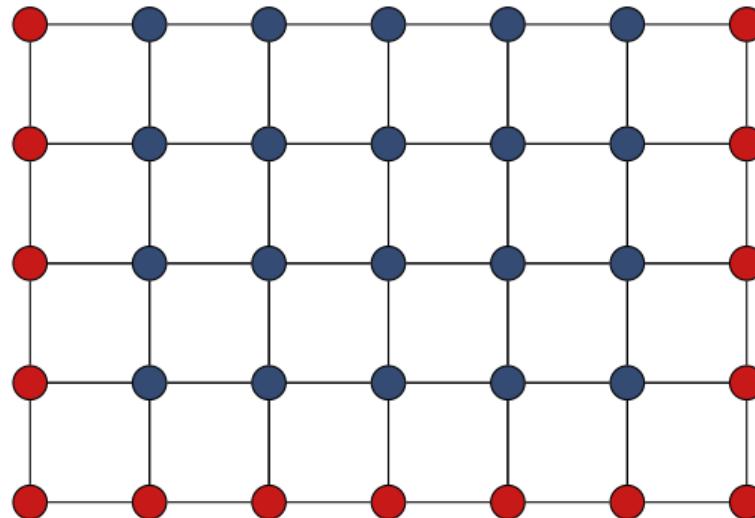
- Grid point at which dependent variable has to be computed
- Grid point at which boundary condition is specified

Typical example: Poisson equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = f(x, y)$$

Efficiency (memory requirements, CPU time) of the numerical method is of crucial importance.

Example parabolic PDE (initial value problem: IVP)



- Grid point at which dependent variable has to be computed
- Grid point at which boundary condition is specified

Typical example: Poisson equation

$$\frac{\partial c}{\partial t} + u \frac{\partial c}{\partial x} = \mathcal{D} \frac{\partial^2 c}{\partial x^2} + R$$

Stability (in numerical sense) of the numerical method is of crucial importance.

Boundary conditions

- Dirichlet or fixed condition: prescribed value of f at boundary

$$f = f_0 \quad f_0 \text{ is a known function}$$

- Neumann condition: prescribed value of derivative of f at boundary

$$\frac{\partial f}{\partial n} = q \quad q \text{ is a known function}$$

- Mixed or Robin condition: relation between f and $\frac{\partial f}{\partial n}$ at boundary

$$a \frac{\partial f}{\partial n} + bf = c \quad a, b \text{ and } c \text{ are known functions}$$

Numerical solution method

Finite differences (method of lines, MOL):

- ① Discretize spatial domain in discrete grid points
- ② Find suitable approximation for the spatial derivatives
- ③ Substitute approximations in PDE, which gives a system of ODE's, one for every grid points
- ④ Advance in time with a suitable ODE solver

Alternative methods: collocation, Galerkin or Finite Element methods

Today's outline

● Introduction

● Instationary diffusion equation

- Discretization
- Solving the diffusion equation
- Non-linear source terms

● Convection

- Discretization
- Central difference scheme
- Upwind scheme

● Conclusions

- Other methods
- Summary

Instationary diffusion equation (Fick's second law)

$$\frac{\partial c}{\partial t} = D \frac{\partial^2 c}{\partial x^2}, \quad \text{with} \quad \begin{aligned} t = 0; 0 \leq x \leq \ell &\Rightarrow c = c_0 \\ t > 0; x = 0 &\Rightarrow c = c_L \\ t > 0; x = \ell &\Rightarrow c = c_R \end{aligned}$$

Second derivative $\frac{\partial^2 c}{\partial x^2}$

$$c_{i+1} = c_i + \left. \frac{\partial c}{\partial x} \right|_i \Delta x + \frac{1}{2} \left. \frac{\partial^2 c}{\partial x^2} \right|_i \Delta x^2 + \frac{1}{6} \left. \frac{\partial^3 c}{\partial x^3} \right|_i \Delta x^3 + \dots$$

$$c_{i-1} = c_i - \left. \frac{\partial c}{\partial x} \right|_i \Delta x + \frac{1}{2} \left. \frac{\partial^2 c}{\partial x^2} \right|_i \Delta x^2 - \frac{1}{6} \left. \frac{\partial^3 c}{\partial x^3} \right|_i \Delta x^3 + \dots$$

$$c_{i+1} + c_{i-1} = 2c_i + \left. \frac{\partial^2 c}{\partial x^2} \right|_i \Delta x^2 + \mathcal{O}(\Delta x^4)$$

$$\Rightarrow \left. \frac{\partial^2 c}{\partial x^2} \right|_i = \frac{c_{i+1} - 2c_i + c_{i-1}}{\Delta x^2} + \mathcal{O}(\Delta x^2)$$

Due to symmetric discretization: second order (central discretization).

Instationary diffusion equation (Fick's second law)

An alternative discretization:

$$\frac{\partial^2 c}{\partial x^2} \Big|_i = \frac{\frac{\partial c}{\partial x} \Big|_{i+\frac{1}{2}} - \frac{\partial c}{\partial x} \Big|_{i-\frac{1}{2}}}{\Delta x} + \mathcal{O}(\Delta x^2)$$


$$= \frac{\frac{c_{i+1} - c_i}{\Delta x} - \frac{c_i - c_{i-1}}{\Delta x}}{\Delta x} = \frac{c_{i+1} - 2c_i + c_{i-1}}{\Delta x^2}$$

This is convenient for the derivation of $\frac{\partial}{\partial x} \left(D \frac{\partial c}{\partial x} \right)$:

$$\begin{aligned} \frac{\partial}{\partial x} \left(D \frac{\partial c}{\partial x} \right) &= \frac{D_{i+\frac{1}{2}} \frac{\partial c}{\partial x} \Big|_{i+\frac{1}{2}} - D_{i-\frac{1}{2}} \frac{\partial c}{\partial x} \Big|_{i-\frac{1}{2}}}{\Delta x} = \frac{D_{i+\frac{1}{2}} \frac{c_{i+1} - c_i}{\Delta x} - D_{i-\frac{1}{2}} \frac{c_i - c_{i-1}}{\Delta x}}{\Delta x} \\ &= \frac{D_{i+\frac{1}{2}} c_{i+1} - (D_{i+\frac{1}{2}} + D_{i-\frac{1}{2}}) c_i + D_{i-\frac{1}{2}} c_{i-1}}{(\Delta x)^2} \end{aligned}$$

Instationary diffusion equation (Fick's second law)

$$\frac{\partial^2 f}{\partial x^2} \quad i-1 \quad i-\frac{1}{2} \quad i \quad i+\frac{1}{2} \quad i+1$$

$$f_{i+\frac{1}{2}} = f_i + \frac{1}{2} \Delta x \left. \frac{\partial f}{\partial x} \right|_i \Delta x + \frac{1}{2} \left(\frac{1}{2} \Delta x \right)^2 \left. \frac{\partial^2 f}{\partial x^2} \right|_i + \mathcal{O}(\Delta x^3)$$

$$f_{i-\frac{1}{2}} = f_i - \frac{1}{2} \Delta x \left. \frac{\partial f}{\partial x} \right|_i \Delta x + \frac{1}{2} \left(\frac{1}{2} \Delta x \right)^2 \left. \frac{\partial^2 f}{\partial x^2} \right|_i + \mathcal{O}(\Delta x^3)$$

$$f_{i+\frac{1}{2}} - f_{i-\frac{1}{2}} = \Delta x \left. \frac{\partial f}{\partial x} \right|_i + \mathcal{O}(\Delta x^3)$$

$$\Rightarrow \left. \frac{\partial f}{\partial x} \right|_i = \frac{f_{i+\frac{1}{2}} - f_{i-\frac{1}{2}}}{\Delta x} + \mathcal{O}(\Delta x^2)$$

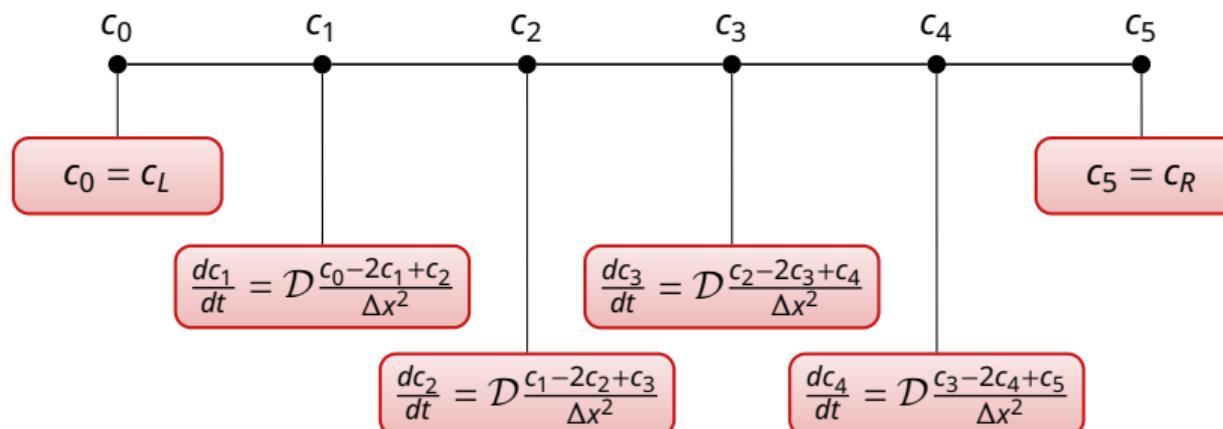
Symmetric discretization yields second order!

Instationary diffusion equation: spatial discretization

Substitution of spatial derivatives yields:

$$\frac{dc_i}{dt} = \mathcal{D} \frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2} \quad \text{for } i = 0, \dots, N$$

For example, using 6 (ridiculously low number!) grid points:



Instationary diffusion equation: boundary conditions

Two options:

- ① Keep boundary conditions as additional equations:

$$c_0 = c_L, \frac{dc_1}{dt} = D \frac{c_0 - 2c_1 + c_2}{\Delta x^2}, \frac{dc_2}{dt} = D \frac{c_1 - 2c_2 + c_3}{\Delta x^2},$$
$$\frac{dc_3}{dt} = D \frac{c_2 - 2c_3 + c_4}{\Delta x^2}, \frac{dc_4}{dt} = D \frac{c_3 - 2c_4 + c_5}{\Delta x^2}, c_5 = c_R$$

- ② Substitute boundary conditions to reduce number of equations:

$$\frac{dc_1}{dt} = D \frac{c_L - 2c_1 + c_2}{\Delta x^2}, \frac{dc_2}{dt} = D \frac{c_1 - 2c_2 + c_3}{\Delta x^2},$$
$$\frac{dc_3}{dt} = D \frac{c_2 - 2c_3 + c_4}{\Delta x^2}, \frac{dc_4}{dt} = D \frac{c_3 - 2c_4 + c_R}{\Delta x^2}$$

Instationary diffusion equation: temporal discretization

$$\frac{dc_i}{dt} = \mathcal{D} \frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2}$$

Time discretization: forward Euler (explicit)

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = \mathcal{D} \frac{c_{i-1}^n - 2c_i^n + c_{i+1}^n}{\Delta x^2}$$

$$\Rightarrow c_i^{n+1} = \text{Fo} c_{i-1}^n + (1 - 2\text{Fo}) c_i^n + \text{Fo} c_{i+1}^n \quad \text{with } \text{Fo} = \frac{\mathcal{D} \Delta t}{\Delta x^2}$$

Straightforward updating (explicit equation), simple to implement in a program but stability constraint $\text{Fo} = \frac{\mathcal{D} \Delta t}{\Delta x^2} < \frac{1}{2}$!

Small $\Delta x \Rightarrow$ small $\Delta t \Rightarrow$ patience required ☺

Instationary diffusion equation: temporal discretization

$$\frac{dc_i}{dt} = \mathcal{D} \frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2}$$

Time discretization: backward Euler (implicit)

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = \mathcal{D} \frac{c_{i-1}^{n+1} - 2c_i^{n+1} + c_{i+1}^{n+1}}{\Delta x^2}$$

$$\Rightarrow -\text{Fo}c_{i-1}^{n+1} + (1 + 2\text{Fo})c_i^{n+1} - \text{Fo}c_{i+1}^{n+1} = c_i^n \quad \text{with } \text{Fo} = \frac{\mathcal{D}\Delta t}{\Delta x^2}$$

Requires the solution of a system of linear equations, but no stability constraints!

Note: extension to higher order schemes (with time step adaptation) straightforward. Often second or third order optimal, because for each Euler-like step in the additional order an often large system needs to be solved (not treated in this course).

Solving the instationary diffusion equation: example

Solve the diffusion problem using explicit discretization:

$$\frac{\partial c_i}{\partial t} = \mathcal{D} \frac{\partial^2 c}{\partial x^2} \quad \text{with} \quad \begin{aligned} 0 &\leq x \leq \delta, \delta = 5 \cdot 10^{-3} \text{ m} \\ \delta/\Delta x &= 100 \text{ grid cells} \\ \mathcal{D} &= 1 \cdot 10^{-8} \text{ m}^2 \text{s}^{-1} \\ t_{\text{end}} &= 5000 \text{ s} \\ c_L &= 1 \text{ mol m}^{-3}, c_R = 0 \text{ mol m}^{-3} \end{aligned}$$

$$c_i^{n+1} = Foc_{i-1}^n + (1 - 2Fo)c_i^n + Foc_{i+1}^n \quad \text{with } Fo = \frac{\mathcal{D}\Delta t}{\Delta x^2}$$

- ① Initialise variables
- ② Compute time step so that $Fo \leq \frac{1}{2} \Rightarrow \Delta t = 0.125\text{s}!$
- ③ Compute 40000 time steps times 100 grid nodes!
- ④ Store solution

Solving the instationary diffusion equation: example

Initialise the variables and matrices:

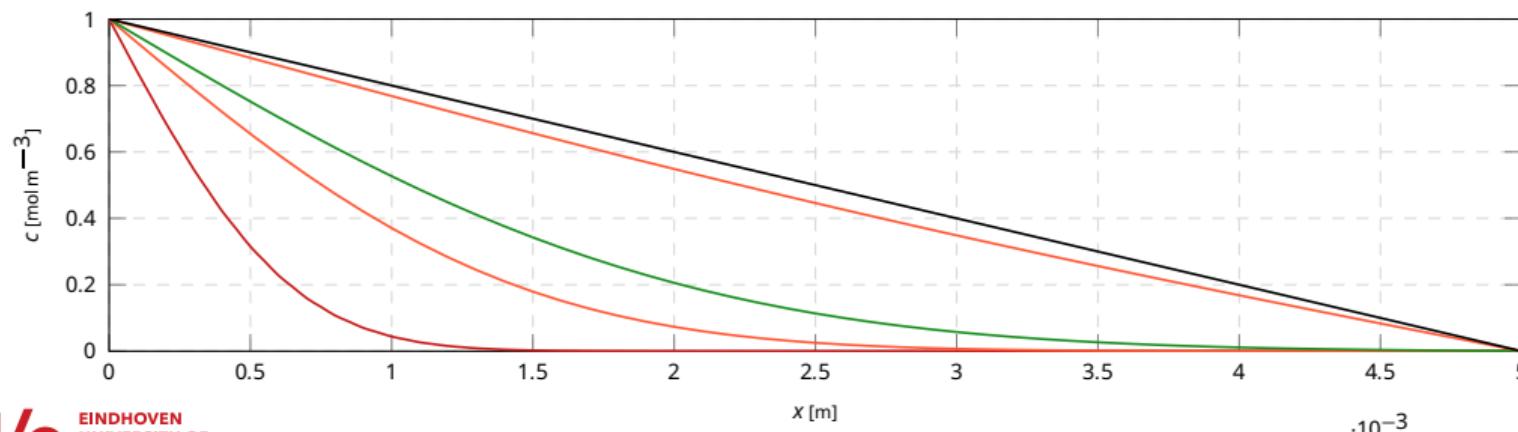
```
1 import numpy as np
2
3 Nx = 100 # Nx grid points
4 Nt = 40000 # Nt time steps
5 D = 1e-8 # m/s
6 c_L = 1.0; c_R = 0 # mol/m3
7 t_end = 5000.0 # s
8 x_end = 5e-3 # m
9
10 # Time step and grid size
11 dt = t_end / Nt
12 dx = x_end / Nx
13
14 # Fourier number
15 Fo = D * dt / dx / dx
16
17 # Initial matrices for solutions (Nx times Nt)
18 c = np.zeros((Nt + 1, Nx + 1)) # All concentrations are zero
19 c[:, 0] = c_L # Concentration at the left side
20 c[:, Nx] = c_R # Concentration at the right side
21
22 # Grid node and time step positions
23 x = np.linspace(0, x_end, Nx + 1)
```

Solving the instationary diffusion equation: example

Compute the solution (nested time-and-grid loop):

- Create a time-loop
- Create a loop over *internal* grid points
- Update each node using $c_i^{n+1} = Foc_{i-1}^n + (1 - 2Fo)c_i^n + Foc_{i+1}^n$
- Plot the solution for selected time steps

Plotting the solution at $t = \{12.5, 62.5, 125, 625, 5000\}$ s.



Solving the instationary diffusion equation: example

A double-loop can impose serious computation times if the number of grid points increases:

```
1 for n in range(Nt - 1): # time loop
2     for i in range(1, Nx): # Nested loop for grid nodes
3         c[n+1, i] = Fo * c[n, i-1] + (1 - 2*Fo) * c[n, i] + Fo * c[n, i+1]
```

Remedy: vectorization. Construct a 3-point stencil Laplacian matrix first, then use the matrix product to evolve the simulation:

```
1 from scipy.sparse import diags
2
3 # Construct sparse matrix
4 e = np.ones(Nx-1)
5 md = np.concatenate(([1], (1 - 2 * Fo) * e, [1]))
6 ld = np.concatenate((Fo * e, [0]))
7 ud = np.concatenate(([0], Fo * e))
8 A = diags([ld, md, ud], offsets=[-1, 0, 1])
9
10 # Time evolution
11 for n in range(Nt - 1): # time loop
12     c[n+1, :] = A.dot(c[n,:])
```

Solving the diffusion equation implicitly

Linear system $Ax = b$ from $-Foc_{i-1}^{n+1} + (1 + 2Fo)c_i^{n+1} - Foc_{i+1}^{n+1} = c_i^n$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -Fo & (1 + 2Fo) & -Fo & 0 & \cdots & 0 \\ 0 & -Fo & (1 + 2Fo) & -Fo & \cdots & 0 \\ 0 & 0 & -Fo & (1 + 2Fo) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} c_0^{n+1} \\ c_1^{n+1} \\ c_2^{n+1} \\ c_3^{n+1} \\ \vdots \\ c_m^{n+1} \end{pmatrix} = \begin{pmatrix} c_0^n \\ c_1^n \\ c_2^n \\ c_3^n \\ \vdots \\ c_m^n \end{pmatrix}$$

$$1 \times c_0^{n+1} = c_0^n \text{ (boundary condition)}$$

$$-Foc_0^{n+1} + (1 + 2Fo)c_1^{n+1} - Foc_2^{n+1} = c_1^n$$

$$-Foc_1^{n+1} + (1 + 2Fo)c_2^{n+1} - Foc_3^{n+1} = c_2^n$$

$$-Foc_2^{n+1} + (1 + 2Fo)c_3^{n+1} - Foc_4^{n+1} = c_3^n$$

$$1 \times c_m^{n+1} = c_m^n \text{ (boundary condition)}$$

Solving the diffusion equation implicitly in Python

To solve the linear system, we need to define matrix A . It is clear that storing many zeros is not efficient in terms of memory. We use a *sparse matrix* format. Two alternative ways to set up the matrix:

Set individual elements of the matrix:

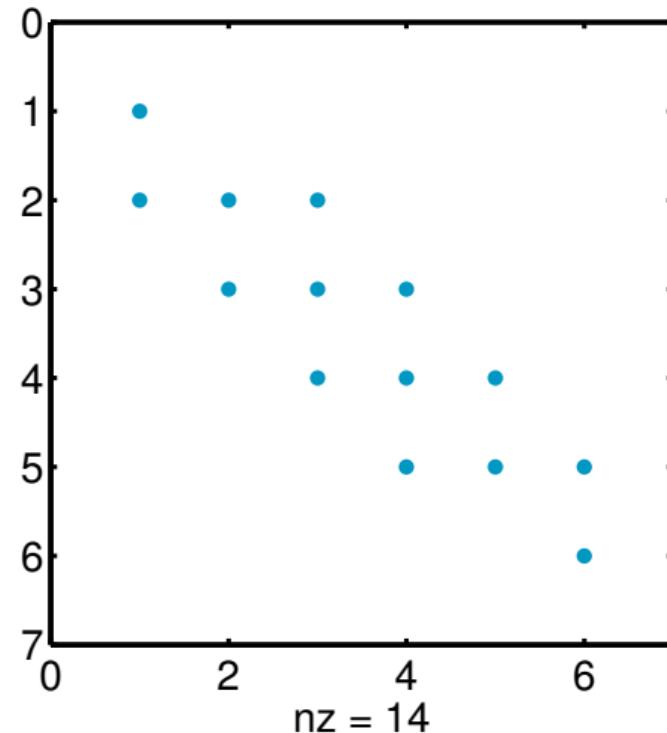
- Loop over the internal cells
- Set the coefficients in matrix A (main diagonal + elements left/right to it)
- Then set the coefficients for the boundary cells

Set matrix using bands:

- Consider the matrix structure (previous slide) and create vectors containing the values in each band
- Recall the `sp.sparse.diags` function to set entire bands to a sparse matrix

Solving the diffusion equation implicitly in Python

The command `plt.spy(A)` shows a figure with the non-zero positions.



Solving the diffusion equation implicitly in Python

The concentration matrix is initialised and the boundary conditions are set as follows:

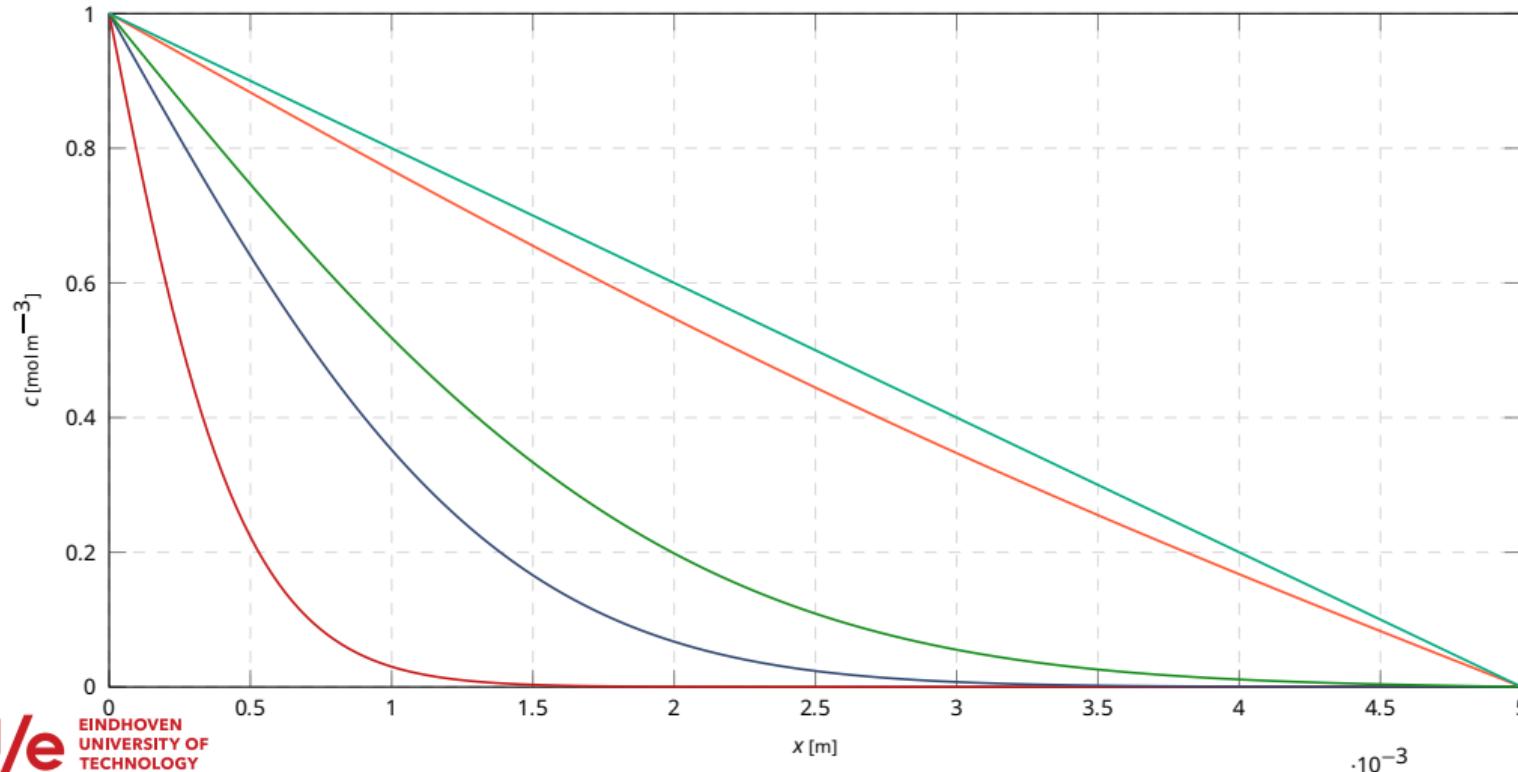
```
1 # Initial matrices for solutions (Nx times Nt)
2 c = np.zeros((Nt+1, Nx+1)) # All concentrations are zero
3 c[:, 0] = c_L # Concentration at left side
4 c[:, Nx] = c_R # Concentration at right side
```

The right hand side vector (*b*) can now be set during the time-loop:

```
1 from scipy.sparse.linalg import spsolve
2
3 for n in range(Nt-1): # time loop
4     b = c[n, :] # Set right hand side
5     solX = spsolve(A, b) # Solve linear system
6     c[n+1, :] = solX # Store solution each time step
```

Solving the diffusion equation implicitly in Matlab

Plotting the solution at $t = \{12.5, 62.5, 125, 625, 5000\}$ s.



About explicit vs. implicit solutions

- Explicit solution:
 - Easy to implement
 - Very small time steps required.
 - This problem took about 0.5 s.
- Implicit solution:
 - Harder to implement, needs sparse matrix solver
 - No stability constraint
 - This problem took about 0.05 s
- The difference will become much larger for systems with e.g. more grid nodes!

Extension with non-linear source terms

$$\frac{\partial c}{\partial t} = D \frac{\partial^2 c}{\partial x^2} + R(c) \quad \text{with} \quad \begin{aligned} t &= 0; 0 \leq x \leq \ell \Rightarrow c = c_0 \\ t &> 0; x = 0 \Rightarrow c = c_L \\ t &> 0; x = \ell \Rightarrow c = c_R \end{aligned}$$

- Forward Euler (explicit): simply add to right-hand side

$$\begin{aligned} \frac{c_i^{n+1} - c_i^n}{\Delta t} &= D \frac{c_{i-1}^n - 2c_i^n + c_{i+1}^n}{\Delta x^2} + R(c_i^n) \\ \Rightarrow c_i^{n+1} &= Foc_{i-1}^n + (1 - 2Foc_i^n)c_i^n + Foc_{i+1}^n + R_i^n \Delta t \end{aligned}$$

- Backward Euler (implicit): linearization required

$$\begin{aligned} R(c_i^{n+1}) &= R(c_i^n) + \left. \frac{dR}{dc} \right|_i^n (c_i^{n+1} - c_i^n) \\ \frac{c_i^{n+1} - c_i^n}{\Delta t} &= D \frac{c_{i-1}^{n+1} - 2c_i^{n+1} + c_{i+1}^{n+1}}{\Delta x^2} + R(c_i^{n+1}) \\ \Rightarrow -Foc_{i-1}^{n+1} + (1 + 2Foc_i^n - \left. \frac{dR}{dc} \right|_i^n \Delta t) c_i^{n+1} - Foc_{i+1}^{n+1} &= c_i^n + \left(R_i^n - \left. \frac{dR}{dc} \right|_i^n c_i^n \right) \Delta t \end{aligned}$$

Today's outline

● Introduction

● Instationary diffusion equation

- Discretization
- Solving the diffusion equation
- Non-linear source terms

● Convection

- Discretization
- Central difference scheme
- Upwind scheme

● Conclusions

- Other methods
- Summary

Extension with convection terms

$$\frac{\partial c}{\partial t} = \mathcal{D} \frac{\partial^2 c}{\partial x^2} - u \frac{\partial c}{\partial x} + R$$

Discretization of first derivative $\frac{dc}{dx}$,
looks simple but is numerical headache!

Central discretization:

$$\frac{dc}{dx} = \frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

⇒ simple and easy, too bad it doesn't work: yields unstable solutions if convection dominated.

Central difference scheme of 1st derivative

Unsteady convection:

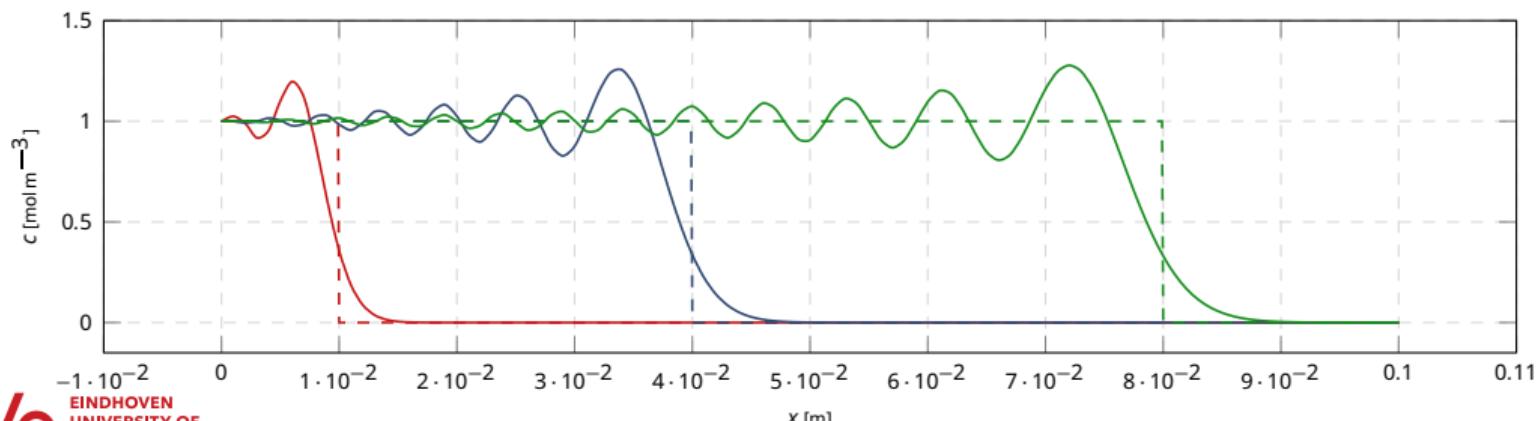
$$\frac{\partial c}{\partial t} = -u \frac{\partial c}{\partial x}$$

Central difference for first derivative:

$$\frac{dc}{dx} = \frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

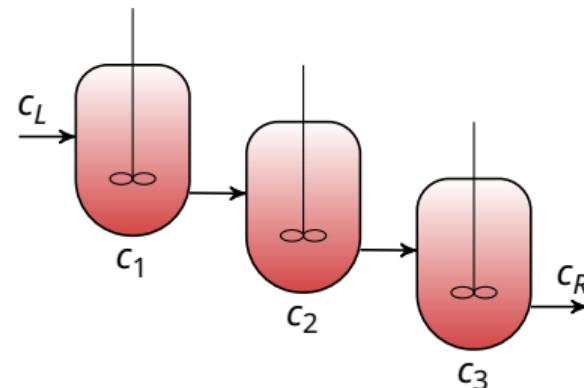
Forward Euler discretization of temporal and spatial domain:

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = -u \frac{c_{i+1} - c_{i-1}}{2\Delta x} \Rightarrow c_i^{n+1} = c_i^n - u \frac{c_{i+1}^n - c_{i-1}^n}{2\Delta x} \Delta t$$



Extension with convection terms

Solution: upwind discretization, like CSTR's in series:



First order upwind: $-u \frac{dc}{dx} \Big|_i = \begin{cases} -u \frac{c_i - c_{i-1}}{\Delta x} & \text{if } u \geq 0 \\ -u \frac{c_{i+1} - c_i}{\Delta x} & \text{if } u < 0 \end{cases}$

Stable if $\text{Co} = \frac{u\Delta t}{\Delta x} < 1$ (with Co the Courant number).

Courant number). However, only 1st order accurate (large smearing of concentration fronts). Higher order upwind requires TVD schemes (trick of the trade)...

First order upwind scheme of 1st derivative

Unsteady convection:

$$\frac{\partial c}{\partial t} = -u \frac{\partial c}{\partial x}$$

Upwind scheme for first derivative:

$$-u \frac{dc}{dx} \Big|_i = \begin{cases} -u \frac{c_i - c_{i-1}}{\Delta x} & \text{if } u \geq 0 \\ -u \frac{c_{i+1} - c_i}{\Delta x} & \text{if } u < 0 \end{cases}$$

Forward Euler discretization of temporal and spatial domain:

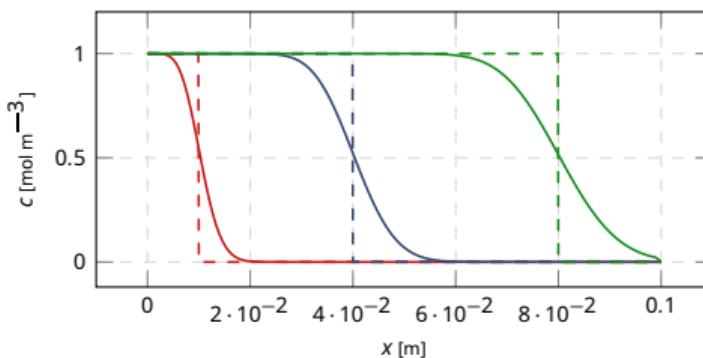
$$\begin{aligned} \frac{c_i^{n+1} - c_i^n}{\Delta t} &= -u \frac{c_{i+1} - c_{i-1}}{2\Delta x} \\ \Rightarrow c_i^{n+1} &= \begin{cases} c_i^n - u\Delta t \frac{c_i - c_{i-1}}{\Delta x} & \text{if } u \geq 0 \\ c_i^n - u\Delta t \frac{c_{i+1} - c_i}{\Delta x} & \text{if } u < 0 \end{cases} \end{aligned}$$

Upwind scheme: example

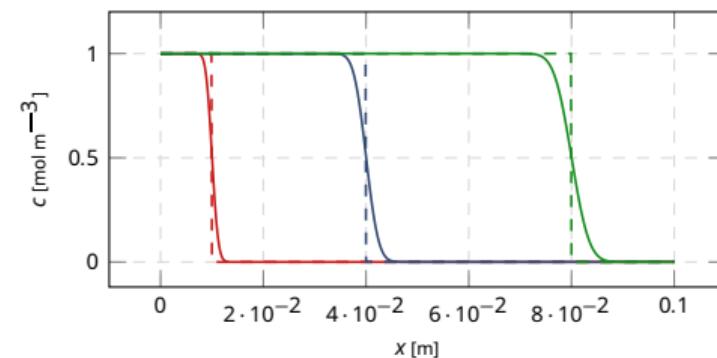
Unsteady convection through a pipe:

$$\frac{\partial c}{\partial t} = -u \frac{\partial c}{\partial x} \quad \text{with} \quad u = 0.1 \text{ m s}^{-1} \Rightarrow c_i^{n+1} = c_i^n - u \frac{c_i - c_{i-1}}{\Delta x} \Delta t$$

Using 100 grid cells



Using 1000 grid cells



Today's outline

● Introduction

● Instationary diffusion equation

- Discretization
- Solving the diffusion equation
- Non-linear source terms

● Convection

- Discretization
- Central difference scheme
- Upwind scheme

● Conclusions

- Other methods
- Summary

Extension to systems of PDE's

- Explicit methods: straightforward extension
- Implicit methods: yields block-tridiagonal matrix (note ordering of equations: all variables per grid cell)

Extension to 2D or 3D systems

Spatial discretization in 2 directions — different methods available:

- Explicit
- Fully implicit
 - 1D gives tri-diagonal matrix
 - 2D gives penta-diagonal matrix
 - 3D gives hepta-diagonal matrix

Use of dedicated matrix solvers (e.g. ICCG, multigrid, ...)

- Alternating direction implicit (ADI)
 - Per direction implicit, but still overall unconditionally stable

Further extensions for parabolic PDEs

- Higher order temporal discretization (multi-step) with time step adaptation
- Non-uniform grids with automatic grid adaptation
- Higher-order discretization methods, especially higher order TVD (flux delimited) schemes for convective fluxes (e.g. WENO schemes)
- Higher-order finite volume schemes (Riemann solvers)

Summary

- Several classes of PDEs were introduced
 - Elliptic, Parabolic, Hyperbolic PDEs
- Diffusion equation: discretization of temporal and spatial domain was discussed
 - Solutions of the diffusion equation using explicit and implicit methods
 - How to add non-linear source terms
- Convection: upwind vs. central difference schemes

Curve fitting, regression and optimization

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

- Introduction
 - Curve fitting
 - Regression
 - Fitting numerical models
 - Optimization
 - Linear programming
 - Summary

Overview

- We are going to fit measurements to models today
 - You will also learn what R^2 actually means
 - We get introduced to constrained and unconstrained optimization.
 - We will use the simplex method to solve linear programming problems (LP)

Today's outline

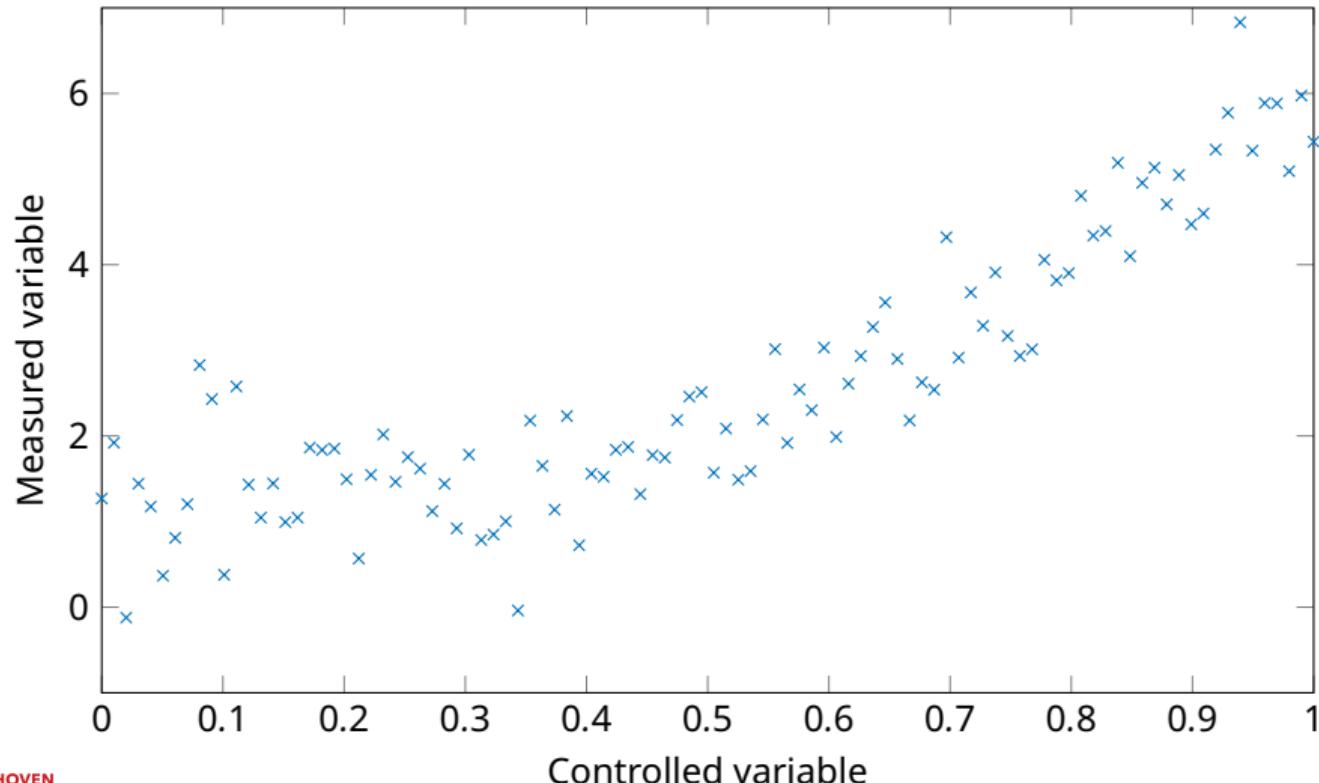
- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

Let's do an 'experiment' to gather data

```
1 def generate_random_data(N=101,p=[1,1/3,1.5,3.5],draw=False):
2     # Generate linear space of control points
3     # N - Number of data points
4     # p - Coefficients of polynomial
5     x = np.linspace(0, 1, N) # Points (independent variable)
6
7     # Generate 'measurement values' with errors following a normal distribution
8     # Initialize randomizer
9     pd = norm(loc=0, scale=0.1)
10    # Add scatter data to the polynomial
11    y = np.polyval(p,x) + pd.rvs(size=N)
12
13    # Plot the generated data
14    if draw:
15        plt.plot(x, y, 'x')
16        plt.show()
17    return x,y
```

Gather some data by calling the function and storing x and y

Fitting models to data



How to fit a model to the data?

We would like to fit the following model to the data:

$$\hat{y} = a_0x^3 + a_1x^2 + a_2x + a_3$$

First attempt - using the `polyfit` function we have seen with the interpolation lecture:

```
1 def fit_using_polyfit(x,y,n=2,draw=False):
2     p = np.polyfit(x,y,n)
3     if draw:
4         plt.plot(x, y, 'x')
5         plt.plot(x, np.polyval(p,x), '-')
6         plt.show()
7     return p
```

If we print `p`, we get the coefficients. But this is a black box, what does it do?

How to fit a model to the data?

We would like to fit the following model to the data:

$$\hat{y} = a_0x^3 + a_1x^2 + a_2x + a_3$$

Attempt to solve a linear system: If we have N data points, we could write the model as the product of a matrix and a vector:

$$\begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_{N-1} \end{bmatrix} = \begin{bmatrix} x_0^3 & x_0^2 & x_0 & 1 \\ x_1^3 & x_1^2 & x_1 & 1 \\ x_2^3 & x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{N-1}^3 & x_{N-1}^2 & x_{N-1} & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$$\hat{y} = Xa$$

X is called the design matrix
and a are the fit parameters.

Residuals

Second step: work out the residuals for each data point:

$$d_i = (y_i - \hat{y}_i)$$

Third step: work out the sum of squares of the residuals:

$$SSE = \sum_i d_i^2 = \sum_i (y_i - \hat{y}_i)^2$$

This can be written using the dot-product operation:

$$SSE = \sum_i d_i^2 = d \cdot d = d^T \cdot d = (y_i - \hat{y}_i)^T \cdot (y_i - \hat{y}_i)$$

Minimizing the sum of squares

Choose the parameter vector such that the sum of squares of the residuals is minimized; the partial derivative with respect to each parameter should be set to zero:

$$\frac{\partial}{\partial a_i} \left[(y - (Xa)^T)(y - Xa) \right]$$

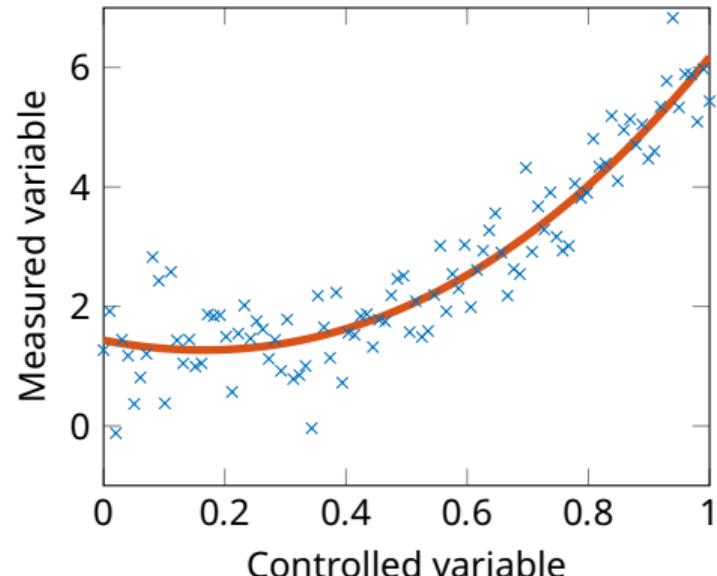
In Python, we can solve our linear system $\hat{Y} = Xa$ simply by running `a = np.linalg.solve(X,y)`.

- If there are more data points ($N > 4$), we can write an analogue, but maybe a consistent solution does not exist (the system is over specified).
- However, Python will find values for the vector a which minimize $\|y - aX\|^2$, so i.e. a least squares fit.

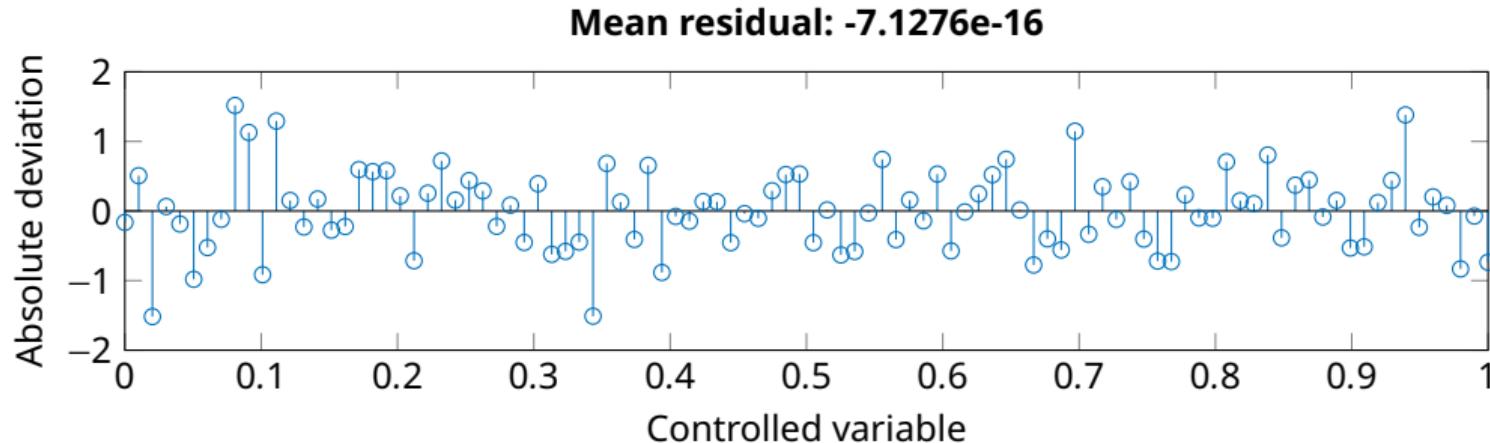
Fitting our problem: Least squares solver

As a follow-up of the script provided in slide 489

```
1 def fit_using_lstsq(x,y,n=2,draw=False):
2     xmat = np.vander(x,n+1)
3     sol = np.linalg.lstsq(xmat,y,rcond=None)
4     A = sol[0]
5     yhat = xmat@A
6     if draw:
7         plt.plot(x,y,'x')
8         plt.plot(x,yhat,'-')
9         plt.title('Fit using lstsq')
10        plt.show()
11    return A,yhat
```



How good is the model?



- For a model to make sense the data points should be scattered randomly around the model predictions, the mean of the residuals d should be zero: $d_i = (y_i - \hat{y}_i)$
- It's always good to check if the residuals are not correlated with the measured values, if that is the case, it can indicate that your model is wrong.

Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

Regression coefficients

- Variance measured in the data (y) is:

$$\sigma_y^2 = \frac{1}{N} \sum_i (y_i - \bar{y})^2$$

- Variance of the residuals is:

$$\sigma_{\text{error}}^2 = \frac{1}{N} \sum_i (d_i)^2$$

- Variance in the model is:

$$\sigma_{\text{model}}^2 = \frac{1}{N} \sum_i (\hat{y}_i - \bar{\hat{y}})^2$$

Regression coefficients

Given that the error is uncorrelated we can state that:

$$\sigma_y^2 = \sigma_{\text{error}}^2 + \sigma_{\text{model}}^2$$

$$R^2 = \frac{\sigma_{\text{model}}^2}{\sigma_y^2} = 1 - \frac{\sigma_{\text{error}}^2}{\sigma_y^2}$$

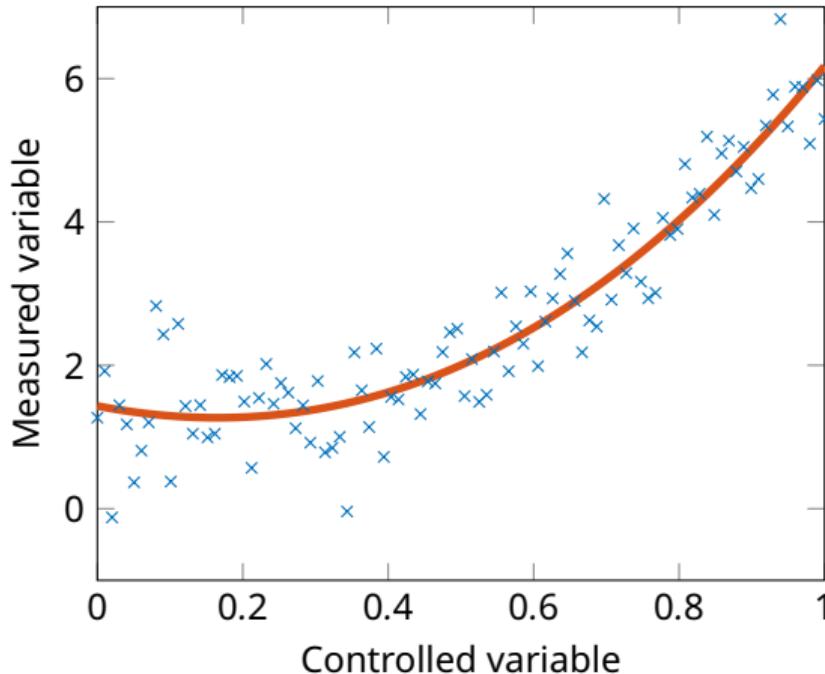
$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}}$$

- SSE: Sum of errors (residuals) squared (difference between data and model)
- SST: Total sum of squares (variance of the data)
- SSR: Sum of squares (model)

Back to the example

The statistics:

	Value
N	100
SSE	32.042
SST	896.907
SSR	928.950
R^2	0.964



Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

Curve fitting from command line: `scipy.optimize.curve_fit`

Python offers various non-linear parameter and curve fitting tools that can be run from the command line. The function `curve_fit` from the `scipy.optimize` module allows to fit a model to a given dataset. Again, based on the data generated in slide 489:

```
1 from poly_regression import generate_random_data
2 from scipy.optimize import curve_fit
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 # Define the model function
7 def curve_fit_model_1(xdata, a0, a1, a2, a3):
8     return a0*xdata**3 + a1*xdata**2 + a2*xdata + a3
```

```
1 if __name__ == '__main__':
2     x,y = generate_random_data()
3     a0 = [1, 2, 1, 3] # Initial guess of coefficients
4
5     # Perform fitting, store resulting coeffs in a_fit
6     a_fit, _ = curve_fit(curve_fit_model_1, x, y, p0=a0)
7
8     # Run the model once more, with fitted coefficients
9     y_model = curve_fit_model_1(x, *a_fit) # unpack coefficients using *
```

Curve fitting from command line: `scipy.optimize.curve_fit`

For fitting a polynomial model, this function works as well as `polyfit`. But it also allows other (much more complex) types of model to be defined:

```
1 def curve_fit_model_2(xdata, a0, a1, a2):
2     return a0*np.exp(a1*xdata) + a2
3
4 # Initial guess of coefficients
5 a0 = [1,1,1]
6
7 # Perform fitting of the data to another model
8 a_fit, _ = curve_fit(curve_fit_model_2, x, y, p0=a0)
9
10 # Run the model once more, with fitted coefficients
11 y_model = curve_fit_model_2(x, *a_fit)
```

The model functions take individual parameters separately, we use the unpacking syntax (*) to pass the fitted parameters when generating the `y_model` data.

Dynamic fitting of non-linear equations

You may encounter situations where the model data is slightly more complicated to obtain (e.g., a numerical model based on ODEs where coefficients are unknown), or you want to perform fitting of multiple functions/coefficients, or just want to automate things via scripts. Python's Scipy library gives access to powerful functions such as `least_squares` and `curve_fit`.

General use of `scipy.optimize.least_squares`

```
1 from scipy.optimize import least_squares  
2  
3 result = least_squares(fun, k0, bounds=(lb, ub), xtol=1.0E-6, max_nfev=1000)
```

- `fun` is a function handle to the fit criterion (e.g., `myFitCrit`). The fit criterion function `myFitCrit` should return the residuals vector, e.g., $d_i = (y_i - \hat{y}_i)$. Here, y_i would again be the measurement data and \hat{y} the solution computed by a model.
- `k0` is the initial guess for the fitting coefficient (or: array of initial guesses when fitting multiple coefficients).
- `lb` and `ub` are the lower and upper boundaries for `k0`. These should both be the size of the `k0`-array.
- Use arguments such as `xtol` and `max_nfev` for more fine-grained control on the fit procedure.

General use of `scipy.optimize.curve_fit`

```
1 from scipy.optimize import curve_fit
2
3 popt, pcov = curve_fit(fun, xdata, ydata, p0=k0, bounds=(lb, ub))
```

- `fun` is the model function that you want to fit to your data. It takes the independent variable as the first argument and the parameters to fit as separate remaining arguments.
- `xdata` and `ydata` are the data points that you are fitting the model function to.
- `k0` is the initial guess for the parameters to be fitted.
- `lb` and `ub` are the lower and upper bounds for the parameters, respectively.
- `popt` will contain the optimized parameters, and `pcov` will contain the covariance matrix, which can give you an idea of the uncertainties of the estimates.

Example use of `scipy.optimize.curve_fit`

We have experimental data stored in a file, possibly in a .csv or .txt format, containing T and U data. We want to fit a model with coefficients k_1 and k_2 with the following structure:

$$\frac{du}{dt} = -k_1 u + k_2$$

- First, we define a function that describes our model:

```
1 import scipy as sp
2 import numpy as np
3
4 def simpleode(t, u, k1, k2):
5     dudt = -k1*u + k2
6     return dudt
```

Note that we supply the coefficients k_1 and k_2 as arguments to the function.

- We create a fit criterion function:

```
1 def fitcrit(t,k1, k2):
2     u0 = [1.0]
3     tspan = [0, max(t)]
4     sol = sp.integrate.solve_ivp(simpleode, tspan, u0, args=(k1, k2), t_eval=t)
5     return sol.y[0]
```

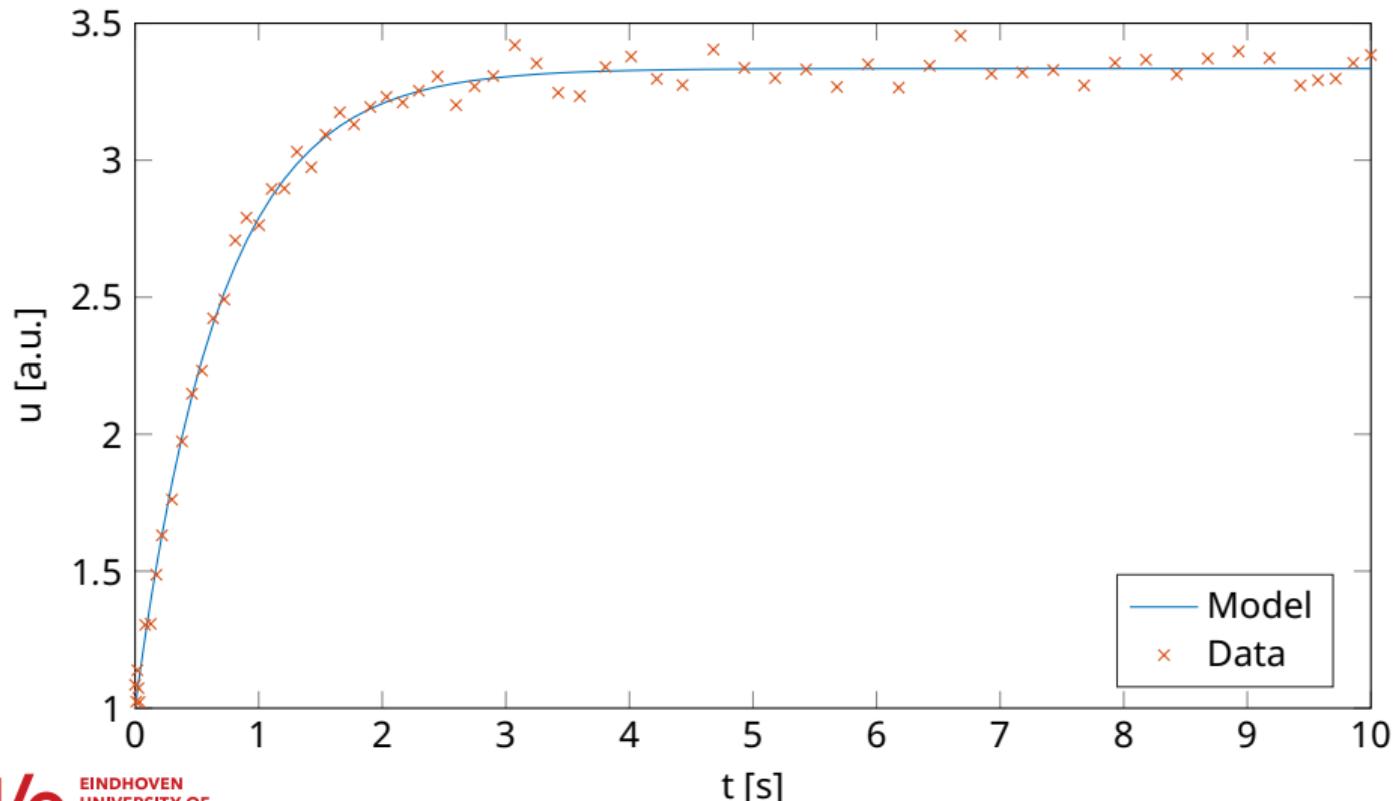
Example use of `scipy.optimize.curve_fit`

Now let's make a script that uses `curve_fit` to yield k-values fitted to our dataset:

```
1 # Load your data here (adjust as necessary)
2 T, U = np.loadtxt('./scripts/optimization/tudataset1.txt', unpack=True, skiprows=1)
3
4 # Initial guesses for model parameters
5 k0 = [1.0, 1.0]
6
7 # Perform the curve fitting
8 params, params_covariance = sp.optimize.curve_fit(fitcrit, T, U, p0=k0)
9 print('Fitted coefficients:', params)
```

Our fitted coefficients are stored in `params`. The `params_covariance` gives an estimate of the covariance of the estimated parameters, offering an insight into the uncertainty of the fit.

Example use of lsqnonlin



Postprocessing of results

The data returned by `curve_fit` can be used to obtain the 95% confidence intervals for the fitted parameters. Recall the command:

```
1 params, params_covariance = curve_fit(fitcrit, T, U, p0=k0)
```

We can use the square root of the diagonal of the covariance matrix, multiplied by a factor from the t-distribution to get the confidence bounds:

```
1 from scipy import stats
2 import numpy as np
3
4 alpha = 0.05 # 95% confidence interval = 100*(1-alpha)
5 n = len(U) # number of data points
6 p = len(params) # number of parameters
7
8 dof = max(0, n - p) # number of degrees of freedom
9 # t value for the dof and confidence level
10 tval = stats.t.ppf(1.0-alpha/2., dof)
11 sigma = np.sqrt(np.diag(params_covariance))
12 ci = sigma * tval
13
14 print('Confidence intervals:')
15 print('k1:', params[0] - ci[0], params[0] + ci[0])
16 print('k2:', params[1] - ci[1], params[1] + ci[1])
```

Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

What is optimization?

Optimization is minimization or maximization of an objective function (also called a performance index or goal function) that may be subject to certain constraints.

- $\min f(x)$: Goal function
- $g(x) = 0$: Equality constraints
- $h(x) \geq 0$: Inequality constraints

Optimization Spectrum

Problem	Method	Solvers
LP	Simplex method	Linprog
	Barrier methods	CPLEX (GAMS, AIMMS, AMPL, OPB)
NLP	Lagrange multiplier method	Fminsearch/fmincon (Matlab)
	Successive linear programming	MINOS (GAMS, AMPL)
	Quadratic programming	CONOPT (GAMS)
MIP	Branch and bound	
MILP	Dynamic programming	Bintprog (Matlab)
MINLP	Generalized Benders decomposition	DICOPT (GAMS)
	Outer approximation method	BARON (GAMS)
MIQP	Disjunctive programming	

Factors of concern

- Continuity of the functions
- Convexity of the functions
- Global versus local optima
- Constrained versus unconstrained optima

Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

Linear programming

In linear programming the objective function and the constraints are linear functions.

For example:

$$\max z = f(x_1, x_2) = 40x_1 + 88x_2$$

s.t. (subject to)

$$2x_1 + 8x_2 \leq 60$$

$$5x_1 + 2x_2 \leq 60$$

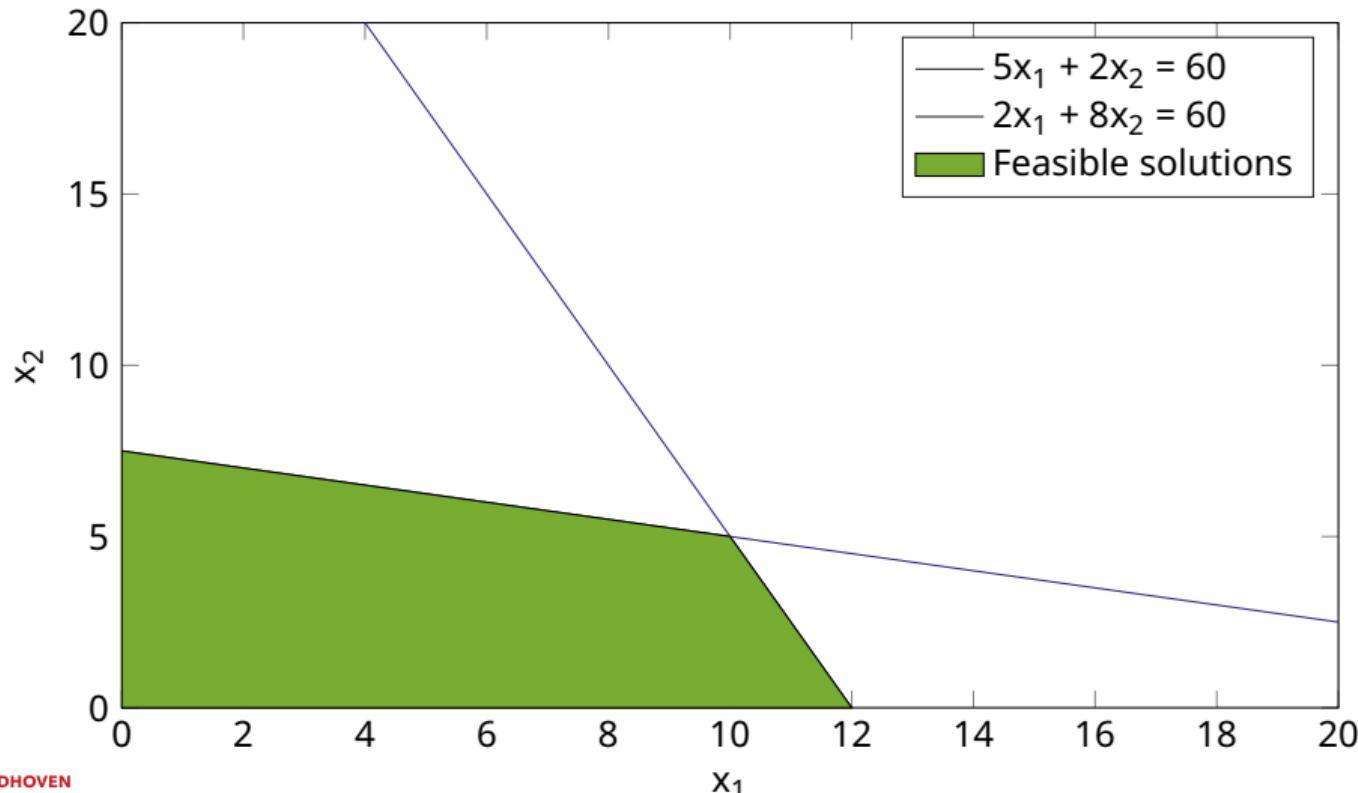
$$x_1 \geq 0$$

$$x_2 \geq 0$$

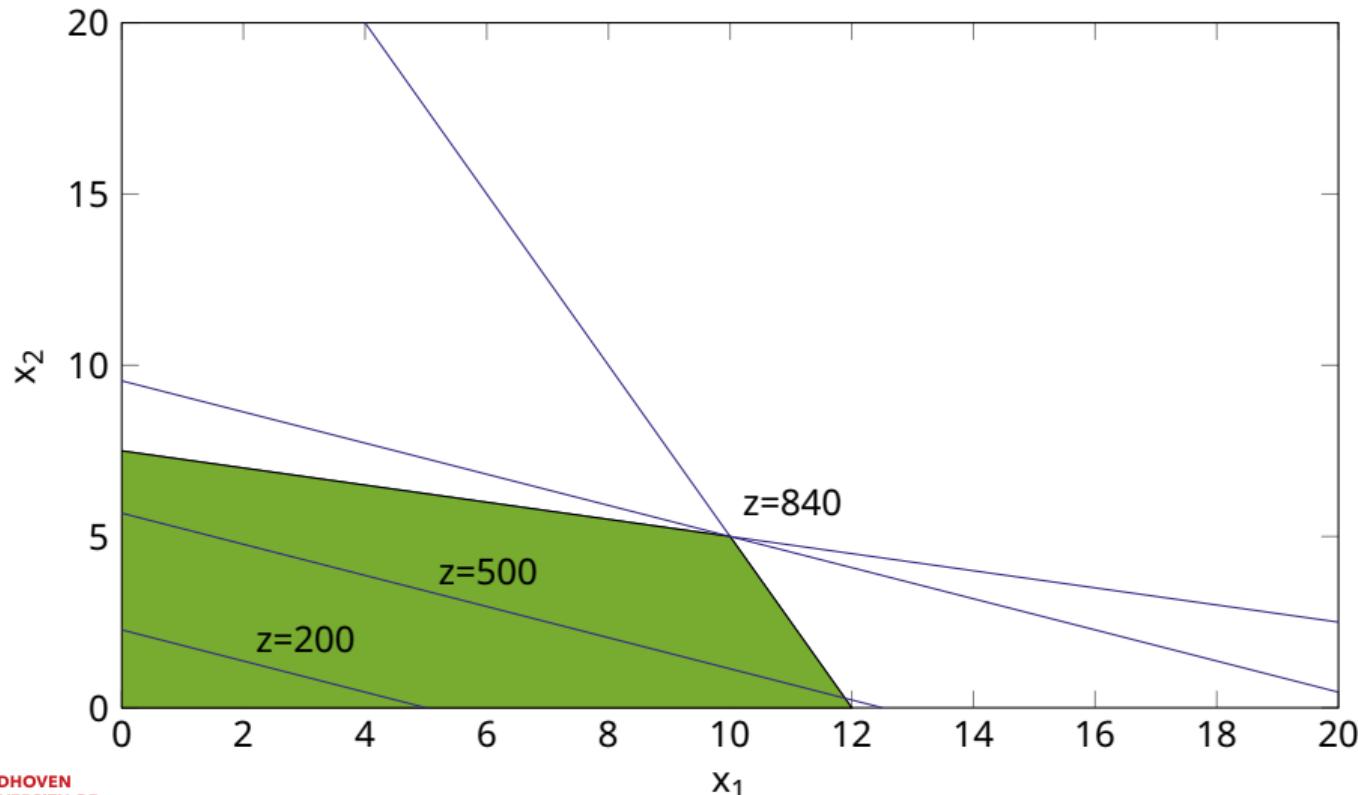
If the constraints are satisfied, but the objective function is not maximized/minimized we speak of a feasible solution.

If also the objective function is maximized/minimized, we speak of an optimal solution!

Plotting the constraints



Plotting the constraints



Normal form of an LP problem

$$\max z = f(x_1, x_2) = 40x_1 + 88x_2$$

s.t.

$$2x_1 + 8x_2 \leq 60$$

$$5x_1 + 2x_2 \leq 60$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

$$\max z = f(x) = 40x_1 + 88x_2$$

s.t.

$$2x_1 + 8x_2 + x_3 = 60$$

$$5x_1 + 2x_2 + x_4 = 60$$

$$x_i \geq 0 \quad i \in \{1, 2, 3, 4\}$$

x_3 and x_4 are called slack variables, they are non auxiliary variables introduced for the purpose of converting inequalities into equalities

The simplex method

We can formulate our earlier example to the normal form and consider it as the following augmented matrix with $T_0 = [z \ x_1 \ x_2 \ x_3 \ x_4 \ b]$:

$$T_0 = \begin{bmatrix} 1 & -40 & -88 & 0 & 0 & 0 \\ 0 & 2 & 8 & 1 & 0 & 60 \\ 0 & 5 & 2 & 0 & 1 & 60 \end{bmatrix}$$

This matrix is called the (initial) simplex table. Each simplex table has two kinds of variables, the basic variables (columns having only one nonzero entry) and the nonbasic variables

The simplex method

$$T_0 = \begin{bmatrix} 1 & -40 & -88 & 0 & 0 & 0 \\ 0 & 2 & 8 & 1 & 0 & 60 \\ 0 & 5 & 2 & 0 & 1 & 60 \end{bmatrix}$$

Every simplex table has a feasible solution. It can be obtained by setting the nonbasic variables to zero: $x_1 = 0, x_2 = 0, x_3 = 60/1, x_4 = 60/1, z = 0$.

The optimal solution?

- The optimal solution is now obtained stepwise by pivoting in such way that z reaches a maximum.
- The big question is, how to choose your pivot equation ...

Step 1: Selection of the pivot column

Select as the column of the pivot, the first column with a negative entry in Row 1. In our example, that's column 2 (-40)

$$T_0 = \begin{bmatrix} 1 & -40 & -88 & 0 & 0 & 0 \\ 0 & 2 & 8 & 1 & 0 & 60 \\ 0 & 5 & 2 & 0 & 1 & 60 \end{bmatrix}$$

Step 2: Selection of the pivot row

Divide the right sides by the corresponding column entries of the selected pivot column. In our example that is $60/2 = 30$ and $60/5 = 12$.

$$T_0 = \begin{bmatrix} 1 & -40 & -88 & 0 & 0 & 0 \\ 0 & 2 & 8 & 1 & 0 & 60 \\ 0 & 5 & 2 & 0 & 1 & 60 \end{bmatrix}$$

Take as the pivot equation the equation that gives the smallest quotient, so $60/5$.

Step 3: Elimination by row operations

- Row 1 = Row 1 + 8 * Row 3
- Row 2 = Row 2 - 0.4 * Row 3

$$T_1 = \begin{bmatrix} 1 & 0 & -72 & 0 & 8 & 480 \\ 0 & 0 & 7.2 & 1 & -0.4 & 36 \\ 0 & 5 & 2 & 0 & 1 & 60 \end{bmatrix}$$

The basic variables are now x_1, x_3 and the nonbasic variables are x_2, x_4 . Setting the nonbasic variables to zero will give a new feasible solution: $x_1 = 60/5, x_2 = 0, x_3 = 36/1, x_4 = 0, z = 480$.

The simplex method

- We moved from $z = 0$ to $z = 480$. The reason for the increase is because we eliminated a negative term from the equation, so: elimination should only be applied to negative entries in Row 1, but no others.
- Although we found a feasible solution, we did not find the optimal solution yet (the entry of -72 in our simplex table) —> repeat step 1 to 3.

The simplex method

Another iteration is required:

- Step 1: Select column 3
- Step 2: $36/7.2 = 5$ and $60/2 = 30 \rightarrow$ select 7.2 as the pivot
- Elimination by row operations:
 - Row 1 = Row 1 + 10*Row 2
 - Row 3 = Row 3 - (2/7.2)*Row 2

$$T_2 = \begin{bmatrix} 1 & 0 & 0 & 10 & 4 & 840 \\ 0 & 0 & 7.2 & 1 & -0.4 & 36 \\ 0 & 5 & 0 & -1/36 & 1/0.9 & 50 \end{bmatrix}$$

- The basic feasible solution: $x_1 = 50/5, x_2 = 36/7.2, x_3 = 0, x_4 = 0, z = 840$ (no more negative entries: so this solution is also the optimal solution)

Using Python for LP problems

We are going to solve the following LP problem:

$$\text{min } f(x) = -5x_1 - 4x_2 - 6x_3$$

s.t.

$$x_1 - x_2 + x_3 \leq 20$$

$$3x_1 + 2x_2 + 4x_3 \leq 42$$

$$3x_1 + 2x_2 \leq 30$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

$$x_3 \geq 0$$

Using the function `linprog` from `scipy.optimize`:

```
1 from scipy.optimize import linprog
2
3 c = [-5, -4, -6]
4 A = [[1, -1, 1], [3, 2, 4], [3, 2, 0]]
5 b = [20, 42, 30]
6 bounds = [(0, None), (0, None), (0, None)]
7
8 res = linprog(c, A_ub=A, b_ub=b, bounds=bounds)
```

Gives (after accessing appropriate attributes of the result object):

```
1 x = res.x
2 fun = res.fun
3 slack = res.slack
4 success = res.success
```

Summary

- Curve fitting: Manual procedures for polynomial fitting in Python
- Curve fitting: Python's SciPy library for curve fitting
- Curve fitting: Python's non-linear least-squares solver `least_squares`
- Curve fitting: Python's non-linear least-squares solver `curve_fit`
- Optimization: An introduction to the Simplex method in Python
- Optimization: Use of the `linprog` function in the SciPy library

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

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

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...

Tips:

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

Hamming's motto

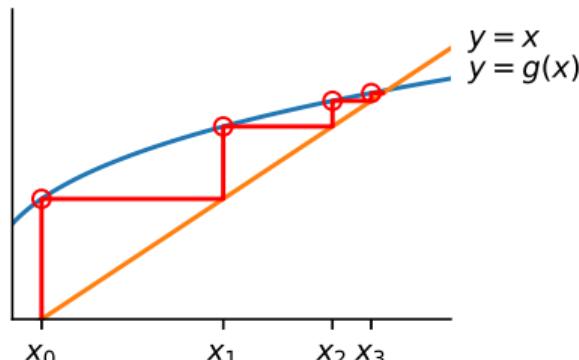
The purpose of computing is insight,
not numbers!!

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

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

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:

```
1 def demo_f1(x):  
2     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:

```
1 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 $f(x)$:

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

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

```
1 f = MyFunc
2 a = 4
3 b = f(a)
```

2

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

```
1 from scipy.optimize import fsolve
2 ans = fsolve(MyFunc, 5)
3 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:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 def draw_my_function(func):
5     # Draws a function in the range [0, 10] using 20 data points.
6     # 'func' is a function that can be any actual function.
7     x = np.linspace(0, 10, 20)
8     y = func(x)
9     plt.plot(x, y, "-o")
10    plt.show()
```

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

```
1 f = lambda x: x**2 - 4*x + 2
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
:		:
10		3.990

Converges!

Second attempt:

Iteration	Formula	Result
1	$x = (x^3 - 3x^2 - 4)/3$	-1
2		-2.375
3		-11.439
:		:
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:

```
1 x = 2.5
2 print(f"i: {0}, x: {x:.6e}")
3 for i in range(1, 21):
4     x = (3*x**2 + 3*x + 4)**(1/3)
5     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.
- Then, create a function to carry out the Direct Iteration algorithm.

```
1 def MyFnc1(x):
2     return (3*x**2 + 3*x + 4)**(1/3)
3
4 def MyFnc2(x):
5     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.

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

```
1 DirectIterationMethod(MyFnc1, 2.5, 1e-3)
2 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

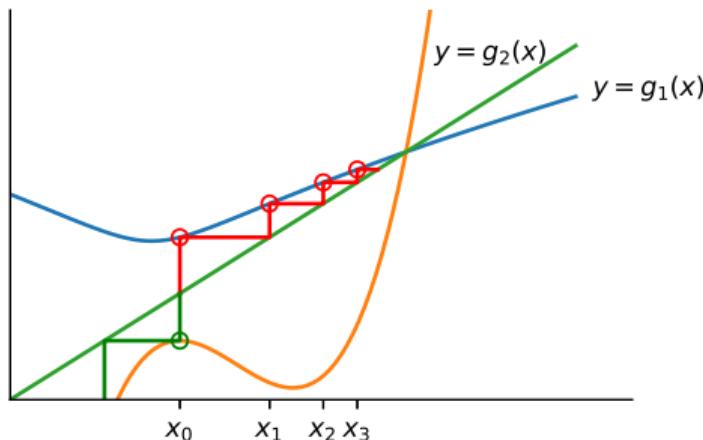
Direct Iteration Method

- 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

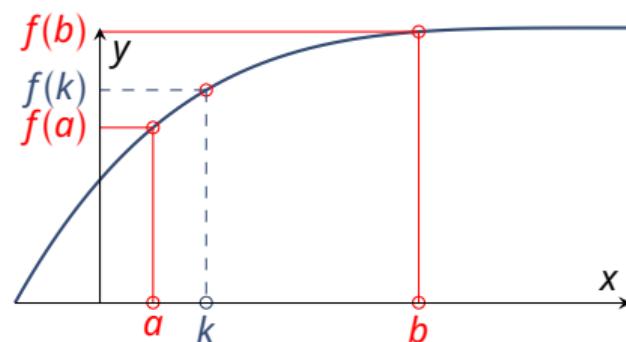
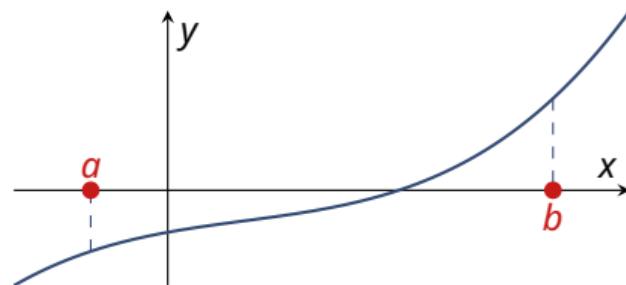
● Bisection method

● 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.

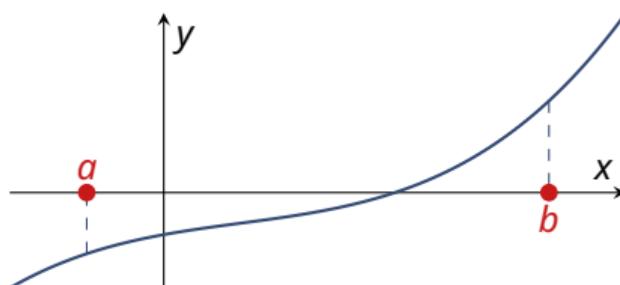
Intermediate value theorem

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.

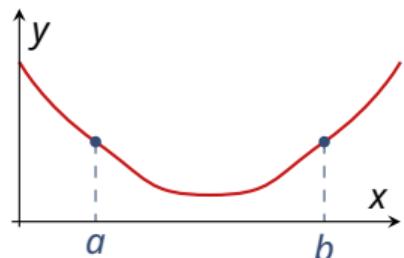


General best advice:

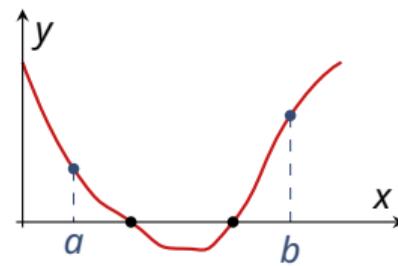
- Always bracket a root before attempting to converge on a solution.
- Never allow your iteration method to get outside the best bracketing bounds...

General Idea

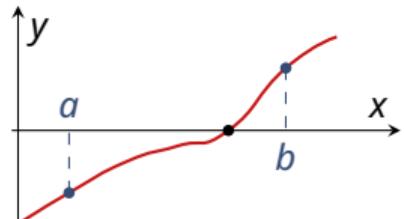
Potential issues to be cautious of while bracketing:



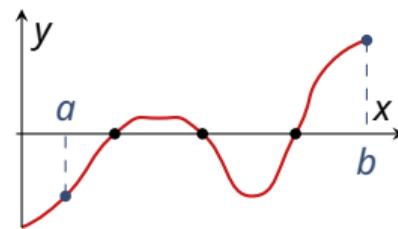
No answer (no root found)



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:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 5, 50)
5 y = x**2 - 4*x + 2
6 plt.figure()
7 plt.plot(x, y, x, np.zeros(len(x)))
8 plt.axis('tight')
9 plt.grid(True)
10 plt.show()
```

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

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

Bracketing - exercise 2

```
1 def find_root_by_bracketing(func, x1, x2, tol=1e-6, max_iter=1000):
2     # Ensure the bracket is valid
3     if func(x1) * func(x2) > 0:
4         print('The bracket is invalid. The function must have opposite signs at
5             the two endpoints.')
6         return False
7
8     # Loop until we find the root or exceed the maximum number of iterations
9     for i in range(max_iter):
10         # Find the midpoint
11         x_mid = (x1 + x2) / 2
12
13         # Check if we found the root
14         if abs(func(x_mid)) < tol:
15             print(f'Root found: {x_mid}')
16             return True
17
18         # Narrow down the bracket
19         if func(x_mid) * func(x1) < 0:
20             x2 = x_mid
21         else:
22             x1 = x_mid
23
24     # If we reach here, we did not find the root within the maximum number of
25     # iterations
26     print('Failed to find the root within the maximum number of iterations.')
27     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

```
1 def brak(func, x1, x2, n):
2     nroot = 0
3     dx = (x2 - x1) / n
4     xb1 = []
5     xb2 = []
6
7     x = x1
8     for i in range(n):
9         x += dx
10        if func(x) * func(x - dx) <= 0:
11            nroot += 1
12            xb1.append(x - dx)
13            xb2.append(x)
14
15    for i in range(nroot):
16        print(f'Root {i+1} in bracketing interval
17              [{xb1[i]}, {xb2[i]}]')
18    else:
19        if nroot == 0:
20            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

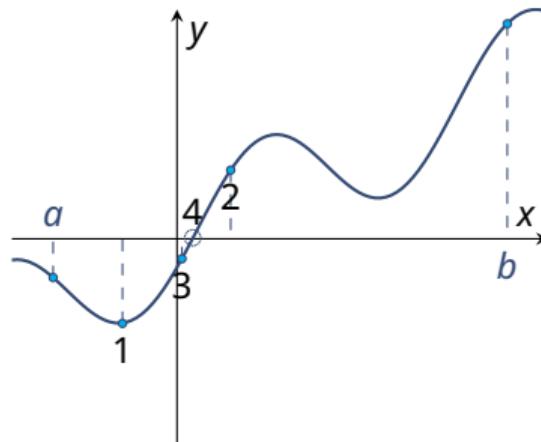
● Secant/False Position

● 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
:	:	:	:	:	:	:	:
25	0.585786	0.585786	1×10^{-7}	-6.8×10^{-8}	0.585786	1.58×10^{-8}	5.96×10^{-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

```
1 def bisection(func, a, b, tol, maxIter):
2     if func(a) * func(b) > 0:
3         print('Error: f(a) and f(b) must have different signs.')
3         )
4     return None
5
6     iter = 0
7     while (b - a) / 2 > tol:
8         iter += 1
9         if iter >= maxIter:
10            print('Maximum iterations reached')
11            return None
12
13         c = (a + b) / 2
14         print(f'Iteration {iter}: Current estimate: {c}')
15
16         if func(c) == 0:
17             return c
18
19         if np.sign(func(c)) != np.sign(func(a)):
20             b = c
21         else:
22             a = c
23
24     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.

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

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:

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

- $m = 1$: 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

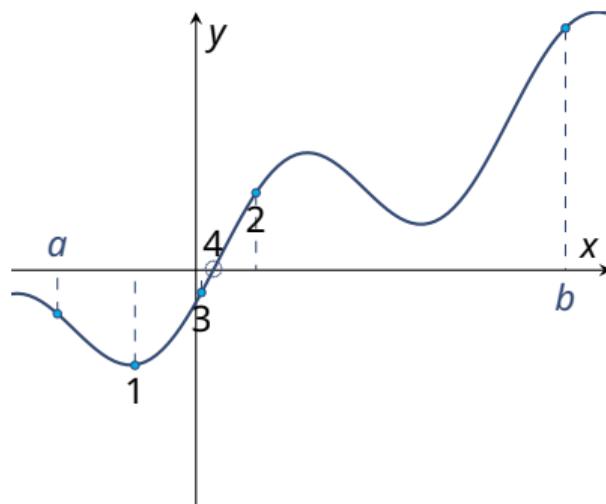
Secant and False Position Method

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

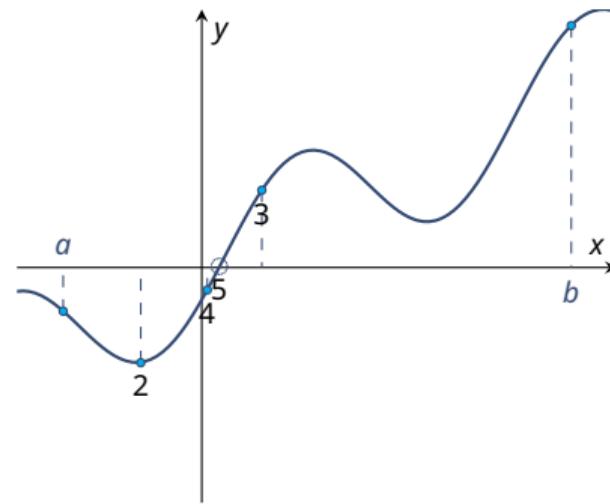
Secant Method



- Slightly faster convergence:

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

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

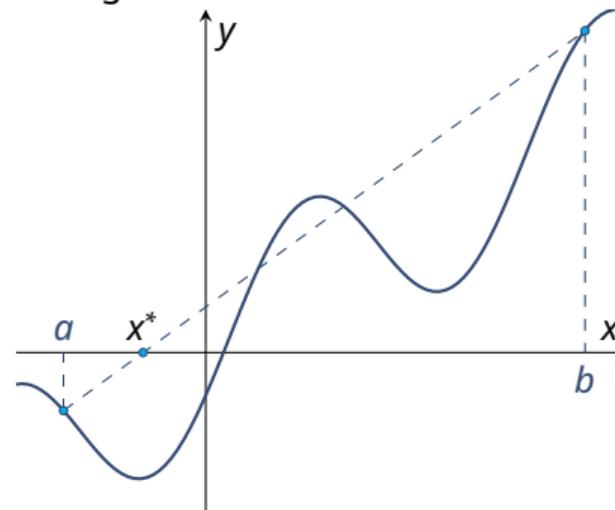
Exercise 4:

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

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



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	fabs	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:

- $a=IF((a*fa)<0, a, xb)$
- $b=IF((b*fb)<0, b, xb)$

$$xb = a - \frac{fa * (xb - xa)}{(fb - fa)}$$

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

```
1 def false_position(f, x0, x1, tol, max_iter):
2     if f(x0) * f(x1) > 0:
3         raise ValueError('f(x0) and f(x1) must have different signs.')
4
5     history = []
6
7     for i in range(max_iter):
8         x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
9         history.append(x2)
10
11        if abs(f(x2)) < tol:
12            break
13
14        if f(x2) * f(x0) < 0:
15            x1 = x2
16        else:
17            x0 = x2
18
19    root = x2
20    return root, history
```

Calling the function:

```
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

```
1 def secant_method(f, x0, x1, tol, max_iter):
2     history = [x0, x1]
3
4     for i in range(1, max_iter):
5         x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
6         history.append(x2)
7
8         if abs(x2 - x1) < tol:
9             break
10
11         x0 = x1
12         x1 = x2
13
14     root = x1
15     return root, history
```

Calling the function:

```
1 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

```
1 from scipy.optimize import root_scalar
2
3 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:

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

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

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

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

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

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

- 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

```

1 def brent_method(f, a, b, tol=1e-6, max_iter=100):
2     if f(a) * f(b) >= 0:
3         raise ValueError("f(a) and f(b) must have different signs.")
4     # Initialize variables
5     c = a
6     fa = f(a)
7     fb = f(b)
8     fc = fa
9     history = [a, b]
10    d = e = b - a
11    for _ in range(max_iter):
12        if fa * fc > 0:
13            c = a
14            fc = fa
15            d = e = b - a
16        if abs(fc) < abs(fb):
17            a, b, c = b, c, a
18            fa, fb, fc = fb, fc, fa
19            tol1 = 2 * 1.0e-16 * abs(b) + 0.5 * tol
20            xm = 0.5 * (c - b)
21            if abs(xm) <= tol1 or fb == 0:
22                return b, history
23            if abs(e) >= tol1 and abs(fa) > abs(fb):
24                s = fb / fa
25                if a == c:
26                    # Linear interpolation (Secant method)
27                    p = 2 * xm * s
28                    q = 1 - s

```

```

28                q = 1 - s
29            else:
30                # Inverse quadratic interpolation
31                q = fa / fc
32                r = fb / fc
33                p = s * (2 * xm * q * (q - r) - (b - a) * (r - 1))
34                q = (q - 1) * (r - 1) * (s - 1)
35            if p > 0:
36                q = -q
37            p = abs(p)
38
39            if 2 * p < min(3 * xm * q - abs(tol1 * q), abs(e * q)):
40                e = d
41                d = p / q
42            else:
43                d = xm
44                e = d
45            else:
46                d = xm
47                e = d
48            a = b
49            fa = fb
50            if abs(d) > tol1:
51                b += d
52            else:
53                b += tol1 if xm > 0 else -tol1
54
55            fb = f(b)
56            history.append(b)
57            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:

- 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		

- Navigate to Data → What-if Analysis → Goal Seek and input:
 - Set cell: B2
 - To value: 0
 - By changing cell: B1
- 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	x_1	3	$=2*B2*B3-B3+2$
3	x_2	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

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`.

```
1 from scipy.optimize import root_scalar  
2  
3 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 (≥ 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.

```
1 from scipy.optimize import fsolve
2
3 def equations(x):
4     return [2*x[0]*x[1] - x[1] + 2, 2*x[1] - 4*x[0] - 4]
5
6 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 δx and well-behaved functions, non-linear terms in the Taylor series become unimportant.

$$f(x) \approx f(x_i) + f'(x_i)\delta x + \mathcal{O}(\delta x^2) + \dots$$

$$0 \approx f(x_i) + f'(x_i)\delta x$$

$$\delta x \approx -\frac{f(x_i)}{f'(x_i)}$$

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

- 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:

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

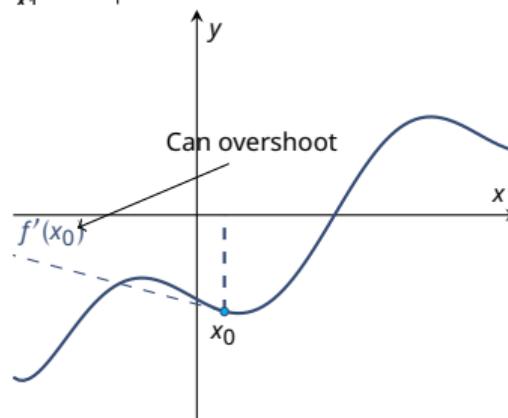
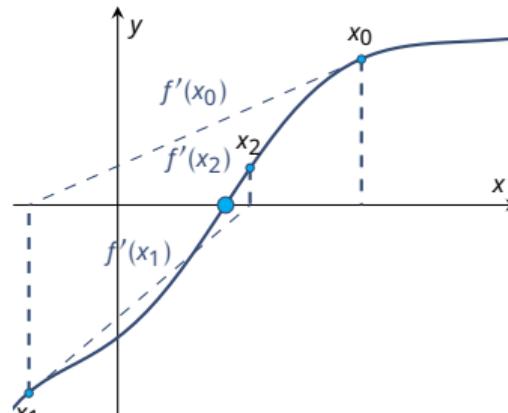
When it works:

- Converges enormously fast when it functions correctly.

When it does not work:

- Underrelaxation can sometimes be helpful
 - Underrelaxation formula:

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



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

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

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

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

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

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

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

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 ε 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 ε 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 :

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

- 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 :

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

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 ε 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.

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:

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

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

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

```
1 def newton1D(f, df, x0, tol, max_iter):
2     x = x0
3     e = [0] * max_iter
4     p = float('nan')
5     for i in range(max_iter):
6         x_new = x - f(x) / df(x)
7         e[i] = abs(x_new - x)
8         if i >= 2:
9             p = (log(e[i]) - log(e[i - 1])) / (log(e[i - 1]) - log(e[i - 2]))
10            print(f'x: {x_new:.10f}, e: {e[i]:.10f}, p: {p:.10f}')
11            if e[i] < tol:
12                break
13            x = x_new
14    return x
```

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

```
1 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 in Python

Exercise 6: Numerical Differentiation

```
1 from math import log
2 def newton1Dnum(f, h, x0, tol, max_iter):
3     x = x0
4     e = [0] * max_iter
5     p = float('nan')
6     for i in range(max_iter):
7         x_new = x - f(x) / ((f(x + h) - f(x)) / h) # NUMERICAL DIFFERENTIATION
8         e[i] = abs(x_new - x)
9         if i >= 2:
10             p = (log(e[i]) - log(e[i - 1])) / (log(e[i - 1]) - log(e[i - 2]))
11             print(f'x: {x_new:.10f}, e: {e[i]:.10f}, p: {p:.10f}')
12             if e[i] < tol:
13                 break
14         x = x_new
15     return x
```

- A command involving numerical differentiation in Python:

```
1 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 δx and δy :

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

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

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 \mathbf{f}

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

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:

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

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 \mathbf{f} :

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

Solution via Cramer's rule:

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

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

- Solutions for δx and δy :

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

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

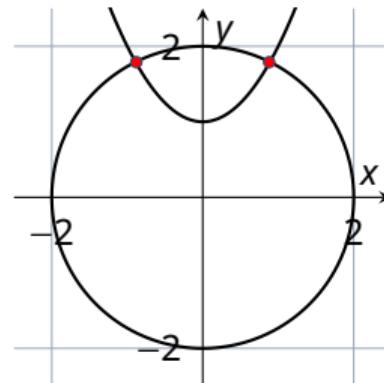
Newton-Raphson Method: multi-dimensional case

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



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

$$① f_i(x_1, x_2, \dots, x_N) = 0$$

$$② \mathbf{x} = [x_1, x_2, \dots, x_N] \quad \mathbf{f} = [f_1, f_2, \dots, f_N]$$

$$③ f_i(\mathbf{x} + \delta\mathbf{x}) = f_i(\mathbf{x}) + \sum_{j=1}^N \frac{\partial f_i}{\partial x_j} \delta x_j + O(\delta\mathbf{x}^2)$$

$$④ J_{ij} = \frac{\partial f_i}{\partial x_j} f(\mathbf{x} + \delta \mathbf{x}) = f(\mathbf{x}) + \mathbf{J} \delta \mathbf{x} + O(\delta \mathbf{x}^2)$$

$$⑤ \mathbf{J} \cdot \delta \mathbf{x} = -\mathbf{f}(\mathbf{x})$$

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

Newton-Raphson Method

Multi-variate Newton-Raphson in Python:

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

```
1 def my_jac(x):
2     jac = np.zeros((2, 2))
3     jac[0, 0] = 2 * x[0]
4     jac[0, 1] = 2 * x[1]
5     jac[1, 0] = 2 * x[0]
6     jac[1, 1] = -1
7     return jac
```

```
1 import numpy as np
2 def newton_nd(f, x0, tol, max_iter):
3     x = np.array(x0)
4     err = np.zeros(max_iter)
5     p = np.zeros(max_iter)
6     for i in range(max_iter):
7         delta_x = -np.linalg.solve(J(x), f(x))
8         x += delta_x
9         err[i] = np.linalg.norm(delta_x)
10        if i > 0:
11            p[i] = np.log(err[i]) / np.log(err[i-1])
12        else:
13            p[i] = float('nan')
14        print(f'i = {i}: x = {x}, err = {err[i]:.6e}, p = {p[i]:.6f}')
15        if err[i] < tol:
16            break
17    return x
```

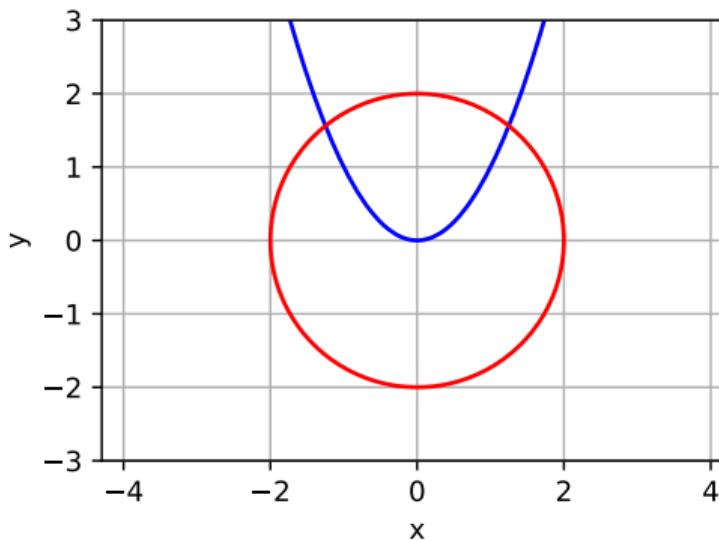
```
1 newton_nd(my_equations, my_jac, [1, 2], 1e-12, 100)
```

Newton-Raphson Method

Multi-variate Newton-Raphson in Python:

Plotting the functions:

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



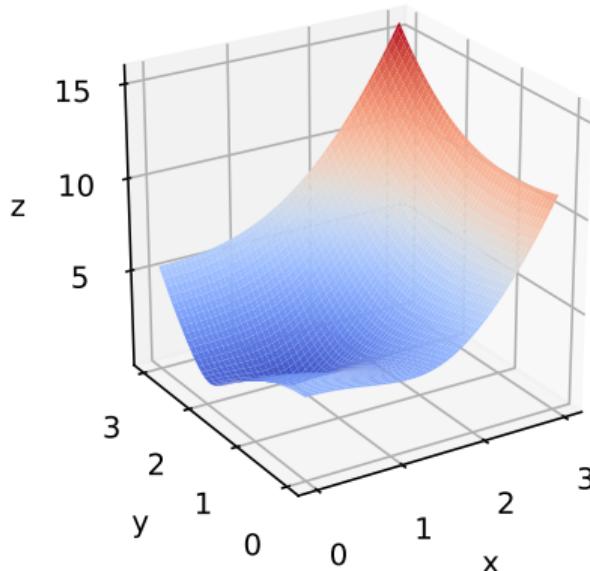
- Code can be found in `plot_implicit.py`
- Uses contour plot at $f(x,y) = 0$

Newton-Raphson Method

Multi-variate Newton-Raphson in Python:

Plotting the norm of the function:

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



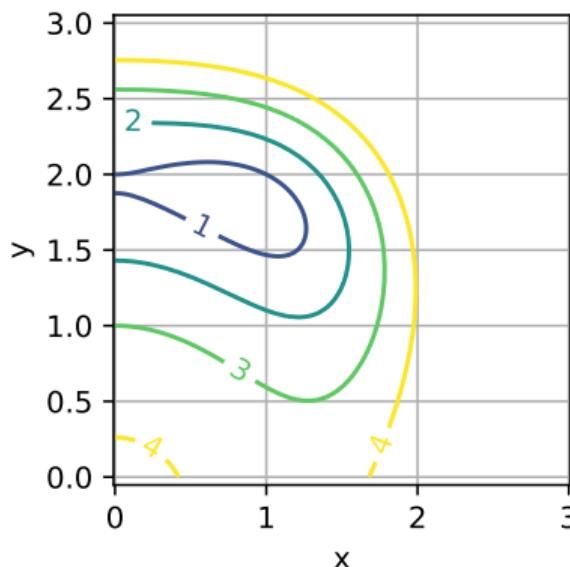
- Code can be found in `plot_implicit.py`
- Uses contour plot at $f(x,y) = 0$

Newton-Raphson Method

Multi-variate Newton-Raphson in Python:

Plotting the norm of the function:

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

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

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 \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:

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

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 $\delta \mathbf{x}^n$?

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

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

- 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!

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

- Requires 11 iterations (compare with Newton: 5 iterations)
But much fewer function evaluations per iteration!

```
1 import numpy as np
2 from numpy.linalg import inv
3
4 def broyden(F, x0, tol=1e-6, max_iter=100):
5     x = np.array(x0)
6     B = np.eye(x.size)
7     for i in range(max_iter):
8         Fx = F(x)
9         if np.linalg.norm(Fx) < tol:
10             print(f"Converged after {i} iterations.
11                 ")
12             return x
13     x_new = x - inv(B)@Fx
14     delta_x = x_new - x
15     delta_Fx = F(x_new) - Fx
16     B += np.outer((delta_Fx - B@delta_x)/(
17                     delta_x@delta_x), delta_x)
18     x = x_new
19     print("Max iterations reached.")
20     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:

$$\begin{array}{c} \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix} \rightarrow \\ \begin{bmatrix} 5.290 & -3.864 \\ 2.493 & -2.934 \end{bmatrix} \rightarrow \\ \begin{bmatrix} -0.384 & -5.879 \\ 2.500 & -3.884 \end{bmatrix} \rightarrow \\ \begin{bmatrix} 3.116 & 3.238 \\ 2.912 & -1.458 \end{bmatrix} \rightarrow \end{array} \quad \begin{array}{c} \begin{bmatrix} 3. & -1. \\ 4. & -1. \end{bmatrix} \rightarrow \\ \begin{bmatrix} 7.363 & -1.931 \\ 3.556 & -1.943 \end{bmatrix} \rightarrow \\ \begin{bmatrix} 10.416 & 6.344 \\ 5.947 & 0.018 \end{bmatrix} \rightarrow \\ \begin{bmatrix} 1.992 & 3.272 \\ 1.989 & -1.430 \end{bmatrix} \rightarrow \end{array} \quad \begin{array}{c} \begin{bmatrix} -1.0 & -9.0 \\ 3.4 & -2.2 \end{bmatrix} \rightarrow \\ \begin{bmatrix} 2.349 & -0.773 \\ 3.547 & -1.941 \end{bmatrix} \rightarrow \\ \begin{bmatrix} 9.781 & 5.515 \\ 5.641 & -0.382 \end{bmatrix} \rightarrow \\ \dots \rightarrow \end{array} \quad \begin{array}{c} \begin{bmatrix} -1.062 & -9.260 \\ 3.411 & -2.154 \end{bmatrix} \rightarrow \\ \begin{bmatrix} -0.934 & -6.772 \\ 2.351 & -4.124 \end{bmatrix} \rightarrow \\ \begin{bmatrix} 3.577 & 3.630 \\ 3.362 & -1.074 \end{bmatrix} \rightarrow \\ \dots \rightarrow \end{array}$$

- 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!).