

Numerical Methods in the Python Ecosystem

<https://jankoslavic.github.io/pyNumericalMethods/>

Janko Slavič

October 13, 2026

Contents

0	Study and Examination Rules: 2026/27	1
0.1	Lectures / exercises	1
0.2	Continuous study	1
0.3	Exam	7
0.4	References	7
0.5	Requirements for taking the course in advance?	8
1	Introduction to Python	9
1.1	What is the Python ecosystem?	9
1.2	Who learns Python, and why use it?	11
1.3	Uses of Python?	12
1.4	Python vs Matlab	12
1.5	Installing Python	12
1.6	Jupyter notebook (Jupyter lab)	13
1.7	First program and PEP	16
1.8	Python basics	18
1.9	To conclude: a cheat sheet :)	31
2	Functions, exceptions, modules	32
2.1	Functions	32
2.2	Handling exceptions	40
2.3	Basics of modules	41
2.4	Using modules from the internet	44
2.5	Additional	45
2.6	Formatting date and time	45
3	Modules, NumPy, Matplotlib	48
3.1	Basics of the <code>numpy</code> module	48
3.2	Basics of the <code>matplotlib</code> module	60
3.3	Additional:	66
3.4	<code>Numba</code>	66
3.5	<code>matplotlib</code> : animations, callback, XKCD style	66
3.6	For the most enthusiastic	66
4	Object-oriented programming, symbolic computation	67
4.1	Object-oriented programming	67
4.2	Symbolic computation with <code>SymPy</code>	71
4.3	Additional	91
4.4	<code>sympy.mechanics</code>	91

5	Introduction to numerical methods and systems of linear equations	92
5.1	Introduction to numerical methods	92
5.2	Introduction to systems of linear equations	93
5.3	Gaussian elimination	99
5.4	Additional	102
5.5	Example of symbolic solving of a system of linear equations with <code>sympy</code>	102
6	Systems of linear equations - continued	103
6.1	LU decomposition	103
6.2	Pivoting	106
6.3	The <code>SciPy</code> module	110
6.4	Computing the inverse matrix	111
6.5	Solving overdetermined systems	112
6.6	Iterative methods	115
7	Interpolation	117
7.1	Introduction	117
7.2	Polynomial interpolation	118
7.3	Lagrange method	119
7.4	Cubic splines	125
7.5	Additional	128
7.6	A few comments on the <code>scipy.interpolate</code> module	128
8	Approximation (curve fitting)	131
8.1	Introduction	131
8.2	Least-squares method for a linear function	132
8.3	Least-squares method for an arbitrary polynomial	135
8.4	Approximation with an arbitrary function	140
8.5	Additional	141
9	Root finding	145
9.1	Introduction	145
9.2	Incremental method	147
9.3	Iterative incremental method	149
9.4	Bisection method	151
9.5	Secant method	154
9.6	Newton's method	158
9.7	Solving systems of nonlinear equations	161
9.8	Additional	164
10	Numerical differentiation	165
10.1	Introduction	165
10.2	Approximating the first derivative using the finite-difference method	166
10.3	Central difference scheme	168
10.4	Non-central difference scheme	175
10.5	Using <code>numpy.gradient</code>	176
10.6	Round-off error in numerical differentiation	178
10.7	Complex-step derivative approximation	182
10.8	Additional	184

11 Numerical integration	185
11.1 Introduction	185
11.2 Newton-Cotes approach	187
11.3 Gaussian integration approach	202
11.4 <code>scipy.integrate</code>	207
12 Numerical solution of differential equations - initial value problem	210
12.1 Introduction	210
12.2 Euler's method	210
12.3 Second-order Runge–Kutta method	216
12.4 Fourth-order Runge–Kutta method	218
12.5 Using <code>scipy</code> to solve ordinary differential equations	221
12.6 System of ordinary differential equations	223
12.7 Stability of solving differential equations*	229
12.8 Additional: Example of the Van der Pol oscillator	232
12.9 Additional: symbolic solving of a second-order differential equation	233
13 Numerical solution of differential equations - boundary value problem	238
13.1 Solving two-point boundary value problems	238
13.2 Shooting method	238
13.3 Finite difference method	248
13.4 Additional: symbolic solution of the beam	255
14 Testing code correctness, user interface	257
14.1 Testing code correctness	257
14.2 Graphical user interface	259
14.3 A few review questions!	262
15 References	263

Chapter 0

Study and Examination Rules: 2026/27

0.1 Lectures / exercises

Lectures:

- Monday 8:15 V/2: Prof. Janko Slavič, PhD (DS-P3, 01 4771 226, janko.slavic@fs.uni-lj.si)

Exercises:

Time	Asst. Domen Gorjup, PhD	Asst. Marko Zupan	Asst. Gal Stanovnik
Thursday			
8:00	T1 (I/4b)	T2 (RU SP-24)	
10:00	T3 (I/4b)	T4 (RU SP-24)	
Friday			
8:00		F1 (I/4b)	F2 (RU SP-24)
12:00	F3 (I/4b)		F4 (RU SP-24)

Office hours:

- Thursday 11:30 - 12:00 in classroom I/4b, by prior arrangement with the assistant.

Assistants:

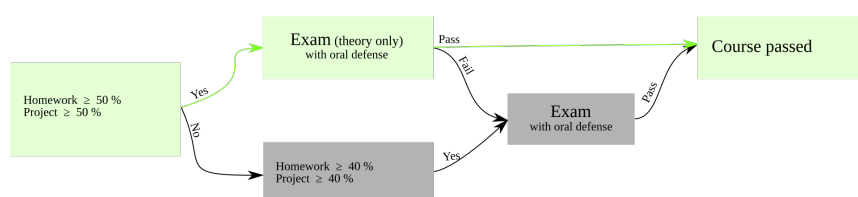
- Asst. Domen Gorjup, PhD (DS-P5, 01 4771 228, domen.gorjup@fs.uni-lj.si)
- Asst. Marko Zupan (DS-P5, 01 4771 230, marko.zupan@fs.uni-lj.si)
- Asst. Gal Stanovnik (DS-P5, 01 4771 230, gal.stanovnik@fs.uni-lj.si)

Tracking continuous study, announcements, etc.

We will run continuous study with the help of: moj.ladisk.si. To log in, use the email you have in VIS: [video walkthrough of logging in](#).

Course-related materials can also be found in the [online classroom](#).

0.2 Continuous study



0.2.1 Grade composition

- 5% participation in lectures
- 10% weekly homework (at least 50%)
- 40% individual project (at least 50%)
- 45% theory test (at least 50%): [see past tests](#)

You must come to lectures and exercises prepared; we will use some flipped-classroom principles!

0.2.2 Exercise templates

For each exercise, a **template** (Jupyter notebook) with problem definitions and prepared code is available. Download the template before the exercise and complete it during the exercise with your own notes and solutions.

The exercise templates are in the [Exercise templates](#) folder.

0.2.3 Individual project

It relates to any content of any course at the Faculty of Mechanical Engineering and is defined by the student themselves. The project must include the following content:

- *symbolic computation,*
- *systems of linear equations,*
- *interpolation or approximation,*
- *root finding,*
- *integration or differentiation,*
- *solving differential equations.*

0.2.4 Use of artificial intelligence in the project

In the individual project, **we expect you to use artificial intelligence tools** (AI; e.g., Claude, ChatGPT, Copilot): to explain methods, write and review code, find bugs, and format the report. With AI you can achieve considerably more in the available time than without it, so the expectations regarding the scope of the project are adjusted accordingly.

Expected scope of the project

For a grade of 10, we expect a project for which **the time invested without AI would substantially exceed the scope of this course**. This is a measure of the scope and quality of the result, not of the time you spend on the project: with AI, such a project is feasible within the expected workload of the course.

Such a project typically has:

- a **real engineering problem** (e.g., from another course, from practice, or from a laboratory), not a textbook exercise made up just to use the prescribed methods,
- numerical content that is **connected** into a whole (e.g., interpolated measurements are the input to a differential equation, whose solution is then compared with the measurements),
- an **analysis of the influence of parameters** or a comparison of several methods with an estimate of error and convergence,
- **verified results** (analytical solution, another method, measurements, literature),
- a usable software product: a module with tests, a user interface, an animation, a web application, data acquisition, etc.

Approximate project grade by scope:

Grade	Expected scope
6–7	All six numerical topics are correctly applied and explained, but they are poorly connected; the problem is simple.
8	A meaningful whole, results are verified, some analysis of the influence of parameters.
9	A real problem, connected methods, comparison of methods and error estimates, a well-developed software part (module, tests).
10	As for 9, with a scope for which the time invested without AI would substantially exceed the scope of this course: e.g., real data or measurements, several connected models, a user interface or application, an in-depth critical analysis.

Your responsibility when using AI

AI is a tool; you are the author of the project and the person responsible for it.

- **You must understand and be able to explain everything you submit:** every method, every part of the code, and every result. At the defense, we may ask you about any part of the project.
- **Verify the results.** AI often suggests an inappropriate method, a wrong boundary condition, or code that runs but computes the wrong thing. Verifying the results is part of the grade.
- **Describe your use of AI** in the *Use of artificial intelligence* section at the end of the report (5–10 lines): which tools you used and for what, where the AI was wrong and how you found out, and what you did yourself.
- We still check for **plagiarism** (see below). The fact that the code was written by AI is no excuse for matching another project; if you start from your own problem, matching does not occur.
- The use of AI is **not allowed at the exam**. It therefore makes sense to let AI help you understand the project, not just write the code.

0.2.5 What we grade in the individual project

The purpose of the project is to demonstrate the **appropriate selection and use of numerical methods, to comment on how they work, and to evaluate the obtained results**. We grade four areas:

- **Numerical methods: 45%**
 - appropriateness and correctness of the methods used: 30%,
 - reasoning and comments: 15%.
- **Report and code: 25%.**
- **Personal approach and creative addition: 25%.**
- **Code tests and/or user interface: 5%.**

What we look at in each area is described in more detail below.

Grading criteria by area

1. Numerical methods (each of the six prescribed sections is graded separately)

- *Correctness and appropriateness of the method:*
 - the problem is solved **numerically**; a purely symbolic solution (**sympy**) or just writing down the equations is not enough,

- the method suits the problem (e.g., solve an overdetermined system with the least-squares method, do not approximate a periodic signal with a polynomial, convert a higher-order differential equation into a system of first-order equations, etc.),
- where possible, the result is **verified**: by comparison with an analytical solution, another numerical method, or an existing function from `numpy/scipy` (this also applies to solutions suggested by AI),
- using existing functions (`numpy`, `scipy`) is encouraged; your own implementation must be justified and compared with an existing function.
- *Reasoning and comments*: for every method you use, explain in writing:
 - **why** you chose this method and not another (e.g., Newton’s method or bisection? why LU decomposition? why splines and not a polynomial? etc.),
 - **briefly**: what the assumptions of the method are and what its **limitations** are (accuracy, convergence, order of the method, choice of step size, end points in differentiation ...),
 - what the result means for your physical problem (**interpretation**).
 - **Important**: in the project, **do not repeat derivations or the theoretical background of the methods**, etc., unless this is essential for understanding or running your code. Theory is the subject of the written part of the exam.

2. Report and code

- *Report*:
 - basic information (first and last name, enrollment number, date) and an authorship statement,
 - problem definition with a description and a **figure** of the problem, and a list of sources,
 - sections following the [template](#) (see also the [example project](#)); the headings make it clear where each numerical topic is covered,
 - explanations in **markdown** cells, equations written in LaTeX,
 - results presented graphically (`matplotlib`), with labeled axes and units,
 - a short description of the use of AI in the *Use of artificial intelligence* section.
- *Code*:
 - after **Restart** → **Run All** the report runs **without errors**, file paths are relative (all required files are included, the code works anywhere, not only on your computer),
 - the code is readable and in a sensible order,
 - functions used more than once are collected in **your own module** (`.py`), which you import into the report,
 - functions have **docstrings** (description, parameters, return value) and meaningful comments,
 - additional packages are listed with installation instructions (e.g., `requirements.txt`).

3. Personal approach and creative addition

- *Scope and ambition*: see *Expected scope of the project* above; for a grade of 10, we expect a scope for which the time invested without AI would substantially exceed the scope of this course.
- *Personal approach*: the problem is a meaningful whole and not a set of unrelated tasks; the numerical content arises from the problem; the analysis is thorough (e.g., influence of parameters, comparison of methods, error estimate); the results are critically evaluated (are they physically meaningful?).
- *Creative addition*: content that was not covered in lectures and exercises (see the list below). We grade the quality of the implementation, not just its presence; for grades 9 and 10, a creative addition is expected.

4. Code tests and/or user interface

At least one test of your own functions (e.g., with `numpy.testing` or `pytest`) or a simple user interface. The test must be run in the report and must pass.

The most common shortcomings of past projects that led to lower grades:

- a method is used, but it is not explained why this one, how it works, and how accurate the result is,
- a section is solved only symbolically,
- an own implementation of a method without justification and without comparison with an existing function,
- results without interpretation,
- the code does not run because of wrong paths or missing packages,
- code written by AI that the student cannot explain at the defense,
- results that nobody has verified.

Creative addition: examples of content

A **creative addition** is treated as programming content that was not specifically covered in lectures and exercises. With AI tools, such additions are much more accessible, so we expect them for grades 9 and 10. For example:

- animation of results using matplotlib,
- other forms of advanced visualization (Plotly, Bokeh, vispy, pyqtgraph, VTK...),
- automatic reading of data from the web, sending results by email,
- machine-generated reports,
- use of the Pandas library, writing results to a database, MS Excel,
- use of the Raspberry Pi or Arduino platforms, data acquisition,
- building a web application (Flask, Django),
- use of modules and libraries such as scikit-learn, TensorFlow, PyOpenCV.

Instructions for submitting the project

- The deadline for submitting the project is listed below, among the important dates.
- To create the project, use the [template](#); for guidance, see the [example project](#).
- Submit the project as a zip file named: **First and last name, enrollment number.zip**.
Example: Janez Novak, 23201111.zip.
- The size of the zip file should not exceed **20 MB**.
- **Submit the ZIP in the [online classroom](#).**

Before submitting:

1. in the project, comment out (add `#`) any installation of additional packages and functions such as `%%timeit`:

```
# !pip install package_name
# %%timeit
```

2. Check that the paths to images and other files are given in *relative* form (e.g., `'image.jpg'`, **not** `'D:\Numerical methods\project\image.jpg'`).
3. do Kernel → Restart and then Cell → Run All. Make sure that all cells execute correctly.

Plagiarism in the individual project

All submitted individual projects are checked with a plagiarism-detection system.

The system checks the similarity of submitted documents against all other projects from the current and past academic years. Any matching of text and of procedures in the submitted code is flagged as plagiarism (even if it is used in a different context, a different order, with renamed variables, etc.). The content of the lecture templates published in this repository is excluded from the check.

If a significant part of the content of your project matches another project, we cannot grade it as your own independent work. This means that you **have not completed the exercises and do not meet the requirements to take the exam**.

0.2.6 Project grading procedure

The project is graded in two steps: automated review and oral defense

1. Automated review of the submitted reports.

All submitted reports are first reviewed automatically: we check whether the report runs without errors and whether it contains all the prescribed content, and we compare it with the other projects (see *Plagiarism* above).

2. Oral defense with the course instructor, about **10 minutes per student**.

The student briefly presents the project (the problem, the chosen methods, the key results), and the instructor then asks a few questions about the details, e.g.: why you chose a particular method, how your code works, how you verified the result, how you used AI. We may also ask you to make a small change to the code (e.g., change a parameter or a boundary condition) and show its effect.

The final project grade is based on both steps. A project that the student cannot explain at the defense cannot be graded as independent work.

The defense of individual projects will take place approximately 1 week after submission.

0.2.7 Homework (moj.ladisk.si)

We assign weekly homework in the moj.ladisk.si system. The homework assignments are Jupyter Notebook documents (`.ipynb`); you solve them in the Visual Studio Code environment.

The use of AI for homework is allowed, but we advise you to solve it yourself: AI is not available at the exam, and the homework is the main preparation for it.

General instructions for solving

- Save the files you download from the web as a `.zip` archive (if your browser does not add the `.zip` extension to the file, do it yourself).
- **Fully extract** the archive of files you download from the web.
- In the folder with the extracted files, open a command window and run the command `jupyter notebook`.
- Run the cells in the `.ipynb` homework file in order (including the cells at the very top of the file!).
- When solving, results are sent to the moj.ladisk.si online system and are automatically checked on the fly. An **internet connection is required** when solving.

We have also prepared [video instructions for solving](#).

Time limit for solving homework

Homework has a defined *solving window* time:

- Until the solving window opens, **early solving of the assignment without a time limit** is possible.
- During the solving window, the allowed time for solving the assignment is limited. The time limit starts from the moment you first open the assignment *within the solving window*.
- After the window closes, the assignment closes and solving is **no longer possible**.

We advise you to *solve as much of the assignment as possible during the early-solving period, before the solving window opens*.

Using a Joker to postpone an obligation

Each student has two Jokers available in the course, which you can use to postpone an obligation to the future.

- Using a Joker, you can move the solving window of a homework assignment forward by ≤ 14 days.
- Using a Joker is **not possible if you have already opened the assignment**.
- Before the start of the solving window, one Joker is needed to move the assignment; during the solving window, two are needed.

0.2.8 Proposed grade based on continuous study

Proposed grade:

- 50 to 60%: 6
- 60 to 70%: 7
- 70 to 80%: 8
- 80 to 90%: 9
- 90% and above: 10

(Undefined ranges are intermediate grades.)

The individual project and the proposed grade are defended in an oral defense (see *Project grading procedure*).

In the defense, we also take into account the qualitative assessment of participation in exercises and lectures.

If you meet the continuous-work requirements (homework, project), you have **one** opportunity to take the test on theory only at the winter exam dates after the end of the semester.

0.2.9 Important dates

1. Submission of the individual project: **Jan 12, 2027 by 12:00 ***
2. The theory test is taken on any winter exam date (you have one opportunity).
3. Defenses: after the exam date.

* If submitted at least 1 week **before the deadline**, the result is multiplied by 1.1; in the case of a **late submission**, the result is multiplied by 0.9^n , where n is the number of started days of delay.

0.3 Exam

Meeting the requirements to take the exam: each individual event (except theory and participation in lectures) that makes up the grade must be greater than 40%.

The exam is taken in two parts: 60 min for three theoretical questions (see a [sample exam](#)) and 50 min of testing the application of numerical methods on the computer (typically two problems; you may bring any written sources, but there is no internet access).

The use of AI tools is **not allowed** in either part of the exam.

For a passing grade, both parts must be passed.

0.4 References

1. Slavič J.: Programiranje in numerične metode s Pythonom, [executable book](#) or [online book](#), 2015-
2. Petrišič J.: Uvod v Matlab za inženirje, 2011
3. Demšar J. Python za programerje, 2012 (see the [online classroom at FRI](#))
4. Nicolas P. Rougier: From Python to Numpy, 2017 [online book](#)
5. Kiusalaas J: Numerical Methods in Engineering with Python 3, 2013
6. Fangohr H. Python for Computational Science and Engineering, 2014 ([available online](#))
7. Bucky R. Python Programming Tutorials ([videos on YouTube](#))
8. [Projekt TOMO](#) (more content; see especially the courses at FMF and FRI)
9. [Scientific Python Lectures](#) (formerly *Scipy Lecture Notes*)
10. Using the Numpy, Scipy, etc. packages for engineers [Mr. P Solver \(YouTube\)](#)

You can find some online classrooms on the topic of Python at pinm.ladisk.si

0.5 Requirements for taking the course in advance?

The course cannot be taken in advance.

Chapter 1

Introduction to Python

1.1 What is the Python ecosystem?

Many readers find the word *ecosystem* confusing. The [Dictionary of the Slovenian Literary Language](#) today recognizes only the usage related to ecology. In computer science, however, over the past few decades the word ecosystem has also been used in the context of the so-called digital ecosystem; in the case of Python this means the entire collection of different packages ([pypi.org](#) alone currently hosts more than 685 thousand different packages) that are built on the Python programming language and are interconnected.

The Python ecosystem is much more than just the Python programming language! In the first four lectures, this book focuses on the part of this ecosystem that is interesting and useful primarily for engineering professions.

Python is a **high-level** programming language, with a high level of abstraction. **Low-level** programming languages, in contrast, have a low level of abstraction (e.g., machine code).

Let us look at the computation of a [Fibonacci number](#), which follows the definition:

$$F_1 = F_2 = 1 \quad \text{in} \quad F_n = F_{n-1} + F_{n-2}. \quad (1.1)$$

```
8B542408 83FA0077 06B80000 0000C383
FA027706 B8010000 00C353BB 01000000
B9010000 008D0419 83FA0376 078BD98B
C84AEBF1 5BC3
```

Computation of F_n in [machine code](#):

```

fib:
    mov edx, [esp+8]
    cmp edx, 0
    ja @f
    mov eax, 0
    ret

    @@:
    cmp edx, 2
    ja @f
    mov eax, 1
    ret

    @@:
    push ebx

```

MASM code

C

```

int Fibonacci(int n)
{
    if ( n == 0 )
        return 0;
    else if ( n == 1 )
        return 1;
    else
        return ( Fibonacci(n -1) + Fibonacci(n -2) );
}

```

Python

```

def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n -1) + fib(n -2)

numbers = [1., 2, 3, 4, 5] # list of numbers
[number**2 for number in numbers if number>3]

[16, 25]

```

Python enables a high level of abstraction and therefore abstract code. Now we will look at an example of Python code:

```

numbers = [1, 2, 3, 4, 5] # list of numbers
[number for number in numbers if number%2]

```

Even though at this point we do not yet have enough knowledge, let us try to understand/read it:

1. the first line defines a list of numbers; everything that follows the `#` character is a comment
2. the result of the second line is a list, which is defined by the square brackets; inside the brackets we read:
 - return `number` for each (**for**) `number` in (**in**) the list `numbers`, if (**if**) dividing `number` by 2 leaves a remainder of 1.

We can see that the level of abstraction really is high, since we needed a lot of words to explain the content!

Let us look at the result:

```
numbers = [1, 2, 3, 4, 5] # list of numbers
[number for number in numbers if number%2]

[1, 3, 5]
```

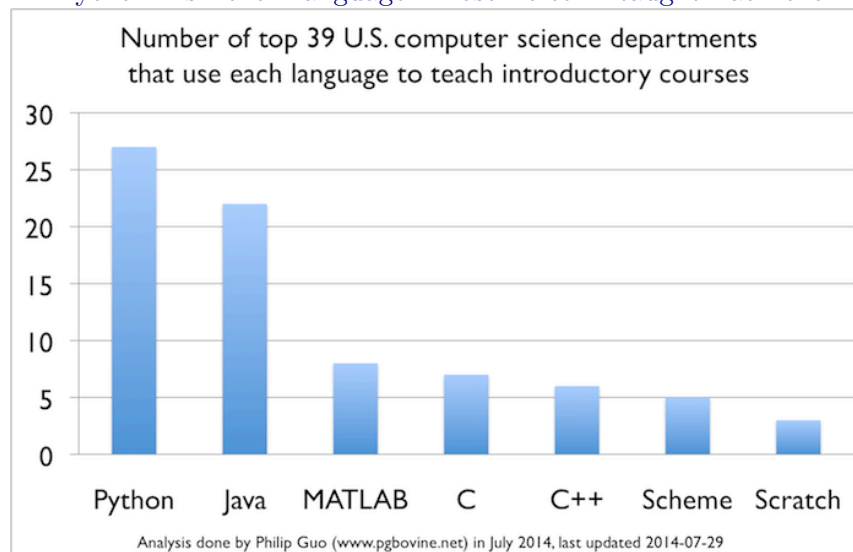
A brief overview of the beginnings of some programming languages:

- 1957 Fortran, * 1970 Pascal, * 1972 C
- 1980 C++, * 1984 Matlab, * 1986 Objective-C
- **1991 Python**, * 1995 Java, * 1995 Delphi
- 2006 Rust, 2009 Go, 2012 Julia

1.2 Who learns Python, and why use it?

USA

[Python is the language most often taught at the best American universities.](#)



At the University of Berkeley at the [start of the 2017/18 academic year](#) (more than 1200 students in the *Data science* course, which is based on the Jupyter notebook technology).

```
%%html
<blockquote class="twitter -tweet" data -lang="en"><p lang="en" dir="ltr">First day
of class in Foundations of <a
href="https://twitter.com/hashtag/Datascience?src=hash">#Datascience</a> for <a
href="https://twitter.com/BerkeleyDataSci">@BerkeleyDataSci</a> - all materials at <a
href="https://t.co/yYoJbaKlnL">https://t.co/yYoJbaKlnL</a> <a
href="https://t.co/J6DLd0VQTl">pic.twitter.com/J6DLd0VQTl</a></p>&mdash; Cathryn
Carson (@CathrynCarson) <a
href="https://twitter.com/CathrynCarson/status/900408152348270592">August 23,
2017</a></blockquote>
<script async src="//platform.twitter.com/widgets.js" charset="utf -8"></script>
```

First day of class in Foundations of #Datascience for @BerkeleyDataSci - all materials at <https://t.co/yYoJbaKlnL> pic.twitter.com/J6DLd0VQTl
 — Cathryn Carson (@CathrynCarson) August 23, 2017

Over the past 10 years Python has been rapidly gaining popularity and is today one of the best ([IEEE Spectrum](#).) and also one of the most popular programming languages ([Why is Python Growing so Quickly?](#) and [Python's explosive growth](#)).

1.3 Uses of Python?

Some international companies: *Nasa, Google, Cern*.

Some companies in Slovenia: *Gorenje, Kolektor, Iskra Mehanizmi, Hidria, Mahle, Domel, ebm-papst Slovenia, Efaflex...*

More and more programs have built-in scripting support for Python: *Abaqus, Ansys Workbench, MSC (Adams, SimXpert), MS Excel, ParaView, LMS Siemens, AVL*.

Some commercial programs written in Python: *BitTorrent, Dropbox*.

An extensive list is available [here](#).

1.4 Python vs Matlab

In the past, Matlab was often taught at universities and also had many users in industry. For years now, the Python ecosystem has been a substantially more powerful tool for most tasks. You will find a huge number of comparisons online, e.g.: [Python vs Matlab](#).

A short comparison based on this [source](#):

Selected criteria:

	Fortran	C	C++	Matlab	Python
performance	+	+	+	o	o
object orientation	-	-	+	-	+
exceptions	-	-	+	-	+
open source	+	+	+	-	+
easy to learn	o+	o	o-	+	+

legend:

+	=	good/yes	o
o	=	so-so	
-	=	poor/no	

The syntax is generally very similar. An important difference is that in **Matlab** indices start at 1, while in **Python** they start at 0.

If you are younger than 25-30 years, you can probably skip the rest of this cell.

If you are not, you probably learned Matlab and are wondering why your club has fewer and fewer members. It is probably time to switch to Python; here is an article that can help you: [MATLAB vs Python: Why and How to Make the Switch](#).

1.5 Installing Python

Go to www.python.org and install Python; *choose the latest version (currently 3.14)* (64 bit). Follow the prepared [video instructions](#).

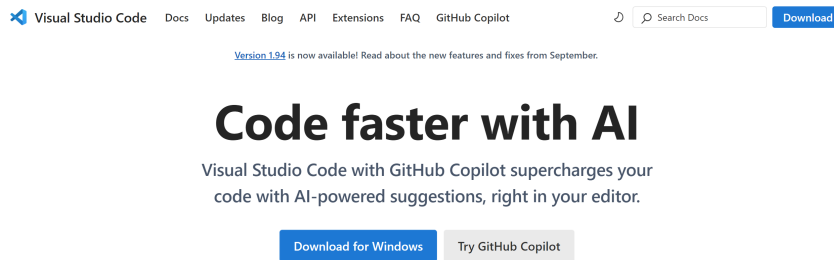
1.5.1 Using the GitHub repository and updates

This book is continuously being updated; the latest version is available at:

- in source form at github.com/jankoslavic/pynm,
- in web form at <https://jankoslavic.github.io/pynm>.

To synchronize with the online repository, it is best to use one of the Git clients, e.g., [GitHub Desktop](#) or the one within Visual Studio Code (presented later). Git is, in fact, a very powerful tool and allows a group of developers to work relatively easily on the same code and to keep control over different versions of the files.

1.5.2 Visual Studio Code



In this course we will mostly program in Jupyter Lab/Notebook within [Visual Studio Code - VSC](#), which enables very efficient programming using fully formatted text and interactive displays. To install VSC, follow the prepared [video instructions](#).

When we want to program more extensive and more generally useful packages and dedicated programs, we usually also make use of other features that the VSC integrated development environment (IDE) offers. In recent times VSC has been gaining popularity because of its speed, ease of use, and extensibility. How VSC can help us with programming is shown excellently in a video on the topic of [productivity with VSC](#).

1.5.3 Installing add-ons and updates

[pypi.org](#) hosts many different packages! We install them from the command prompt* using the **pip** command.

* You can open the command prompt in the Windows operating system by clicking "Start" and typing "command prompt" and pressing "enter", or by opening a terminal within VSC ([video](#)).

We will look at the details of installing add-ons next time. To make sure you have the latest version of the packages, do the following:

1. Open the command prompt,
2. run the command to update the packages we will need throughout the course:

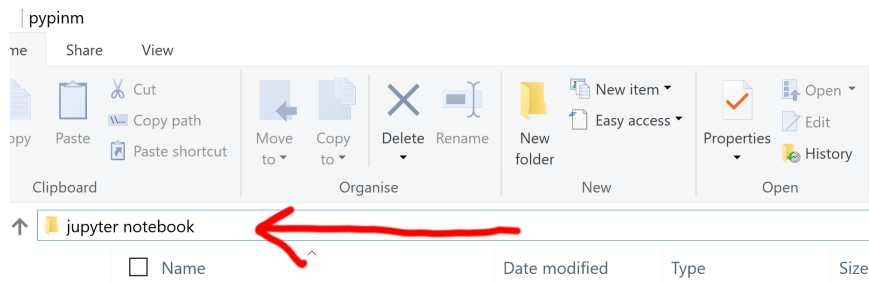
```
| pip install numpy scipy matplotlib sympy requests jupyter - --upgrade
```

1.6 Jupyter notebook (Jupyter lab)

[Jupyter notebook](#) is an interactive environment that works as a web application and can be used with various programming languages (e.g., Python, Julia, R, C, Fortran ... the name JuPyt[e]R was also formed from the first three).

The easiest way to try out Jupyter notebook is here: try.jupyter.org or colab.research.google.com.

We start *Jupyter notebook* (in any folder) from the *command prompt* or from the *File Explorer* with the command: `jupyter notebook`,



or within VSC as presented in the [video](#).

Notes:

- by running `jupyter lab` we get an upgrade of the `jupyter notebook` environment (a slightly more demanding environment, but a substantially more advanced one),
- here we assume that the path to the `jupyter[.exe]` file is added to the environment variables (and that you can run it from the command prompt in any folder),
- if you want to open the command prompt in any folder, then in the *File Explorer* hold **shift** and right-click the folder; then choose *Open command window here*.

We access the Jupyter server through a web browser. First you are shown a list of folders and files you can access within the folder in which you ran the command `jupyter notebook`. Then look for the *new* icon in the upper-right corner and click *Python 3*. This will start your first Jupyter notebook (with a Python kernel). Find the menu bar and on the right side click *Help* and then *User interface tour*. Also take a look at *Help/Keyboard shortcuts*.

A Jupyter notebook is composed of so-called *cells*.

A cell has two states:

- **Command mode:** this state is entered by pressing [escape],
- **Edit mode:** this state is entered by pressing [enter].

In both cases a cell is executed by pressing [shift + enter].

Cells can be of different types; here we will use two types:

- **Code:** in this case the cell contains program code (the shortcut in command mode is the [y] key),
- **Markdown:** in this case the cell contains text formatted according to the *Markdown* standard (the shortcut in command mode is the [m] key).

1.6.1 Markdown

Jupyter notebook allows writing in *markdown* mode, which was introduced in 2003-2004 by [John Gruber](#). The purpose of markdown is for the writer to focus on the content, while using a few simple rules for formatting. Later, markdown can be converted into web form, into pdf (e.g., via LaTeX), etc. The essential rules (related mainly to use on GitHub) are [given here](#); below we will briefly review them.

We define headings with the # character:

- # means a first-level heading,
- ## means a second-level heading, etc*.

* If you look at the code, you notice that the \ character appears before the # character so that the # character itself is *displayed* and its function for a first-level heading is not used. The same applies to other special characters.

Text formatting is defined by special characters; if we want a word (or several words) in *italic* form, we have to place such text between two * characters: the text in edit mode in the form `*example*` would be displayed as *example*.

Overview of the symbols that define the format:

- *italic*: *
- **bold**: **

- `~~strikethrough~~`: `~~`
- `symbol`: ```
- `functions`: `"`
- `code block`: `"`"`

Note: for an example of a Python code block, see the introductory section, the Fibonacci example.

Within markdown we can also use LaTeX syntax. LaTeX is a very powerful system for text editing; at the Faculty of Mechanical Engineering, University of Ljubljana, the templates for diploma theses are prepared in LaTeX and in Word. Here we will use LaTeX only for writing mathematical expressions.

There are two modes:

- inline mode: starts and ends with the `\character, example` `:a3=\sqrt{365}$`,
- block mode: starts and ends with the two characters `\`, example: `$$\sum F=m\,,\ddot{x}$$`, which renders as:

$$\sum F = m \ddot{x} \quad (1.2)$$

Above we have already used lists, which we begin with the `*` character on a new line:

- we can write lists
 - with several levels
 - * 3rd. level
 - 2nd. level
- we can write *italic*
- we can write **bold**.

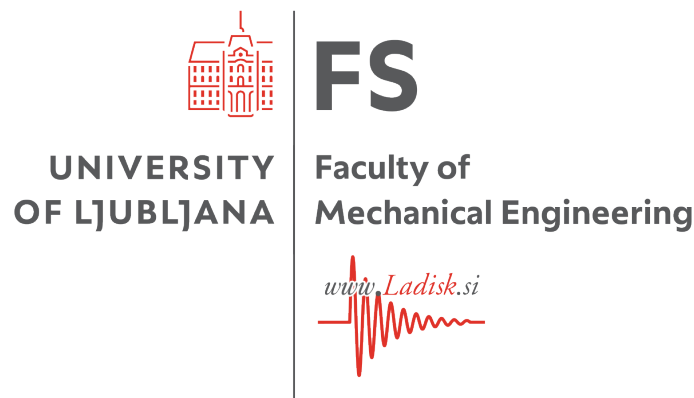
If we want numbered lists, we start with `"1."` and then continue with `"1."`:

1. first element
2. second
3. next ...

We can also include *html* (Hypertext Markup Language) syntax in the document. The simplest thing to include as html are images.

We use:

```
| ![Ladisk (this is the optional image description)](./fig/ladisk.png)
```



For display:

We can also include external resources (e.g., a video from YouTube). Since this is a potential security risk, we do not do it in a *markdown* cell as above, but in a *code* cell, using the so-called magic command `%%html`, which defines that the entire block will be in html syntax (by default, *code* cells contain Python code):

```
%%html
<iframe width="560" height="315" src="https://www.youtube.com/embed/VLcPd07 -gnk"
frameborder="0" allowfullscreen></iframe>
```

There are many more magic commands in the notebook (see the [documentation](#)), and we will learn some of them later, when we need them!

So far we have already several times used references to external resources; we do it like this:

[Link name](address of the resource)

Example: [Website of the Ladisk laboratory](https://www.ladisk.si) is formatted as: [Website of the Ladisk laboratory](https://www.ladisk.si).

Within markdown we can prepare simple tables. Preparation is relatively easy: we simply insert vertical bars `|` between the words where the vertical line should be. The header row is followed by a row that defines the alignment in the table.

Example:

```
| Name | Mass [kg] | Height [cm] |
|: -| -:|: -:|
| Lakotnik | 180 | 140 |
| Trdonja | 30 | 70 |
| Zvitorepec | 40 | 100 |
```

Results in:

Name	Mass [kg]	Height [cm]
Lakotnik	180	140
Trdonja	30	70
Zvitorepec	40	100

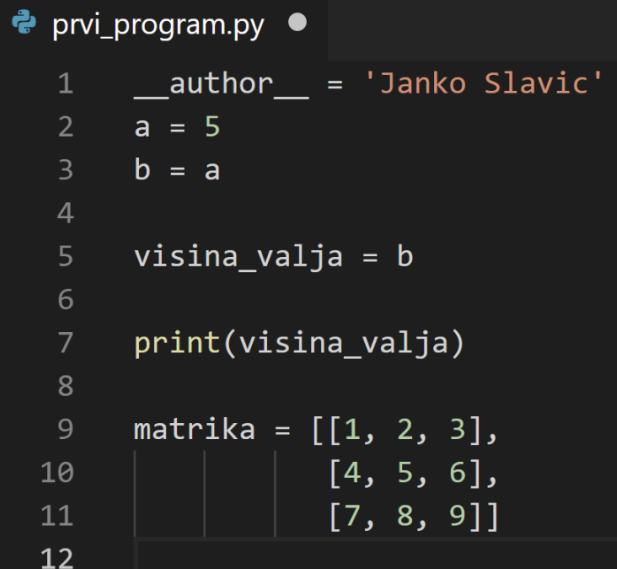
We notice that the colon defines the alignment of the text in the column (left `:-`, right `-:`, or centered `:-:`).

1.7 First program and PEP

Now we are ready for the first program written in Python:

```
a = 5.0
print(a)
```

We can write the program in any text editor (e.g., *Notepad*) and save it in a file `name.py`.



```

prvi_program.py
1  __author__ = 'Janko Slavic'
2  a = 5
3  b = a
4
5  visina_valja = b
6
7  print(visina_valja)
8
9  matrika = [[1, 2, 3],
10             [4, 5, 6],
11             [7, 8, 9]]
12

```

Example [code](#) in the *Visual Studio Code* environment:

We then run this file in the command prompt (we assume that the path to the python.exe file is added to the environment variables; otherwise see this [source](#)):

```
python name.py
>>> 5.0
```

With the characters >>> we denote the result that we get in the command prompt.

The above example of executing a Python program represents the so-called *classic way of running programs* from beginning to end. Here we will more often use the so-called *interactive way*, where we execute code within an individual *code cell*:

```

a = 5.0 # this is a comment
print(a) # prints the value of a

5.0

```

1.7.1 PEP - Python Enhancements Proposal

Python has high standards regarding quality and aesthetics. Early in the development of the language, the principle of proposing improvements through the so-called *Python Enhancements Proposal* (PEP for short) became established.

PEP8

One of the more important ones is [PEP8](#), which defines style:

- indentation: 4 spaces,
- one space around the operator =, that is: `a = 5` and not `a=5`,
- variable and function names should be descriptive, written with underscores: `cylinder_width = 5`,
- for classes we use the so-called CamelCase format.
- for arithmetic operators we place spaces by feel: `x = 3*a + 4*b` or `x = 3 * a + 4 * b`,
- there is no space after an opening parenthesis or before a closing parenthesis, and likewise no space before a parenthesis when calling a function: `x = sin(x)`, but not `x = sin( x )` or `x = sin (x)`,
- a space after a comma: `range(5, 10)` and not `range(5,10)`,
- no spaces at the end of a line or in an empty line,
- within a function we can occasionally add a *single* empty line,

- between functions we place two empty lines,
- we import (`import`) each module on a separate line,
- first we import the standard Python libraries, then other libraries (e.g., `numpy`), and finally our own.

Note: modern development tools help us format code that conforms to PEP8 (in VSC press `ctrl+shift+P`, type “*Format document*”, and press `enter`)!

PEP20

PEP20 is the next important PEP, also called *The zen of Python*; these are guidelines that we try to follow when programming. Here we list only a few of the guidelines:

- Beautiful is better than ugly.
- Simple is better than complicated.
- Flat is better than nested.
- Readability counts.
- Special cases aren’t special enough to break the rules.
- If the implementation is hard to explain, it’s probably a bad idea; if it’s easy to explain, it may be a good one.

1.8 Python basics

1.8.1 Basic data types and operators

Dynamic typing

Python is a dynamically typed language. Variable types are not checked. As a consequence, some operations are not possible on certain types, or are adapted to the data type (later, for example, we will look at multiplying a string of characters). The type is determined upon assignment of a value; let us look at an example:

```
| a = 1 # this is a comment: a is an integer
| a
|
| 1
```

We display the type with the command `type`:

```
| type(a)
|
| int
```

We can overwrite the value of `a` with a new value:

```
| a = 1.0 # this is now a floating -point number (float)
| a      # if we write `a` like this on the last line, the value will be printed
|
| 1.0
|
| type(a)
|
| float
```

To print a value we can also use the function `print()`, which we will get to know in more detail later:

```
| print(a)
```

```
| 1.0
```

We can call the documentation of a function by pressing [shift + tab]. E.g.: we write `print` and press [shift + tab], and the help will appear in a pop-up window; if we hold [shift], an additional press of [tab] shows extended help, and a further double press of [tab] (so four times in total) shows the help in a new window.

We can also access help by placing `?` before the function (for help) or `??` (which also shows the source code, if available).

Example:

```
| ?print  
| ??print
```

Logical operators

The logical operators are ([source](#)):

- `or`, example: `a or b`; note: the second element is checked only if the first is not true,
- `and`, example: `a and b`; note: the second element is checked only if the first is true,
- `not`, example: `not a`; note: the non-logical operators have precedence: `not a == b` is interpreted as `not (a == b)`.

Comparison operators

The comparison operators ([source](#)):

- `<` less than,
- `<=` less than or equal,
- `>` greater than,
- `>=` greater than or equal,
- `==` equal,
- `!=` not equal,
- `is` identity of the object of the compared operands,
- `is not` whether the operands are not the same object.

Example:

```
| 5 < 6
```

```
| True
```

We can make a chain of logical operators:

```
| 1 < 3 < 6 < 7
```

```
| True
```

let us check identity (whether it is the same object):

```
| a = 1  
| a == 1
```

```
| True
```

Let us also use a logical operator:

```
| 1 < 3 and 6 < 7
```

```
| True
```

```
| 1 < 3 and not 6 < 7
|
| False
```

Here it is worth pointing out that in logical expressions, in addition to `False`, the value `0`, `None`, an empty list (or string, tuple, ...) are also falsy; **everything else is True**.

Let us look at a few examples:

```
| 1 and True
|
| True
|
| 0 and True
|
| 0
|
| (1,) and True
|
| True
```

Data types for numeric values

We will look at the most common data types that we will use later ([source](#)):

- `int` for representing an arbitrarily large integer,
- `float` for representing floating-point numbers,
- `complex` for representing numbers in complex form.

We will get to know the data types in more detail (representation accuracy and the like) in the introduction to numerical methods.

Examples:

```
| whole_number = -2**3000          # `**` represents the power operator
| real_number = 3.141592
| complex_number = 1 + 3j          # `3j` represents the imaginary part
```

Operators on numeric values

Data types sorted by increasing priority ([source](#)):

1. `x + y` sum,
2. `x - y` difference,
3. `x * y` product,
4. `x / y` division,
5. `x // y` integer division (the result is an integer rounded down),
6. `x % y` remainder of integer division,
7. `-x` negation,
8. `+x` unchanged,
9. `abs(x)` absolute value,
10. `int(x)` integer of type `int`,
11. `float(x)` real number of type `float`,
12. `complex(re, im)` complex number of type `complex`,
13. `c.conjugate()` complex conjugate number,
14. `divmod(x, y)` returns the pair `(x // y, x % y)`,
15. `pow(x, y)` returns `x` to the power `y`,
16. `x ** y` returns `x` to the power `y`.

For arithmetic operators it is worth pointing out the variant in which the result is assigned to the operand ([source](#)). This is called *augmented assignment*.

Let us look at an example. Instead of:

```
a = 1
a = a + 1 # we add the value 1
a
```

```
2
```

we can write:

```
a = 1
a += 1
a
```

```
2
```

Bit-level operators

We will rarely use bit-level operators in engineering practice; nevertheless, let us list them by increasing priority ([source](#)):

- `x | y` bitwise *or* of `x` and `y`,
- `x ^ y` *exclusive or* (only one, not both) at the bit level of `x` and `y`,
- `x & y` bitwise *and* of `x` and `y`,
- `x <<n` shift the bits of `x` by `n` bits to the left,
- `x >>n` shift the bits of `x` by `n` bits to the right,
- `~x` negate the bits of `x`.

Let us look at an example ([source](#)):

```
a = 21 # 21 in bit form: 0001 0101
b = 9  # 9 in bit form: 0000 1001

c = a & b # 0000 0001 in bit form represents the number 1 in decimal
c
```

```
1
```

Let us display the number in bit form:

```
bin(c)

'0b1'
```

1.8.2 Composite data structures

String

A string is composed of characters according to the *Unicode* standard. We will not go into the details; it is important to note that, starting from version 3, Python made great progress ([source](#)), and that we can use arbitrary characters from the *Unicode* standard both in text and in program files .py.

A string starts and ends with a double `"` or single `'` quotation mark.

```
b = 'text' # b is a string
b

'text'
```

Here there are no problems with names that we are more used to:

```
\pi = 3.14
```

The character π is written in the notebook by typing `\pi` (the names are usually the same as those used in LaTeX) and then pressing [tab].

Some operations on strings:

```
print('short ' + b) # addition
print(3 * 'This is string multiplication. ')      # string multiplication
print('Integer:', int('5') + int('5'))           # conversion of the string '5' to an
integer
print('Floating -point number:', float('5') + float('5')) # conversion of the string
'5' to a floating -point number

short text
This is string multiplication. This is string multiplication. This is string
multiplication.
Integer: 10
Floating -point number: 10.0
```

Tuple

Tuples ([source](#)) are lists of arbitrary **objects** that we separate with a comma and write inside round parentheses:

```
tuple_1 = (1, 'programming', 5.0)

tuple_1

(1, 'programming', 5.0)
```

We have already used the word *object* several times. What is an object? We will look at objects and classes in more detail later; for now we are satisfied with the fact that an object is more than just a *name* for a certain value; an object also has methods. E.g., the second element is of type `str`, which also has the method `replace` for replacing characters:

```
tuple_1[1].replace('r', '8')

'p8og8amming'
```

If a tuple has only one object, then we have to end it with a comma (otherwise it is not distinguished from the object in parentheses):

```
tuple_2 = (1,)
type(tuple_2)

tuple
```

Tuples can also be written without round parentheses:

```
tuple_3 = 1, 'without parentheses', 5.

tuple_3

(1, 'without parentheses', 5.0)
```

We access objects in a tuple via the index (**they start at 0**):

```
tuple_1[0]

1
```

A tuple **cannot be changed** (it is *immutable*); we cannot remove or add elements. Calling the expression below would lead to an error:

```
tuple_1[0] = 3
>>> TypeError: 'tuple' object does not support item assignment
```

List

Lists ([source](#)) are written similarly to tuples, but square brackets are used:

```
| my_list = [1., 2, 'd']  
|  
| my_list  
|  
| [1.0, 2, 'd']
```

Unlike tuples, lists can be changed:

```
| my_list[0] = 10  
| my_list  
|  
| [10, 2, 'd']
```

Lists are *mutable*, and one has to be aware that the **name** in fact **points** to a location in the computer's memory:

```
| b = my_list  
| b  
|  
| [10, 2, 'd']
```

Now **b** also points to the same location in memory.

Let us see what happens to **my_list** if we change an element of the list **b**:

```
| b[0] = 9  
| b  
|  
| [9, 2, 'd']  
|  
| my_list  
|  
| [9, 2, 'd']
```

If we want to make a copy of the data, then we have to do it like this:

```
| b = my_list.copy()  
| b[0] = 99  
| b  
|  
| [99, 2, 'd']  
|  
| my_list  
|  
| [9, 2, 'd']
```

Example of a list of lists:

```
| matrix = [[1, 2, 3],  
|           [4, 5, 6],  
|           [7, 8, 9]]  
|  
| matrix  
|  
| [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

We will learn more about matrices later.

Selected operations on lists

Here we will look at some of the most common operations; for more, see the [documentation](#).

First let us look at adding an element:

```
| my_list = [0, 1, 'test']
| my_list.append('adding an element')
| my_list
|
| [0, 1, 'test', 'adding an element']
```

Then inserting at a specific position (at index 1; the other elements are shifted):

```
| my_list.insert(1,'to the second position')
| my_list
|
| [0, 'to the second position', 1, 'test', 'adding an element']
```

Then we can remove a specific element. We do this with `my_list.pop(i)` or `del my_list[2]`. The first way returns the removed element, the second only removes the element:

```
| removed_element = my_list.pop(2) # remove the element with index 2 and return it
| ## the command: del my_list[2] would have a similar effect
| print('list:', my_list)
| print('removed element:', removed_element)
|
| list: [0, 'to the second position', 'test', 'adding an element']
| removed element: 1
```

With the method `.index(v)` we find the first element with value `v`:

```
| my_list
|
| [0, 'to the second position', 'test', 'adding an element']
|
| my_list.index('test')
|
| 2
```

Why use a tuple and not a list?

Tuples cannot be changed, so their numerical implementation is substantially lighter (in the numerical sense) and memory can be used better. Consequently, tuples are **faster** than lists; in addition, we cannot **change** them by accident. For a broader answer, see this [source](#).

Sets

Sets are in principle similar to the *sets* we know from mathematics. Sets in Python ([documentation](#)), unlike the other composite structures:

- **do not allow duplicate elements** and
- **do not have an ordered sequence** (important, e.g., for correct use in loops).

It is worth emphasizing that adding an element and determining membership of an element in a set are very fast operations.

We create them with curly braces `{}` or with the command `set()`:

```
| A = {'a', 4, 'a', 2, 'z', 3, 3}
| A
|
| {2, 3, 4, 'a', 'z'}
```

If we want to get rid of duplicate elements in a list, a very simple way is to create a set from the list:

```
| B = set([6, 4, 6, 6])
| B
| {4, 6}
```

We can use the typical mathematical operations on sets:

```
| B - A    # elements in B without elements that are also in A
| {6}
| A | B    # elements in A or B
| {2, 3, 4, 6, 'a', 'z'}
| A & B    # elements in A and B
| {4}
| A ^ B    # elements in A or B, but not in both
| {2, 3, 6, 'a', 'z'}
```

Dictionary

A dictionary ([documentation](#)) can be defined explicitly with *key : value* pairs, separated by commas inside curly braces:

```
| parameters = {'height': 5.2, 'width': 43, 'g': 9.81}
| parameters
| {'height': 5.2, 'width': 43, 'g': 9.81}
```

We access the value of an element by using the key:

```
| parameters['height']
| 5.2
```

More often we use the methods:

- `values()`, which returns the values of the dictionary,
- `keys()`, which returns the keys of the dictionary,
- `items()`, which returns a list of tuples (key, value); we will often need this in loops.

Example:

```
| parameters.items()
| dict_items([('height', 5.2), ('width', 43), ('g', 9.81)])
```

Dictionaries can be changed; e.g., we add an element:

```
| parameters['centroid_position'] = 99999
| parameters
| {'height': 5.2, 'width': 43, 'g': 9.81, 'centroid_position': 99999}
```

We remove elements (similar to above, `pop` removes and returns the value, `del` only removes):

```
| parameters.pop('centroid_position')
| 99999
| parameters
| {'height': 5.2, 'width': 43, 'g': 9.81}
```

Operations on composite data structures

Most composite data structures allow the following operations ([documentation](#)):

- `x in s` returns `True` if any element of `s` is equal to `x`, otherwise returns `False`,
- `x not in s` returns `False` if any element of `s` is equal to `x`, otherwise returns `True`,
- `s + t` concatenates `s` and `t`,
- `s * n` or `n * s` results in `s` repeated `n` times,
- `s[i]` returns the `i`-th element of `s`,
- `s[i:j]` returns `s` from element `i` (inclusive) to `j` (not included),
- `s[i:j:k]` slicing `s` from element `i` to `j` with step `k`,
- `len(s)` returns the length of `s`,
- `min(s)` returns the smallest element of `s`,
- `max(s)` returns the largest element of `s`,
- `s.index(x[, i[, j]])` returns the index of the first occurrence of `x` in `s` (from index `i` to `j`),
- `s.count(x)` returns the number of elements.

Some examples:

```
| 'a' in 'abc' # in compares whether an object is in a string, list, ...
| True
```

Above we had the dictionary `parameters`:

```
| {'g': 9.81, 'height': 5.2, 'width': 43}
```

Let us check whether it contains an element with the key `width`:

```
| 'width' in parameters
| True
```

Counting and slicing composite data structures

Assume the string below:

```
| alphabet = 'abcdefghijklmnoprstuvz'
```

In Python we start counting indices at 0! Online you can find discussions about whether it is correct for indices to start at 0 or 1. Why start at 0 is explained very nicely by Prof. Dr. Janez Demšar in the book *Python za programerje* ([source](#), page 28), or also by Prof. Dr. Edsger W. Dijkstra in a note on the [web](#).

We call slicing the operation in which we cut certain members out of a list.

The general rule is:

```
| s[i:j:k]
```

which means that the list `s` is sliced from element `i` to `j` with step `k`. If we do not provide one of the elements, then the (default) value is used:

- `i` defaults to 0, which means the first element,
- `j` defaults to `len(s)` (the sequence length), which means the last element is also included,
- `k` defaults to 1.

The first character is therefore:

```
| alphabet[0]
```

```
| 'a'
|
| The first three characters are:
|
| alphabet[:3]
|
| 'abc'
|
| Of the first 15 characters, every third one is:
|
| alphabet[:15:3]
|
| 'adgjm'
|
| The last fifteen characters, every third one:
|
| alphabet[-15::3]
|
| 'hknru'
```

When we want to reverse the order, we can do it like this (we read from the beginning to the end with step -1, which means from the end toward the beginning with step 1:):

```
| alphabet[::-1]
|
| 'zvutsrponmlkjihgfedcba'
```

1.8.3 Program flow control

We will look at some basic tools for controlling the flow of program execution ([documentation](#)).

The if statement

A typical form of the if statement ([documentation](#)):

```
| if condition0:
|     print('condition0 was satisfied')
| elif condition1:
|     print('condition1 was satisfied')
| elif condition2:
|     print('condition2 was satisfied')
| else:
|     print('No condition was satisfied')
```

Example:

```
| a = 6
| if a > 5:
|     print('a is more than 5') # executes if the condition is true.
|     print(13*' -')
|
| a is more than 5
| - - - - -
```

Above we indented the function `print`, since it belongs to the block of code that executes in the case `a > 0`. Some programming languages put the code inside a block in curly or other brackets and additionally indent it. In Python it is only indented. The indentation is typically **4 spaces**. It can also be a **tab** (less common), but within a single .py file we may use only one indentation method.

We can also write statements on a single line, but such use is discouraged, since it reduces the readability of the code:

```
| if a > 5: print('this form is discouraged'); print(40*' -'); print('using a semicolon
| instead of a new line')
|
| this form is discouraged
| - - - - -
| using a semicolon instead of a new line
```

The if expression

In addition to the if statement, Python also has an if expression (the *ternary operator*, see the [documentation](#)).

The syntax is:

```
[execute if True] if condition else [execute if False]
```

A simple example:

```
'It is true' if 5==3 else 'It is not true'
```

```
'It is not true'
```

If expressions are very useful, and we will often use them in list comprehensions!

The while loop

The syntax of the while loop ([documentation](#)) is:

```
while condition:
    [code to execute]
else:
    [code to execute]
```

Here it should be emphasized that the `else` part is optional and executes upon exiting the `while` loop. The `break` command interrupts the loop and does not execute the `else` part. The `continue` command interrupts the execution of the current code and goes immediately to testing the condition.

A simple example:

```
a = 0
while a < 3:
    print(a)
    a += 1
else:
    print(' - -')
```

```
0
1
2
- -
```

The for loop

In Python we will use the for loop **very often** ([documentation](#)). The syntax is:

```
for element in list_with_elements:
    [code to execute]
else:
    [code to execute]
```

Similar to `while`, here too the `else` part is optional.

Let us look at an example:

```
names = ['Jaka', 'Miki', 'Luka', 'Anja']
for name in names: # an assignment is hidden here; ``name`` gets a new value each
time
    print(name)
```

```
Jaka  
Miki  
Luka  
Anja
```

It is very useful to loop over dictionaries:

```
print('The dictionary is:', parameters)  
for key, val in parameters.items():  
    print(key, '=', val)
```

```
The dictionary is: {'height': 5.2, 'width': 43, 'g': 9.81}  
height = 5.2  
width = 43  
g = 9.81
```

List comprehensions

In programming we often encounter the need to perform operations on each individual element of a list. We have already learned the `for` loop, which we can use in such a case.

Let us look at an example where we compute the square of numbers:

```
my_list = [1, 2, 3] # source list  
result = [] # prepare an empty list into which we will add the results  
for element in my_list:  
    result.append(element**2)  
result  
  
[1, 4, 9]
```

We can achieve the same result much more simply with so-called **list comprehensions** (see the [documentation](#)).

The above example would be:

```
my_list = [1, 2, 3] # source list  
result = [element**2 for element in my_list] # <<< on the right -hand side of = is a  
list comprehension  
result  
  
[1, 4, 9]
```

A list comprehension is represented by the code `[element**2 for element in my_list]`, which in fact says exactly what is written: compute the square for each element in the list. The square brackets indicate that the result is a list.

A list comprehension has the following syntax:

```
[code for element in list]
```

and it has primarily two advantages:

- **clarity** / **compactness** of the code and
- slightly **faster** execution.

We have already been able to judge the clarity/compactness. How do we check the speed? The easiest way to do this is with the so-called magic command `%%timeit`, that is, **measure the time** (see the [documentation](#)); below we will use the block magic command (denoted by `%`):

```
%%timeit -n1000
```

where the parameter `-n1000` limits the time measurement to 1000 repetitions.

First the longer way:

```
| my_list = range(1000)

| %%timeit -n1000
| result = []
| for element in my_list:
|     result.append(element**2)

| 54.6 \mus \pm 4.42 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

Then the list comprehension (clearer and slightly faster):

```
| %%timeit -n1000
| result = [element**2 for element in my_list]

| 48.2 \mus \pm 1.18 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

Let us look at two more useful examples:

- using the `if` expression to compute a new value,
- using the `if` expression to decide whether a new value is computed at all.

When computing a new value, we insert the `if` expression in the part before `for`:

```
| my_list = [1, 2, 3]
| [el+10 if el<2 else el**2 for el in my_list]

| [11, 4, 9]
```

Often, however, we do not want to perform the computation at all for certain elements; in this case we insert the `if` expression after the list:

```
| [el**2 for el in my_list if el >1]

| [4, 9]
```

The `zip` function

When we want to combine several lists in a loop, we use the `zip` function ([documentation](#)); example:

```
| x = [1, 2, 3]
| y = [10, 20, 30]

| for xi, yi in zip(x, y):
|     print('The pairs are: ', xi, 'and', yi)

| The pairs are:  1 and 10
| The pairs are:  2 and 20
| The pairs are:  3 and 30
```

The `range` function

We will often use the `for` loop together with the `range` function ([documentation](#)). The syntax is:

```
| range(stop)
| range(start, stop[, step])
```

So if we call the function with one parameter, it is **stop**: **up to** which number we count. In the case of a call with two parameters **start**, **stop**, we count from, to! The third parameter is optional and defines the step **step**.

Let us look at an example:

```
for i in range(3):  
    print(i)
```

```
0  
1  
2
```

The enumerate function

We also often use the `for` loop with the `enumerate` function, which numbers the elements ([documentation](#)).

Let us look at an example:

```
for i, name in enumerate(names):  
    print(i, name)
```

```
0 Jaka  
1 Miki  
2 Luka  
3 Anja
```

1.9 To conclude: a cheat sheet :)

There are many of them; here is a [link](#) to one.

Some more reading: [automatetheboringstuff.com](#).

And one tweet:

```
%%html  
<blockquote class="twitter -tweet" data -lang="en"><p lang="en" dir="ltr">The most  
important slide from my <a href="https://twitter.com/PyData">@PyData</a> talk. <a  
href="https://t.co/Hc8j4ZpHV4">pic.twitter.com/Hc8j4ZpHV4</a></p>&mdash; Almar Klein  
(@almarklein) <a  
href="https://twitter.com/almarklein/status/708909917684568064">March 13,  
2016</a></blockquote>  
<script async src="//platform.twitter.com/widgets.js" charset="utf -8"></script>
```

The most important slide from my @PyData talk. [pic.twitter.com/Hc8j4ZpHV4](https://t.co/Hc8j4ZpHV4)
— Almar Klein (@almarklein) [March 13, 2016](#)

Chapter 2

Functions, exceptions, modules

2.1 Functions

2.1.1 The print function

`print` is one of the most frequently used functions. It prints all the arguments passed to it, puts a space between them, and at the end moves to a new line.

Let's look at an example:

```
| print('first argument', 'second', '=', 5.)  
  
| first argument second = 5.0
```

For more general use, see the [documentation](#):

```
| print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False),
```

where the arguments are:

- `*objects` represents the arguments we want to print (the full meaning of the asterisk will become clear when we discuss *functions*, below),
- `sep` is the separator string,
- `end` is the terminating string,
- `file` is the output *file* (`sys.stdout` is the *standard output*, which in this particular case means the *screen*, i.e. the command line; we will change this later),
- `flush`, when `True`, finalizes the output (otherwise, due to optimization, it may remain in the buffer).

For the `end` argument above we used a so-called **escape character**: `\`.

The purpose of the escape character is to give the following character a special meaning; some common uses are:

- `\n` inserts a *line break*,
- `\t` inserts a *tab*,
- `\b` deletes the previous character (*backspace*),
- `\'` prints a single quote `'`,
- `\"` prints a double quote `"`,
- `\\` displays the *backslash*,
- `\` at the end of a line: the string continues on the next line (without a newline character).

Example (how we usually do *not* program):

```
| print('fi','ve', sep=' -', end='=')  
| print('5:')  
  
| fi -ve=5:)
```

And one more example with escape characters:

```
print('Strings in Python (per PEP8) should not be longer than 79 characters. \
The reason is that we do not have to scroll the window left and right. \
So that we can still write strings that are longer than 79 characters, \
we use a single separator character \\ (note how it \
is printed: \\)',)
```

```
Strings in Python (per PEP8) should not be longer than 79 characters. The reason is
that we do not have to scroll the window left and right. So that we can still write
strings that are longer than 79 characters, we use a single separator character \
(note how it is printed: \)
```

2.1.2 String formatting

For formatting strings we have several tools available:

- **f-string** (*f* for “formatted”), [documentation](#),
- **the .format() method**, [documentation](#),
- **% formatting**, [documentation](#).

Let’s look at some examples:

```
a = 3.14
print(f'pi = {a}')           # f -string
print('pi = {a}'.format(a = a)) # .format
print('pi = %(a)3.2f' % {'a': a}) # %

pi = 3.14
pi = 3.14
pi = 3.14
```

The *f-string* has been implemented since Python version 3.6 ([PEP 498](#)) and is the simplest way of formatting. We mention the other approaches only so that the reader becomes acquainted with them and so that we can link to the documentation.

An *f-string* always begins with **f** before the first quote and allows very general formatting (see the [documentation](#)).

The value we want to format inside the string is placed in curly braces:

```
f'text... {arg_name1:fmt1} ... text...'
```

Inside the curly braces:

- before **:** we give the name of the value we want to format
- after **:** we format how the variable is printed; most often we will use the *fmt* forms:
 - **w.df** - floating-point representation (**f**loat)
 - **w.de** - representation with an exponent (**e**xponent)
 - **w.dg** - general format (**g**eneral)
 - **ws** - string of characters (**s**tring).

w is the (minimum) total width, and **d** is the number of digits after the decimal point.

Let’s look at an example:

```
height = 1.84
name = 'Marko'
text = f'{name} is {height} m tall, or also: {height:e} m, {height:7.3f} m.'
text

'Marko is 1.84 m tall, or also: 1.840000e+00 m,   1.840 m.'
```

We could do the same with the *format* method by referring to the argument index:

```
text = '{1} is {0} m tall, or also: {0:e} m, {0:7.3f} m.'.format(height, name)
text
```

```
'Marko is 1.84 m tall, or also: 1.840000e+00 m, 1.840 m.'
```

Example using a dictionary:

```
parameters = {'height': 5., 'density':100.111, 'user name': 'Janko'}
f'height = {parameters["height"]:7.3f}, density = {parameters["density"]:7.3e}'
'height = 5.000, density = 1.001e+02'
```

Formatting with *fmt* (the *Format Specification Mini-Language*) is extremely rich. The part after the colon has the general form (see the [documentation](#)):

```
[[fill]align][sign][#][0][width][grouping_option][.precision][type]
```

where the parameters in square brackets are optional (all of them are optional):

- **fill** can be any character,
- **align** specifies the alignment, options: "<" | ">" | "=" | "",
- **sign** specifies the sign, options: "+" | "-" | " ",
- **width** defines the minimum total width,
- **grouping_option** grouping options, options: "_" | ",",
- **precision** defines the number of digits after the decimal separator,
- **type** defines how the data should be displayed, options: "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"

We align text to the left with "<", to the right with ">", and centered with ". Example of centered/right alignment (width 20 characters):

```
f'{height:*~20}' # before ^ is the character we use to fill the left and right sides
```

```
'*****1.84*****'
```

```
f'{parameters["height"]:_>20}'
```

```
'_____5.0'
```

```
c = 4 -2j
```

```
f'The complex number {c} consists of a real ({c.real}) \
and an imaginary ({c.imag}) part.'
```

```
'The complex number (4 -2j) consists of a real (4.0) and an imaginary ( -2.0) part.'
```

You can look at formatting of date and time in the appendix below.

2.1.3 File handling

File handling is important, since we often store data in files or read it from them. A file we want to read from or write to must first be opened. We do this with the **open** command ([documentation](#)):

```
open(file, mode='r', buffering= -1, encoding=None,
      errors=None, newline=None, closefd=True, opener=None)
```

Of all the arguments, only **file** is required; it defines the path to the file.

The **mode** argument can be:

- 'r' for reading (*read*, default),
- 'w' for writing (*write*, the file is cleared first),

- 'x' for exclusive writing; if the file already exists it returns an error,
- 'a' for appending to the end of an existing file (*append*),
- 'b' binary format,
- 't' text format (default),
- '+' the file is opened for both reading and writing.

The last three ('b', 't', '+') are used in combination with the first three ('r', 'w', 'x').

Example: `mode='r+b'` means reading and writing a binary file.

We will look in more detail only at reading and writing text files.

The other (optional) arguments of the `open` function are described in detail in the [documentation](#).

The default is `mode='r'` (i.e. 'rt', since text mode is the default), which means the file is opened for reading in text form.

Let's first look at writing to a (text) file (`mode='w'`):

```
| file = open('data/prikaz vpisa.txt', mode='w')
```

'data/prikaz vpisa.txt' is a relative path (relative to the location where the *Jupyter notebook* is run) to the file.

Let's write the first line using the `write` method:

```
| file.write('test of the first line\n')
```

```
| 23
```

The `write` method returns the number of characters written. If you check the contents of the file at this moment, it is very likely still empty. The reason is that the content is still in the buffer, which is flushed to disk only occasionally or when the file is closed (the reason for this behavior is the speed of writing to disk; for details see the documentation of the `buffering` argument of the `open` function).

Let's write the second line:

```
| file.write('second line\n', )
```

```
| 12
```

Now let's also enter some numerical values:

```
| for d in range(5):
|     file.write(f'{d:7.2e}\t {d:9.4e}\n')
```

We close the file:

```
| file.close()
```

Now we open the file for reading and writing, `mode='r+'`:

```
| file = open('data/prikaz vpisa.txt', mode='r+')
```

Let's check all the lines with a `for` loop; in the code below, `file` returns one line on each iteration:

```
| for line in file:
|     print(line, end='')
```

```
| test of the first line
| second line
| 0.00e+00          0.0000e+00
| 1.00e+00          1.0000e+00
| 2.00e+00          2.0000e+00
| 3.00e+00          3.0000e+00
| 4.00e+00          4.0000e+00
```

In the introduction to the `print` function we mentioned the `file` argument, which defines the file to write to (the default is the standard output, i.e. the *screen*). If we pass as the `file` argument a file that is opened for writing:

```
print('end', file=file)
file.close()
```

the content is written to the file. You can check the result in the file!

This is a nice opportunity to learn about the `with` statement ([documentation](#)). The full scope of the `with` statement goes beyond the purpose of this course, but its use in file handling is very simple, so let's look at an example:

```
with open('data/prikaz vpisa.txt') as file:
    for line in file:
        print(line, end='')
```

```
test of the first line
second line
0.00e+00      0.0000e+00
1.00e+00      1.0000e+00
2.00e+00      2.0000e+00
3.00e+00      3.0000e+00
4.00e+00      4.0000e+00
end
```

The `with` statement stores the result of the `open` function in (*as*) the name `file`, and then we print the values for each line. When exiting the `with` statement, the `file.close()` method is called automatically. The `with` statement therefore allows us to write clear and compact code.

2.1.4 Functions in general

Python has many functions already built in (see the [documentation](#)); but we can also write functions ourselves or import them from so-called modules (above we imported `datetime`, more on this later).

A generic example of a function is ([documentation](#)):

```
def function_name(parameters):
    '''docstring'''
    [code]
    return content
```

Here we point out:

- `def` denotes the definition of a function,
- **function_name per PEP8, function names are written in lowercase, with multiple words joined by an underscore,**
- `parameters` the parameters of the function, more on which below,
- `docstring` (optional) the documentation of the function,
- `[code]` the code of the function,
- `return` (optional) the command to exit the function; whatever follows is returned as the result (if there is no `return`, or nothing after `return`, then `None` is returned)

Example of a simple function:

```
def area(length, width):
    S = length * width
    return S
```

```
area(1, 20)
```

```
20
```

2.1.5 Passing arguments to functions

Python knows two ways of passing arguments:

- by **position** (*positional*)
- by **name**.

Let's look at both ways with an example:

```
| area(3, 6) # positional
|
| 18
|
| area(width = 3, length = 6) # by name
|
| 18
```

It is recommended to pass arguments by *name*; this reduces the chance of error!

If we do not give a name, then the name is assumed based on the order of the arguments; example:

```
| area(3, width=4)
|
| 12
```

When we mix *positional* and *named* argument passing, the **positional arguments must come first, and only then the named ones**. At the same time, a named argument must not redefine a positional one; since **length** is the first argument, calling code like this:

```
| area(3, length=4)
```

would cause an error.

Example of good practice when defining arguments

We often extend functions and add new arguments, or sometimes we cannot even define the number of arguments to a function in advance!

Python has no problem with this. Arguments that are not explicitly defined are handled with:

- `*[name0]` the tuple `[name0]` contains all **positional arguments** that are not explicitly defined,
- `**[name1]` the dictionary `[name1]` contains all **named arguments** that are not explicitly defined.

Let's look at an example:

```
| def some_function(student, exam = True):
|     ''' First version of the function'''
|     print(f'Student {student} has registered for the exam: {exam}.')
|
| some_function('Janez', exam=True)
|
| Student Janez has registered for the exam: True.
```

Now let's prepare a second, upgraded version (it allows the same use as the first version):

```
| def some_function(*args_tuple, **kwargs_dict):
|     ''' Second version of the function '''
|     student = args_tuple[0]
|     exam = kwargs_dict.get('exam', True) # True is the default value
|     print_figure = kwargs_dict.get('print_figure', False) # False is the default
|     value
|
|     print(f'Student {student} has registered for the exam: {exam}.')
|     if print_figure:
|         print('figure:')
```

```
| some_function('Janez', exam=True)
| Student Janez has registered for the exam: True.
| some_function('Janez', exam=True, print_figure=True, nonexistent_argument='no')
| Student Janez has registered for the exam: True.
| figure:)
```

Passing functions as an argument

Here we would like to point out that arguments can also be other functions.

Let's look at an example:

```
| def format_a(value):
|     return f'Display of value: {value}'
|
| def format_b(value):
|     return f'Display of value: {value:3.2f}'
|
| def display(formatter, value):
|     print(formatter(value))
```

So we have two functions, `format_a` and `format_b`, which format a numerical value slightly differently.

Let's look at their use:

```
| display(format_a, 3)
| Display of value: 3
| display(format_b, 3)
| Display of value: 3.00
```

2.1.6 Function docstring

The *docstring* is an optional part of a function definition that documents the function and its use. From this point of view, it is highly recommended to use it.

The [documentation](#) for the docstring lists the essential elements:

- the first line should be a short summary of the function,
- if there are multiple lines, the second should be blank (to visually separate the first line from the rest of the text),
- use three quotes (double `"""` or single `'`), which denote a multiline string of characters.

Example of a documented function:

```
| def area(length = 1, width = 10):
|     """ Computation of the area of a rectangle
|
|     The function returns the value of the area.
|
|     Arguments:
|     length: length of the rectangle
|     width: width of the rectangle
|     """
|     return length * width
```

Example of a call with default arguments (while calling the function you can press `[shift]+[tab]` to access the help):

```
| area()
| 10
```

2.1.7 Local/global names

A simple rule of thumb for functions is: **a function sees out, but others do not see in**. A function has internal names that do not collide with the same names in other functions.

Let's look at an example:

```
outer = 3
def first():
    inner = 5
    print(f'This is the outer value: {outer}, and this is the inner one: {inner}')

first()

This is the outer value: 3, and this is the inner one: 5
```

Since the name `inner` is not defined outside the function `first`, calling:

```
inner
```

outside the function would return an error (we cannot see into the function).

Here it is worth mentioning that **passing external names into a function outside of the function's arguments is discouraged**.

2.1.8 return

The `return` command returns values from a function. So far we have seen results in the form of a single variable; but this variable can also be a tuple.

Let's look at an example:

```
def return_tuple():
    return 1, 2, 3 # this is an equivalent way of writing the tuple: (1, 2, 3)

return_tuple()

(1, 2, 3)
```

We can capture the result of the function like this:

```
result = return_tuple()
result

(1, 2, 3)
```

Or by unpacking the values (this is generally true for tuples):

```
i1, i2, i3 = return_tuple()
i1 # printing only one value

1
```

2.1.9 Anonymous function/expression

An anonymous function (see the [documentation](#)) is a function to which we do not give a name (which is why it is *anonymous*), but instead define it with the `lambda` command. A typical use is:

```
lambda [parameter list]: expression
```

We will look at the `lambda` function with a sorting example:

```
| lst = [ -4, 3, -8, 6]
```

from the smallest to the largest value. We can do this with the built-in `sort` method ([documentation](#)):

```
| sort(*, key=None, reverse=False)
```

The method has two optional arguments:

- **key** the key by which we sort the elements
- **reverse**, if `False`, the comparison between key values uses `<`, otherwise `>`.

Note: `list.sort()` performs the sorting on the existing list and does not return a list. The `sorted()` function (see the [documentation](#)), however, returns a list of the sorted values.

Let's look at an example:

```
| lst = [ -4, 3, -8, 6]
| lst.sort()
| lst
```

```
| [ -8, -4, 3, 6]
```

Now let's use a key where the squared value is used for comparing the sorting values:

```
| lst.sort(key = lambda a: a**2)
| lst
```

```
| [3, -4, 6, -8]
```

2.2 Handling exceptions

While programming, exceptions occur relatively often; for example: division by zero.

```
| 1/0
|
| -----
| -----
| ZeroDivisionError                                Traceback (most recent call last)
| <ipython -input -50 -05c9758a9c21> in <module>()
| - - - -> 1 1/0
|
| ZeroDivisionError: division by zero
```

In case of an exception, Python warns about it and interrupts execution. It is not good for a program to be interrupted or to crash; for this reason a beginner programmer would probably handle the above situation with an `if` statement. However, such an approach is neither very practical nor general, and that is why exception handling has become established in most modern programming languages.

We will look at some basics that will above all make it easier for students to read and understand the code of other authors.

We handle exceptions with the `try` statement:

```
| try:
|     [code0]
| except:
|     [code1]
| finally:
|     [code2]
```

In the `try` statement, `except` is executed if an exception occurs in the code `[code0]`; `finally` is optional and is executed in any case when exiting the `try` statement.

Let's look at an example where we simply continue execution:

```
try:
    1/0
except:
    pass
```

We use the `pass` command when the syntax requires a statement (code) to execute, but we do not want any action (see the [documentation](#)).

With `except` we can also catch different exceptions; we do it like this:

```
try:
    1/0
except ZeroDivisionError:
    print('Division by zero!') # or some other action
except:
    print('Unexpected error.') # discouraged; it is good to anticipate the error!
```

```
Division by zero!
```

Now let's use the same code for the case of dividing a number by a string:

```
try:
    1/'this just will not work'
except ZeroDivisionError:
    print('Division by zero!') # or some other action
except:
    print('Unexpected error.') # discouraged; it is good to anticipate the error!
```

```
Unexpected error.
```

2.2.1 Raising exceptions

We can also raise exceptions ourselves; we do this with the `raise` command ([documentation](#)).

An example of raising an error is:

```
raise Exception('Description of the exception')
```

a simple but complete example, where we would expect `person_name` to be of type `str`:

```
person_name = 1
try:
    if type(person_name) != str:
        raise Exception(f'The name is not of the expected type `str`.')
except Exception as exception:
    print(exception)
```

```
The name is not of the expected type `str`.
```

2.3 Basics of modules

With modules we can easily reuse already-written code. Simply put, modules in Python allow us to relatively easily maintain clarity and order. Usually we package some self-contained function, a group of functions, or, for example, an object into a module (later we will learn what an object is).

For a detailed description of importing modules, see the [documentation](#).

In the `modules` folder there is a file `first_module.py` with the following content:

```
def square(x=1):
    return x**2
```

Important: we often arrange modules hierarchically into directories; a directory of modules is called a *package*. If the directory contains a file (it can even be empty) named `__init__.py`, it is a regular package; since Python 3.3, however, a directory behaves as a package even without it (a so-called *namespace package*). In the example above, the `modules` folder thus behaves as a *package*, and in it there is the *module* `first_module` (the file `first_module.py`).

We typically import modules in two ways:

First way:

```
from modules import first_module
```

in this case we import the namespace `first_module`; this means that we call the function `square()` like this: `first_module.square()`.

Second way of importing a module:

```
import modules.first_module
```

We import the *namespace* `modules.first_module`; this means that we call the function `square()` like this: `modules.first_module.square()`. (The command `import modules` alone does **not** import submodules automatically.)

If we want to import the functions in the `first_module` module, we do it like this:

```
from modules import first_module
```

Now we can call the `square` function like this:

```
first_module.square(5)

25
```

With the `from` command we can import only part of a module. Example:

```
from modules.first_module import square
```

Now we call the `square` function directly:

```
square(6)

36
```

We can also rename functions or modules. Example:

```
from modules.first_module import square as sqr
sqr(7)

49
```

Where does Python look for modules?

1. In the current folder (the one in which you launched `jupyter notebook`),
2. in the folders defined in `PYTHONPATH`,
3. in the folders where Python is installed (the `site-packages` subdirectory).

2.3.1 Importing modules and the Jupyter notebook

In this book, for pedagogical reasons, we will always import modules at the point where we first need them. **The rules of good practice and clear code recommend that the imports of all modules be done at the top of the Jupyter notebook!**

2.3.2 The pickle module

We will often use the `pickle` module for very fast and compact storing and reading of data. `pickle` stores the values of a variable in the same form as they are written in memory. In the case of real numbers, it therefore **does not lose accuracy** (writing to text form loses accuracy, since we define the number of printed digits, which can be smaller than the number of decimal places in memory). In addition, the representation **in memory is usually less demanding** than the representation in text form. Since the values do not need to be converted to/from text form, writing/reading is also **very fast!** Here we will look at the basic usage; `pickle` is described comprehensively in the [documentation](#).

`pickle` has two important methods:

- `dump(obj, file, protocol=None, *, fix_imports=True)` for storing the object `obj`,
- `load(file, *, fix_imports=True, encoding="ASCII", errors="strict")` for reading the file `file`.

We will look at the usage with an example. First we import the module:

```
| import pickle
```

Let's write the values of the dictionary `a` to the file `data/stanje1.pkl`:

```
| a = {'number': 1, 'size': 10}
|
| with open('data/stanje1.pkl', 'wb') as file:
|     pickle.dump(a, file, protocol=-1) # note: use protocol=-1 to use the latest
|     version
```

Note that the file was opened for writing in binary form `wb`. It is also important that we use the latest version of the protocol (the `protocol` parameter); the default one is not optimized for speed and does not write in binary form.

Now let's read the values back into `b` and display it:

```
| with open('data/stanje1.pkl', 'rb') as file:
|     b = pickle.load(file)
```

```
| b
```

```
| {'number': 1, 'size': 10}
```

Above we claimed that the approach is very fast; let's verify this.

First we prepare the data (a list of 50000 values of type `int`):

```
| data = list(range(50000))
```

Writing in text form

```
| %%timeit
| file = open('data/data.txt', mode='w')
| for d in data:
|     file.write(f'{d:7.2e}\n' )
| file.close()
```

```
| 25.1 ms \pm 394 \mus per loop (mean \pm std. dev. of 7 runs, 10 loops each)
```

Writing with `pickle`:

```
| %%timeit
| with open('data/data.pkl', 'wb') as file:
|     pickle.dump(data, file, protocol=-1)
```

```
| 988 \mus \pm 50 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

That is almost a 20-fold speedup!

2.4 Using modules from the internet

In addition to the built-in modules and those we prepare ourselves, there is also a large set of modules that we can find on the internet and that add new functionality to Python. It is these modules that make the Python ecosystem so useful.

Technically, a module is a single file; when some larger module contains several modules, we can start talking about *packages*. We will learn about the most important packages in the field of engineering sciences within this book.

We can *update* modules or packages, or install *new* ones; for this we are helped by so-called *package managers*. The most common package manager is `pip` ([documentation](#)).

2.4.1 The pip package manager

`pip` is a package manager with a long history and supports a large number of packages ([pypi.org](#)).

We find already installed packages with a command in the command line ([documentation](#)):

```
| pip list
```

Instead of output to the command line, we will often use the option of the *Jupyter notebook* where, with an exclamation mark (!) in a code cell, we execute a command in the command line (this is a short form of the magic command `%sx` - shell execute, see the [documentation](#)). Note: two exclamation marks would return the entire result directly into the notebook (since the output is very long, we avoided this).

Let's look at an example:

```
| result = !pip list
| result[:5]

['Package          Version',
 '-----',
 'attrs            26.1.0',
 'btrfsutil        7.1',
 'certifi          2026.7.22']
```

We install packages with the command (in the command line):

```
| pip install [package_name]
```

we update with:

```
| pip install [package_name] --upgrade
```

and we remove with:

```
| pip uninstall [package_name]
```

For more, see the [documentation](#).

Example of installing a package

Otherwise, we most often find packages on the internet ([pypi.org](#)). Go to the mentioned portal and look for packages on the topic of downloading videos from [www.youtube.com](#). By searching for “youtube download” we find a promising and well-maintained package `yt-dlp`, which we install (from the command line):

```
| pip install yt-dlp
```

Now let's use the module (for usage details see the [documentation](#)). First we import the entire package:

```
import yt_dlp
settings = {'format': 'best[ext=mp4]'} # best quality in mp4 format
with yt_dlp.YoutubeDL(settings) as yt: # we create an instance of the object (what
    exactly this means, we will learn later)
    yt.download(['https://www.youtube.com/watch?v=P4IrAXbpeyI']) # download the video
```

2.5 Additional

2.6 Formatting date and time

First we import the module for working with date and time, `datetime`, and display the current time with the date (we use the `now()` function):

```
import datetime
now = datetime.datetime.now()
now

datetime.datetime(2026, 9, 25, 15, 0, 35, 454864)
```

We format the time information:

```
f'{now:%Y -%m -%d %H:%M:%S}'

'2026 -09 -25 15:00:35'
```

Formatting date and time follows these methods:

- `strftime` ([documentation](#)) is used to convert from `datetime` to a string of characters,
- `strptime` ([documentation](#)) is used to convert from a string of characters to the `datetime` type.

How to print the appropriate form (i.e. how to use the characters `%Y`, `%m`, etc.) is likewise explained in detail in the [documentation](#).

Let's look at an example of converting the `datetime` type to a string (relative to the above, we added `%a` so that the name of the day is printed):

```
now.strftime('%Y -%m -%d %H:%M:%S %a')

'2026 -09 -25 15:00:35 Fri'
```

Now in the opposite direction, i.e. from a string to `datetime`:

```
datetime.datetime.strptime('2017 -09 -28 05:31:45', '%Y -%m -%d %H:%M:%S')

datetime.datetime(2017, 9, 28, 5, 31, 45)
```

2.6.1 Some modules

1. `openpyxl` for writing and reading Excel `xlsx/xlsm` files: [source](#),
 - Regular expressions (a very powerful tool for searching through strings): [source](#).

2.6.2 The sys module

Python has a large number of built-in modules (and many more can be downloaded from the internet, e.g. [here](#)) that add various functionalities.

The `sys` module provides access to objects used by the interpreter.

Let's look at an example; first we import the module:

```
| import sys
```

We can check the folders where modules are searched for with the command `sys.path` ([documentation](#)):

```
| sys.path

['c:\\_j\\Work Janko\\Pedagosko delo\\_predavanja\\Programiranje in numericne
metode\\pynm\\notebooks',
 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\python311.zip',
 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\DLLs',
 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\Lib',
 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311',
 '',
 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site
-packages',
 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site
-packages\\win32',
 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site
-packages\\win32\\lib',
 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site
-packages\\Pythonwin']
```

2.6.3 The os module

The `os` module is intended for working with the operating system and takes care of compatibility across different operating systems (if, for example, you write a program on a *Windows* system, it will also work on a *linux* system).

We import the module:

```
| import os
```

We look at the current folder:

```
| os.path.curdir
```

```
| '.'
```

or the current folder in absolute form:

```
| os.path.abspath(os.path.curdir)
```

```
| 'c:\\_j\\Work Janko\\Pedagosko delo\\_predavanja\\Programiranje in numericne
metode\\pynm\\notebooks'
```

How do we find all the files and folders in a directory?

```
| lst = os.listdir()
```

Let's display the first four with the `ipynb` extension:

```
i = 0
for name in lst:
    if os.path.isfile(name):
        if name[-5:] == 'ipynb':
            print(f'I found a file: {name:s}')
            i += 1
        if i > 3:
            break
```

```
I found a file: Lecture 01 - Introduction to Python.ipynb
I found a file: Lecture 09 - Root finding.ipynb
I found a file: Lecture 08 - Approximation.ipynb
I found a file: Lecture 14 - Testing code correctness, user interface.ipynb
```

For more, see the [documentation](#).

Chapter 3

Modules, NumPy, Matplotlib

3.1 Basics of the numpy module

As mentioned above, we can install the `numpy` module with the package manager `pip` using the following command (if we do not yet have it installed):

```
| pip install numpy
```

or update the package to the latest version:

```
| pip install numpy - -upgrade
```

First we import the module (the established convention is to import it under the short alias `np`):

```
| import numpy as np
```

This is one of the most important modules. In short: it is a highly optimized module for numerical computations; excellent tutorials are collected at: github.com/numpy/numpy-tutorials. For an excellent introduction to NumPy you can watch the [YouTube video: NumPy Tutorial \(2021\): For Physicists, Engineers, and Mathematicians](#)

Let's first look at the syntax for a vector of zeros ([documentation](#)):

```
| numpy.zeros(shape, dtype=float, order='C')
```

the arguments are:

- **shape** defines the shape (it can be a multidimensional numerical array),
- **dtype** defines the data type,
- **order** defines the ordering (it can be C or F as in Fortran).

Let's look at an example:

```
| np.zeros((3,4))
```

```
| array([[0., 0., 0., 0.],  
        [0., 0., 0., 0.],  
        [0., 0., 0., 0.]])
```

or:

```
| np.zeros((3,5))
```

```
| array([[0., 0., 0., 0., 0.],  
        [0., 0., 0., 0., 0.],  
        [0., 0., 0., 0., 0.]])
```

`ones` behaves similarly to `zeros`, but instead of zeros the value is 1 ([documentation](#)).

Let's look at an example where we also specify the type `int` (the default type is `float`):

```
| np.ones(4, dtype=int)
| array([1, 1, 1, 1])
```

We will also often use the range of values `arange` ([documentation](#)):

```
| numpy.arange([start, ]stop, [step, ]dtype=None)
```

where the arguments are:

- `start` the starting value of the range (default 0),
- `stop` the ending value of the range,
- `step` the step, and
- `dtype` the value type (if the type is not given, the type of the other arguments is used, e.g. `step`).

Let's look at an example of a range from 0 to 9 (as always in Python, the *from* value is included, but the *to* value is not):

```
| np.arange(9)
| array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

or from 7 to 12 with a step of 2, but as floating-point numbers:

```
| np.arange(7, 12, 2, dtype=float)
| array([ 7.,  9., 11.])
```

We will also often use one more function, namely `linspace` ([documentation](#)):

```
| numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

which returns, for a defined range, a numerical array of values at equal distances (equidistant spacing).

The arguments are:

- `start` the starting value of the range,
- `stop` the ending value of the range,
- `num` the number of points/nodes,
- `endpoint` whether the value at `stop` is included or not,
- `retstep` if `True`, the function returns a tuple (`result`, `step`)
- `dtype` the value type (if the type is not given, the type of the other arguments is used, e.g. `step`).

An example of generating 10 points over the range from $-\pi$ up to and including $+\pi$:

```
| np.linspace(-np.pi, np.pi, 10)
| array([-3.14159265, -2.44346095, -1.74532925, -1.04719755, -0.34906585,
|         0.34906585,  1.04719755,  1.74532925,  2.44346095,  3.14159265])
```

By the way, above we learned that `numpy` has built-in constants (e.g. π):

Let's also look at the built-in functions for generating random numbers. These are found in `numpy.random` ([documentation](#)).

First let's look at the function `numpy.random.seed()`, which is used to reset the random number generator ([documentation](#)). This means that with the same seed, different users can generate the same random numbers!

Note: the functions `numpy.random.seed()`, `rand()`, `random()`, `normal()`, etc. belong to the older (*legacy*) interface, which still works. The modern approach is to create your own generator and call its methods ([documentation](#)):

```
rng = np.random.default_rng(0) # seed 0
rng.random(3)                  # uniform distribution on [0, 1)
rng.normal(size=3)              # normal distribution
```

The line below:

```
np.random.seed(0)
```

will cause a call to the uniformly distributed random number generator `numpy.random.rand()` ([documentation](#)) to always result in the same values:

```
np.random.rand(3)
array([0.5488135 , 0.71518937, 0.60276338])
```

Let's test whether what we wrote is true, and reset the seed and call the generator:

```
np.random.seed(0)
np.random.random(3)
array([0.5488135 , 0.71518937, 0.60276338])
```

3.1.1 Notation of matrices and vectors

A matrix has dimensions $m \times n$, where m in the first position is the number of rows and n is the number of columns. An example of defining a matrix of dimension $m \times n = 3 \times 2$ is:

```
a = np.zeros((3, 2))
a
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

A vector can be written as a **row vector**:

```
b = np.zeros(3) # (1 x 3)
b
array([0., 0., 0.]])
```

or as a **column vector**:

```
c = np.zeros((3, 1)) # 3 x 1
c
array([[0.],
       [0.],
       [0.]])
```

In the `numpy` module we can represent vectors and matrices as:

- `numpy.array` (recommended, [documentation](#)),
- `numpy.matrix` ([documentation](#)).

The first form (`numpy.array`) is recommended; it does not follow the mathematical notation exactly (more on this later), but it allows for **simpler** programming and is also a numerically **more efficient** approach ([source](#)). We will not cover the `numpy.matrix` approach here.

We refer to an *array* as a **multidimensional numerical array**, or sometimes a **multidimensional list** (since arrays share some similarities with ordinary lists).

3.1.2 Slicing

We already looked at **slicing** of lists in the chapter Introduction to Python. Similar slicing, but more general, also applies to numerical arrays of the `numpy` module.

The slicing syntax ([documentation](#)) is:

```
| numpy_array[from:to:step]
```

where:

- indexing starts at 0 (as usual in Python),
- **from** means $\geq$,
- **to** means $<$,
- **from**, **to**, **step** are optional parameters,
- if the **from** parameter is not given, it means from the *beginning*,
- if the **to** parameter is not given, it means up to and including the last element,
- if the **step** parameter is not given, it means a step of 1.

Example from element 3 to element 8 with a step of 2:

```
| b = np.arange(10)
| b[3:8:2]
```

```
| array([3, 5, 7])
```

Example from element 3 onward:

```
| b[3:]
| array([3, 4, 5, 6, 7, 8, 9])
```

Example from element 3 onward, but every third one:

```
| b[3::3]
| array([3, 6, 9])
```

Example of the last 5 elements, but every other one:

```
| b[-5::2]
| array([5, 7, 9])
```

Example of the last 5 elements without the last 2, but every other one:

```
| b[-5: -2:2]
| array([5, 7])
```

3.1.3 Let's also look at slicing a multidimensional numerical array.

We perform multidimensional slicing by separating the dimensions with a comma:

```
| numpy_array[slice0, slice1,...]
```

where `slice0` slices index 0 (the first dimension) in the form `from:to:step`, `slice1` slices index 1 (the second dimension), and so on.

Let's look at an example; first we prepare a list of 15 numbers (`np.arange`), then with the method `reshape()` ([documentation](#)) we change the shape into a 3 x 5 matrix:

```
| a = np.arange(15)
| a = a.reshape((3,5))
| a
|
| array([[ 0,  1,  2,  3,  4],
|        [ 5,  6,  7,  8,  9],
|        [10, 11, 12, 13, 14]])
```

We can check the shape of a numerical array with the attribute `shape` ([documentation](#)).

```
| a.shape
|
| (3, 5)
```

We will, however, learn about attributes in detail when we cover classes.

Let's display the row with index 0:

```
| a[0]
|
| array([0, 1, 2, 3, 4])
```

We would get the same result by displaying the row with index 0 and all columns:

```
| a[0,:]
|
| array([0, 1, 2, 3, 4])
```

We often want to access columns, e.g. the column with index 1 (so we slice all rows and the column with index 1):

```
| a[:,1]
|
| array([ 1,  6, 11])
```

Let's also look at an example of slicing the first two rows and the last two columns:

```
| a[:2, -2:]
|
| array([[3, 4],
|        [8, 9]])
```

Similar logic is used for dimensions higher than 2.

3.1.4 Operations on numerical arrays

Let's now look in more detail at some of the basic advantages of the numerical array `numpy.array` compared to an ordinary Python list.

First we prepare an ordinary Python list:

```
| a = [1, 2, 3, 4, 5, 6, 7]
| a
|
| [1, 2, 3, 4, 5, 6, 7]
```

And then the numerical array `numpy.array` (directly from the list `a`):

```
| b = np.array(a)
| b
|
| array([1, 2, 3, 4, 5, 6, 7])
```

When we print `b`, we are notified that it is an `array(...)`.

Let's see how a Python list behaves under multiplication:

```
| 2*a
| [1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7]
```

We notice that the list is duplicated, not the values, which is perhaps what we would expect.

Let's see how the numerical array `numpy.array` behaves under multiplication:

```
| 2*b
| array([ 2,  4,  6,  8, 10, 12, 14])
```

We notice that the values are doubled; just as we would expect when multiplying, for example, a scalar value and a vector!

3.1.5 Arithmetic operations

The selection of arithmetic operations that `numpy` performs at the level of individual elements is (in increasing priority):

- `x + y` sum,
- `x - y` difference,
- `x * y` product,
- `x / y` division,
- `x // y` integer division (the result is an integer rounded down),
- `x % y` remainder of integer division,
- `x ** y` returns `x` to the power `y`.

Example:

```
| b + 3*b - b**2
| array([ 3,  4,  3,  0, -5, -12, -21])
```

All arithmetic operations are listed in the [documentation](#), and besides the short forms we also have *long* ones, e.g.: `numpy.power(x, y)` instead of `x**y`; example:

```
| np.power(b, 2)
| array([ 1,  4,  9, 16, 25, 36, 49])
```

By the way, we notice that the result is of type `int32` (integer). When we created the name `b`, we created a numerical array with elements of type `int32`.

3.1.6 Mathematical functions

`numpy` provides practically all the necessary mathematical (and other) operations; let's list them by groups, as they are structured in the [documentation](#):

- [trigonometric functions](#),
- [hyperbolic functions](#),
- [rounding functions](#),
- [functions for sums, products, and derivatives](#),
- [exponents and logarithms](#),
- [other special](#) and [miscellaneous](#) functions.

Let's look at an example of the function `sin()`:

```
| a = np.linspace(0, 2*np.pi, 5)
| np.sin(a)
|
| array([ 0.0000000e+00,  1.0000000e+00,  1.2246468e -16, -1.0000000e+00,
|        -2.4492936e -16])
|
| Let's also look at the execution speed:
|
| %timeit -n100 np.sin(a)
|
| 585 ns \pm 81.6 ns per loop (mean \pm std. dev. of 7 runs, 100 loops each)
```

3.1.7 Data types

numpy has predefined data types (statically). The full list of possible types is given in the [documentation](#).

We will focus mainly on the following types:

- `int` - integer (arbitrarily large)
- `float` - floating-point number ([documentation](#))
- `complex` - complex floating-point number
- `object` - Python object.

Let's look at a few examples (integers, floating-point numbers, and complex numbers):

```
| np.arange(5, dtype=int)
|
| array([0, 1, 2, 3, 4])
|
| np.arange(5, dtype=float)
|
| array([0., 1., 2., 3., 4.])
|
| np.arange(5, dtype=complex)
|
| array([0.+0.j, 1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j])
```

3.1.8 Changing the elements of a numerical array (`numpy.array`)

We change data in a way similar to ordinary lists; if we use slicing, we must make sure that the data on the left and right sides of the equals sign have the same shape (`array.shape`).

Let's look at an example where, in a 3 x 4 matrix of zeros, we change the element with index [2, 3]:

```
| a = np.zeros((3, 4))
| a[2, 3] = 100
| a
|
| array([[ 0.,  0.,  0.,  0.],
|        [ 0.,  0.,  0.,  0.],
|        [ 0.,  0.,  0., 100.]])
```

Now let's also change the elements of the first two rows and first two columns to the value 1:

```
| a[:2, :2] = np.ones((2, 2))
| a
|
| array([[ 1.,  1.,  0.,  0.],
|        [ 1.,  1.,  0.,  0.],
|        [ 0.,  0.,  0., 100.]])
```

Let's multiply the column with index 1 by 2:

```
a[:,1] = 2 * a[:,1]
a
array([[ 1.,  2.,  0.,  0.],
       [ 1.,  2.,  0.,  0.],
       [ 0.,  0.,  0., 100.]])
```

Be careful that in this way we create a *view* of the data (**we do not make a copy of the data**).

As an example, let's first create a new name **view**:

```
view = a[:, 2]
view
array([0., 0., 0.] )
```

Now let's change the selected values of the numerical array **a**:

```
a[:, 2] = 5
a
array([[ 1.,  2.,  5.,  0.],
       [ 1.,  2.,  5.,  0.],
       [ 0.,  0.,  5., 100.]])
```

We did not change the values of **view**. But because the name points to the same location as **a[:, 2]**, the values are changed:

```
view
array([5., 5., 5.] )
```

If we want a copy, we have to do it like this:

```
copy_arr = a[:, 2].copy()
copy_arr
array([5., 5., 5.] )
```

and the result **copy_arr** remains unchanged:

```
a[:, 2] = 2
copy_arr
array([5., 5., 5.] )
```

3.1.9 Basics of matrix computation

If you would like to review the mathematical basics of matrix computation, then follow this [link](#) (it is a short and good overview by Prof. Dr. T. Košir). We will look at how to perform matrix computations using the **numpy** package.

First let's define the matrices **A** and **B**:

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

and the vectors **x** and **y**.

```
| x = np.array([1, 2])
| y = np.array([3, 4])
```

We compute the dot product of two vectors with the function `dot()` ([documentation](#)):

```
| numpy.dot(a, b, out=None)
```

where the arguments `a` and `b` represent numerical arrays `numpy.array` (of arbitrary dimension) that we want to multiply. If `a` and `b` are of dimension 1, the dot product is performed. For dimension 2 (matrices), the matrix product is computed. For using the function `dot()` with dimensions greater than 2: see the [documentation](#).

Let's look at an example of multiplying two vectors; this can be done like this:

```
| np.dot(x, y)
| np.int64(11)
```

or also like this:

```
| x.dot(y)
| np.int64(11)
```

Above we mentioned that `numpy.array` does not consistently follow the mathematical notation. If it did, one of the vectors would have to be a row vector and the other a column vector. `numpy` simplifies this, and therefore the code is easier to read and shorter.

Let's now look at multiplying a matrix by a vector (notice that transposing `x` is not necessary):

```
| np.dot(A, x)
| array([5, 7])
```

We can, of course, prepare the mathematically correct transposed form of the vector (but we see that the notation is awkward):

```
| A.dot(np.transpose([x]))
| array([[5],
|         [7]])
```

Transposition also has a short form, via the attribute `T`, e.g. for the matrix `A`:

```
| A.T
| array([[1, 3],
|         [2, 2]])
```

Let's look at an example of multiplying two matrices:

```
| np.dot(A, B)
| array([[5, 5],
|         [7, 7]])
```

Since Python 3.5 the operator `@` ([documentation](#)) is also used for matrix (and vector) multiplication, which allows for short and clear notation.

The examples above written with the `@` operator:

```
| x @ y
```

```
| np.int64(11)
|
| A @ x
|
| array([5, 7])
|
| A @ B
|
| array([[5, 5],
|        [7, 7]])
```

We compute the **cross product** with the function `numpy.cross()` ([documentation](#)):

```
| numpy.cross(a, b, axisa= -1, axisb= -1, axisc= -1, axis=None)
```

where `a` and `b` define the components of the vectors. The vectors must have three components, and the result is a vector with three components. Planar vectors (only the x and y components) are no longer accepted since NumPy 2.0; we pad them with the component $z = 0$, and only the z component of the result is nonzero. The function can also be used on multidimensional numerical arrays, and the remaining arguments serve that purpose (see the documentation).

Example of the cross product of planar vectors:

```
| x = np.array([1, 0])
| y = np.array([0, 1])
| np.cross(np.append(x, 0), np.append(y, 0))[2] # pad with z=0 and take the z component
|
| np.int64(1)
```

Example of the cross product of spatial vectors:

```
| x = np.array([1, 0, 0])
| y = np.array([0, 1, 0])
| np.cross(x, y)
|
| array([0, 0, 1])
```

3.1.10 Some functions of the numpy library

We will also look at some functions that we need more or less often.

We define the identity matrix with the function `numpy.identity()` ([documentation](#)):

```
| numpy.identity(n, dtype=None)
```

where the argument `n` defines the number of rows and columns of the square matrix. The type `dtype` is `float` by default.

Example of the identity matrix:

```
| A = np.identity(3)
| A
|
| array([[1., 0., 0.],
|        [0., 1., 0.],
|        [0., 0., 1.]])
```

We access the diagonal elements of a matrix using the function `numpy.diagonal()` ([documentation](#))

```
| numpy.diagonal(a, offset=0, axis1=0, axis2=1)
```

If the matrix is two-dimensional, then the function with the default arguments returns the diagonal. If the dimension is higher than 2, the axes `axis1` and `axis2` are used to extract a two-dimensional array, and then the diagonal is determined according to the elements `[i, i+offset]`.

Let's look at an example of extracting the diagonal:

```
| np.diagonal(A)
```

```
| array([1., 1., 1.])
```

and the use of `offset=1` for the adjacent diagonal (first we prepare a nonsymmetric matrix):

```
| A[0, 1] = 10
```

```
| np.diagonal(A, offset=1)
```

```
| array([10., 0.])
```

The function `numpy.trace()`, which computes the sum (trace) of the diagonal elements, has a syntax similar to `numpy.diagonal()` ([documentation](#)):

```
| numpy.trace(a, offset=0, axis1=0, axis2=1,  
|           dtype=None, out=None)
```

Example of the trace of the diagonal:

```
| np.trace(A)
```

```
| np.float64(3.0)
```

and then of the adjacent diagonal:

```
| np.trace(A, offset=1)
```

```
| np.float64(10.0)
```

We are often interested in the largest or smallest element of some numerical array. `numpy` is very general here. Let's look at the example of the function `numpy.max()` ([documentation](#)):

```
| numpy.max(a, axis=None, out=None)
```

Let's highlight the argument `axis`, which tells us along which index we are searching for the maximum value. If `axis=None`, the largest value in the entire array is determined.

Example of computing the largest value of the entire array (let's first look at `A`):

```
| A
```

```
| array([[ 1., 10.,  0.],  
|        [ 0.,  1.,  0.],  
|        [ 0.,  0.,  1.]])
```

```
| np.max(A)
```

```
| np.float64(10.0)
```

Example of computing the largest value across rows (i.e. by column):

```
| np.max(A, axis=0)
```

```
| array([ 1., 10.,  1.])
```

Example of computing the largest value across columns (i.e. by row):

```
| np.max(A, axis=1)
| array([10.,  1.,  1.]
```

The counterpart of the function `max()` is `numpy.argmax()`, which determines the indices of the largest value.

Example of use:

```
| A
| array([[ 1., 10.,  0.],
|        [ 0.,  1.,  0.],
|        [ 0.,  0.,  1.]])
|
| np.argmax(A, axis=0)
|
| array([0, 0, 2])
```

3.1.11 Linear algebra with numpy

Later we will learn about linear algebra in more detail and will write algorithms ourselves. Here let's look at some basics that are built into the `numpy` module.

As an example, let's first define a matrix and a vector:

```
| A = np.array([[4, -2],
|               [-2, 4]])
| b = np.array([1, 2])
```

We compute the inverse matrix using the function `numpy.linalg.inv()` ([documentation](#)):

```
| numpy.linalg.inv(a)
|
| Example:
|
| np.linalg.inv(A)
|
| array([[0.33333333, 0.16666667],
|        [0.16666667, 0.33333333]])
```

We solve the system of linear equations defined by the coefficient matrix `a` and the constant vector `b` using the function `numpy.linalg.solve()` ([documentation](#)):

```
| numpy.linalg.solve(a, b)
|
| Example:
|
| solution = np.linalg.solve(A, b)
| solution
|
| array([0.66666667, 0.83333333])
```

We check the equality of the elements of the numerical arrays `a` and `b` (within a certain tolerance) with the function `numpy.isclose()` ([documentation](#)):

```
| numpy.isclose(a, b, rtol=1e -05, atol=1e -08, equal_nan=False)
|
| Example:
|
| np.isclose(np.dot(A, solution), b)
|
| array([ True,  True])
```

3.1.12 Vectorization of algorithms

In this section we want to highlight the vectorization of algorithms. Given how Python and `numpy` work, loops should be avoided. We can perform computations significantly faster if we manage to vectorize them; this means that we perform computations at the level of vectors (or numerical arrays) and not of individual elements.

As an example, let's first prepare the data (two vectors of length 1000)

```
N = 1000
a = np.arange(N)
b = np.arange(N)
```

Let's compute the dot product of the vectors using a `for` loop:

```
c = 0
for i in range(N):
    c += a[i] * b[i]
c
```

```
np.int64(332833500)
```

Let's measure the speed:

```
%%timeit -n100
c = 0
for i in range(N):
    c += a[i] * b[i]
```

```
208 \mus \pm 20.8 \mus per loop (mean \pm std. dev. of 7 runs, 100 loops each)
```

We obtain the same result in vector form:

```
%%timeit -n100
c = a @ b
```

```
1.44 \mus \pm 153 ns per loop (mean \pm std. dev. of 7 runs, 100 loops each)
```

We see that the vector approach is significantly faster (for an even faster approach see `numba` in the appendix)!

3.2 Basics of the `matplotlib` module

In Python we have several options for displaying results graphically. The most useful packages are:

- `matplotlib` for high-quality, highly customizable figures (relatively slow),
- `pyqtgraph` for high-quality and customizable user interfaces (very fast),
- `bokeh` for interactive display in the browser (relatively fast).

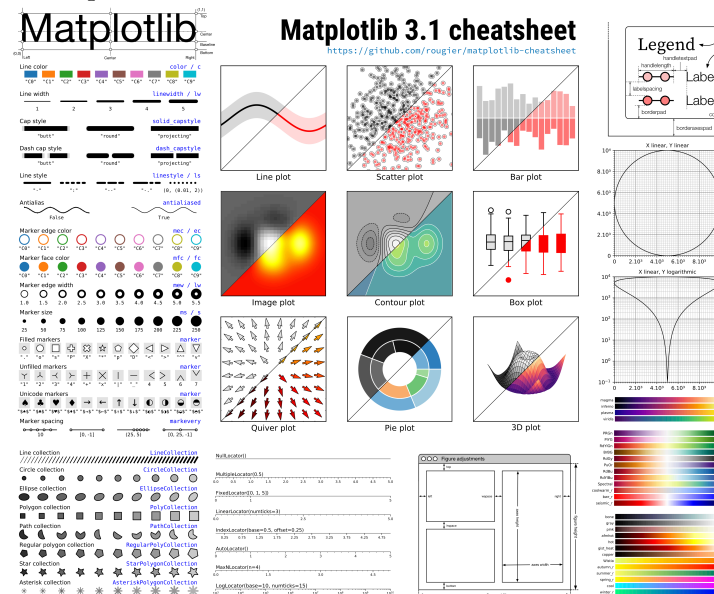
There are many others; a good overview was given by Jake VanderPlas at the [PyCon 2017](#) conference (he is also the author of the package for declarative visualization: *Altair*).

3.2.1 Basic usage

The most widespread and most generally used is the package `matplotlib`:



The capabilities of the package are best illustrated by the [gallery](#). It is a very sophisticated package, and here we will look at some basics based on examples.



When using it, you may find the [cheat sheet](#) helpful:
Typically we import `matplotlib.pyplot` as `plt`:

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

Within a *Jupyter notebook* there are two ways to display a figure (in square brackets is the magic command for triggering it):

1. `[%matplotlib inline]` : figures are embedded in the notebook (**inline** mode),
2. `[%matplotlib widget]` : figures are interactively embedded in the notebook (**inline interactive** mode), requires installation of the **ipywidgets** package,
3. `[%matplotlib notebook]` : figures are interactively embedded in the notebook (**inline interactive** mode).

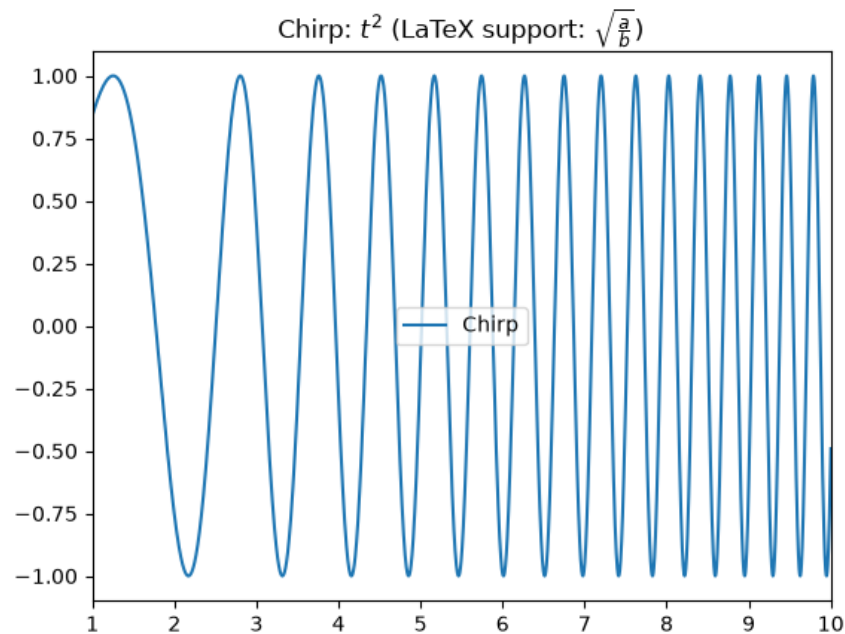
Note: *the interactive mode does not display correctly in the passive, web/pdf version of this book.*

Here we will most often use the *inline* mode:

```
%matplotlib inline
```

A short example:

```
t = np.linspace(1, 130, 44000)
chirp = np.sin(t**2)
plt.plot(t, chirp, label='Chirp')
plt.xlim(1, 10)
plt.title('Chirp:  $t^2$  (LaTeX support:  $\sqrt{\frac{a}{b}}$ )')
plt.legend();
plt.show()
```



By the way, why do we call this a chirp (or a quadratic chirp)? To get the answer, let's play the data through a speaker:

```
| from IPython.display import Audio, display
| display(Audio(data=chirp, rate=44000))
```

```
| <IPython.lib.display.Audio object>
```

Let's now activate the interactive mode (see also `%matplotlib`):

```
| %matplotlib notebook
| %matplotlib notebook
```

```
| plt.plot([1,2,3], [2,4,5]);
```

```
| <IPython.core.display.Javascript object>
```

We can now interactively click on the figure above and also augment it with code:

```
| plt.title('Title added later!')
| plt.xlabel('Time [t]');
```

3.2.2 Interactive usage

In this course we will often use interactivity when working with graphical displays. For this purpose we first import `interact` from the `ipywidgets` package:

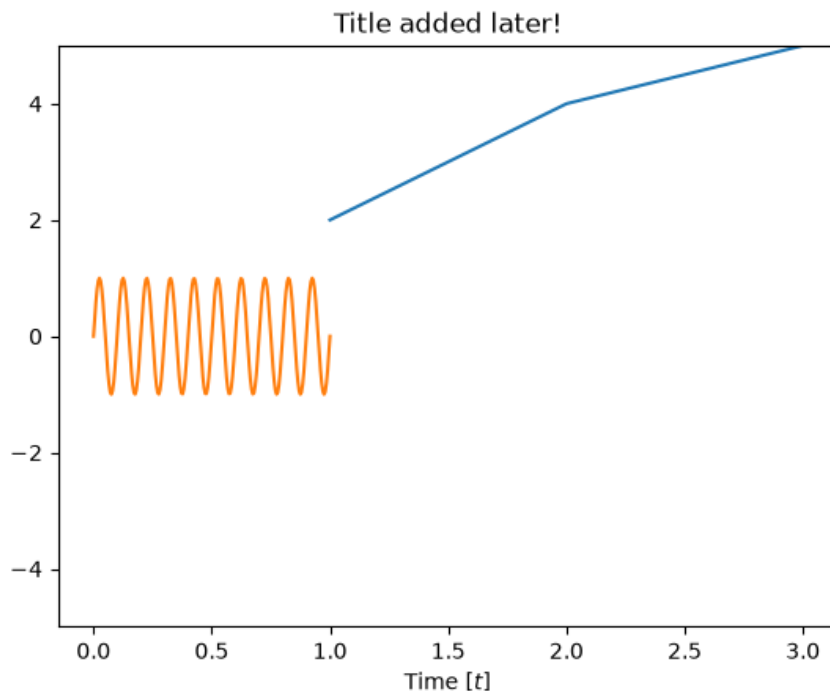
```
| from ipywidgets import interact
```

Then we define the figure as a function with the arguments `amplitude`, `fr`, `phase`, and `damping`:

```
| def figure_plot(amplitude=1, fr=10, phase=0, damping=0.):
|     t = np.linspace(0, 1, 200)
|     f = amplitude * np.sin(2*np.pi*fr*t - phase) * np.exp(-damping*2*np.pi*fr*t)
|     plt.plot(t, f)
|     plt.ylim(-5, 5)
|     plt.show()
```

Let's go back to inline usage and call the function for plotting the figure (with default arguments):

```
%matplotlib inline
figure_plot()
```



If we pass the figure-plotting function to the `interact` function, the latter takes care of the interactive controls with which we can change the parameters for calling the `figure_plot` function:

```
interact(figure_plot);

interactive(children=(IntSlider(value=1, description='amplitude', max=3, min= -1),
IntSlider(value=10, descript...
```

We can also define the parameter ranges ourselves:

```
interact(figure_plot, amplitude=(1, 5, 1), damping=(0, 1, 0.05), fr=(10, 100, 1),
phase=(0, 2*np.pi, np.pi/180));

interactive(children=(IntSlider(value=1, description='amplitude', max=5, min=1),
IntSlider(value=10, descripti...
```

3.2.3 Advanced usage

Displaying multiple functions

Let's look at a simple example of displaying multiple functions (first, optionally, let's define the fonts for output to pdf):

```
from matplotlib import rc
rc('font',**{'family':'serif'})
plt.rcParams['pdf.fonttype'] = 42

x = np.linspace(0, 10, 100)

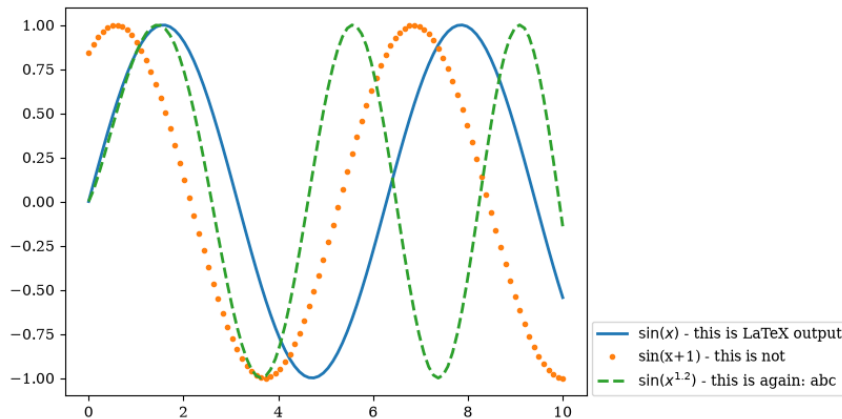
y1 = np.sin(x)
```

```

y2 = np.sin(x+1)
y3 = np.sin(x**1.2)

plt.plot(x, y1, '-', label=r'$\sin(x)$ - this is LaTeX output', linewidth = 2);
plt.plot(x, y2, '.', label='sin(x+1) - this is not', linewidth = 2);
plt.plot(x, y3, '- -', label=r'$\sin(x^{\{1.2\}})$ - this is again: abc', linewidth = 2);
plt.legend(loc=(1.01,0));
plt.savefig('data/prvi plot.pdf')

```



Displaying multiple figures

We display multiple figures using the method `subplot`, which defines a grid and the position of the next plot.

```

plt.subplot(2, 2, 1)
plt.plot(x, y1, 'r')
plt.subplot(2, 2, 2)
plt.plot(x, y2, 'g')

```

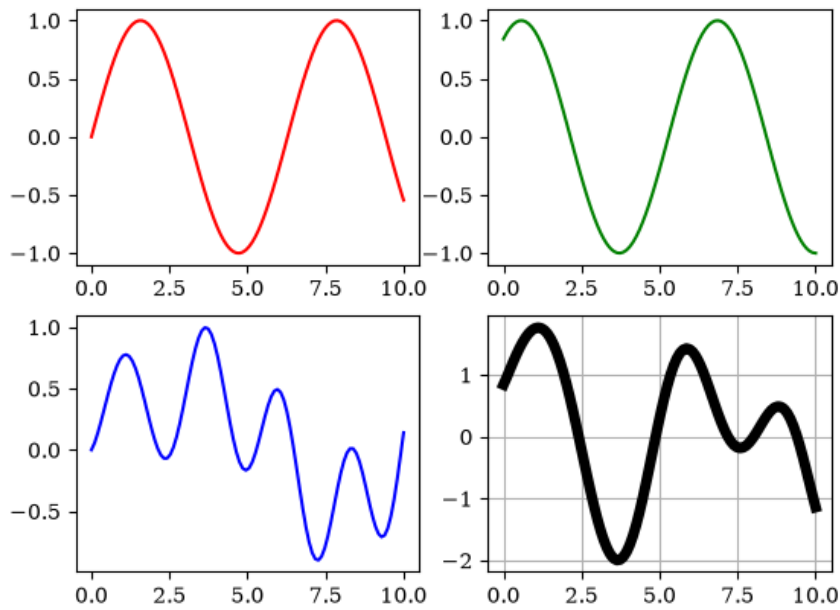
In the example above `plt.subplot(2, 2, 1)` means: let the grid be 2 x 2, draw into figure 1 (top left). Next `plt.subplot(2, 2, 2)` is called, which means a 2 x 2 grid, draw into figure 2 (top right), and so on. Notice that the figure index starts at 1; one could say this is an inconsistency in Python, but the reason is that `matplotlib` was born out of the idea of giving Python users a plotting style as similar as possible to the one they knew in Matlab (where the index starts at 1).

A working example is:

```

plt.subplot(2, 2, 1)
plt.plot(x, y1, 'r')
plt.subplot(2, 2, 2)
plt.plot(x, y2, 'g')
plt.subplot(2, 2, 3)
plt.plot(x, y2*y3, 'b')
plt.subplot(2, 2, 4)
plt.plot(x, y2+y3, 'k', linewidth=5);
plt.grid()

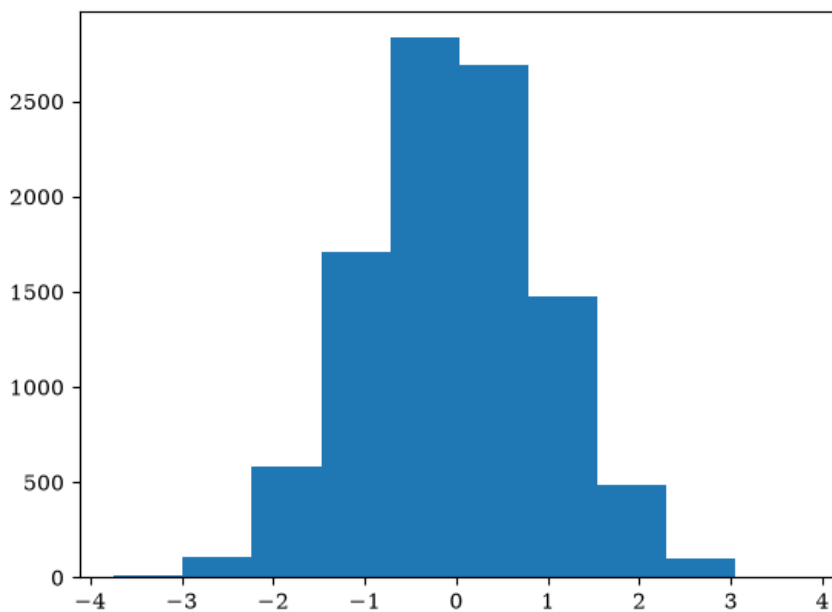
```



Histogram

Let's generate 10000 normally distributed samples and display them as a histogram:

```
np.random.seed(0)
x = np.random.normal(size=10000)
plt.hist(x);
```



3.2.4 Using examples from matplotlib.org

We can import examples from the [gallery](http://matplotlib.org/gallery) using the magic function `%load`.

Try:

- `%load http://matplotlib.org/mpl_examples/lines_bars_and_markers/line_demo.py`
- `%load http://matplotlib.org/examples/widgets/slider_demo.py`

3.3 Additional:

3.4 Numba

The `numba` package has been developing rapidly in recent times and can speed up numerical execution even further (also in connection with graphics cards, i.e. GPUs).

As an example, here we use `jit` ([just-in-time compilation](#)) from the `numba` package:

```
import numpy as np
from numba import jit
```

We use `jit` as a [decorator](#) of the function we want to speed up:

```
@jit
def dot_product(a, b):
    c = 0
    for i in range(N):
        c += a[i] * b[i]
    return c
```

Now let's define the vectors:

```
N = 1000
a = np.arange(N)
b = np.arange(N)
```

Let's check the speed of the `numpy` dot product:

```
%%timeit -n1000
a @ b

1.41 \mus \pm 46.9 ns per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

Let's also check the speed of the `numba`-accelerated version. We perform the first call so that the compilation runs, then we measure the time:

```
dot_product(a, b)

332833500

%%timeit -n1000
dot_product(a, b)

621 ns \pm 29.5 ns per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

We see that we improved the speed even further!

3.5 matplotlib: animations, callback, XKCD style

With `matplotlib` we can also create **animations**. You can find two examples here:

- [Simple animation](#),
- [Double pendulum](#).

More in the [documentation](#).

A figure with a **callback**:

- [Drawing a line](#).

You can also prepare *hand-drawn* figures (**XKCD style**):

- [XKCD style](#).

3.6 For the most enthusiastic

1. Learn something new at chrisalbon.com.

Chapter 4

Object-oriented programming, symbolic computation

4.1 Object-oriented programming

In programming there are various approaches; the Python documentation (docs.python.org) mentions, for example:

1. **procedural**: a list of instructions on what to do (e.g.: `C`, `Pascal`)
2. **declarative**: we describe what we want, and the programming language carries it out (e.g.: `SQL`)
3. **functional**: programming based on functions (e.g.: `Haskell`)
4. **object-oriented**: the program is based on objects that have properties, functions, ... (e.g.: `Java`, `Smalltalk`)

Python is an object-oriented programming language, but it does not force us to use objects in every case. As we will see later, object-oriented programming has many advantages, but it can often be cumbersome and would make a program unnecessarily complex. For this reason, we avoid explicit object-oriented programming whenever possible.

Object-oriented programming in Python is based on **classes**, and objects are **instances** of a class. We will look at only some of the basics of object-oriented programming (so that you can more easily understand the code of other authors and adapt it to your own needs).

A class is defined with the `class` statement ([documentation](#)):

```
class ClassName:
    '''docstring'''
    [expression 1]
    [expression 2]
    :
    .
```

where the class name (i.e., `ClassName`) is, per PEP8, written with a capital first letter. If the name consists of several words, each one is capitalized (the so-called *CamelCase* convention).

Let's look at an example:

```
class Rectangle:
    """Class for a rectangle object"""

    def __init__(self, width=1, height=1): # this is the object constructor. It runs
    when we call Rectangle(width=1, height=4)
        self.width = width
        self.height = height # height is an attribute of the object

    def area(self):
        return self.width * self.height
```

```
def set_width(self, width=1):  
    self.width = width
```

Before we go into the details of understanding the code, let's create an instance of the class (i.e., an object):

```
my_rectangle = Rectangle()
```

Functions defined inside a class are called **methods** when we call them on objects.

In the example above, we use the **area** method like this:

```
my_rectangle.area()
```

```
1
```

The name **self** is a reference to the instance of the class (the object that will be created). Names defined inside the class become **attributes** of the object.

An example of the **height** attribute:

```
my_rectangle.height
```

```
1
```

Where does the result 1 come from? When we create an object, the initialization function `__init__()` runs first; its arguments are the arguments we pass to the class.

Example:

```
your_rectangle = Rectangle(height=5, width=3)  
your_rectangle.area()
```

```
15
```

We have also prepared a method that changes the **width** attribute:

```
my_rectangle.set_width(width=100)  
my_rectangle.area()
```

```
100
```

We can also change attributes directly, but we usually avoid this (because of the possibility of errors and misuse).

Example:

```
my_rectangle.height = 3  
my_rectangle.area()
```

```
300
```

4.1.1 Inheritance

An important property of classes is inheritance; the name itself says it all: just as we humans inherit from our parents, the same holds for classes. Each class (**class**) can thus inherit the properties of some other class ([documentation](#)). It can even have several parents (we will not go into these details here).

The syntax of a class that inherits is:

```
class Child(Parent):
    [expression]
    .
    .
```

Note: even if we do not define a *parent* for a class, the class inherits from `object`.
An example where the new class `Square` inherits from an existing one (`Rectangle`):

```
class Square(Rectangle):
    "Square class"

    def __init__(self, width=1):
        # call the initialization of the Rectangle class
        super().__init__(width=width, height=width)

    def set_width(self, width):
        self.width = width
        self.height = width
```

Now let's look at how it is used:

```
my_square = Square(width=4)
```

The `Square` class does not define a method for computing the area, but it inherited it from the `Rectangle` class, and therefore *has a method* for computing the area:

```
my_square.area()
```

```
16
```

If we change the width, the area changes accordingly:

```
my_square.set_width(5)
my_square.area()
```

```
25
```

Example of inheriting from the list class

First we prepare a list:

```
my_list = list([1,2,3])
my_list
```

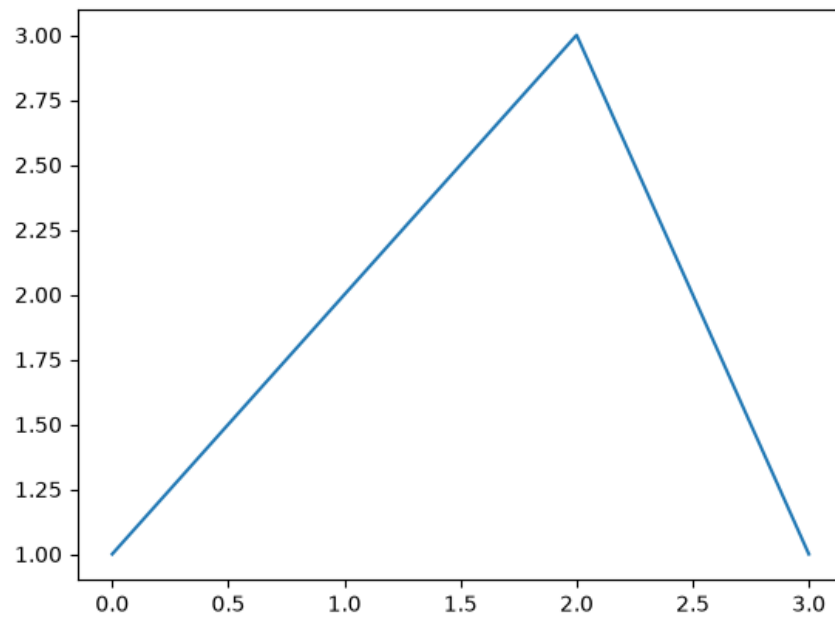
```
[1, 2, 3]
```

If we want to add a value to the list, we use the `append` method (this is a method that objects of type `list` have):

```
my_list.append(1)
```

Then we plot the list (first we import `matplotlib`):

```
import matplotlib.pyplot as plt
plt.plot(my_list)
plt.show()
```



If we do something like this often, then it is better to prepare our own `List` class that inherits from `list` and add a method `draw` for plotting:

```
class List(list):  
    def draw(self):  
        plt.plot(self, 'r.', label='Long text')  
        plt.legend()  
        plt.ylim( -5, 5)  
        plt.show()
```

An instance of the object is:

```
my_list_obj = List([1, 2, 3])  
type(my_list_obj)
```

```
__main__.List
```

```
my_list_obj
```

```
[1, 2, 3]
```

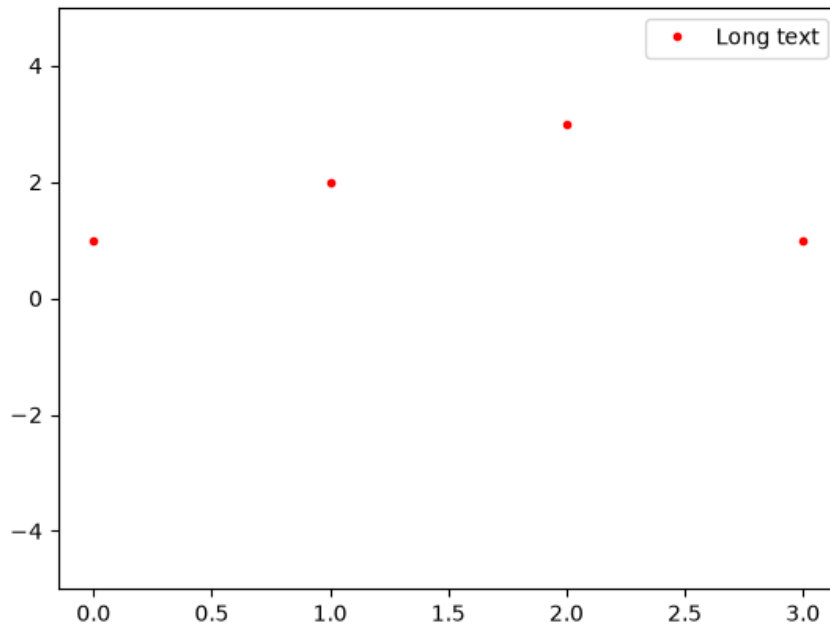
Although we did not define the `append` method, the new class inherited it from the `list` class:

```
my_list_obj.append(1)  
my_list_obj
```

```
[1, 2, 3, 1]
```

It also has a method for plotting:

```
my_list_obj.draw()
```



4.2 Symbolic computation with SymPy

The term **symbolic computation** means that we solve mathematical expressions by machine in the form of abstract symbols (and **not numerically**). Machine symbolic computation helps us when we are interested in the **result in symbolic form** and the expressions are too large to solve by hand on paper. We also resort to machine solving to reduce the chance of error (in extensive computations people can make mistakes).

Symbolic computation is by no means a substitute for numerical methods!

We will look at some of the basics; some recommended additional resources are:

- J.R. Johansson [Scientific python lectures](#),
- [SymPy](#) - the official documentation of the module,
- An excellent article by some of the SymPy authors: [Meurer et. al, 2017](#).

SymPy is one of the computer algebra systems (CAS), which, in addition to its capabilities, has the advantage of being written entirely in Python (the alternative Python package [Sage](#), for example, is not written entirely in Python).

Some dedicated commercial tools:

- [Mathematica](#)
- [Maple](#)

First we import the SymPy module; typically the package is imported as `sym`:

```
| import sympy as sym
```

You may notice that SymPy is also imported with `from sympy import *`. As a rule, we avoid this, since it unnecessarily fills the program's base namespace with SymPy names. In the latter case, the package's functions (e.g., `sympy.Sum`) are accessed directly (e.g., `Sum`), which can be appealing but starts to bother us when we add other packages (e.g., `numpy`), which, in addition to confusion, can lead to functions with the same name being “overwritten”.

In order to get nicely formatted LaTeX output, we use:

```
| sym.init_printing()
```

4.2.1 Defining variables and numerical computation

We define variables like this:

```
| x, y, k = sym.symbols('x, y, k')
```

We can check the type:

```
| type(x)
| sympy.core.symbol.Symbol
```

We notice that the variable `x` is now a `Symbol` from the `sympy` package.

Now we can make a simple computation:

```
| x**y + k
```

$$k + x^y \quad (4.1)$$

We can also name a function; here is an example where we use the sine function and the constant π :

```
| f = sym.sin(1.2*sym.pi + x)**2
| f
```

$$\sin^2(x + 1.2\pi) \quad (4.2)$$

The reader may be wondering why we need a *new* function `sympy.sin()`, since we already have the one from the `numpy` package! The reason is that symbolic computation requires a completely different treatment than numerical computation, and therefore the code behind it is entirely different.

If we want to write an equation, i.e., to equate one expression to another, we do it like this:

```
| equation = sym.Eq(sym.sin(k*x), 0.5)
| equation
```

$$\sin(kx) = 0.5 \quad (4.3)$$

Undefined mathematical functions are written as ([documentation](#)):

```
| g = sym.Function('g')
```

Now we can, for example, define a differential equation:

```
| sym.Eq(g(x).diff(x), x)
```

$$\frac{d}{dx}g(x) = x \quad (4.4)$$

When defining variables, we can add assumptions:

```
| x = sym.Symbol('x', positive=True)
| x.is_positive
| True
```

The assumptions are then taken into account in the computation. In general, we know that $\sqrt{(x^2)} \neq x$, but if x is positive, then $\sqrt{(x^2)} = x$, and `sympy`, given the assumptions, computes the correct result:

```
| sym.sqrt(x**2)
```

$$x \tag{4.5}$$

With the `assumptions0` method we look at an object's assumptions:

```
| x.assumptions0
| {'commutative': True,
|  'complex': True,
|  'extended_negative': False,
|  'extended_nonnegative': True,
|  'extended_nonpositive': False,
|  'extended_nonzero': True,
|  'extended_positive': True,
|  'extended_real': True,
|  'finite': True,
|  'hermitian': True,
|  'imaginary': False,
|  'infinite': False,
|  'negative': False,
|  'nonnegative': True,
|  'nonpositive': False,
|  'nonzero': True,
|  'positive': True,
|  'real': True,
|  'zero': False}
```

SymPy knows the types of numbers ([documentation](#)):

- `Float` real numbers (floating point, arbitrary precision),
- `Rational` rational numbers,
- `Integer` integers,
- complex numbers are written with the imaginary unit `I` (e.g., `2 + 3*sym.I`).

These types are important, as they can affect the way of solving and the solution.

Rational numbers

Above we have already defined *real numbers*. Let's now look at *rational numbers* with an example:

```
| r1 = sym.Rational(4, 5)
| r2 = sym.Rational(5, 4)
```

```
| r1
```

$$\frac{4}{5} \tag{4.6}$$

Some computations:

```
| r1+r2
```

$$\frac{41}{20} \tag{4.7}$$

```
| r1/r2
```

$$\frac{16}{25} \tag{4.8}$$

Complex numbers

The imaginary number is written with `I` (this is different from `numpy`, where the imaginary part is defined with `j`):

```
| 1+1*sym.I
```

$$1 + i \tag{4.9}$$

```
| sym.I**2
```

$$-1 \tag{4.10}$$

Numerical computation

In symbolic computation we first solve analytical expressions, simplify them, etc., and then we often also want to compute a concrete result.

Let's look at an example:

```
| x = sym.symbols('x')
| f = sym.exp(x**x**x + sym.pi)
| f
```

$$e^{x^{x^x} + \pi} \tag{4.11}$$

If we now want to use a value instead of x , e.g., `0.5`, we do it with the `subs()` method ([documentation](#)):

```
| f.subs(x, 0.5)
```

$$1.84512555475729e^\pi \tag{4.12}$$

Above we used the constant π ([documentation](#)); some commonly used constants are:

- `sympy.pi` for the number π ,
- `sympy.E` for the natural number e ,
- `sympy.oo` for infinity.

As we saw above, `subs()` performs the substitution and then the simplifications that are obvious; it did not compute the number π as a real (decimal) number. We must explicitly request this with the method:

- `evalf` (evaluate function, [documentation](#)) or
- `N`,

both of which have the argument `n` (number of digits).

Let's look at an example:

```
| f.subs(x, 2).evalf(n=80)
```

$$205630752.2559788837336514598572458946969835445645441306426707100142289480369288 \tag{4.13}$$

It is similar with `N`:

```
| sym.N(f.subs(x, 2), n=80)
```

205630752.2559788837336514598572458946969835445645441306426707100142289480369288 (4.14)

In passing, we have shown that, provided the expression contains no floating-point numbers, we can display the result to arbitrary precision ([documentation](#)).

In the `subs` function we can also use a dictionary. Example:

```
| x, y = sym.symbols('x, y')
| parameters = {x: 4, y: 10}
|
| (x + y).subs(parameters)
```

14 (4.15)

or a list of tuples:

```
| (x + y).subs([(x, 4), (y, 10)])
```

14 (4.16)

Similarly, the `sympy.evalf()` method has a `subs` argument that accepts a dictionary of substitutions, for example:

```
| (y**x).evalf(subs=parameters)
```

10000.0 (4.17)

The `sympy.subs` method can also replace a symbol (or expression) with another expression:

```
| (x + y).subs(x, y + sym.oo)
```

$2y + \infty$ (4.18)

4.2.2 SymPy and NumPy

We often connect `sympy` with `numpy`. As an example, let's look at how we would compute the expression:

```
| x = sym.symbols('x')
| f = sym.sin(x) + sym.exp(x)
| f
```

$e^x + \sin(x)$ (4.19)

efficiently and numerically at a thousand values of x .

First we import the `numpy` package:

```
| import numpy as np
```

We prepare a numerical array of values:

```
| x_vec = np.linspace(0, 10, 1000, endpoint=False)
| x_vec[:10]
```

```
| array([0.    , 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09])
```

Based on what is written above and our prior knowledge, we use a list comprehension:

```
| y_vec = np.array([f.evalf(subs={x: value}) for value in x_vec])
```

We notice that this is slow, so let's measure the time required:

```
| %%timeit -n1
| y_vec = np.array([f.evalf(subs={x: value}) for value in x_vec])

| 88.3 ms \pm 6.5 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)
```

Using the `lambdify` function

A significantly faster approach is to use the `lambdify` approach, where a compiled function is prepared, optimized for numerical execution. The syntax of the `sympy.lambdify()` function is ([documentation](#)):

```
| sympy.lambdify(symbols, function, modules=None)
```

where the arguments are:

- `symbols` the symbols used in `function` that are replaced with numerical values,
- `function` represents a `sympy` function,
- `modules` represents which package the compiled form is prepared for. If `numpy` is installed, it is the default for that module.

Example of use:

```
| f_fast = sym.lambdify(x, f, modules='numpy')
| y_vec_fast = f_fast(x_vec)
```

Let's check the speed:

```
| %%timeit -n100
| f_fast(x_vec)

| 29.8 \mus \pm 2.33 \mus per loop (mean \pm std. dev. of 7 runs, 100 loops each)
```

We observe roughly a 10,000-fold speedup!

Let's also look at an example of using a function of several variables:

```
| f_fast2 = sym.lambdify((x, y), (x + y + sym.pi)**2, 'numpy')
| x = np.linspace(0, 10, 10)
| y = x
| f_fast2(x, y)

| array([ 9.8696044 , 28.77051002, 57.54795885, 96.20195089,
|        144.73248614, 203.1395646 , 271.42318627, 349.58335115,
|        437.62005924, 535.53331054])
```

4.2.3 Graphical display

SymPy has data plotting based on `matplotlib`. The plotting is more limited compared to `matplotlib` and we use it for simple plots. ([documentation](#)).

We will look at simple examples related to the `sympy.plotting.plot` function; first we import the function:

```
| from sympy.plotting import plot
```

The syntax for using the `sympy.plotting.plot()` function ([documentation](#)) is:

```
| plot(expression, range, **kwargs)
```

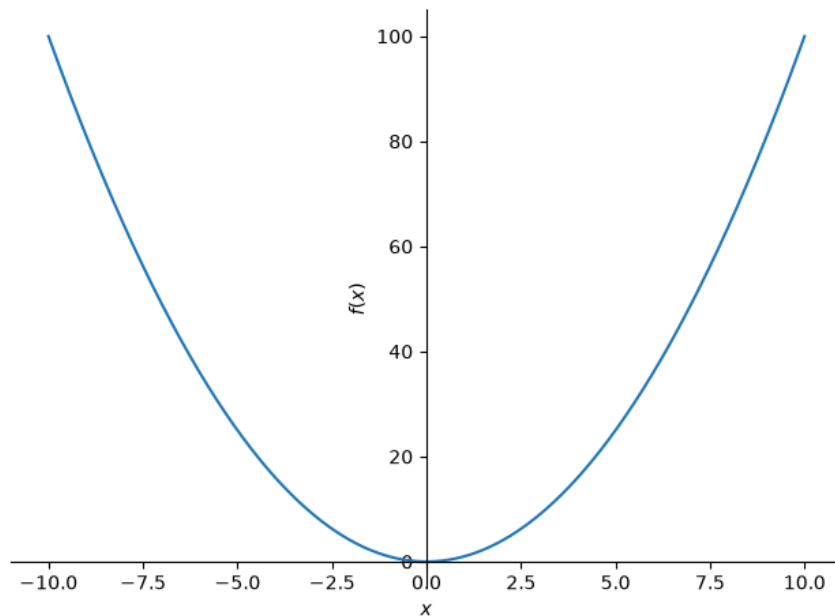
where the arguments are:

- **expression** is a mathematical expression or several expressions,
- **range** is the range of the plot (the default range is $(-10, 10)$),
- ****kwargs** are keyword arguments, i.e., a dictionary of various options.

The function returns an instance of the `sympy.Plot()` object.

A minimal example of a single function:

```
| x = sym.symbols('x')
| plot(x**2)
```



```
| <sympy.plotting.backends.matplotlibbackend.matplotlib.MatplotlibBackend at
| 0x7f8fed169b20>
```

Note: the last line points to the result in the form of a `Plot` instance; we hide the output of the object `<sympy.plotting.plot.Plot at 0x...>` by using a semicolon:

```
| plot(x**2);
```

but the figure is still displayed.

Some common arguments are:

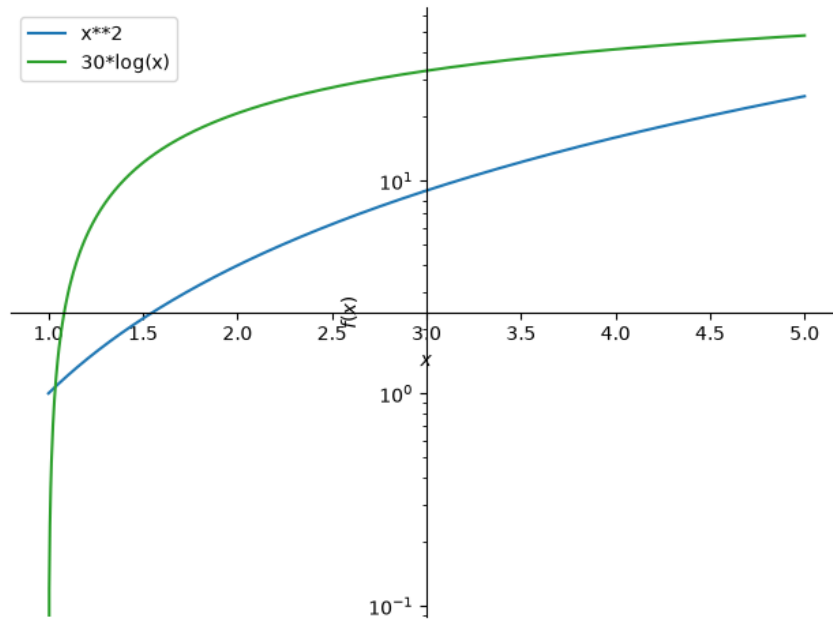
- **show** displays the figure (default `True`),
- **line_color** the color of the plot,
- **xscale** and **yscale** the display mode (options: `linear` or `log`),
- **xlim** and **ylim** the display limits for the axes (a tuple of two (`min`, `max`) values).

Let's prepare two figures where the y axis is logarithmic and the plot range is from 1 to 5:

```
| plot1 = plot(x**2, (x, 1, 5), show=False, legend=True, yscale='log')
| plot2 = plot(30*sym.log(x), (x, 1, 5), show=False, line_color='C2', legend=True,
| yscale='log')
```

Now we extend the first figure with the second and display the result:

```
plot1.extend(plot2)
plot1.show()
```



Parametric plot

Similarly, we use the function `sympy.plotting.plot_parametric` for a parametric plot ([documentation](#)):

```
plot_parametric(expr_x, expr_y, range, **kwargs)
```

where the new arguments are:

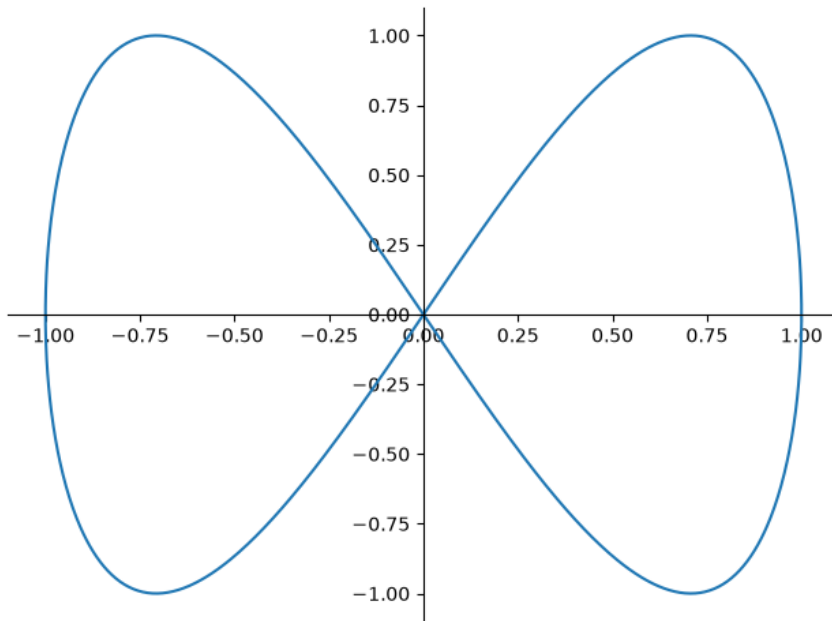
- `expr_x` and `expr_y` the definitions of the position of coordinates x and y ,
- `**kwargs` a dictionary of options.

We import the function:

```
from sympy.plotting import plot_parametric
```

Let's show its use with an example:

```
plot_parametric(sym.sin(x), sym.sin(2*x), (x, 0, 2*sym.pi));
```



Plotting in space

The function `sympy.plotting.plot3D` ([documentation](#)) for plotting in space has the syntax:

```
| plot3d(expression, range_x, range_y, **kwargs)
```

where the arguments are:

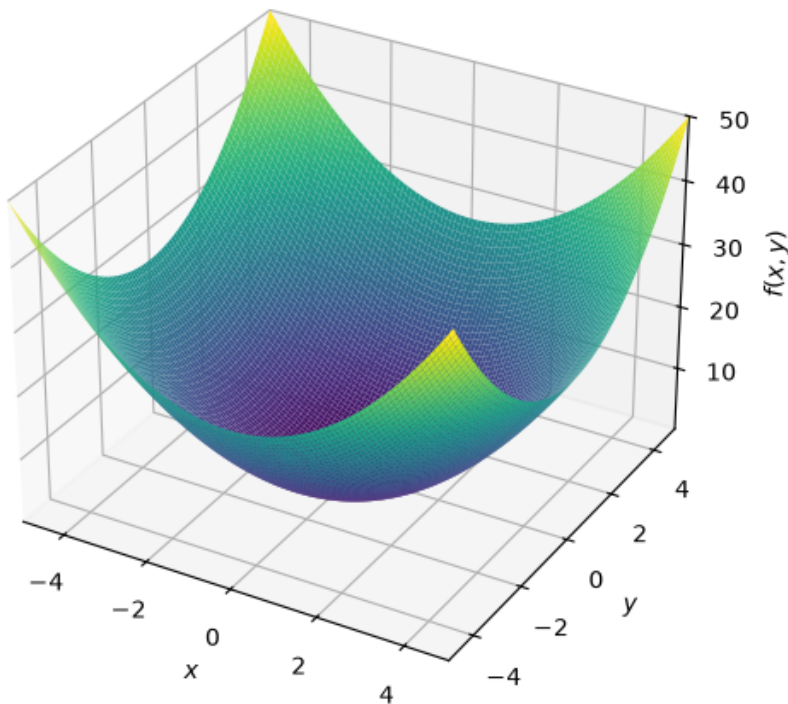
- `expression` the definition of the surface,
- `range_x` and `range_y` the range of coordinate `x` and `y`,
- `**kwargs` a dictionary of options.

We import the function:

```
| from sympy.plotting import plot3d
```

Let's show its use with an example:

```
| x, y = sym.symbols('x y')
| plot3d(x**2 + y**2, (x, -5, 5), (y, -5, 5));
```



For other plots, see the [documentation](#).

4.2.4 Algebra

In this section we will look at some of the basics of using SymPy for algebraic operations.

Using `expand` and `factor`

Let's define a mathematical expression:

```
x = sym.symbols('x')
f = (x+1)*(x+2)*(x+3)
f
```

$$(x + 1)(x + 2)(x + 3) \quad (4.20)$$

and now **expand** it (see the [documentation](#)):

```
aa = sym.expand(f)
aa
```

$$x^3 + 6x^2 + 11x + 6 \quad (4.21)$$

If we want to look at the coefficients in front of `x`, we do it with the `coeff()` method:

```
aa.coeff(x)
```

$$11 \quad (4.22)$$

The arguments of the function define what kind of expansion we want ([documentation](#)). If we want, e.g., a trigonometric expansion, then we use `trig=True`:

```
| a, b = sym.symbols('a, b')
| sym.expand(sym.sin(a+b))
```

$$\sin(a + b) \quad (4.23)$$

```
| sym.expand(sym.sin(a+b), trig=True)
```

$$\sin(a) \cos(b) + \sin(b) \cos(a) \quad (4.24)$$

The inverse operation of expansion is **factorization** ([documentation](#)):

```
| sym.factor(x**3 + 6 * x**2 + 11*x + 6)
```

$$(x + 1)(x + 2)(x + 3) \quad (4.25)$$

If we are interested in the individual factors, then we do this with the `sympy.factor_list` function:

```
| sym.factor_list(x**3 + 6 * x**2 + 11*x + 6)
```

$$(1, [(x + 1, 1), (x + 2, 1), (x + 3, 1)]) \quad (4.26)$$

Simplifying expressions with `simplify`

The `sympy.simplify()` function ([documentation](#)) tries to simplify expressions into simpler ones (e.g., by canceling variables).

For special purposes we can also simplify with:

- `sympy.trigsimp`,
- `sympy.powsimp`,
- `sympy.logcombine`.

For more, see the [documentation](#).

Examples of simplification:

```
| sym.simplify((x+1)*(x+1)*(x+3))
```

$$(x + 1)^2 (x + 3) \quad (4.27)$$

```
| sym.simplify(sym.sin(a)**2 + sym.cos(a)**2)
```

$$1 \quad (4.28)$$

```
| sym.simplify(sym.cos(x)/sym.sin(x))
```

$$\frac{1}{\tan(x)} \quad (4.29)$$

4.2.5 Using apart and together

We use these functions to work with fractions:

```
| f1 = 1/((1 + x) * (5 + x))
| f1
```

$$\frac{1}{(x+1)(x+5)} \quad (4.30)$$

We perform partial fraction decomposition with the `sympy.apart()` function ([documentation](#)):

```
| f2 = sym.apart(f1, x)
| f2
```

$$-\frac{1}{4(x+5)} + \frac{1}{4(x+1)} \quad (4.31)$$

and then back again in the opposite direction with the `sympy.together()` function:

```
| sym.together(f2)
```

$$\frac{1}{(x+1)(x+5)} \quad (4.32)$$

In the latter case we arrive at a similar result with `sympy.simplify()`:

```
| sym.simplify(f2)
```

$$\frac{1}{(x+1)(x+5)} \quad (4.33)$$

4.2.6 Differentiation

Differentiation is in principle a relatively simple mathematical operation, which we perform with the `sympy.diff()` function ([documentation](#)):

Let's prepare an example:

```
| x, y, z = sym.symbols('x, y, z')
| f = sym.sin(x*y) + sym.cos(y*z)
| f
```

$$\sin(xy) + \cos(yz) \quad (4.34)$$

Let's differentiate it with respect to x :

```
| sym.diff(f, x)
```

$$y \cos(yz) \quad (4.35)$$

or also

```
| f.diff(x)
```

$$y \cos(xy) \quad (4.36)$$

Higher-order derivatives are defined like this:

```
| sym.diff(f, x, x, x)
```

$$-y^3 \cos(xy) \quad (4.37)$$

or (the same result a bit differently):

```
| sym.diff(f, x, 3)
```

$$-y^3 \cos(xy) \quad (4.38)$$

The derivative with respect to several variables $\frac{d^3 f}{dx dy^2}$ is performed like this:

```
| sym.diff(f, x, 1, y, 2)
```

$$-x(xy \cos(xy) + 2 \sin(xy)) \quad (4.39)$$

4.2.7 Integration

The `integrate` function can be used for indefinite integration ([documentation](#)):

```
| integrate(f, x)
```

or for definite integration:

```
| integrate(f, (x, a, b))
```

where the arguments are:

- `f` the function we integrate,
- `x` the variable with respect to which we integrate,
- `a` and `b` the limits of integration.

An example of indefinite integration:

```
| x = sym.symbols('x')
| f = sym.sin(x*y) + sym.cos(y*z)
| sym.integrate(f, x)
```

$$x \cos(yz) + \begin{cases} -\frac{\cos(xy)}{y} & \text{for } y \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (4.40)$$

We notice that `sympy` correctly takes into account the possibility that $y = 0$.

Another example of definite integration:

```
| sym.integrate(f, (x, -1, 1))
```

$$2 \cos(yz) \quad (4.41)$$

An example where the limits are at infinity (we use the constant for infinity `sympy.oo`):

```
| sym.integrate(sym.exp(-x**2), (x, -sym.oo, sym.oo))
```

$$\sqrt{\pi} \quad (4.42)$$

4.2.8 Sum and product of a series

We define the sum of a series with the `sympy.Sum()` function ([documentation](#)):

```
| sympy.Sum(expression, (var, start, end))
```

where the arguments are:

- **expression** the expression we are summing,
- **var, start** and **end** the variable that increases from **start** to **end** (**end** is included).

An example of the sum of a series:

```
| n = sym.Symbol('n')
| f = sym.Sum(1/x**n, (n, 1, sym.oo))
| f
```

$$\sum_{n=1}^{\infty} x^{-n} \quad (4.43)$$

Only when we use the `doit()` method is the computation carried out:

```
| f.doit()
```

$$\begin{cases} \frac{1}{x(1-\frac{1}{x})} & \text{for } \frac{1}{|x|} < 1 \\ \sum_{n=1}^{\infty} x^{-n} & \text{otherwise} \end{cases} \quad (4.44)$$

Let's also look at the numerical result:

```
| f.subs({x: 5}).evalf()
```

$$0.25 \quad (4.45)$$

We define the product of a series similarly with the `sympy.Product` function ([documentation](#)):

```
| sympy.Product(expression, (var, start, end))
```

where the arguments are:

- **expression** the expression we are multiplying,
- **var, start** and **end** the variable that increases from **start** to **end** (**end** is included).

Example:

```
| f = sym.Product(1/n, (n, 1, 5))
| f
```

$$\prod_{n=1}^5 \frac{1}{n} \quad (4.46)$$

```
| f.doit()
```

$$\frac{1}{120} \quad (4.47)$$

4.2.9 Limits

We compute limits with the `sympy.limit()` function ([documentation](#)):

```
| sympy.limit(f, x, x0)
```

where the arguments are:

- `f` the expression whose limit we are seeking,
- `x` the variable that approaches `x0`,
- `x0` the limit point.

Example:

```
| x = sym.symbols('x')
| f = sym.sin(x)/x
| f
```

$$\frac{\sin(x)}{x} \quad (4.48)$$

```
| sym.limit(f, x, 0)
```

$$1 \quad (4.49)$$

As an example, let's look at the use of the limit in the definition of the derivative:

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h, y) - f(x, y)}{h}. \quad (4.50)$$

Let's prepare the function `f` and its derivative:

```
| x, y, z, h = sym.symbols('x, y, z, h')
| f = sym.sin(x*y) + sym.cos(y*z)
```

The derivative of the function is:

```
| sym.diff(f, x)
```

$$y \cos(xy) \quad (4.51)$$

We compute the same result also using the limit:

```
| sym.limit((f.subs(x, x+h) - f)/h, h, 0)
```

$$y \cos(xy) \quad (4.52)$$

4.2.10 Taylor series

We compute Taylor series with the `sympy.series()` function ([documentation](#)):

```
| sympy.series(expression, x=None, x0=0, n=6, dir='+')
```

where the arguments are:

- `expression` the expression whose series we are determining,
- `x` the independent variable,
- `x0` the value around which we determine the series (default 0),

- **n** the order of the series (default 6),
- **dir** the direction of the series expansion (+ or -).

Example:

```
x = sym.symbols('x')
sym.series(sym.exp(x), x) # default values x0=0, and n=6
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + O(x^6) \quad (4.53)$$

If we want to define a different expansion point (**x0=2**) and use more terms (**n=8**), we do it like this:

```
s1 = sym.series(sym.exp(x), x, x0=2, n=8)
s1
```

$$e^2 + (x-2)e^2 + \frac{(x-2)^2 e^2}{2} + \frac{(x-2)^3 e^2}{6} + \frac{(x-2)^4 e^2}{24} + \frac{(x-2)^5 e^2}{120} + \frac{(x-2)^6 e^2}{720} + \frac{(x-2)^7 e^2}{5040} + O((x-2)^8; x \rightarrow 2) \quad (4.54)$$

The result also includes the order of validity; in this way we can control the validity of the computation ($\mathcal{O}$).

Example:

```
s1 = sym.cos(x).series(x, 0, 5)
s1
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} + O(x^5) \quad (4.55)$$

```
s2 = sym.sin(x).series(x, 0, 2)
s2
```

$$x + O(x^2) \quad (4.56)$$

The computations **s1** and **s2** have different orders of validity, so the product:

```
s1 * s2
```

$$(x + O(x^2)) \left(1 - \frac{x^2}{2} + \frac{x^4}{24} + O(x^5)\right) \quad (4.57)$$

is accurate only up to order $\mathcal{O}(x^2)$, which **sympy** handles appropriately:

```
s3 = sym.simplify(s1 * s2)
s3
```

$$x + O(x^2) \quad (4.58)$$

The information about the order of validity can be removed:

```
s3.removeO()
```

$$x \quad (4.59)$$

4.2.11 Linear algebra

Matrices and vectors

We define matrices and vectors with the `Matrix` function. While with `numpy.array` it is not necessary to strictly adhere to the mathematical notation of vectors and matrices, with `sympy` it is required.

Let's look at an example; first we prepare the variables:

```
| m11, m12, m21, m22 = sym.symbols('m11, m12, m21, m22')
| b1, b2 = sym.symbols('b1, b2')
```

Then the matrix and the column vector:

```
| A = sym.Matrix([[m11, m12], [m21, m22]])
| A
```

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \quad (4.60)$$

```
| b = sym.Matrix([[b1], [b2]])
| b
```

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad (4.61)$$

Now let's look at some typical operations; first the multiplication of a matrix and a vector:

```
| A * b
```

$$\begin{bmatrix} b_1 m_{11} + b_2 m_{12} \\ b_1 m_{21} + b_2 m_{22} \end{bmatrix} \quad (4.62)$$

Then the dot product of two vectors (we must be careful to transpose one of the vectors):

```
| b.T*b
```

$$[b_1^2 + b_2^2] \quad (4.63)$$

The determinant and the inverse matrix:

```
| A.det()
```

$$m_{11}m_{22} - m_{12}m_{21} \quad (4.64)$$

```
| A.inv()
```

$$\begin{bmatrix} \frac{m_{22}}{m_{11}m_{22}-m_{12}m_{21}} & -\frac{m_{12}}{m_{11}m_{22}-m_{12}m_{21}} \\ -\frac{m_{21}}{m_{11}m_{22}-m_{12}m_{21}} & \frac{m_{11}}{m_{11}m_{22}-m_{12}m_{21}} \end{bmatrix} \quad (4.65)$$

Multiplication and power of a matrix:

```
| A*A
```

$$\begin{bmatrix} m_{11}^2 + m_{12}m_{21} & m_{11}m_{12} + m_{12}m_{22} \\ m_{11}m_{21} + m_{21}m_{22} & m_{12}m_{21} + m_{22}^2 \end{bmatrix} \quad (4.66)$$

```
| A**2
```

$$\begin{bmatrix} m_{11}^2 + m_{12}m_{21} & m_{11}m_{12} + m_{12}m_{22} \\ m_{11}m_{21} + m_{21}m_{22} & m_{12}m_{21} + m_{22}^2 \end{bmatrix} \quad (4.67)$$

4.2.12 Root finding (solving equations)

We solve equations and systems of equations with the `sympy.solve()` function ([documentation](#)). Solving the following equations is supported:

- polynomial equations,
- transcendental equations,
- piecewise-defined equations as a combination of the above two types,
- systems of linear and polynomial equations,
- systems of equations with inequalities.

The syntax is:

```
| sympy.solve(f, *symbols, **flags)
```

where the arguments are:

- `f` an expression or a list of expressions,
- `*symbols` the symbol or list of symbols we want to determine,
- `**flags` a dictionary of options.

Let's look at an example:

```
| x = sym.symbols('x')
| f = sym.sin(x)
| eq = sym.Eq(f, 1/2)
| eq
```

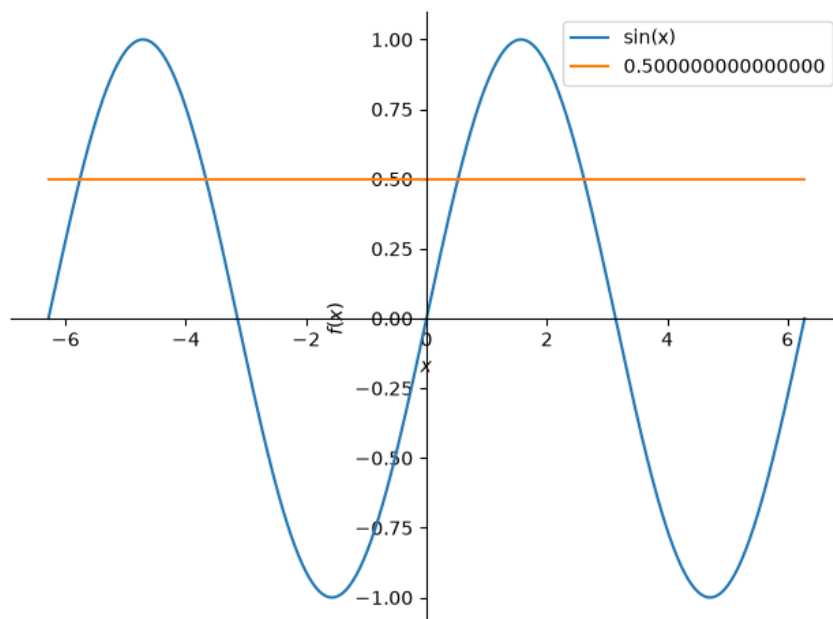
$$\sin(x) = 0.5 \quad (4.68)$$

```
| sympy.solve(eq, x)
```

$$[0.523598775598299, 2.61799387799149] \quad (4.69)$$

Let's plot the solution (notice that we found only two of the infinitely many solutions):

```
| p1 = sym.plotting.plot(sym.sin(x), (x, -2*sym.pi, 2*sym.pi), line_color='C0',
| show=False, legend=True)
| p2 = sym.plotting.plot(0.5, (x, -2*sym.pi, 2*sym.pi), line_color='C1', show=False,
| legend=True)
| p1.extend(p2)
| p1.show()
```



A quadratic equation:

```
a, b, c, x = sym.symbols('a, b, c, x')
sym.solve(a*x**2 + b*x + c, x)
```

$$\left[\frac{-b - \sqrt{-4ac + b^2}}{2a}, \frac{-b + \sqrt{-4ac + b^2}}{2a} \right] \quad (4.70)$$

A system of equations:

```
x, y = sym.symbols('x y')
sym.solve([x + y - 1, x - y - 1], [x, y])
```

$$\{x : 1, y : 0\} \quad (4.71)$$

For nonlinear systems we can also use numerical solving with the `sympy.nsolve()` function ([documentation](#)):

```
sympy.nsolve(f, [args,] x0, modules=['mpmath'], **kwargs)
```

where the arguments are:

- `f` the equation or system of equations we are solving,
- `args` the variables (optional),
- `x0` the initial estimate (a scalar or a vector),
- `modules` the package used to compute the numerical value (the same logic as with the `lambdify` function; the default is the `mpmath` package),
- `**kwargs` a dictionary of options.

Let's look at the example from above:

```
x = sym.symbols('x')
eq = sym.Eq(sym.sin(x), 0.5)
sol = sym.nsolve(eq, x, 3)
sol
```

$$2.61799387799149 \quad (4.72)$$

4.2.13 Solving differential equations

Undefined functions

Before we look at differential equations, we must get to know undefined functions. We define such functions with `sympy.Function` [documentation](#).

As an example, let's look at how we define the equation $\ddot{x}(t) = g$. First let's define the undefined function:

```
x = sym.Function('x')
```

Then the symbols:

```
t, g = sym.symbols('t, g')
```

And the equation:

```
sym.Eq(x(t).diff(t,2), g)
```

$$\frac{d^2}{dt^2}x(t) = g \quad (4.73)$$

Differential equations

We solve differential equations and systems of differential equations with the `sympy.dsolve()` function ([documentation](#)):

```
sympy.dsolve(eq, func=None, hint='default', simplify=True,
            ics=None, xi=None, eta=None, x0=0, n=6, **kwargs)
```

where the selected arguments are:

- **eq** the differential equation or system of differential equations,
- **func** the solution we are seeking,
- **ics** the initial and boundary conditions of the differential equation.

Let's look at the example of a mass m sliding on a surface with friction coefficient μ ; the initial velocity is v_0 , and the displacement $x_0 = 0$.

Let's define the symbols:

```
x = sym.Function('x')
t, m, mu, g, v0, x0 = sym.symbols('t m mu g v0 x0', real=True, positive=True)
```

Let's define the differential equation:

```
eq = sym.Eq(m*x(t).diff(t,2), -mu*g*m)
eq
```

$$m \frac{d^2}{dt^2} x(t) = -gm\mu \quad (4.74)$$

Let's look at the properties of the differential equation:

```
sym.ode_order(eq, x(t))
```

$$2 \tag{4.75}$$

```
sym.classify_ode(eq)
```

```
('factorable',
'nth_algebraic',
'nth_linear_constant_coeff_undetermined_coefficients',
'nth_linear_euler_eq_nonhomogeneous_undetermined_coefficients',
'nth_linear_constant_coeff_variation_of_parameters',
'nth_linear_euler_eq_nonhomogeneous_variation_of_parameters',
'nth_algebraic_Integral',
'nth_linear_constant_coeff_variation_of_parameters_Integral',
'nth_linear_euler_eq_nonhomogeneous_variation_of_parameters_Integral',
'2nd_nonlinear_autonomous_conserved',
'2nd_nonlinear autonomous conserved Integral')
```

Let's solve it:

```
solution = sym.dsolve(eq, x(t),
                      ics={
                          x(0):0,           #at time 0s the displacement is
                          zero              #at time 0s the velocity equals
                          x(t).diff(t).subs(t, 0): v0
                      })
solution
```

$$x(t) = -\frac{g\mu t^2}{2} + tv_0 \quad (4.76)$$

We retrieve the right-hand side of the equation like this (**rhs** - right-hand side):

```
| solution.rhs
```

$$-\frac{g\mu t^2}{2} + tv_0 \quad (4.77)$$

4.3 Additional

4.4 `sympy.mechanics`

`sympy` has built-in support for classical mechanics ([documentation](#)). A comprehensive tutorial was presented at the scientific conference [SciPy 2016](#).

Chapter 5

Introduction to numerical methods and systems of linear equations

5.1 Introduction to numerical methods

When we want to simulate a chosen physical process, we usually proceed as follows:

1. we set up a mathematical model*,
2. we choose a numerical method and its parameters,
3. we write a program (using built-in functions to help us),
4. we run the computation, then analyze and evaluate the results.

* If the mathematical model can be solved analytically, numerical solution is not needed.

We try to solve a mathematical model analytically, since such a solution is not burdened with errors. For this reason, in mathematics we learn to solve systems of equations, to integrate, to differentiate, and so on. The essence of **numerical methods** is that we solve mathematical models **numerically**, that is, on the basis of **discrete values**. As we will learn later, the numerical approach, compared to the analytical one, allows us to solve substantially larger and more complex problems.

5.1.1 Round-off error

In what follows we will look at some of the limitations and challenges of the numerical approach. The first limitation is that in a computer real values are always stored with finite precision. In Python, numbers are often stored in double precision with approximately 15 significant digits.

A double-precision number in Python is called **float64** and is stored in memory in binary form using 64 bits (1 bit for the sign, 11 bits for the exponent and 52 bits for the mantissa). Since the mantissa is defined using 52 binary digits, its representation can incur a *relative error* of at most $\epsilon \approx 2.2 \cdot 10^{-16}$. This error is called the **machine round-off error** and can occur at every intermediate computation!

If there are many steps, the error can grow very large, and therefore it is important that its effect on the result is as small as possible!

Below is an example of more detailed information for the double-precision data type (**float**); for this we use the built-in module **sys** to call parameters and functions of the Python system ([documentation](#)):

```
import sys
sys.float_info.epsilon
1e16+2
#sys.float_info #also check the broader output!

1.0000000000000002e+16
```

In addition to the double-precision number, other data types are used; a good overview of the various types is given in the [numpy](#) and [python](#) documentation.

Here let us look at an example of the `int8` type, which means an integer stored with 8 bits (8 bits = 1 byte). With it we can represent integers from -128 to +127 in the binary system:

```
import numpy as np
number = np.int8(1) # also try the numbers: -128 and then 127, 128, 129. What
happens?
f'Number {number} of type {type(number)} stored in binary form:{number:8b}'

"Number 1 of type <class 'numpy.int8'> stored in binary form:      1"
```

5.1.2 Method error

In addition to the round-off error, we often also encounter the **method error**, which we make when we replace the exact analytical procedure for solving a mathematical model with an approximate numerical one.

An important property of numerical algorithms is **stability**. This means that a small change in the input data causes a small change in the results. If the results change a great deal for a small change at the input, we say that the **algorithm is unstable**. In practice, therefore, we use stable algorithms; but we will see later that stability can also be conditioned by the input data!

We also know of the instability of the mathematical model/problem/equation; in that case we speak of **ill-conditioning**.

During the execution of a numerical computation, errors can propagate. As a consequence, the result of an operation is less accurate (it has fewer reliable digits) than the reliability of the computation's data.

Let us now look at a general approach to estimating the error. We denote the exact value by r and the approximation by a_1 ; it holds that $r = a_1 + e_1$, where e_1 is the error. If with a numerical algorithm we compute a substantially better approximation a_2 , then $r = a_2 + e_2$.

Since $a_1 + e_1 = a_2 + e_2$, then under the assumption $|e_1| \gg |e_2|$ and $|e_2| \approx 0$ we can derive $a_2 - a_1 = e_1 - e_2 \approx e_1$.

$|a_1 - a_2|$ is therefore a pessimistic estimate of the absolute error,

while $\left| \frac{a_1 - a_2}{a_2} \right|$ is an estimate of the relative error.

5.2 Introduction to systems of linear equations

Under the heading above we mean a system of m linear equations ($E_i, i = 0, 1, \dots, m-1$) with n unknowns ($x_j, j = 0, 1, \dots, n-1$):

$$\begin{array}{rclclcl} E_0 : & A_{0,0} x_0 & + & A_{0,1} x_1 & + & \dots & + & A_{0,n-1} x_{n-1} & = & b_0 \\ E_1 : & A_{1,0} x_0 & + & A_{1,1} x_1 & + & \dots & + & A_{1,n-1} x_{n-1} & = & b_1 \\ & \vdots & & & & & & \vdots & & \\ E_{m-1} : & A_{m-1,0} x_0 & + & A_{m-1,1} x_1 & + & \dots & + & A_{m-1,n-1} x_{n-1} & = & b_{m-1}. \end{array} \quad (5.1)$$

The coefficients $A_{i,j}$ and b_i are known numbers.

If the right-hand side equals zero, that is $b_i = 0$, we call the system **homogeneous**, otherwise the system is **nonhomogeneous**.

A system of equations can also be written in matrix form:

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (5.2)$$

where $\mathbf{A}$ and $\mathbf{b}$ are a known matrix and vector, while the vector $\mathbf{x}$ is unknown. We call the matrix $\mathbf{A}$ the **coefficient matrix**, the vector $\mathbf{b}$ the **constant vector** (also: the vector of free terms, or the right-hand-side column vector), and $\mathbf{x}$ the **vector of unknowns**. If we add the

vector $\mathbf{b}$ as a column to the matrix $\mathbf{A}$, we obtain the so-called **augmented matrix**, which we denote $[\mathbf{A}|\mathbf{b}]$.

A note on notation:

- scalar variables are written in italics, e.g.: a, A ,
- vector variables are written in lowercase bold, e.g.: $\mathbf{a}$,
- matrix variables are written in uppercase bold, e.g.: $\mathbf{A}$.

5.2.1 On the solution of a system of linear equations

```
from IPython.display import YouTubeVideo
YouTubeVideo('7YbyijGUbYw', width=800, height=300)
```



If we perform **elementary operations** on a system of linear equations:

- multiplying any equation by a constant (different from zero),
- changing the order of the equations,
- adding one equation (multiplied by a constant) to another equation,

we do not change the solution of the system and obtain an equivalent system of equations.

Using elementary operations on the rows of the matrix $\mathbf{A}$, we can transform it into the so-called **reduced row echelon form**:

1. if there are zero rows, they are at the bottom of the matrix,
2. the first nonzero element lies to the right of the first nonzero elements of the preceding rows,
3. the first nonzero element in a *row* is called the **pivot** and is equal to 1,
4. the pivot is the only nonzero element in its *column*.

The **rank of a matrix** represents the number of nonzero rows in the reduced row echelon form of the matrix; the number of nonzero rows represents the **number of linearly independent equations and equals the number of pivot elements**. The rank of a matrix is therefore equal to the number of linearly independent rows of the matrix. Transposing a matrix does not change its rank, so the rank of a matrix also equals the number of linearly independent *columns* of the matrix.

Example of transforming the matrix $\mathbf{A}$:

```
import numpy as np #import numpy
A = np.arange(9).reshape((3,3))+1
A
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Clearly the nonzero element $A[0,0]$ has the value 1 and is the pivot element. We multiply the first row $A[0,:]$ by -4 and add the product to the second row $A[1,:]-4A[0,:]$:

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

```
array([[ 1,  2,  3],
       [ 0, -3, -6],
       [ 7,  8,  9]])
```

We do similarly for the third row:

```
A[2,:] -= A[2,0]*A[0,:]
A
```

```
array([[ 1,  2,  3],
       [ 0, -3, -6],
       [ 0, -6, -12]])
```

We now divide the second row by $A[1,1]$ to obtain the pivot:

```
A[1,:] = A[1,:]/A[1,1]
A
```

```
array([[ 1,  2,  3],
       [ 0,  1,  2],
       [ 0, -6, -12]])
```

We subtract the second row from the others, so that in the second column we get zeros everywhere except for the value 1 in the second row:

```
A[0,:] -= A[0,1]*A[1,:] # subtraction from the first row
A
```

```
array([[ 1,  0, -1],
       [ 0,  1,  2],
       [ 0, -6, -12]])
```

```
A[2,:] -= A[2,1]*A[1,:] # subtraction from the last row
A
```

```
array([[ 1,  0, -1],
       [ 0,  1,  2],
       [ 0,  0,  0]])
```

We have two nonzero rows; the matrix A has two pivots and represents two linearly independent equations. The rank of the matrix is 2.

```
A
```

```
array([[ 1,  0, -1],
       [ 0,  1,  2],
       [ 0,  0,  0]])
```

The rank of a matrix can also be determined using the `numpy` function `numpy.linalg.matrix_rank` ([documentation](#)):

```
| matrix_rank(M, tol=None)
```

where `M` is the matrix whose rank we are looking for, and `tol` is an optional parameter that sets the threshold below which values in the algorithm are treated as equal to zero.

```
| np.linalg.matrix_rank(A)
```

```
| np.int64(2)
```

If $r = \text{rank}(\mathbf{A}) = \text{rank}([\mathbf{A}|\mathbf{b}])$, then a solution **exists** (we also say that the system is **consistent**).

A consistent system has:

- exactly one solution when the number of unknowns n equals the rank r (the solution is independent), and
- infinitely many solutions when the rank r is less than the number of unknowns n (the solution depends on $n - r$ parameters).

First we will restrict ourselves to a system of $m = n$ linear equations with n unknowns where $n = r$:

$$\mathbf{A} \mathbf{x} = \mathbf{b}. \quad (5.3)$$

Under the above conditions the coefficient matrix $\mathbf{A}$ is nonsingular ($|\mathbf{A}| \neq 0$) and the system has the solution:

$$\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}. \quad (5.4)$$

Let us look at an example of a system where the **equations are linearly dependent** ($r < n$):

```
| A = np.array([[1 , 2],
|               [2, 4]])
| b = np.array([1, 2])
| Ab = np.column_stack((A,b))
| Ab
| array([[1, 2, 1],
|        [2, 4, 2]])
```

Using the `numpy` library, let us now look at the rank of the coefficient matrix and of the augmented matrix, as well as the determinant, using `numpy.linalg.det` ([documentation](#)):

```
| det(a)
```

where `a` is the matrix (or list of matrices) whose determinant we are looking for; the function `det` returns the determinant (or list of determinants).

```
| f'rank(A)={np.linalg.matrix_rank(A)}, rank(Ab)={np.linalg.matrix_rank(Ab)}, \
| number of unknowns: {A.shape[1]}, det(A)={np.linalg.det(A)}'
| 'rank(A)=1, rank(Ab)=1, number of unknowns: 2, det(A)=0.0'
```

Let us also look at an example where **there is no solution at all** (an inconsistent system):

```
| A = np.array([[1 , 2],
|               [2, 4]])
| b = np.array([1, 1])
| Ab = np.column_stack((A,b))
| Ab
| array([[1, 2, 1],
|        [2, 4, 1]])
| f'rank(A)={np.linalg.matrix_rank(A)}, rank(Ab)={np.linalg.matrix_rank(Ab)}, \
| number of unknowns: {A.shape[1]}, det(A)={np.linalg.det(A)}'
| 'rank(A)=1, rank(Ab)=2, number of unknowns: 2, det(A)=0.0'
```

5.2.2 Norm and conditioning of systems of equations

A numerical problem is ill-conditioned if a small change in the data causes a large change in the result. In the case of a *small change in the data* that causes a *small change in the results*, the problem is **well-conditioned**.

A system of equations is usually well-conditioned if the absolute values of the diagonal elements of the coefficient matrix are large compared to the absolute values of the off-diagonal elements.

For a system of linear equations $\mathbf{A} \mathbf{x} = \mathbf{b}$ we can compute the **condition number**:

$$\text{cond}(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\|. \quad (5.5)$$

By $\|\mathbf{A}\|$ we denote the **norm** of the matrix.

There are several ways to compute the norm; let us state two:

- the Euclidean norm (also the Frobenius norm):

$$\|\mathbf{A}\|_e = \sqrt{\sum_{i=1}^n \sum_{j=1}^n A_{ij}^2} \quad (5.6)$$

- the row-sum norm, also called the infinity norm:

$$\|\mathbf{A}\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^n |A_{ij}| \quad (5.7)$$

We compute the conditioning with the built-in function `numpy.linalg.cond` ([documentation](#)):

```
| cond(x, p=None)
```

which takes two parameters: the matrix `x` and the optional norm type `p` (the default type is `None`; in that case the 2-norm is used (the ratio of the largest and smallest singular values); for the Frobenius norm you must pass `p='fro'`).

If the condition number is small, then the matrix is well-conditioned, and conversely - for ill-conditioning the condition number increases greatly.

Unfortunately, computing the conditioning of a matrix is numerically relatively demanding.

Example of an ill-conditioned matrix

We will look at an ill-conditioned system, where with a tiny change to the coefficient matrix we will cause a large change in the solution.

The coefficient matrix:

```
| A = np.array([[1 , 1],
|               [1, 1.00001]])
| np.linalg.cond(A)
|
| np.float64(400002.0000028382)
```

The constant vector:

```
| b = np.array([3, -3])
| Ab = np.column_stack((A,b))
| Ab
|
| array([[ 1.      ,  1.      ,  3.      ],
|        [ 1.      , 1.00001, -3.      ]])
```

Let us check the rank and the determinant:

```
f'rank(A)={np.linalg.matrix_rank(A)}, rank(Ab)={np.linalg.matrix_rank(Ab)}, \
number of unknowns: {A.shape[1]}, det(A)={np.linalg.det(A)}'

'rank(A)=2, rank(Ab)=2, number of unknowns: 2, det(A)=1.000000000006551e -05'
```

We subtract the first equation from the second:

```
Ab[1,:] -= Ab[0,:]
Ab

array([[ 1.e+00,  1.e+00,  3.e+00],
       [ 0.e+00,  1.e -05, -6.e+00]])
```

We determine x1:

```
x1 = Ab[1,2]/Ab[1,1]
x1

np.float64( -599999.9999960692)
```

It remains to determine x0:

```
x0 = (Ab[0,2] - Ab[0,1]*x1)/Ab[0,0]
x0

np.float64(600002.9999960692)
```

We slightly change the coefficient matrix and repeat the solving:

```
A = np.array([[1 , 1],
              [1, 1.0001]]) # <= here we changed 1.00001 to 1.0001
np.linalg.cond(A)

np.float64(40002.00007491187)
```

We repeat the computation:

```
Ab = np.column_stack((A,b))
f'rank(A)={np.linalg.matrix_rank(A)}, rank(Ab)={np.linalg.matrix_rank(Ab)}, \
number of unknowns: {len(A[:,0])}, det(A)={np.linalg.det(A)}'
Ab[1,:] -= Ab[0,:]
x1_ = Ab[1,2]/Ab[1,1]
x0_ = (Ab[0,2] - Ab[0,1]*x1_)/Ab[0,0]
```

We compare the two solutions:

```
[x0, x1] # first solution

[np.float64(600002.9999960692), np.float64( -599999.9999960692)]

[x0_, x1_] # second solution

[np.float64(60003.00000000661), np.float64( -60000.00000000661)]
```

We find that a tiny change in one coefficient of the coefficient matrix caused a large change in the result. A small change in the data cannot be avoided, because of how the data is stored in the computer.

5.2.3 Numerical solution of systems of linear equations

We will look at two approaches to solving systems of linear equations that are in principle different:

A) the **direct approach**: we perform elementary operations on the system of equations, by which we transform the system into one that is easier to solve,

B) the **iterative approach**: we choose an initial approximation and then iteratively improve the approximation.

5.3 Gaussian elimination

Assume that we are solving a system of n equations for n unknowns that has rank n . Such a system has a unique solution.

Gaussian elimination is one of the direct methods, since by means of elementary row operations we reduce the system of equations to an upper-triangular system (below the main diagonal in the augmented matrix the values are zero).

First we prepare the augmented coefficient matrix:

$$[\mathbf{A}|\mathbf{b}] = \left[\begin{array}{cccc|c} A_{0,0} & A_{0,1} & \cdots & A_{0,n-1} & b_0 \\ A_{1,0} & A_{1,1} & \cdots & A_{1,n-1} & b_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{n-1,0} & A_{n-1,1} & \cdots & A_{n-1,n-1} & b_{n-1} \end{array} \right] \quad (5.8)$$

We will look at Gaussian elimination through an example:

```
A = np.array([[ 8., -6, 3],
               [-6, 6, -6],
               [ 3, -6, 6]])
b = np.array([-14, 36, 6])
Ab = np.column_stack((A,b))

Ab
array([[ 8., -6.,  3., -14.],
       [-6.,  6., -6.,  36.],
       [ 3., -6.,  6.,  6.]])
```

Step 0: we multiply the first row by $\text{Ab}[1,0]/\text{Ab}[0,0]=-6/8$ and subtract it from the second:

```
Ab[1,:] -= Ab[1,0]/Ab[0,0] * Ab[0,:]
Ab
array([[ 8. , -6. ,  3. , -14. ],
       [ 0. ,  1.5, -3.75, 25.5 ],
       [ 3. , -6. ,  6. ,  6. ]])
```

Next we multiply the first row by $\text{Ab}[2,0]/\text{Ab}[0,0]=3/8$ and subtract it from the third:

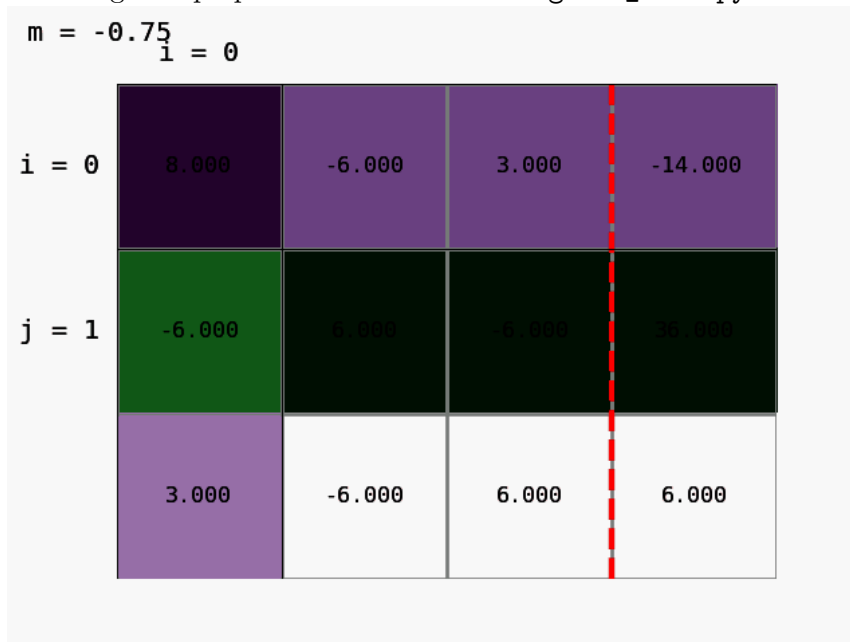
```
Ab[2,:] -= Ab[2,0]/Ab[0,0] * Ab[0,:]
Ab
array([[ 8. , -6. ,  3. , -14. ],
       [ 0. ,  1.5, -3.75, 25.5 ],
       [ 0. , -3.75, 4.875, 11.25 ]])
```

Step 1: we multiply the second row by $\text{Ab}[2,1]/\text{Ab}[1,1]=-3.75/1.5$ and subtract it from the third:

```
Ab[2,:] -= Ab[2,1]/Ab[1,1] * Ab[1,:]
Ab
```

```
| array([[ 8. , -6. ,  3. , -14. ],
|        [ 0. ,  1.5, -3.75, 25.5 ],
|        [ 0. ,  0. , -4.5 , 75. ]])
```

Using the prepared module `modules/gauss_demo.py` we can also animate the procedure:



We have obtained an upper-triangular matrix and Gaussian elimination is finished. We can compute the solution, that is, determine the vector of unknowns x by **back substitution**.

From the last row of the upper-triangular matrix we compute x_2 :

```
| Ab
|
| array([[ 8. , -6. ,  3. , -14. ],
|        [ 0. ,  1.5, -3.75, 25.5 ],
|        [ 0. ,  0. , -4.5 , 75. ]])
|
| x = np.zeros(3) #prepare an empty list
| x[2] = Ab[2, -1]/Ab[2,2]
| x
|
| array([ 0. ,  0. , -16.66666667])
```

Using the second-to-last row we compute x_1 :

```
| Ab
|
| array([[ 8. , -6. ,  3. , -14. ],
|        [ 0. ,  1.5, -3.75, 25.5 ],
|        [ 0. ,  0. , -4.5 , 75. ]])
|
| x[1] = (Ab[1, -1] - Ab[1,2]*x[2]) / Ab[1,1]
| x
|
| array([ 0. , -24.66666667, -16.66666667])
```

Using the first row we then compute x_0 :

```
| Ab
|
| array([[ 8. , -6. ,  3. , -14. ],
|        [ 0. ,  1.5, -3.75, 25.5 ],
|        [ 0. ,  0. , -4.5 , 75. ]])
```

```
| x[0] = (Ab[0,3] - Ab[0,1:3]@x[1:]) / Ab[0,0]
| x
| array([ -14.          , -24.66666667, -16.66666667])
|
| Let us check the solution:
|
| A @ x - b
| array([ -3.55271368e -15,  7.10542736e -15,  7.10542736e -15])
```

Summary of Gaussian elimination

In the module `tools.py` we save the functions:

```
| def gaussian_elimination(A, b, show_steps = False):
|     Ab = np.column_stack((A, b))
|     for p, pivot_row in enumerate(Ab[: -1]):
|         for row in Ab[p+1:]:
|             if pivot_row[p]:
|                 row[p:] -= pivot_row[p:]*row[p]/pivot_row[p]
|             else:
|                 raise Exception('Division by 0.')
|         if show_steps:
|             print('Step: {:g}'.format(p))
|             print(Ab)
|     return Ab
|
| def gaussian_el_solution(Ub):
|     n = len(Ub)
|     x = np.zeros(n)
|     for p, pivot_row in enumerate(Ub[: -1]):
|         x[n - p - 1] = (pivot_row[ -1] - pivot_row[n - p: -1] @ x[n - p:] ) /
|             (pivot_row[n - p - 1])
|     return x
```

The algorithm by which we compute the solution from an upper-triangular system of equations $\mathbf{U}\mathbf{x} = \mathbf{b}$ is called **back substitution**; $\mathbf{U}$ is an upper-triangular matrix.

If we were to solve the system $\mathbf{L}\mathbf{x} = \mathbf{b}$ where $\mathbf{L}$ is a lower-triangular matrix, we would call this method **forward substitution**.

```
| Ub = gaussian_elimination(A, b, show_steps=False)
| Ub
|
| array([[ 8.  , -6.  ,  3.  , -14.  ],
|        [ 0.  , 1.5 , -3.75, 25.5 ],
|        [ 0.  , 0.  , -4.5 , 75.  ]])
|
| gaussian_el_solution(Ub)
|
| array([ -14.          , -24.66666667, -16.66666667])
```

5.3.1 Numerical complexity

We assess numerical complexity by the number of mathematical operations required for the computation. To solve n linear equations with Gaussian elimination, we thus need approximately $n^3/3$ mathematical operations. To determine the unknowns $\mathbf{x}$ we need an additional approximately n^2 operations.

In Gaussian elimination we carried out the elimination only for the terms below the diagonal; if we continued the elimination and carried it out also for the terms above the diagonal, we would perform the so-called *Gauss-Jordan* elimination, for which, however, we need approximately $n^3/2$ operations in total, i.e., an additional approximately $n^3/6$ (which is considered the main drawback of this method).

5.3.2 Using the numpy library

Solving a system of linear equations with `numpy.linalg.solve` ([documentation](#)):

```
| solve(a, b)
```

where `a` is the coefficient matrix (or list of matrices) and `b` is the constant vector (or list of vectors). The function returns the vector (or list of vectors) of solutions.

```
| np.linalg.solve(A, b)
```

```
| array([ -14.          , -24.66666667, -16.66666667])
```

5.4 Additional

5.5 Example of symbolic solving of a system of linear equations with sympy

```
| import sympy as sym
| sym.init_printing()
```

```
| A11, A12, A21, A22 = sym.symbols('A11, A12, A21, A22')
| x1, x2 = sym.symbols('x1, x2')
| b1, b2 = sym.symbols('b1, b2')
| A = sym.Matrix([[A11, A12],
|                  [A21, A22]])
| x = sym.Matrix([[x1],
|                  [x2]])
| b = sym.Matrix([[b1],
|                  [b2]])
```

```
| eq = sym.Eq(A*x,b)
| eq
```

$$\begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad (5.9)$$

```
| %%timeit
| solution = sym.solve(eq,[x1, x2])
| solution
```

$$\left\{ x_1 : \frac{-A_{12}b_2 + A_{22}b_1}{A_{11}A_{22} - A_{12}A_{21}}, x_2 : \frac{A_{11}b_2 - A_{21}b_1}{A_{11}A_{22} - A_{12}A_{21}} \right\} \quad (5.10)$$

```
| A.det()
```

$$A_{11}A_{22} - A_{12}A_{21} \quad (5.11)$$

Chapter 6

Systems of linear equations - continued

6.1 LU decomposition

To solve a system of linear equations, Gaussian elimination requires the fewest arithmetic operations.

When the coefficient matrix $\mathbf{A}$ does not change and only the constant vector $\mathbf{b}$ changes, we can avoid repeating Gaussian elimination on the coefficient matrix. By decomposing the matrix $\mathbf{A}$ we can reach the solution with fewer arithmetic operations. To this end we will look at the LU decomposition!

Any matrix can be written as a product of two matrices:

$$\mathbf{A} = \mathbf{B} \mathbf{C}. \quad (6.1)$$

Here, there are infinitely many ways to write the matrices $\mathbf{B}$ and $\mathbf{C}$.

In the LU decomposition we require that the matrix $\mathbf{B}$ is lower triangular and the matrix $\mathbf{C}$ is upper triangular:

$$\mathbf{A} = \mathbf{L} \mathbf{U}. \quad (6.2)$$

Each of the matrices $\mathbf{L}$ and $\mathbf{U}$ has $(n+1)n/2$ nonzero elements; together that is $n^2 + n$ unknowns. The known matrix $\mathbf{A}$ defines n^2 values. So to uniquely determine the matrices $\mathbf{L}$ and $\mathbf{U}$ we are missing n equations. Here we will use the LU decomposition, which obtains the additional equations through the condition $L_{ii} = 1, i = 0, 1, \dots, n-1$.

The system of linear equations:

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (6.3)$$

is therefore written using the decomposition of the matrix $\mathbf{A}$:

$$\mathbf{L} \underbrace{\mathbf{U} \mathbf{x}}_{\mathbf{y}} = \mathbf{b}. \quad (6.4)$$

We now reach the solution of the system $\mathbf{A} \mathbf{x} = \mathbf{b}$ by solving two triangular systems of equations.

First we compute the vector $\mathbf{y}$:

$$\mathbf{L} \mathbf{y} = \mathbf{b}. \quad (\text{forward substitution}) \quad (6.5)$$

Once $\mathbf{y}$ is computed, we can determine $\mathbf{x}$ from:

$$\mathbf{U} \mathbf{x} = \mathbf{y} \quad (\text{back substitution}) \quad (6.6)$$

6.1.1 LU decomposition of the coefficient matrix $\mathbf{A}$

In what follows we will show that Gaussian elimination actually represents an LU decomposition of the coefficient matrix $\mathbf{A}$. We will rely on symbolic computation, so let's import the `sympy` package:

```
import sympy as sym # import sympy
sym.init_printing() # for nice display of expressions
```

Let's begin the demonstration with an example of symbolically written matrices $\mathbf{L}$ and $\mathbf{U}$ of dimension 3×3 :

```
L21, L31, L32 = sym.symbols('L21, L31, L32')
U11, U12, U13, U22, U23, U33 = sym.symbols('U11, U12, U13, U22, U23, U33')
L = sym.Matrix([[ 1, 0, 0],
                 [L21, 1, 0],
                 [L31, L32, 1]])
U = sym.Matrix([[U11, U12, U13],
                 [ 0, U22, U23],
                 [ 0, 0, U33]])
```

The coefficient matrix $\mathbf{A}$ written with the elements of the matrices $\mathbf{L}$ and $\mathbf{U}$ is therefore:

```
A = L*U
A
```

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + U_{22} & L_{21}U_{13} + U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & L_{31}U_{13} + L_{32}U_{23} + U_{33} \end{bmatrix} \quad (6.7)$$

Let's now perform Gaussian elimination on the coefficient matrix $\mathbf{A}$.

Using the first row, we perform Gaussian elimination in the first column:

```
A[1,:] -= L21 * A[0,:]
A[2,:] -= L31 * A[0,:]
A
```

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & L_{32}U_{22} & L_{32}U_{23} + U_{33} \end{bmatrix} \quad (6.8)$$

We continue in the second column:

```
A[2,:] -= L32 * A[1,:]
A
```

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \quad (6.9)$$

From the elimination above we observe:

1. the matrix $\mathbf{U}$ equals the matrix we obtain if we perform Gaussian elimination on the coefficient matrix $\mathbf{A}$.
- the off-diagonal terms of $\mathbf{L}$ are the factors we used in Gaussian elimination.

6.1.2 Numerical implementation of the LU decomposition

We will look at the numerical implementation on the system defined as:

```
import numpy as np

A = np.array([[8, -6, 3],
              [-6, 6, -6],
              [3, -6, 6]], dtype=float)
b = np.array([-14, 36, 6], dtype=float)
```

Let's perform Gaussian elimination and store the coefficients by which we multiply the pivot row m into the matrix L at the position with the indices that the eliminated element has in the matrix A .

```
(v, s) = A.shape # v=number of rows, s=number of columns
U = A.copy()      # prepare the matrix U as a copy of A
L = np.zeros_like(A) # prepare the matrix L of the same dimension as A (values 0)
## elimination
for p, pivot_row in enumerate(U[: -1]):
    for i, row in enumerate(U[p+1:]):
        if pivot_row[p]:
            m = row[p]/pivot_row[p]
            row[p:] = row[p:] -pivot_row[p:]*m
            L[p+1+i, p] = m
    print('Step: {}'.format(p))
    print(U)
```

```
Step: 0
[[ 8.  -6.   3. ]
 [ 0.   1.5 -3.75]
 [ 0.  -3.75  4.875]]
Step: 1
[[ 8.  -6.   3. ]
 [ 0.   1.5 -3.75]
 [ 0.   0.  -4.5 ]]
```

L

```
array([[ 0.   ,  0.   ,  0.   ],
       [-0.75,  0.   ,  0.   ],
       [ 0.375, -2.5  ,  0.   ]])
```

Let's fill in the diagonal of L :

```
for i in range(v):
    L[i, i] = 1.
```

L

```
array([[ 1.   ,  0.   ,  0.   ],
       [-0.75,  1.   ,  0.   ],
       [ 0.375, -2.5  ,  1.   ]])
```

Now let's solve the lower triangular system of equations $L\mathbf{y} = \mathbf{b}$:

```
## forward substitution
y = np.zeros_like(b)
for i, b_ in enumerate(b):
    y[i] = (b_ - np.dot(L[i, :i], y[:i]))
```

```
| y
| array([ -14. ,  25.5,  75. ])

| We continue by solving the upper triangular system  $\mathbf{U} \mathbf{x} = \mathbf{y}$ :

| U
| array([[ 8. , -6. ,  3. ],
|        [ 0. ,  1.5, -3.75],
|        [ 0. ,  0. , -4.5 ]])

| y
| array([ -14. ,  25.5,  75. ])

| ## back substitution
| x = np.zeros_like(b)
| for i in range(v - 1, -1, -1):
|     x[i] = (y[i] - np.dot(U[i, i+1:], x[i+1:])) / U[i, i]
| x
| array([ -14.          , -24.66666667, -16.66666667])
```

As we noted above, when the constant vector $\mathbf{b}$ changes, repeating Gaussian elimination is not necessary. We only need to perform forward and then back substitution. Let's look at an example:

```
| b = np.array([ -1.,  6.,  7.])
| y = np.zeros_like(b)
| for i, b_ in enumerate(b):# forward substitution
|     y[i] = (b_ - np.dot(L[i, :i], y[:i]))
| x = np.zeros_like(b)
| for i in range(v - 1, -1, -1):# back substitution
|     x[i] = (y[i] - np.dot(U[i, i+1:], x[i+1:])) / U[i, i]
| x
| array([ -4.33333333, -7.88888889, -4.55555556])
| A.dot(x)
| array([ -1.,  6.,  7.])
```

6.2 Pivoting

Let's look at the following system of equations:

```
| A = np.array([[0, -6, 6],
|               [-6, 6, -6],
|               [8, -6, 3]], dtype=float) # try omitting dtype=float here and check the
|               result
| b = np.array([6, 36, -14], dtype=float)
```

If we were to perform Gaussian elimination in the first column of the matrix $\mathbf{A}$:

```
| A[1,:] - A[1,0]/A[0,0] * A[0,:]
```

```

/tmp/ipykernel_342481/3999418912.py:1: RuntimeWarning: divide by zero encountered in
scalar divide
  A[1,:] - A[1,0]/A[0,0] * A[0,:]
/tmp/ipykernel_342481/3999418912.py:1: RuntimeWarning: invalid value encountered in
multiply
  A[1,:] - A[1,0]/A[0,0] * A[0,:]

array([ nan, -inf,  inf])

```

We notice that we have a problem with division by 0 in the first row. The elementary operations we can perform on the system allow swapping arbitrary rows. We can rearrange the system so that the pivot element is not equal to 0. Even so, it can happen that the pivot element we divide by has a very small value ε . Because this would increase the round-off error, among all the rows we choose for the pivot row the one whose pivot has the largest absolute value.

If during Gaussian elimination we swap rows so that the pivot element is the largest, we call this **row pivoting** or also **partial pivoting**. This makes Gaussian elimination numerically stable.

It can be shown that when solving a system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ in which the matrix $\mathbf{A}$ is diagonally dominant, row pivoting is not necessary. Solving without pivoting is numerically stable.

A square matrix $\mathbf{A}$ of dimension n is **diagonally dominant** if the absolute value of the diagonal element of each row is greater than the sum of the absolute values of the other elements in the row:

$$|A_{ii}| > \sum_{j=1, j \neq i}^n |A_{ij}| \quad (6.10)$$

6.2.1 Gaussian elimination with partial pivoting

We will look at Gaussian elimination with **partial pivoting**. Without partial pivoting, in the i -th elimination step we choose row i as the pivot. With partial pivoting we first check whether the i -th diagonal element is, in absolute value, the largest element in column i on or below the diagonal; if not, we swap row i with the row below it that has the largest element in absolute value in column i . With partial pivoting we reduce the influence of the round-off error on the result.

If we were to perform **full pivoting**, in addition to swapping rows we would also use swapping the order of the variables (column swapping). Full pivoting improves stability, but it is rarely used and we will not address it here.

The algorithm for Gaussian elimination with partial pivoting is therefore:

```

def gaussian_elimination_pivoting(A, b, show_steps=False):
    """ Returns the Gaussian elimination of the augmented coefficient matrix,
    using partial pivoting.

    :param A: coefficient matrix
    :param b: constant vector
    :param show_steps: whether to print the individual steps
    :return Ab: trapezoidal augmented coefficient matrix
    """
    Ab = np.column_stack((A, b))
    for p in range(len(Ab) - 1):
        p_max = np.argmax(np.abs(Ab[p:,p]))+p
        if p != p_max:
            Ab[[p], :], Ab[[p_max], :] = Ab[[p_max], :], Ab[[p], :]
        pivot_row = Ab[p, :]
        for row in Ab[p + 1:]:

```

```

        if pivot_row[p]:
            row[p:] -= pivot_row[p:] * row[p] / pivot_row[p]
    if show_steps:
        print('Step: {}'.format(p))
        print('Pivot row:', pivot_row)
        print(Ab)
    return Ab

```

A

```

array([[ 0., -6.,  6.],
       [-6.,  6., -6.],
       [ 8., -6.,  3.]])

```

```
Ab = gaussian_elimination_pivoting(A, b, show_steps=True)
```

```

Step: 0
Pivot row: [  8.  -6.   3. -14.]
[[  8.   -6.    3.  -14. ]
 [  0.   1.5  -3.75  25.5 ]
 [  0.  -6.    6.    6.  ]]
Step: 1
Pivot row: [ 0. -6.  6.  6.]
[[  8.   -6.    3.  -14. ]
 [  0.   -6.    6.    6.  ]
 [  0.    0.  -2.25  27.  ]]

```

6.2.2 LU decomposition with partial pivoting

Similarly to Gaussian elimination, we can also extend the LU decomposition with partial pivoting. Solving thus becomes numerically **stable**. In doing so we must store the information about the row swaps, which we then pass to the function for solving the corresponding triangular systems.

```

def LU_decomposition_pivoting(A, show_steps=False):
    """ Returns the LU decomposition matrix and the vector of swapped rows
    (pivoting),
    using partial pivoting.

    :param A: coefficient matrix
    :param show_steps: print the individual steps
    :return LU: LU matrix
    :return pivoting: vector of row swaps (important when finding the solution)
    """
    LU = A.copy()
    pivoting = np.arange(len(A))
    for p in range(len(LU) - 1):
        p_max = np.argmax(np.abs(LU[p:,p]))+p
        if p != p_max:
            LU[[p], :], LU[[p_max], :] = LU[[p_max], :], LU[[p], :]
            pivoting[p], pivoting[p_max] = pivoting[p_max], pivoting[p]
        pivot_row = LU[p, :]
        for row in LU[p + 1:]:
            if pivot_row[p]:
                m = row[p] / pivot_row[p]
                row[p:] -= pivot_row[p:] * m
                row[p] = m
            else:
                raise Exception('Division by 0.')
    if show_steps:
        print('Step: {}'.format(p))
        print('Pivot row:', pivot_row)

```

```

        print(LU)
    return LU, pivoting

```

Let's look at an example:

```

A = np.array([[0, -6, 6],
              [-6, 6, -6],
              [8, -6, 3]], dtype=float)
b = np.array([-14, 36, 6], dtype=float)

lu, piv = LU_decomposition_pivoting(A, show_steps=True)

Step: 0
Pivot row: [ 8. -6.  3.]
[[ 8.  -6.   3. ]
 [ -0.75  1.5 -3.75]
 [ 0.  -6.   6. ]]
Step: 1
Pivot row: [ 0. -6.  6.]
[[ 8.  -6.   3. ]
 [ 0.  -6.   6. ]
 [ -0.75 -0.25 -2.25]]

```

In the example above we used a compact way of storing the triangular matrices **L** and **U**; each is in fact defined by 6 elements, and for the matrix **L** we know that the diagonal elements equal 1. The matrix `lu` thus contains $3 \times 3 = 9$ elements:

```

lu

array([[ 8.  , -6.  ,  3.  ],
       [ 0.  , -6.  ,  6.  ],
       [-0.75, -0.25, -2.25]])

```

On the diagonal and above the diagonal are the values of the upper triangular matrix **U**, while below the diagonal are the subdiagonal elements of the matrix **L**. When computing the solution we will take into account that the diagonal values of **L** equal 1.

The numeric list `piv` tells us how the rows were swapped, which must be taken into account when computing the solution:

```

piv

array([2, 0, 1])

```

Let's now also determine the solution (the code for computing the solution is in the module `tools.py`)

```

from modules import tools

r = tools.LU_solve_pivoting(lu, b, piv)
r

array([-3.66666667, -14.11111111, -16.44444444])

```

Checking the solution:

```

A@r

array([-14.,  36.,   6.])

```

6.3 The SciPy module

The SciPy module is built on the `numpy` module and contains many different high-level programs/modules/functions. The theoretical background of the modules is of course various numerical algorithms; we will get to know some of them within this textbook as well. A good source for these numerical algorithms in connection with SciPy is the [documentation](#). For an excellent introduction to SciPy you can watch the [YouTube video: SciPy Tutorial \(2022\): For Physicists, Engineers, and Mathematicians](#).

A brief overview of the module hierarchy:

- Linear algebra ([scipy.linalg](#))
- Integration ([scipy.integrate](#))
- Optimization ([scipy.optimize](#))
- Interpolation ([scipy.interpolate](#))
- Fourier transform ([scipy.fft](#))
- Sparse matrices ([scipy.sparse](#))
- Statistics ([scipy.stats](#))
- Signal processing ([scipy.signal](#))
- Special functions ([scipy.special](#))
- Multidimensional image processing ([scipy.ndimage](#))
- File handling ([scipy.io](#))

In the following lectures we will look at some of the submodules in more detail.

Let's look at how the *LU decomposition* is implemented within SciPy:

```
| from scipy.linalg import lu_factor, lu_solve
```

The function `scipy.linalg.lu_factor` ([documentation](#)):

```
| lu_factor(a, overwrite_a=False, check_finite=True)
```

requires the input of a coefficient matrix (or a list of coefficient matrices) `a`; `overwrite_a`, if `True`, overwrites the value of `a` with the result of the decomposition (this can be important to save memory and increase speed). The function `lu_factor` returns a tuple `(lu, piv)`:

- `lu` - the $\mathbf{L} \setminus \mathbf{U}$ matrix (a matrix of the same dimension as `a`, but below the diagonal it contains the elements of $\mathbf{L}$, while the remaining elements define $\mathbf{U}$; the diagonal elements of $\mathbf{L}$ have values 1).
- `piv` - the pivot indices, representing a permutation matrix $\mathbf{P}$: row `i` of the matrix `a` was swapped with row `piv[i]` (at each step the previous state is taken into account, slightly different logic than in our function `LU_decomposition_pivoting`).

`lu_factor` is used together with the function `scipy.linalg.lu_solve` ([documentation](#)), which gives us the solution of the system:

```
| lu_solve(lu_and_piv, b, trans=0, overwrite_b=False, check_finite=True)
```

`lu_and_piv` is the tuple of the result `(lu, piv)` from `lu_factor`, `b` is the constant vector (or a list of vectors). The other parameters are optional.

Let's look at how to use it:

```
| A = np.array([[0, -6, 6],
|               [-6, 6, -6],
|               [8, -6, 3]], dtype=float)
| b = np.array([-14, 36, 6], dtype=float)
|
| lu, piv = lu_factor(A)
|
| lu
```

```

array([[ 8.  , -6.  ,  3.  ],
       [ 0.  , -6.  ,  6.  ],
       [-0.75, -0.25, -2.25]])

piv

array([2, 2, 2], dtype=int32)

Let's obtain the solution:

r = lu_solve((lu, piv), b)
r

array([-3.66666667, -14.11111111, -16.44444444])

Let's check the validity of the solution:

A@r

array([-14.,  36.,   6.])

```

6.4 Computing the inverse matrix

The inverse matrix of a square matrix $\mathbf{A}$ of order $n \times n$ is denoted by $\mathbf{A}^{-1}$. It is a matrix of order $n \times n$, such that

$$\mathbf{A} \mathbf{A}^{-1} = \mathbf{A}^{-1} \mathbf{A} = \mathbf{I}, \quad (6.11)$$

where $\mathbf{I}$ is the identity matrix.

The most efficient way to compute the inverse of a matrix $\mathbf{A}$ is to solve the matrix equation:

$$\mathbf{A} \mathbf{X} = \mathbf{I}. \quad (6.12)$$

The matrix $\mathbf{X}$ is the inverse of the matrix $\mathbf{A}$: $\mathbf{A}^{-1} = \mathbf{X}$.

Computing the inverse matrix is therefore equal to solving n systems of n linear equations:

$$\mathbf{A} \mathbf{x}_i = \mathbf{b}_i, \quad i = 0, 1, 2, \dots, n-1, \quad (6.13)$$

where $\mathbf{b}_i$ is the i -th column of the matrix $\mathbf{B} = \mathbf{I}$.

Computational cost: we perform the LU decomposition on the matrix $\mathbf{A}$ (computational cost of order n^3) and then find the solution for each $\mathbf{x}_i$ ($2n^2$ arithmetic operations for each i). The total computational cost is therefore still of order n^3 . (If we were to perform Gaussian elimination n times, the cost would be of order n^4 .)

```

lu_piv = lu_factor(A)
lu_piv

(array([[ 8.  , -6.  ,  3.  ],
       [ 0.  , -6.  ,  6.  ],
       [-0.75, -0.25, -2.25]]),
 array([2, 2, 2], dtype=int32))

```

Here the point of the LU decomposition now becomes apparent; we compute the decomposition only once and then, for each constant vector, find the solution by solving two triangular systems. If we were to solve the systems by Gaussian elimination, we would need $(n^3 + n^2)n$ arithmetic operations.

Let's compute the inverse of the matrix $\mathbf{A}$. First we prepare the identity matrix using `np.identity()`:

```
| np.identity(len(A))

| array([[1., 0., 0.],
|        [0., 1., 0.],
|        [0., 0., 1.]])
```

Then we solve the system of equations for each column of the identity matrix:

```
| A_inv = np.zeros_like(A) # zeros_like returns a matrix of the shape and type of
| matrix A with values 0
| for b, a_inv in zip(np.identity(len(A)), A_inv):
|     a_inv[:] = lu_solve(lu_piv, b)
| A_inv = A_inv.T
```

The solution is therefore:

```
| A_inv

| array([[ -0.16666667, -0.16666667,  0.          ],
|        [ -0.27777778, -0.44444444, -0.33333333],
|        [ -0.11111111, -0.44444444, -0.33333333]])
```

Let's check the solution:

```
| A@A_inv

| array([[ 1.00000000e+00,  0.00000000e+00,  1.11022302e -16],
|        [ 2.22044605e -16,  1.00000000e+00, -1.11022302e -16],
|        [-1.66533454e -16,  0.00000000e+00,  1.00000000e+00]])
```

Solution using NumPy:

```
| %%timeit
| np.linalg.inv(A)

| array([[ -0.16666667, -0.16666667,  0.          ],
|        [ -0.27777778, -0.44444444, -0.33333333],
|        [ -0.11111111, -0.44444444, -0.33333333]])
```

With appropriate use of functions from the SciPy module we can reach the solution even faster. Into the function `lu_solve` we can insert a vector **b** or a matrix **B**, each column *i* of which represents a new constant vector **b_i**.

```
| %%timeit
| lu_solve(lu_piv, np.identity(len(A)))

| array([[ -0.16666667, -0.16666667,  0.          ],
|        [ -0.27777778, -0.44444444, -0.33333333],
|        [ -0.11111111, -0.44444444, -0.33333333]])
```

6.5 Solving overdetermined systems

```
| from IPython.display import YouTubeVideo
| YouTubeVideo('N6upu99LbfQ', width=800, height=300)
```

$$\begin{array}{c} \mathbf{A} \end{array} \begin{bmatrix} 1.17 & -1.08 \\ 2.76 & -1.66 \\ 5.68 & -2.38 \\ 9.07 & -3.01 \\ 26.95 & -5.19 \\ 105.54 & -10.27 \end{bmatrix} \begin{array}{c} \mathbf{x} \\ \begin{bmatrix} a \\ b \end{bmatrix} \end{array} = \begin{array}{c} \mathbf{b} \\ \begin{bmatrix} 3.05 \\ 3.27 \\ 4.25 \\ 8.15 \\ 30.07 \\ 110.21 \end{bmatrix} \end{array}$$

$\text{rang}(\mathbf{A}) = 2$
 $\text{rang}(\mathbf{A}|\mathbf{b}) = 3$

www.Ladish.it

When we solve a system of m linear equations with n unknowns and $m > n$ holds and the rank of the augmented matrix $[\mathbf{A} | \mathbf{b}]$ equals $n + 1$, we have an overdetermined (inconsistent) system.

An overdetermined (also **inconsistent system**):

$$\mathbf{A} \mathbf{x} = \mathbf{b}, \quad (6.14)$$

has no solution. However, we can find the best approximate solution using the **least-squares** method.

The sum of squares of the residuals is defined by the dot product:

$$\|\mathbf{r}\|^2 = (\mathbf{A} \mathbf{x} - \mathbf{b})^T (\mathbf{A} \mathbf{x} - \mathbf{b}) \quad (6.15)$$

which we reshape into:

$$\|\mathbf{r}\|^2 = \mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x} - 2\mathbf{b}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{b}, \quad (6.16)$$

where we have used the fact that, due to the scalar value, $\mathbf{b}^T \mathbf{A} \mathbf{x} = (\mathbf{b}^T (\mathbf{A} \mathbf{x}))^T = (\mathbf{A} \mathbf{x})^T \mathbf{b}$.

The solution of the equation, the gradient of the sum of squares, determines its minimum:

$$\nabla_x \|\mathbf{r}\|^2 = 2\mathbf{A}^T \mathbf{A} \mathbf{x} - 2\mathbf{A}^T \mathbf{b} = 0 \quad (6.17)$$

Thus, from the normal equation:

$$\mathbf{A}^T \mathbf{A} \mathbf{x} = \mathbf{A}^T \mathbf{b} \quad (6.18)$$

we determine the best approximate solution:

$$\mathbf{x} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}. \quad (6.19)$$

By introducing the **pseudo-inverse matrix**:

$$\mathbf{A}^+ = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \quad (6.20)$$

of the matrix $\mathbf{A}$, the solution of the overdetermined system is written:

$$\mathbf{x} = \mathbf{A}^+ \mathbf{b}. \quad (6.21)$$

The procedure presented above is relatively simple, but it is numerically demanding and in some cases can be ill-conditioned; in practice it is recommended that the pseudo-inverse matrix be computed using the functions:

- `numpy.linalg.pinv` from the `numpy` module ([documentation](#)),
- The functions `pinv` or `pinvh` from the `scipy.linalg` module (the choice depends on the problem being addressed; see the [documentation](#)),

which are based on better numerical methods.

Example of a system with a unique solution:

```
## number of equations equal to the number of unknowns
A = np.array([[1., 2],
              [2, 3]])
b = np.array([5., 8])
np.linalg.solve(A, b)

array([1., 2.])
```

Now let's make an overdetermined system (the function `numpy.vstack` ([documentation](#)) stacks lists by columns, `numpy.random.seed` ([documentation](#)) resets the random number generator to the seed value `seed`, while `numpy.random.normal` ([documentation](#)) generates a normally distributed list of length `size` and standard deviation `scale`):

```
vA = np.vstack([A,A,A])
np.random.seed(seed=0)
vA += np.random.normal(scale=0.01, size=vA.shape) # corrupt the solution -> the
system is overdetermined
vA # coefficient matrix

array([[1.01764052, 2.00400157],
       [2.00978738, 3.02240893],
       [1.01867558, 1.99022722],
       [2.00950088, 2.99848643],
       [0.99896781, 2.00410599],
       [2.00144044, 3.01454274]])

vb = np.hstack([b,b,b])
vb += np.random.normal(scale=0.01, size=vb.shape)
vb # constant vector

array([5.00761038, 8.00121675, 5.00443863, 8.00333674, 5.01494079,
       7.99794842])
```

Now let's solve the overdetermined system:

```
Ap = np.linalg.pinv(vA)
Ap.dot(vb)

array([0.94512271, 2.02675939])
```

We see that the overdetermined system with random values (simulation of measurement noise) gives a similar result to the solution without noise. If we were to increase the noise level, the deviation from the unique solution would increase.

We can also determine the pseudo-inverse matrix ourselves and check the difference with the built-in function:

```
%%timeit # check the speed!
Ap2 = np.linalg.inv(vA.T@vA) @ vA.T

Ap2 - Ap

array([[ 4.44089210e -16,  3.44169138e -15, -1.44328993e -15,
         3.88578059e -15, -2.66453526e -15,  3.21964677e -15],
       [ 4.44089210e -16, -1.05471187e -15,  1.55431223e -15,
        -9.99200722e -16,  2.10942375e -15, -1.38777878e -15]])
```

6.6 Iterative methods

We often encounter large systems of linear equations whose coefficient matrix has few nonzero elements (such matrices are called **sparse**).

Iterative methods perform very well when solving such systems of linear equations; their advantages compared to direct methods are:

- arithmetic operations are performed only on the nonzero elements (despite the iterative solving, there can be fewer of them)
- the required memory space can be incomparably smaller.

6.6.1 Gauss-Seidel method

In what follows we will look at the idea of the *Gauss-Seidel* iterative method. First we write the system of equations $\mathbf{A} \mathbf{x} = \mathbf{b}$ as:

$$\sum_{j=0}^{n-1} A_{ij} x_j = b_i \quad i = 0, 1, \dots, n-1. \quad (6.22)$$

Suppose we are in step $k-1$ of the iterative method and the approximations $x_j^{(k-1)}$ ($j = 0, 1, \dots, n-1$) are known. From the sum above we isolate the term i :

$$A_{ii} x_i^{(k-1)} + \sum_{j=0, j \neq i}^{n-1} A_{ij} x_j^{(k-1)} = b_i \quad i = 0, 1, \dots, n-1. \quad (6.23)$$

Since the approximations $x_j^{(k-1)}$ do not exactly satisfy the linear problem, we can determine a new approximation $x_i^{(k)}$ from the equation above:

$$x_i^{(k)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=0}^{i-1} A_{ij} x_j^{(k)} - \sum_{j=i+1}^{n-1} A_{ij} x_j^{(k-1)} \right) \quad (6.24)$$

We have split the sum into two parts and, for computing the i -th term, taken into account the terms with index smaller than i already determined in the k -th iteration.

We stop the iterative approach when we reach the desired accuracy of the solution ϵ :

$$\|x_i^{(k)} - x_i^{(k-1)}\| < \epsilon \quad (6.25)$$

6.6.2 Example

```
A = np.array([[8, -1, 1],
              [-1, 6, -1],
              [0, -1, 6]], dtype=float)
b = np.array([-14, 36, 6], dtype=float)
```

Initial approximation:

```
x = np.zeros(len(A))
x
array([0., 0., 0.]
```

Let's prepare the matrix $\mathbf{A}$ without the diagonal elements (Why? Try to answer below, once we have performed the iterations.)

```
K = A.copy() # make a copy so that we don't overwrite the data
np.fill_diagonal(K, np.zeros(3)) # change only the diagonal elements
```

K

```
array([[ 0., -1.,  1.],
       [-1.,  0., -1.],
       [ 0., -1.,  0.]])
```

Let's perform the iterations:

$$x_i^{(k)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=0}^{i-1} A_{ij} x_j^{(k)} - \sum_{j=i+1}^{n-1} A_{ij} x_j^{(k-1)} \right) \quad (6.26)$$

(Since we will write the values into `x` right away, there is no need to split the sum into two parts)

```
for k in range(3):
    xk_1 = x.copy()
    print(5*'-' + f'iteration {k}' + 5*' -')
    for i in range(len(A)): # observe what happens when you run this cell several
        times!
            x[i] = (b[i] - K[i,:].dot(x))/A[i,i]
            print(f'Approximation for element {i}', x)
    e = np.linalg.norm(x - xk_1)
    print(f'Norm {e}')
```

```
- - - - -iteration 0 - - - - -
Approximation for element 0 [ -1.75  0.    0. ]
Approximation for element 1 [ -1.75      5.70833333  0. ]
Approximation for element 2 [ -1.75      5.70833333  1.95138889]
Norm 6.281360365408393
- - - - -iteration 1 - - - - -
Approximation for element 0 [ -1.28038194  5.70833333  1.95138889]
Approximation for element 1 [ -1.28038194  6.11183449  1.95138889]
Approximation for element 2 [ -1.28038194  6.11183449  2.01863908]
Norm 0.6227976321231091
- - - - -iteration 2 - - - - -
Approximation for element 0 [ -1.23835057  6.11183449  2.01863908]
Approximation for element 1 [ -1.23835057  6.13004808  2.01863908]
Approximation for element 2 [ -1.23835057  6.13004808  2.02167468]
Norm 0.04590845211691733
```

Let's check the solution

A@x

```
array([ -14.01517799,  35.9969644 ,   6.    ])
```

The method works well if the matrix is diagonally dominant (there are, however, methods that work even when the matrix is not diagonally dominant; see e.g.: J. Petrišič, Reševanje enačb, 1996, p. 149: Conjugate gradient method).

Chapter 7

Interpolation

7.1 Introduction

In **interpolation** we start from a table of (distinct) values x_i, y_i :

x	y
x_0	y_0
x_1	y_1
$\dots$	$\dots$
x_{n-1}	y_{n-1}

and we want to determine intermediate values. If we want to determine values outside the range of x in the table, we speak of **extrapolation**.

In **interpolation** we connect the points by assuming some function and adding the condition that the function *must* pass through the given points.

In **approximation** (curve fitting) we assume a function that fits the data as closely as possible (according to a chosen criterion).

Let us look at an example:

x	y
1.0	0.54030231
2.5	-0.80114362
4.0	-0.65364362

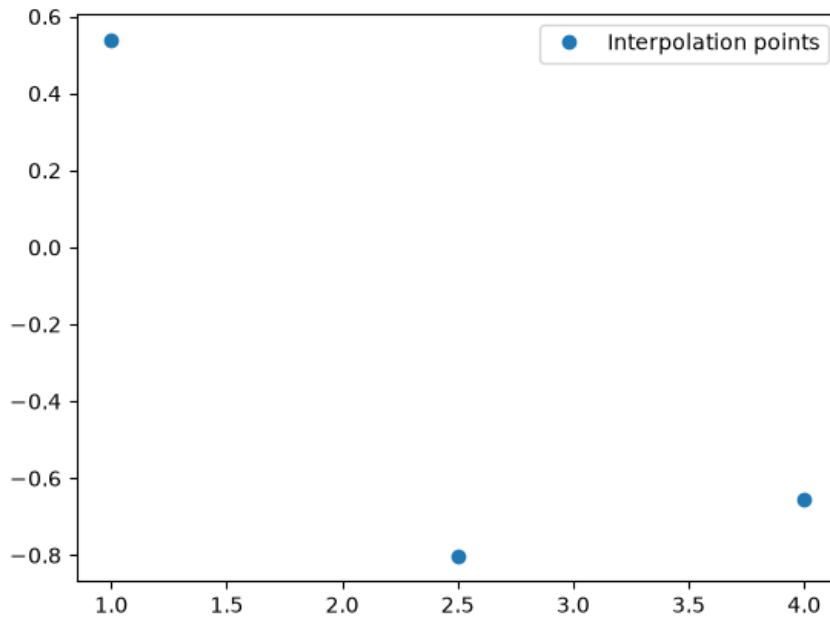
In interpolation we start from a table of values. So that we can easily show the error later, we generated the table above using the expression $y = \cos(x)$!

Let us prepare a numerical example; first we import the packages:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Next we prepare the table and the plot:

```
n = 3
x = np.linspace(1, 4, n)
f = np.cos # generalize the interpolated function (you can change it to another
function)
f_name = f.__str__().split('\')[1] # automatically take the name of the function
y = f(x) + np.random.rand(len(x))*0.3
plt.plot(x, y, 'o', label='Interpolation points');
plt.legend();
```



7.2 Polynomial interpolation

Polynomial interpolation seems most suitable, since it is simple!

A polynomial of degree $n - 1$:

$$y = a_0 x^{n-1} + a_1 x^{n-2} + \cdots + a_{n-2} x + a_{n-1}. \quad (7.1)$$

is defined by n constants a_i . To determine the n constants, we need n (distinct) equations. For each pair x_i, y_i we can therefore write:

$$y_i = a_0 x_i^{n-1} + a_1 x_i^{n-2} + \cdots + a_{n-2} x_i + a_{n-1}. \quad (7.2)$$

Since we are given n pairs, we can determine the n unknown constants a_i that define a polynomial of degree $n - 1$. The system of n linear equations can be written as:

$$\begin{bmatrix} x_0^{n-1} & x_0^{n-2} & \cdots & x_0^0 \\ x_1^{n-1} & x_1^{n-2} & \cdots & x_1^0 \\ \vdots & \vdots & \ddots & \vdots \\ x_{n-1}^{n-1} & x_{n-1}^{n-2} & \cdots & x_{n-1}^0 \end{bmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} \quad (7.3)$$

We write the system of linear equations in the form:

$$\mathbf{M} \mathbf{a} = \mathbf{b}. \quad (7.4)$$

Let us define the coefficient matrix $\mathbf{M}$:

```
M = np.asarray([[_**p for p in reversed(range(len(x)))] for _ in x])
M
array([[ 1.  ,  1.  ,  1.  ],
       [ 6.25,  2.5 ,  1.  ],
       [16.  ,  4.  ,  1.  ]])
```

We compute the coefficients $a_0, a_1, \dots$:

```
solution = np.linalg.solve(M, y)
solution
```

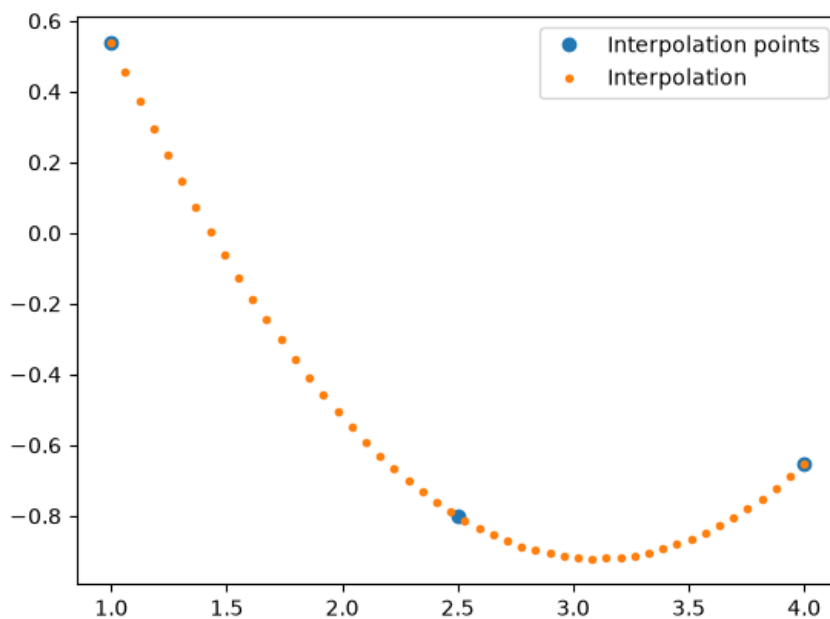
```
| array([ 0.33087687, -2.05236633,  2.26179176])
```

Let us prepare the interpolation polynomial as a Python function:

```
| def y_function(x, solution):
|     A = np.asarray([[_**p for p in reversed(range(len(solution)))] for _ in x])
|     return A.dot(solution)
```

Plotting the interpolation polynomial on a denser grid of points:

```
| xint = np.linspace(np.min(x), np.max(x), 50)
| yint = y_function(xint, solution)
| plt.plot(x, y, 'o', label='Interpolation points')
| plt.plot(xint, yint, '.', label='Interpolation')
| plt.legend();
```



The drawbacks of the procedure above are:

- the number of numerical operations grows proportionally to n^3 ,
- the problem can be ill-conditioned (as the polynomial degree increases, the ill-conditioning grows rapidly):

```
| np.linalg.cond(M)
```

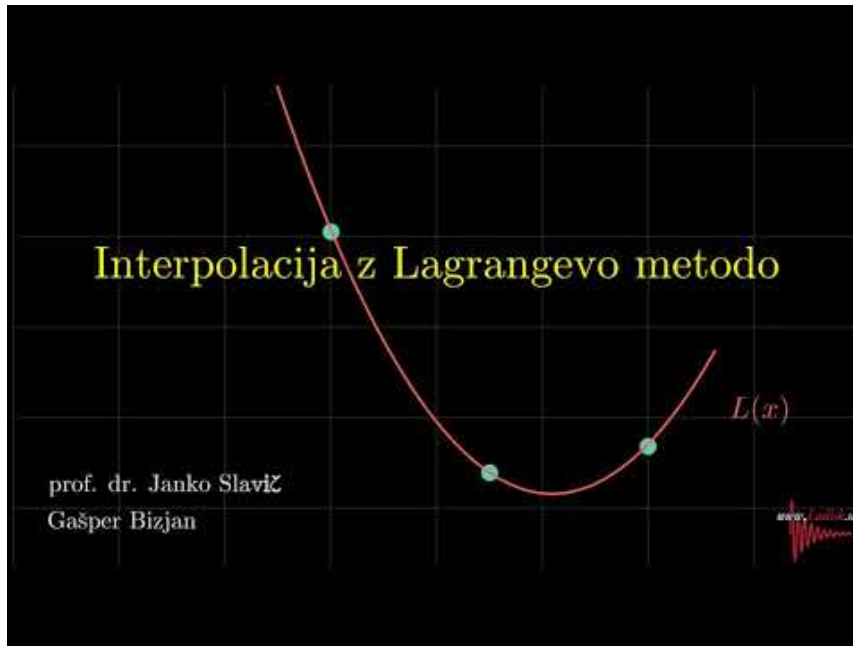
```
| np.float64(71.3022787031109)
```

Instruction: go back a few lines and change the number of interpolation points n to a higher value (e.g., 10).

7.3 Lagrange method

The *Lagrange* method does not require solving a system of equations and, from the standpoint of the number of computational operations (it grows proportionally to n^2 ([source](#))), it is better than the previously presented polynomial interpolation (where the number of operations grows proportionally to n^3), in which we solved a system of linear equations. The solution, of course, is exactly the same!

```
| from IPython.display import YouTubeVideo
| YouTubeVideo('c_TbU8XBHsw', width=800, height=300)
```



The Lagrange interpolation polynomial of degree $n - 1$ is defined as:

$$P_{n-1}(x) = \sum_{i=0}^{n-1} y_i l_i(x), \quad (7.5)$$

where l_i is the Lagrange polynomial:

$$l_i(x) = \prod_{j=0, j \neq i}^{n-1} \frac{x - x_j}{x_i - x_j}. \quad (7.6)$$

Let us look at the interpolation for the x and y data shown above.

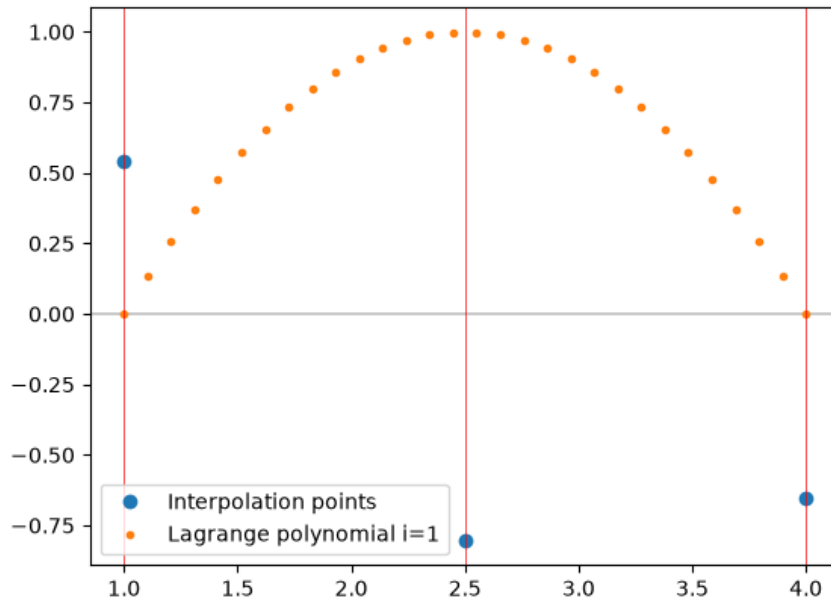
First let us define the Lagrange polynomials $l_i(x) = \prod_{j=0, j \neq i}^{n-1} \frac{x - x_j}{x_i - x_j}$:

```
def lagrange(x, x_int, i):
    """ Returns the values of the i -th Lagrange polynomial

    x: independent variable (scalar or numerical array)
    x_int: list of interpolation points
    i: index of the polynomial
    """
    Lx = 1.0
    for j in range(len(x_int)):
        if j != i:
            Lx *= (x - x_int[j]) / (x_int[i] - x_int[j])
    return Lx

def figure(i=0):
    xint = np.linspace(np.min(x), np.max(x), 30)
    plt.plot(x, y, 'o', label='Interpolation points')
    plt.axhline(0, color='k', linewidth=0.3);
    plt.plot(xint, lagrange(xint, x_int=x, i=i), '.', label=f'Lagrange polynomial i={i}');
    for _ in x:
        plt.axvline(_, color='r', linewidth=0.5);
    plt.legend()
    plt.show()

figure(i=1)
```



We observe that the i -th Lagrange polynomial has the value 1 at x_i , and zero at the other given points!

So if we multiply the Lagrange polynomial for $i = 0$ by y_0 , we will obtain the correct value at $x = x_0$, and zero at the other interpolation points; let us therefore implement the Lagrange interpolation polynomial:

$$P_{n-1}(x) = \sum_{i=0}^{n-1} y_i l_i(x), \quad (7.7)$$

```
def lagrange_interpolation(x, x_int, y_int):
    """ Returns the values of the Lagrange interpolation

    x: independent variable (scalar or numerical array)
    x_int: abscissa of the interpolation points
    y_int: ordinate of the interpolation points
    """
    y = 0.
    for i in range(len(x_int)):
        Lx = 1.0
        for j in range(len(x_int)):
            if j != i:
                Lx *= (x - x_int[j]) / (x_int[i] - x_int[j])
        y += y_int[i] * Lx
    return y
```

Let us prepare the figure:

```
def figure(i=0):
    xint = np.linspace(np.min(x), np.max(x), 30)
    plt.plot(x, y, 'o', label='Interpolation points')
    plt.plot(xint, lagrange(xint, x_int=x, i=i), '.', label=f'Lagrange polynomial i={i}');
    plt.plot(xint, lagrange_interpolation(xint, x_int=x, y_int=y),
             '.', label='Lagrange interp. polynomial');
    plt.axhline(0, color='k', linewidth=0.3);
    for _ in x:
        plt.axvline(_, color='r', linewidth=0.5);
    plt.legend()
    plt.show()
```

From `ipywidgets` we import `interact`, which is a powerful tool for automatically generating a (simple) user interface within the `jupyter` environment. Here we will use a relatively simple interaction with the figure; for an overview of all the capabilities we refer the curious reader to the [documentation](#).

Importing the `interact` function

```
| from ipywidgets import interact
|
| interact(figure);
|
| interactive(children=(IntSlider(value=0, description='i', max=1), Output()),
| _dom_classes=('widget -interact'),...
```

From the figure we see that the Lagrange polynomial i has the value 1 only at x_i and the value zero at the other points $\neq i$; when we multiply the Lagrange polynomial $l_i(x)$ by y_i we satisfy the i -th point from the table. Consequently, the Lagrange interpolation, with the sum of the Lagrange polynomials, interpolates the table.

Polynomial interpolation with a large number of points can be an ill-conditioned problem and we therefore advise against it.

7.3.1 Error estimate

If $f(x)$ is the function we are interpolating and $P_{n-1}(x)$ is the interpolation polynomial of degree $n-1$, then it can be shown (see, e.g.: Burden, Faires, Burden: Numerical Analysis) that the error of polynomial interpolation is:

$$e = f(x) - P_{n-1}(x) = \frac{f^{(n)}(\xi)}{n!} (x - x_0)(x - x_1) \cdots (x - x_{n-1}), \quad (7.8)$$

where $f^{(n)}$ is the derivative of the function, $n-1$ is the degree of the interpolation polynomial, and ξ is a value on the interpolated interval $[x_0, x_{n-1}]$.

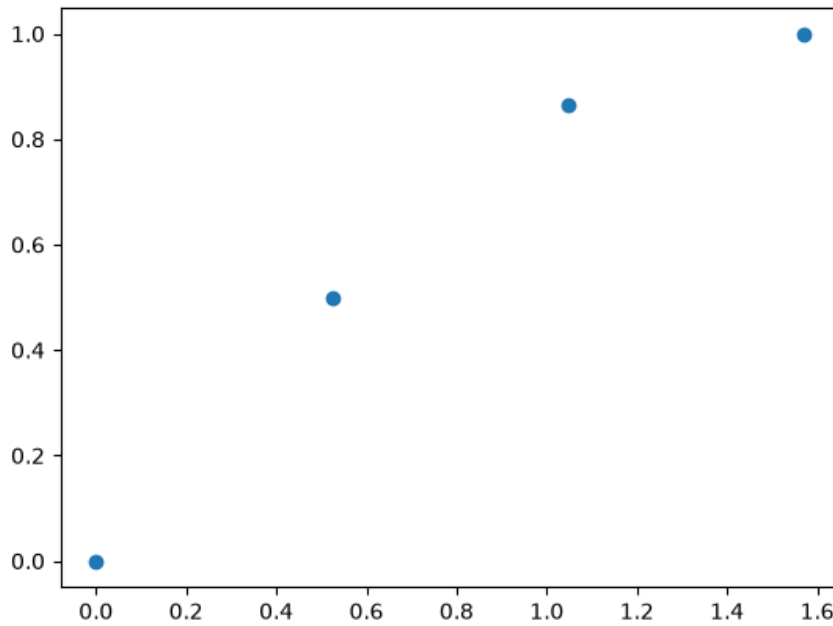
7.3.2 Example

Here we will look at the interpolation of the points:

```
| x = np.array([0.          ,  0.52359878,  1.04719755,  1.57079633])
| y = np.array([ 0.          ,  0.5          ,  0.8660254 ,  1.          ])
```

Let us plot the points:

```
| plt.plot(x, y, 'o');
```



Linear interpolation for the value at $x=1.57079633/2$:

```
| y_linear = lagrange_interpolation(x=x[ -1]/2, x_int=x[1:3], y_int=y[1:3])
| y_linear
|
| np.float64(0.6830127)
```

Quadratic:

```
| y_quadratic = lagrange_interpolation(x=x[ -1]/2, x_int=x[0:3], y_int=y[0:3])
| y_quadratic
|
| np.float64(0.6997595236464176)
```

Cubic

```
| y_cubic = lagrange_interpolation(x=x[ -1]/2, x_int=x, y_int=y)
| y_cubic
|
| np.float64(0.705889286844634)
```

7.3.3 Example of the error estimate

In interpolation we usually do not know the function $f(x)$, and we estimate the error using the formula:

$$e = \frac{f^{(n)}(\xi)}{n!} (x - x_0) (x - x_1) \cdots (x - x_n). \quad (7.9)$$

Here the value ξ is not known; since in the case of linear interpolation ($n = 2$) the second derivative of the sine function ($f^{(n)}$) is between -1 and +1, it holds that:

$$|e| \leq \left| \frac{-1}{2!} (\pi/4 - \pi/6) (\pi/4 - \pi/3) \right| = \frac{1}{2} \frac{\pi}{12} \frac{\pi}{12} = \frac{\pi^2}{288} = 0.034 \quad (7.10)$$

In addition to the **Lagrange method**, we could here also look at the **Newton method** of interpolation.

7.3.4 Interpolation using scipy

Let us look at interpolation within the module `scipy.interpolate` ([documentation](#)).

We will use the function for interpolating a table with splines, `scipy.interpolate.interp1d` ([documentation](#)):

```
interp1d(x, y, kind='linear', axis= -1, copy=True, bounds_error=None, fill_value=nan,
         assume_sorted=False)
```

We must provide at least two parameters: the lists of interpolation points `x` and `y`. The default parameter `kind='linear'` means that we interpolate with a piecewise linear function. `interp1d` returns a function `f` that we call (e.g., `y = f(x)`) to compute the interpolated value.

The `kind` parameter can also be, e.g.: `'zero'`, `'slinear'`, `'quadratic'` and `'cubic'`; in that case an interpolation spline of order 0, 1, 2, or 3 is used. We will look at splines in the next section.

Note: `interp1d` is marked as *legacy* in SciPy (an older interface that is no longer developed but still works). For new programs SciPy recommends `numpy.interp` (piecewise linear interpolation), `scipy.interpolate.CubicSpline` ([documentation](#)) and `scipy.interpolate.make_interp_spline` ([documentation](#)); all of them return an object that we call in the same way as `f` below.

```
from scipy.interpolate import interp1d
```

Let us define the data table:

```
x = np.array([ 1.          ,  2.14285714,  3.28571429,  4.42857143,  5.57142857,
               6.71428571,  7.85714286,  9.          ])
```

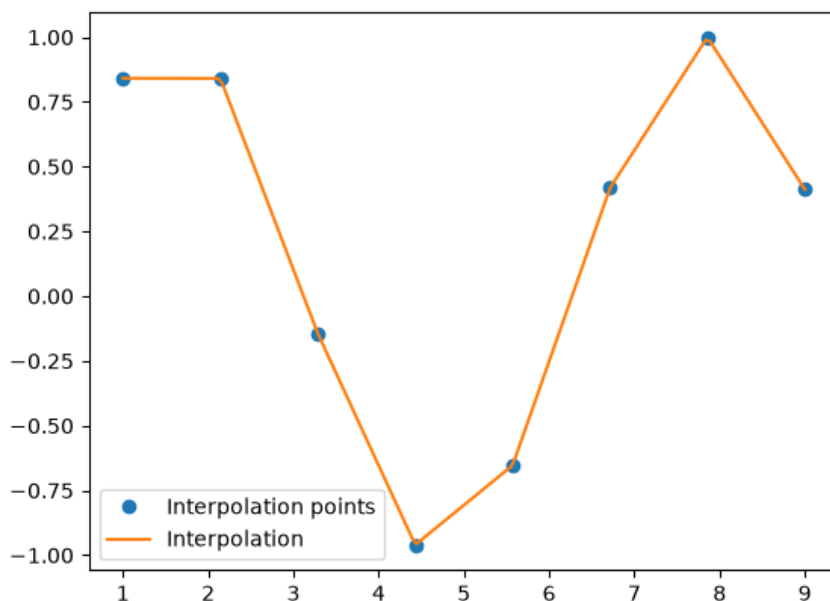
```
y = np.array([ 0.84147098,  0.84078711, -0.14362322, -0.95999344, -0.65316501,
               0.41787078,  0.999995  ,  0.41211849])
```

```
len(x)
```

```
8
```

```
from scipy.interpolate import interp1d
```

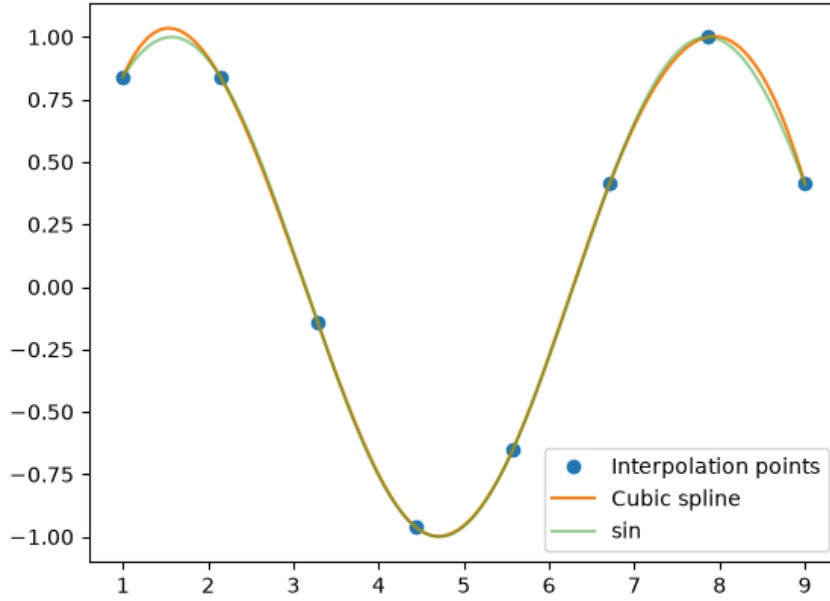
```
f = interp1d(x, y, kind='linear')
x_g = np.linspace(x[0], x[-1], 20*len(x) -1)
plt.plot(x, y, 'o', label='Interpolation points')
plt.plot(x_g, f(x_g), '-', label='Interpolation')
plt.legend();
```



7.4 Cubic splines

Before we go into the theory of splines, let us look at the result we obtain by calling the function `interp1d` with the parameter `kind='cubic'` (the result is a cubic spline).

```
f = interp1d(x, y, kind='cubic')
plt.plot(x, y, 'o', label='Interpolation points')
plt.plot(x_g, f(x_g), '-', label='Cubic spline')
plt.plot(x_g, np.sin(x_g), label='sin', alpha=0.5)
plt.legend();
```



Cubic splines are a common method of interpolation.

We require that: $x_0 < x_1 < \dots < x_n$.

From point x_i to x_{i+1} let the spline be the polynomial:

$$f_{i,i+1}(x) = a_{i,3}x^3 + a_{i,2}x^2 + a_{i,1}x + a_{i,0}, \quad (7.11)$$

where the values of the constants $a_{i,j}$ are unknown.

If, for example, we have $n + 1$ points, then n polynomials need to be determined.

The entire spline over the $n + 1$ points is defined by:

$$f(x) = \begin{cases} f_{0,1}(x); & x \in [x_0, x_1) \\ f_{1,2}(x); & x \in [x_1, x_2) \\ \vdots & \\ f_{n-1,n}(x); & x \in [x_{n-1}, x_n] \end{cases} \quad (7.12)$$

Each polynomial $f_{i,i+1}$ is defined by 4 constants $a_{i,j}$; in total we therefore have to compute $4n$ constants $a_{i,j}$.

How do we determine the constants $a_{i,j}$?

To determine the $4n$ unknowns we need $4n$ equations. Let us look at how we obtain them:

- n equations come from the interpolation condition: $y_i = f_{i,i+1}(x_i)$, $i = 0, 1, 2, \dots, n-1$
- 1 equation from the last point: $y_n = f_{n-1,n}(x_n)$
- $3(n-1)$ equations come from the C^2 continuity condition:

$$\lim_{x \rightarrow x_i^-} f(x) = \lim_{x \rightarrow x_i^+} f(x), \quad (7.13)$$

$$\lim_{x \rightarrow x_i^-} f'(x) = \lim_{x \rightarrow x_i^+} f'(x) \quad (7.14)$$

and

$$\lim_{x \rightarrow x_i^-} f''(x) = \lim_{x \rightarrow x_i^+} f''(x). \quad (7.15)$$

In total we have defined $4n - 2$ equations, so two are still missing!

The different types of splines differ in how we determine these two equations. In what follows we will look at *natural cubic splines*.

7.4.1 Natural cubic splines

Natural cubic splines are based on the idea of the Euler beam:

$$E I \frac{d^4 y}{dx^4} = q(x), \quad (7.16)$$

where E is the elastic modulus, I the second moment of area, and $q(x)$ the external distributed load. Since there is no external distributed load ($q(x) = 0$), it holds that:

$$E I \frac{d^4 y}{dx^4} = 0. \quad (7.17)$$

It follows that at each point a thin beam can be described by a third-degree polynomial.

C^2 continuity is ensured provided that the intermediate supports of the beam are hinges (so the moment has no discontinuous change).

The 2 missing unknowns for natural cubic splines are determined from the condition that the moment at the ends is equal to zero (hinged support):

$$f''(x_0) = 0 \quad \text{and} \quad f''(x_n) = 0 \quad (7.18)$$

The derivation is shown in more detail in the book Kiusalaas J: Numerical Methods in Engineering with Python 3, 2013, page 120 (see also J. Petrišič: Interpolacija, Faculty of Mechanical Engineering, 1999); a detailed derivation is beyond the scope of this book.

Here we will look only at the final result, which can be derived under the conditions above. In the case of an equidistant partition $h = x_{i+1} - x_i$ we thus derive the system of equations ($i = 1, \dots, n - 1$):

$$k_{i-1} + 4k_i + k_{i+1} = \frac{6}{h^2} (y_{i-1} - 2y_i + y_{i+1}). \quad (7.19)$$

where the unknown k_i is the second derivative of the piecewise function $k_i = f''_{i,i+1}(x_i)$.

We obtain a solvable system of equations if we add the boundary conditions for natural cubic splines:

$$k_0 = k_n = 0. \quad (7.20)$$

Once we determine the unknowns k_i , we use them in the piecewise defined function:

$$f_{i,i+1}(x) = \frac{k_{i+1}}{6} \left(\frac{(x - x_i)^3}{h} - (x - x_i) h \right) - \frac{k_i}{6} \left(\frac{(x - x_{i+1})^3}{h} - (x - x_{i+1}) h \right) + \frac{y_{i+1}(x - x_i) - y_i(x - x_{i+1})}{h}. \quad (7.21)$$

7.4.2 Numerical implementation

First we prepare a function that, for the given interpolation points, solves the system of linear equations and returns the coefficients k_i :

```
def cubic_splines_coefficients(x, y):
    """ Returns the cubic spline coefficients `k`, the coefficient matrix `A`, and
    the constants.

    x and y represent the list of known values; x must be equidistant.
    """
    n = len(x)
    A = np.zeros((n, n)) # prepare the coefficient matrix
    h = x[1] - x[0] # step h
    for i in range(n):
        if i==0 or i==n-1:
            A[i,i] = 1. # k_0 and k_n are zero, so here we put 1
                        # and in the constant vector we will put 0, so k_0 and k_n
                        # will be 0
        else:
            A[i, i-1:i+2] = np.asarray([1., 4., 1.])
    b = np.zeros(n)
    b[1:-1] = (6/h**2)*(y[: -2] - 2*y[1: -1] + y[2:]) # right -hand side of the
    equation above
    k = np.linalg.solve(A,b)
    return k, A, b
```

Note: in the linear problem above, the computation can be sped up greatly if we take advantage of the tridiagonality of the coefficient matrix! (See the [answer on stackoverflow.com](#)).

Let us look at an example of computing the coefficients:

```
x = np.asarray([1, 2, 3, 4, 5])
y = np.asarray([0, 1, 0, 1, 0])

k, A, b = cubic_splines_coefficients(x, y)
print('Coefficient matrix A of the linear system:\n', A)
print('Constant vector b of the linear system:      ', b)
print('The coefficients k are:', k)
```

```
Coefficient matrix A of the linear system:
[[1. 0. 0. 0. 0.]
 [1. 4. 1. 0. 0.]
 [0. 1. 4. 1. 0.]
 [0. 0. 1. 4. 1.]
 [0. 0. 0. 0. 1.]]
Constant vector b of the linear system:      [ 0. -12. 12. -12.  0.]
The coefficients k are: [ 0.          -4.28571429  5.14285714 -4.28571429  0.          ]
```

Next we also need the cubic polynomial in a given interval; let us implement the expression:

$$f_{i,i+1}(x) = \frac{k_{i+1}}{6} \left(\frac{(x-x_i)^3}{h} - (x-x_i)h \right) - \frac{k_i}{6} \left(\frac{(x-x_{i+1})^3}{h} - (x-x_{i+1})h \right) + \frac{y_{i+1}(x-x_i) - y_i(x-x_{i+1})}{h} \quad (7.22)$$

```
def cubic_splines(k, x, y, x_find):
    """ Returns the cubic spline at the partition `xint`.

    :param k: cubic spline coefficients
    :param x and y: known values, x must be equidistant
```

```

:param x_find: values where we want to compute the cubic spline
"""
h = x[1] - x[0]
i = int((x_find - x[0])//h)
if i >= len(k) - 1:
    i = len(k) - 2
out = ((x_find - x[i])**3/h - (x_find - x[i])*h)*k[i+1]/6.0 \
      - ((x_find - x[i+1])**3/h - (x_find - x[i+1])*h)*k[i]/6.0 \
      + (y[i+1]*(x_find - x[i]) \
        - y[i]*(x_find - x[i+1]))/h
return out

```

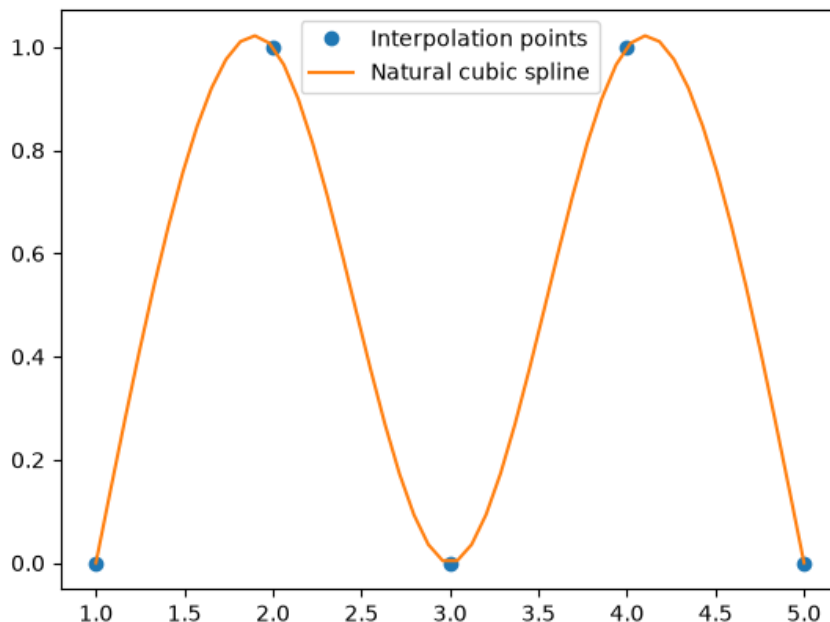
We compute the interpolated values:

```

xint = np.linspace(np.min(x), np.max(x), 50)
yint = np.asarray([cubic_splines(k, x, y, _) for _ in xint])

plt.plot(x, y, 'o', label='Interpolation points')
plt.plot(xint, yint, label='Natural cubic spline')
plt.legend();

```



7.5 Additional

- 2D interpolation: https://www.youtube.com/watch?v=__cJLVhdj0j4
- Machine translation: <https://pypi.org/project/goslate>

7.6 A few comments on the `scipy.interpolate` module

SciPy has a large number of different interpolations implemented (see the [documentation](#)). From the standpoint of usage, we will touch here on the object-oriented implementation `scipy.interpolate.InterpolatedUnivariateSpline` ([documentation](#)) (the older approach is based on functional programming, see the [documentation](#) of `scipy.interpolate.splrep`):

```

InterpolatedUnivariateSpline(x, y, w=None, bbox=[None, None], k=3, ext=0,
check_finite=False)

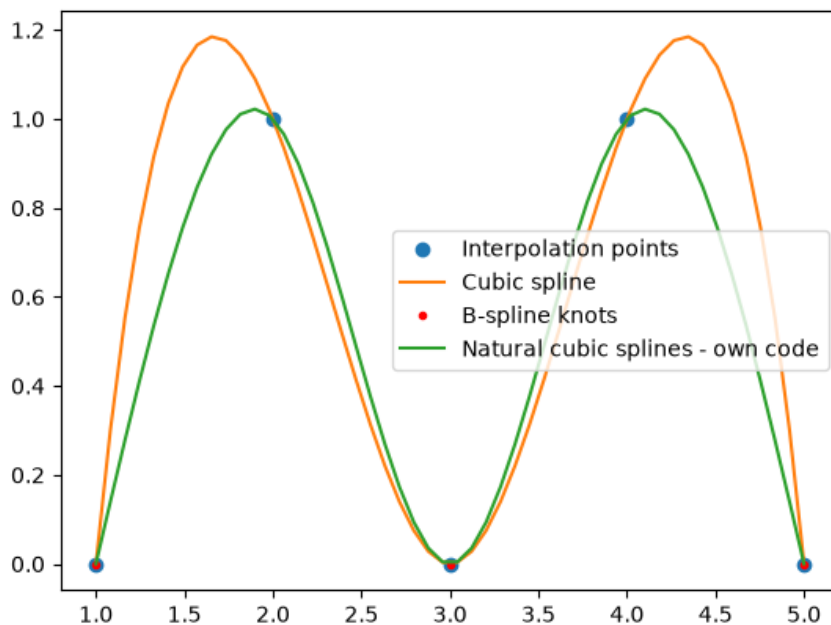
```

When initializing the `InterpolatedUnivariateSpline` object we must pass the interpolation points `x` and `y`. The argument `k`, with a default value of `k=3`, defines the order of the interpolation spline ($1 \leq k \leq 5$). An important optional parameter is also `w`, which defines the weights of the individual interpolation points (we use it if we want to give greater emphasis to certain regions).

Note: `InterpolatedUnivariateSpline` (and the related `UnivariateSpline`) have been marked as *legacy* since SciPy 1.15; the more modern equivalents are `make_interp_spline` (interpolation) and `make_smoothing_spline` or `make_splrep` (approximation, see the next lecture). The object interface below still works and, being clearer, is well suited for learning.

```
| from scipy.interpolate import InterpolatedUnivariateSpline

| spl = InterpolatedUnivariateSpline(x, y, k=3) # check out the options!
| plt.plot(x, y, 'o', label='Interpolation points')
| plt.plot(xint, spl(xint), label='Cubic spline');
| plt.plot(spl.get_knots(), spl(spl.get_knots()), 'r.', label='B -spline knots')
| plt.plot(xint, yint, label='Natural cubic splines - own code');
| plt.legend();
```



Since these are B-splines, the result is different from the one we derived with natural cubic splines. Unlike natural cubic splines, which have knots at the interpolation points, the knots of B-splines adapt to the data. In this particular example the knots are at the points:

```
| spl.get_knots()

| array([1., 3., 5.])
```

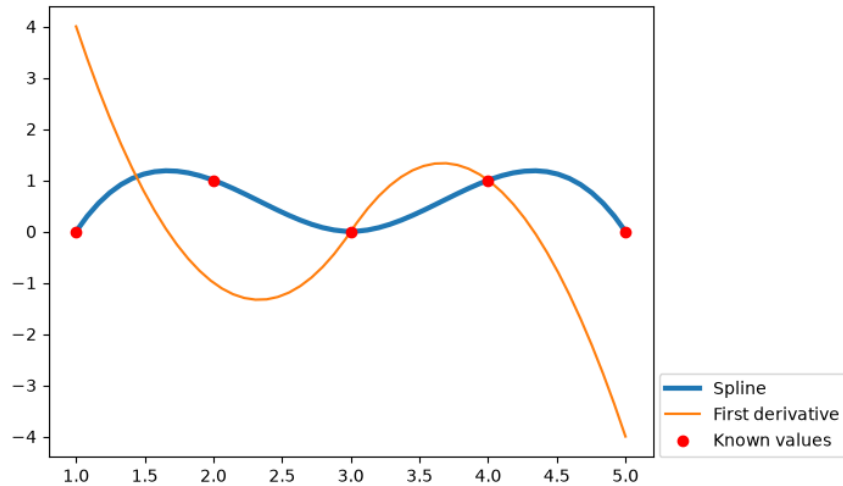
7.6.1 Differentiating, integrating ... splines

Splines can be differentiated and integrated, since they are polynomials. The `InterpolatedUnivariateSpline` object is thus already prepared for differentiation, integration, finding roots (zeros), knots ... (see the [documentation](#)).

For the first derivative of the spline in the object `spl`, for example, we use the method `spl.derivative(1)`, which returns a new spline object (its order is now lower by 1):

```
| spl1 = spl.derivative(1)
| #spl2 = spl.derivative(2)
| #spl3 = spl.derivative(3)
| plt.plot(xint, spl(xint), lw=3, label='Spline')
```

```
plt.plot(xint, spl1(xint), label='First derivative')
#plt.plot(xint, spl2(xint), label='Second derivative')
#plt.plot(xint, spl3(xint), label='Third derivative')
plt.plot(x, y, 'ro', label='Known values')
#plt.plot(spl.get_knots(), spl(spl.get_knots()), 'k.', label='B-spline knots')
plt.legend(loc=(1.01, 0));
```



Chapter 8

Approximation (curve fitting)

8.1 Introduction

In engineering practice we often encounter a table of data that may be affected by measurement or numerical errors.

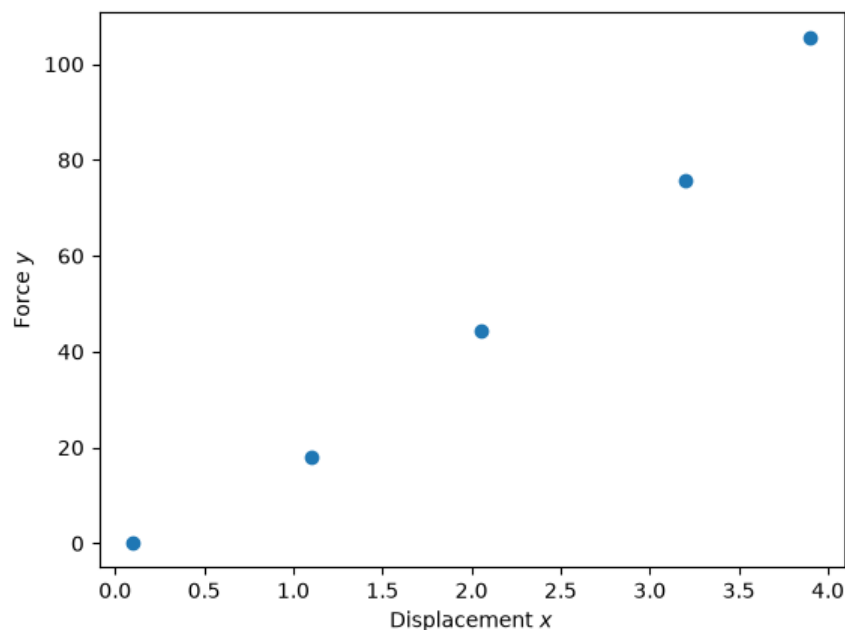
Let us look at an example of measuring a (linear) spring (x is the displacement, y is the measured force):

```
import numpy as np

x = np.array([0.1, 1.1, 2.05, 3.2, 3.9 ])
y = np.array([0.22, 18.15, 44.33, 75.59, 105.63])
```

Let us look at the data in a figure; first we import the required packages:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(x, y, 'o');
plt.xlabel('Displacement  $x$ ');
plt.ylabel('Force  $y$ ');
```



For this particular example, given our knowledge of the physical background of a linear spring, it would be appropriate to try to describe the measurements with a linear function:

$$f(x) = a_0 x + a_1 \tag{8.1}$$

We know a table of n data points x_i, y_i for $i = 0, 1, \dots, n-1$; there are more of them than we need to determine the two constants a_0 and a_1 , so we have an overdetermined system of linear equations:

$$y_i = a_0 x_i + a_1 \quad \text{za} \quad i = 0, 1, \dots, n-1. \quad (8.2)$$

We are looking for values of the constants a_0 and a_1 such that the function $f(x)$ *best* matches y_i at the known points x_i .

So first we need a criterion for the *best* match.

For the values x_i, y_i from the table we could look for values of a_0 and a_1 at which the sum of the absolute values of the deviations P is the smallest:

$$P(a_0, a_1) = \sum_{i=0}^{n-1} |y_i - (a_0 x_i + a_1)|. \quad (8.3)$$

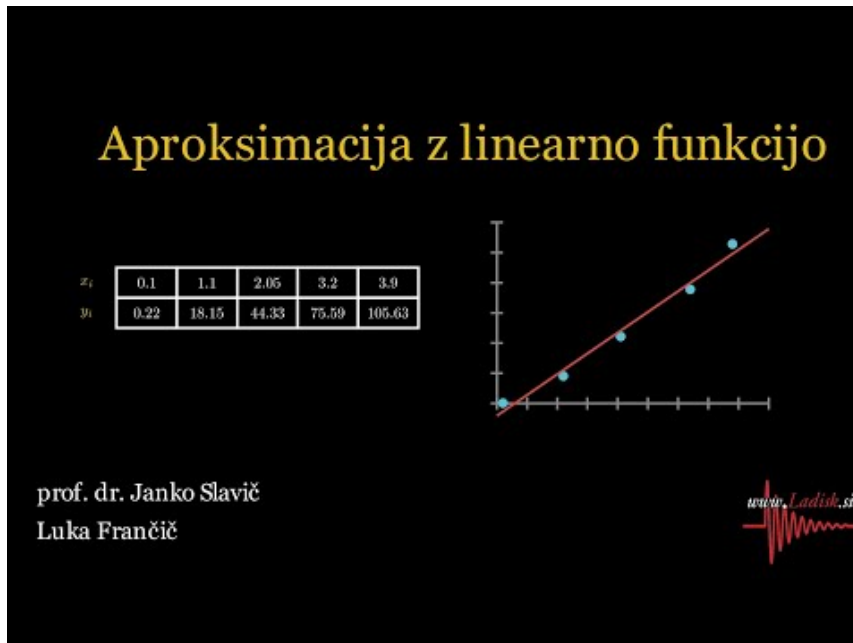
However, since such a function $P(a_0, a_1)$ is not continuously differentiable, we prefer to use the **least-squares method**:

$$S(a_0, a_1) = \sum_{i=0}^{n-1} (y_i - (a_0 x_i + a_1))^2. \quad (8.4)$$

Such a function $S(a_0, a_1)$ is continuous and continuously differentiable. By taking partial derivatives with respect to the parameters a_0 and a_1 we can find a stationary point (the partial derivatives are equal to 0). We will go through the procedure for a linear function in the next section.

8.2 Least-squares method for a linear function

```
from IPython.display import YouTubeVideo
YouTubeVideo('8 -oixY6GG9E', width=800, height=300)
```



We need to find the constants a_0, a_1 such that the sum of the squares of the differences between the function and the tabulated value (x_i, y_i , where $i = 0, 1, \dots, n-1$ and n is the number of tabulated data points):

$$S(a_0, a_1) = \sum_{i=0}^{n-1} (y_i - (a_0 x_i + a_1))^2 \quad (8.5)$$

is the smallest. The value will be smallest at the stationary point, which we determine by taking the partial derivative with respect to the parameters a_0 and a_1 .

First we take the partial derivative with respect to the parameter a_0 :

$$\frac{\partial S(a_0, a_1)}{\partial a_0} = 2 \sum_{i=0}^{n-1} (y_i - a_0 x_i - a_1) (-x_i) \quad (8.6)$$

We rearrange the expression:

$$\frac{\partial S(a_0, a_1)}{\partial a_0} = -2 \left(\sum_{i=0}^{n-1} y_i x_i - a_0 \sum_{i=0}^{n-1} x_i^2 - a_1 \sum_{i=0}^{n-1} x_i \right) \quad (8.7)$$

We proceed similarly for a_1 :

$$\frac{\partial S(a_0, a_1)}{\partial a_1} = 2 \sum_{i=0}^{n-1} (y_i - a_0 x_i - a_1) (-1) \quad (8.8)$$

$$\frac{\partial S(a_0, a_1)}{\partial a_1} = -2 \left(\sum_{i=0}^{n-1} y_i - a_0 \sum_{i=0}^{n-1} x_i - a_1 \sum_{i=0}^{n-1} 1 \right) \quad (8.9)$$

Since at the stationary point $\partial S(a_0, a_1)/\partial a_0 = 0$ and $\partial S(a_0, a_1)/\partial a_1 = 0$, from the expressions above we derive:

$$a_0 \sum_{i=0}^{n-1} x_i^2 + a_1 \sum_{i=0}^{n-1} x_i = \sum_{i=0}^{n-1} y_i x_i \quad (8.10)$$

and

$$a_0 \sum_{i=0}^{n-1} x_i + a_1 n = \sum_{i=0}^{n-1} y_i. \quad (8.11)$$

We have obtained a system of two linear equations for the unknowns a_0 and a_1 , which we know how to solve. We call it the *normal system* (the number of equations equals the number of unknowns).

Let us write the normal system in matrix form:

```
A = [[np.sum(x**2), np.sum(x)], # coefficient matrix
      [np.sum(x), len(x)]]
b = [np.dot(y,x), np.sum(y)]    # vector of constants
A = np.asarray(A)
b = np.asarray(b)
print('A:', A)
print('b:', b)
```

```
A: [[30.8725 10.35 ]
     [10.35   5.   ]]
b: [764.7085 243.92 ]
```

Now we need to solve the linear system:

$$\mathbf{A} \mathbf{a} = \mathbf{b}, \quad (8.12)$$

Note: here we have written the vector of unknowns as $\mathbf{a} = (a_0, a_1)$.

We solve the system:

```
a0, a1 = np.linalg.solve(A, b)
a0, a1

(np.float64(27.497258679085512), np.float64(-8.135325465707014))
```

Let us also check the condition number:

```
np.linalg.cond(A)

np.float64(25.20071350870242)
```

Now we will look at the result. First we prepare a figure that will also contain information about the sum of squares of the deviation of $f(x_i)$ from the tabulated values y_i .

```
def figure(slope=a0, shift=a1):
    d=1
    def linear_f(x, a0, a1):
        return a0*x+a1
    def S(x, y, f):
        return np.sum(np.power(y -f,2))
    plt.plot(x,y,'.', label='Data table')
    linear_f1 = linear_f(x, slope, shift)
    linear_f1_LS = linear_f(x, a0, a1)
    plt.plot(x, linear_f1, ' - ', label='Selected parameters')
    plt.plot(x, linear_f1_LS, ' - ', label='Least -squares method')
    error = S(x, y, linear_f(x, slope, shift))
    error_change_in_a0_direction = (S(x, y, linear_f(x, slope+d, shift)) -error)/d
    error_change_in_a1_direction = (S(x, y, linear_f(x, slope, shift+d)) -error)/d
    title = f'S: {error:g}, \
        $\Delta S/\Delta a_0$: {error_change_in_a0_direction:g}, \
        $\Delta S/\Delta a_1$: {error_change_in_a1_direction:g}'
    plt.title(title)
    plt.legend()
    plt.ylim(-10,110)
    plt.show()

from ipywidgets import interact
interact(figure, slope=(0, 50, 2), shift=(-10, 10, 1));

interactive(children=(IntSlider(value=27, description='slope', max=50, step=2),
    IntSlider(value=-8, descriptio...
```

8.2.1 Using the pseudo-inverse matrix

We can reach the same result using the pseudo-inverse matrix. We look for $y(x) = a_0 x + a_1$ and set up the overdetermined system $\mathbf{A} \mathbf{a} = \mathbf{y}$, where the coefficient matrix $\mathbf{A}$ is defined according to the values x_i ($i = 0, 1, \dots, n-1$):

```
A=np.array([x, np.ones_like(x)]).T
A

array([[0.1 , 1.  ],
       [1.1 , 1.  ],
       [2.05, 1.  ],
       [3.2 , 1.  ],
       [3.9 , 1.  ]])
```

We have denoted the vector of constants by $\mathbf{b}$, but in our case this is simply the vector of values $\mathbf{y}$ with elements y_i ($i = 0, 1, 2, \dots, n-1$):

```
| y
| array([ 0.22, 18.15, 44.33, 75.59, 105.63])
```

Let us check the rank of the coefficient matrix:

```
| np.linalg.matrix_rank(A)
| np.int64(2)
```

And also the rank of the augmented matrix:

```
| Ab = np.hstack((A,np.array([y]).T))
| Ab
| array([[1.0000e -01, 1.0000e+00, 2.2000e -01],
|       [1.1000e+00, 1.0000e+00, 1.8150e+01],
|       [2.0500e+00, 1.0000e+00, 4.4330e+01],
|       [3.2000e+00, 1.0000e+00, 7.5590e+01],
|       [3.9000e+00, 1.0000e+00, 1.0563e+02]])
| np.linalg.matrix_rank(Ab)
| np.int64(3)
```

Since we are solving a system of m linear equations with n unknowns, and $m > n$ holds and the rank of the augmented matrix is $n + 1$, we have an overdetermined system.

We determine the vector of unknown parameters $\mathbf{a}$ using the pseudo-inverse matrix:

$$\mathbf{a} = \mathbf{A}^+ \mathbf{y} \quad (8.13)$$

```
| np.linalg.pinv(A).dot(y)
| array([27.49725868, -8.13532547])
```

8.3 Least-squares method for an arbitrary polynomial

We will generalize the linear approximation presented above to an arbitrary polynomial of degree m :

$$f(a_0, a_1, \dots, a_m, x) = \sum_{s=0}^m a_s \underbrace{x^{m-s}}_{f_s(x)}, \quad (8.14)$$

where $f_s(x) = x^{m-s}$ is called a basis function ($s = 0, 1, 2, \dots, m$).

Let the data table be defined by x_i, y_i , where $i = 0, 1, 2, \dots, n-1$.

Note: for compactness of notation we will write the constants a in vector form $\mathbf{a} = [a_0, a_1, \dots, a_m]$.

We use the least-squares method:

$$S(\mathbf{a}) = \sum_{i=0}^{n-1} (y_i - f(\mathbf{a}, x_i))^2 = \sum_{i=0}^{n-1} \left(y_i - \sum_{s=0}^m a_s x_i^{m-s} \right)^2. \quad (8.15)$$

The necessary condition for an extremum of a function of $m+1$ independent variables is that we find a stationary point for each a_v , so we look for $\partial S(\mathbf{a}) / \partial a_v = 0$ (instead of s we have used the index v).

First we determine the partial derivative for the chosen a_v :

$$\frac{\partial S(\mathbf{a})}{\partial a_v} = \sum_{i=0}^{n-1} -2 \left(y_i - \sum_{s=0}^m a_s x_i^{m-s} \right) x_i^{m-v} \quad (8.16)$$

Note: $\frac{\partial}{\partial a_v} \left(\sum_{s=0}^m a_s x_i^{m-s} \right) = x_i^{m-v}$.

Since the partial derivative at the stationary point is equal to 0, we rearrange the expression above:

$$\sum_{i=0}^{n-1} \left(\sum_{s=0}^m a_s x_i^{m-s} \right) x_i^{m-v} = \sum_{i=0}^{n-1} y_i x_i^{m-v} \quad (8.17)$$

We rearrange the expression:

$$\sum_{i=0}^{n-1} \sum_{s=0}^m a_s x_i^{2m-s-v} = \sum_{i=0}^{n-1} y_i x_i^{m-v} \quad (8.18)$$

We swap the order of summation and derive:

$$\sum_{s=0}^m \left(a_s \sum_{i=0}^{n-1} x_i^{2m-s-v} \right) = \sum_{i=0}^{n-1} y_i x_i^{m-v} \quad \text{za: } v = 0, 1, \dots, m \quad (8.19)$$

We have derived equation v of the system of $m + 1$ linear equations:

$$\mathbf{A} \mathbf{a} = \mathbf{b} \quad (8.20)$$

The element $A_{v,s}$ of the coefficient matrix is:

$$A_{v,s} = \sum_{i=0}^{n-1} x_i^{2m-v-s}, \quad (8.21)$$

The element of the vector of constants is:

$$b_v = \sum_{i=0}^{n-1} y_i x_i^{m-v} \quad (8.22)$$

8.3.1 Numerical example

Let us use the data from the first problem and try to approximate it with a 2nd-degree polynomial ($m = 2$).

The data table is:

```
| x
| array([0.1 , 1.1 , 2.05, 3.2 , 3.9 ])
| y
| array([ 0.22, 18.15, 44.33, 75.59, 105.63])
```

Let us compute the coefficient matrix:

$$A_{v,s} = \sum_{i=0}^{n-1} x_i^{2m-v-s} \quad (8.23)$$

```

m = 2 #degree
A = np.zeros((m+1,m+1))
for v in range(m+1):
    for s in range(m+1):
        A[v,s] = np.sum(x**(2*m -v -s))
A

array([[355.32690625, 102.034125 , 30.8725   ],
       [102.034125 , 30.8725   , 10.35      ],
       [ 30.8725   , 10.35      , 5.         ]])

```

Let us also compute the vector of constants:

$$b_v = \sum_{i=0}^{n-1} y_i x_i^{m-v} \quad (8.24)$$

```

b = np.zeros(m+1)
for v in range(m+1):
    b[v] = np.dot(y,x**(m -v))
b

array([2588.934425, 764.7085 , 243.92   ])

```

Let us check the condition number:

```

np.linalg.cond(A)

np.float64(963.2125856290679)

```

We solve the system:

```

a = np.linalg.solve(A, b)
a

array([ 3.17760897, 14.67380042, -1.21091344])

```

Given the definition of the approximating polynomial:

$$f(a_0, a_1, \dots, a_m, x) = \sum_{v=0}^m a_v x^{m-v} \quad (8.25)$$

which in this particular case is the approximating polynomial:

$$f(x) = 3.18393375 x^2 + 14.64106847 x - 1.22621065 \quad (8.26)$$

Let us define a numerical implementation:

```

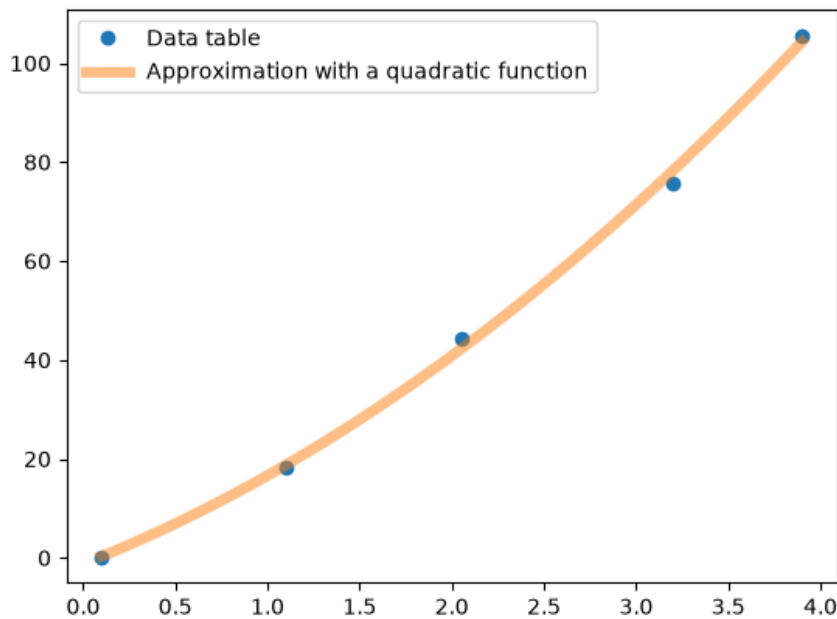
def apr_polynomial(x, a):
    """Returns the values of the approximating polynomial

    :param x: values at which we compute the approximated result
    :param a: coefficients of the approximating polynomial
    """
    m = len(a) - 1
    return np.sum(np.asarray([_x**(m -v) for v,_ in enumerate(a)]), axis=0)

```

We plot:

```
x_g = np.linspace(np.min(x), np.max(x), 100) # more points for plotting
plt.plot(x, y, 'o', label='Data table')
plt.plot(x_g, apr_polynomial(x_g, a), lw=5, alpha=0.5, label='Approximation with a
quadratic function')
plt.legend();
```



Let us also look at the approximation error:

$$e_i = y_i - f(x_i) \quad (8.27)$$

for $i = 0, 1, 2, \dots, n - 1$.

With a correctly performed approximation, some e_i are positive and some negative. Let us check whether this is true in our case:

```
e = y - apr_polynomial(x, a)
e
array([-0.06824269, -0.62517387,  2.1057209 , -2.69396374,  1.28165939])
```

Note: the higher the degree of the polynomial we use, the greater the probability of poor conditioning. For this reason we do not overdo the polynomial degree (in practice we mainly use low degrees)!

8.3.2 Using numpy for polynomial approximation

Let us look at how to use the `numpy` library for polynomial approximation.

First we use the function `numpy.polyfit` ([documentation](#)):

```
polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)
```

which requires three parameters: `x` and `y` represent the data table (also possible as lists of vectors), and `deg` is the polynomial degree. The remaining parameters are optional (e.g., `w` to use weights in the approximation).

The `polyfit` function returns a list of polynomial coefficients (highest order first); the result can also be a list of lists (if the input data is a list of vectors).

Note: `numpy.polyfit` and `numpy.poly1d` belong to the older interface for polynomials; for new programs `numpy` recommends the class `numpy.polynomial.Polynomial` ([documentation](#)): `p = Polynomial.fit(x, y, deg=2)` returns an object that we call as a function (`p(x)`), and

`p.convert().coef` gives the coefficients in **increasing** order of powers (exactly the opposite of `polyfit`).

Let us look at its use for the example treated previously:

```
| coef = np.polyfit(x, y, deg=2)
| coef
|
| array([ 3.17760897, 14.67380042, -1.21091344])
|
| a # result of our own implementation
|
| array([ 3.17760897, 14.67380042, -1.21091344])
```

Once we have the coefficients, we can create a polynomial object by calling `numpy.poly1d` ([documentation](#)):

```
| poly1d(c_or_r, r=False, variable=None)
```

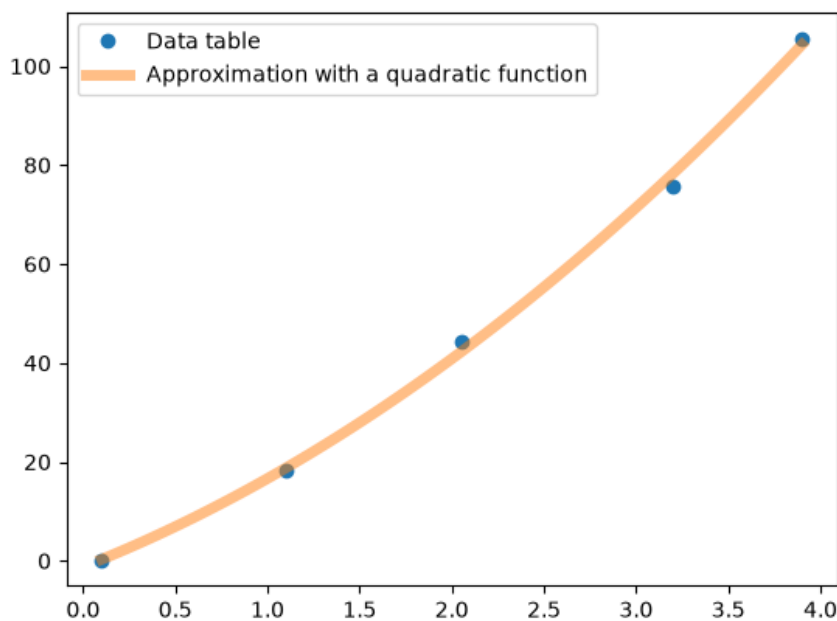
where `c_or_r` represents the list of polynomial coefficients, or the roots of the polynomial if `r=True`. The function returns an instance of the object, by calling which we can compute the values of the approximating polynomial at `x`, but we can also compute other things, such as the roots of the polynomial.

Let us look at an example:

```
| p = np.poly1d(coef) # p is an instance of the poly1d object. We now obtain the
| approximation with y = p(x).
```

Let us plot the values:

```
| plt.plot(x, y, 'o', label='Data table')
| plt.plot(x_g, p(x_g), lw=5, alpha=0.5, label='Approximation with a quadratic
| function')
| plt.legend();
```



Let us compute the roots of the polynomial:

```
| p.roots
|
| array([-4.69897273,  0.08109792])
```

8.4 Approximation with an arbitrary function

In approximation we are not limited to polynomials only. We can approximate data tables:

- with a linear combination of linearly independent basis functions, or
- with a function in which the parameters appear in a nonlinear relationship (e.g., $a_0 \sin(a_1 x + a_2)$).

For details, see the reference J. Petrišič: Uvod v Matlab za inženirje, Fakulteta za strojništvo 2013, p. 145.

We will focus on using the `scipy` package for approximation with a nonlinear function, which is based on the least-squares method.

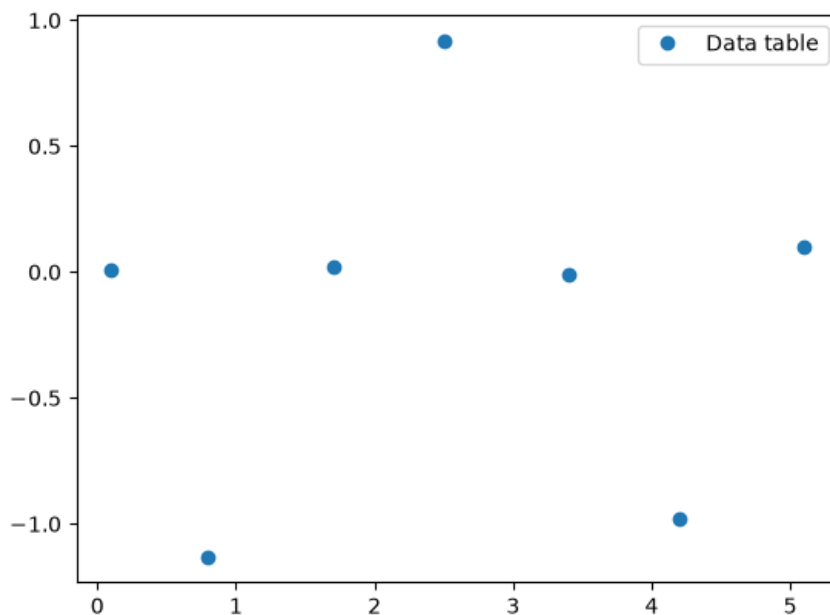
8.4.1 Approximation with a harmonic function

The data table is defined as:

```
x = np.array([ 0.1, 0.8, 1.7, 2.5, 3.4, 4.2, 5.1])
y = np.array([ 0.01, -1.13, 0.02, 0.92, -0.01, -0.98, 0.1])
```

Let us plot the data table:

```
plt.plot(x, y, 'o', label='Data table')
plt.legend();
```



We will perform the approximation with a nonlinear function using `scipy.optimize.curve_fit` ([documentation](#)):

```
curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False,
check_finite=True, bounds=(-inf, inf), method=None, jac=None, **kwargs)
```

which requires three parameters: `f` represents the definition of the Python function with which we want to approximate, and whose parameters we vary using the least-squares method. `xdata` and `ydata` represent the data table. It is also recommended to define an estimate of the sought parameters `p0`. The remaining parameters are optional.

The function returns two arrays: `popt`, which represents the parameters found, and `pcov`, which represents the estimated covariance of `popt`.

First let us define a Python function whose first parameter is the independent variable `x`, followed by the parameters we want to determine:

```
def func(x, A, \omega, \phi):
    return A*np.sin( \omega*x+ \phi)
```

where A is the amplitude, ω the angular frequency, and ϕ the phase of the harmonic function. With the help of the figure we can guess the initial estimates: $A=1$, $\omega=1$, $\phi=0$

Now we import `curve_fit` and run the optimization procedure:

```
from scipy.optimize import curve_fit

popt, pcov = curve_fit(func, x, y, p0=[1, 1, 0])
popt

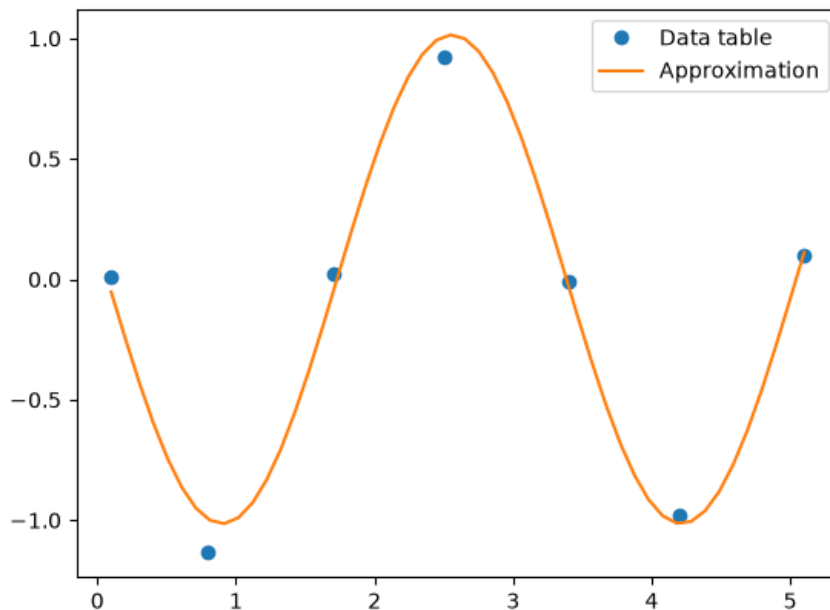
array([ 1.01461945,  1.89652046, -3.27943097])
```

We have computed the expected values (see above).

```
func(x, *popt)

array([ -0.0525477 , -0.99608783, -0.05612665,  1.00860622, -0.02753942,
        -1.01426499,  0.1110182 ])

x_g = np.linspace(np.min(x), np.max(x), 50)
y_g = func(x_g, *popt) # note how we passed the parameters to the function
plt.plot(x, y, 'o', label='Data table')
plt.plot(x_g, y_g, label='Approximation')
plt.legend();
```



8.5 Additional

Make an `.exe` of your program:

- <https://pypi.org/project/py2exe/>
- <http://www.pyinstaller.org/>



Take a look at the [pandas package](#).

8.5.1 Approximation with splines using SciPy

Let the data table be:

```
x = np.linspace( -3, 3, 20)
x

array([ -3.          , -2.68421053, -2.36842105, -2.05263158, -1.73684211,
       -1.42105263, -1.10526316, -0.78947368, -0.47368421, -0.15789474,
         0.15789474,  0.47368421,  0.78947368,  1.10526316,  1.42105263,
         1.73684211,  2.05263158,  2.36842105,  2.68421053,  3.          ])

np.random.seed(0) # seed of the random number generator
y = np.exp( -x**2) + 0.1 * np.random.normal(scale=.5, size=len(x))
y

array([ 0.08832603,  0.02075073,  0.0526001 ,  0.12684217,  0.14234432,
         0.08387244,  0.34226064,  0.52862107,  0.79385312,  0.99590738,
         0.98257964,  0.87172774,  0.57424082,  0.30083997,  0.15492949,
         0.06565014,  0.08950146, -0.00659471,  0.01639626, -0.04258138])
```

Let us look at the object `scipy.interpolate.UnivariateSpline` ([documentation](#)), which enables both interpolation and approximation with splines:

```
UnivariateSpline(x, y, w=None, bbox=[None, None], k=3, s=None, ext=0,
check_finite=False)
```

The parameters `x` and `y` represent the data table.

Note: `UnivariateSpline` has been marked as *legacy* since SciPy 1.15; the more modern equivalent for approximation (smoothing) with splines is `scipy.interpolate.make_smoothing_spline` ([documentation](#)) or `make_splrep`. The object interface below still works and is clearer for learning.

The optional parameter `s` sets the value that the sum of squares of the differences between the approximating spline and the approximation points must not exceed:

```
sum((w[i] * (y[i] -spl(x[i])))**2, axis=0) <= s
```

`w` are the weights of the individual points.

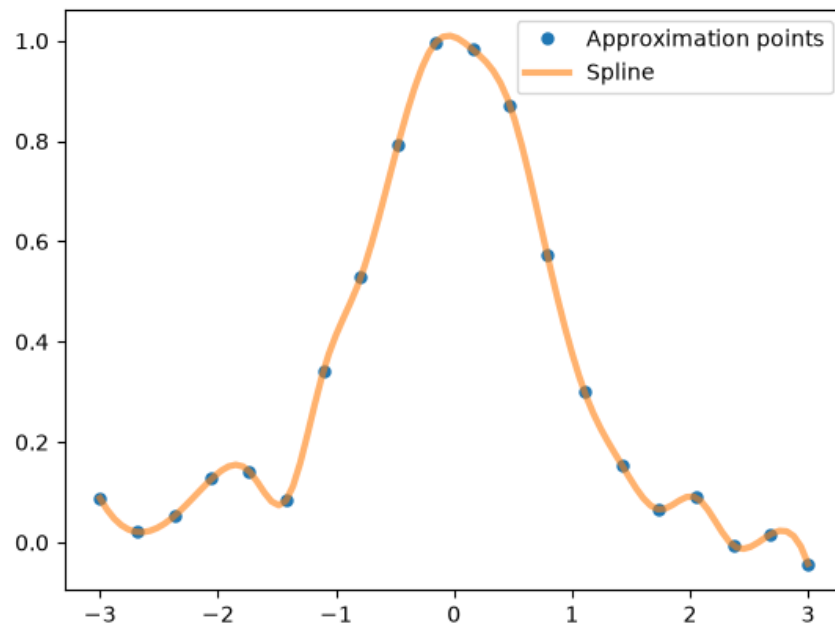
If we set `s=0`, we require interpolation.

The parameter `k` defines the degree of the polynomial spline (the default is `k=3`).

We perform approximation with splines by defining, in addition to the data table `x` and `y`, the parameter `s` as well. Let us perform interpolation:

```
from scipy.interpolate import UnivariateSpline
spl = UnivariateSpline(x, y, s=0.)
```

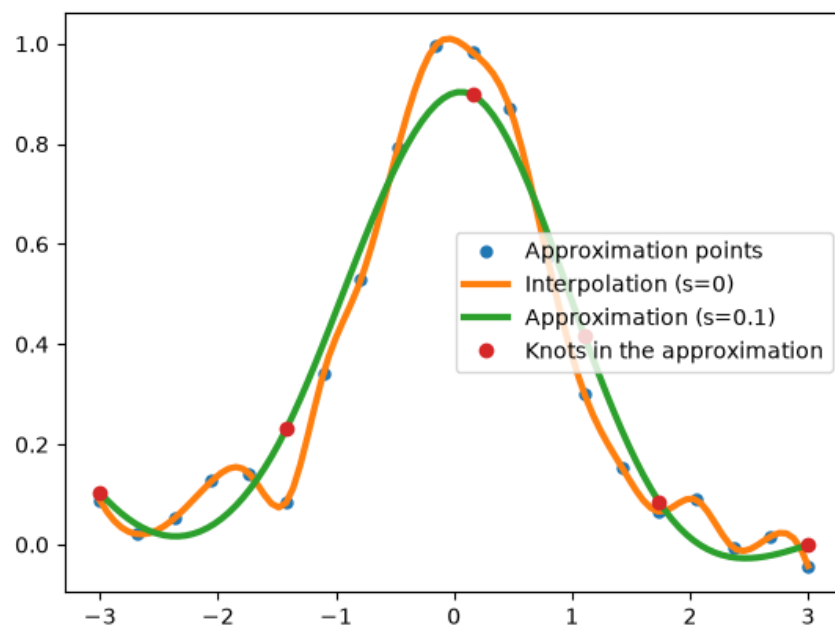
```
x_g = np.linspace(-3, 3, 100)
plt.plot(x, y, 'o', ms=5, label='Approximation points')
plt.plot(x_g, spl(x_g), lw=3, label='Spline', alpha=0.6)
plt.legend();
```



Let us also perform the approximation:

```
spl_a = UnivariateSpline(x, y, s=.1)

plt.plot(x, y, 'o', ms=5, label='Approximation points')
plt.plot(x_g, spl(x_g), lw=3, label='Interpolation (s=0)');
plt.plot(x_g, spl_a(x_g), lw=3, label='Approximation (s=0.1)');
plt.plot(spl_a.get_knots(), spl_a(spl_a.get_knots()), 'o', label='Knots in the
approximation');
plt.legend();
```



The actual residual:

```
spl_a.get_residual()
```

| 0.10002662427563205

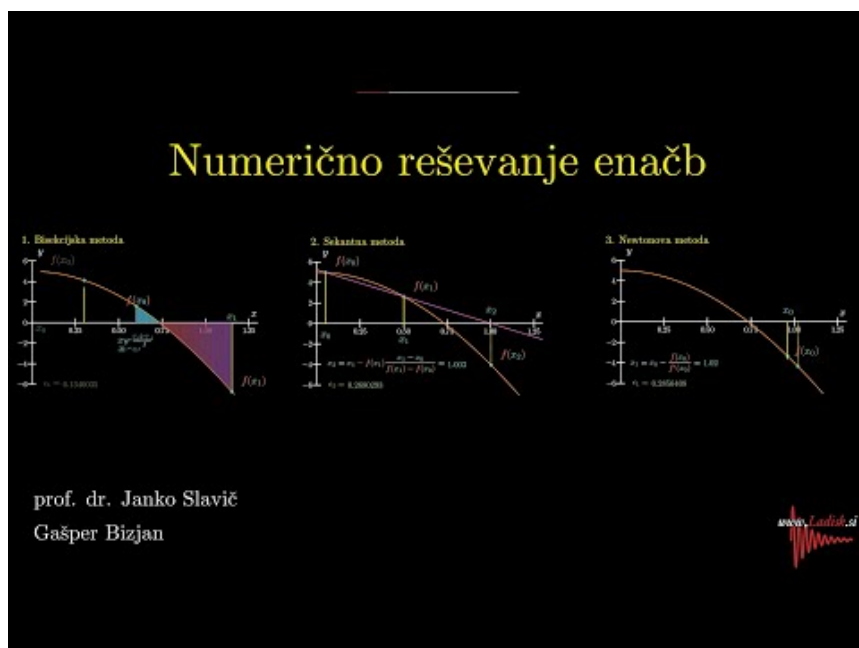
Chapter 9

Root finding

9.1 Introduction

As an introduction, let's watch the video below:

```
from IPython.display import YouTubeVideo
YouTubeVideo('17d55KE8SIU', width=800, height=300)
```



In root finding we consider an arbitrary equation that depends on the variable x and we look for the solution:

$$f(x) = 0. \quad (9.1)$$

The solutions of the equation are also called *roots*. A root of the equation $f(x) = 0$ is at the same time a zero of the function $y = f(x)$.

The function $y = f(x)$ can have zeros of order:

- first-order zero: the function crosses the horizontal axis at a nonzero angle,
- even-order zeros: the function touches the horizontal axis but does not cross it,
- odd-order zeros: the function crosses the horizontal axis; for a zero of order 3 or higher there is an inflection point (the tangent is parallel to the horizontal axis).

It is important to point out here that we are looking for the solution of an arbitrary equation $f(x) = 0$. While for linear, quadratic, or cubic equations we can determine analytical solutions, for most nonlinear equations an analytical solution cannot be found. For this reason numerical approaches are all the more important.

9.1.1 Restrictions on the function $f(x)$

We require that the function $y = f(x)$ be continuous on the closed interval $[x_0, x_1]$. When computing zeros, we will restrict ourselves only to first-order zeros.

9.1.2 Example

An arbitrary function $y = f(x)$ can be defined with a *Python function*; as an example, let's define a polynomial here:

```
def f(x):
    return x**3 - 10*x**2 + 5
```

Since this is the polynomial $x^3 - 10x^2 + 5$ with coefficients $[1, -10, 0, 5]$, it is better to define it using `np.poly1d`:

```
numpy.poly1d(c_or_r, r=False, variable=None)
```

where the parameters are:

- `c_or_r` the coefficients of the polynomial in order of decreasing power, or, if `r=True`, the roots of the polynomial,
- `r` is `False` by default, which means that the polynomial coefficients are given,
- `variable` the variable that is printed when using the `print()` function.

Note: `np.poly1d` belongs to the older interface; the more modern one is the class `numpy.polynomial.Polynomial` ([documentation](#)), which expects the coefficients in **increasing** order of powers: `Polynomial([5, 0, -10, 1])`. The object is likewise called as a function, and the zeros are obtained with the method `roots()`.

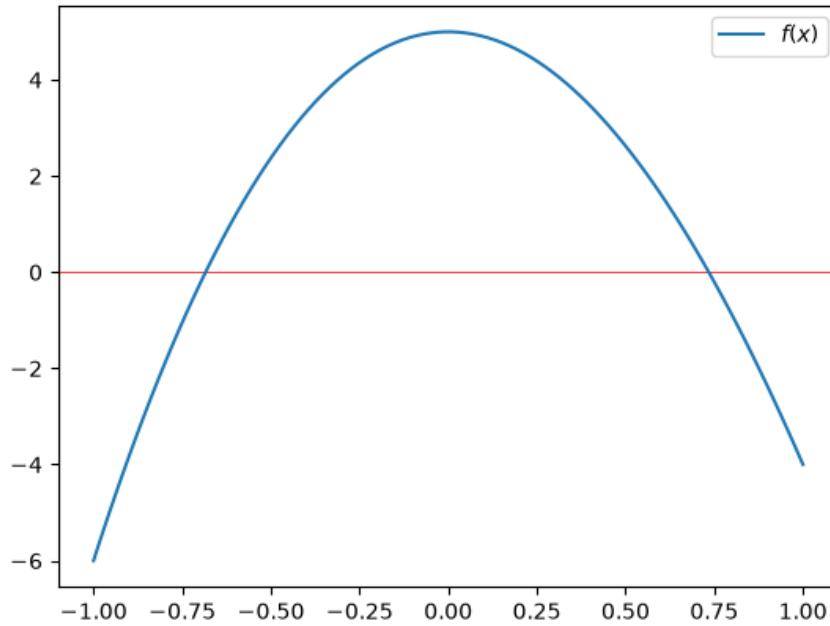
Let's import `numpy` and define the polynomial:

```
import numpy as np # import numpy
f = np.poly1d([1, -10, 0, 5]) # define the polynomial coefficients
print(f) # display the polynomial
```

```
3      2
1 x - 10 x + 5
```

Let's plot the function $f(x)$:

```
import matplotlib.pyplot as plt          # import matplotlib
%matplotlib inline
x_r = np.linspace(-1, 1, 100)
plt.axhline(0, color='r', lw=0.5)       # horizontal line,
plt.plot(x_r, f(x_r), label='$f(x)$')    # so that a zero is somewhere near $x = 0.7$.
plt.legend();
```



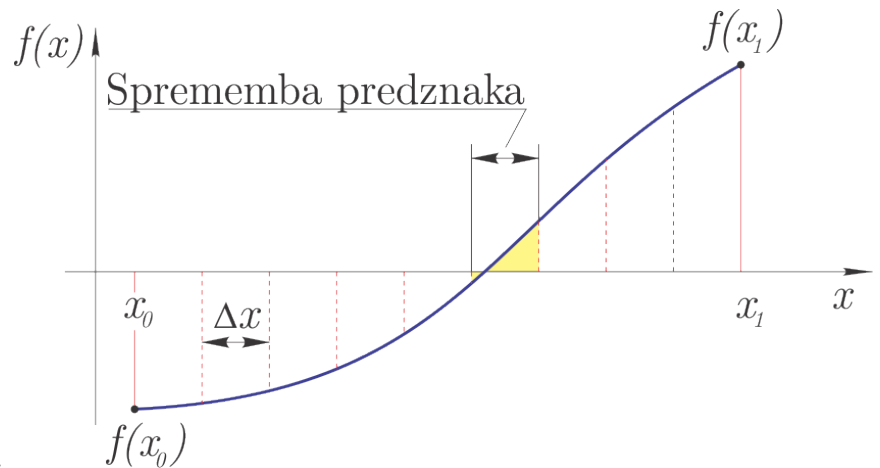
We notice that the zeros of the function $f(x)$ are near -0.7 and $+0.7$. The `poly1d` object has an attribute `roots`, or also `r` (see the documentation), which returns these zeros:

```
| f.r
|
| array([ 9.94949106,  0.73460351, -0.68409457])
```

In what follows, our goal will be to numerically determine a zero for an arbitrary function `f`.

9.2 Incremental method

Incremental solving is based on the idea that if the function $f(x)$ has different signs at x_0 and x_1 , then there is at least one zero in between. We therefore divide the closed interval $[x_0, x_1]$ into segments of width Δx ; on the segment where we observe a sign change, there is at least one zero of the function.



The method is shown in the figure.

For a zero we require:

$$|x_{i+1} - x_i| < \varepsilon \quad \text{in} \quad |f(x_{i+1})| + |f(x_i)| < D, \quad (9.2)$$

where ε is the required accuracy of the solution and D is a chosen small value that prevents us from identifying a pole as a zero (which is otherwise not possible because of the continuity condition).

The incremental method has some drawbacks:

- it is very slow,
- it can miss two zeros that are very close to each other,
- it does not detect multiple even-order zeros (a local extremum that only touches the horizontal axis).

The incremental method belongs to the so-called *bracketed* methods, since it looks for zeros of the function only on the interval $[x_0, x_1]$. Later we will also learn about *open* methods, which can converge to a zero outside the given interval.

Because of all the drawbacks listed above, the incremental method is often used only to compute an initial estimate of a zero.

9.2.1 Numerical implementation

Let's now look at the incremental search for zeros of a function:

```
def incremental(fun, x0, x1, dx):
    """ Returns the first interval (x1, x2) where a zero lies

    :param fun: function whose zeros we are looking for
    :param x1: lower search bound
    :param x2: upper search bound
    :param dx: search increment
    """
    x_d = np.arange(x0, x1, dx) # prepare the x values
    f_d = np.sign(fun(x_d))      # prepare the signs of the function
    f_d = f_d[1:]*f_d[: -1]      # multiply neighboring elements
    i = np.argmin(f_d)           # first crossing through zero
    # sum of abs function values
    x0 = x_d[i]
    x1 = x_d[i+1]
    D = np.abs(fun(x0)) + np.abs(fun(x1))
    return np.asarray([x0, x1]), D
```

Let's now look at the usage on the polynomial defined above:

```
incr_result, D = incremental(f, 0., 1., 0.001)
incr_result

array([0.734, 0.735])
```

The zero is isolated with an accuracy of 0.001; let's also check the sum of the absolute function values:

```
D

np.float64(0.013071529000000304)
```

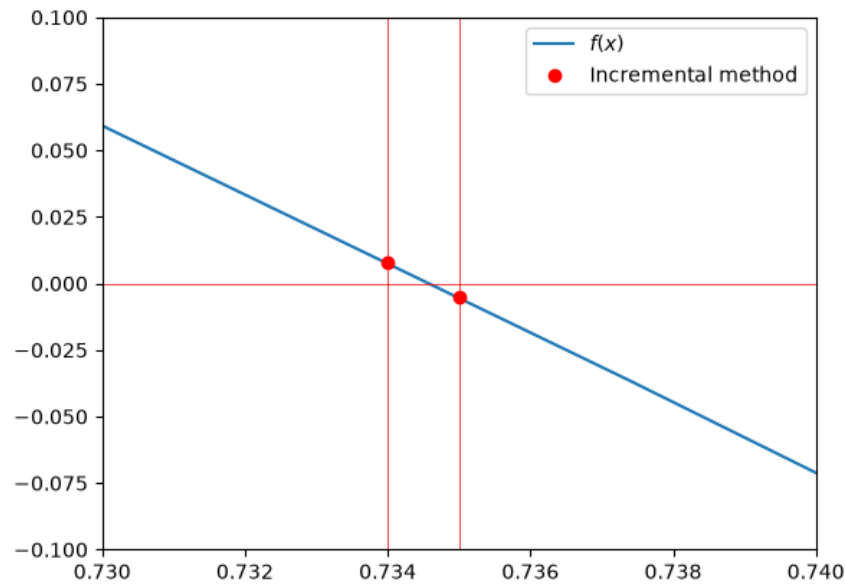
We find that it is relatively small; with the following methods we will, however, try to substantially improve the result.

Let's prepare the figure:

```
def fig():
    plt.plot(x_r, f(x_r), label='$f(x)$')
    plt.axhline(0, color='r', lw=0.5) # horizontal line
    plt.axvline(incr_result[0], color='r', lw=0.5) # vertical line
    plt.axvline(incr_result[1], color='r', lw=0.5) # vertical line
    plt.plot(incr_result, f(incr_result), 'ro', label='Incremental method')
    plt.xlim(0.73, 0.74)
    plt.ylim(-0.1, 0.1)
    plt.legend();
```

Let's display the result:

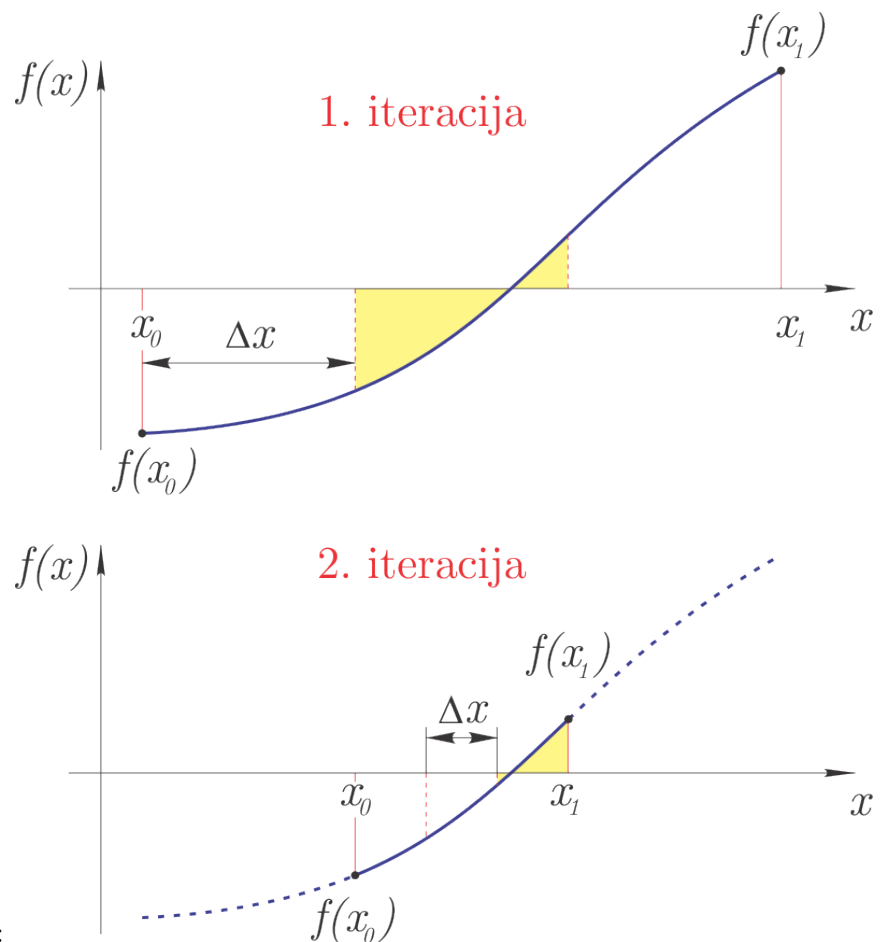
```
| fig()
```



So in order to compute the solution on the interval $[0, 1]$ with an accuracy of $\Delta x = 0.001$, we had to call the function $f(x)$ 1000 times. This is a very inefficient method, so we will look for better approaches; first with a simple iterative incremental approach.

9.3 Iterative incremental method

In the first iteration, the iterative incremental method uses the incremental method to bracket the search interval for zeros at a relatively large step. The interval found in the first iteration is divided into smaller intervals in the second iteration, and the incremental search for the zero is repeated. The third iteration then restricts itself to the interval determined in the second, and so on. We stop iterating once we have reached the prescribed accuracy of the solution ϵ .



The method is shown in the figure:

9.3.1 Numerical implementation

```
def incremental_super(fun, x0, x1, iterations=3):
    """ Returns the interval (x0, x1) where a zero lies

    :param fun: function whose zero we are looking for
    :param x0: lower search bound
    :param x1: upper search bound
    :iterations: number of iterations of the incremental method
    """
    for i in range(iterations):
        dx = (x1 - x0)/10
        x0x1, _ = incremental(fun, x0, x1, dx)
        x0, x1 = x0x1
    # sum of abs function values
    D = np.abs(fun(x0)) + np.abs(fun(x1))
    return np.asarray([x0, x1]), D
```

With 30 calls to the function $f(x)$ we thus obtain accuracy similar to the previous 1000:

```
result30, D30 = incremental_super(f, 0., 1., iterations=3)
result30

array([0.734, 0.735])

incr_result

array([0.734, 0.735])
```

Of course, we can substantially improve the accuracy by increasing the number of iterations:

```
result80, D80 = incremental_super(f, 0., 1., iterations=8)
result80
array([0.7346035 , 0.73460351])
```

Let's also check the criterion of the sum of the absolute function values, which must be small:

```
D80
np.float64(1.3073143190212022e -07)
```

9.4 Bisection method

On the interval $[x_0, x_1]$, where we know that a zero of the function exists (the signs of $f(x_0)$ and $f(x_1)$ differ), we can use the *bisection method*.

The idea of the method is:

- divide the interval $[x_0, x_1]$ in half (hence the name: *bi-section*): $x_2 = (x_0 + x_1)/2$,
- if $f(x_0)$ and $f(x_2)$ have different signs, the new search interval for the zero is $[x_0, x_2]$, otherwise: $[x_2, x_1]$,
- based on the previous step, we define a new closed interval $[x_0, x_1]$ and continue the iterative procedure until we reach the desired accuracy $|x_1 - x_0| < \varepsilon$.

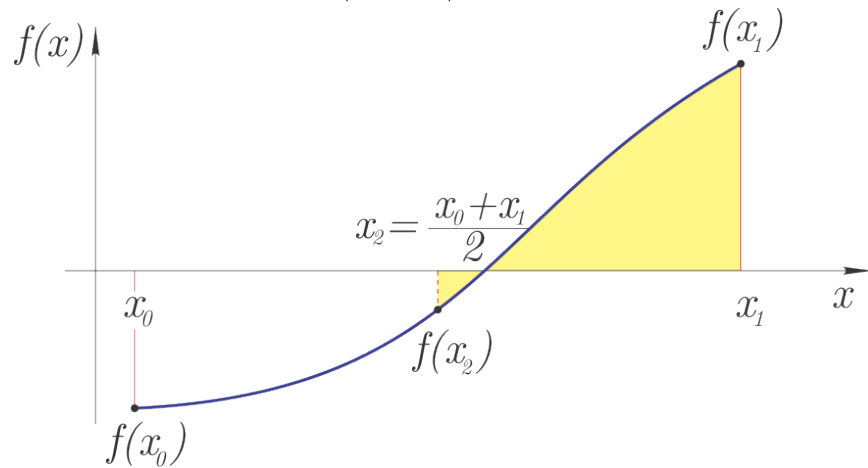


Figure of the method:

The bisection method belongs to the *bracketed* methods, which return a zero of the function on the given interval $[x_0, x_1]$.

9.4.1 Error estimate

If we start with the interval $\Delta x = |x_1 - x_0|$, then the accuracy of the bisection method after the first bisection step is:

$$\varepsilon_1 = \Delta x/2, \quad (9.3)$$

after the second step:

$$\varepsilon_2 = \Delta x/2^2 \quad (9.4)$$

and after n steps:

$$\varepsilon_n = \Delta x/2^n. \quad (9.5)$$

Usually we require that the solution be given with an accuracy ε , and from the equation above we can derive the number of bisection steps required:

$$n = \frac{\log\left(\frac{\Delta x}{\varepsilon}\right)}{\log(2)}. \quad (9.6)$$

Of course, the number of steps is an integer.

9.4.2 Numerical implementation

```
def bisection(fun, x0, x1, tol=1e-3, Dtol=1e-1, verbose=True):
    """ Returns a zero with accuracy tol

    :param fun: function whose zero we are looking for
    :param x0: lower search bound
    :param x1: upper search bound
    :param tol: required accuracy
    :param Dtol: maximum sum of absolute values of the solution
    :verbose: whether to print a short report at the end
    """
    if np.sign(fun(x0)) == np.sign(fun(x1)):
        raise Exception('Root is not bracketed.')
    n = np.ceil(np.log(np.abs(x1 - x0)/tol)/np.log(2)).astype(int) # number of
    iterations
    for i in range(n):
        x2 = (x0 + x1) / 2
        f1 = fun(x0)
        f3 = fun(x2)
        f2 = fun(x1)
        if np.sign(fun(x2)) != np.sign(fun(x0)):
            x1 = x2
        else:
            x0 = x2
    D = np.abs(fun(x0)) + np.abs(fun(x1))
    if D > Dtol:
        raise Exception('Likely a pole or multiple zeros.')
    r = (x0+x1)/2
    if verbose:
        decimals = int(np.log10(1/tol)) # does not always work for all cases:)
        print(f'Solution: {r:5.{decimals}f}, number of iterations: {n:g}, D:
              {D:5.5f}')
    return r
```

Now let's try to find the zero with an accuracy of $1e-3$:

```
bisection(f, 0, 1, tol=1e-3);
```

```
Solution: 0.735, number of iterations: 10, D: 0.01277
```

In ten iterations we obtained the same accurate result as above with the iterative incremental method `result30`. Let's also look at computing the zero with even greater accuracy:

```
bisection(f, 0, 1, tol=1e-6);
```

```
Solution: 0.734603, number of iterations: 20, D: 0.00001
```

We can check the execution speed with the so-called *magic function* `timeit` ([documentation](#)), which runs the function several times and analyzes the execution time. If the magic function is preceded by the double symbol `%%`, it executes and times the entire cell, but if only a single `%`, then only one line.

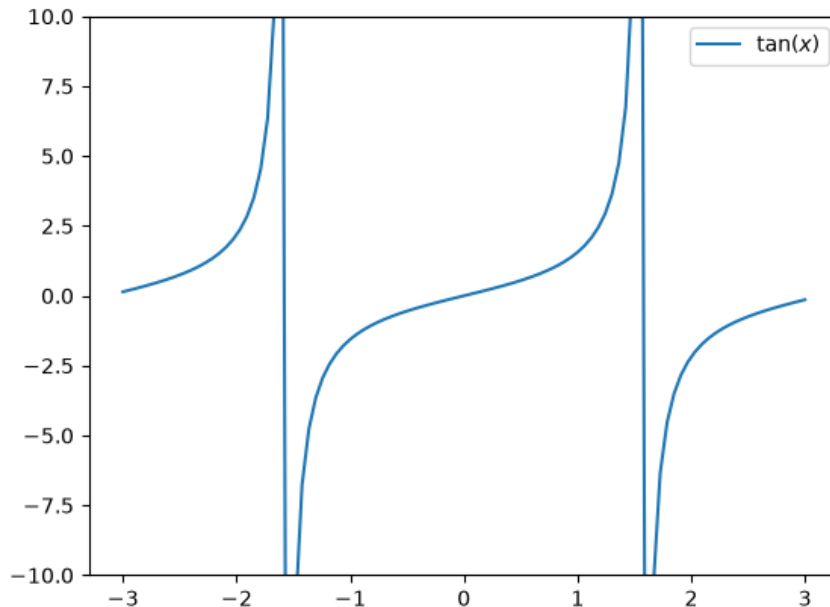
```
%%timeit
bisection(f, 0., 1., verbose=False)
```

```
348 \mus \pm 4.2 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

Searching for a zero near a pole

Let's now look at searching for a zero of the `tan` function near a pole (the zero does not actually exist):

```
x_t = np.linspace( -3, 3, 100)
plt.plot(x_t, np.tan(x_t), label='$\\tan(x)$')
plt.ylim( -10, 10)
plt.legend();
```



In the case of searching on the interval $[-1, 1]$ we find the correct zero:

```
| bisection(np.tan, -1, 1, tol=1e -3);
|
| Solution: -0.000, number of iterations: 11, D: 0.00098
```

In the case of searching near a pole, however, the program warns us about it (the function call is commented out here, otherwise the automatic generation is interrupted here):

```
| ## bisection(np.tan, -3, 0, tol=1e -6)
```

The bisection method built into `scipy` does not have such a check (for the sake of speed) and will return a result, which will, however, be wrong. So we must be careful when using it.

9.4.3 Using `scipy.optimize.root_scalar`

The bisection method is a *slow* but reliable method for finding zeros and is implemented within `scipy`. First we import it:

```
| from scipy.optimize import root_scalar
|
| root_scalar(f, args=(), method='bisect', bracket=[a,b], fprime=None, fprime2=None,
| x0=None, x1=None,xtol=None, rtol=None, maxiter=None, options=None)
```

To invoke the bisection method with the `root_scalar` function, we must pass three parameters to the function: the function `f` and the closed interval `[a, b]` (via the `bracket` parameter). The signs of `f(a)` and `f(b)` must be different. The other parameters, e.g. the absolute `xtol` and relative `rtol` error, and the maximum number of iterations `maxiter`, are optional - they have default values. For more, see the [documentation](#).

Let's look at the usage:

```
| root_scalar(f, bracket=[0, 1], method='bisect')
```

```
| converged: True
|         flag: converged
| function_calls: 41
|       iterations: 39
|         root: 0.7346035077880515
|       method: bisect
```

and the speed:

```
| %timeit root_scalar(f, bracket=[0, 1], method='bisect')
```

```
| 324 \mus \pm 2.83 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

We can also check the function $\tan(x)$, first on an interval where the function is continuous:

```
| root_scalar(np.tan, bracket=[ -1, 1], method='bisect')
```

```
| converged: True
|         flag: converged
| function_calls: 3
|       iterations: 1
|         root: 0.0
|       method: bisect
```

Then also near a pole, where it is not continuous:

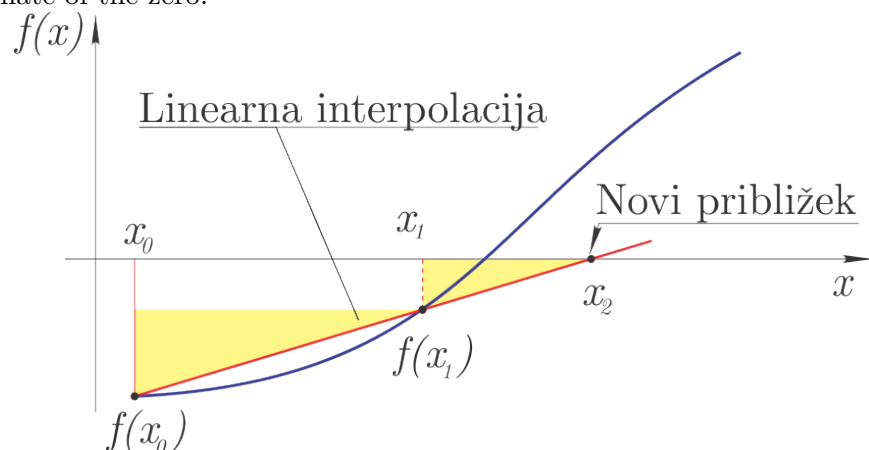
```
| root_scalar(np.tan, bracket=[ -3, -1], method='bisect')
```

```
| converged: True
|         flag: converged
| function_calls: 42
|       iterations: 40
|         root: -1.5707963267941523
|       method: bisect
```

which is a wrong solution!

9.5 Secant method

The secant method requires two initial estimates x_0 and x_1 and a function $f(x)$. Assuming linear interpolation between the points $x_0, f(x_0)$ and $x_1, f(x_1)$ (we draw a *secant* through the points, hence the name), x_2 is determined, where the linear interpolating function has a zero. x_2 represents the new estimate of the zero.



Based on the figure:

we can write (similar triangles are marked in yellow in the figure):

$$\frac{f(x_1)}{x_2 - x_1} = \frac{f(x_0) - f(x_1)}{x_1 - x_0}. \quad (9.7)$$

It follows that the new estimate of the zero is:

$$x_2 = x_1 - f(x_1) \frac{x_1 - x_0}{f(x_1) - f(x_0)}. \quad (9.8)$$

In the next step of the secant method we perform the following substitutions: $x_0 = x_1$ and $x_1 = x_2$.

The secant method belongs to the *open* methods, since it can find a zero of the function that lies outside the domain $[x_0, x_1]$.

9.5.1 Error estimate

Conservatively, we can estimate the error from the difference between two consecutive estimates:

$$\varepsilon = |x_{n-1} - x_n| \quad (9.9)$$

9.5.2 Convergence and order of convergence

Convergence means that the sequence of estimates converges to the solution of the equation α (α is the solution of the equation).

The order of convergence indicates the speed of convergence.

If we denote by ε the error of the estimate and reduce the error linearly with each iteration step (C is a constant):

$$\varepsilon_n = C \varepsilon_{n-1}^1, \quad (9.10)$$

we speak of **order of convergence 1** (ε has the power 1)!

In the previously discussed bisection method, we halve the error at each step ($\varepsilon_n/\varepsilon_{n-1} = C = 1/2$). *The bisection method has order of convergence 1.*

The order of convergence of the secant method is *higher* and can be estimated as:

$$\varepsilon_n = C \varepsilon_{n-1}^{1.618}. \quad (9.11)$$

From the estimate above it follows that at each iteration step the number of correct digits increases by about 60%. Since the order of convergence is higher than 1 and lower than quadratic, such convergence is called *superlinear* convergence.

9.5.3 Numerical implementation

```
def secant(fun, x0, x1, tol=1e-3, Dtol=1e-1, max_iter=50, verbose=True):
    """ Returns a zero with accuracy tol

    :param fun: function whose zero we are looking for
    :param x0: lower search bound
    :param x1: upper search bound
    :param tol: required accuracy
    :param max_iter: maximum number of iterations before execution is interrupted
    :param Dtol: maximum sum of absolute values of the solution
    :param verbose: whether to print a short report at the end
    """
    if np.sign(fun(x0)) == np.sign(fun(x1)):
        raise Exception('Root is not bracketed.')
    for i in range(max_iter):
        f0 = fun(x0)
```

```

f1 = fun(x1)
x2 = x1 - f1 * (x1 - x0)/(f1 - f0)
x0 = x1
x1 = x2
if verbose:
    print('{:g}. step: x0={:g}, x1={:g}'.format(i+1, x0, x1))
if np.abs(x1 - x0) < tol:
    r = (x0 + x1) / 2
    D = np.abs(fun(x0)) + np.abs(fun(x1))
    if D > Dtol:
        raise Exception('Likely a pole or multiple zeros.')
    r = (x0 + x1) / 2
    if verbose:
        decimals = int(np.log10(1/tol)) # does not always work for all
        cases:)
        print(f'Solution: {r:5.{decimals}f}, D: {D:5.5f}')
    return r

```

Let's look at the usage:

```
| secant(f, 0, 1., tol=1.e -8, verbose=True);
```

```

1. step: x0=1, x1=0.555556.
2. step: x0=0.555556, x1=0.707845.
3. step: x0=0.707845, x1=0.737957.
4. step: x0=0.737957, x1=0.734549.
5. step: x0=0.734549, x1=0.734603.
6. step: x0=0.734603, x1=0.734604.
7. step: x0=0.734604, x1=0.734604.
Solution: 0.73460351, D: 0.00000

```

and the speed

```
| %timeit secant(f, 0, 1., tol=1.e -8, verbose=False)
```

```
| 119 \mus \pm 389 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)
```

As we wrote above, the secant method is of the open type. The solution of the equation can be outside the given interval. Let's look at an example:

```
| secant(np.tan, 1, 2);
```

```

1. step: x0=2, x1=1.41615.
2. step: x0=1.41615, x1=1.85165.
3. step: x0=1.85165, x1=1.69887.
4. step: x0=1.69887, x1=1.97485.
5. step: x0=1.97485, x1=2.09379.
6. step: x0=2.09379, x1=2.43519.
7. step: x0=2.43519, x1=2.76579.
8. step: x0=2.76579, x1=3.05013.
9. step: x0=3.05013, x1=3.13625.
10. step: x0=3.13625, x1=3.14158.
11. step: x0=3.14158, x1=3.14159.
Solution: 3.142, D: 0.00002

```

9.5.4 Using `scipy.optimize.root_scalar`

Within `scipy` the secant method is defined within the `scipy.optimize.root_scalar` function:

```
| root_scalar(f, x0=1, method='secant')
```

```

converged: True
          flag: converged
function_calls: 7
  iterations: 6
         root: 0.7346035077893033
        method: secant

```

In the case of the secant method, the second bound of the interval is computed according to the code:

```

if x0 >= 0:
    x1 = x0*(1 + 1e -4) + 1e -4
else:
    x1 = x0*(1 + 1e -4) - 1e -4

```

Let's also look at the speed

```

| %timeit root_scalar(f, x0=1, method='secant')

| 153 \mus \pm 491 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

```

Related to the secant method is *Ridder's* method; we will not go into the details of the method here, but it is of the bracketed type, has a quadratic order of convergence, and we call it using the `root_scalar` function:

```

| root_scalar(f, bracket=[0, 1], method='ridder')

converged: True
          flag: converged
function_calls: 14
  iterations: 6
         root: 0.7346035077883111
        method: ridder

```

Brent's method (`brentq`) is the default and recommended choice in Scipy for finding zeros on an interval. It combines the reliability of bisection with the speed of the secant method/inverse quadratic interpolation. Usage and speed:

```

| root_scalar(f, bracket=[0, 1], method='brentq')

converged: True
          flag: converged
function_calls: 9
  iterations: 8
         root: 0.7346035077893034
        method: brentq

```

A speed comparison shows that it is usually faster than bisection:

```

| %timeit root_scalar(f, bracket=[0, 1], method='brentq')

| 78.5 \mus \pm 184 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

```

Scipy also supports the [TOMS 748](#) algorithm, which is considered the asymptotically most efficient method for finding zeros on an interval:

```

| root_scalar(f, bracket=[0, 1], method='toms748')

```

```

converged: True
      flag: converged
function_calls: 9
      iterations: 4
      root: 0.7346035077893033
      method: toms748

```

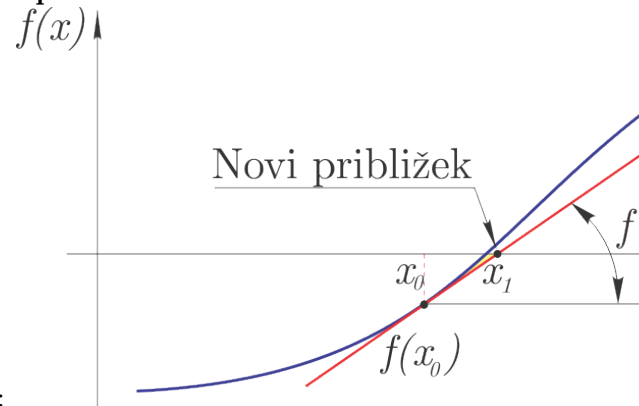
A speed comparison (in most cases it should be the fastest):

```
%timeit root_scalar(f, bracket=[0, 1], method='toms748')
```

```
530 \mus \pm 2.08 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

9.6 Newton's method

The methods presented so far do not require additional derivatives of the function; Newton's method, which we will look at next, requires one initial estimate x_0 and, in addition to the definition of the function $f(x)$, also its derivative $f'(x)$. In the literature, Newton's method is also found under the terms **tangent** method and **Newton-Raphson** method.



The working principle of the method is shown in the figure:

We could derive the method graphically (from the figure), but here let's look at the derivation using the Taylor series:

$$f(x_{i+1}) = f(x_i) + f'(x_i) (x_{i+1} - x_i) + \mathcal{O}((x_{i+1} - x_i)^2), \quad (9.12)$$

if the value of the function at x_{i+1} is to be zero, then:

$$0 = f(x_i) + f'(x_i) (x_{i+1} - x_i) + \mathcal{O}((x_{i+1} - x_i)^2). \quad (9.13)$$

We introduce a method error and neglect the higher-order terms in the Taylor series. We can derive:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \quad (9.14)$$

x_{i+1} is thus the new estimate of the sought zero.

The algorithm of Newton's method is:

1. we compute a new estimate x_{i+1} ,
2. we stop the computation if the maximum number of iterations is reached (we have not found the solution of the equation),
3. if $|x_{i+1} - x_i| < \varepsilon$ we stop the computation (we have computed an estimate of the zero), otherwise we increase the index i and go to the first step.

Notes:

- ε is the required absolute accuracy,

- *Newton's method* can diverge, so in the algorithm we prescribe a maximum number of iterations.

Above we mentioned that Newton's method is one of the better methods for finding zeros of functions. It does, however, also have some drawbacks/limitations:

- it belongs to the *open* methods,
- quadratic convergence is guaranteed only in a sufficiently small neighborhood of the solution of the equation,
- we must know the derivative of the function.

9.6.1 Order of convergence

The order of convergence of Newton's method is quadratic:

$$\varepsilon_n = C \varepsilon_{n-1}^2, \quad (9.15)$$

where C is:

$$C = -\frac{f''(x)}{2f'(x)}. \quad (9.16)$$

The convergence is therefore fast; in each new iteration the number of correct digits in the estimate doubles.

9.6.2 Numerical implementation

```
def newton_method(fun, dfun, x0, tol=1e-3, Dtol=1e-1, max_iter=50, verbose=True):
    # named `newton_method` because `newton` is a built-in function in `scipy`
    """ Returns a zero with accuracy tol

    :param fun: function whose zero we are looking for
    :param dfun: f'
    :param x0: initial estimate
    :param tol: required accuracy
    :param max_iter: maximum number of iterations before execution is interrupted
    :param Dtol: maximum sum of absolute values of the solution
    :param verbose: whether to print a short report at the end
    """
    for i in range(max_iter):
        x1 = x0 - fun(x0)/dfun(x0)
        if np.abs(x1 - x0) < tol:
            r = (x0+x1)/2
            D = np.abs(fun(x0)) + np.abs(fun(x1))
            if D > Dtol:
                raise Exception('Likely a pole or multiple zeros.')
            if verbose:
                decimals = int(np.log10(1/tol)) # does not always work for all
                cases:)
                print(f'Solution: {x1:5.{decimals}f}, number of iterations: {i+1}, D:
                {D:5.8f}')
            return x1
        x0 = x1
    raise Exception('The method does not converge after {:g}
    iterations'.format(max_iter))
```

Let's define the polynomial f and its first derivative df :

```
def f(x):
    return x**3 - 10*x**2 + 0*x + 5
def df(x):
    return 3*x**2 - 20*x
```

Let's now compute the zero:

```
| newton_method(fun=f, dfun=df, x0=1, tol=1e -8);  
| Solution: 0.73460351, number of iterations: 5, D: 0.00000000
```

Let's check the execution speed:

```
| %timeit newton_method(fun=f, dfun=df, x0=1, tol=1e -8, verbose=False)  
| 4.04 \mus \pm 35.4 ns per loop (mean \pm std. dev. of 7 runs, 100,000 loops each)
```

9.6.3 Using `scipy.optimize.root_scalar`

Within `scipy` Newton's method is defined within the `scipy.optimize.root_scalar` function:

```
| root_scalar(f, x0=1, method='newton', fprime=df)  
  
| converged: True  
|           flag: converged  
| function_calls: 10  
| iterations: 5  
|           root: 0.7346035077893033  
|           method: newton
```

And let's measure the speed:

```
| %timeit root_scalar(f, x0=1, method='newton', fprime=df)  
| 91.3 \mus \pm 284 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)
```

If we also provide the second derivative `fprime2` to `root_scalar`, Halley's method is used, which is based on the expression:

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2f'^2(x_n) - f(x_n)f''(x_n)} \quad (9.17)$$

```
| def ddf(x):  
|     return 6*x - 20  
  
| root_scalar(f, x0=1, method='halley', fprime=df, fprime2=ddf)  
  
| converged: True  
|           flag: converged  
| function_calls: 10  
| iterations: 3  
|           root: 0.7346035077893033  
|           method: halley
```

And the speed:

```
| %timeit root_scalar(f, x0=1, method='halley', fprime=df, fprime2=ddf)  
| 62.3 \mus \pm 536 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)
```

9.7 Solving systems of nonlinear equations

We solve a system of equations, which we write in vector form as follows:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}. \quad (9.18)$$

In scalar form we write the vector expression above as:

$$\begin{aligned} f_0(x_0, x_1, \dots, x_{n-1}) &= 0 \\ f_1(x_0, x_1, \dots, x_{n-1}) &= 0 \\ &\vdots \\ f_{n-1}(x_0, x_1, \dots, x_{n-1}) &= 0. \end{aligned} \quad (9.19)$$

Solving a system of n nonlinear equations is significantly more demanding than solving a single nonlinear equation. Such a system of equations can have multiple solutions, and which one we compute depends on the initial conditions. Usually, a good choice of initial conditions is helped by the physical problem we are solving.

For computing the solution of a system of equations, *Newton's* method turns out to be the simplest and often also the best (there are also other methods, which are, however, often variations of Newton's method).

Similarly to the derivation of Newton's method for solving a single equation, here too we start with the expansion of the function f_i into a Taylor series:

$$f_i(\mathbf{x} + \Delta\mathbf{x}) = f_i(\mathbf{x}) + \sum_{j=0}^{n-1} \frac{\partial f_i}{\partial x_j} \Delta x_j + \mathcal{O}(\|\Delta\mathbf{x}\|^2). \quad (9.20)$$

We introduce a method error when we neglect the second- and higher-order terms and write the expression in matrix form:

$$\mathbf{f}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x}) \Delta\mathbf{x}, \quad (9.21)$$

where $\mathbf{J}(\mathbf{x})$ is the *Jacobian* matrix at the values $\mathbf{x}$. The elements of the Jacobian matrix are:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}. \quad (9.22)$$

If $\mathbf{x} + \Delta\mathbf{x}$ is to be a solution of the system of equations, then:

$$\mathbf{0} = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x}) \Delta\mathbf{x} \quad (9.23)$$

and it therefore follows:

$$\mathbf{J}(\mathbf{x}) \Delta\mathbf{x} = -\mathbf{f}(\mathbf{x}). \quad (9.24)$$

We have derived a system of linear equations; the coefficient matrix is denoted by $\mathbf{J}(\mathbf{x})$, the vector of unknowns is $\Delta\mathbf{x}$, and the vector of constants is $-\mathbf{f}(\mathbf{x})$.

Note: analytical computation of the Jacobian matrix can be time-consuming, and therefore we often compute it approximately at $\mathbf{x}$ numerically:

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(\mathbf{x} + \mathbf{e}_j h) - f_i(\mathbf{x})}{h}, \quad (9.25)$$

where h is a small displacement and $\mathbf{e}_j$ is a unit displacement in the direction x_j . If the Jacobian matrix is computed numerically, we speak of the secant method and not Newton's method.

For the numerical computation we can use the function `scipy.optimize.approx_fprime` (for details, see the [documentation](#)).

9.7.1 Numerical implementation

The algorithm is therefore:

1. We choose an initial estimate $\mathbf{x}_0$, a maximum number of iterations, and set the index to zero: $i = 0$.
2. We compute the Jacobian matrix $\mathbf{J}(\mathbf{x}_i)$ and solve the linear system: $\mathbf{J}(\mathbf{x}_i) \Delta \mathbf{x}_i = -\mathbf{f}(\mathbf{x}_i)$.
3. We compute a new estimate: $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta \mathbf{x}_i$.
4. If the error is smaller than required, the procedure stops*. We also stop the procedure if the number of iterations is greater than allowed, otherwise we increase the index $i = i + 1$ and return to step 2.

* Note:

We can estimate the error by the norm of the difference of two consecutive estimates:

$$\sum_{j=0}^{n-1} |x_{i,j} - x_{i-1,j}| < \varepsilon, \quad (9.26)$$

where i is the iteration index and j is the element index.

9.7.2 Using `scipy.optimize.root`

The function `scipy.optimize.root` has extensive [documentation](#) and enables a number of different approaches:

```
root(fun, x0, args=(), method='hybr', jac=None, tol=None, callback=None,
options=None)
```

If we use the default parameters, we only need to define the vector function `fun` and the initial value `x0`.

Let's import the function:

```
from scipy.optimize import root
```

Let's look at the usage on an example (this is an example on p. 76, Jože Petrišič, Reševanje enačb, 1996, FS, UNI-LJ):

$$f_0(\mathbf{x}) = x_0^2 + x_0 x_1 - 10 = 0, \quad (9.27)$$

$$f_1(\mathbf{x}) = x_1 + 3x_0 x_1^2 - 57 = 0, \quad (9.28)$$

where the vector is $\mathbf{x} = [x_0, x_1]$.

First let's define a Python function that returns the list of function results $[f_0(\mathbf{x}), f_1(\mathbf{x})]$:

```
def func(x):
    return [x[0]**2 + x[0]*x[1] - 10,
            x[1] + 3*x[0]*x[1]**2 - 57]
```

Let's also define the Jacobian matrix:

```
def J(x):
    return np.array([[2*x[0]+x[1], x[0]],
                    [3*x[1]**2, 1+6*x[0]*x[1]]])
```

Let's use the initial values $x_0 = 1.5$ and $x_1 = 3.5$ and solve the problem:

```
solution = root(fun=func, x0=[1.5, 3.5], jac=J)
solution
```

```

message: The solution converged.
success: True
status: 1
  fun: [-6.972e -12  3.148e -11]
   x: [ 2.000e+00  3.000e+00]
method: hybr
  nfev: 12
  njev: 1
  fjac: [[ -2.235e -01 -9.747e -01]
         [ 9.747e -01 -2.235e -01]]
   r: [-2.954e+01 -3.782e+01 -6.940e+00]
  qtf: [-2.836e -08 -1.324e -08]

```

The root function returns an extensive result. The most important are the attribute `x`, which represents the sought solution, and the attribute `success`, which tells whether the solution has converged:

```
| solution.x
```

```
| array([2., 3.])
```

```
| solution.success
```

```
| True
```

```

import matplotlib.pyplot as plt

# Prepare the grid of points
x0_range = np.linspace(-5, 5, 400)
x1_range = np.linspace(-5, 5, 400)
X0, X1 = np.meshgrid(x0_range, x1_range)

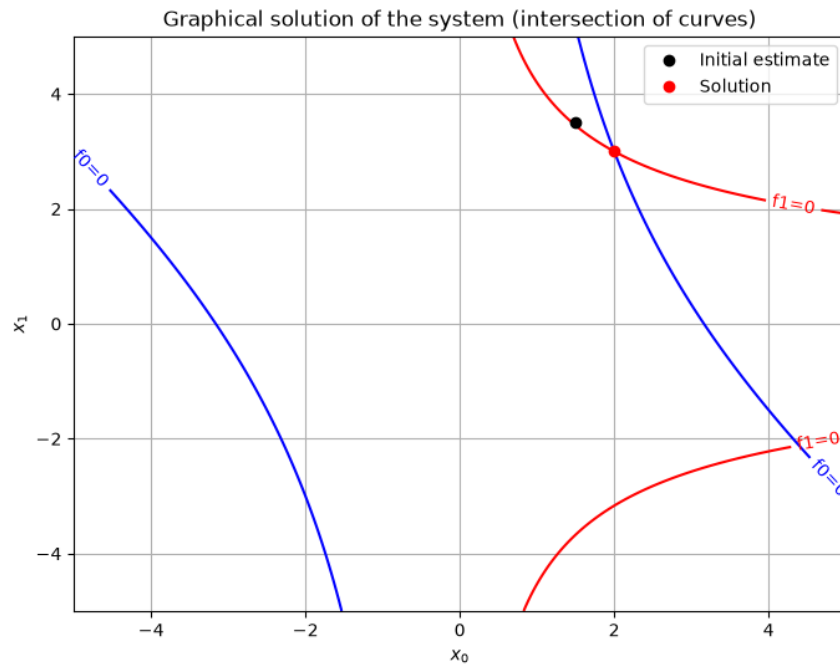
# Definition of the system functions
F0 = X0**2 + X0*X1 - 10
F1 = X1 + 3*X0*X1**2 - 57

plt.figure(figsize=(8, 6))

# Plot the zero contour lines (where f(x) = 0)
CS0 = plt.contour(X0, X1, F0, levels=[0], colors='blue')
CS1 = plt.contour(X0, X1, F1, levels=[0], colors='red')

# Labels
plt.clabel(CS0, inline=1, fontsize=10, fmt='f0=0')
plt.clabel(CS1, inline=1, fontsize=10, fmt='f1=0')
plt.plot(1.5, 3.5, 'ko', label='Initial estimate')
plt.plot(solution.x[0], solution.x[1], 'ro', label='Solution')
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.title('Graphical solution of the system (intersection of curves)')
plt.grid(True)
plt.legend()
plt.show()

```



9.8 Additional

Those of you who are enthusiastic about the [Raspberry Pi](#) and use their camera (e.g. [this one without the infrared filter](#)) might be interested in the [picamera](#) library.

9.8.1 Using `sympy.solve` to solve equations

For smaller systems we can also find the solution symbolically. Let's look at the example above:

```
import sympy as sym
sym.init_printing()
x, y = sym.symbols('x, y')
functions = [x**2 + x*y - 10, y + 3*x*y**2 - 57]
functions
```

$$[x^2 + xy - 10, 3xy^2 + y - 57] \quad (9.29)$$

```
sol = sym.solve(functions, x, y)
print(f'Number of solutions: {len(sol)}')
print(f'First solution: {sol[0]}')
```

```
Number of solutions: 4
First solution: (2, 3)
```

Chapter 10

Numerical differentiation

10.1 Introduction

Every elementary function can be differentiated analytically. The definition of the derivative is:

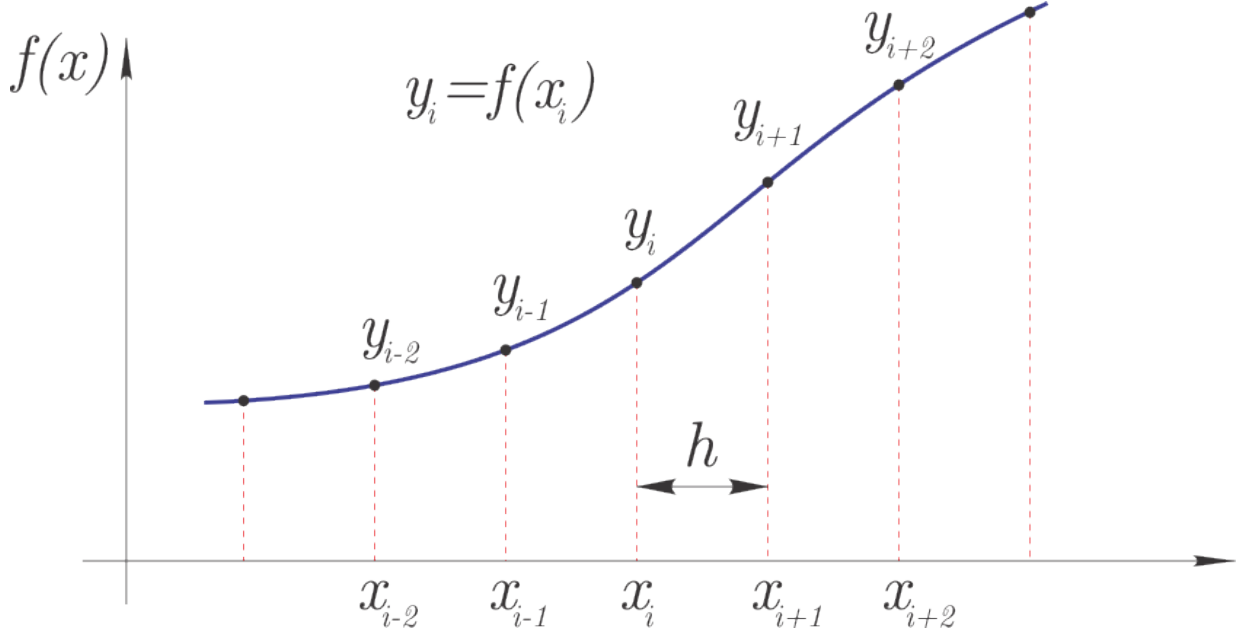
$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}. \quad (10.1)$$

Directly applying the equation above leads to subtracting very similar function values ($f(x + \Delta x)$, $f(x)$) that are burdened by round-off error, and then dividing them by a small value Δx ; as a result, the derivative has substantially fewer significant digits than the function values. We avoid numerical differentiation whenever we can; in some cases, however (e.g., solving differential equations), it is an indispensable tool!

In numerical differentiation we have two fundamentally different approaches:

1. first perform **interpolation/approximation**, then compute the derivative based on the known interpolation/approximation functions (we already discussed this topic in interpolation and approximation), and
2. computing the derivative **directly from the values in a table**.

In this chapter we will learn how to numerically compute the derivative of a function $f(x)$; here the values of the function $f(x)$ are given in tabular form (pairs x_i, y_i), as shown in the figure:



We will first focus on equidistant values x_i spaced with a step h ; the function values will be $y_i = f(x_i)$.

Based on the definition of the derivative above, the first derivative (at location i) could be written as:

$$y'_i = \frac{y_{i+1} - y_i}{h}, \quad (10.2)$$

where $h = x_{i+1} - x_i$. By rearranging the equation:

$$y'_i = -\frac{y_i}{h} + \frac{y_{i+1}}{h}, \quad (10.3)$$

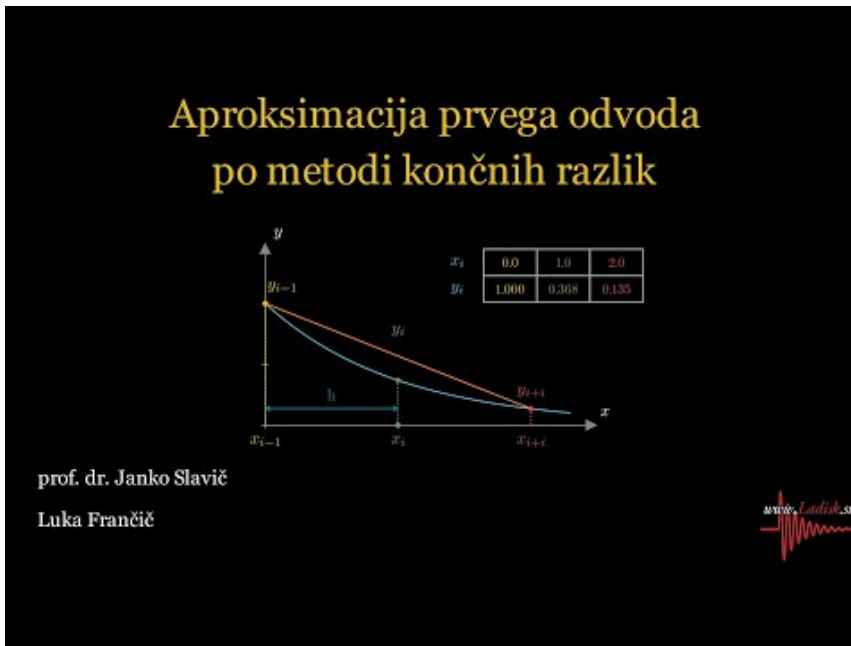
we can also say that for the first derivative of the function at location i we **weight** the function value at i by $-1/h$ and the function value at $i+1$ by $+1/h$.

In what follows we will look at the theoretical background of how to determine the appropriate **weights** for various orders of derivatives, what options we have here, and how this affects the order of accuracy.

10.2 Approximating the first derivative using the finite-difference method

As an introduction, let's watch the video below:

```
from IPython.display import YouTubeVideo
YouTubeVideo('YYuGL -VP2BE', width=800, height=300)
```



The derivative $f'(x)$ can be approximated based on a Taylor series expansion. We call this method the **finite-difference method** or the **difference method**.

Let's expand the **forward Taylor series** (forward, because of the $+h$ term):

$$f(x+h) = \sum_{n=0}^{\infty} \frac{h^n}{n!} \frac{d^n}{dx^n} f(x) = f(x) + h f'(x) + \underbrace{\frac{h^2}{2} f''(x) + \dots}_{\mathcal{O}(h^2)} \quad (10.4)$$

The term $\mathcal{O}(h^2)$ denotes a second-order error. If we express the first derivative from the equation:

$$f'(x) = \frac{1}{h} (f(x+h) - f(x)) - \underbrace{\frac{h}{2} f''(x) + \dots}_{\mathcal{O}(h^1)} \quad (10.5)$$

We find that we can estimate the first derivative at the point x_i (that is: $f'_o(x_i)$) based on two consecutive function values:

$$f'_o(x_i) = \frac{1}{h} (y_{i+1} - y_i) \quad (10.6)$$

and in doing so we make a **method error**, which is of first order $\mathcal{O}(h^1)$.

We used $y_i = f(x_i)$ (see the figure above).

The error is:

$$e = -\frac{h}{2} f''(\xi), \quad (10.7)$$

where ξ is an unknown value on the interval $[x_i, x_{i+1}]$ and we neglected the higher-order terms. The following expression therefore holds:

$$f'(x_i) = f'_o(x_i) + e \quad (10.8)$$

Now let's see how we arrive at the same result with a machine-assisted derivation; first we import `sympy`:

```
| import sympy as sym
| sym.init_printing()
```

Let's define the symbols:

```
| f = sym.Function('f')
| x, h = sym.symbols('x, h')
```

Then we continue with the **forward Taylor series** expansion:

```
| f(x+h).series(h, n=2)
```

$$f(x) + h \left. \frac{d}{d\xi_1} f(\xi_1) \right|_{\xi_1=x} + \mathcal{O}(h^2) \quad (10.9)$$

The term $\mathcal{O}(h^2)$ contains terms of second and higher order. The equation above uses a temporary variable for differentiation ξ_1 ; let's carry out the differentiation and substitute $\xi_1 = x$:

```
| f(x+h).series(h, n=2).doit()
```

$$f(x) + h \frac{d}{dx} f(x) + \mathcal{O}(h^2) \quad (10.10)$$

We write the equation:

```
| equation = sym.Eq(f(x+h), f(x+h).series(h, n=2).doit())
| equation
```

$$f(h+x) = f(x) + h \frac{d}{dx} f(x) + \mathcal{O}(h^2) \quad (10.11)$$

We solve it for the first derivative $f'(x)$:

```
| f1_forward_exact = sym.solve(equation, f(x).diff(x))[0]
| f1_forward_exact
```

$$\frac{f(h+x) - f(x) + O(h^2)}{h} \quad (10.12)$$

If we do not account for the second- and higher-order derivatives, we have therefore made an error:

```
| f1_forward_0 = f1_forward_exact.expand().get0()
| f1_forward_0
```

$$O(h) \quad (10.13)$$

The error $\mathcal{O}(h^1)$ is therefore of first order, and if we neglect this term we make a *method error* and obtain an estimate of the derivative:

```
| f1_forward_estimate = f1_forward_exact.expand().remove0()
| f1_forward_estimate
```

$$-\frac{f(x)}{h} + \frac{f(h+x)}{h} \quad (10.14)$$

We find that this is the same expression we derived above, so it is:

$$y'_i = \frac{1}{h} (-y_i + y_{i+1}). \quad (10.15)$$

The weights are therefore:

Derivative $\downarrow \setminus$ Values $\rightarrow$	y_i	y_{i+1}
$y'_i = \frac{1}{h}$	-1	1

10.3 Central difference scheme

10.3.1 Derivative $f'(x)$

First let's look at the **backward Taylor series** expansion:

```
| f(x -h).series(h, n=3).doit()
```

$$f(x) - h \frac{d}{dx} f(x) + \frac{h^2 \frac{d^2}{dx^2} f(x)}{2} + O(h^3) \quad (10.16)$$

We find that in the difference between the forward and backward series the even-order terms cancel out; let's define:

```
| def difference(n=3):
|     return f(x+h).series(h, n=n).doit() - f(x -h).series(h, n=n).doit()
| difference(n=3)
```

$$2h \frac{d}{dx} f(x) + O(h^3) \quad (10.17)$$

We carry out the following steps:

1. subtract the backward Taylor series from the forward series, the even-order derivatives cancel out,
2. solve the equation for the first derivative,

3. determine the method error,
4. determine the estimate of the derivative.

Let's carry out the steps above:

```
f1_cent_exact = sym.solve(
    sym.Eq(f(x+h) - f(x-h), difference(n=3)), # step 1
    f(x).diff(x))[0] # step 2
f1_cent_0 = f1_cent_exact.expand().get0() # step 3
f1_cent_estimate = f1_cent_exact.expand().remove0() # step 4
```

The estimate of the 1st derivative is therefore:

```
f1_cent_estimate
```

$$-\frac{f(-h+x)}{2h} + \frac{f(h+x)}{2h} \quad (10.18)$$

Or:

$$y'_i = \frac{1}{2h} (-y_{i-1} + y_{i+1}) \quad (10.19)$$

The weights are therefore:

Derivative ↓ \ Values	y_{i-1}	y_i	y_{i+1}
→			
$y'_i = \frac{1}{2h} \cdot$	-1	0	1

The method error is therefore of second order:

```
f1_cent_0
```

$$O(h^2) \quad (10.20)$$

10.3.2 Example: $\exp(-x)$

Let's look at an example of the exponential function $f(x) = \exp(-x)$ and, for the point $x = 1.0$, compute the first derivative $f'(x) = -\exp(-x)$ at the steps $h_0 = 1$ and $h_1 = 0.1$.

First we prepare a table of numerical values and the exact result:

```
import numpy as np
x0 = np.array([0., 1., 2.]) # step h=1
y0 = np.exp(-x0)
h0 = x0[1] - x0[0]

x1 = np.array([0.9, 1.0, 1.1]) # step h=0.1
y1 = np.exp(-x1)
h1 = x1[1] - x1[0]

f1_exact0 = - np.exp(-1) # exact result
f1_exact1 = - np.exp(-1) # exact result
```

Then we use the forward scheme:

```
f1_forward_estimate # as a reminder
```

$$-\frac{f(x)}{h} + \frac{f(h+x)}{h} \quad (10.21)$$

```
| f1_forward0 = (y0[1:] - y0[: -1])/h0 # step h_0
| f1_forward1 = (y1[1:] - y1[: -1])/h1 # step h_1
```

Let's compute the error at $x = 1.0$:

```
| f1_exact0 - f1_forward0[1]

-0.135335283236613 (10.22)
```

```
| f1_exact1 - f1_forward1[1] # step h1

-0.0177958664378144 (10.23)
```

We can confirm that the error at step $h/10$ is indeed approximately $1/10$ of that at step h . Now let's also look at the error for the central difference scheme, which is of second order:

```
| f1_cent_estimate


$$-\frac{f(-h+x)}{2h} + \frac{f(h+x)}{2h} \quad (10.24)$$

```

```
| f1_cent0 = (y0[2:] - y0[: -2])/(2*h0) # step h_0
| f1_cent1 = (y1[2:] - y1[: -2])/(2*h1) # step h_1
```

Let's analyze the error:

```
| f1_exact0 - f1_cent0[0] # step h0

0.0644529172102513 (10.25)
```

```
| f1_exact1 - f1_cent1[0] # step h1

0.000613439041155517 (10.26)
```

We can confirm that the error at step $h/10$ is indeed approximately $1/100$ of that at step h .

10.3.3 Derivative $f''(x)$

If we add the forward and backward Taylor series, the odd-order derivatives cancel out:

```
| def total(n=3):
|     return f(x+h).series(h, n=n).doit() + f(x-h).series(h, n=n).doit()
| total(n=4)
```

$$2f(x) + h^2 \frac{d^2}{dx^2} f(x) + O(h^4) \quad (10.27)$$

Let's determine the second derivative:

```
| f2_cent_exact = sym.solve(
|     sym.Eq(f(x+h) + f(x-h), total(n=4)), # step 1
|     f(x).diff(x,2))[0] # step 2
| f2_cent_0 = f2_cent_exact.expand().get0() # step 3
| f2_cent_estimate = f2_cent_exact.expand().remove0() # step 4
```

The estimate of the second derivative is:

```
| f2_cent_estimate
```

$$-\frac{2f(x)}{h^2} + \frac{f(-h+x)}{h^2} + \frac{f(h+x)}{h^2} \quad (10.28)$$

Or:

$$y_i'' = \frac{1}{h^2} (y_{i-1} - 2y_i + y_{i+1}) \quad (10.29)$$

The weights are therefore:

Derivative ↓ \ Values	y_{i-1}	y_i	y_{i+1}
$\rightarrow$			
$y_i'' = \frac{1}{h^2}$	1	-2	1

The method error is again of second order:

```
| f2_cent_0
```

$$O(h^2) \quad (10.30)$$

10.3.4 Derivative $f'''(x)$

If we want to determine the third derivative, we have to expand the Taylor series to order 5:

```
| eq_h = sym.Eq(f(x+h) -f(x -h), difference(n=5))
| eq_h
```

$$-f(-h+x) + f(h+x) = 2h \frac{d}{dx} f(x) + \frac{h^3 \frac{d^3}{dx^3} f(x)}{3} + O(h^5) \quad (10.31)$$

Using the 1st derivative we derived above would not help us, since the order of the error is $O(h^2)$, which means that in the equation above, when dividing by h^3 , we would get $O(h^{-1})$.

Let's use a trick: we repeat the expansion, but based on additional points that are $2h$ and $-2h$ away from x :

```
| eq_2h = eq_h.subs(h, 2*h)
| eq_2h
```

$$-f(-2h+x) + f(2h+x) = 4h \frac{d}{dx} f(x) + \frac{8h^3 \frac{d^3}{dx^3} f(x)}{3} + O(h^5) \quad (10.32)$$

Now we have two equations and two unknowns; we will solve the system step by step:

1. solve equation `eq_h` for the first derivative,
2. solve equation `eq_2h` for the first derivative,
3. equate the results of the first two steps and solve for the third derivative,
4. determine the method error,
5. determine the estimate of the derivative.

Let's carry out the stated steps:

```
f3_cent_exact = sym.solve(
    sym.Eq(sym.solve(eq_h, f(x).diff(x))[0], # step 1
    sym.solve(eq_2h, f(x).diff(x))[0]),      # step 2
    f(x).diff(x,3))[0]                      # step 3
f3_cent_0 = f3_cent_exact.expand().get0()    # step 4
f3_cent_estimate = f3_cent_exact.expand().remove0() # step 5
```

The estimate of the 3rd derivative is:

```
f3_cent_estimate
```

$$-\frac{f(-2h+x)}{2h^3} + \frac{f(-h+x)}{h^3} - \frac{f(h+x)}{h^3} + \frac{f(2h+x)}{2h^3} \quad (10.33)$$

Or:

$$y_i''' = \frac{1}{h^3} (-y_{i-2}/2 + y_{i-1} - y_{i+1} + y_{i+2}/2) \quad (10.34)$$

The weights are therefore:

Derivative ↓ \	y_{i-2}	y_{i-1}	y_i	y_{i+1}	y_{i+2}
Values →					
$y_i''' = \frac{1}{h^3} \cdot$	-0.5	1	0	-1	0.5

Let's confirm that the method error is of second order:

```
f3_cent_0
```

$$O(h^2) \quad (10.35)$$

10.3.5 Derivative $f^{(4)}(x)$

Let's repeat a procedure similar to the one for the 3rd derivative, but for the 4th derivative we add the forward and backward Taylor series (up to order 6):

```
eq_h = sym.Eq(f(x+h)+f(x-h), total(n=6))
eq_h
```

$$f(-h+x) + f(h+x) = 2f(x) + h^2 \frac{d^2}{dx^2} f(x) + \frac{h^4 \frac{d^4}{dx^4} f(x)}{12} + O(h^6) \quad (10.36)$$

Let's prepare an additional equation based on the points that are $2h$ and $-2h$ away from x :

```
eq_2h = eq_h.subs(h, 2*h)
eq_2h
```

$$f(-2h+x) + f(2h+x) = 2f(x) + 4h^2 \frac{d^2}{dx^2} f(x) + \frac{4h^4 \frac{d^4}{dx^4} f(x)}{3} + O(h^6) \quad (10.37)$$

From the two equations we determine the 4th derivative:

```
f4_cent_exact = sym.solve(
    sym.Eq(sym.solve(eq_h, f(x).diff(x,2))[0], # step 1
    sym.solve(eq_2h, f(x).diff(x,2))[0]),      # step 2
    f(x).diff(x,4))[0]                      # step 3
f4_cent_0 = f4_cent_exact.expand().get0()    # step 4
f4_cent_estimate = f4_cent_exact.expand().remove0() # step 5
```

The estimate of the 4th derivative is:

```
| f4_cent_estimate
```

$$\frac{6f(x)}{h^4} + \frac{f(-2h+x)}{h^4} - \frac{4f(-h+x)}{h^4} - \frac{4f(h+x)}{h^4} + \frac{f(2h+x)}{h^4} \quad (10.38)$$

Or:

$$y_i^{(4)} = \frac{1}{h^4} (y_{i-2} - 4y_{i-1} + 6y_i - 4y_{i+1} + y_{i+2}) \quad (10.39)$$

The weights are therefore:

Derivative ↓ \	y_{i-2}	y_{i-1}	y_i	y_{i+1}	y_{i+2}
Values →					
$y_i^{(4)} = \frac{1}{h^4}$	1	-4	6	-4	1

Let's confirm that the method error is of second order:

```
| f4_cent_0
```

$$O(h^2) \quad (10.40)$$

10.3.6 Summary of the central difference scheme

Above we derived the first four derivatives with a 2nd-order method error. The key point of the derivations above is that they give us the **weights** by which we must multiply the function values in order to compute the approximation of a given derivative. For this reason, we will collect these weights here. If you are impatient, you can jump to the table below. Continue reading if you want to learn how to organize the previously derived expressions by machine.

First let's collect all the derivative estimates into a list:

```
| derivatives = [f1_cent_estimate, f2_cent_estimate, f3_cent_estimate,
| f4_cent_estimate]
| derivatives
```

$$\left[-\frac{f(-h+x)}{2h} + \frac{f(h+x)}{2h}, -\frac{2f(x)}{h^2} + \frac{f(-h+x)}{h^2} + \frac{f(h+x)}{h^2}, -\frac{f(-2h+x)}{2h^3} + \frac{f(-h+x)}{h^3} - \frac{f(h+x)}{h^3} + \frac{f(2h+x)}{2h^3} \right] \quad (10.41)$$

We have 5 function values available (at locations $x-2h, x-h, x, x+h, x+2h$), which we put into a list:

```
| function_values = [f(x-2*h), f(x-h), f(x), f(x+h), f(x+2*h)]
```

We compute the weight of the first derivative for the function value $f(x-h)$:

```
| f1_cent_estimate.expand().coeff(function_values[1])
```

$$-\frac{1}{2h} \quad (10.42)$$

Now let's generalize and compute the weights for all function values and for all derivative estimates:

```
central_diff_scheme = [[derivative.expand().coeff(fv) for fv in function_values] \
                        for derivative in derivatives]
central_diff_scheme
```

$$\left[\left[0, -\frac{1}{2h}, 0, \frac{1}{2h}, 0 \right], \left[0, \frac{1}{h^2}, -\frac{2}{h^2}, \frac{1}{h^2}, 0 \right], \left[-\frac{1}{2h^3}, \frac{1}{h^3}, 0, -\frac{1}{h^3}, \frac{1}{2h^3} \right], \left[\frac{1}{h^4}, -\frac{4}{h^4}, \frac{6}{h^4}, -\frac{4}{h^4}, \frac{1}{h^4} \right] \right] \quad (10.43)$$

The summaries above can also be written in tabular form:

Derivative ↓ \ Values →	y_{i-2}	y_{i-1}	y_i	y_{i+1}	y_{i+2}
$y'_i = \frac{1}{h}$	0	-0.5	0	0.5	0
$y''_i = \frac{1}{h^2}$	0	1	-2	1	0
$y'''_i = \frac{1}{h^3}$	-0.5	1	0	-1	0.5
$y^{(4)}_i = \frac{1}{h^4}$	1	-4	6	-4	1

The central difference scheme shown has a 2nd-order error $\mathcal{O}(h^2)$.

Note: if you are interested in a generalization of the difference method, please see the [findiff](#) package.

10.3.7 Improved approximation - Richardson extrapolation

If the exact derivative is computed as:

$$f'(x_i) = f'_o(x_i) + e, \quad (10.44)$$

where $f'_o(x_i)$ is the numerically computed derivative at the point x_i and e is the error estimate. For a method of accuracy order n : $\mathcal{O}(h^n)$ at step h the following holds:

$$f'(x_i) = f'_o(x_i, h) + K h^n, \quad (10.45)$$

where K is an unknown constant.

If we halve the step and assume that K does not change, the following holds:

$$f'(x_i) = f'_o\left(x_i, \frac{h}{2}\right) + K \left(\frac{h}{2}\right)^n. \quad (10.46)$$

From both equations we eliminate the constant K and determine the improved approximation:

$$\bar{f}'(x_i) = \frac{2^n f'_o\left(x_i, \frac{h}{2}\right) - f'_o(x_i, h)}{2^n - 1} \quad (10.47)$$

Example

Let's look at the example $f(x) = \sin(x)$ (the analytical derivative is: $f'(x) = \cos(x)$):

```
x = np.linspace(0, 2*np.pi, 9)
y = np.sin(x)
```

At step h we have the function values defined at:

```
x
array([0., 0.78539816, 1.57079633, 2.35619449, 3.14159265,
       3.92699082, 4.71238898, 5.49778714, 6.28318531])
```

The numerical derivative at $x = \pi$:

```
| x[4]
```

$$3.14159265358979 \quad (10.48)$$

and at step h is

```
| h = x[1] - x[0]
| f_estimate_h = ( -0.5*y[3] + 0.5*y[5])/h
| f_estimate_h
```

$$-0.900316316157106 \quad (10.49)$$

The computation at step $2h$:

```
| h2 = x[2] - x[0]
| f_estimate_2h = ( -0.5*y[2] + 0.5*y[6])/h2
| f_estimate_2h
```

$$-0.636619772367581 \quad (10.50)$$

Let's compute the improved estimate for $x = \pi$:

```
| f_estimate_improved = (2**2 * f_estimate_h - f_estimate_2h)/(2**2 -1)
| f_estimate_improved
```

$$-0.988215164086948 \quad (10.51)$$

We see that the improved estimate is closest to the theoretical value $\cos(\pi) = -1$.

10.4 Non-central difference scheme

The central difference scheme we learned above is very useful and relatively accurate. However, since we cannot always use it (for example, at the beginning or end of the table), we have to rely on **non-central difference schemes** for computing derivatives.

We have:

- the **forward difference scheme**, which approximates the derivative at a point with the function values at the following points, and
- the **backward difference scheme**, which approximates the derivative at a point with the values at the preceding points.

The derivations are similar to what we showed for the central difference scheme, so we will not treat them here and will only show the final result.

10.4.1 Forward difference scheme

Forward difference scheme with error order $\mathcal{O}(h^1)$:

Derivative $\downarrow \backslash$ Values $\rightarrow$	y_i	y_{i+1}	y_{i+2}	y_{i+3}	y_{i+4}
$y'_i = \frac{1}{h}$	-1	1	0	0	0
$y''_i = \frac{1}{h^2}$	1	-2	1	0	0
$y'''_i = \frac{1}{h^3}$	-1	3	-3	1	0
$y^{(4)}_i = \frac{1}{h^4}$	1	-4	6	-4	1

Forward difference scheme with error order $\mathcal{O}(h^2)$:

Derivative $\downarrow \setminus$ Values $\rightarrow$	y_i	y_{i+1}	y_{i+2}	y_{i+3}	y_{i+4}	y_{i+5}
$y'_i = \frac{1}{2h}$	-3	4	-1	0	0	0
$y''_i = \frac{1}{h^2}$	2	-5	4	-1	0	0
$y'''_i = \frac{1}{2h^3}$	-5	18	-24	14	-3	0
$y^{(4)}_i = \frac{1}{h^4}$	3	-14	26	-24	11	-2

10.4.2 Backward difference scheme

Backward difference scheme with error order $\mathcal{O}(h^1)$:

Derivative $\downarrow \setminus$ Values $\rightarrow$	y_{i-4}	y_{i-3}	y_{i-2}	y_{i-1}	y_i
$y'_i = \frac{1}{h}$	0	0	0	-1	1
$y''_i = \frac{1}{h^2}$	0	0	1	-2	1
$y'''_i = \frac{1}{h^3}$	0	-1	3	-3	1
$y^{(4)}_i = \frac{1}{h^4}$	1	-4	6	-4	1

Backward difference scheme with error order $\mathcal{O}(h^2)$:

Derivative $\downarrow \setminus$ Values $\rightarrow$	y_{i-5}	y_{i-4}	y_{i-3}	y_{i-2}	y_{i-1}	y_i
$y'_i = \frac{1}{2h}$	0	0	0	1	-4	3
$y''_i = \frac{1}{h^2}$	0	0	-1	4	-5	2
$y'''_i = \frac{1}{2h^3}$	0	3	-14	24	-18	5
$y^{(4)}_i = \frac{1}{h^4}$	-2	11	-24	26	-14	3

10.5 Using `numpy.gradient`

To compute numerical derivatives (2nd-order central difference scheme) we can also use `numpy.gradient()` ([documentation](#)):

```
| gradient(f, *varargs, **kwargs)
```

where `f` represents a table of values (in the form of a numerical array) of the function whose derivative we are looking for. `f` can be one or more dimensions. The positional parameters `varargs` define the spacing between the values of the argument of the function `f`; the default value is 1. This value can be a scalar, but it can also be a list of values of the independent variable (or even a combination of both). At the edges, the gradient method uses the forward or backward scheme; the parameter `edge_order` defines the order of the scheme used at the edges (we can choose between 1 or 2, the default value is 1).

The result of the `gradient` function is a numerical array (or an array of numerical arrays) with the computed derivatives.

For details see the [documentation](#).

For differentiating a function (and not a table of values), see: [scipy.diffentiate](#).

10.5.1 Example

We will look at an example of how to use the **weights**, the gradient function, and the special handling at the edges. First we prepare a table of data:

```
x, h = np.linspace(0, 1, 20, retstep=True)
y = np.sin(2*np.pi*x)
```

Weights of the difference schemes:

```
central = np.array([-0.5, 0, 0.5]) # weights of the central scheme; see also the
findiff package (below)
forward = np.array([-3/2, 2, -1/2])
backward = np.array([1/2, -2, 3/2])
```

Now we compute the derivative of the interior points (the first way is with list comprehensions, the second is vectorized):

```
derivative_interior = np.array([y[i -1:i+2] @ central/h for i in range(1, len(x)
-1)]) # list comprehension
derivative_interior = np.convolve(y, central[:: -1], mode='valid') / h # vectorized
```

At the edges we use the forward or backward difference scheme:

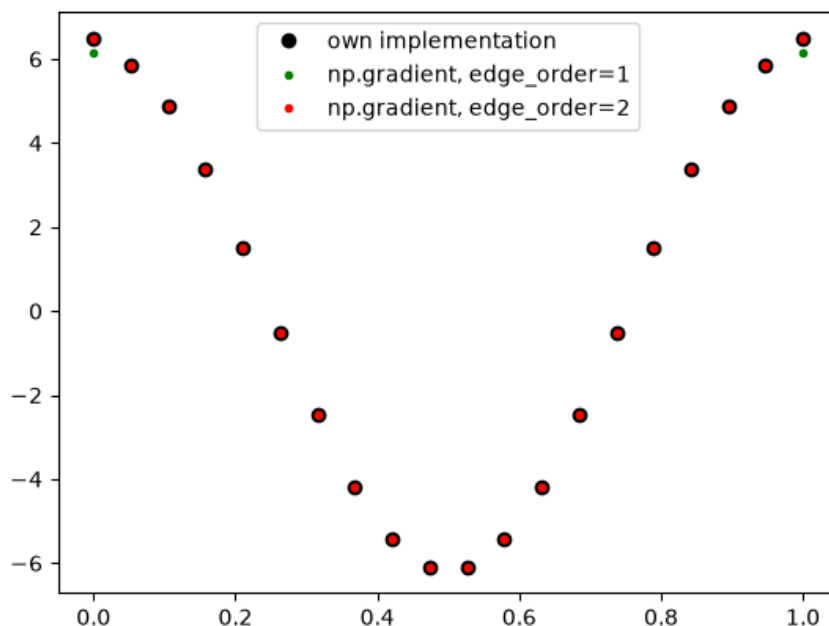
```
derivative_first = y[:len(forward)] @ forward / h # forward
derivative_last = y[-len(backward):] @ backward / h # backward
```

Let's assemble the result:

```
derivative_full = np.hstack([derivative_first, derivative_interior, derivative_last])
```

We display the result together with the result of the `np.gradient` function:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(x, derivative_full, 'ko', lw=3, label='own implementation')
plt.plot(x, np.gradient(y, h), 'g.', label='np.gradient, edge_order=1')
plt.plot(x, np.gradient(y, h, edge_order=2), 'r.', label='np.gradient, edge_order=2')
plt.legend();
```



10.5.2 Using `numpy.diff`

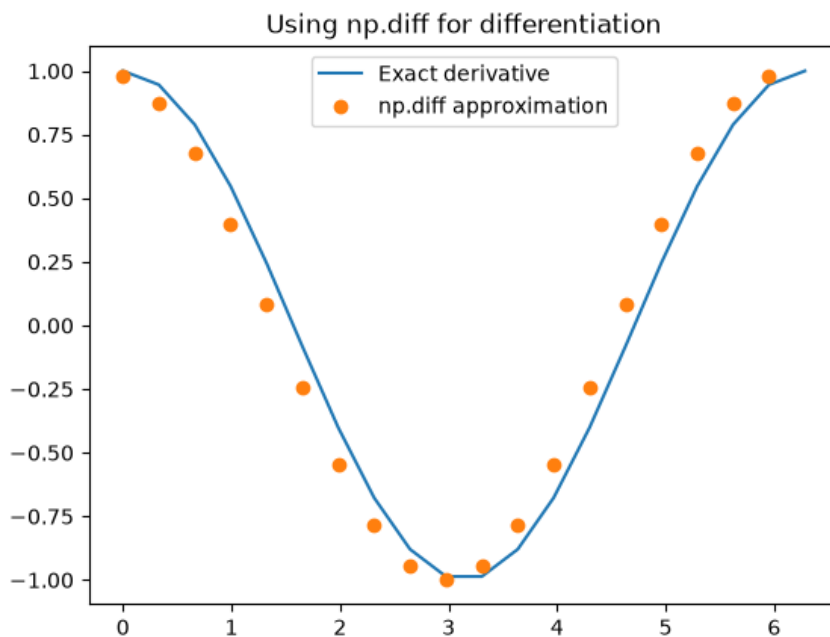
Let's also present here the frequently used function `numpy.diff` ([documentation](#)), which computes the n -th discrete difference along a given axis. The first difference is given as $a[i+1] - a[i]$. This is the basic operation for numerical differentiation (it is a first-order accurate derivative), but we have to divide the result by the step h .

When using `np.diff`, the length of the array is reduced by 1.

```
x_d = np.linspace(0, 2*np.pi, 20)
y_d = np.sin(x_d)
h_d = x_d[1] - x_d[0]

# First derivative with np.diff (forward difference)
dy = np.diff(y_d) / h_d
x_diff = x_d[:-1] # the x axis shrinks

plt.figure()
plt.plot(x_d, np.cos(x_d), label='Exact derivative')
plt.plot(x_diff, dy, 'o', label='np.diff approximation')
plt.legend()
plt.title('Using np.diff for differentiation')
plt.show()
```



10.6 Round-off error in numerical differentiation

Above we focused on the method error. In numerical differentiation, however, we must also pay very close attention to the **round-off** (or *representation*) error! Let's look at the first derivative (using the central difference scheme) written with the method error ($k h^2$) and the round-off error ε :

$$y'_i = \frac{1}{2h} ((-y_{i-1} \pm \varepsilon) + (y_{i+1} \pm \varepsilon)) + k h^2 \quad (10.52)$$

In the worst case the round-off errors add up and the total error is:

$$n = \frac{\varepsilon}{h} + k h^2 \quad (10.53)$$

When h is large, the method error kh^2 dominates; when h is small, the round-off error dominates. The error has a minimum when:

$$n' = -\frac{\varepsilon}{h^2} + 2kh = 0. \quad (10.54)$$

It follows that:

$$h = \sqrt[3]{\frac{\varepsilon}{2k}}. \quad (10.55)$$

10.6.1 Example

Below we will look at an example where we vary the precision in three steps:

1. `float16` - 16-bit format: sign 1 bit, exponent 5 bits, mantissa 10 bits
2. `float32` - 32-bit format: sign 1 bit, exponent 8 bits, mantissa 23 bits
3. `float64` - 64-bit format: sign 1 bit, exponent 11 bits, mantissa 52 bits (this is the default precision).

For more about types in `numpy` see the [documentation](#)

Now let's determine the basic round-off error for each type:

```
eps16 = np.finfo(np.float16).eps
eps32 = np.finfo(np.float32).eps
eps64 = np.finfo(np.float64).eps
print(f'The basic round -off error for the types \
`float16`, `float32`, and `float64` is:\n{[eps16, eps32, eps64]}')
```

```
The basic round -off error for the types `float16`, `float32`, and `float64` is:
[np.float16(0.000977), np.float32(1.1920929e -07), np.float64(2.220446049250313e
-16)]
```

As an example let's look at addition: we add half of the basic round-off error `eps16` to the number 1. and convert to the type `float16`; we find that the new value is still equal to the value 1.:

```
(1.+eps16/2).astype('float16')
np.float16(1.0)
```

First let's define the function `exp(x)`, which will give a result with the precision defined by the parameter `dtype`:

```
def fun(x, dtype=float): # function
    return np.exp(-dtype(x)).astype(dtype)
```

Let's also define a function for the analytically determinable derivative (we will need this later to determine the relative error):

```
def f1_fun(x): # "exact" derivative of the function
    return -np.exp(-x)
```

Similarly to addition above, we can also see for the function value that a change in the value x smaller than ϵ leads to the same result:

```
fun(1., dtype=np.float16)
np.float16(0.368)
fun(1+eps16/2, dtype=np.float16)
```

```
| np.float16(0.368)
```

Now let's use the central difference scheme for the first derivative. Here the numbers should be written with precision `dtype`, and with the help of the exact derivative the relative error is also computed:

```
| def f1_CDS(fun, x, h, dtype=np.float64):
|     f1_estimate = (fun(x+h, dtype=dtype) - fun(x -h, dtype=dtype))/(2*dtype(h))
|     f1_exact = f1_fun(x)
|     relative_error = (f1_exact - f1_estimate) / f1_exact
|     return f1_estimate, relative_error
```

Let's look at an example of the derivative at $x = 1.0$ (the Python function returns the value and the relative error):

```
| f1_CDS(fun, x=1., h=.01, dtype=np.float16)
|
| (np.float16( -0.3662), np.float64(0.004535463210798897))
|
| f1_CDS(fun, x=1., h=.01, dtype=np.float64)
```

$$(-0.367885572526119, -1.66667500022579 \cdot 10^{-5}) \quad (10.56)$$

Now let's define the step:

```
| h=0.25**np.arange(30)
| h[:10]
|
| array([1.00000000e+00, 2.50000000e -01, 6.25000000e -02, 1.56250000e -02,
|        3.90625000e -03, 9.76562500e -04, 2.44140625e -04, 6.10351562e -05,
|        1.52587891e -05, 3.81469727e -06])
```

We compute the derivative estimate for various precisions (because of division by 0 we get a warning):

```
| f1_16 = f1_CDS(fun, x=1., h=h, dtype=np.float16)
| f1_32 = f1_CDS(fun, x=1., h=h, dtype=np.float32)
| f1_64 = f1_CDS(fun, x=1., h=h, dtype=np.float64)
|
| /tmp/ipykernel_355507/3908051182.py:2: RuntimeWarning: invalid value encountered in
| divide
|     f1_estimate = (fun(x+h, dtype=dtype) - fun(x -h, dtype=dtype))/(2*dtype(h))
```

We plot the various types as a function of the step size h . First we import the required libraries:

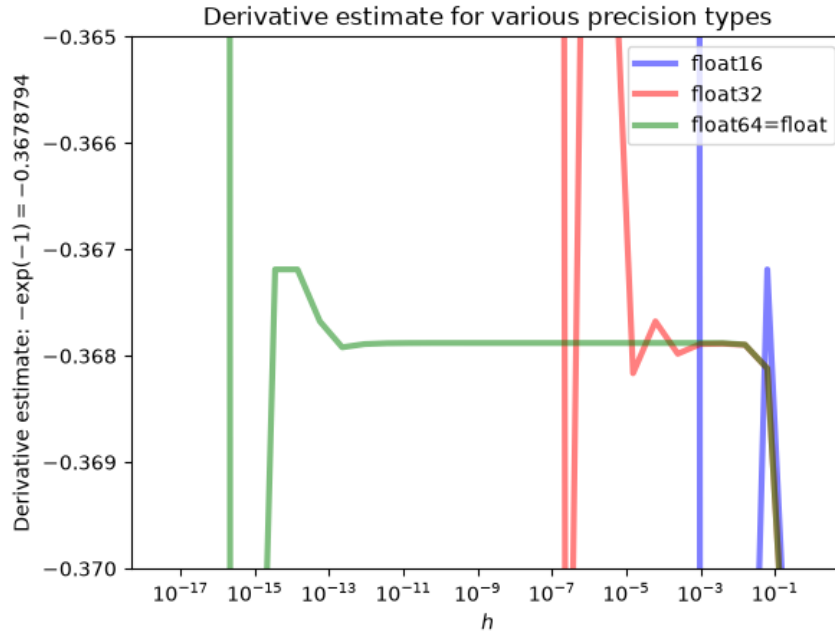
```
| import matplotlib.pyplot as plt
| %matplotlib inline
```

Let's define the figure:

```
| def fig_estimate():
|     plt.semilogx(h, f1_16[0], 'b', lw=3, alpha=0.5, label='float16')
|     plt.semilogx(h, f1_32[0], 'r', lw=3, alpha=0.5, label='float32')
|     plt.semilogx(h, f1_64[0], 'g', lw=3, alpha=0.5, label='float64=float')
|     plt.title('Derivative estimate for various precision types')
|     plt.xlabel('$h$')
|     plt.ylabel('Derivative estimate: $-\exp(-1) = -0.3678794$')
|     plt.ylim(-0.37, -0.365)
|     plt.legend();
```

Let's display it:

```
| fig_estimate()
```



At a relatively large step h the method error dominates, while at a small step the round-off error dominates; the optimal step can be estimated based on:

$$h = \sqrt[3]{\frac{\varepsilon}{2k}}. \quad (10.57)$$

In this particular case:

$$k = -\frac{f'''(x)}{6} = -\frac{-e^{-x}}{6} = \frac{1}{6e} \quad (x = 1) \quad (10.58)$$

It follows that:

$$h = \sqrt[3]{3\varepsilon e}. \quad (10.59)$$

We compute a suitable step for 16-, 32-, and 64-bit format:

```
| np.power(3*eps16*np.exp(1),1/3)
```

$$0.199697177510055 \quad (10.60)$$

```
| np.power(3*eps32*np.exp(1),1/3)
```

$$0.00990623467670444 \quad (10.61)$$

```
| np.power(3*eps64*np.exp(1),1/3)
```

$$1.21885484320102 \cdot 10^{-5} \quad (10.62)$$

The 64-bit format performs best, but even with it a step smaller than about $1\text{e-}5$ is not recommended!

10.7 Complex-step derivative approximation

The complex-step derivative approximation is useful when we have any analytic function defined but do not have an analytical expression for the first derivative; for details see [source](#). The idea stems from the Taylor series expansion of the function $f(x)$, but the step is taken in the direction of the imaginary axis $i h$ ($i = \sqrt{-1}$):

$$f(x + i h) = f(x) + i h \frac{d}{dx} f(x) - \frac{h^2}{2} \frac{d^2}{dx^2} f(x) + O(h^3) \quad (10.63)$$

If we now assume that the function $f(x)$ maps real values to the real axis and that x and h are real values, then we can derive:

$$\mathbf{Im}(f(x + i h)) = h \frac{d}{dx} f(x) + O(h^3) \quad (10.64)$$

$$\mathbf{Re}(f(x + i h)) = f(x) - \frac{h^2}{2} \frac{d^2}{dx^2} f(x) + O(h^3) \rightarrow \mathbf{Re}(f(x + i h)) = f(x) + O(h^2) \quad (10.65)$$

From the first equation (above) we then derive the expression for the first derivative:

$$f'(x) = \frac{\mathbf{Im}(f(x + i h))}{h} + O(h^2) \quad (10.66)$$

The result is in a way very surprising. Above, with a one-step forward scheme, we managed to obtain accuracy $O(h^1)$, while here it is $O(h^2)$!

An important advantage of the complex-step method is that it **has no problem with subtracting nearly equal numbers** (cancellation error), which plagues classical difference methods at very small steps h . Therefore we can choose an extremely small h (e.g., 10^{-200}) and obtain a practically exact result (down to machine precision).

If we now look at the use of the method on the example from above, $\exp(-x)$, let's first recall the exact result at the value $x = 1$:

```
| f1_exact0
```

$$-0.367879441171442 \quad (10.67)$$

When called with a step $h = 0.1$ we get two (three) correct digits (similarly to the central difference scheme), and by reducing the step to one tenth we would roughly double the number of correct digits.

```
| h1_complex = 0.1
| y1_complex = np.exp(-(1.+1.j*h1_complex))
| f1_complex = np.imag(y1_complex)/h1_complex
| f1_complex
```

$$-0.367266615262709 \quad (10.68)$$

Since we have no problem with subtracting nearly equal numbers, we can reduce the step extremely and obtain an exact result (down to machine precision):

```
| h1_complex = 1e-10
| y1_complex = np.exp(-(1.+1.j*h1_complex))
| f1_complex = np.imag(y1_complex)/h1_complex
| f1_complex
```

$$-0.367879441171442 \quad (10.69)$$

```
| np.sin(2*x0*1e -5j)
|
| array([0.+0.e+00j, 0.+2.e -05j, 0.+4.e -05j])
|
| np.sin(x0**2+2*x0*1e -5j)
|
| array([ 0.          +0.00000000e+00j,  0.84147098+1.08060461e -05j,
|        -0.7568025  -2.61457448e -05j])
```

Let's also look at an example of differentiating the function `sin(x**2)`. From the code and the figure below we can see a substantially faster convergence of the complex-step method. Why does the method work so well? If we substitute $x + ih$ into x^2 , we get: $(x + ih)^2 = x^2 + 2ixh - h^2$

Since h is very small, h^2 is negligible in the real part, while the imaginary part $2xh$ carries linearly through the sine. In this way we obtain an approximation of the theoretical derivative $2x \cos(x^2)$. The method *sees* the derivative in the imaginary part without subtraction, so no round-off error occurs.

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

# Function: sin(x^2)
def f(x):
    return np.sin(x**2)

# Analytical derivative: 2x * cos(x^2)
def df_analytical(x):
    return 2*x * np.cos(x**2)

x0 = 1.5
exact = df_analytical(x0)

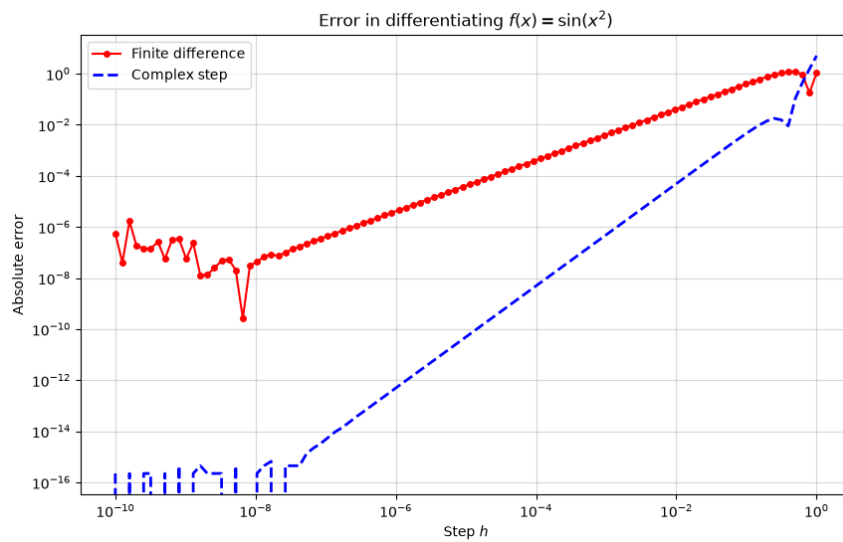
h_vals = np.logspace( -10, 0, 100)
err_fd = []
err_cs = []

for h in h_vals:
    # Finite difference (Forward)
    fd_approx = (f(x0 + h) - f(x0)) / h
    err_fd.append(abs(fd_approx - exact))

    # Complex step
    # f(x + ih) = sin((x+ih)^2) \approx sin(x^2 + 2ixh)
    #             \approx sin(x^2) + i * 2xh * cos(x^2)
    # Imag/h gives us exactly 2x*cos(x^2)
    cs_approx = f(x0 + 1j*h).imag / h
    err_cs.append(abs(cs_approx - exact))

plt.figure(figsize=(10, 6))
plt.loglog(h_vals, err_fd, 'r -o', label='Finite difference', markersize=4)
plt.loglog(h_vals, err_cs, 'b - -', label='Complex step', linewidth=2)

plt.xlabel('Step $h$')
plt.ylabel('Absolute error')
plt.title(r'Error in differentiating $f(x)=\sin(x^2)$')
plt.grid(True, which="both", alpha=0.5)
plt.legend()
plt.show()
```



10.8 Additional

[Lecture video on the topic of numerical differentiation.](#)

Chapter 11

Numerical integration

11.1 Introduction

In this chapter, for a given function $f(x)$ we will compute the definite integral:

$$\int_a^b f(x) \, dx \quad (11.1)$$

where a and b are the limits of integration, and $f(x)$ are the function values obtained either from a table of values or from an analytical function.

In numerical integration, the integral is estimated by I , and the following holds:

$$\int_a^b f(x) \, dx = I + E, \quad (11.2)$$

where E is the error of the integral estimate.

We will compute the numerical integral based on a discrete sum:

$$I = \sum_{i=0}^m A_i f(x_i), \quad (11.3)$$

where A_i are the weights, x_i are the nodes on the interval $[a, b]$, and $m + 1$ is the number of nodes.

We will look at two different approaches to numerical integration:

1. the *Newton-Cotes approach*, which is based on equidistant nodes (a constant integration step), and
2. the *Gaussian integration approach*, where the nodes are placed so as to achieve exactness for polynomials.

11.1.1 Motivational example

For numerical integration, we will use a concrete example to guide us:

$$\int_1^2 x \sin(x) \, dx \quad (11.4)$$

Let us prepare the nodes. Let the basic step be $h = 0.25$; in this case we have four subintervals and five nodes, at step $2h$ there are three nodes, and at step $4h$ only two (the endpoints):

```
import numpy as np
xg, hg = np.linspace(1, 2, 100, retstep=True) # dense points (for plotting)
x2v, h2v = np.linspace(1, 2, 2, retstep=True) # step h2v = 1 (2 nodes)
x3v, h3v = np.linspace(1, 2, 3, retstep=True) # step h3v = 0.5 (3 nodes)
x4v, h4v = np.linspace(1, 2, 4, retstep=True) # step h4v = 0.33.. (4 nodes)
x5v, h5v = np.linspace(1, 2, 5, retstep=True) # step h5v = 0.25 (5 nodes)
```

Let us also prepare the function values:

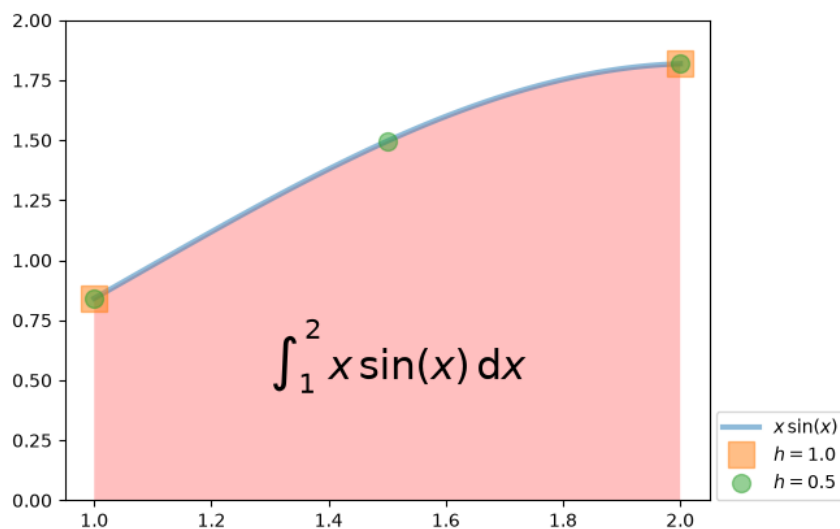
```
yg = xg * np.sin(xg)
y2v = x2v * np.sin(x2v)
y3v = x3v * np.sin(x3v)
y4v = x4v * np.sin(x4v)
y5v = x5v * np.sin(x5v)
```

Let us prepare a plot of the data:

```
import matplotlib.pyplot as plt
%matplotlib inline
def fig_motivation():
    plt.fill_between(xg, yg, alpha=0.25, facecolor='r')
    plt.annotate(r'$\int_1^2 x \sin(x) \, dx$', (1.3, 0.5), fontsize=22)
    plt.plot(xg, yg, lw=3, alpha=0.5, label=r'$x \sin(x)$')
    plt.plot(x2v, y2v, 's', alpha=0.5, label=f'$h={h2v}$', markersize=14)
    plt.plot(x3v, y3v, 'o', alpha=0.5, label=f'$h={h3v}$', markersize=10)
    plt.legend(loc=(1.01, 0))
    plt.ylim(0, 2)
    plt.show()
```

Let us display the data:

```
fig_motivation()
```



Let us compute the exact result analytically:

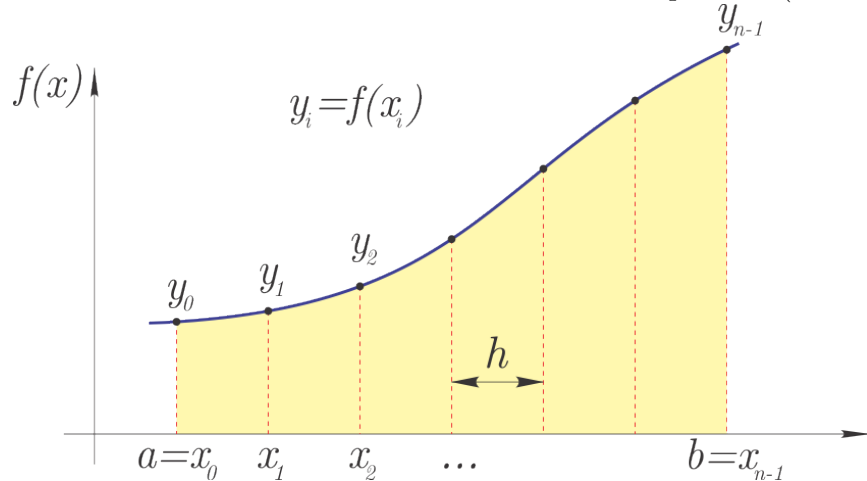
```
import sympy as sym
sym.init_printing()
x = sym.symbols('x')
I_exact = float(sym.integrate(x*sym.sin(x), (x, 1, 2)).evalf())
I_exact
```

$$1.44042242098021$$

(11.5)

11.2 Newton-Cotes approach

The figure shows a general case in which the distance between the nodes x_i is equal to h (this is an



equidistant partition).

In this chapter we will first look at the trapezoidal and the composite trapezoidal rule, and later the Simpson's and Romberg methods will follow.

11.2.1 Trapezoidal rule

The trapezoidal rule interpolates the values of the integrand $f(x)$ on the (sub)interval $[x_0, x_1]$ with a linear function. For two nodes, this means that we approximate the area under the graph of the function as:

$$I_{\text{trapezno}} = \sum_{i=0}^1 A_i f(x_i) = \frac{h}{2} \cdot (f(x_0) + f(x_1)). \quad (11.6)$$

And the weights are:

$$A_0 = A_1 = \frac{1}{2} h. \quad (11.7)$$

Numerical implementation

The numerical implementation is:

```
def trapezoidal(y, h):
    """
    Trapezoidal rule of integration.

    :param y: function values
    :param h: integration step
    """
    return (y[0] + y[-1])*h/2
```

Numerical example

In our concrete case this means that we weight the first and last function value by $h/2$. In our example $h = 1$:

```
I_trapezoidal = trapezoidal(y2v, h=h2v)
I_trapezoidal
```

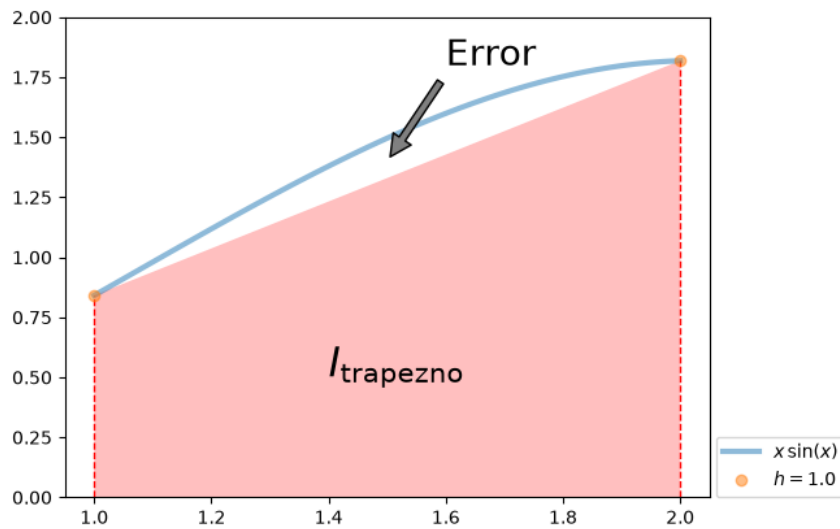
$$1.33003291922963 \quad (11.8)$$

Let us prepare a figure:

```
def fig_trapezoidal():
    plt.fill_between(x2v, y2v, alpha=0.25, facecolor='r')
    plt.vlines(x2v, 0, y2v, color='r', linestyle='dashed', lw=1)
    plt.annotate('$I_{\mathrm{trapezno}}$', (1.4, 0.5), fontsize=22)
    plt.annotate('Error', fontsize=20, xy=(1.5, 1.4), xytext=(1.6, 1.8),
        arrowprops=dict(facecolor='gray', shrink=0.05))
    plt.plot(xg, yg, lw=3, alpha=0.5, label='$x\\sin(x)$')
    plt.plot(x2v, y2v, 'o', alpha=0.5, label=f'$h={h2v}$')
    plt.legend(loc=(1.01, 0))
    plt.ylim(0, 2)
    plt.show()
```

We display:

```
fig_trapezoidal()
```



Error of the trapezoidal rule

The difference between the analytical value of the integral and the numerical approximation I is the error of the method:

$$E = \int_a^b f(x) dx - I, \quad (11.9)$$

If the function $f(x)$ is at least twice differentiable, one can derive (see, e.g., the reference: Burden, Faires, Burden: Numerical Analysis 10th Ed) an error estimate that applies only to the trapezoidal approximation over the entire integration interval:

$$E_{\text{trapezno}} = -\frac{h^3}{12} f''(\xi), \quad (11.10)$$

where $h = b - a$ and ξ is an unknown value on the interval $[a, b]$.

11.2.2 Composite trapezoidal rule

If we divide the interval $[a, b]$ into n subintervals and apply the trapezoidal rule of integration on each of them, we speak of the *composite trapezoidal rule*.

In this case, for each subinterval i we apply the trapezoidal rule, and thus for the bounds of the subinterval x_i and x_{i+1} we use the weights $A_i = A_{i+1} = h/2$. Since the interior nodes are doubled, it follows that:

$$A_0 = A_n = \frac{h}{2} \quad \text{and for the other nodes: } A_i = h. \quad (11.11)$$

Here we have assumed subintervals of equal width:

$$h = \frac{x_n - x_0}{n} \quad (11.12)$$

It therefore follows that:

$$I_{\text{trapezno sest}} = \sum_{i=0}^n A_i f(x_i) = \left(\frac{y_0}{2} + y_1 + y_2 + \cdots + y_{n-1} + \frac{y_n}{2} \right) h. \quad (11.13)$$

Numerical implementation

The numerical implementation is:

```
def trapezoidal_comp(y, h):
    """
    Composite trapezoidal rule of integration.

    :param y: function values
    :param h: integration step
    """
    return (np.sum(y) - 0.5*y[0] - 0.5*y[-1])*h
```

Numerical example

Above we already prepared the data for two subintervals (three nodes):

```
x3v
array([1. , 1.5, 2. ])
h3v
```

$$0.5 \quad (11.14)$$

Let us compute the estimate of the integral with the composite trapezoidal rule:

```
I_trapezoidal_comp = trapezoidal_comp(y3v, h=h3v)
I_trapezoidal_comp
```

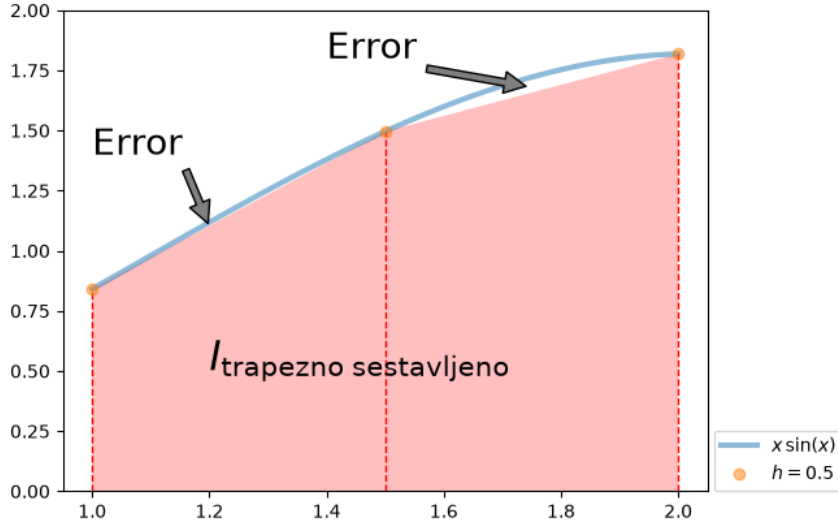
$$1.41313769956786 \quad (11.15)$$

Let us prepare a figure:

```
def fig_trapezoidal_comp():
    plt.fill_between(x3v, y3v, alpha=0.25, facecolor='r')
    plt.vlines(x3v, 0, y3v, color='r', linestyle='dashed', lw=1)
    plt.annotate('$I_{\mathrm{trapezno\\;sestavljeno}}$', (1.2, 0.5), fontsize=22)
    plt.annotate('Error', fontsize=20, xy=(1.75, 1.68), xytext=(1.4, 1.8),
        arrowprops=dict(facecolor='gray', shrink=0.05))
    plt.annotate('Error', fontsize=20, xy=(1.2, 1.1), xytext=(1., 1.4),
        arrowprops=dict(facecolor='gray', shrink=0.05))
    plt.plot(xg, yg, lw=3, alpha=0.5, label='$x\\,\\sin(x)$')
    plt.plot(x3v, y3v, 'o', alpha=0.5, label=f'$h={h3v}$')
    plt.legend(loc=(1.01, 0))
    plt.ylim(0, 2)
    plt.show()
```

We display:

```
| fig_trapezoidal_comp()
```



Error of the composite trapezoidal rule

The error of the composite trapezoidal rule stems from the error of the trapezoidal rule; here we make such an error n times.

Since $h \cdot n = b - a$ holds, we derive the error of the composite trapezoidal rule as:

$$E_{\text{trapezno sest}} = -\frac{h^2(b-a)}{12} f''(\eta), \quad (11.16)$$

η is a value on the interval $[a, b]$. The error is of second order $\mathcal{O}(h^2)$.

A better approximation of the integral

Next we will look at the so-called **Richardson extrapolation**, in which we compute a better approximation from the integral estimates with step h and $2h$.

If we compute the integral I numerically at two different steps h and $2h$, the following holds:

$$\int_a^b f(x) dx = I_h + E_h = I_{2h} + E_{2h}, \quad (11.17)$$

where I_h and I_{2h} are the approximations of the integral with step h and $2h$, and E_h and E_{2h} are the error estimates at step h and $2h$. We derive $I_{2h} - I_h = E_h - E_{2h}$

Next we write:

$$E_h = -\frac{h^2(b-a)}{12} f''(\eta) = h^2 K. \quad (11.18)$$

Under the assumption that $f''(\eta)$ is the same at step h and $2h$ (η is actually different at step h and $2h$), we write:

$$E_{2h} = -\frac{(2h)^2(b-a)}{12} f''(\eta) = 4h^2 K \quad (11.19)$$

It follows that:

$$I_{2h} - I_h = -3K h^2. \quad (11.20)$$

Now we can estimate the error at step h :

$$E_h = h^2 K = \frac{I_h - I_{2h}}{3}. \quad (11.21)$$

Based on the error estimate, we can compute a better numerical approximation I_h^* :

$$I_h^* = I_h + \frac{1}{3}(I_h - I_{2h}) \quad (11.22)$$

or

$$I_h^* = \frac{4}{3}I_h - \frac{1}{3}I_{2h} \quad (11.23)$$

Numerical example

We previously computed the integral with the trapezoidal rule at step $h = 1$ and at step $h = 0.5$ already; the results were:

```
[I_trapezoidal, I_trapezoidal_comp]
[1.33003291922963, 1.41313769956786]
(11.24)
```

Using the formula above, we compute a better approximation:

```
I_trapezoidal_better = 4/3*I_trapezoidal_comp - 1/3*I_trapezoidal
print(f'Exact result:      {I_exact}\nBetter approximation:
{I_trapezoidal_better}')

Exact result:      1.4404224209802097
Better approximation: 1.4408392930139313
```

The composite trapezoidal rule is also implemented in the `numpy` package, via the `numpy.trapezoid` function (`numpy.trapz` in older versions):

```
| trapezoid(y, x=None, dx=1.0, axis=-1)
```

- `y` represents the table of function values,
- `x` is an optional parameter and defines the nodes; if the parameter is not defined, equidistant nodes spaced by `dx` are assumed,
- `dx` defines the (constant) integration step, with a default value of 1,
- `axis` defines the *axis* along which to integrate (in case `y` is a multidimensional numeric array).

The function returns the computed integral according to the composite trapezoidal rule. More information can be found in the [documentation](#).

numpy implementation of the composite trapezoidal rule

Let us look at an example:

```
%%timeit
I_trapezoidal_np = np.trapezoid(y3v, dx=h3v)
I_trapezoidal_np
1.41313769956786
(11.25)
```

11.2.3 Simpson's and other methods

Above we looked at the trapezoidal rule, which is based on a linear interpolating function on an individual subinterval. With higher-order interpolation we can derive further integration methods.

We need to compute:

$$\int_a^b f(x) dx. \quad (11.26)$$

We tabulate the integrand $f(x)$ and interpolate the table with the Lagrange interpolating polynomial $P_n(x)$ of degree n :

$$P_n(x) = \sum_{i=0}^n f(x_i) l_i(x), \quad (11.27)$$

where $y_i = f(x_i)$ are the function values at the nodes x_i and the Lagrange polynomial l_i is defined as:

$$l_i(x) = \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j}. \quad (11.28)$$

For the numerical computation of the integral $\int_a^b f(x) dx$ (the limits are: $a = x_0$, $b = x_n$), instead of the function $f(x)$ we substitute the interpolating polynomial $P_n(x)$ into the integral:

$$I = \int_{x_0}^{x_n} P_n(x) dx = \int_{x_0}^{x_n} \sum_{i=0}^n f(x_i) l_i(x) dx. \quad (11.29)$$

Since integration is a linear operation, we can swap the integration and the sum:

$$I = \sum_{i=0}^n f(x_i) \underbrace{\int_{x_0}^{x_n} l_i(x) dx}_{A_i}. \quad (11.30)$$

We integrate the Lagrange polynomial and obtain the weights A_i :

$$A_i = \int_{x_0}^{x_n} l_i(x) dx \quad (11.31)$$

Derivation of the trapezoidal rule using Lagrange polynomials

Let us look at how we can computationally derive the weights A_i for the case of the trapezoidal rule using a first-degree Lagrange interpolating polynomial.

First, let us define the variable x , the nodes x_0 and x_1 , and the step h in symbolic form:

```
| x, x0, x1, h = sym.symbols('x x0, x1, h')
```

Let us prepare a Python function that returns, in symbolic form, a list of n coefficients of the Lagrange polynomials $[l_0(x), l_1(x), \dots, l_{n-1}(x)]$ of degree $n - 1$:

```
| def lagrange(n, x, nodes_prefix='x'):
|     if isinstance(nodes_prefix, str):
|         nodes = sym.symbols(f'{{nodes_prefix}}:{{n}}')
|     coeffs = []
|     for i in range(0, n):
|         numer = []
|         denom = []
|
|         for j in range(0, n):
```

```

        if i == j:
            continue

        numer.append(x      - nodes[j])
        denom.append(nodes[i] - nodes[j])

    numer = sym.Mul(*numer)
    denom = sym.Mul(*denom)

    coeffs.append(numer/denom)
return coeffs

```

First, let us look at the Lagrange polynomials for linear interpolation ($n = 2$):

```

lag = lagrange(2, x)
lag

```

$$\left[\frac{x - x_1}{x_0 - x_1}, \frac{x - x_0}{-x_0 + x_1} \right] \quad (11.32)$$

Now we integrate the Lagrange polynomial $l_0(x)$ over the entire interval:

```

int0 = sym.integrate(lag[0], (x, x0, x1))
int0

```

$$-\frac{x_0^2}{2x_0 - 2x_1} + \frac{x_0x_1}{x_0 - x_1} + \frac{x_1^2}{2x_0 - 2x_1} - \frac{x_1^2}{x_0 - x_1} \quad (11.33)$$

We simplify the expression and obtain:

```

int1 = int0.factor()
int1

```

$$-\frac{x_0 - x_1}{2} \quad (11.34)$$

Since the width of the subinterval is constant, $x_1 = h_0 + h$, we perform the substitution:

```

substitutions = {x1: x0+h}
int1.subs(substitutions)

```

$$\frac{h}{2} \quad (11.35)$$

The steps above for the Lagrange polynomial $l_0(x)$ can be generalized to a list of Lagrange polynomials:

```

x, x0, x1, h = sym.symbols('x, x0, x1, h')
substitutions = {x1: x0+h}
A_trapezoid = [sym.integrate(li, (x, x0, x1)).factor().subs(substitutions)
               for li in lagrange(2, x)] # for each Lagrange polynomial `li` in the
               list lagrange(2,x)
A_trapezoid

```

$$\left[\frac{h}{2}, \frac{h}{2} \right] \quad (11.36)$$

We derived the weights that we used in the trapezoidal method:

$$A_0 = h/2 \quad A_1 = h/2. \quad (11.37)$$

The trapezoidal rule is:

$$I_{\text{trapezno}} = \frac{h}{2} (y_0 + y_1) \quad (11.38)$$

The error estimate is (reference: Burden, Faires, Burden: Numerical Analysis 10th Ed):

$$E_{\text{trapezno}} = -\frac{h^3}{12} f''(\xi). \quad (11.39)$$

Computing the weights for Simpson's rule

After showing the concise computation for the trapezoidal rule above, we can proceed similarly for quadratic interpolation over three points ($n = 3$).

The computation of the weights is analogous to the one above:

```
x, x0, x1, x2, h = sym.symbols('x, x0, x1, x2, h')
substitutions = {x1: x0+h, x2: x0+2*h}
A_Simpson1_3 = [sym.integrate(li, (x, x0, x2)).factor().subs(substitutions).factor()
                 for li in lagrange(3, x)]
A_Simpson1_3
```

$$\left[\frac{h}{3}, \frac{4h}{3}, \frac{h}{3} \right] \quad (11.40)$$

Simpson's rule (this rule is also called Simpson's 1/3 rule) is:

$$I_{\text{Simpsonovo}} = \frac{h}{3} (y_0 + 4y_1 + y_2) \quad (11.41)$$

The error estimate is (reference: Burden, Faires, Burden: Numerical Analysis 10th Ed):

$$E_{\text{Simpsonovo}} = -\frac{h^5}{90} f^{(4)}(\xi). \quad (11.42)$$

It should be pointed out that the error is locally of fifth order $\mathcal{O}(h^5)$ and is defined by the unknown value of the fourth derivative $f^{(4)}(\xi)$; consequently, this rule is exact for polynomials of degree 3 or less.

Example of use:

```
I_Simps = h3v/3 * np.sum(y3v * [1, 4, 1])
I_Simps
```

$$1.44083929301393 \quad (11.43)$$

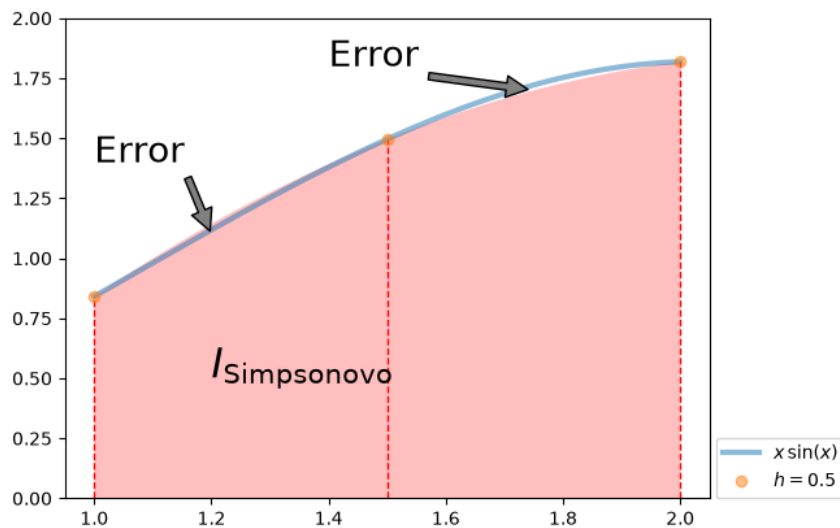
Let us prepare a figure. Since Simpson's rule is based on quadratic interpolation, we must first prepare the interpolating polynomial (we use `scipy.interpolate`):

```
from numpy.polynomial import Polynomial # interpolating polynomial through the nodes
```

```
def fig_simpson():
    y_interpolate = Polynomial.fit(x3v, y3v, deg=len(x3v) -1)
    plt.fill_between(xg, y_interpolate(xg), alpha=0.25, facecolor='r')
    plt.vlines(x3v, 0, y3v, color='r', linestyle='dashed', lw=1)
    plt.annotate('$I_{\mathrm{Simpsonovo}}$', (1.2, 0.5), fontsize=22)
    plt.annotate('Error', fontsize=20, xy=(1.75, 1.7), xytext=(1.4, 1.8),
        arrowprops=dict(facecolor='gray', shrink=0.05))
    plt.annotate('Error', fontsize=20, xy=(1.2, 1.1), xytext=(1., 1.4),
        arrowprops=dict(facecolor='gray', shrink=0.05))
    plt.plot(xg, yg, lw=3, alpha=0.5, label='$x\\sin(x)$')
    plt.plot(x3v, y3v, 'o', alpha=0.5, label=f'$h={h3v}$')
    plt.legend(loc=(1.01, 0))
    plt.ylim(0, 2)
    plt.show()
```

We display:

```
fig_simpson()
```



scipy.integrate.newton_cotes

The coefficients of the Newton-Cotes integration approach can also be obtained using `scipy.integrate.newton_cotes()` ([documentation](#)):

```
newton_cotes(rn, equal=0)
```

where the parameters are:

- **rn**, which defines the number of subintervals (a non-constant step is also possible, see the [documentation](#)),
- **equal**, which defines whether equally wide subintervals are required.

The function returns a tuple, in which the first element represents the numeric array of weights and the second element the error estimate.

Let us look at an example:

```
from scipy import integrate
integrate.newton_cotes(2)
```

```
(array([0.33333333, 1.33333333, 0.33333333]), -0.011111111111111112)
```

Computing the weights for Simpson's 3/8 rule

We can continue with cubic interpolation over four points ($n = 4$):

```
x, x0, x1, x2, x3, h = sym.symbols('x, x0, x1, x2, x3, h')
substitutions = {x1: x0+h, x2: x0+2*h, x3: x0+3*h}
A_Simpson3_8 = [sym.integrate(li, (x, x0, x3)).factor().subs(substitutions).factor()
                 for li in lagrange(4, x)]
A_Simpson3_8
```

$$\left[\frac{3h}{8}, \frac{9h}{8}, \frac{9h}{8}, \frac{3h}{8} \right] \quad (11.44)$$

Simpson's 3/8 rule is:

$$I_{\text{Simpsonovo } 3/8} = \frac{3h}{8} (y_0 + 3y_1 + 3y_2 + y_3) \quad (11.45)$$

The error estimate is (reference: Burden, Faires, Burden: Numerical Analysis 10th Ed):

$$E_{\text{Simpsonovo } 3/8} = -\frac{3h^5}{80} f^{(4)}(\xi). \quad (11.46)$$

Let us look at an example of use. We use the previously prepared table of function values at four points:

```
y4v
array([0.84147098, 1.2959172 , 1.65901326, 1.81859485])
I_Simps38 = 3*h4v/8 * np.sum(y4v * [1, 3, 3, 1])
I_Simps38
```

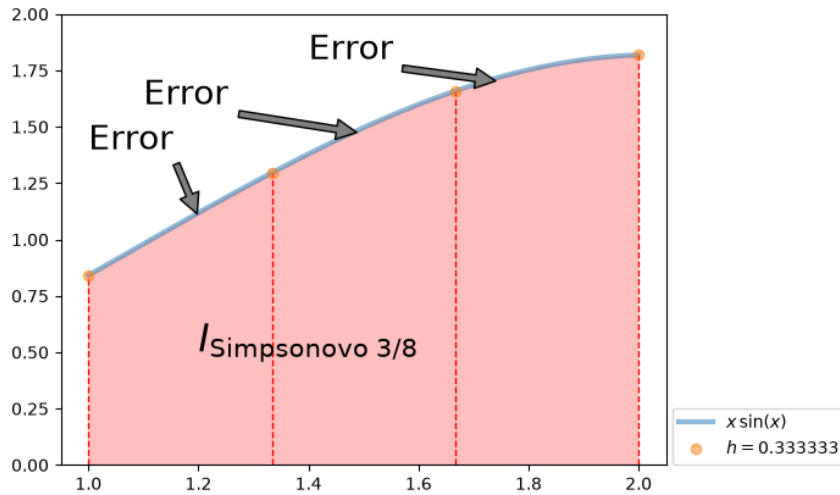
$$1.44060715408392 \quad (11.47)$$

Let us also prepare a plot:

```
def fig_simpson38():
    y_interpolate = Polynomial.fit(x4v, y4v, deg=len(x4v) -1)
    plt.fill_between(xg, y_interpolate(xg), alpha=0.25, facecolor='r')
    plt.vlines(x4v, 0, y4v, color='r', linestyle='dashed', lw=1)
    plt.annotate('$I_{\mathrm{Simpsonovo};3/8}$', (1.2, 0.5), fontsize=22)
    plt.annotate('Error', fontsize=20, xy=(1.75, 1.7), xytext=(1.4, 1.8),
                 arrowprops=dict(facecolor='gray', shrink=0.05))
    plt.annotate('Error', fontsize=20, xy=(1.5, 1.47), xytext=(1.1, 1.6),
                 arrowprops=dict(facecolor='gray', shrink=0.05))
    plt.annotate('Error', fontsize=20, xy=(1.2, 1.1), xytext=(1., 1.4),
                 arrowprops=dict(facecolor='gray', shrink=0.05))
    plt.plot(xg, yg, lw=3, alpha=0.5, label='$x$, $\sin(x)$')
    plt.plot(x4v, y4v, 'o', alpha=0.5, label=f'$h={h4v:.6f}$')
    plt.legend(loc=(1.01, 0))
    plt.ylim(0, 2)
    plt.show()
```

We display:

```
fig_simpson38()
```



11.2.4 Composite Simpson's rule

We divide the interval $[a, b]$ into an even number n of subintervals of equal width $h = (b - a)/n$, which defines the nodes $x_i = a + i h$ for $i = 0, 1, \dots, n$. In this case we write the composite Simpson's rule:

$$\int_a^b f(x) dx = \frac{h}{3} \left(f(x_0) + 4 \sum_{i=1}^{n/2} f(x_{2i-1}) + 2 \sum_{i=1}^{n/2-1} f(x_{2i}) + f(x_n) \right) - \underbrace{\frac{b-a}{180} h^4 f^{(4)}(\eta)}_{E_{\text{Sestavljeno Simpsonovo } 1/3}}, \quad (11.48)$$

where η is an unknown value on the interval $[a, b]$. The error is of fourth order $\mathcal{O}(h^4)$.

Numerical implementation:

```
def simpson_comp(y, h):
    """
    Composite Simpson's rule of integration.

    :param y: function values
    :param h: integration step
    """
    return h/3 * (y[0] + 4*np.sum(y[1: -1:2]) + 2*np.sum(y[2: -1:2]) + y[-1])
```

```
I_Simps_comp = simpson_comp(y5v, h=h5v)
I_Simps_comp
```

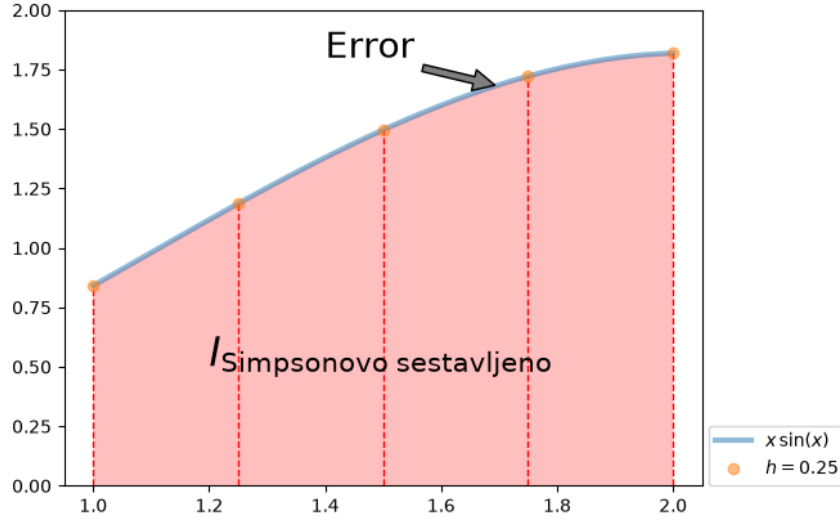
1.44044796026391 (11.49)

Let us prepare a figure:

```
def fig_simpson_comp():
    y_interpolate = Polynomial.fit(x5v, y5v, deg=len(x5v) - 1)
    plt.fill_between(xg, y_interpolate(xg), alpha=0.25, facecolor='r')
    plt.vlines(x5v, 0, y5v, color='r', linestyle='dashed', lw=1)
    plt.annotate('$I_{\mathrm{Simpsonovo}}$', (1.2, 0.5), fontsize=22)
    plt.annotate('Error', fontsize=20, xy=(1.70, 1.68), xytext=(1.4, 1.8),
        arrowprops=dict(facecolor='gray', shrink=0.05))
    plt.plot(xg, yg, lw=3, alpha=0.5, label='$x \sin(x)$')
    plt.plot(x5v, y5v, 'o', alpha=0.5, label=f'$h={h5v}$')
    plt.legend(loc=(1.01, 0))
    plt.ylim(0, 2)
    plt.show()
```

We display:

```
| fig_simpson_comp()
```



A better estimate of the integral

The error of the composite Simpson's method is defined by:

$$E_{\text{Sestavljeno Simpsonovo } 1/3} = -\frac{b-a}{180} h^4 f^{(4)}(\eta), \quad (11.50)$$

where η is an unknown value on the interval $[a, b]$.

We determine an improved estimate of the integral in a manner similar to the composite trapezoidal method; we estimate the integral I at two different steps h and $2h$, and the following holds exactly:

$$\int_a^b f(x) dx = I_h + E_h = I_{2h} + E_{2h}, \quad (11.51)$$

where I_h is the approximation of the integral with step h and E_h is the error estimate at step h ; the same holds analogously for I_{2h} and E_{2h} .

If we assume that $f^{(4)}(\eta)$ is the same in both cases, we can determine the difference $I_{2h} - I_h = E_h - E_{2h}$.

Next we write:

$$E_h = -\frac{b-a}{180} h^4 f^{(4)}(\eta) = h^4 K. \quad (11.52)$$

Under the assumption that $f^{(4)}(\eta)$ is the same at step h and $2h$ (η is actually different at step h and $2h$), we write:

$$E_{2h} = -\frac{(b-a)}{180} (2h)^4 f^{(4)}(\eta) = 16 h^4 K \quad (11.53)$$

It follows that:

$$I_{2h} - I_h = -15K h^4. \quad (11.54)$$

Now we can estimate the error at step h :

$$E_h = h^4 K = \frac{I_h - I_{2h}}{15}. \quad (11.55)$$

Based on the error estimate, we can compute a better approximation I_h^* :

$$I_h^* = I_h + \frac{1}{15} (I_h - I_{2h}) \quad (11.56)$$

or

$$I_h^* = \frac{16}{15} I_h - \frac{1}{15} I_{2h} \quad (11.57)$$

Numerical example

We previously computed the integral with Simpson's rule at step $h = 0.5$ and at step $h = 0.25$ already; the results were:

```
| [I_Simps, I_Simps_comp]
                                     [1.44083929301393, 1.44044796026391]
                                     (11.58)
```

Using the formula above, we compute a better approximation:

```
| I_Simps_better = 16/15*I_Simps_comp - 1/15*I_Simps
| print(f'Exact result:      {I_exact}\nBetter approximation: {I_Simps_better}')
|
| Exact result:      1.4404224209802097
| Better approximation: 1.4404218714139077
```

We obtain a better numerical approximation, but we lose the error estimate!

Simpson's method in `scipy.integrate.simpson`

The composite Simpson's rule is implemented in `scipy` in `scipy.integrate.simpson()` ([documentation](#)):

```
| simpson(y, x=None, *, dx=1.0, axis=-1)
```

where the parameters are:

- `y` the table of function values that we integrate,
- `x` the nodes, an optional parameter; if `x=None`, equidistant subintervals of width `dx` are assumed,
- `dx` the width of the equidistant subintervals, i.e., the integration step,
- `axis` the integration axis (important in the case of a multidimensional numeric array).

In older SciPy versions the function was called `simps` and also had the parameter `even`; both were removed in SciPy 1.14.

Let us look at an example; first we import the `simpson` function:

```
| from scipy.integrate import simpson
|
| %%timeit
| simpson(y5v, dx=h5v)
```

1.44044796026391 (11.59)

11.2.5 Romberg method

The Romberg method is based on Richardson extrapolation. Let us assume that we integrate the integral $\int_a^b f(x)dx$ on the interval $[a, b]$, which we divide into 2^{n-1} subintervals ($n = 1, 2, 4, 8, \dots$).

We denote the result of the trapezoidal rule at $n = 1$ by $R_{\underbrace{1}_n, \underbrace{1}_j}$, where j denotes the accuracy of the obtained result $\mathcal{O}(h^{2j})$.

If we apply the trapezoidal integration rule at $n = 1, 2, 4, \dots$ subintervals, we compute:

- $R_{1,1}$, step $h_1 = h$, order of accuracy $\mathcal{O}(h_1^2)$,
- $R_{2,1}$, step $h_2 = h/2$, order of accuracy $\mathcal{O}(h_2^2)$,
- $R_{3,1}$, step $h_3 = h/4$, order of accuracy $\mathcal{O}(h_3^2)$,
- $R_{4,1}$, step $h_4 = h/8$, order of accuracy $\mathcal{O}(h_4^2)$,
- $\dots$
- $R_{n,1}$, step $h_n = h/(2^{n-1})$, order of accuracy $\mathcal{O}(h_n^2)$.

Based on Richardson extrapolation we compute a better approximation

- $R_{2,2} = R_{2,1} + \frac{1}{3} (R_{2,1} - R_{1,1})$, step $h_2 = h/2$, order of accuracy $\mathcal{O}(h_2^4)$,
- $R_{3,2} = R_{3,1} + \frac{1}{3} (R_{3,1} - R_{2,1})$, step $h_3 = h/4$, order of accuracy $\mathcal{O}(h_3^4)$,
- $\dots$
- $R_{n,2} = R_{n,1} + \frac{1}{3} (R_{n,1} - R_{n-1,1})$, step $h_n = h/(2^{n-1})$, order of accuracy $\mathcal{O}(h_n^4)$.

Then we continue with Richardson extrapolation:

- $R_{3,3} = R_{3,2} + \frac{1}{15} (R_{3,2} - R_{2,2})$, step $h_3 = h/4$, order of accuracy $\mathcal{O}(h_3^6)$,
- $\dots$
- $R_{n,3} = R_{n,2} + \frac{1}{15} (R_{n,2} - R_{n-1,2})$, step $h_n = h/(2^{n-1})$, order of accuracy $\mathcal{O}(h_n^6)$.

Richardson extrapolation can be generalized:

- $R_{n,j} = R_{n,j-1} + \frac{1}{4^j - 1} (R_{n,j-1} - R_{n-1,j-1})$, step $h_n = h/(2^{n-1})$, order of accuracy $\mathcal{O}(h_n^{2j})$

Here, the better approximation $R_{2,2}$ at step $h/2$ is equal to the result obtained by Simpson's method at step $h/2$. Similarly, the better approximation $R_{3,2}$ at step $h/4$ is equal to the numerical integral of Simpson's method at step $h/4$. And further, $R_{3,3}$ is equal to the corrected approximation of Simpson's method at step $h/4$.

Let us prepare the numerical data:

```
x9v, h9v = np.linspace(1, 2, 9, retstep=True) # step h9v = 0.125 (9 nodes)
y9v = x9v * np.sin(x9v)
```

Let us look at the example from above. First, we compute the integral with the composite trapezoidal method at different steps (second-order error):

```
## O(h^2)
R1_1 = trapezoidal_comp(y9v[:8], 8*h9v) # h=1.0
R2_1 = trapezoidal_comp(y9v[:4], 4*h9v) # h=0.5
R3_1 = trapezoidal_comp(y9v[:2], 2*h9v) # h=0.25
R4_1 = trapezoidal_comp(y9v, h9v) # h=0.125
[R1_1, R2_1, R3_1, R4_1]
```

$$[1.33003291922963, 1.41313769956786, 1.4336203950899, 1.43872310575291] \quad (11.60)$$

Then we compute better approximations (we obtain a fourth-order error):

```
## O(h^4)
R2_2 = R2_1 + 1/3 * (R2_1 - R1_1)
R3_2 = R3_1 + 1/3 * (R3_1 - R2_1)
R4_2 = R4_1 + 1/3 * (R4_1 - R3_1)
[R2_2, R3_2, R4_2]
```

$$[1.44083929301393, 1.44044796026391, 1.44042400930725] \quad (11.61)$$

The results represent the result of Simpson's method at step $h = 0.5$, $h = 0.25$ and $h = 0.125$:

```
[simpson_comp(y9v[::4], 4*h9v), simpson_comp(y9v[::2], 2*h9v), simpson_comp(y9v,
h9v)]
```

$$[1.44083929301393, 1.44044796026391, 1.44042400930725] \quad (11.62)$$

We again compute better approximations (we obtain a sixth-order error):

```
## 0(h6)
R3_3 = R3_2 + 1/15 * (R3_2 - R2_2)
R4_3 = R4_2 + 1/15 * (R4_2 - R3_2)

[R3_3, R4_3]
```

$$[1.44042187141391, 1.44042241257681] \quad (11.63)$$

The result represents the better result of Simpson's method at step $h = 0.25$ and $h = 0.125$:

```
a4h, a2h, ah = [simpson_comp(y9v[::4], 4*h9v), simpson_comp(y9v[::2], 2*h9v),
simpson_comp(y9v, h9v)]
[16/15*a2h - 1/15*a4h, 16/15*ah - 1/15*a2h]
```

$$[1.44042187141391, 1.44042241257681] \quad (11.64)$$

We again compute better approximations (we obtain an eighth-order error):

```
## 0(h8)
R4_4 = R4_3 + 1/63 * (R4_3 - R3_3)
R4_4
```

$$1.4404224211667 \quad (11.65)$$

The Romberg method thus offers high result accuracy at a relatively low numerical cost. The error is estimated by:

$$E = |R_{n,n} - R_{n-1,n-1}|. \quad (11.66)$$

Romberg method in `scipy.integrate.romb`

The Romberg method is implemented in `scipy` in `scipy.integrate.romb()` ([documentation](#)):

```
| romb(y, dx=1.0, axis= -1, show=False)
```

where the parameters are:

- `y` the table of function values that we integrate,
- `dx` the width of the equidistant subintervals, i.e., the integration step,
- `axis` the integration axis (important in the case of a multidimensional numeric array),
- `show` if `True`, displays the elements of the Richardson extrapolation.

Let us look at the example from above:

```
| from scipy.integrate import romb
```

```

| y9v
|
| array([0.84147098, 1.01505104, 1.18623077, 1.34872795, 1.49624248,
|        1.62261343, 1.72197541, 1.78891084, 1.81859485])
|
| romb(y9v, dx=h9v, show=True)
|
| Richardson Extrapolation Table for Romberg Integration
|=====
| 1.33003
| 1.41314  1.44084
| 1.43362  1.44045  1.44042
| 1.43872  1.44042  1.44042  1.44042
|=====

```

$$1.4404224211667 \quad (11.67)$$

11.3 Gaussian integration approach

The Newton-Cotes approach is based on a polynomial of degree n , and the error is of degree $n+1$. This means that the integration yields an exact result if the integrated function is a polynomial of degree n or less; for even n , owing to symmetry, even of degree $n+1$ (Simpson's rule, $n=2$, is e.g. exact also for cubic polynomials).

The Gaussian integration approach is fundamentally different: the goal is to replace the integral of the function $f(x)$ with a weighted sum of function values at discrete values $f(x_i)$:

$$\int_a^b f(x) dx \approx \sum_{i=0}^{n-1} A_i f(x_i). \quad (11.68)$$

Here both the weight A_i and the location of the node x_i are unknown. For polynomials of degree n we will also need more points $(x_i, f(x_i))$.

In what follows, we will learn that we can compute a numerically exact integral very efficiently. An additional advantage of Gaussian integration is that it can compute the integral of functions with singularities (e.g.: $\int_0^1 \sin(x)/\sqrt{(x)} dx$).

11.3.1 Gaussian quadrature with one node

Let us assume that we want to integrate a polynomial of degree $n=1$ (a linear function):

$$f(x) = P_1(x) = a_0 + a_1 x. \quad (11.69)$$

Let us compute:

$$\int_{x_L}^{x_D} P_1(x) dx = \left(a_0 x + a_1 \frac{x^2}{2} \right)_{x_L}^{x_D} = -a_0 x_L + a_0 x_D - \frac{a_1 x_L^2}{2} + \frac{a_1 x_D^2}{2}. \quad (11.70)$$

On the other hand, we want to compute the integral from the appropriately weighted A_0 value of the function $f(x_0)$ at an unknown node x_0 (only one node!):

$$\int_{x_L}^{x_D} P_1(x) dx = A_0 P_1(x_0) = A_0 a_0 + A_0 a_1 x_0. \quad (11.71)$$

By equating the expressions above, we derive:

$$-a_0 x_L + a_0 x_D - \frac{a_1 x_L^2}{2} + \frac{a_1 x_D^2}{2} = A_0 a_0 + A_0 a_1 x_0. \quad (11.72)$$

a_0 and a_1 are the coefficients of the linear function, which can take arbitrary values, so it holds that:

$$a_0 (-x_L + x_D - A_0) = 0 \quad \text{and} \quad a_1 \left(-\frac{x_L^2}{2} + \frac{x_D^2}{2} - A_0 x_0 \right) = 0. \quad (11.73)$$

This is a system of linear equations with the solution:

$$A_0 = x_D - x_L, \quad x_0 = \frac{x_L + x_D}{2}. \quad (11.74)$$

If the integrated function is linear, we will compute the true value based only on the value at a single point!

So that the Gaussian integration approach is independent of the integration limits x_L , x_D , we introduce standard limits.

Standard limits: $x_L = -1$ and $x_D = 1$

For generality, we transform the limits $x \in [x_L, x_D]$ into $\xi \in [-1, +1]$ using:

$$x = \frac{x_D + x_L}{2} + \frac{x_D - x_L}{2} \xi \quad (11.75)$$

and

$$dx = \frac{x_D - x_L}{2} d\xi. \quad (11.76)$$

The following holds:

$$\int_{x_L}^{x_D} f(x) dx = \int_{-1}^1 g(\xi) d\xi, \quad (11.77)$$

where:

$$g(\xi) = \frac{x_D - x_L}{2} f \left(\frac{x_L + x_D}{2} + \frac{x_D - x_L}{2} \xi \right). \quad (11.78)$$

In the case of standard limits, at one Gauss point the weight is $A_0 = 2$ and $x_0 = 0$ is the value at which we must evaluate the function $f(x_0)$.

Computational derivation of the weights and the Gauss point

Let us define the symbols and set up the equation:

```
a_0, a_1, x, x_L, x_D, A_0, x_0 = sym.symbols('a_0, a_1, x, x_L, x_D, A_0, x_0') #
symbols
P1 = a_0 + a_1*x # linear
polynomial
eq = sym.Eq(P1.integrate((x, x_L, x_D)).expand(), A_0*P1.subs(x, x_0)) # theoretical
integral = estimate with weights
eq
```

$$a_0 x_D - a_0 x_L + \frac{a_1 x_D^2}{2} - \frac{a_1 x_L^2}{2} = A_0 (a_0 + a_1 x_0) \quad (11.79)$$

Let us prepare two equations (for the first we assume $a_0 = 0, a_1 = 1$, for the second we assume $a_0 = 1, a_1 = 0$) and solve the system for A_0 and x_0 :

```
| sym.solve([eq.subs(a_0, 0).subs(a_1, 1), eq.subs(a_1, 0).subs(a_0, 1)], [A_0, x_0])
```

$$\left[\left(x_D - x_L, \frac{x_D + x_L}{2} \right) \right] \quad (11.80)$$

For additional explanation, I recommend this [video](#).

11.3.2 Gaussian integration method with multiple nodes

We want to derive a Gaussian integration method that will take into account v nodes on the interval $[a, b]$ and will exactly compute the integral of polynomials up to degree $n = 2v - 1$. The following must hold:

$$\int_a^b P_{2v-1}(x) dx = \sum_{i=0}^{v-1} A_i P_{2v-1}(x_i). \quad (11.81)$$

In the derivation we will restrict ourselves to the standard limits $x_L = -1$, $x_D = 1$, where the polynomial of degree $n = 2v - 1$ is defined as:

$$P_{2v-1}(x) = \sum_{i=0}^{2v-1} a_i x^i. \quad (11.82)$$

With two Gauss points/nodes we exactly compute the integral of a polynomial up to third order, and with three Gauss nodes we exactly compute the integral of a polynomial up to fifth order!

Computational derivation

Let us first prepare the symbolic notation of the polynomial and the corresponding variables:

```
def P_etc(n=1, a='a', x='x'):
    polynomial
    ai = sym.symbols(f'{a}:{n+1}')
    x = sym.symbols(x)
    return ai, x, sum([ai[i]*x**i for i in range(n+1)])
```

n is the degree of the
list of a_i
variable x

Now let us find the weights A_i and the nodes x_i for the case of two Gauss nodes; the polynomial is therefore of third degree.

```
v = 2 # number of nodes
ai, x, P = P_etc(n=2*v -1)
xi = sym.symbols(f'x:{v}') # list of x_i
Ai = sym.symbols(f'A:{v}') # list of A_i
print(f'Nodes: {xi}\nWeights: {Ai}')
```

```
Nodes: (x0, x1)
Weights: (A0, A1)
```

The polynomial:

```
P
```

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 \quad (11.83)$$

Similarly to the one node above, here let us define the equations:

```
eqs = [sym.Eq(P.integrate((x, -1, 1)).coeff(a_), \
    sum([Ai[i]*P.subs(x, xi[i]) \
        for i in range(v)]).expand().coeff(a_)) \
    for a_ in ai]
eqs
```

$$\left[2 = A_0 + A_1, 0 = A_0 x_0 + A_1 x_1, \frac{2}{3} = A_0 x_0^2 + A_1 x_1^2, 0 = A_0 x_0^3 + A_1 x_1^3 \right] \quad (11.84)$$

We solve them for the unknown x_i and A_i :

```
sol = sym.solve(eqs, sym.flatten((xi, Ai)))
sol
```

$$\left[\left(-\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, 1, 1 \right), \left(\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, 1, 1 \right) \right] \quad (11.85)$$

We determined a list of two (equal) solutions.

First, the nodes are defined: $x_0 = -\sqrt{3}/3$ and $x_1 = \sqrt{3}/3$, with the corresponding weights $A_0 = A_1 = 1$.

The code above is derived in general - you can increase the number of nodes v and compute the nodes and the corresponding weights.

Below is a table of nodes and weights for one, two, and three nodes (for the limits $a = -1$, $b = 1$):

Number of points 1:

i	Node x_i	Weight A_i
0	0	2

Number of points 2:

i	Node x_i	Weight A_i
0	$-\frac{\sqrt{3}}{3}$	1
1	$+\frac{\sqrt{3}}{3}$	1

Number of points 3:

i	Node x_i	Weight A_i
0	$-\frac{\sqrt{15}}{5}$	$\frac{5}{9}$
1	0	$\frac{8}{9}$
2	$\frac{\sqrt{15}}{5}$	$\frac{5}{9}$

For more nodes and also the error estimate, see [Mathworld Legendre-Gauss Quadrature](#).

For the case of three Gauss points, we compute the numerical integral (standard limits):

$$I_{\text{Gauss3}} = \frac{5}{9} f\left(-\frac{\sqrt{15}}{5}\right) + \frac{8}{9} f(0) + \frac{5}{9} f\left(\frac{\sqrt{15}}{5}\right). \quad (11.86)$$

Numerical implementation

The numerical implementation (including the transformation of the limits) for one, two, or three nodes:

```
def gaussian(fun, a, b, nodes=1):
    """
    Gaussian integration method.

    :param fun: object of the function to integrate
    :param a:   lower limit
    :param b:   upper limit
    :param nodes: number of nodes (1, 2, or 3)
    """
    def g(xi): # function for transforming the limits
        return (b - a)/2 * fun((b+a +xi * (b - a))/2)
```

```

    if nodes == 1:
        return 2*g(0.)
    elif nodes == 2:
        return 1. * g( -np.sqrt(3)/3) + 1. * g(np.sqrt(3)/3)
    elif nodes == 3:
        return 5/9 * g( -np.sqrt(15)/5) + 8/9 * g(0.) + 5/9 * g(np.sqrt(15)/5)

```

Let us look at an example. First, let us define the function we want to integrate:

```

def f(x):
    return x*np.sin(x)

```

Now we pass the function (in this concrete case `f`) and not the value (e.g., `f(0.)`) to the `gaussian` function. First for one node, then two and three:

```

gaussian(fun=f, a=1., b=2., nodes=1)

```

1.49624247990608 (11.87)

```

gaussian(fun=f, a=1., b=2., nodes=2)

```

1.44014401845179 (11.88)

```

gaussian(fun=f, a=1., b=2., nodes=3)

```

1.44042294912151 (11.89)

`scipy.integrate.fixed_quad`

Within `scipy`, the Gaussian integration method is implemented in the `scipy.integrate.fixed_quad()` function ([documentation](#)):

```

fixed_quad(func, a, b, args=(), n=5)

```

where the parameters are:

- `func` is the name of the function we call,
- `a` is the lower limit,
- `b` is the upper limit,
- `args` is a tuple of any additional arguments of the function `func`,
- `n` is the number of nodes of the Gaussian integration, default `n=5`.

The function returns a tuple with the integration result `val` and the value `None`: `(val, None)`

Let us look at the example from above:

```

from scipy.integrate import fixed_quad
fixed_quad(f, a=1, b=2, n=3)[0]

```

1.44042294912151 (11.90)

The result is equal to the previous one:

```

gaussian(fun=f, a=1., b=2., nodes=3)

```

1.44042294912151 (11.91)

11.4 scipy.integrate

`scipy.integrate` is a powerful tool for numerical integration (see the [documentation](#)). In what follows, we will look at some of the functions.

11.4.1 Integration functions that require a defining *function*:

- `quad(func, a, b[, args, full_output, ...])` computes the definite integral of `func(x)` over the limits `[a, b]`,
- `dblquad(func, a, b, gfun, hfun[, args, ...])` computes the definite integral of `func(x,y)`,
- `tplquad(func, a, b, gfun, hfun, qfun, rfun)` computes the definite integral of `func(x,y,z)`,
- `nquad(func, ranges[, args, opts, full_output])` computes the definite integral of the n -dimensional function `func( ...)`,

`scipy.integrate.quad`

Let us look at a very useful function for integration, `scipy.integrate.quad()` ([documentation](#)):

```
quad(func, a, b, args=(), full_output=0, epsabs=1.49e -08, epsrel=1.49e -08,
    limit=50,
    points=None, weight=None, wvar=None, wopts=None, maxp1=50, limlst=50)
```

The selected parameters are:

- `func` the Python function we integrate,
- `a` the lower limit of integration (`np.inf` can be used for an infinite limit),
- `b` the upper limit of integration (`np.inf` can be used for an infinite limit),
- `full_output` to display all results, default 0,
- `epsabs` the allowed absolute error,
- `epsrel` the allowed relative error.

Let us look at the example from above:

```
from scipy import integrate

integrate.quad(f, a=1, b=2)
```

$$(1.44042242098021, 1.59919013695854 \cdot 10^{-14}) \quad (11.92)$$

`scipy.integrate.dblquad`

This is a function similar to `quad`, but for integration over two variables ([documentation](#)). The function `scipy.integrate.dblquad()` computes the double integral of `func(y, x)` over the limits from `x = [a, b]` and `y = [gfun(x), hfun(x)]`.

```
dblquad(func, a, b, gfun, hfun, args=(), epsabs=1.49e -08, epsrel=1.49e -08)
```

The selected parameters are:

- `func` is the Python function we integrate,
- `a` is the lower limit of integration of `x` (`np.inf` can be used for an infinite limit),
- `b` is the upper limit of integration of `x` (`np.inf` can be used for an infinite limit),
- `gfun` is the Python function that defines the lower limit of `y` as a function of `x`,
- `hfun` is the Python function that defines the upper limit of `y` as a function of `x`.

Let us look at an example of computing the area of a half-circle with radius 1:

$$\int_{-1}^1 \left(\int_0^{\sqrt{1-x^2}} 1 \, dy \right) dx = \frac{\pi}{2} \quad (11.93)$$

Let us define the appropriate Python functions and compute the result:

```
def func(y, x): # the integration function is simple, a constant = 1!
    return 1.
def gfun(x):    # lower limit = 0
    return 0.
def hfun(x):    # upper limit
    return np.sqrt(1 - x**2)
integrate.dblquad(func=func, a= -1, b=1, gfun=gfun, hfun=hfun)
```

$$(1.5707963267949, 1.00023545002159 \cdot 10^{-9}) \quad (11.94)$$

11.4.2 Integration functions that require a table of values:

- `trapezoid(y[, x, dx, axis])` composite trapezoidal rule,
- `cumulative_trapezoid(y[, x, dx, axis, initial])` cumulative integral of the integrand (returns the result at each node),
- `simpson(y[, x, dx, axis])` Simpson's method,
- `romb(y[, dx, axis, show])` Romberg method.

Here we will look at the `cumulative_trapezoid` function on an example. Consider the case where a mass $m = 1$ (we omit the units) acted upon by a force $F = \sin(t)$. From Newton's 2nd law it follows that: $F = m\ddot{x}$. We integrate the acceleration to compute the velocity; then we integrate once more for the displacement. Let us first define the function for the acceleration:

```
def acceleration(t):
    F = np.sin(t)
    m = 1
    return F/m
```

We are interested in what happens over a time of 3 seconds:

```
t, h = np.linspace(0, 3, 100, retstep=True)
```

Let us compute the table of accelerations and then integrate for the velocity (here it is important to define the initial value):

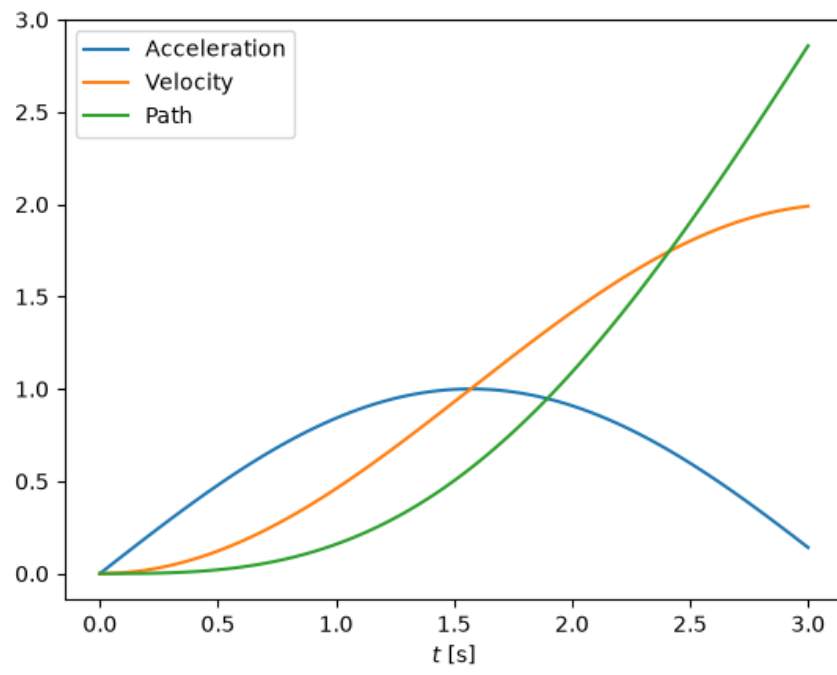
```
a = acceleration(t)
v = integrate.cumulative_trapezoid(y=a, dx=h, initial=0)
```

Now let us integrate the velocity once more to compute the path:

```
s = integrate.cumulative_trapezoid(y=v, dx=h, initial=0)
```

Let us display the result:

```
plt.plot(t, a, label='Acceleration')
plt.plot(t, v, label='Velocity')
plt.plot(t, s, label='Path')
plt.xlabel('$t$ [s]')
plt.legend();
```



Chapter 12

Numerical solution of differential equations - initial value problem

```
import numpy as np
from ipywidgets import interact
import matplotlib.pyplot as plt
%matplotlib inline
import sympy as sym
sym.init_printing()
```

12.1 Introduction

12.1.1 Writing (a single) differential equation

Assume that a first-order differential equation can be written in *explicit* form:

$$y' = f(t, y), \quad (12.1)$$

where $f(t, y)$ is a given function and $y' = dy/dt$.

Additionally, an initial condition is given:

$$y(t_0) = y_0. \quad (12.2)$$

The goal of solving the differential equation is to compute the function $y(t)$ that solves the initial value problem defined above. Under certain conditions on the function $f(t, y)$, the initial value problem has a unique solution on an interval containing t_0 .

In numerical solving we always compute a table of the function $y(t_i)$ that solves the given initial value problem. The nodes t_i are usually equidistant:

$$t_0, t_0 + h, t_0 + 2h, \dots \quad (12.3)$$

and we call h the (time) step (of integration).

Here we will look at some numerical methods for solving differential equations with an initial condition.

12.2 Euler's method

The **explicit Euler's method** is based on the Taylor series expansion of the function y :

$$y(t + h) = y(t) + f(t, y(t)) h + \mathcal{O}(h^2). \quad (12.4)$$

We make a method error of $\mathcal{O}(h^2)$ because we neglect the second and higher-order derivatives; now, given a known value $y(t)$ and the derivative $y'(t) = f(t, y)$, we can estimate the value at the next time step $t + h$. Once we know the values at $t + h$, we repeat the procedure!

Steps of Euler's method:

1. Set $i = 0$, t_0 , $y_0 = y(t_0)$.
2. Compute the function value at $t_{i+1} = t_i + h$: $y_{i+1} = y_i + f(t_i, y_i) h$.
3. $i = i + 1$ and continue at step 2.

We solve the differential equation on the interval $[t_0, t_n]$ and $h = (t_n - t_0)/n$. n is the number of integration steps (how many times we perform step 2 in the algorithm above).

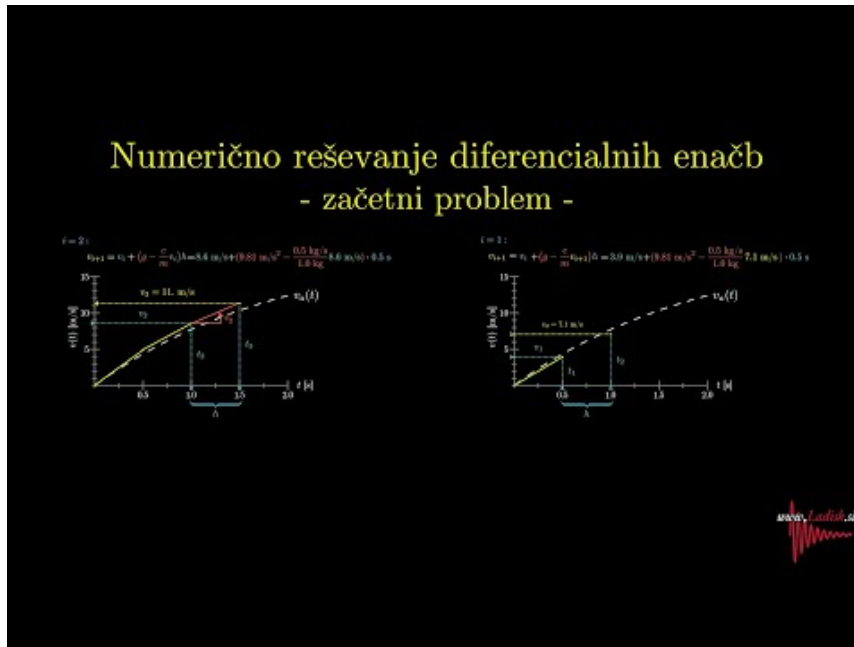
Numerical solution of the initial value problem:

$$y_0, y_1, y_2, \dots, y_n \quad (12.5)$$

at the values of the independent variable:

$$t_0, t_1, t_2 \dots t_n. \quad (12.6)$$

```
from IPython.display import YouTubeVideo
YouTubeVideo('cfZ8v0b -R8o', width=800, height=300)
```



12.2.1 Error of Euler's method

The error of Euler's method at each step is of order $\mathcal{O}(h^2)$.

Since over the interval from t_0 to t_n we make this error n times, the cumulative error is $n \mathcal{O}(h^2) = \frac{t_n - t_0}{h} \mathcal{O}(h^2) = \mathcal{O}(h)$.

Locally the error is second order, but globally the error is first order, and because Euler's method is so inaccurate, we rarely use it in practice!

Error estimate

The exact solution $y(t_n)$ at step size h is:

$$y(t_n) = y_{n,h} + E_h, \quad (12.7)$$

where $y_{n,h}$ is the numerical approximation and E_h is the method error. Since the global error is first order, we can write the error as:

$$E_h = k h. \quad (12.8)$$

Similarly, for step size $2h$ we can write:

$$y(t_n) = y_{n,2h} + E_{2h}, \quad (12.9)$$

where $y_{n,2h}$ is the numerical approximation and E_{2h} is the method error:

$$E_{2h} = k 2 h. \quad (12.10)$$

Assuming that the constant k is the same for step h and step $2h$, we can determine the error estimate for the better approximation E_h . Clearly:

$$y_{n,h} + k h = y_{n,2h} + 2 k h \quad (12.11)$$

then we determine the *error estimate*:

$$E_h = k h = y_{n,h} - y_{n,2h}. \quad (12.12)$$

12.2.2 A note on the implicit Euler's method

In the explicit Euler's method we compute the solution at t_{i+1} from the computed value at t_i .

If the unknown value of the solution at t_{i+1} , that is y_{i+1} , appeared on the right-hand side as well, we would be talking about the **implicit Euler's method** (or *backward Euler's method*):

$$y_{i+1} = y_i + f(t_{i+1}, y_{i+1}) h. \quad (12.13)$$

Because the sought value y_{i+1} appears on both sides of the equation, to determine y_{i+1} we must solve a (nonlinear) equation. The advantage of the implicit Euler's method is that it is more stable (e.g., for stiff systems, which we will learn about later) than the explicit form, but it is numerically more demanding (because of solving the equation).

12.2.3 Numerical implementation

First we import the required libraries:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Next let us define Euler's method:

```
def euler(f, t, y0, *args, **kwargs):
    """
    Euler's method for solving a system of differential equations: y' = f(t, y)

    :param f: function returning the first derivative - f(t, y)
    :param t: time vector where we compute the solution
    :param y0: initial value
    :param args: additional arguments of the function f (positional)
    :param kwargs: additional arguments of the function f (keyword)
    :return y: returns an np.array ``y`` of function values.
    """
    y = np.zeros_like(t)
    y[0] = y0
    h = t[1] - t[0]
    for i in range(len(t) - 1):
        y[i+1] = y[i] + f(t[i], y[i], *args, **kwargs) * h
    return y
```

Let us prepare a function for the error estimate (in numerical terms it would be better to include the error estimate inside the `euler` function, but for clarity we present it separately):

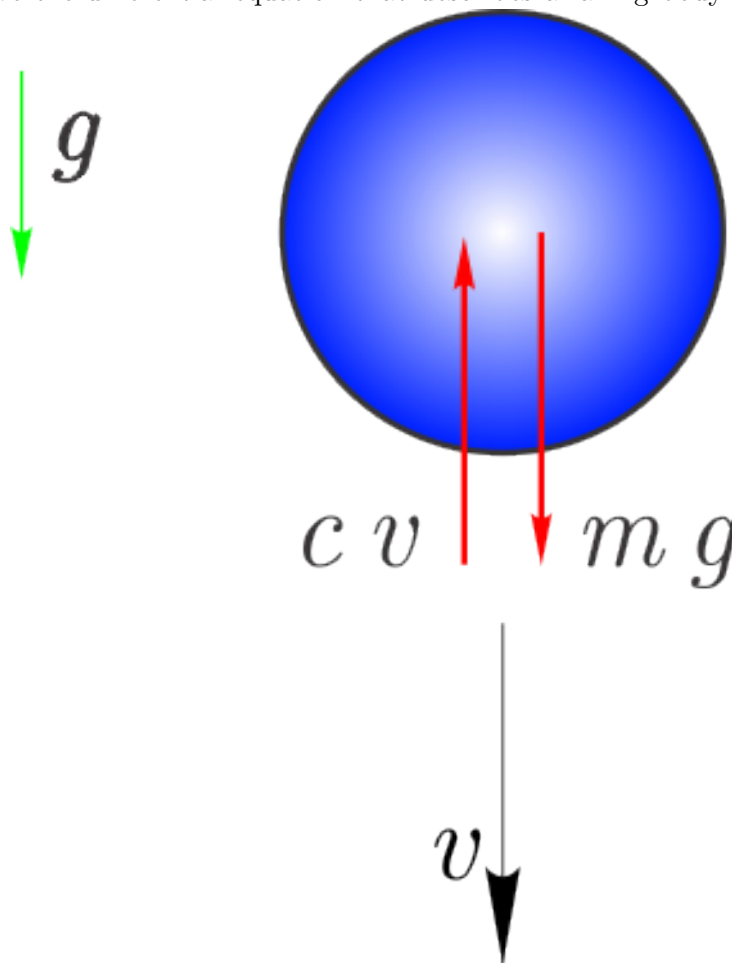
```

def euler_error(f, t, y0, *args, **kwargs):
    """ Error estimate of Euler's method; the arguments are the same as for the
    `euler` function
    """
    n = len(t)
    if n < 5:
        raise Exception('There must be at least 5 nodes.')
    if n%2==0: # even number of nodes; remove one point and make it odd (so the
    number of subintervals is even)
        n = n - 1
    y_h = euler(f, t[:n], y0, *args, **kwargs)
    y_2h = euler(f, t[:n:2], y0, *args, **kwargs)
    E_h = y_h[-1] - y_2h[-1]
    return E_h

```

12.2.4 Numerical example

As an example, let us solve the differential equation that describes a falling body subjected to



gravity and air resistance:

According to Newton's second law, we can write the differential equation:

$$m g - c v = m v', \quad (12.14)$$

where m is the mass, g the gravitational acceleration, c the air-resistance coefficient, and v the velocity. We would like to solve the differential equation subject to the initial condition:

$$v(0) = 0 \text{ m/s}. \quad (12.15)$$

The right-hand side / first-derivative function $f(t, y)$ is:

$$f(t, y) = g - c \frac{y}{m} \quad (12.16)$$

and the initial condition:

$$y_0 = 0. \quad (12.17)$$

Let us define the right-hand-side function:

```
def f_air_resistance(t, y, g=9.81, m=1., c=0.5):
    return g - c*y/m
```

Let us define the initial condition and the time vector where we want the result:

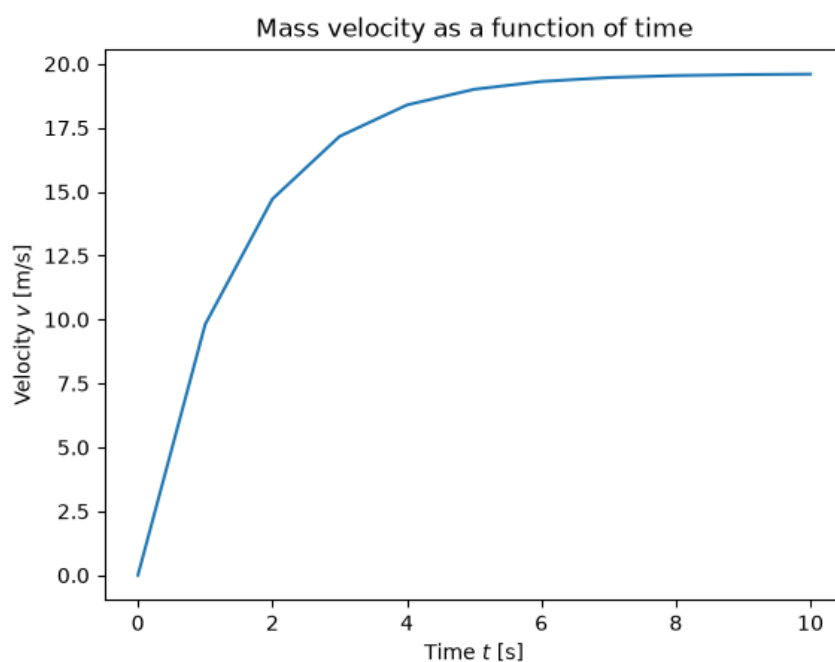
```
v0 = 0
t = np.linspace(0, 10, 11)
t
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.] )
```

We call the `euler` function to compute the values of y (velocity v):

```
y = euler(f_air_resistance, t, y0=v0)
y
array([ 0.,  9.81, 14.715, 17.1675, 18.39375,
        19.006875, 19.3134375, 19.46671875, 19.54335938, 19.58167969,
        19.60083984])
```

We plot the result:

```
plt.plot(t, y)
plt.title('Mass velocity as a function of time')
plt.xlabel('Time $t$ [s]')
plt.ylabel('Velocity $v$ [m/s]')
plt.show()
```

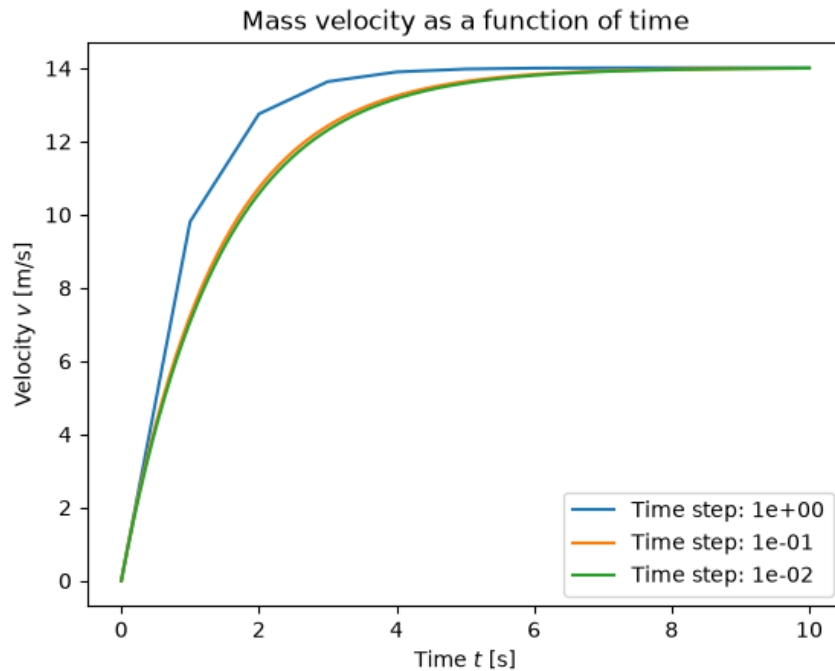


Let us now check the effect of the time step:

```

for n in [11, 101, 1001]:
    t = np.linspace(0, 10, n)
    y = euler(f_air_resistance, t, y0=v0, c=0.7)
    plt.plot(t, y, label=f'Time step: {t[1]:1.0e}')
plt.title('Mass velocity as a function of time')
plt.xlabel('Time $t$ [s]')
plt.ylabel('Velocity $v$ [m/s]')
plt.legend()
plt.show()

```



We observe that the numerical error decreases substantially when the step changes from 1 to 0.1!

Let us also estimate the error at 100 and 1000 subintervals:

```

n=101
t = np.linspace(0, 10, n)
euler_error(f_air_resistance, t, y0=v0)

```

$$-0.015043713735345 \quad (12.18)$$

```

n=1001
t = np.linspace(0, 10, n)
euler_error(f_air_resistance, t, y0=v0)

```

$$-0.00163798622449463 \quad (12.19)$$

When we reduced the step to a tenth, the error decreased proportionally as well (first-order error).

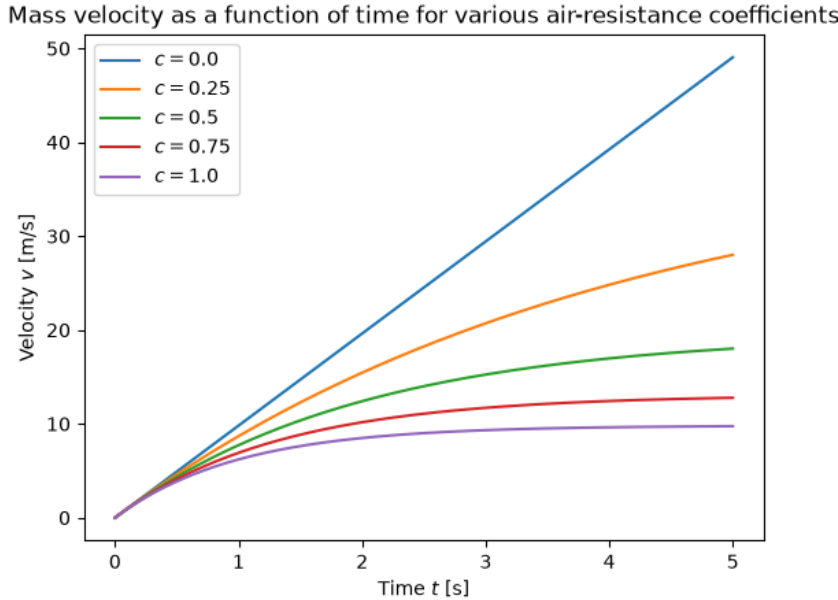
Let us also look at the case where the air resistance `c` is an argument of the `euler` function and is passed via `**kwargs` to the `f_air_resistance()` function:

```

for c in np.linspace(0, 1, 5):
    t = np.linspace(0, 5, 1001)
    y = euler(f_air_resistance, t, y0=v0, c=c)
    plt.plot(t, y, label=f'$c={c}$')

```

```
plt.title('Mass velocity as a function of time for various air -resistance
coefficients')
plt.xlabel('Time $t$ [s]')
plt.ylabel('Velocity $v$ [m/s]')
plt.legend()
plt.show()
```



12.3 Second-order Runge–Kutta method

Euler’s method is first order (the global error $\mathcal{O}(h)$ is indeed first order). If we wanted to derive a second-order-error method, we would need to use the Taylor series expansion of $y(t+h)$, where we neglect the third and higher derivatives:

$$y(t+h) = y(t) + y'(t)h + \frac{1}{2}y''(t)h^2 + \mathcal{O}(h^3). \quad (12.20)$$

The local error of the method will thus be third order and the global error second order.

We use the substitutions $y'(t) = f(t, y)$ and $y''(t) = f'(t, y)$:

$$y(t+h) = y(t) + f(t, y)h + \frac{1}{2}f'(t, y)h^2 + \mathcal{O}(h^3). \quad (12.21)$$

Because the right-hand side $f(t, y)$ depends on both the independent variable t and the dependent variable y , we must use implicit differentiation:

$$f'(t, y) = \frac{\partial f(t, y)}{\partial t} + \underbrace{\frac{\partial f(t, y)}{\partial y} \frac{dy}{dt}}_{y' = f(t, y)} = \frac{\partial f(t, y)}{\partial t} + \frac{\partial f(t, y)}{\partial y} f(t, y). \quad (12.22)$$

We substitute into the Taylor series expression:

$$y(t+h)_{\text{Taylor}} = y(t+h) = y(t) + f(t, y)h + \frac{1}{2} \left(\frac{\partial f(t, y)}{\partial t} + \frac{\partial f(t, y)}{\partial y} f(t, y) \right) h^2. \quad (12.23)$$

As can be seen from the expression above, we need additional derivatives. This poses a certain difficulty that can be avoided in various ways; in what follows we will look at the *Runge–Kutta* approach. Because we will need the expression above later, we have named it here $y(t+h)_{\text{Taylor}}$.

12.3.1 Idea of the Runge–Kutta approach

The dilemma above is resolved by the *Runge–Kutta* method (developed in 1901) with an idea we have already encountered in Gaussian integration: it tries to find a more accurate solution with a weighted additional value of the function f :

$$y(t+h)_{\text{Runge-Kutta}} = y(t) + c_0 f(t, y) h + c_1 \underbrace{f(t+ph, y+qh f(t, y))}_A h. \quad (12.24)$$

where c_0 , c_1 , p , and q are unknown constants (in principle from 0 up to and including 1). If we used $c_1 = 0$ in the expression above, we would derive a first-order method; but with an additional function value (A) it will turn out that we derive a second-order method.

We continue the search for the unknown constants c_0 , c_1 , p , q by writing A in the form of a first-order Taylor series:

$$f(t+ph, y+qh f(t, y)) = \underbrace{f(t, y) + \frac{\partial f(t, y)}{\partial t} (ph) + \frac{\partial f(t, y)}{\partial y} (qh f(t, y))}_B. \quad (12.25)$$

Now we substitute the derived B back into the expression for $y(t+h)_{\text{Runge-Kutta}}$:

$$y(t+h)_{\text{Runge-Kutta}} = y(t) + c_0 f(t, y) h + c_1 \left(f(t, y) + \frac{\partial f(t, y)}{\partial t} (ph) + \frac{\partial f(t, y)}{\partial y} (qh f(t, y)) \right) h. \quad (12.26)$$

We continue the derivation and rearrange the equation so that it resembles the one derived above with the Taylor series $y(t+h)_{\text{Taylor}}$:

$$y(t+h)_{\text{Runge-Kutta}} = y(t) + (c_0 + c_1) f(t, y) h + \frac{1}{2} \left(\frac{\partial f(t, y)}{\partial t} 2c_1 p + 2c_1 q \frac{\partial f(t, y)}{\partial y} f(t, y) \right) h^2. \quad (12.27)$$

Now let us compare with the expression derived above:

$$y(t+h)_{\text{Taylor}} = y(t) + f(t, y) h + \frac{1}{2} \left(\frac{\partial f(t, y)}{\partial t} + \frac{\partial f(t, y)}{\partial y} f(t, y) \right) h^2. \quad (12.28)$$

We find that for equality to hold we must have:

$$c_0 + c_1 = 1, \quad 2c_1 p = 1, \quad 2c_1 q = 1. \quad (12.29)$$

So we have three equations and four unknowns. We can therefore choose one of the constants arbitrarily and compute the other three. If, for example, we choose $c_0 = 0$, we would call this the *modified Euler's method* and the remaining parameters would be: $c_1 = 1$, $p = q = 1/2$. The choice of parameters has no essential effect on the solution. It is worth noting, however, that even the second-order Runge–Kutta method is rarely used, since better methods exist.

We substitute the parameters c_0 , c_1 , p , and q into the first equation of this section. Once the initial time t_0 and the initial condition y_0 are defined, we use the second-order Runge–Kutta method:

$$y_{i+1} = y_i + f \left(t_i + \frac{1}{2} h, y_i + \frac{1}{2} h f(t_i, y_i) \right) h. \quad (12.30)$$

12.4 Fourth-order Runge–Kutta method

Similarly to how we derived the second-order Runge–Kutta method, the fourth-order Runge–Kutta method is derived. For the fourth-order method, too, there are several variants, and by the fourth-order Runge–Kutta method we mean the following method:

$$y_{i+1} = y_i + \frac{1}{6}(k_0 + 2k_1 + 2k_2 + k_3), \quad (12.31)$$

where:

$$\begin{aligned} k_0 &= h f(t_i, y_i) \\ k_1 &= h f\left(t_i + \frac{h}{2}, y_i + \frac{k_0}{2}\right) \\ k_2 &= h f\left(t_i + \frac{h}{2}, y_i + \frac{k_1}{2}\right) \\ k_3 &= h f(t_i + h, y_i + k_2). \end{aligned} \quad (12.32)$$

The steps of the fourth-order Runge–Kutta method are:

1. Set $i = 0$ and $t_0, y_0 = y(t_0)$,
2. Compute the coefficients: k_0, k_1, k_2, k_3 ,
3. Compute the value of the solution of the differential equation at $t_{i+1} = t_i + h$: $y_{i+1} = y_i + \frac{1}{6}(k_0 + 2k_1 + 2k_2 + k_3)$,
4. $i = i + 1$ and continue at step 2.

12.4.1 Error of the fourth-order Runge–Kutta method

The fourth-order Runge–Kutta method is named so because it has a local error of fifth order $\mathcal{O}(h^5)$, but we make this error n times, so the global error is fourth order $\mathcal{O}(h^4)$.

Error estimate

The exact result $y(t_n)$ at step size h is:

$$y(t_n) = y_{n,h} + E_h, \quad (12.33)$$

where $y_{n,h}$ is the numerical approximation of the solution and E_h is the method error. Since the global error is fourth order, we can write the error as:

$$E_h = k h^4. \quad (12.34)$$

Similarly, for step size $2h$ we can write:

$$y(t_n) = y_{n,2h} + E_{2h}, \quad (12.35)$$

where $y_{n,2h}$ is the numerical approximation of the solution and E_{2h} is the method error:

$$E_{2h} = k (2h)^4 = 16 k h^4. \quad (12.36)$$

Assuming that the constant k is the same for step h and step $2h$, we can compute the error estimate for the better approximation E_h .

First, it is true that:

$$y_{n,h} + k h^4 = y_{n,2h} + 16 k h^4, \quad (12.37)$$

it follows:

$$15 k h^4 = y_{n,h} - y_{n,2h} \quad (12.38)$$

and then we determine the *error estimate* of the more accurate solution:

$$E_h = \frac{y_{n,h} - y_{n,2h}}{15}. \quad (12.39)$$

12.4.2 Numerical implementation

```
def runge_kutta_4(f, t, y0, *args, **kwargs):
    """
    Fourth -order Runge-Kutta method for solving a differential equation: y' = f(t,
    y)

    :param f: function called with the parameters t and y, returning
               the value of the first derivative
    :param t: equidistant time vector, i.e., the independent variable
    :param y0: initial value
    :param args: additional arguments of the function f (positional)
    :param kwargs: additional arguments of the function f (keyword)
    :return y: function values.
    """
    def RK4(f, t, y, *args, **kwargs):
        k0 = h*f(t, y, *args, **kwargs)
        k1 = h*f(t + h/2.0, y + k0/2.0, *args, **kwargs)
        k2 = h*f(t + h/2.0, y + k1/2.0, *args, **kwargs)
        k3 = h*f(t + h, y + k2, *args, **kwargs)
        return (k0 + 2.0*k1 + 2.0*k2 + k3)/6.0

    y = np.zeros_like(t)
    y[0] = y0
    h = t[1] - t[0]

    for i, ti in enumerate(t[: -1]):
        y[i+1] = y[i] + RK4(f, ti, y[i], *args, **kwargs)

    return y
```

Function for the error estimate:

```
def runge_kutta_4_error(f, t, y0, *args, **kwargs):
    """ Error estimate of the fourth -order Runge-Kutta method; arguments are the
    same as for `runge_kutta_4`
    """
    n = len(t)
    if n < 5:
        raise Exception('There must be at least 5 nodes.')
    if n%2==0: # even number of nodes; remove one point and make it odd (so the
    number of subintervals is even)
        n = n - 1
    y_h = runge_kutta_4(f, t[:n], y0, *args, **kwargs)
    y_2h = runge_kutta_4(f, t[:n:2], y0, *args, **kwargs)
    E_h = (y_h[ -1] - y_2h[ -1])/15
    return E_h
```

12.4.3 Numerical example

Let us now look at the example of computing the velocity of a falling mass:

```
def f_air_resistance(t, y, g=9.81, m=1., c=0.5):
    return g - c*y/m
```

Let us provide the initial condition and the time vector where we want the result:

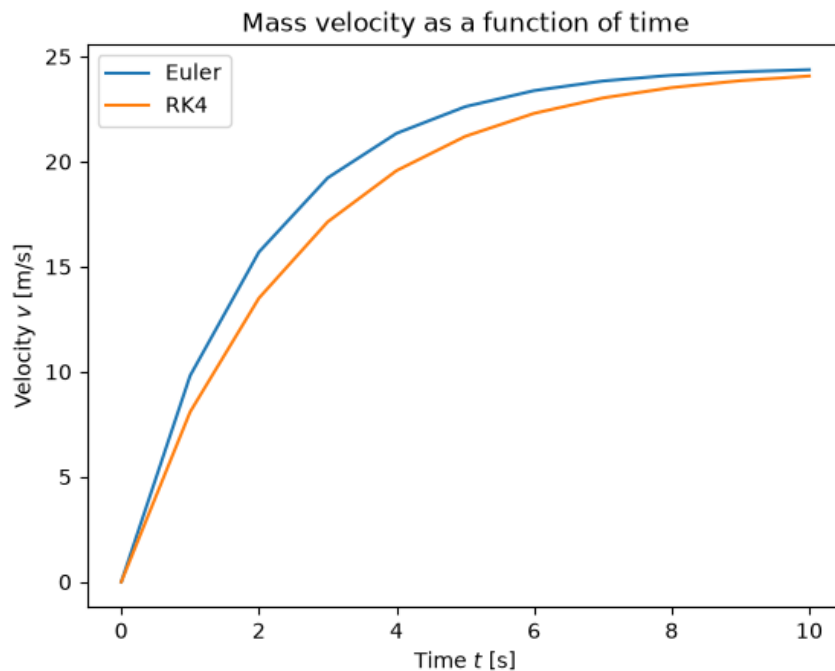
```
v0 = 0
t = np.linspace(0, 10, 11)
t
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.] )
```

For comparison, let us compute the solution with the `euler` and `runge_kutta_4` functions:

```
y_euler = euler(f_air_resistance, t, y0=v0, c=0.4)
y_rk4 = runge_kutta_4(f_air_resistance, t, y0=v0, c=0.4)
```

We plot the result:

```
plt.plot(t, y_euler, label='Euler')
plt.plot(t, y_rk4, label='RK4')
plt.title('Mass velocity as a function of time')
plt.xlabel('Time $t$ [s]')
plt.ylabel('Velocity $v$ [m/s]')
plt.legend()
plt.show()
```



Let us also look at the numerical error:

```
n=101
t = np.linspace(0, 10, n)
runge_kutta_4_error(f_air_resistance, t, y0=v0, c=0.4)
```

$$4.10657750175384 \cdot 10^{-8} \quad (12.40)$$

```
n=1001
t = np.linspace(0, 10, n)
runge_kutta_4_error(f_air_resistance, t, y0=v0, c=0.4)
```

$$3.85990498822745 \cdot 10^{-12} \quad (12.41)$$

When the step is reduced to a tenth, the error decreased by about 10^4 times (which matches the expectation for a fourth-order method).

12.5 Using scipy to solve ordinary differential equations

The `scipy` package has [many numerical methods](#) implemented for solving initial value problems of ordinary differential equations. Here we will look at the function `scipy.integrate.solve_ivp`, which is suitable for most initial value problems.

12.5.1 `scipy.integrate.solve_ivp`

Syntax for use (IVP - Initial Value Problem):

```
scipy.integrate.solve_ivp(fun, t_span, y0, method='RK45',
                          t_eval=None, dense_output=False,
                          events=None, vectorized=False,
                          args=None, **options)
```

An explanation of all the arguments is in the [documentation](#); here we will highlight a few:

- `fun` is the right-hand side (`func(t, y...)`),
- `t_span` a tuple (`t0, tf`) defining the initial time `t0` and final time `tf`,
- `y0` a **list** of initial value(s),
- `method` defines the numerical method (the default is the explicit 'RK45'; for stiff systems the following are better: 'Radau', 'BDF', 'LSODA'),
- `t_eval` is used if we want the solution at specific values of the independent variable,
- `dense_output` whether a continuous solution is also prepared, default `False`,
- `events` for tracking events that stop solving the initial value problem (e.g., when the mass touches the ground and the relative distance to the ground is zero).
- `args` for passing arguments into the function `fun` (see the example below).

The result of calling `solve_ivp` is an object with attributes (selected):

- `t` the values of the independent variable at which the result is computed,
- `y` the result,
- `sol` the continuous-solution function (only in the case of `dense_output=True`)
- `success` is `True` if the solution was computed successfully or terminated due to an event (`events`).

Numerical example

Let us define the initial condition and the time vector where we want the result (we show only the first ten elements):

```
v0 = 0
t = np.linspace(0, 10, 101)
t[:10]

array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

We pass the arguments (`g=9.81`, `m=1.`, `c=0.4`) into the function via the `args` parameter:

```
from scipy.integrate import solve_ivp
y_ivp = solve_ivp(f_air_resistance, t_span=(t[0], t[-1]), y0=[v0], args=(9.81, 1,
0.4))
```

We could also use a `lambda` expression ([documentation](#)):

```
y_ivp = solve_ivp(lambda t,y: f_air_resistance(t, y, c=0.4), t_span=(t[0], t[-1]),
y0=[v0])
```

The `lambda` expression used above, `lambda t, y: f_air_resistance(t, y, c=0.4)`, is equivalent to:

```
def function_name(t, y):
    return f_air_resistance(t, y, c=0.4)
```

Let us look at the result:

```
y_ivp
```

```
message: The solver successfully reached the end of the integration interval.
success: True
status: 0
  t: [ 0.000e+00  1.000e -04  1.100e -03  1.110e -02  1.111e -01
      1.111e+00  2.982e+00  5.236e+00  7.948e+00  1.000e+01]
  y: [[ 0.000e+00  9.810e -04  1.079e -02  1.086e -01  1.066e+00
      8.800e+00  1.708e+01  2.150e+01  2.350e+01  2.407e+01]]
sol: None
t_events: None
y_events: None
nfev: 56
njev: 0
nlu: 0
```

Because `solve_ivp` is already set up for a system of first-order ordinary differential equations, the result is given in two dimensions. The sought result is therefore: `y_ivp.y[0]`.

For comparison, here is also the result with our own fourth-order Runge–Kutta implementation:

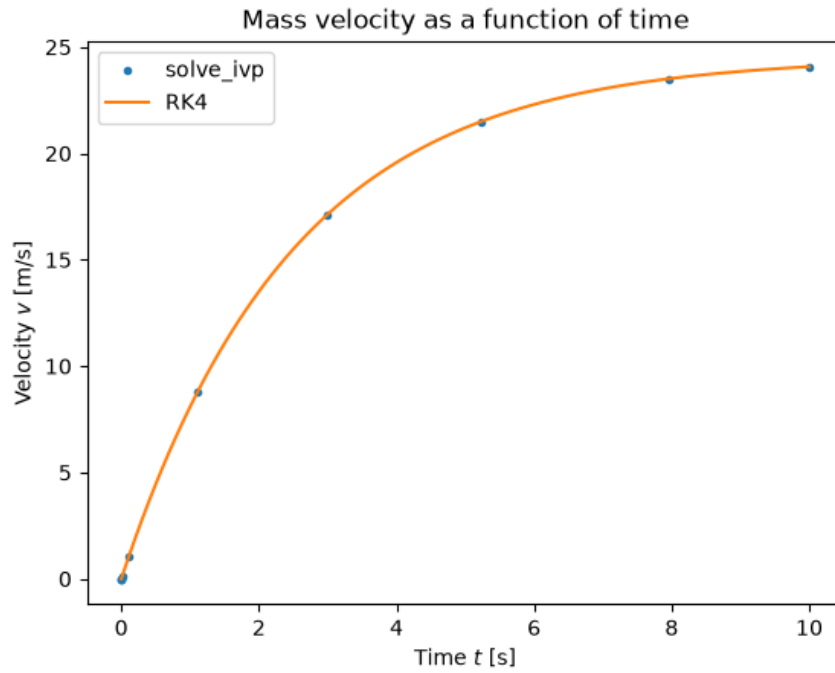
```
y_rk4 = runge_kutta_4(f_air_resistance, t, y0=v0, c=0.4)
```

```
y_rk4[:10]
```

```
array([0.          , 0.96163898, 1.88557156, 2.77327623, 3.6261735 ,
      4.44562819, 5.23295161, 5.98940363, 6.71619474, 7.41448797])
```

We plot the result:

```
plt.plot(y_ivp.t, y_ivp.y[0], '.', label='solve_ivp')
plt.plot(t, y_rk4, label='RK4')
plt.title('Mass velocity as a function of time')
plt.xlabel('Time $t$ [s]')
plt.ylabel('Velocity $v$ [m/s]')
plt.legend()
plt.show()
```



12.6 System of ordinary differential equations

Above we looked at solving the initial value problem of a single ordinary differential equation; now we will generalize the solving to an *initial value problem for a system of m first-order ordinary differential equations*.

Such a system is written as:

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}), \quad (12.42)$$

where additional (initial) conditions are given:

$$\mathbf{y}(t_0) = \mathbf{y}_0. \quad (12.43)$$

By t we denote the independent variable (it need not always be time) and by $\mathbf{f}$ the vector of right-hand sides.

We compute the solution of a system of m differential equations $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ at $a = t_0, t_1, \dots, t_{n-1} = b$, that is m function values at each step $y_k(t_i)$, $k = 0, 1, \dots, m-1$ and $i = 0, 1, \dots, n-1$. It can be proved that any method we derived for first-order differential equations can also be used for a system of first-order ordinary differential equations if we replace the scalar quantities with the corresponding vector ones.

12.6.1 Numerical implementation

The Euler and fourth-order Runge–Kutta methods need only minor adjustments (y_0 is a numerical array, $\mathbf{f}$ returns a list of derivative values):

```
def euler_system(f, t, y0, *args, **kwargs):
    """
    Euler's method for solving a system of first -order ordinary differential
    equations:  $y' = f(t, y)$ 

    :param f: function called with the parameters t and y, returning a list
              of right -hand -side functions
    :param t: equidistant (time) vector of the independent variable
    :param y0: list of initial values
```

```

:param args: additional arguments of the function f (positional)
:param kwargs: additional arguments of the function f (keyword)
:return y: returns an np.array ``y`` of function values.
"""
y = np.zeros((t.shape[0], len(y0)))
y[0] = np.copy(y0)
h = t[1] - t[0]
for i, ti in enumerate(t[: -1]):
    # here is the essence of Euler's method
    y[i+1] = y[i] + f(ti, y[i], *args, **kwargs)*h
return y

def runge_kutta_4_system(f, t, y0, *args, **kwargs):
    """
    Fourth -order Runge-Kutta method for solving a system of first -order ordinary
    differential equations: y' = f(t, y)

    :param f: function called with the parameters t and y, returning a list
              of right -hand -side functions
    :param t: equidistant (time) vector of the independent variable
    :param y0: list of initial values
    :param args: additional arguments of the function f (positional)
    :param kwargs: additional arguments of the function f (keyword)
    :return y: returns an np.array ``y`` of function values.
    """
    def RK4(f, t, y, *args, **kwargs):
        k0 = h*f(t, y, *args, **kwargs)
        k1 = h*f(t + h/2.0, y + k0/2.0, *args, **kwargs)
        k2 = h*f(t + h/2.0, y + k1/2.0, *args, **kwargs)
        k3 = h*f(t + h, y + k2, *args, **kwargs)
        return (k0 + 2.0*k1 + 2.0*k2 + k3)/6.0

    y = np.zeros((t.shape[0], len(y0)))
    y[0] = np.copy(y0)
    h = t[1] - t[0]

    for i, ti in enumerate(t[: -1]):
        y[i+1] = y[i] + RK4(f, ti, y[i], *args, **kwargs)

    return y

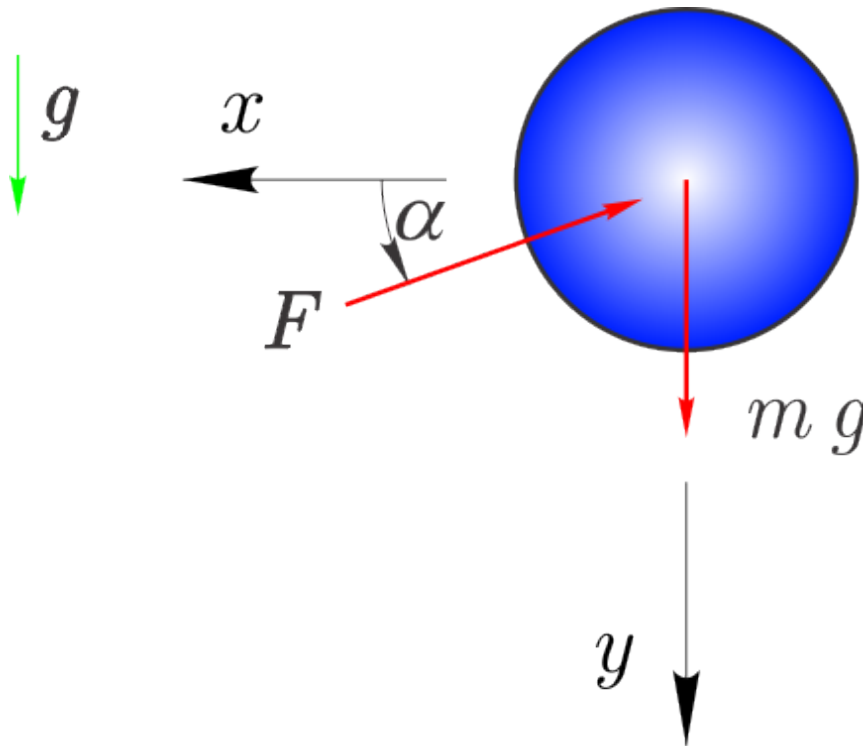
```

Numerical example

We extend the falling mass to planar motion; let the magnitude of the air-resistance force be defined as:

$$|\mathbf{F}(\mathbf{v})| = c|\mathbf{v}|^2. \quad (12.44)$$

The air-resistance force acts in the opposite direction to the velocity vector.



With the help of the figure, we define the force in the x direction:

$$F_x = -c (v_x^2 + v_y^2) \cos(\alpha) = -c (v_x^2 + v_y^2) \frac{v_x}{\sqrt{v_x^2 + v_y^2}} = -c v_x \sqrt{v_x^2 + v_y^2}. \quad (12.45)$$

Similarly, the force in the y direction is:

$$F_y = -c v_y \sqrt{v_x^2 + v_y^2}. \quad (12.46)$$

According to Newton's second law, we write a system of two (coupled) first-order differential equations:

$$F_x = m v'_x, \quad (12.47)$$

$$m g + F_y = m v'_y, \quad (12.48)$$

m is the mass, g the gravitational acceleration, c the air-resistance coefficient, and v_x and v_y the velocities in the x and y directions, respectively. We would like to solve the differential equation subject to the initial condition:

$$v_x(0) = v_y(0) = 5 \text{ m/s}. \quad (12.49)$$

Let us define the list of right-hand sides:

```
def f_air_resistance_force(t, y, g=9.81, m=1., c=0.5):
    vx, vy = y
    return np.array([ -c*vx*np.sqrt(vx**2+vy**2)/m, g -c*vy*np.sqrt(vx**2+vy**2)/m])
```

Let us define the initial condition and the time vector where we want the result:

```
y0 = np.array([5., 5.])
t = np.linspace(0, 5, 101)
t[:5]

array([0. , 0.05, 0.1 , 0.15, 0.2 ])
```

For comparison, let us compute the solution with the `runge_kutta_4` and `solve_ivp` functions:

```
y_RK4 = runge_kutta_4_system(f_air_resistance_force, t=t, y0=y0)
y_ivp = solve_ivp(f_air_resistance_force, t_span=(t[0], t[-1]), y0=y0, t_eval=t)
```

Let us look at the result:

```
y_ivp.y[:, :5]

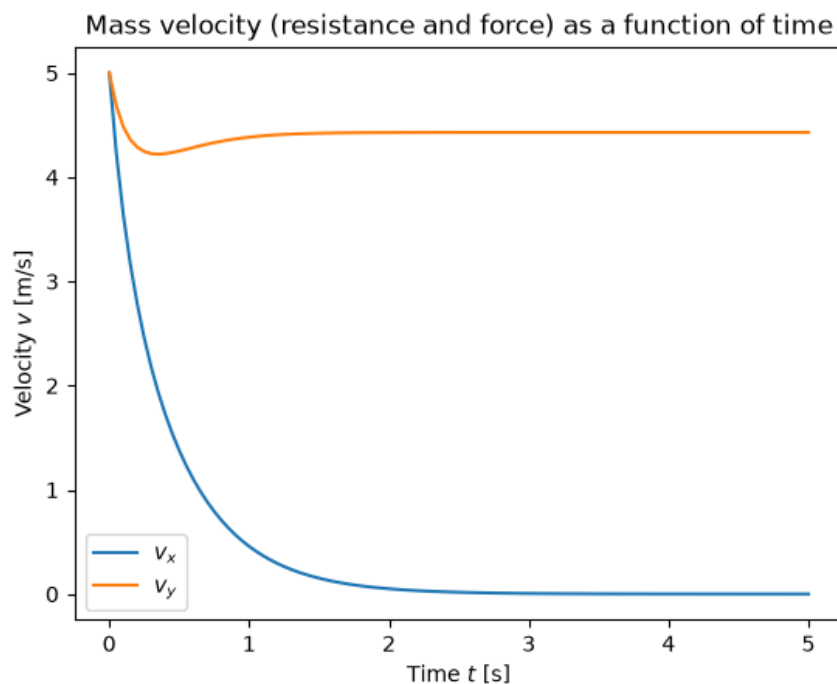
array([[5.          , 4.23209907, 3.63858619, 3.16619484, 2.78087608],
       [5.          , 4.68459713, 4.48359747, 4.35932897, 4.28742151]])

y_RK4[:5].T

array([[5.          , 4.23211316, 3.6399137 , 3.16660748, 2.77780226],
       [5.          , 4.68462761, 4.48494405, 4.35999931, 4.28466858]])
```

We plot the result:

```
plt.plot(t, y_ivp.y[0], label='$v_x$')
plt.plot(t, y_ivp.y[1], label='$v_y$')
plt.title('Mass velocity (resistance and force) as a function of time')
plt.xlabel('Time $t$ [s]')
plt.ylabel('Velocity $v$ [m/s]')
plt.legend()
plt.show()
```



12.6.2 Reformulating a higher-order differential equation into a system of first-order differential equations

We will look at how an ordinary differential equation of arbitrary order n :

$$y^{(n)} = f(t, y, y', y'', \dots, y^{(n-1)}), \quad (12.50)$$

with initial conditions:

$$y(t_0) = k_0, \quad y'(t_0) = k_1, \quad \dots \quad y^{(n-1)}(t_0) = k_{n-1} \quad (12.51)$$

is reformulated into a system of first-order differential equations.

First, instead of derivatives we introduce new variables:

$$y_i = y^{(i)}, \quad i = 0, 1, \dots, n-1. \quad (12.52)$$

The n th-order differential equation written with the new variables is:

$$y'_{n-1} = f(t, y_0, y_1, y_2, \dots, y_{n-1}). \quad (12.53)$$

We differentiate the new functions with respect to the independent variable:

$$y'_i = y^{(i+1)} = y_{i+1}, \quad i = 0, 1, \dots, n-2 \quad (12.54)$$

and

$$y'_{n-1} = f(t, y_0, y_1, y_2, \dots, y_{n-1}). \quad (12.55)$$

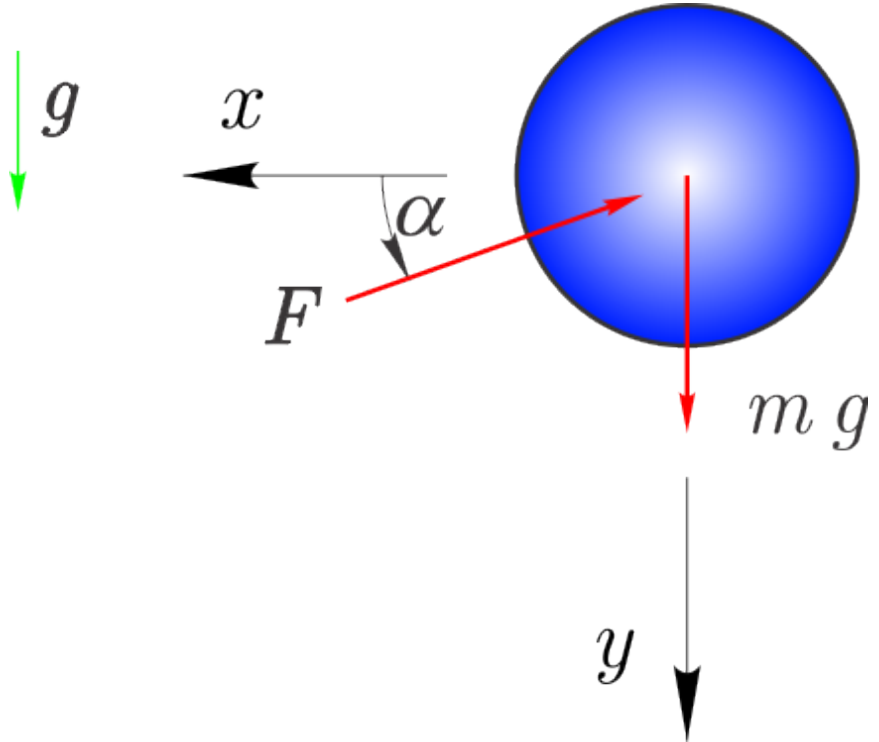
We have obtained a system of first-order ordinary differential equations:

$$\begin{aligned} y'_0 &= y_1 \\ y'_1 &= y_2 \\ &\dots \\ y'_{n-1} &= f(t, y_0, y_1, y_2, \dots, y_{n-1}) \end{aligned} \quad (12.56)$$

with initial conditions:

$$y_0(t_0) = k_0, \quad y_1(t_0) = k_1, \quad \dots \quad y_{(n-1)}(t_0) = k_{n-1}. \quad (12.57)$$

Numerical example



We return to the falling mass:

But this time let us write Newton's second law in terms of displacement (a second-order differential equation):

$$F_x = m x'', \quad (12.58)$$

$$m g + F_y = m y'', \quad (12.59)$$

m is the mass, g the gravitational acceleration, c the air-resistance coefficient, and x'' and y'' the acceleration in the chosen coordinate system. Initial conditions:

$$x(0) = y(0) = 0 \text{ m} \quad \text{and} \quad x'(0) = y'(0) = 5 \text{ m/s}. \quad (12.60)$$

We have a system of two second-order differential equations. By introducing new variables y_i :

$$y_0 = x, \quad y_1 = x', \quad y_2 = y, \quad y_3 = y'. \quad (12.61)$$

We prepare the system of first-order differential equations:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= F_x/m \\ y_2' &= y_3 \\ y_3' &= g + F_y/m \end{aligned} \quad (12.62)$$

Let us define the Python function of right-hand sides / first derivatives:

```
def f_air_resistance_coupled(t, y, g=9.81, m=1., c=0.5):
    x, vx, y, vy = y
    return np.array([vx, -c*vx*np.sqrt(vx**2+vy**2)/m, vy, g
                    -c*vy*np.sqrt(vx**2+vy**2)/m])
```

Let us define the initial condition and the time vector where we want the result:

```
y0 = np.array([0., 5., 0., 5.])
t = np.linspace(0, 5, 101)
t[:5]

array([0.   , 0.05, 0.1  , 0.15, 0.2  ])
```

We compute the solution:

```
y_ivp = solve_ivp(f_air_resistance_coupled, t_span=(t[0], t[-1]), y0=y0, t_eval=t)
```

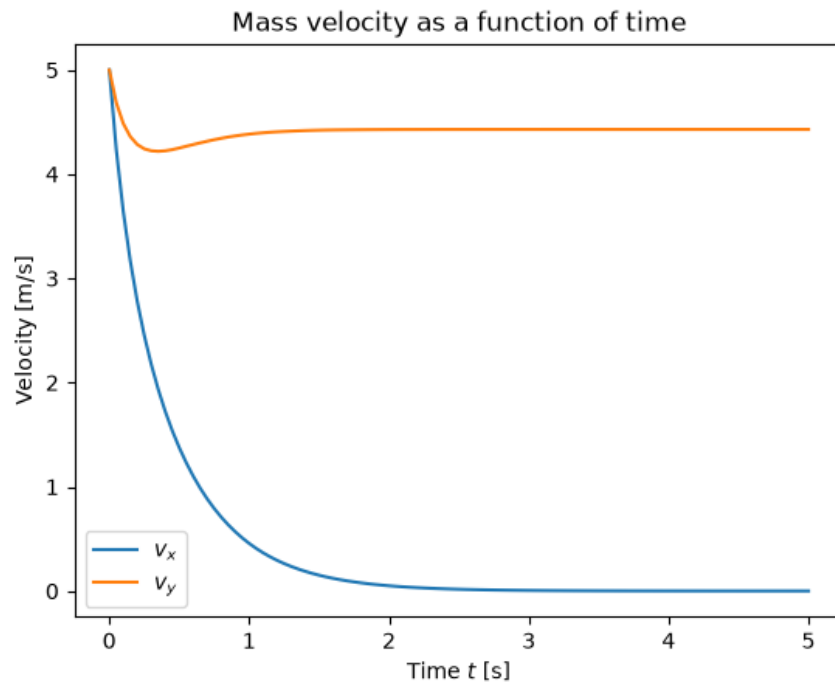
Let us look at the result $([x, x', y, y'])$:

```
y_ivp.y[:, :5]

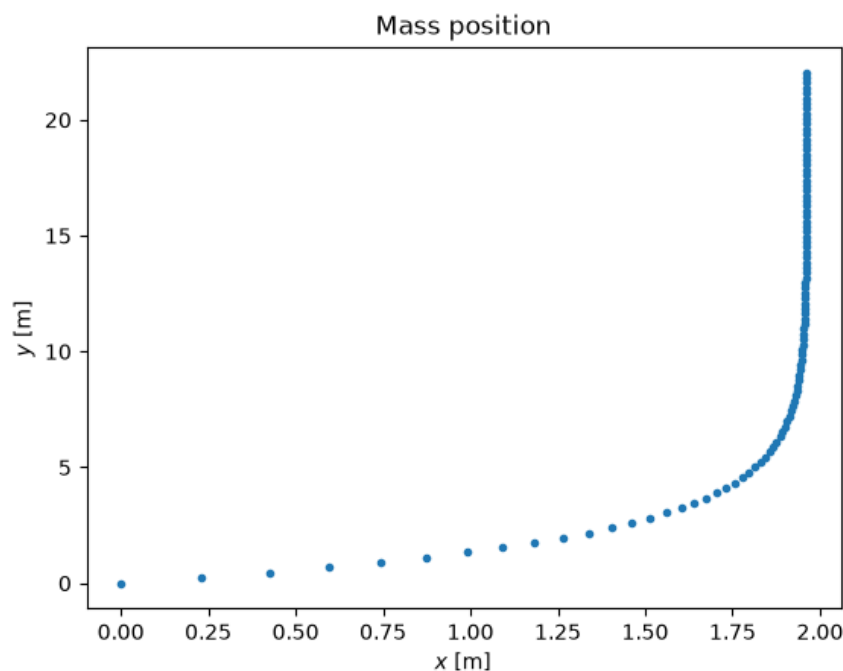
array([[0.          , 0.22992477, 0.42611996, 0.59587483, 0.74424286],
       [5.          , 4.23183658, 3.64002431, 3.16703014, 2.77775353],
       [0.          , 0.24153615, 0.47038377, 0.69124852, 0.9072755 ],
       [5.          , 4.68435696, 4.48497294, 4.36034218, 4.28444004]])
```

We plot the results; first the velocity, then the position (the y coordinate is positive downward)!

```
plt.plot(t, y_ivp.y[1], label='$v_x$')
plt.plot(t, y_ivp.y[3], label='$v_y$')
plt.title('Mass velocity as a function of time')
plt.xlabel('Time $t$ [s]')
plt.ylabel('Velocity [m/s]')
plt.legend()
plt.show()
```



```
plt.scatter(y_ivp.y[0], y_ivp.y[2], marker='.')
plt.xlabel('$x$ [m]')
plt.ylabel('$y$ [m]')
plt.title('Mass position')
plt.show()
```



12.7 Stability of solving differential equations*

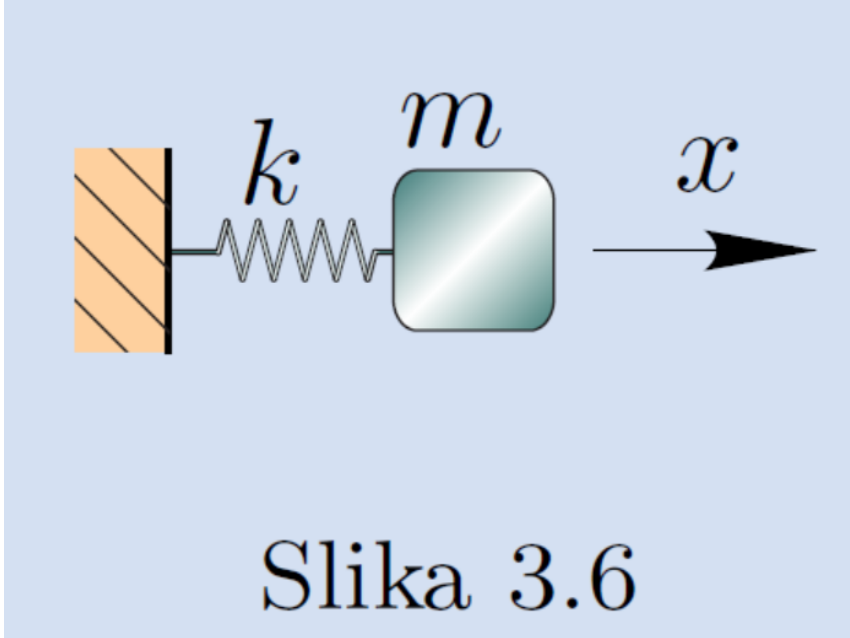
The numerical solution of differential equations is subject to *method error* and *round-off error*. These errors differ for different numerical methods.

Solving a differential equation is **stable** if a small change in the initial condition leads to a small change in the computed solution; otherwise we speak of **instability of the solution**.

Stability depends on the differential equation, on the numerical method used, and on the integration step h .

12.7.1 Example of a simple oscillator

Let us first look at an example of solving the differential equation of a simple oscillator.



The figure (source: Slavič, Dinamika, mehanska nihanja in mehanika tekočin, 2017) shows a dynamic system (mass m , stiffness k) whose differential equation is

$$m x'' + k x = 0. \quad (12.63)$$

We reformulate this differential equation into the standard form of free vibration:

$$x'' + \omega_0^2 x = 0, \quad (12.64)$$

where the natural angular frequency is:

$$\omega_0 = \sqrt{\frac{k}{m}} \quad (12.65)$$

and we expect a response of the form:

$$x(t) = A \cos(\omega_0 t) + B \sin(\omega_0 t). \quad (12.66)$$

If the initial conditions are:

$$x(0 \text{ s}) = x_0 \quad \text{and} \quad x'(0 \text{ s}) = 0 \text{ m/s}, \quad (12.67)$$

the solution of the initial value problem is:

$$x(t) = x_0 \cos(\omega_0 t). \quad (12.68)$$

Numerical example

First let us define the vector of initial conditions and the function of right-hand sides / first derivatives (we convert the second-order differential equation into a system of first-order differential equations $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$), where $y_0 = x, y_1 = x'$:

```
def f_oscillator(t, y, omega0=2*np.pi):
    """
    Right -hand -side function for an oscillator with one degree of freedom

    :param t: time
    :param y: list of initial values
    :param omega: natural angular frequency
    :return y': list of derivative values
    """
    return np.array([y[1], -omega0**2*y[0]])
```

Let us define the data and the analytical solution:

```
x0 = 1.
omega0 = 2*np.pi
x_initial_conditions = np.array([x0, 0.])
t1 = 4.
time = np.linspace(0, t1, 500)
displacement = x0*np.cos(omega0*time) # analytical solution
velocity = -x0*omega0*np.sin(omega0*time) # analytical solution
```

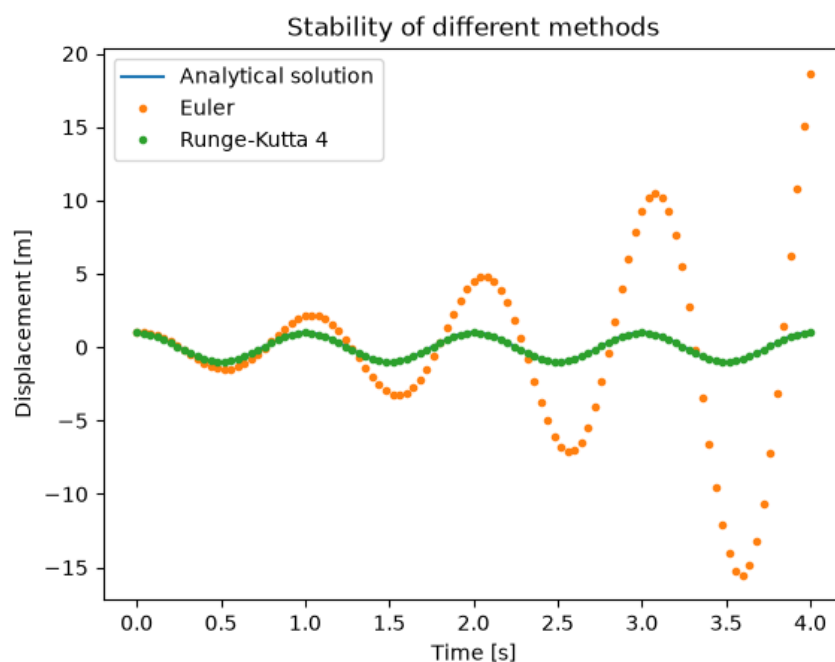
Solution using the Euler and fourth-order Runge-Kutta methods:

```
t_Eu = np.linspace(0, t1, 101)
t_RK4 = t_Eu
dt = t_Eu[1]

x_Eu = euler_system(f_oscillator, t_Eu, x_initial_conditions)
x_RK4 = solve_ivp(f_oscillator, t_span=(t_RK4[0], t_RK4[-1]),
y0=x_initial_conditions, t_eval=t_RK4).y
```

Let us plot the results:

```
plt.plot(time, displacement, label='Analytical solution')
plt.plot(t_Eu, x_Eu[:,0], '.', label='Euler')
plt.plot(t_RK4, x_RK4[0], '.', label='Runge -Kutta 4')
plt.legend()
plt.title('Stability of different methods')
plt.ylabel('Displacement [m]')
plt.xlabel('Time [s]')
plt.show()
```



We observe that Euler's method is unstable, and if we increased the step, the fourth-order Runge-Kutta method would also become unstable.

Why is Euler's method so unstable?

```
| x_Eu[:10,0]
|
| array([ 1.          ,  1.          ,  0.93683453,  0.8105036 ,  0.62499707,
|         0.3882947 ,  0.1121141 , -0.18859332, -0.49638247, -0.79225904])
```

Let us recall Euler's method:

$$x(t+h) = x(t) + x'(t)h, \quad (12.69)$$

which tells us that we determine the displacement $x(t+h)$ from the position $x(t)$ and the velocity $x'(t)$. The initial conditions stem from the maximum deflection $x(t=0) = 1$, and at that point the velocity is $x'(t=0) = 0$, which means that $x(t+h) = 1$. Already in the first step we therefore make a relatively large error. But why does the value then start to grow in an alternating fashion?

Recall that the analytical solution is $x(t) = x_0 \cos(\omega_0 t)$, and therefore $x'(t) = -\omega_0 x_0 \sin(\omega_0 t)$.

We substitute the prepared expressions into Euler's method and rearrange:

$$x(t+h) = x(t) + x'(t)h = x_0 (\cos(\omega_0 t) - \omega_0 h \sin(\omega_0 t)). \quad (12.70)$$

Assume we look at the state at a time $t = \pi/(2\omega_0)$, when $\cos(\omega_0 t) = 0$ and $\sin(\omega_0 t) = 1$:

$$x(t+h) = x_0 \underbrace{(-\omega_0 h)}_A. \quad (12.71)$$

If the absolute value of the expression A is greater than 1, then at time $t+h$ the value will be greater than in the previous step, and probably again in the next one. It follows that instability can occur. To avoid it, the following must hold:

$$|A| < 1 \quad \rightarrow \quad h < \frac{1}{\omega_0}. \quad (12.72)$$

Note: in some books you will also see the value $h < 2/\omega_0$; a unique bound for all differential equations cannot be defined; in general, however, the step is defined relative to the shortest period T in the differential equation (e.g.: $h < 2/\omega_0$ is essentially the same as $h < 2/(2\pi/T)$, i.e., $h < T/\pi$). The period T is defined relative to the highest natural frequency of the system $T = 1/f_{\max}$, which we compute from the eigenvalue of the system.

12.8 Additional: Example of the Van der Pol oscillator

The purpose of this example is to show how the choice of integrator can affect the speed of solving a problem! The Van der Pol oscillator is described [here](#).

Let us define the list of derivatives:

```
def f_van_der_pol(t, y, mu=1000):
    """
    Right -hand -side function for the Van der Pol oscillator

    :param t: time
    :param y: list of initial values
    :param mu: damping and nonlinearity parameter
    :return y': list of derivative values
    """
    return np.array([y[1], mu*(1 -y[0]**2)*y[1] -y[0]])
```

```
x_initial_conditions = np.array([1.5, 0.])
dt = 0.1
t1 = 3000
```

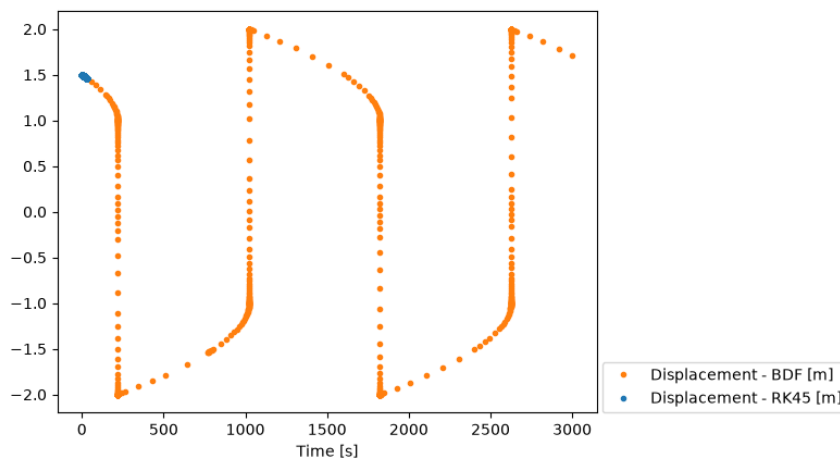
Solution using the RK45 method (this is an explicit scheme, which is not suitable for stiff systems of differential equations; solving is very slow, so we compute the solution only up to $t_1/100$):

```
vp_RK45 = solve_ivp(f_van_der_pol, t_span=(0., t1/100), y0=x_initial_conditions,
method='RK45')
```

The implicit BDF scheme (Backward Differentiation Formulas) turns out here to be much more suitable. Because of its stability, the steps can be much larger and therefore the solving is much faster:

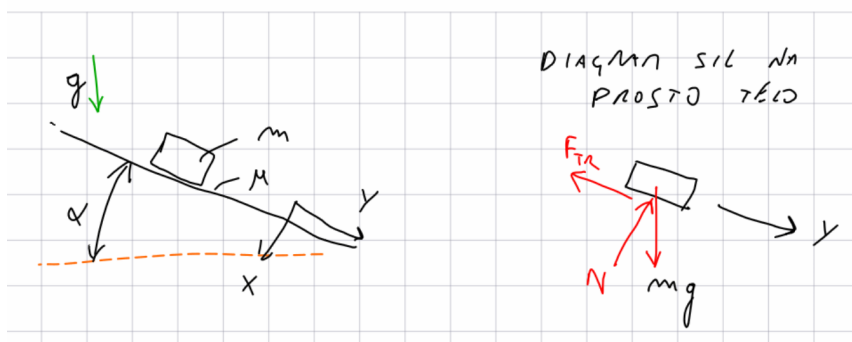
```
vp_BDF = solve_ivp(f_van_der_pol, t_span=(0., t1), y0=x_initial_conditions,
method='BDF')
```

```
plt.plot(vp_BDF.t, vp_BDF.y[0], 'C1.', label='Displacement - BDF [m]')
plt.plot(vp_RK45.t, vp_RK45.y[0], 'C0.', label='Displacement - RK45 [m]')
plt.xlabel('Time [s]')
plt.legend(loc=(1.01, 0));
```



12.9 Additional: symbolic solving of a second-order differential equation

We will look at the example shown in the figure, where a mass m is on an incline of slope α . The coefficient of friction is μ , and the gravitational acceleration is g . The initial velocity is $\dot{y}(0s) = v_0$, the displacement $y(0s) = 0m$.



The equation of motion (only for the y direction) is defined according to Newton's second law (see the free-body force diagram).

Derivation of the equation of motion

```
y = sym.Function('y')
m, mu, g, alpha, t, v0 = sym.symbols('m, mu, g, alpha, t, v0')
eq = sym.Eq(m*y(t).diff(t,2), m*g*sym.sin(alpha) -m*g*sym.cos(alpha)*mu)
eq
```

$$m \frac{d^2}{dt^2} y(t) = -gm\mu \cos(\alpha) + gm \sin(\alpha) \quad (12.73)$$

The solution of the equation is:

```
dsol = sym.dsolve(eq, y(t))
dsol
```

$$y(t) = C_1 + C_2 t + \frac{gt^2(-\mu \cos(\alpha) + \sin(\alpha))}{2} \quad (12.74)$$

To determine C_1 and C_2 , we substitute $t = 0$ s:

```
dsol.args[1].subs(t, 0)
```

$$C_1 \quad (12.75)$$

Then we differentiate with respect to time and again substitute $t = 0$ s:

```
dsol.args[1].diff(t).subs(t, 0)
```

$$C_2 \quad (12.76)$$

Based on the initial conditions, we have therefore determined the constants:

```
initial_conditions = {'C1': 0, 'C2': v0}
```

The solution follows:

```
solution = dsol.args[1].subs(initial_conditions)
solution
```

$$\frac{gt^2(-\mu \cos(\alpha) + \sin(\alpha))}{2} + tv_0 \quad (12.77)$$

Let us prepare functions for the numerical call:

```
data = {mu: 0.3, alpha: 15*np.pi/180, v0: 1., g: 9.81} # here we use np.pi so that we
have a numerical value
```

```
displacement = sym.lambdify(t, solution.subs(data), 'numpy')
velocity = sym.lambdify(t, solution.diff(t).subs(data), 'numpy')
```

```
print('Displacement at 0s: {:.g}m'.format(displacement(0)))
print('Velocity at 0s: {:.g}m/s'.format(velocity(0)))
```

```
Displacement at 0s: 0m
Velocity at 0s: 1m/s
```

Let us prepare the plot:

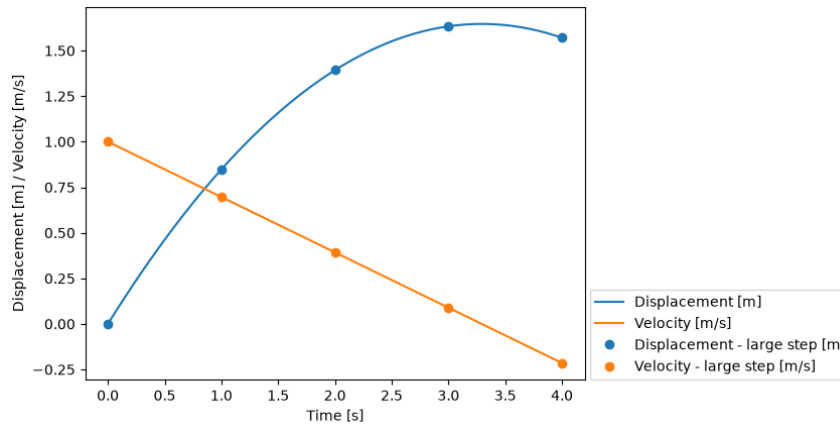
```

time = np.linspace(0, 4, 100)
time2 = np.linspace(0, 4, 5)

def figure():
    plt.plot(time, displacement(time), 'C0', label='Displacement [m]')
    plt.plot(time, velocity(time), 'C1', label='Velocity [m/s]')
    plt.plot(time2, displacement(time2), 'C0o', label='Displacement - large step [m]')
    plt.plot(time2, velocity(time2), 'C1o', label='Velocity - large step [m/s]')
    plt.xlabel('Time [s]')
    plt.ylabel('Displacement [m] / Velocity [m/s]')
    plt.legend(loc=(1.01, 0));
    plt.show()

figure()

```



Symbolic reformulation of the differential equation into a system of first-order differential equations

Let us recall the original differential equation:

eq

$$m \frac{d^2}{dt^2} y(t) = -gm\mu \cos(\alpha) + gm \sin(\alpha) \quad (12.78)$$

Let us define new variables and prepare the function f :

```

y0 = sym.Function('y0')
y1 = sym.Function('y1')
f = sym.simplify(eq.args[1]/m)
f

```

$$g(-\mu \cos(\alpha) + \sin(\alpha)) \quad (12.79)$$

Now let us connect the new variables.

Let dy_0/dt be equal to y_1 :

```

eq1 = sym.Eq(y0(t).diff(t), y1(t))
eq1

```

$$\frac{d}{dt} y_0(t) = y_1(t) \quad (12.80)$$

Let the derivative dy_1/dt (which is essentially y'') be equal to the function f :

```
eq2 = sym.Eq(y1(t).diff(t), f)
eq2
```

$$\frac{d}{dt}y_1(t) = g(-\mu \cos(\alpha) + \sin(\alpha)) \quad (12.81)$$

We write the expressions above in vector form:

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}). \quad (12.82)$$

```
y_derivative = [y0(t).diff(t), y1(t).diff(t)]
y_derivative
```

$$\left[\frac{d}{dt}y_0(t), \frac{d}{dt}y_1(t) \right] \quad (12.83)$$

```
f_vec = [y1(t), f]
f_vec
```

$$[y_1(t), g(-\mu \cos(\alpha) + \sin(\alpha))] \quad (12.84)$$

Let us now recall `f_vec`:

```
f_vec
```

$$[y_1(t), g(-\mu \cos(\alpha) + \sin(\alpha))] \quad (12.85)$$

If we solve numerically, then we write the function `f(t, y)` as:

```
acceleration = float((eq.args[1]/m).simplify().subs(data))
acceleration # investigate why we defined it this way here! hint: type(acceleration)
```

$$-0.303704874312999 \quad (12.86)$$

```
def F_block(t, y):
    return np.array([y[1], acceleration], dtype=float)
```

Let us check the function at the initial time $t = 0$ s and at the initial conditions $[y_0, y_1] = [0, v_0]$:

```
y_initial_conditions = np.array([0, data[v0]])
y_initial_conditions
```

```
array([0., 1.])
```

```
F_block(0., y_initial_conditions)
```

```
array([ 1., -0.30370487])
```

Now let us use Euler's method:

```
%%timeit
x_Eu = np.linspace(0, 4, 5)
y_Eu = euler_system(F_block, x_Eu, np.array([0, 1.]))
y_Eu
```

```
array([[ 0.          ,  1.          ],
       [ 1.          ,  0.69629513],
       [ 1.69629513,  0.39259025],
       [ 2.08888538,  0.08888538],
       [ 2.17777075, -0.2148195 ]])
```

We plot and compare with the analytical solution:

```
def plot_euler(n=5):
    x_Eu = np.linspace(0, 4, n)
    y_Eu = euler_system(F_block, x_Eu, np.array([0, 1.]))
    plt.title('Euler\'s method with step $h={:g}$'.format(x_Eu[1] - x_Eu[0]))
    plt.plot(time, displacement(time), 'CO', label='Displacement - analytical [m]')
    plt.plot(time, velocity(time), 'C1', label='Velocity - analytical [m/s]')
    plt.plot(x_Eu, y_Eu[:, 0], 'CO.', label='Displacement - Euler [m]')
    plt.plot(x_Eu, y_Eu[:, 1], 'C1.', label='Velocity - Euler [m/s]')
    plt.xlabel('Time [s]')
    plt.ylabel('Displacement [m] / Velocity [m/s]')
    plt.ylim(-0.5, 2.5)
    plt.legend(loc=(1.01, 0))
    plt.show();

interact(plot_euler, n=(3, 10, 1));

interactive(children=(IntSlider(value=5, description='n', max=10, min=3), Output()),
            _dom_classes=('widget -int...
```

Chapter 13

Numerical solution of differential equations - boundary value problem

13.1 Solving two-point boundary value problems

By a two-point boundary value problem we mean a second-order ordinary differential equation of the form:

$$\ddot{y} = f(t, y, \dot{y}), \quad (13.1)$$

subject to prescribed *boundary conditions*:

$$y(a) = \alpha \quad \text{in} \quad y(b) = \beta. \quad (13.2)$$

The methods we learned for solving *initial value problems* are not directly applicable here, because we do not have the derivative specified at the initial point $t = a$.

In what follows we will look at two different approaches to solving boundary value problems:

1. the so-called **shooting method**,
2. the **finite difference method**.

13.2 Shooting method

We solve the boundary value problem:

$$\ddot{y} = f(t, y, \dot{y}), \quad y(a) = \alpha, \quad y(b) = \beta, \quad (13.3)$$

which we convert into an initial value problem by choosing:

$$\dot{y}(a) = u. \quad (13.4)$$

We solve the problem with numerical methods for the initial value problem and denote the solution by $\theta(u, t)$.

The boundary value problem is solved once we choose u such that:

$$r(u) = \theta(u, b) - \beta = 0. \quad (13.5)$$

We have obtained a nonlinear equation that we must solve; to compute the value of the boundary residual $r(u)$ we must numerically solve the initial value problem.

To solve the equation $r(u) = 0$ we can use the secant method. We choose u_0 and u_1 and at the i -th step we compute:

$$u_{i+1} = u_i - r(u_i) \frac{u_i - u_{i-1}}{r(u_i) - r(u_{i-1})}, \quad i = 2, 3, \dots \quad (13.6)$$

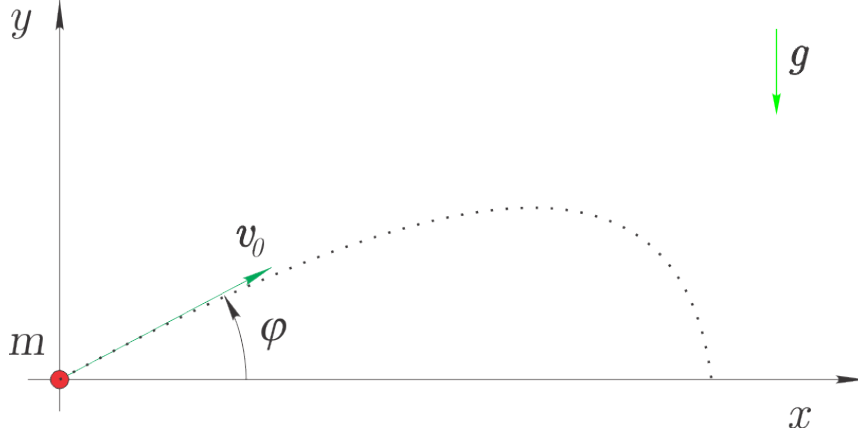
We stop when:

$$|r(u_{i+1})| < \epsilon. \quad (13.7)$$

The solution of the shooting method carries the error ϵ of the method for solving the nonlinear equation and the error of the numerical method for solving the initial value problem.

13.2.1 Numerical example: projectile motion

The figure shows a projectile of mass m launched with velocity $\mathbf{v}_0$.



The magnitude of the air-drag force is $|\mathbf{F}| = c|\mathbf{v}|^2$, then the equations of motion are:

$$x''(t) = F_x/m \quad y''(t) = F_y/m - g. \quad (13.8)$$

The force components are (see the derivation in the chapter on solving the initial value problem of a system of differential equations):

$$F_x = -c x' \sqrt{x'^2 + y'^2}, \quad F_y = -c y' \sqrt{x'^2 + y'^2}. \quad (13.9)$$

Vertical launch

First we assume that $\varphi = 90^\circ$ and therefore there is no motion in the x direction. We are interested in the solution where the projectile is launched from height $y = 0$ m and must, at time $t = b = 1$ s, be at height $y(b) = 10$ m. We have defined the boundary value problem:

$$y''(t) = F_y/m - g, \quad y(0) = 0, \quad y(1) = 10. \quad (13.10)$$

First we must transform the second-order equation:

$$y'' = f(t, y, y') = F_y/m - g \quad (13.11)$$

into a system of two first-order equations. We use $y_i = y^{(i)}$ and take into account $F_y = -c y' \sqrt{y'^2}$.

We differentiate $y'_i = y_{i+1}$ and set up the system of first-order equations:

$$\begin{aligned} y'_0 &= y_1 \\ y'_1 &= -c y' \sqrt{y'^2}/m - g \end{aligned} \quad (13.12)$$

Let us import the modules `numpy` and `scipy.integrate.solve_ivp`:

```
import numpy as np
from scipy.integrate import solve_ivp
```

Let us prepare the list of right-hand-side functions:

```
def f_vert(t, y, g=9.81, m=1., c=0.5):
    return np.array([y[1], -g -c*y[1]*np.sqrt(y[1]**2)/m])
```

and we prepare a function to compute the boundary residual at time b as a function of the initial velocity v_0 (the default air drag is $c = 0.1$):

```
def r(v0=100., t=None, target_position=10., g=9.81, m=1., c=0.1):
    def f_vert_tmp(t, y): # prepare a function with the desired parameters
        return f_vert(t, y, g=g, m=m, c=c)
    sol = solve_ivp(f_vert_tmp, t_span=(t[0], t[-1]), y0=np.array([0., v0]))
    r = sol.y[0, -1] - target_position
    return r
```

Let us check the boundary residual for the initial condition $v_0 = y' = 50$ m/s:

```
t = np.linspace(0, 1, 100)
r(v0=50., t=t)

np.float64(5.63455093999735)
```

We observe that at 1 second the mass is 5.635 m above the target height.

Our goal is for the mass to reach the position of 10 m at 1 second with accuracy $1e-6$:

```
target_position = 10
epsilon = 1e -6
```

Let us now carry out the secant method:

```
x0 = 100
x1 = 50
for i in range(10):
    f0 = r(v0=x0, t=t, target_position=target_position)
    f1 = r(v0=x1, t=t, target_position=target_position)
    x2 = x1 - f1 * (x1 - x0)/(f1 - f0)
    err = r(v0=x2, t=t, target_position=target_position)
    print(f'The new estimate is v0={x2}, the error is {err}.')
    x0 = x1
    x1 = x2
    if abs(err)<epsilon:
        solution = x2
        print(f'Solution v0={solution}')
        break
```

```
The new estimate is v0=5.644178759347113, the error is -9.672536331804414.
The new estimate is v0=33.67258709458857, the error is 2.2187017725280302.
The new estimate is v0=28.442965190719487, the error is 0.8225839451367332.
The new estimate is v0=25.361704474139284, the error is -0.10153714557803362.
The new estimate is v0=25.70025579721459, the error is 0.004276366407131249.
The new estimate is v0=25.686573522825174, the error is 2.16308919487318e -05.
The new estimate is v0=25.686503962731987, the error is -4.623860405672531e -09.
Solution v0=25.686503962731987
```

Let us now look at the computed solution:

```
sol = solve_ivp(lambda t, y: f_vert(t, y, c=0.1),
                t_span=(t[0], t[-1]), y0=np.array([0., solution]),
                t_eval=t
                )
```

Above we used a `lambda` expression ([documentation](#)). The expression `lambda t, y: f_vert(t, y, c=0.1)` is equivalent to:

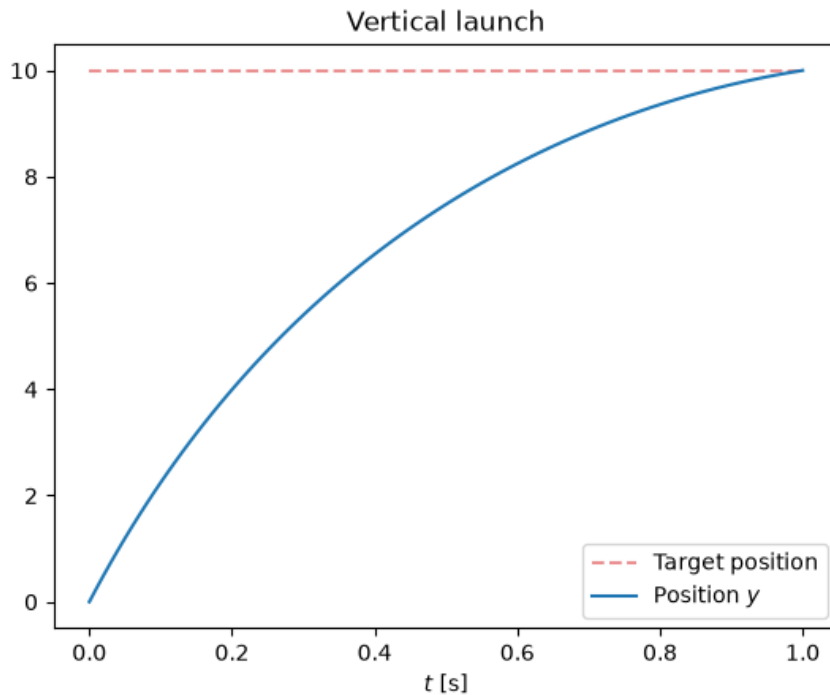
```
def function_name(t, y):
    return f_vert(t, y, c=0.1)
```

Let us import matplotlib:

```
import matplotlib.pyplot as plt
%matplotlib inline
```

We display the result:

```
plt.title('Vertical launch')
plt.hlines(target_position, 0., 1, 'C3', linestyle='dashed', label='Target position',
alpha=0.5)
plt.plot(sol.t, sol.y[0], label='Position $y$')
plt.xlabel('$t$ [s]')
plt.legend(loc=4)
plt.show()
```



Projectile (oblique) launch

Let us now look at a general projectile (oblique) launch (the differential equations are already defined above). First we must transform the system of second-order differential equations into a system of first-order equations.

We use:

$$y_0 = x, \quad y_1 = x', \quad y_2 = y, \quad y_3 = y' \quad (13.13)$$

and we obtain the system of first-order differential equations:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= F_x/m \\ y_2' &= y_3 \\ y_3' &= F_y/m - g. \end{aligned} \quad (13.14)$$

Let us prepare the list of right-hand-side functions:

```
def f_oblique(t, y, g=9.81, m=1., c=0.1):
    x, vx, yy, vy = y
    return np.array([vx, -c*vx*np.sqrt(vx**2+vy**2)/m, vy, -g
                    -c*vy*np.sqrt(vx**2+vy**2)/m])
```

Let us also prepare a function to compute the boundary residual at time $b = 1$ as a function of the initial velocity vector $\mathbf{v}_0$:

```
def r_oblique(v0=[5., 100.], t=None, target_position=np.array([10, 5.]), g=9.81,
m=1., c=0.1):
    sol = solve_ivp(lambda t, y: f_oblique(t, y, g=g, m=m, c=c),
                    t_span=(t[0], t[-1]),
                    y0=[0, v0[0], 0, v0[1]])
    r = sol.y[0:3:2, -1] - target_position
    return r
```

Let us check the boundary residual for the initial condition $\mathbf{v}_0 = [100., 100.]$ m/s:

```
| r_oblique(v0=[100., 100.], t=t)
|
| array([ 9.66195325, 11.84726075])
```

For finding the root of a system of nonlinear functions we have already encountered the function `scipy.optimize.root` ([documentation](#)):

```
| root(fun, x0, args=(), method='hybr', jac=None,
|      tol=None, callback=None, options=None)
```

First we import it:

```
| from scipy import optimize
```

Then we use it with an initial guess $\mathbf{v}_0$:

```
| solution = optimize.root(r_oblique, np.array([100., 100.]), args=(t,))
```

The solution is:

```
| solution
|
| message: The solution converged.
| success: True
| status: 1
|      fun: [ 1.155e -13 -3.260e -13]
|      x: [ 1.938e+01 1.655e+01]
| method: hybr
| nfev: 19
| fjac: [[ -9.339e -01  3.575e -01]
|        [ -3.575e -01 -9.339e -01]]
|      r: [ -4.078e -01  2.409e -01 -3.754e -01]
|      qtf: [ 1.067e -09 -1.311e -09]
```

The attribute `solution.x` contains the vector of computed solutions (`[19.37894314, 16.55482478]`). Let us check the boundary residual at the computed solution:

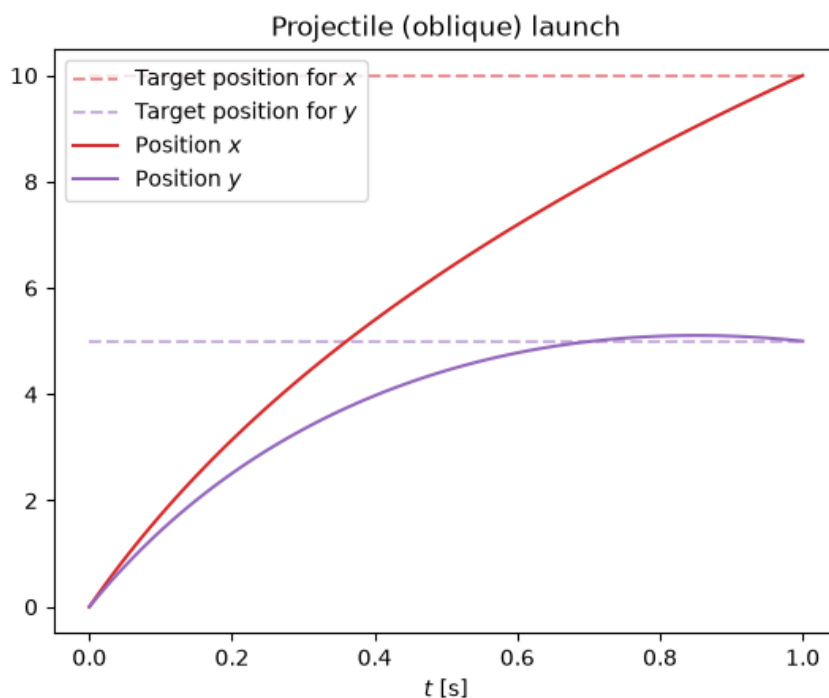
```
| r_oblique(v0=solution.x, t=t)
|
| array([ 1.15463195e -13, -3.25961480e -13])
```

Let us now look at the computed solution:

```
sol = solve_ivp(lambda t, y: f_oblique(t, y, g=9.81, m=1., c=.1),
                t_span=(t[0], t[-1]),
                y0=[0, solution.x[0], 0, solution.x[1]],
                t_eval=t)
```

We display the result:

```
plt.title('Projectile (oblique) launch')
plt.hlines(10,0., 1, 'C3', linestyle='dashed', label='Target position for $x$',
alpha=0.5)
plt.hlines(5,0., 1, 'C4', linestyle='dashed', label='Target position for $y$',
alpha=0.5)
plt.plot(sol.t, sol.y[0], label='Position $x$', color='C3')
plt.plot(sol.t, sol.y[2], label='Position $y$', color='C4')
plt.xlabel('$t$ [s]')
plt.legend()
plt.show()
```



13.2.2 Using `scipy.integrate.solve_bvp`

Instead of `scipy.integrate.solve_ivp` and `scipy.optimize` we can use the built-in function for solving boundary value problems `scipy.integrate.solve_bvp` (BVP - Boundary Value Problem):

```
scipy.integrate.solve_bvp(fun, bc, x, y,
                          p=None, S=None, fun_jac=None, bc_jac=None,
                          tol=0.001, max_nodes=1000, verbose=0)
```

An explanation of all the arguments is in the [documentation](#); here we will highlight a few:

- `fun` is the right-hand side (`func(t, y...)`),
- `bc` is the boundary residual: `bc(y(a), y(b)) = 0`
- `x` numeric array (dimension (`m`)) of the independent variable `x[0]=a` and `x[-1]=b`,
- `y` numeric array (dimension (`n, m`)) of initial values.

The result of calling `solve_bvp` is an object with attributes (selected):

- `x` the values of the independent variable at which the result is computed,
- `y` the result,
- `sol` the result in the form of a cubic spline,
- `success` is `True` if the solution has converged.

Let us look at an example:

```
| from scipy.integrate import solve_bvp
```

Let us define the independent variable

```
| t = np.linspace(0, 1, 15)
```

And the boundary residual (we want the position at time `b` to be $x = 10$ m and $y = 5$ m):

```
| def boundary_residual(ya, yb):
|     return np.array([ya[0], ya[2], yb[0] -10, yb[2] -5])
```

Let us also define the initial velocity values (as an initial guess):

```
| y0 = np.zeros((4, t.size))
| y0[1] = 5 # velocity in the x direction
| y0[3] = 100 # velocity in the y direction
```

```
| y0[:, :3]
```

```
| array([[ 0.,  0.,  0.],
|        [ 5.,  5.,  5.],
|        [ 0.,  0.,  0.],
|        [100., 100., 100.]])
```

And let us solve the boundary value problem and the description of the result

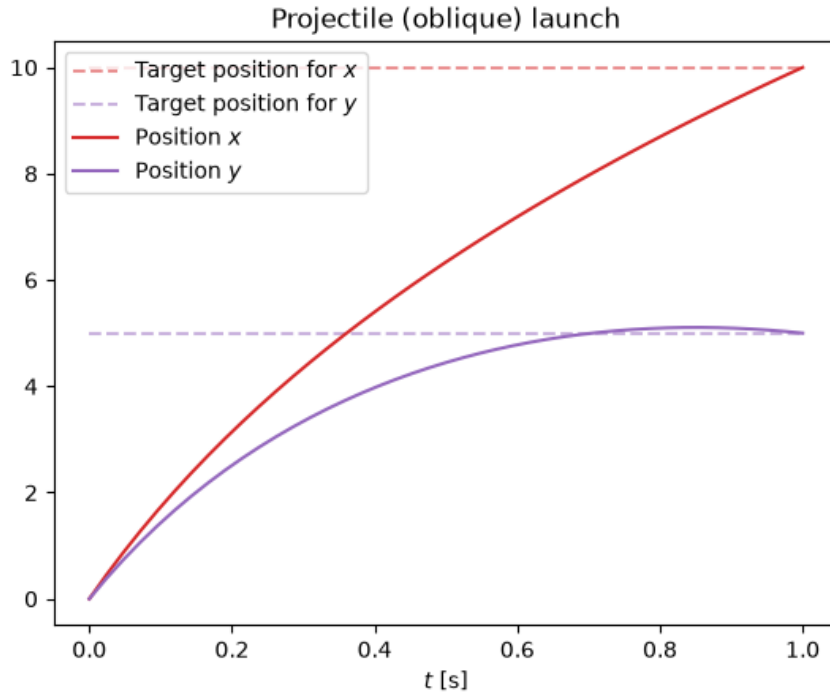
```
| sol = solve_bvp(lambda t, y: f_oblique(t, y, g=9.81, m=1., c=.1),
|               bc=boundary_residual, x=t, y=y0)
```

```
| sol
```

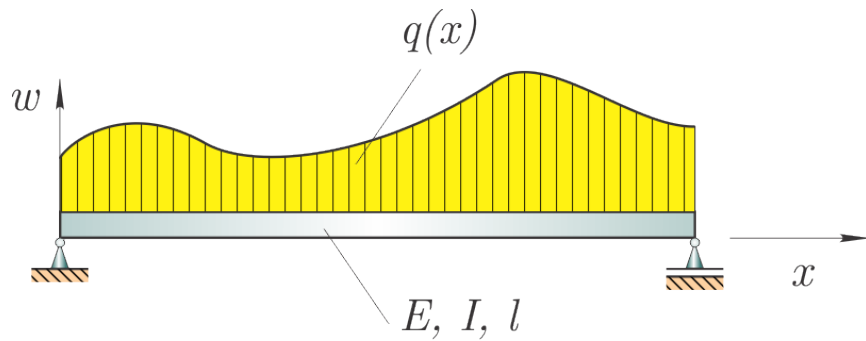
```
| message: The algorithm converged to the desired accuracy.
|         success: True
|         status: 0
|         x: [ 0.000e+00  2.381e -02 ...  9.762e -01  1.000e+00]
|         sol: <scipy.interpolate._interpolate.PPoly object at 0x7e1922de8f20>
|         p: None
|         y: [[ -1.588e -22  4.482e -01 ...  9.853e+00  1.000e+01]
|             [ 1.939e+01  1.828e+01 ...  6.210e+00  6.117e+00]
|             [ -3.176e -22  3.801e -01 ...  5.031e+00  5.000e+00]
|             [ 1.656e+01  1.539e+01 ... -1.189e+00 -1.403e+00]]
|         yp: [[ 1.939e+01  1.828e+01 ...  6.210e+00  6.117e+00]
|             [ -4.944e+01 -4.369e+01 ... -3.926e+00 -3.839e+00]
|             [ 1.656e+01  1.539e+01 ... -1.189e+00 -1.403e+00]
|             [ -5.205e+01 -4.659e+01 ... -9.058e+00 -8.929e+00]]
|         rms_residuals: [ 4.302e -05  3.582e -05 ...  1.297e -06  4.151e -06]
|         niter: 2
```

Let us display the result:

```
plt.title('Projectile (oblique) launch')
plt.hlines(10,0., 1, 'C3', linestyle='dashed', label='Target position for $x$',
alpha=0.5)
plt.hlines(5,0., 1, 'C4', linestyle='dashed', label='Target position for $y$',
alpha=0.5)
plt.plot(sol.x, sol.y[0], label='Position $x$', color='C3')
plt.plot(sol.x, sol.y[2], label='Position $y$', color='C4')
plt.xlabel('$t$ [s]')
plt.legend()
plt.show()
```



13.2.3 Numerical example: loaded beam



Let us look at a beam:

The deflection $w(x)$ of the beam is described by a fourth-order differential equation:

$$-EI \frac{d^4}{dx^4} w(x) + q(x) = 0. \quad (13.15)$$

The known constants are E, I, l and $q(x)$ is the distributed load.

Boundary conditions (simply supported beam):

$$w(0) = w(l) = 0 \quad \text{in} \quad w''(0) = w''(l) = 0 \quad (13.16)$$

The parameters are:

- $I = 2.1 \cdot 10^{-5} \text{ m}^4$,

- $E = 2.1 \cdot 10^{11} \text{ N/m}^2$,
- $l = 10 \text{ m}$.

The distributed load $q(x)$ will be defined later.

First we must transform the fourth-order differential equation into a system of first-order differential equations. We use:

$$y_0 = w, \quad y_1 = w', \quad y_2 = w'', \quad y_3 = w''' \quad (13.17)$$

and we obtain the system of first-order differential equations:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= q(x)/(EI). \end{aligned} \quad (13.18)$$

Let us prepare various distributed loads:

```
def q_constant(x, F_0=1e3, l=10):
    return -F_0*np.ones_like(x)

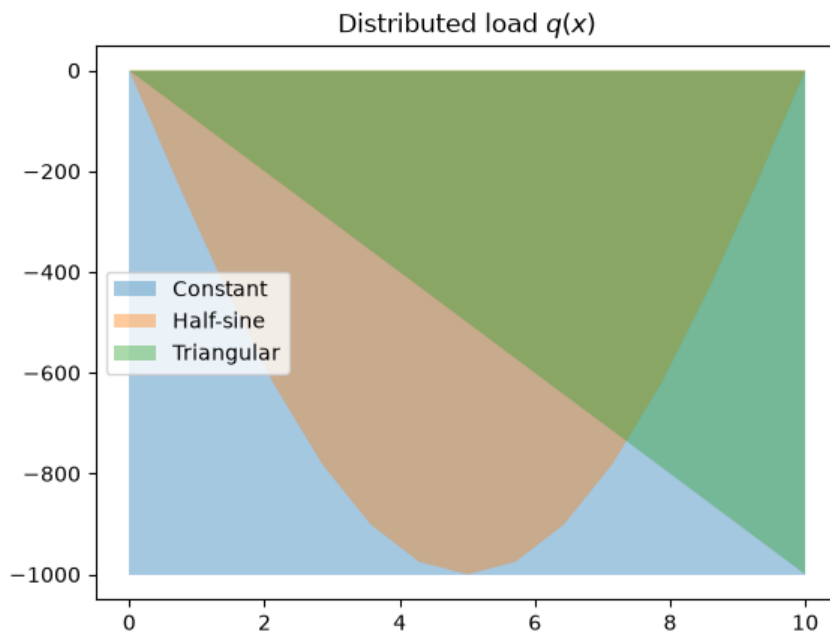
def q_triangular(x, F_0=1e3, l=10):
    return -F_0*x/l

def q_half_sine(x, F_0=1e3, l=10):
    return -F_0*np.sin(np.pi*x/l)
```

Let us define the length (for the other parameters, I, E , we will use the defaults):

```
l = 10.
```

```
x = np.linspace(0, l, 15)
plt.fill_between(x, q_constant(x), alpha=0.4, label='Constant')
plt.fill_between(x, q_half_sine(x), alpha=0.4, label='Half -sine')
plt.fill_between(x, q_triangular(x), alpha=0.4, label='Triangular')
plt.title('Distributed load $q(x)$')
plt.legend()
plt.show()
```



Let us prepare the list of right-hand-side functions:

```
def f_beam_constant(x, y, E=2.1e11, I=2.1e -5):
    return np.array([y[1], y[2], y[3], q_constant(x)/(E*I)], dtype=float)
def f_beam_half_sine(x, y, E=2.1e11, I=2.1e -5):
    return np.array([y[1], y[2], y[3], q_half_sine(x)/(E*I)])
def f_beam_triangular(x, y, E=2.1e11, I=2.1e -5):
    return np.array([y[1], y[2], y[3], q_triangular(x)/(E*I)])
```

Let us now define the boundary conditions, i.e. the boundary residual (the deflection and the moment at the ends are both zero):

```
def boundary_beam(ya, yb):
    return np.array([ya[0], ya[2], yb[0], yb[2]])
```

Let us also define the initial values of the slope (w') and the shear force (w''') (as an initial guess):

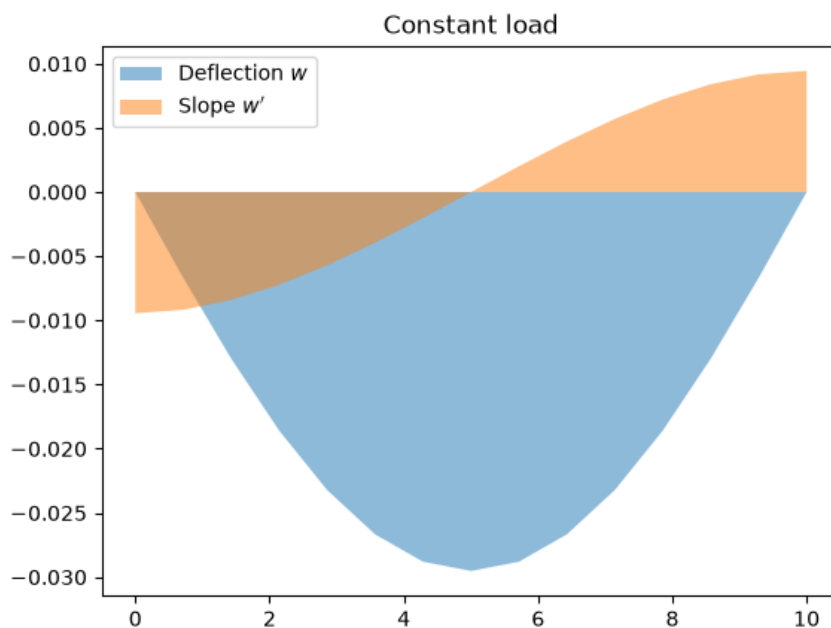
```
y0 = np.zeros((4, x.size), dtype=float)
# those that are not known could also be left as zero:
# y0[1] = 1. # w'
# y0[3] = 1. # w'''
```

And let us solve the boundary value problem (for all three load types):

```
sol_const = solve_bvp(f_beam_constant, bc=boundary_beam, x=x, y=y0)
sol_half_sin = solve_bvp(f_beam_half_sine, bc=boundary_beam, x=x, y=y0)
sol_triangular = solve_bvp(f_beam_triangular, bc=boundary_beam, x=x, y=y0)
```

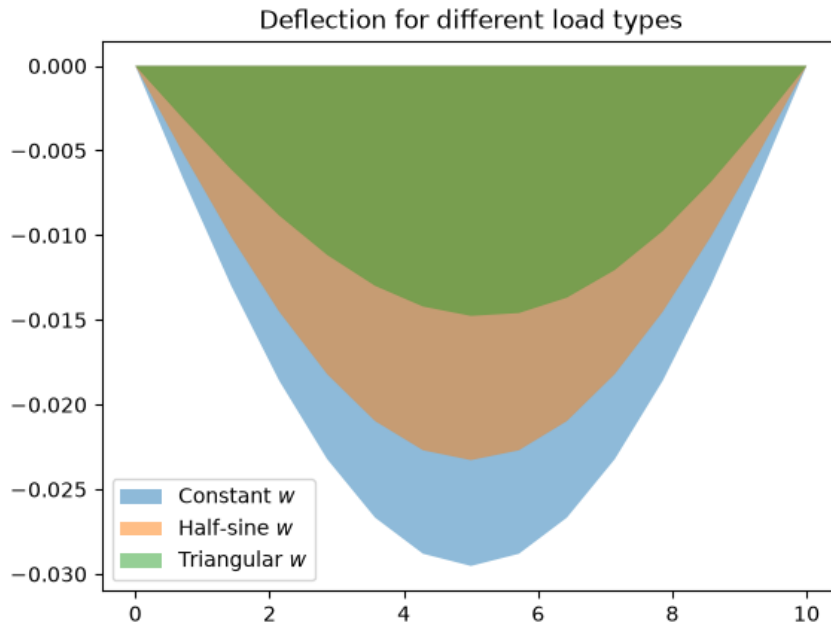
Let us display the result for the constant load:

```
plt.fill_between(sol_const.x, sol_const.y[0], label='Deflection $w$', alpha=0.5)
plt.fill_between(sol_const.x, sol_const.y[1], label='Slope $w\$', alpha=0.5)
plt.title('Constant load')
plt.legend()
plt.show()
```



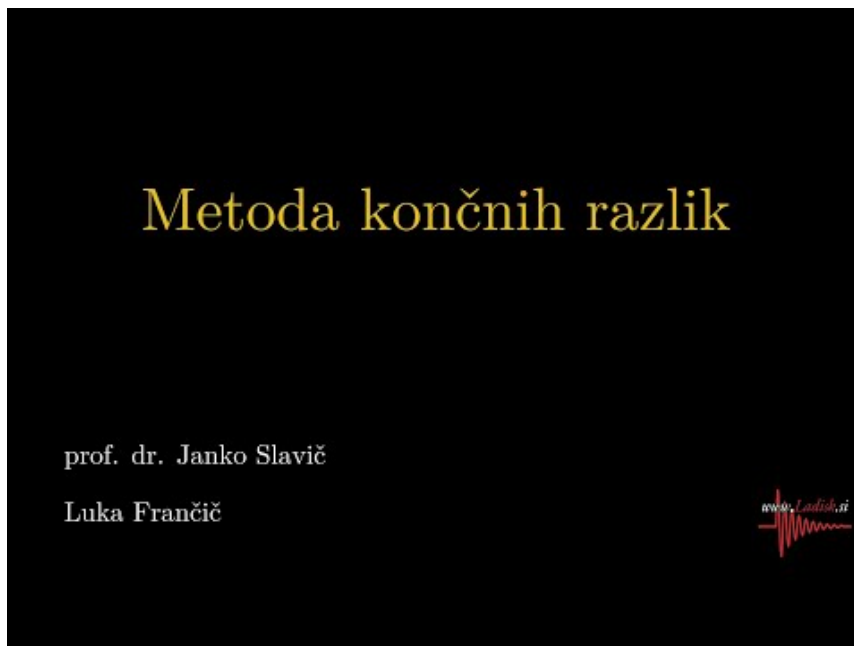
And a comparison of the deflection for the different load types:

```
plt.fill_between(sol_const.x, sol_const.y[0], label='Constant $w$', alpha=0.5)
plt.fill_between(sol_half_sin.x, sol_half_sin.y[0], label='Half -sine $w$',
alpha=0.5)
plt.fill_between(sol_triangular.x, sol_triangular.y[0], label='Triangular $w$',
alpha=0.5)
plt.title('Deflection for different load types')
plt.legend()
plt.show()
```



13.3 Finite difference method

```
from IPython.display import YouTubeVideo
YouTubeVideo('YozKGsNYZ6Y', width=800, height=300)
```



We solve the boundary value problem:

$$y'' = f(t, y, y'), \quad y(a) = \alpha, \quad y(b) = \beta. \quad (13.19)$$

So we have:

$$a = t_0, \quad b = t_{n-1}. \quad (13.20)$$

In the finite difference method for solving the boundary value problem we use a difference scheme. Assume we have an interval $[a, b]$ over which we solve the differential equation (the independent variable) divided into equal subintervals (there are n points):

$$t = [t_0, t_1, \dots, t_{n-1}]. \quad (13.21)$$

We replace the derivatives with the central difference scheme:

Derivative $\downarrow \setminus$ Values $\rightarrow$	y_{i-2}	y_{i-1}	y_i	y_{i+1}	y_{i+2}
$y'_i = \frac{1}{h}$	0	-0.5	0	0.5	0
$y''_i = \frac{1}{h^2}$	0	1	-2	1	0
$y'''_i = \frac{1}{h^3}$	-0.5	1	0	-1	0.5
$y^{(4)}_i = \frac{1}{h^4}$	1	-4	6	-4	1

The central difference scheme shown has a second-order error $\mathcal{O}(h^2)$.

At the i -th point we write the second-order ordinary differential equation with central differences as:

$$\frac{1}{h^2} (y_{i-1} - 2y_i + y_{i+1}) + \mathcal{O}(h^2) = f\left(t_i, y_i, \frac{1}{2h} (-y_{i-1} + y_{i+1}) + \mathcal{O}(h^2)\right). \quad (13.22)$$

If we neglect the error of the method:

$$\frac{1}{h^2} (y_{i-1} - 2y_i + y_{i+1}) = f\left(t_i, y_i, \frac{1}{2h} (-y_{i-1} + y_{i+1})\right), \quad i = 1, 2, \dots, n-2. \quad (13.23)$$

The above equation can be written for the $n-2$ interior points, which means that to obtain a solvable system of equations for n unknowns we are still missing two equations. These two equations are the boundary conditions:

$$y_0 = \alpha, \quad y_{n-1} = \beta. \quad (13.24)$$

In the case of a linear boundary value problem, to compute the n unknowns y_i we must solve a system of n linear equations (and if it is nonlinear, then a system of nonlinear equations).

Error estimate

The exact result $y(t_i)$ at step size h is:

$$y(t_i) = y_{i,h} + E_{i,h}, \quad (13.25)$$

where $y_{i,h}$ is the numerical approximation and E_h the error of the method. Since the global error is of second order, we can write the error as:

$$E_{i,h} = k h^2, \quad (13.26)$$

Similarly, for step size $2h$ we can write:

$$y(t_j) = y_{j,2h} + E_{j,2h}, \quad (13.27)$$

where $y_{j,2h}$ is the numerical approximation and E_{2h} the error of the method and:

$$E_{j,2h} = k (2h)^2 = 4kh^2 \quad (13.28)$$

Assuming that the constant k is the same at step h and step $2h$, we can determine the error estimate at the better approximation E_h . First we equate the exact result $y(t_i)$ at step h and the result $y(t_j)$ at step $2h$ (with $i = 2j$, $j = 1, 2, \dots$):

$$y_{i,h} + kh^2 = y_{j,2h} + 4kh^2 \quad (13.29)$$

it follows that:

$$3kh^2 = y_{2j,h} - y_{j,2h} \quad (13.30)$$

and then we compute the *error estimate*:

$$E_{j,h} = \frac{y_{2j,h} - y_{j,2h}}{3}. \quad (13.31)$$

13.3.1 Numerical example: vertical launch

Above we solved the vertical launch with the shooting method; let us now use the finite difference method. First the boundary value problem:

$$y''(t) = F_y/m - g, \quad y(0) = 0, \quad y(1) = 10. \quad (13.32)$$

is written using the central difference scheme ($F_y = -cy'$):

$$\frac{1}{h^2} (y_{i-1} - 2y_i + y_{i+1}) = -c \left(\frac{1}{2h} (-y_{i-1} + y_{i+1}) \right) / m - g. \quad (13.33)$$

(Here we have assumed that the air drag depends linearly on the velocity. Otherwise we would have a nonlinear boundary value problem, and consequently we would develop a system of nonlinear equations.)

We rearrange the above expression:

$$(2 - ch/m)y_{i-1} - 4y_i + (2 + ch/m)y_{i+1} = -2gh^2, \quad i = 1, 2, \dots, n-2. \quad (13.34)$$

The boundary conditions are:

$$y_0 = 0, \quad y_{n-1} = 10 \quad (13.35)$$

We have thus transformed the boundary value problem into a system of n linear equations. Let us now look at a concrete computation for $n = 11$; first let us define the constants, the time vector $\mathbf{t}$ and the step $\mathbf{h}$:

```
| n = 11 # odd number
| c = 0.5
| m = 1.0
| g = 9.81
| t = np.linspace(0, 1, n)
| h = t[1]
```

Then we continue with the computation of the tridiagonal coefficient matrix $\mathbf{A}$.

We use the function `numpy.diag()` ([documentation](#)):

```
| numpy.diag(v, k=0)
```

with the parameters:

- $\mathbf{v}$ the vector to be placed on the diagonal,
- $\mathbf{k}$ the diagonal to which $\mathbf{v}$ is assigned. $\mathbf{k}=0$ is used for the main diagonal, $\mathbf{k}<0$ and $\mathbf{k}>0$ are used for the diagonals below and above the main diagonal.

```
A = np.diag( -4*np.ones(n), 0) + \
      (2 -c*h/m)*np.diag(np.ones(n -1), -1) + \
      (2+c*h/m)*np.diag(np.ones(n -1), 1)
A[:4,:4]

array([[ -4.   ,  2.05,  0.   ,  0.   ],
       [ 1.95, -4.   ,  2.05,  0.   ],
       [ 0.   ,  1.95, -4.   ,  2.05],
       [ 0.   ,  0.   ,  1.95, -4.   ]])
```

Let us also define the vector of constants:

```
b = -2*g*h**2 * np.ones(n)
```

Now we *adjust* the coefficient matrix $\mathbf{A}$ and the vector of constants $\mathbf{b}$ to satisfy the boundary conditions:

$$y_0 = 0, \quad y_{n-1} = 10. \quad (13.36)$$

```
A[0,0] = 1
A[0,1] = 0
A[-1, -2] = 0
A[-1, -1] = 1
```

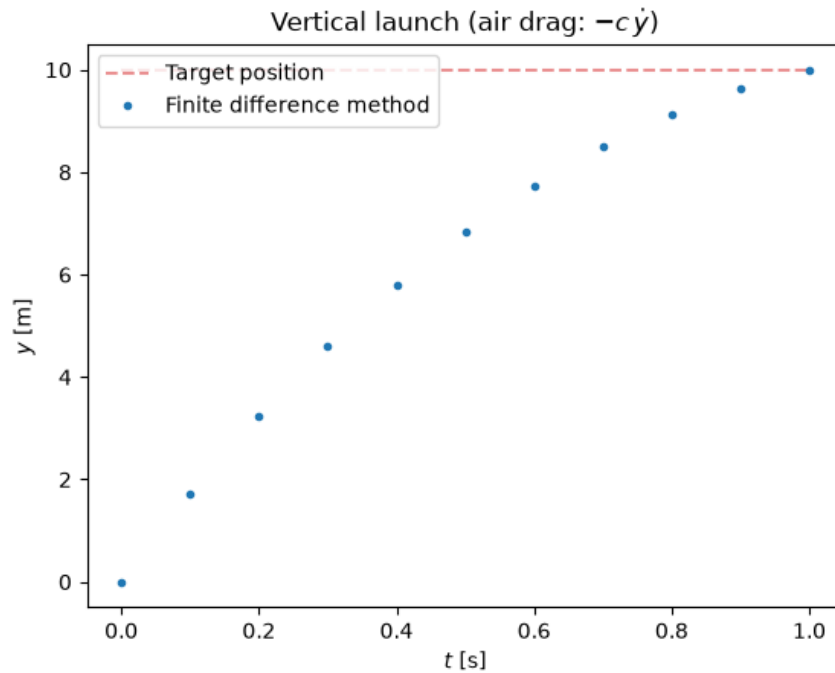
```
b[0] = 0.
b[-1] = 10.
```

Let us solve the system of linear equations:

```
y_fdm = np.linalg.solve(A, b)
```

We display the result:

```
plt.title(r'Vertical launch (air drag: $ -c\,\dot{y}$)')
plt.hlines(10, 0, 1, 'C3', label='Target position', linestyle='dashed', alpha=0.5)
plt.plot(t, y_fdm, '.', label='Finite difference method')
plt.xlabel('$t$ [s]')
plt.ylabel('$y$ [m]')
plt.legend()
plt.show()
```



Using the function `numpy.gradient()` we also compute the velocity and acceleration:

```
v_fdm = np.gradient(y_fdm,h, edge_order=2)
a_fdm = np.gradient(v_fdm,h, edge_order=2)
v_fdm[:4]

array([17.99109496, 16.20009044, 14.45276895, 12.79068266])
```

Let us now also compute the result with a double step:

```
n2h = n//2+1
t2h = np.linspace(0, 1, n2h)
h2h = t2h[1]
A2h = np.diag( -4*np.ones(n2h), 0) + \
      (2 * -c*h2h/(m))*np.diag(np.ones(n2h -1), -1) + \
      (2+c*h2h/(m))*np.diag(np.ones(n2h -1), 1)
b2h = -2*g*h2h**2 * np.ones(n2h)
A2h[0,0] = 1
A2h[0,1] = 0
A2h[-1, -1] = 1
A2h[-1, -2] = 0
b2h[0] = 0.
b2h[-1] = 10.
y2h_fdm = np.linalg.solve(A2h, b2h)
```

Let us compare the first six results at step $2h$:

```
y2h_fdm[:6]

array([ 0.          ,  3.24082594,  5.7992875 ,  7.74037178,  9.12287659,
        10.          ])
```

with every second result at step h :

```
y_fdm[:12:2]

array([ 0.          ,  3.24001809,  5.79815462,  7.73931207,  9.12221538,
        10.          ])
```

Now we can estimate the error:

```
| (y_fdm[:,2] - y2h_fdm)/3
|
| array([ 0.          , -0.00026928, -0.00037763, -0.00035324, -0.0002204  ,
|        0.          ])
```

13.3.2 Numerical example: loaded beam

Let us return to the boundary value problem of the beam with a half-sine load ($q(x) = -F_0 \sin(\pi x/l)$), which we already treated with the shooting method.

We write the fourth-order differential equation using the central difference scheme (for the i -th point):

$$-\frac{EI}{h^4} (1 w_{i-2} - 4 w_{i-1} + 6 w_i - 4 w_{i+1} + 1 w_{i+2}) - F_0 \sin(\pi x_i/l) = 0. \quad (13.37)$$

There are four boundary conditions, first the deflection at the ends:

$$w_0 = w_{n-1} = 0 \quad (13.38)$$

Since there is no moment at the ends, the second derivative is zero. Using the central difference scheme we therefore write the additional equations:

$$w_{-1} - 2 w_0 + w_1 = 0 \quad w_{n-2} - 2 w_{n-1} + w_n = 0. \quad (13.39)$$

If for the independent variable x we use n equidistant points, then we write the fourth-order differential equation for the $n - 2$ interior points. In doing so we obtain the two additional non-physical points w_{-1} and w_n . If we add the four boundary conditions as well, we have a solvable system of linear equations with $n + 2$ unknowns and $n + 2$ equations.

First let us prepare the data, the independent variable $\mathbf{x}$ and the step $\mathbf{h}$:

```
| l = 10.
| E = 2.1e11
| I = 2.1e -5
| F_0= 1e3
| n = 100
| x = np.linspace(0, l, n)
| h = x[1]
```

Then we prepare the coefficient matrix (the matrix has dimension (n, n) ; in the first and last two rows we could also write the values 0, which we will fix later):

```
| A = +1*np.diag(np.ones(n -2+2), -2) \
|     -4*np.diag(np.ones(n -1+2), -1) \
|     +6*np.diag(np.ones(n+2), 0) \
|     -4*np.diag(np.ones(n -1+2), +1) \
|     +1*np.diag(np.ones(n -2+2), +2)
| A[:4,:4]
|
| array([[ 6., -4.,  1.,  0.],
|        [-4.,  6., -4.,  1.],
|        [ 1., -4.,  6., -4.],
|        [ 0.,  1., -4.,  6.]])
```

Let us also define the vector of constants (we add one element at the end and one at the beginning):

```
| b = np.zeros(n+2)
| b[2: -2] = - h**4 * F_0*np.sin(np.pi*x[1: -1]/l)/(E*I)
| b[:4]
```

```
| array([ 0.00000000e+00,  0.00000000e+00, -7.48966545e -10, -1.49717894e -09])
```

Now we *adjust* the coefficient matrix **A** and the vector of constants **b** to satisfy the boundary conditions.

First $w_0 = w_{n-1} = 0$:

```
| A[0,:3]= np.array([0, 1, 0])
| b[0] = 0
| A[-1, -3:]= np.array([0, 1, 0])
| b[-1] = 0
```

Then $w_{-1} - 2w_0 + w_1 = 0$ and $w_{n-2} - 2w_{n-1} + w_n = 0$:

```
| A[1,:4] = np.array([1, -2, 1,0])
| b[1] = 0
| A[-2, -4:] = np.array([0, 1, -2, 1])
| b[-2] = 0
```

```
| A[:5,:5]
```

```
| array([[ 0.,  1.,  0.,  0.,  0.],
|        [ 1., -2.,  1.,  0.,  0.],
|        [ 1., -4.,  6., -4.,  1.],
|        [ 0.,  1., -4.,  6., -4.],
|        [ 0.,  0.,  1., -4.,  6.]])
```

```
| A[-5:,-5:]
```

```
| array([[ 6., -4.,  1.,  0.,  0.],
|        [-4.,  6., -4.,  1.,  0.],
|        [ 1., -4.,  6., -4.,  1.],
|        [ 0.,  0.,  1., -2.,  1.],
|        [ 0.,  0.,  0.,  1.,  0.]])
```

```
| b[:5]
```

```
| array([ 0.00000000e+00,  0.00000000e+00, -7.48966545e -10, -1.49717894e -09,
|        -2.24388381e -09])
```

```
| b[-5:]
```

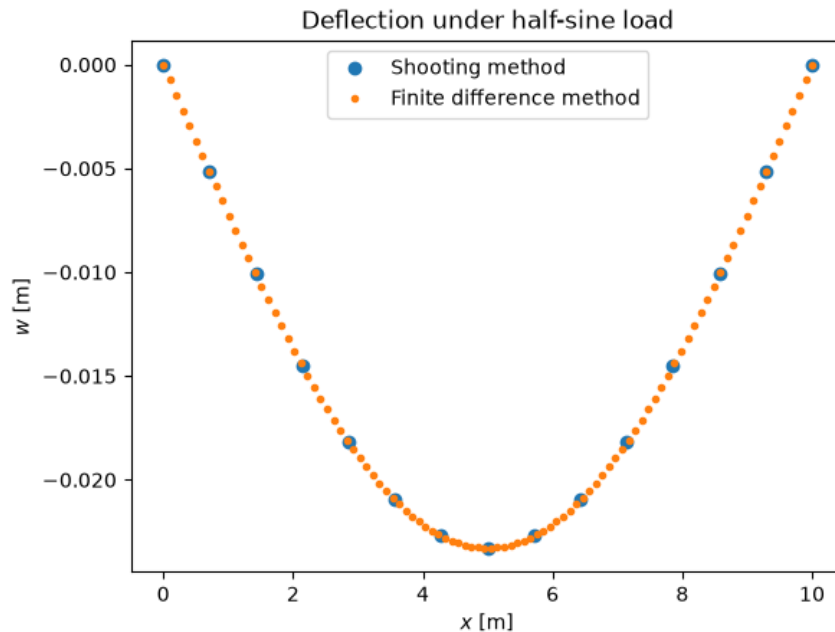
```
| array([-2.24388381e -09, -1.49717894e -09, -7.48966545e -10,  0.00000000e+00,
|        0.00000000e+00])
```

Let us solve the system of linear equations:

```
| y_fdm = np.linalg.solve(A, b)
```

We display the result:

```
| plt.title('Deflection under half -sine load')
| plt.plot(sol_half_sin.x, sol_half_sin.y[0], 'o', label='Shooting method')
| plt.plot(x, y_fdm[1: -1], '.', label='Finite difference method')
| plt.xlabel('$x$ [m]')
| plt.ylabel('$w$ [m]')
| plt.legend()
| plt.show()
```



13.4 Additional: symbolic solution of the beam

Here we will look at solving the boundary value problem symbolically. It should be emphasized that this is merely an example that we can carry out for the beam under consideration with the relatively simple half-sine load. **In practice, of course, the loads and also the beam shapes can be substantially more complex, and then there is almost no option other than numerical solving.**

First we import sympy:

```
import sympy as sym
sym.init_printing()
```

Let us define the variables:

```
w = sym.Function('w')
x, E, I, F_0, l = sym.symbols('x, E, I, F_0, l')
```

Let us define the differential equation (we add the boundary conditions later):

```
eq = sym.Eq( -E*I*w(x).diff(x,4) -F_0*sym.sin(sym.pi*x/l),0)
eq
```

$$-EI \frac{d^4}{dx^4} w(x) - F_0 \sin\left(\frac{\pi x}{l}\right) = 0 \quad (13.40)$$

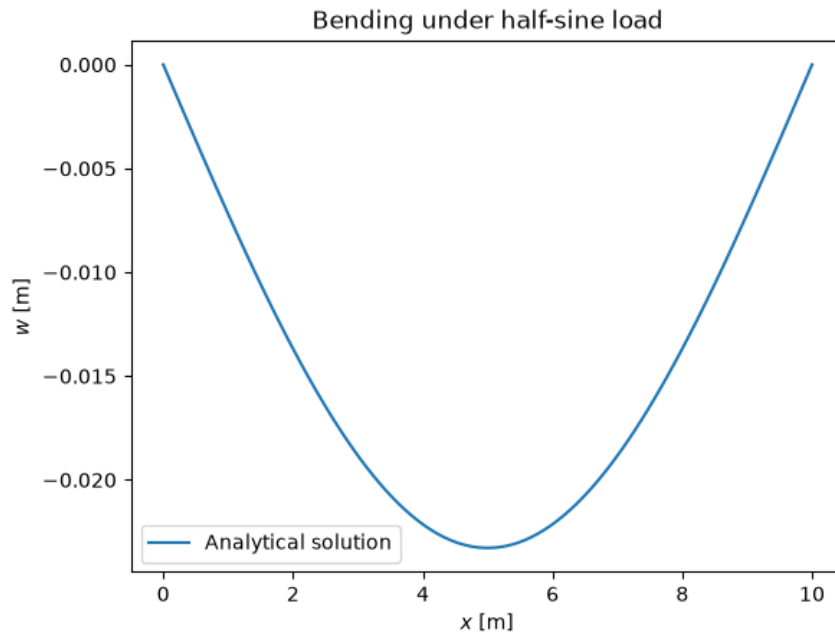
Let us solve it:

```
sol = sym.dsolve(eq, func=w(x),
hint='nth_linear_constant_coeff_undetermined_coefficients',
ics={
    w(0):0, #deflection at x=0 is zero
    w(x).diff(x,2).subs(x, 0): 0, #moment at x=0 is zero
    w(x).subs(x, l): 0, #deflection at x=l is
    zero
    w(x).diff(x, 2).subs(x, l): 0 #moment at x=l is zero
})
sol
```

$$w(x) = -\frac{F_0 l^4 \sin\left(\frac{\pi x}{l}\right)}{\pi^4 EI} \quad (13.41)$$

```
data = {E: 2.1e11, I: 2.1e-5, l: 10, F_0: 1e3}
solution_np = sym.lambdify(x, sol.rhs.subs(data), modules='numpy')
x_ana = np.linspace(0, data[l], 100)
y_ana = solution_np(x_ana)

plt.title('Bending under half -sine load')
plt.plot(x_ana, y_ana, ' -', label='Analytical solution')
plt.xlabel('$x$ [m]')
plt.ylabel('$w$ [m]')
plt.legend()
plt.show()
```



Let us now compare the analytical solution with the solution from the finite difference method and the shooting method:

```
[np.min(y_ana), np.min(y_fdm), np.min(sol_half_sin.y[0])]
```

$$[-0.0232759411601314, -0.0232798479988575, -0.0232786462140205] \quad (13.42)$$

Chapter 14

Testing code correctness, user interface

14.1 Testing code correctness

When developing larger codebases that involve several people, the need for automated testing of code correctness arises. This means that in addition to the code that produces a certain result, we also define test functions; the latter check whether the former return the expected result.

A simple example is:

```
## contents of the file test_first.py (folder modules)
def square(x):
    return x**2 + 1 # obvious bug relative to the function name

def test_square():
    assert square(2) == 4
```

If we run the function `test_square()`, the correctness of `square(2) == 4` is tested; if the result is `True`, nothing happens, but if the result is `False`, an `AssertionError()` is raised (see the [documentation](#) for the `assert` statement). Since the function `square()` returns `x**2+1`, the test will fail.

You are probably wondering what the point of such *correctness testing* is. As code grows large and many people work on it, it inevitably also becomes interconnected. This makes it possible for a developer to add new functionality and unintentionally break existing functionality. If all functionalities are tested, the testing will detect the inadequacy of the proposed change.

Most packages include such correctness testing; as an example, you can look at the source code of the *NumPy* package, which is hosted on [GitHub](#); if you look at the [contents](#) of the `numpy.linalg` submodule for linear algebra:

Branch: master ▾	numpy / numpy / linalg /
 eric-wieser MAINT: Remove unused isscalar import	
..	
 lapack_lite	DOC: Add changelog entry for new lapack_lite
 tests	TST: linalg: add basic smoketest for cholesky
 __init__.py	MAINT: Rearrange files in numpy/testing module.
 info.py	2to3: Apply `print` fixer.
 lapack_lite module.c	MAINT: Include the function name in all argument
 linalg.py	MAINT: Remove unused isscalar import
 setup.py	MAINT: Prefer to load functions from lapack/install

we notice the subfolder `tests`. The latter is entirely dedicated to testing and contains a multitude of Python files that verify the correctness of the submodule.

There are several options for testing code; the following two are commonly used:

- `unittest` (built into Python),
- `pytest`.

Both approaches have similar functionality. `pytest` is the most commonly used (e.g., NumPy/SciPy), and we will look at its basics here.

14.1.1 `pytest`

We install the `pytest` module with the `pip` command.

Some of its features:

- it returns a description of the test that failed,
- automatic discovery of test modules and functions,
- it can also run `unittest` tests.

If you run `pytest` in the command line, the program will automatically search the current folder and all subfolders for files of the form `test_*.py` or `*_test.py` (advanced discovery is described in the [documentation](#)).

If we run the command in the command line:

```
| pytest
```

we get a report like this:

```
===== test session starts =====
platform win32 -- Python 3.6.2, pytest -3.2.1, py -1.4.34, pluggy -0.4.0
rootdir: c:\pypinm\modules, inifile:
collected 4 items

test_tools.py ...
test_first.py F

===== FAILURES =====
----- test_square -----

    def test_square():
>         assert square(2) == 4
E         assert 5 == 4
E         + where 5 = square(2)

test_first.py:5: AssertionError
===== 1 failed, 3 passed in 0.65 seconds =====
```

From the report we see that the program found two files (`test_tools.py` and `test_first.py`) and that an error occurred in one function (`test_square`), while three functions successfully passed the test.

For basic use in numerical computations, we should also highlight the `numpy.testing` module ([documentation](#)), which provides support for testing numerical arrays.

Selected functions are:

- `assert_allclose(actual, desired[, rtol, ...])` ([documentation](#)) checks equality to a required accuracy,
- `assert_array_less(x, y[, err_msg, verbose])` ([documentation](#)) checks whether the elements of `x` are less than the elements of `y`,
- `assert_string_equal(actual, desired)` ([documentation](#)) checks string equality.

Example: test_tools.py:

In the file `test_tools.py`, functions for testing the `tools.py` module are prepared.

First the data is prepared:

```
initial = np.asarray([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])
swapped_cols_0_1 = np.asarray([[2, 1, 3],
                              [5, 4, 6],
                              [8, 7, 9]])
```

Then a function is defined that calls the function `tools.swap_columns()` and compares the result to the expected `swapped_cols_0_1`:

```
def test_column():
    a = initial.copy() # make a copy of the data
    b = tools.swap_columns(a, 0, 1) # b (and also a) have their columns swapped
    np.testing.assert_allclose(b, swapped_cols_0_1)
```

If the function `tools.swap_columns()` works correctly, calling `pytest` in the command line completes successfully. For the remaining tests, see the file `test_tools.py` and the [documentation](#) of the `pytest` package.

14.2 Graphical user interface

A user interface is used for the user's interaction with the program code. User interfaces can be command-line based or graphical. For simple command-line user interfaces, it is best to use [argparse](#); otherwise, here we will focus primarily on the **graphical user interface**.

14.2.1 Graphical user interface inside the browser

Recently, a lot of effort has gone into supporting the development of graphical interfaces within the *Jupyter notebook* or *Jupyter lab* environment. The main advantage of this approach is that upgrading it into a web application is relatively straightforward; see, e.g.: [Dashboarding with Jupyter Notebooks, Voila and Widgets | SciPy 2019 | M. Breddels and M. Renou](#).

14.2.2 Graphical user interface within the operating system

In this section we will look at classic user interfaces that we run within a particular operating system. For programming a graphical user interface, too, there are many different approaches/modules. A few of the most commonly used:

- [PySide6](#) is the official Python binding (*Qt for Python*) for [Qt](#), which is the most widespread platform for user interfaces; its [LGPL](#) license also permits commercial use,
- [PyQt6](#) is an older, independent Qt binding with a practically identical interface (only the import differs), but with a GPL license (commercial use requires a paid license),
- [Kivy](#) for rapid development of modern user interfaces (not as mature as, e.g., Qt),
- [wxWidgets](#) a widely used free platform for building user interfaces.

Among all those listed, we will take a closer look at *PySide6*, which is the official and today the most mature choice. We install it with the command `pip install PySide6`. (If you want to use the code below with *PyQt6*, just replace `PySide6` with `PyQt6` in the imports.)

It is worth mentioning that a user interface can be:

- *drawn* with [QtDesigner](#) (see the [video](#)), or
- *coded*.

Here we will *code* the user interface.

14.2.3 Example

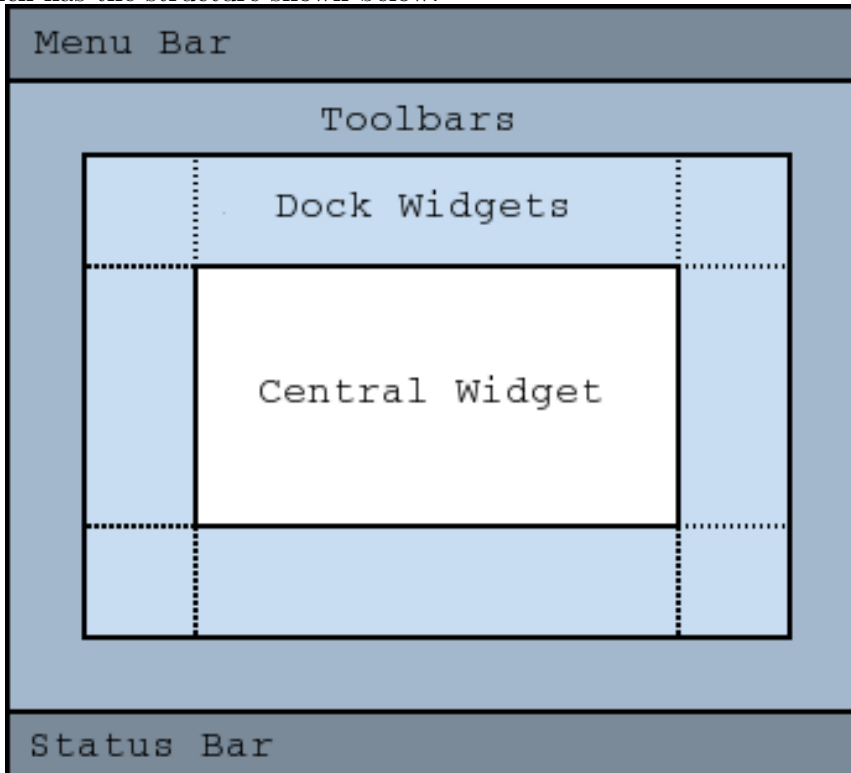
First we import the package for interaction and for graphical objects:

```
from PySide6 import QtCore      # for interaction
from PySide6 import QtWidgets  # graphical objects
```

We will also need the `sys` module for running the program:

```
import sys
```

We typically build a user interface on the class `QtWidgets.QMainWindow` ([documentation](#)), which has the structure shown below:



The essential graphical elements used in such a user interface are:

- *Menu Bar*,
- *Toolbars*,
- *Dock Widgets*,
- *Central Widget*,
- *Status bar*.

It is not necessary to define all of them; below we will look at an example in which we define the *Status bar*, *Central Widget*, and *Menu Bar*. We usually define all elements when initializing an instance of the class (the `__init__` method).

Here we point out that the graphical interface is based on so-called *Widgets* (e.g., for a button, a table, a date, etc.). A demonstration of some of the options is shown in:

- [the Qt6 gallery](#).

A very simple user interface (with inline code comments) is shown below.

```
import sys
from PySide6 import QtCore
from PySide6 import QtGui
from PySide6 import QtWidgets

class MainWindow(QtWidgets.QMainWindow):
```

```

""" The main window inherits from `QtWidgets.QMainWindow`
"""

def __init__(self):
    """ Constructor of the MainWindow object
    """
    QtWidgets.QMainWindow.__init__(self) # parent constructor
    self.setWindowTitle('Program title window')

    # status bar
    self.my_status_bar = QtWidgets.QStatusBar() # new widget
    self.my_status_bar.showMessage('This text disappears after 10s', 10000)
    self.setStatusBar(self.my_status_bar) # here self.my_status_bar is assigned
    to the one prepared in QMainWindow

    # central widget
    self.button1 = QtWidgets.QPushButton('Button 1') # Button labeled 'Button 1'
    self.button1.pressed.connect(self.action_on_button1_press) # connect the
    press to the action function
    self.setCentralWidget(self.button1) # add the button to the `CentralWidget`

    # menus
    self.my_exit_action = QtGui.QAction('&Exit',
                                       self, shortcut=QtGui.QKeySequence.Stand_
                                       ardKey.Close,
                                       statusTip="Exit the program",
                                       triggered=self.close)

    self.my_file_menu = self.menuBar().addMenu('&File') # menu 'File'
    self.my_file_menu.addAction(self.my_exit_action) # link to the action
    `self.my_exit_action` defined above

def action_on_button1_press(self):
    QtWidgets.QMessageBox.about(self,
                                'Title :)',
                                'This is triggered when the button is pressed.')

```

We then run the program inside a `try` statement:

```

try:
    app = QtWidgets.QApplication(sys.argv)
    mainWindow = MainWindow()
    mainWindow.show()
    app.exec()
    sys.exit(0)
except SystemExit:
    print('Closing window.')
except:
    print(sys.exc_info())

```

For a more advanced example, see the files:

- [simple_user_interface.py](#),
- [user_interface.py](#)

A few comments on `user_interface.py`:

1. Look at the override of the `mouseDoubleClickEvent` event and override the inherited `keyPressEvent` event so that pressing any key closes the program (if you are in a `TextEdit` field, then pressing a key will of course print the value of that key).
2. Add another *Widget* from the [list](#).
3. Modify the program so that the sine function is always plotted, and in the `function_text` input field you will enter the number of discrete points (currently there are 100 points). Connect the field to the appropriate functions.

4. Set up error handling for the change above.

14.3 A few review questions!

1. In VisualStudioCode, create a module that has two functions:
 - for multiplying a matrix and a vector,
 - for multiplying two matrices.
2. For the module above, create a script for testing (use `numpy.testing`).
3. In `simple_user_interface.py`, in the initialization method `__init__`, comment out all calls to the `self.init ...` methods except for the method `self.init_status_bar()`. Run the program in normal mode. Set a *break* point on `self.setGeometry(...)` and run the program in *debug* mode.
4. Continue from the previous point and find the shortcut for stepping through lines:
 - stepping over a line,
 - stepping into a line.

Step into `init_status_bar(self)` and stop at the line `self.setStatusBar(self.status_bar)`. Open the console and, via the command line, change the value of `self.status_bar.showMessage()`.

5. Uncomment the lines commented out earlier (above). Add a third button that closes the program.
6. Add another *Widget* from the [list](#).

Chapter 15

References

1. Slavič J.: Programiranje in numerične metode s Pythonom, [executable book](#) or [online book](#), 2015-
2. Petrišič J.: Uvod v Matlab za inženirje, 2011
3. Demšar J. Python za programerje, 2012 (see the [online classroom at FRI](#))
4. Nicolas P. Rougier: From Python to Numpy, 2017 [online book](#)
5. Kiusalaas J: Numerical Methods in Engineering with Python 3, 2013
6. Fangohr H. Python for Computational Science and Engineering, 2014 ([available online](#))
7. Bucky R. Python Programming Tutorials ([videos on YouTube](#))
8. [Projekt TOMO](#) (more content; see especially the courses at FMF and FRI)
9. [Scientific Python Lectures](#) (formerly *Scipy Lecture Notes*)
10. Using the Numpy, Scipy, etc. packages for engineers [Mr. P Solver \(YouTube\)](#)