Introduction to Computer-based Physical Modeling

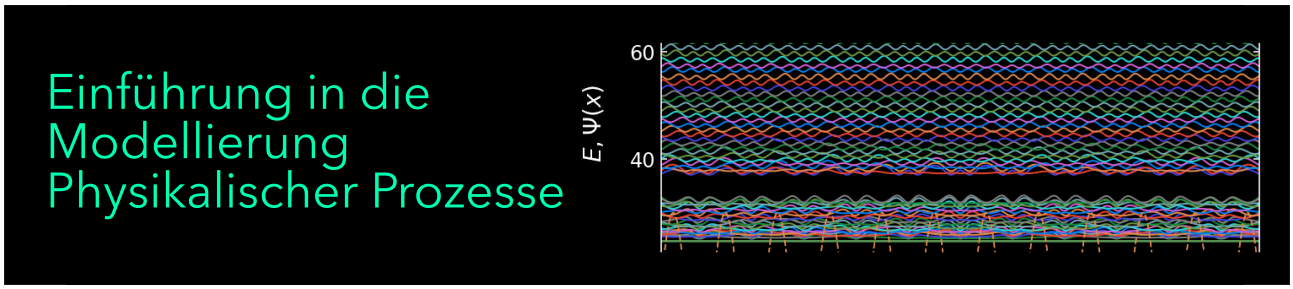
Frank Cichos

2024-08-14

Table of contents

Chapter 1

EMPP 2024



Chapter 2

Willkommen zum Kurs Einführung in die Modellierung Physikalischer Prozesse!

Die Programmiersprache Python ist für alle Arten von wissenschaftlichen und technischen Aufgaben nützlich. Sie können mit ihr Daten analysieren und darstellen. Sie können mit ihr auch wissenschaftliche Probleme numerisch lösen, die analytisch nur schwer oder gar nicht zu lösen sind. Python ist frei verfügbar und wurde aufgrund seines modularen Aufbaus um eine nahezu unendliche Anzahl von Modulen für verschiedene Zwecke erweitert.

Dieser Kurs soll Sie in die Programmierung mit Python einführen. Er richtet sich eher an den Anfänger, wir hoffen aber, dass er auch für Fortgeschrittene interessant ist. Wir beginnen den Kurs mit einer Einführung in die Jupyter Notebook-Umgebung, die wir während des gesamten Kurses verwenden werden. Danach werden wir eine Einführung in Python geben und Ihnen einige grundlegende Funktionen zeigen, wie z.B. das Plotten und Analysieren von Daten durch Kurvenanpassung, das Lesen und Schreiben von Dateien, was einige der Aufgaben sind, die Ihnen während Ihres Physikstudiums begegnen werden. Wir zeigen Ihnen auch einige fortgeschrittene Themen wie die Animation in Jupyter und die Simulation von physikalischen Prozessen in

- Mechanik
- Elektrostatik
- Wellen
- Optik

Falls am Ende des Kurses Zeit bleibt, werden wir auch einen Blick auf Verfahren des maschinellen Lernens werfen, das mittlerweile auch in der Physik zu einem wichtigen Werkzeug geworden ist.

Wir werden keine umfassende Liste von numerischen Simulationsschemata präsentieren, sondern die Beispiele nutzen, um Ihre Neugierde zu wecken. Da es leichte Unterschiede in der Syntax der verschiedenen Python-Versionen gibt, werden wir uns im Folgenden immer auf den Python 3-Standard beziehen.

Der Kurs wird auf Deutsch gehalten werden. Die Webseiten, die Sie für den Überblick zu Python zur Verfügung gestellt bekommen, werden allerdings auf Englisch sein. Übungsaufgaben werden werden auf Deutsch gestellt.

Part I

Lecture 1

Chapter 3

Jupyter Notebooks

3.1 What is Jupyter Notebook?

A Jupyter Notebook is a web browser based **interactive computing environment** that enables users to create documents that include code to be executed, results from the executed code such as plots and images,and finally also an additional documentation in form of markdown text including equations in LaTeX.

These documents provide a **complete and self-contained record of a computation** that can be converted to various formats and shared with others using email, version control systems (like git/GitHub) or nbviewer.jupyter.org.

3.1.1 Key Components of a Notebook

The Jupyter Notebook ecosystem consists of three main components:

1. Notebook Editor
2. Kernels
3. Notebook Documents

Let's explore each of these components in detail:

Notebook Editor

The Notebook editor is an interactive web-based application for creating and editing notebook documents. It enables users to write and run code, add rich text, and multimedia content. When running Jupyter on a server, users typically use either the classic Jupyter Notebook interface or JupyterLab, an advanced version with more features.

Key features of the Notebook editor include:

- **Code Editing:** Write and edit code in individual cells.
- **Code Execution:** Run code cells in any order and display computation results in various formats (HTML, LaTeX, PNG, SVG, PDF).
- **Interactive Widgets:** Create and use JavaScript widgets that connect user interface controls to kernel-side computations.
- **Rich Text:** Add documentation using Markdown markup language, including LaTeX equations.

> **Advance Notebook Editor Info**
>
> The Notebook editor in Jupyter offers several advanced features:
> - **Cell Metadata:** Each cell has associated metadata that can be used to control its behavior. This includes tags for slideshows, hiding code cells, and controlling cell execution.

- **Magic Commands:** Special commands prefixed with `%` (line magics) or `%%` (cell magics) that provide additional functionality, such as timing code execution or displaying plots inline.
- **Auto-completion:** The editor provides context-aware auto-completion for Python code, helping users write code more efficiently.
- **Code Folding:** Users can collapse long code blocks for better readability.
- **Multiple Cursors:** Advanced editing with multiple cursors for simultaneous editing at different locations.
- **Split View:** The ability to split the notebook view, allowing users to work on different parts of the notebook simultaneously.
- **Variable Inspector:** A tool to inspect and manage variables in the kernel's memory.
- **Integrated Debugger:** Some Jupyter environments offer an integrated debugger for step-by-step code execution and inspection.

3.1.2 Kernels

Kernels are the computational engines that execute the code contained in a notebook. They are separate processes that run independently of the notebook editor.

Key responsibilities of kernels include: * Executing user code * Returning computation results to the notebook editor * Handling computations for interactive widgets * Providing features like tab completion and introspection

i Advanced Kernel Info

Jupyter notebooks are language-agnostic. Different kernels can be installed to support various programming languages such as Python, R, Julia, and many others. The default kernel runs Python code, but users can select different kernels for each notebook via the Kernel menu.
Kernels communicate with the notebook editor using a JSON-based protocol over ZeroMQ/WebSockets. For more technical details, see the messaging specification.
Each kernel runs in its own environment, which can be customized to include specific libraries and dependencies. This allows users to create isolated environments for different projects, ensuring that dependencies do not conflict.
Kernels also support interactive features such as:

- **Tab Completion:** Provides suggestions for variable names, functions, and methods as you type, improving coding efficiency.
- **Introspection:** Allows users to inspect objects, view documentation, and understand the structure of code elements.
- **Rich Output:** Supports various output formats, including text, images, videos, and interactive widgets, enhancing the interactivity of notebooks.

Advanced users can create custom kernels to support additional languages or specialized computing environments. This involves writing a kernel specification and implementing the necessary communication protocols.
For managing kernels, Jupyter provides several commands and options:

- **Starting a Kernel:** Automatically starts when a notebook is opened.
- **Interrupting a Kernel:** Stops the execution of the current code cell, useful for halting long-running computations.
- **Restarting a Kernel:** Clears the kernel's memory and restarts it, useful for resetting the environment or recovering from errors.
- **Shutting Down a Kernel:** Stops the kernel and frees up system resources.

Users can also monitor kernel activity and resource usage through the Jupyter interface, ensuring efficient and effective use of computational resources.

3.1.3 JupyterLab Example

The following is an example of a JupyterLab interface with a notebook editor, code cells, markdown cells, and a kernel selector:

3.1.4 Notebook Documents

Notebook documents are self-contained files that encapsulate all content created in the notebook editor. They include code inputs/outputs, Markdown text, equations, images, and other media. Each document is associated with a specific kernel and serves as both a human-readable record of analysis and an executable script to reproduce the work.

Characteristics of notebook documents:

- File Extension: Notebooks are stored as files with a `.ipynb` extension.
- Structure: Notebooks consist of a linear sequence of cells, which can be one of three types:
  - **Code cells:** Contain executable code and its output.
  - **Markdown cells:** Contain formatted text, including LaTeX equations.
  - **Raw cells:** Contain unformatted text, preserved when converting notebooks to other formats.

Advanced Notebook Documents Info

- Version Control: Notebook documents can be version controlled using systems like Git. This allows users to track changes, collaborate with others, and revert to previous versions if needed. Tools like `nbdime` provide diff and merge capabilities specifically designed for Jupyter Notebooks.
- Cell Tags: Cells in a notebook can be tagged with metadata to control their behavior during execution, export, or presentation. For example, tags can be used to hide input or output, skip execution, or designate cells as slides in a presentation.
- Interactive Widgets: Notebook documents can include interactive widgets that allow users to manipulate parameters and visualize changes in real-time. This is particularly useful for data exploration and interactive simulations.
- Extensions: The Jupyter ecosystem supports a wide range of extensions that enhance the functionality of notebook documents. These extensions can add features like spell checking, code formatting, and integration with external tools and services.
- Security: Notebook documents can include code that executes on the user's machine, which poses security risks. Jupyter provides mechanisms to sanitize notebooks and prevent the execution of untrusted code. Users should be cautious when opening notebooks from unknown sources.
- Collaboration: Jupyter Notebooks can be shared and collaboratively edited in real-time using platforms like Google Colab, Microsoft Azure Notebooks, and JupyterHub. These platforms provide cloud-based environments where multiple users can work on the same notebook simultaneously.
- Customization: Users can customize the appearance and behavior of notebook documents using CSS and JavaScript. This allows for the creation of tailored interfaces and enhanced user experiences.
- Export Options: In addition to static formats, notebooks can be exported to interactive formats like dashboards and web applications. Tools like `Voila` convert notebooks into standalone web applications that can be shared and deployed.
- Provenance: Notebooks can include provenance information that tracks the origin and history of data and computations. This is important for reproducibility and transparency in scientific research.
- Documentation: Notebook documents can serve as comprehensive documentation for projects, combining code, results, and narrative text. This makes them valuable for teaching, tutorials, and sharing research findings.
- Performance: Large notebooks with many cells and outputs can become slow and unwieldy. Techniques like cell output clearing, using lightweight data formats, and splitting notebooks into smaller parts can help maintain performance.
- Integration: Jupyter Notebooks can integrate with a wide range of data sources, libraries, and tools. This includes databases, cloud storage, machine learning frameworks, and visualization libraries, making them a versatile tool for data science and research.
- Internal Format: Notebook files are JSON text files with binary data encoded in base64, making them easy to manipulate programmatically.
- Exportability: Notebooks can be exported to various static formats (HTML, reStructuredText, LaTeX, PDF, slide shows) using Jupyter's `nbconvert` utility.
- Sharing: Notebooks can be shared via nbviewer, which renders notebooks from public URLs or

GitHub as static web pages, allowing others to view the content without installing Jupyter.

This integrated system of editor, kernels, and documents makes Jupyter Notebooks a powerful tool for interactive computing, data analysis, and sharing of computational narratives.

3.2 Using the Notebook Editor

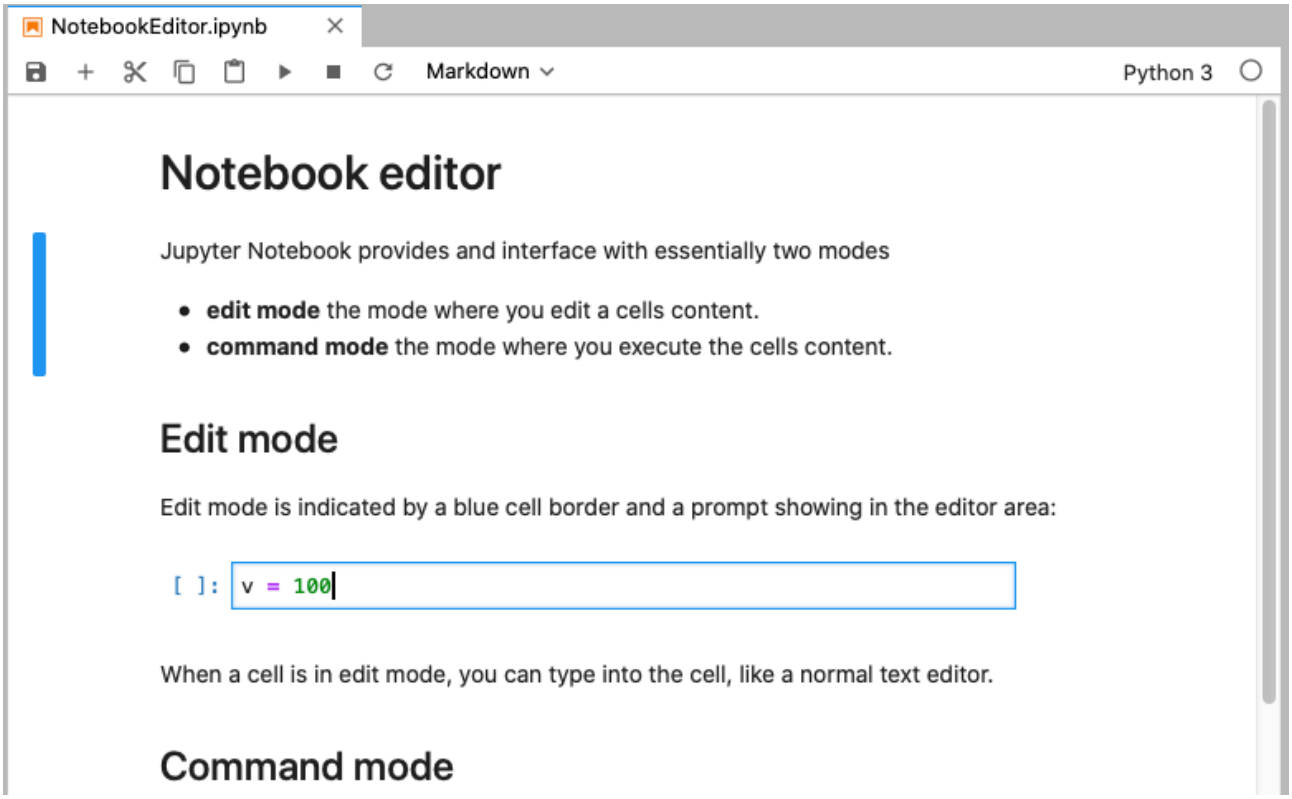


Figure 3.1: Jupyter Notebook Editor

The Jupyter Notebook editor provides an interactive environment for writing code, creating visualizations, and documenting computational workflows. It consists of a web-based interface that allows users to create and edit notebook documents containing code, text, equations, images, and interactive elements. A Jupyter Notebook provides an interface with essentially two modes of operation:

- **edit mode** the mode where you edit a cells content.
- **command mode** the mode where you execute the cells content.

In the more advanced version of JupyterLab you can also have a **presentation mode** where you can present your notebook as a slideshow.

3.2.1 Edit mode

Edit mode is indicated by a blue cell border and a prompt showing in the editor area when a cell is selected. You can enter edit mode by pressing `Enter` or using the mouse to click on a cell's editor area.

Figure 3.2: Edit Mode

When a cell is in edit mode, you can type into the cell, like a normal text editor

3.2.2 Command mode

Command mode is indicated by a grey cell border with a blue left margin. When you are in command mode, you are able to edit the notebook as a whole, but not type into individual cells. Most importantly, in command mode, the keyboard is mapped to a set of shortcuts that let you perform notebook and cell actions efficiently.

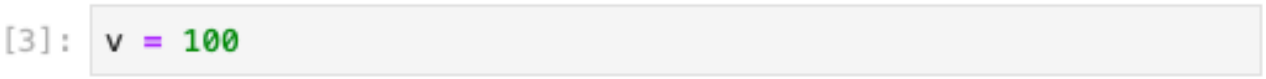


Figure 3.3: Command Mode

If you have a hardware keyboard connected to your iOS device, you can use Jupyter keyboard shortcuts. The modal user interface of the Jupyter Notebook has been optimized for efficient keyboard usage. This is made possible by having two different sets of keyboard shortcuts: one set that is active in edit mode and another in command mode.

3.2.3 Keyboard navigation

In edit mode, most of the keyboard is dedicated to typing into the cell's editor area. Thus, in edit mode there are relatively few shortcuts available. In command mode, the entire keyboard is available for shortcuts, so there are many more. Most important ones are:

1. Switch command and edit mods: `Enter` for edit mode, and `Esc` or `Control` for command mode.
2. Basic navigation: `↑`/`k`, `↓`/`j`
3. Run or render currently selected cell: `Shift`+`Enter` or `Control`+`Enter`
4. Saving the notebook: `s`
5. Change Cell types: `y` to make it a **code** cell, `m` for **markdown** and `r` for **raw**
6. Inserting new cells: `a` to **insert above**, `b` to **insert below**
7. Manipulating cells using pasteboard: `x` for **cut**, `c` for **copy**, `v` for **paste**, `d` for **delete** and `z` for **undo delete**
8. Kernel operations: `i` to **interrupt** and `0` to **restart**

3.2.4 Running code

Code cells allow you to enter and run code. Run a code cell by pressing the button in the bottom-right panel, or `Control`+`Enter` on your hardware keyboard.

```
23752636
```

There are a couple of keyboard shortcuts for running code:

- `Control`+`Enter` run the current cell and enters command mode.
- `Shift`+`Enter` runs the current cell and moves selection to the one below.
- `Option`+`Enter` runs the current cell and inserts a new one below.

3.3 Managing the kernel

Code is run in a separate process called the **kernel**, which can be interrupted or restarted. You can see kernel indicator in the top-right corner reporting current kernel state: means kernel is **ready** to execute code, and means kernel is currently **busy**. Tapping kernel indicator will open **kernel menu**, where you can reconnect, interrupt or restart kernel.

Try running the following cell — kernel indicator will switch from to , i.e. reporting kernel as "busy". This means that you won't be able to run any new cells until current execution finishes, or until kernel is interrupted. You can then go to kernel menu by tapping the kernel indicator and select "Interrupt".

3.4 Markdown in Notebooks

Text can be added to Jupyter Notebooks using Markdown cells. This is extremely useful providing a complete documentation of your calculations or simulations. In this way, everything really becomes an notebook. You can change the cell type to Markdown by using the "Cell Actions" menu, or with a hardware keyboard shortcut `m`. Markdown is a popular markup language that is a superset of HTML. Its specification can be found here:

https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet

Markdown cells can either be **rendered** or **unrendered**.

When they are rendered, you will see a nice formatted representation of the cell's contents.

When they are unrendered, you will see the raw text source of the cell. To render the selected cell, click the button or `shift`+ `enter`. To unrender, select the markdown cell, and press `enter` or just double click.

3.4.1 Markdown basics

Below are some basic markdown examples, in its rendered form. If you which to access how to create specific appearances, double click the individual cells to put the into an unrendered edit mode.

You can make text *italic* or **bold**. You can build nested itemized or enumerated lists:

- First item
  - First subitem
    - First sub-subitem
  - Second subitem
    - First subitem of second subitem
    - Second subitem of second subitem
- Second item
  - First subitem
- Third item
  - First subitem

Now another list:

1. Here we go
   1. Sublist
      2. Sublist
2. There we go
3. Now this

Here is a blockquote:

> Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren't special enough to break the rules. Namespaces are one honking great idea – let's do more of those!

And Web links:

Jupyter's website

3.4.2 Headings

You can add headings by starting a line with one (or multiple) `#` followed by a space and the title of your section. The number of `#` you use will determine the size of the heading

```
# Heading 1
# Heading 2
## Heading 2.1
## Heading 2.2
### Heading 2.2.1
```

3.4.3 Embedded code

You can embed code meant for illustration instead of execution in Python:

```
def f(x):
    """a docstring"""
    return x**2
```

3.4.4 LaTeX equations

Courtesy of MathJax, you can include mathematical expressions both inline: $e^{i\pi} + 1 = 0$ and displayed:

$$e^x = \sum_{i=0}^{\infty} \frac{1}{i!}x^i$$

Inline expressions can be added by surrounding the latex code with `$`:

```
$e^{i\pi} + 1 = 0$
```

Expressions on their own line are surrounded by `$$`:

```
$$e^x=\sum_{i=0}^\infty \frac{1}{i!}x^i$$
```

3.4.5 Images

Images may be also directly integrated into a Markdown block.

To include images use

```
![alternative text](url)
```

for example

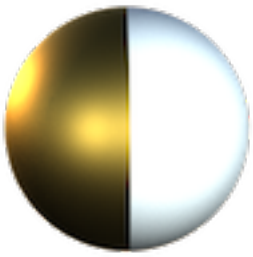


Figure 3.4: alternative text

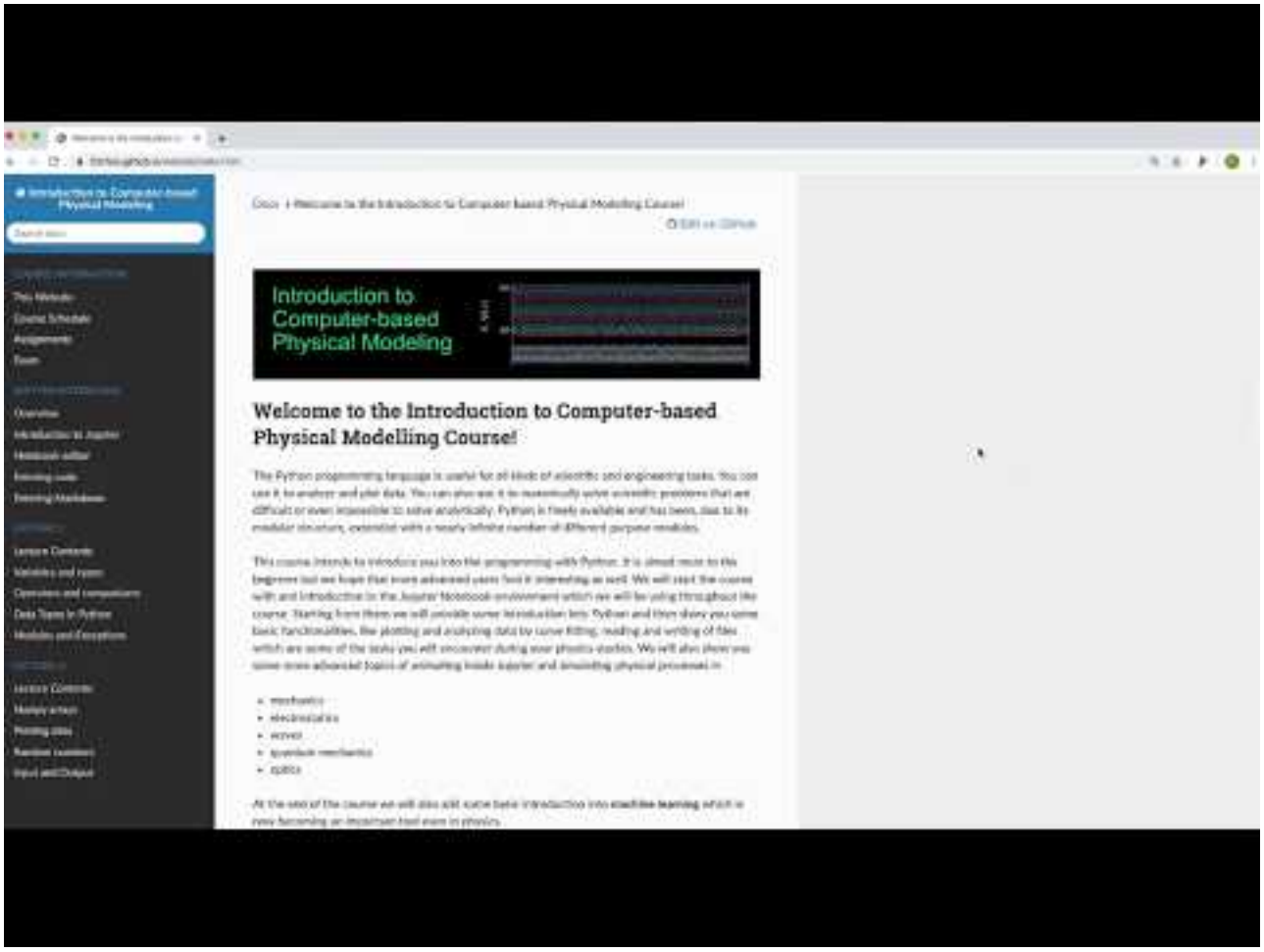
3.4.6 Videos

To include videos, we use HTML code like

```
<video src="mov/movie.mp4" width="320" height="200" controls preload></video>
```

in the Markdown cell. This works with videos stored locally.

You can embed YouTube Videos as well by using the `IPython` module.


Introduction to
Computer-based
Physical Modeling
Welcome to the Introduction to Computer-based Physical Modelling Course!

Chapter 4

Variables & Numbers

4.1 Variables in Python

4.1.1 Symbol Names

Variable names in Python can include alphanumerical characters `a-z`, `A-Z`, `0-9`, and the special character `_`. Normal variable names must start with a letter or an underscore. By convention, variable names typically start with a lower-case letter, while Class names start with a capital letter and internal variables start with an underscore.

> ⚠ Reserved Keywords
>
> There are a number of Python keywords that cannot be used as variable names because Python uses them for other things. These keywords are:
> `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`, `with`, `yield`
> Be aware of the keyword `lambda`, which could easily be a natural variable name in a scientific program. However, as a reserved keyword, it cannot be used as a variable name.

4.1.2 Variable Assignment

The assignment operator in Python is `=`. Python is a dynamically typed language, so we do not need to specify the type of a variable when we create one.

Assigning a value to a new variable creates the variable:

```
#| autorun: false
# variable assignments
x = 1.0
my_favorite_variable = 12.2
x
```

Although not explicitly specified, a variable does have a type associated with it (e.g., integer, float, string). The type is derived from the value that was assigned to it. To determine the type of a variable, we can use the `type` function.

```
#| autorun: false
type(x)
```

If we assign a new value to a variable, its type can change.

```
#| autorun: false
x = 1
```

```
#| autorun: false
type(x)
```

If we try to use a variable that has not yet been defined, we get a `NameError` error.

```
#| autorun: false
#print(g)
```

4.2 Number Types

Python supports various number types, including integers, floating-point numbers, and complex numbers. These are some of the basic building blocks of doing arithmetic in any programming language. We will discuss each of these types in more detail.

4.2.1 Comparison of Number Types

Type	Example	Description	Limits	Use Cases
int	42	Whole numbers	Unlimited precision (bounded by available memory)	Counting, indexing
float	3.14159	Decimal numbers	Typically ±1.8e308 with 15-17 digits of precision (64-bit)	Scientific calculations, prices
complex	2 + 3j	Numbers with real and imaginary parts	Same as float for both real and imaginary parts	Signal processing, electrical engineering
bool	True / False	Logical values	Only two values: True (1) and False (0)	Conditional operations, flags

Examples for Number Types

4.2.2 Integers

Integer Representation: Integers are whole numbers without a decimal point.

```
#| autorun: false
x = 1
type(x)
```

Binary, Octal, and Hexadecimal: Integers can be represented in different bases:

```
#| autorun: false
0b1010111110  # Binary
0x0F          # Hexadecimal
```

4.2.3 Floating Point Numbers

Floating Point Representation: Numbers with a decimal point are treated as floating-point values.

```
#| autorun: false
x = 3.141
type(x)
```

Maximum Float Value: Python handles large floats, converting them to infinity if they exceed the maximum representable value.

```
#| autorun: false
1.7976931348623157e+308 * 2  # Output: inf
```

4.2.4 Complex Numbers

Complex Number Representation: Complex numbers have a real and an imaginary part.

```
#| autorun: false
c = 2 + 4j
type(c)
```

- **Accessors for Complex Numbers:**
  - `c.real`: Real part of the complex number.
  - `c.imag`: Imaginary part of the complex number.

```
#| autorun: false
print(c.real)
print(c.imag)
```

Complex Conjugate: Use the `.conjugate()` method to get the complex conjugate.

```
#| autorun: false
c = c.conjugate()
print(c)
```

Part II

Lecture 2

Chapter 5

Data Types in Python

It's time to look at different data types we may find useful in our course. Besides the number types mentioned previously, there are also other types like **strings**, **lists**, **tuples**, **dictionaries** and **sets**.

Data Type	Classes	Description
Numeric	`int`, `float`, `complex`	Holds numeric values. `int` for integers, `float` for decimal numbers, `complex` for complex numbers.
String	`str`	Stores sequences of characters. Immutable.
Sequence	`list`, `tuple`, `range`	Stores ordered collections of items. `list` is mutable, `tuple` is immutable, `range` represents a sequence of numbers.
Mapping	`dict`	Stores data as key-value pairs. Mutable and unordered.
Boolean	`bool`	Holds either `True` or `False`. Used for logical operations.
Set	`set`, `frozenset`	Holds collections of unique items. `set` is mutable, `frozenset` is immutable.
None	`NoneType`	Represents the absence of a value or a null value.
Binary	`bytes`, `bytearray`, `memoryview`	Used for handling binary data.

Each of these data types has a number of connected `methods` (functions) which allow to manipulate the data contained in a variable. If you want to know which methods are available for a certain object use the command `dir`, e.g.

```
s="string"
dir(s)
```

The following few cells will give you a short introduction into each type.

5.1 Strings

Strings are lists of keyboard characters as well as other characters not on your keyboard. They are useful for printing results on the screen, during reading and writing of data.

```
#| autorun: false
s="Hello" # string variable
type(s)
```

```
#| autorun: false
t="world!"
```

String can be concatenated using the + operator.

```
#| autorun: false
c=s+' '+t
```

```
#| autorun: false
print(c)
```

As strings are lists, each character in a string can be accessed by addressing the position in the string (see Lists section)

```
#| autorun: false
c[1]
```

Strings can also be made out of numbers.

```
#| autorun: false
"975"+"321"
```

If you want to obtain a number of a string, you can use what is known as type casting. Using type casting you may convert the string or any other data type into a different type if this is possible. To find out if a string is a pure number you may use the `str.isnumeric` method. For the above string, we may want to do a conversion to the type *int* by typing:

```
#| autorun: false
("975"+"321").isnumeric() # or you may use as well str.isnumeric("975"+"321")
```

```
#| autorun: false
int("975"+"321")
```

There are a number of methods connected to the string data type. Usually the relate to formatting or finding sub-strings. Formatting will be a topic in our next lecture. Here we just refer to one simple find example.

```
#| autorun: false
t
```

```
#| autorun: false
t.find('rld') ## returns the index at which the sub string 'ld' starts in t
```

```
#| autorun: false
t[2:5]
```

```
#| autorun: false
t.capitalize()
```

5.2 Lists

Lists have a variety of uses. They are useful, for example, in various bookkeeping tasks that arise in computer programming. Like arrays, they are sometimes used to store data. However, lists do not have the specialized properties and tools that make arrays so powerful for scientific computing. So in general, we prefer arrays to lists for working with scientific data. For other tasks, lists work just fine and can even be preferable to arrays.

```
#| autorun: false
a=[]
```

```
#| autorun: false
a = [0, 1, 1, 2, 3, 5, 8, 13]
```

```
#| autorun: false
b = [5., "girl", 2+0j, "horse", 21]
```

Individual elements in a list can be accessed by the variable name and the number (index) of the list element put in square brackets. Note that the index for the elements start at *0* for the first element (left).

> **i** Indices in Python
>
> The first element of a list or array is accessed by the index 0. If the array has N elements, the last entries index is N-1.

```
#| autorun: false
b[0],b[1]
```

Elements may be also accessed from the back by nagative indices. *b[-1]* denotes the last element in the list and *b[-2]*, the element before the last.

```
#| autorun: false
b[-1]
```

```
#| autorun: false
b[-2]
```

The length of a list can be obtained by the `len` command if you need the number of elements in the list for your calculations.

```
#| autorun: false
len(b)
```

There are powerful ways to iterate through a list and also through arrays in form of *iterator*. This is called *list comprehension*. We will talk about them later in more detail. Here is an example, which shows the powerful options you have in Python.

```
#| autorun: false
[x+2 for x in b if type(x)!=str]
```

Individual elements in a list can be replaced by assigning a new value to them

```
#| autorun: false
b[-2]='cat'
```

```
#| autorun: false
b
```

Lists can be concatanated by the + operator

```
#| autorun: false
c=a+b
c
```

A very useful feature for lists in python is the **slicing** of lists. Slicing means, that we access only a range of elements in the list, i.e. element 3 to 7. This is done by inserting the starting and the ending element number separated by a colon (:) in the square brackets. The index numbers can be positive or negative again.

```
#| autorun: false
c[3:6]
```

Inserting a second colon behind the ending element index together with a thrid number allows even to select only ever second or third element from a list.

```
#| autorun: false
c[3:9:2]
```

It is sometimes also useful to reverse a list. This can be easily done with the reverse command.

```
#| autorun: false
c.reverse()
c
```

Lists may be created in different ways. An empty list can be created by assigning emtpy square brackets to a variable name. You can append elements to the list with the help of the append command which has to be added to the variable name as shown below. This way of adding a particular function, which is part of a certain variable class is part of object oriented programming.

```
#| autorun: false
a=[0]
```

```
#| autorun: false
a.append(7)
```

```
#| autorun: false
a
```

```
#| autorun: false
a=[1]*10
```

```
#| autorun: false
a
```

A list of numbers can be easily created by the `range()` command.

```
#| autorun: false
list(range(1,10))
```

```
#| autorun: false
list(range(3,10))
```

```
#| autorun: false
list(range(3,10,2))
```

Lists (and also tuples below) can be multidimensional as well, i.e. for an image. The individual elements may then be addressed by supplying two indices in two square brackets.

```
#| autorun: false
a = [[3, 9], [8, 5], [11, 1]]
```

```
#| autorun: false
a[1]
```

```
#| autorun: false
a[2][0]
```

5.3 Tuples

Tuples are also list, but immutable. That means, if a tuple has been once defined, it cannot be changed. Try to change an element to see the result.

```
#| autorun: false
point = 10, 20

print(point, type(point))
```

Tuples may be unpacked, e.g. its values may be assigned to normal variables in the following way

```
#| autorun: false
point[1]
```

```
#| autorun: false
x, y = point

print("x =", x)
print("y =", y)
```

5.4 Dictionaries

Dictionaries are like lists, but the elements of dictionaries are accessed in a different way than for lists. The elements of lists and arrays are numbered consecutively, and to access an element of a list or an array, you simply refer to the number corresponding to its position in the sequence. The elements of dictionaries are accessed by "keys", which can be either strings or (arbitrary) integers (in no particular order). Dictionaries are an important part of core Python. However, we do not make much use of them in this introduction to scientific Python, so our discussion of them is limited.

```
#| autorun: false
room = {"a":"b", "Frank":322, "Dekan":550}
```

```
#| autorun: false
room['Frank']
```

```
#| autorun: false
room.keys()
```

```
#| autorun: false
room.values()
```

5.5 Sets

Sets are like lists but have immutable unique entries, which means the elements can not be changed once defined. An emtpy set is created by the `set()` method.

```
#| autorun: false
phone_id = {112, 114, 116, 118, 115,112}
print('Phone ID:', phone_id)
```

```
#| autorun: false
type(phone_id)
```

```
#| autorun: false
fruits = {'Banana', 'Mango', 'Kiwi', 'Strawberry', 'Orange'}
print('Fruits:', fruits)
```

```
#| tags: []
mixed_set = {'Python', 112, -2, 'Good'}
print('Set of mixed data types:', mixed_set)
```

You may add elements to a set with the `add` method:

You may also remove objects with the `discard` method:

```
#| autorun: false
fruits.discard("Orange")
```

You may also apply a variety of operation to sets checking their *intersection*, *union* or *difference*.

```
#| autorun: false
A = {1, 3, 5}
B = {1, 2, 3}
```

```
#| autorun: false
#intersection
A & B
```

```
#| autorun: false
#symmetric difference
B^A
```

```
#| autorun: false
# union
A | B
```

5.6 Quiz: Data Types in Python

Let's test your understanding of Python data types!

1. What is the output of the following code?

```
a = [1, 2, 3]
b = (1, 2, 3)
print(type(a), type(b))
```

   - ☐ `<class 'list'> <class 'list'>`
   - ☐ `<class 'list'> <class 'tuple'>`
   - ☐ `<class 'tuple'> <class 'list'>`
   - ☐ `<class 'tuple'> <class 'tuple'>`

2. Which of the following is mutable?

   - ☐ List
   - ☐ Tuple
   - ☐ String
   - ☐ Integer

3. What will be the output of this code?

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
print(my_dict['b'])
```

   - ☐ a
   - ☐ 2
   - ☐ b
   - ☐ KeyError

4. How do you create an empty set in Python?

   - ☐ `{}`
   - ☐ `[]`
   - ☐ `set()`
   - ☐ `()`

5. What is the result of `3 + 4.0`?

   - ☐ 7
   - ☐ 7.0
   - ☐ '7.0'

☐ TypeError

Click to reveal answers

1. `<class 'list'> <class 'tuple'>`
2. List
3. 2
4. `set()`
5. 7.0

Part III

Lecture 3

Chapter 6

Python Overview

6.1 Functions

Functions are reusable blocks of code that can be executed multiple times from different parts of your program. They help in organizing code, making it more readable, and reducing redundancy. Functions can take input arguments and return output values.

6.1.1 Defining a Function

A function in Python is defined using the `def` keyword followed by the name of the function, which is usually descriptive and indicates what the function does. The parameters inside the parentheses indicate what data the function expects to receive. The `->` symbol is used to specify the return type of the function.

Here's an example:

```
#| autorun: false
# Define a function that takes two numbers as input and returns their sum
def add_numbers(a: int, b: int) -> int:
  return a + b
```

6.1.2 Calling a Function

Functions can be called by specifying the name of the function followed by parentheses containing the arguments. The arguments passed to the function should match the number and type of parameters defined in the function. Here's an example:

```
#| autorun: false
# Call the function with two numbers as input
result = add_numbers(2, 3)
print(result)  # prints 5
```

6.2 Loops

Loops are used to execute a block of code repeatedly. There are two main types of loops in Python: `for` loops and `while` loops.

6.2.1 For Loop

A `for` loop in Python is used to iterate over a sequence (such as a list or string) and execute a block of code for each item in the sequence. Here's an example:

```
#| autorun: false
# Define a function that prints numbers from 1 to 10
def print_numbers():
  for i in range(1, 11):
    print(i)

print_numbers()
```

6.2.2 While Loop

A `while` loop in Python is used to execute a block of code while a certain condition is met. The loop continues as long as the condition is true. Here's an example:

```
#| autorun: false
# Define a function that prints numbers from 1 to 10 using a while loop
def print_numbers_while():
  i = 1
  while i <= 10:
    print(i)
    i += 1

print_numbers_while()
```

6.3 Conditional Statements

Conditional statements are used to control the flow of your program based on conditions. The main conditional statements in Python are `if`, `else`, and `elif`.

6.3.1 If Statement

An `if` statement in Python is used to execute a block of code if a certain condition is met. Here's an example:

```
#| autorun: false
# Define a function that prints "hello" or "goodbye" depending on the hour of day
def print_hello():
  current_hour = 12
  if current_hour < 18:
    print("hello")

print_hello()
```

6.3.2 Else Statement

An `else` statement in Python is used to execute a block of code if the condition in an `if` statement is not met. Here's an example:

```
#| autorun: false
# Define a function that prints "hello" or "goodbye" depending on the hour of day
def print_hello_or_goodbye():
  current_hour = 12
  if current_hour < 18:
    print("hello")
  else:
    print("goodbye")

print_hello_or_goodbye()
```

6.3.3 Elif Statement

An `elif` statement in Python is used to execute a block of code if the condition in an `if` statement is not met but under an extra condition. Here's an example:

```
#| autorun: false
# Define a function that prints "hello","goodbye" or "good night" depending on the hour of day
def print_hello_or_goodbye():
  current_hour = 12
  if current_hour < 18:
    print("hello")
  elif <20:
    print("goodbye")
  else:
    print("good night")

print_hello_or_goodbye()
```

Chapter 7

Modules

Most of the functionality in Python is provided by *modules*. The Python Standard Library is a large collection of modules that provides *cross-platform* implementations of common facilities such as access to the operating system, file I/O, string management, network communication, math, web-scraping, text manipulation, machine learning and much more.

To use a module in a Python module it first has to be imported. A module can be imported using the `import` statement. For example, to import the module `math`, which contains many standard mathematical functions, we can do:

```
#| autorun: false
import math
import numpy

x = math.sqrt(2 * math.pi)
x = numpy.sqrt(2 * numpy.pi)

print(x)
```

This includes the whole module and makes it available for use later in the program. Alternatively, we can chose to import all symbols (functions and variables) in a module so that we don't need to use the prefix "`math.`" every time we use something from the `math` module:

```
#| autorun: false
from math import *

x = cos(2 * pi)

print(x)
```

This pattern can be very convenient, but in large programs that include many modules it is often a good idea to keep the symbols from each module in their own namespaces, by using the `import math` pattern. This would eliminate potentially confusing problems.

7.0.1 Namespaces

> **Namespaces**
>
> A namespace is an identifier used to organize objects, e.g. the methods and variables of a module. The prefix `math.` we have used in the previous section is such a namespace. You may also create your own namespace for a module. This is done by using the `import math as mymath` pattern.

```
#| autorun: false
import math as m

x = m.sqrt(2)

print(x)
```

You may also only import specific functions of a module.

```
#| autorun: false
from math import sinh as mysinh
```

7.0.2 Directory of a module

Once a module is imported, we can list the symbols it provides using the `dir` function:

```
#| autorun: false
import math

print(dir(math))
```

And using the function `help` we can get a description of each function (almost .. not all functions have docstrings, as they are technically called, but the vast majority of functions are documented this way).

```
#| autorun: false

help(math.log)
```

```
#| autorun: false
math.log(10)
```

```
#| autorun: false
math.log(8, 2)
```

We can also use the `help` function directly on modules: Try

`help(math)`

Some very useful modules form the Python standard library are `os`, `sys`, `math`, `shutil`, `re`, `subprocess`, `multiprocessing`, `threading`.

A complete lists of standard modules for Python 3 is available at http://docs.python.org/3/library/ .

7.0.3 Advanced topics

> **i** Create Your Own Modules
>
> Creating your own modules in Python is a great way to organize your code and make it reusable. A module is simply a file containing Python definitions and statements. Here's how you can create and use your own module:
>
> #### Creating a Module
>
> To create a module, you just need to save your Python code in a file with a `.py` extension. For example, let's create a module named `mymodule.py` with the following content:

```python
# mymodule.py

def greet(name: str) -> str:
    return f"Hello, {name}!"

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

Using Your Module

Once you have created your module, you can import it into other Python scripts using the `import` statement. Here's an example of how to use the `mymodule` we just created:

```python
# main.py

import mymodule

# Use the functions from mymodule
print(mymodule.greet("Alice"))
print(mymodule.add(5, 3))
```

Importing Specific Functions

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

```python
# main.py

from mymodule import greet, add

# Use the imported functions directly
print(greet("Bob"))
print(add(10, 7))
```

Module Search Path

When you import a module, Python searches for the module in the following locations: 1. The directory containing the input script (or the current directory if no script is specified). 2. The directories listed in the `PYTHONPATH` environment variable. 3. The default directory where Python is installed.
You can view the module search path by printing the `sys.path` variable:

```python
import sys
print(sys.path)
```

Creating Packages

A package is a way of organizing related modules into a directory hierarchy. A package is simply a directory that contains a special file named `__init__.py`, which can be empty. Here's an example of how to create a package:

```
mypackage/
    __init__.py
    module1.py
    module2.py
```

You can then import modules from the package using the dot notation:

```
# main.py

from mypackage import module1, module2

# Use the functions from the modules
print(module1.some_function())
print(module2.another_function())
```

Creating and using modules and packages in Python helps you organize your code better and makes it easier to maintain and reuse.

Namespaces in Packages

You can also create sub-packages by adding more directories with `__init__.py` files. This allows you to create a hierarchical structure for your modules:

```
mypackage/
    __init__.py
    subpackage/
        __init__.py
        submodule.py
```

You can then import submodules using the full package name:

```
# main.py

from mypackage.subpackage import submodule

# Use the functions from the submodule
print(submodule.some_sub_function())
```

Part IV

Lecture 4

Chapter 8

NumPy Module

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object and tools for working with these arrays. The NumPy array, formally called ndarray in NumPy documentation, is the real workhorse of data structures for scientific and engineering applications. The NumPy array is similar to a list but where all the elements of the list are of the same type. The elements of a **NumPy array** are usually numbers, but can also be booleans, strings, or other objects. When the elements are numbers, they must all be of the same type.

```
import numpy as np
```

8.1 Creating Numpy Arrays

There are a number of ways to initialize new numpy arrays, for example from

- a Python list or tuples
- using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`, etc.
- reading data from files which will be covered in the files section

8.1.1 From lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function.

```
#| autorun: false
#this is a list
a = [0, 0, 1, 4, 7, 16, 31, 64, 127]
```

```
#| autorun: false
type(a)
```

```
#| autorun: false
#this creates an array out of a list
b=np.array(a,dtype=float)
type(b)
```

```
#| autorun: false
np.array([[1,2,3],[4,5,6],[7,8,9]])
```

8.1.2 Using array-generating functions

For larger arrays it is inpractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in `numpy` that generate arrays of different forms. Some of the more common are:

```
#| autorun: false
# create a range

x = np.arange(0, 10, 1) # arguments: start, stop, step
x
```

```
#| autorun: false
x = np.arange(-5, -2, 0.1)
x
```

linspace and logspace

The `linspace` function creates an array of N evenly spaced points between a starting point and an ending point. The form of the function is linspace(start, stop, N).If the third argument N is omitted,then N=50.

```
#| autorun: false
# using linspace, both end points ARE included
np.linspace(0,10,25)
```

`logspace` is doing equivelent things with logaritmic spacing. Other types of array creation techniques are listed below. Try around with these commands to get a feeling what they do.

```
#| autorun: false
np.logspace(0, 10, 10, base=np.e)
```

mgrid

`mgrid` generates a multi-dimensional matrix with increasing value entries, for example in columns and rows:

```
#| autorun: false
x, y = np.mgrid[0:1:0.1, 0:5] # similar to meshgrid in MATLAB
```

```
#| autorun: false
x
```

```
#| autorun: false
y
```

diag

`diag` generates a diagonal matrix with the list supplied to it. The values can be also offset from the main diagonal.

```
#| autorun: false
# a diagonal matrix
np.diag([1,2,3])
```

```
#| autorun: false
# diagonal with offset from the main diagonal
np.diag([1,2,3], k=-1)
```

zeros and ones

`zeros` and `ones` creates a matrix with the dimensions given in the argument and filled with 0 or 1.

```
#| autorun: false
np.zeros((3,3))
```

```
#| autorun: false
np.ones((3,3))
```

8.2 Manipulating NumPy arrays

8.2.1 Slicing

Slicing is the name for extracting part of an array by the syntax `M[lower:upper:step]`

```
#| autorun: false
A = np.array([1,2,3,4,5])
A
```

```
#| autorun: false
A[1:4]
```

Any of the three parameters in `M[lower:upper:step]` can be ommited.

```
#| autorun: false
A[::] # lower, upper, step all take the default values
```

```
#| autorun: false
A[::2] # step is 2, lower and upper defaults to the beginning and end of the array
```

Negative indices counts from the end of the array (positive index from the begining):

```
#| autorun: false
A = np.array([1,2,3,4,5])
```

```
#| autorun: false
A[-1] # the last element in the array
```

```
#| autorun: false
A[2:] # the last three elements
```

Index slicing works exactly the same way for multidimensional arrays:

```
#| autorun: false
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
```

```
#| autorun: false
# a block from the original array
A[1:3, 1:4]
```

> **i** Differences
>
> **Slicing** can be effectively used to calculate differences for example for the calculation of derivatives. Here the position y_i of an object has been measured at times t_i and stored in an array each. We wish to calculate the average velocity at the times t_i from the arrays by
>
> $$v_i = \frac{y_i - y_{i-1}}{t_i - t_{i-1}} \tag{8.1}$$

```
#| autorun: false
y = np.array([ 0. , 1.3, 5. , 10.9, 18.9, 28.7, 40. ])
t = np.array([ 0. , 0.49, 1. , 1.5 , 2.08, 2.55, 3.2 ])
```

```
#| autorun: false
v = (y[1:]-y[:-1])/(t[1:]-t[:-1])
v
```

8.2.2 Reshaping

Arrays can be reshaped into any form, which contains the same number of elements.

```
#| autorun: false
a=np.zeros(4)
a
```

```
#| autorun: false
np.reshape(a,(2,2))
```

8.2.3 Adding a new dimension: newaxis

With `newaxis`, we can insert new dimensions in an array, for example converting a vector to a column or row matrix.

```
#| autorun: false
v = np.array([1,2,3])
v
```

```
#| autorun: false
v.shape
```

```
#| autorun: false
# make a column matrix of the vector v
v[:, np.newaxis]
```

```
#| autorun: false
# column matrix
v[:,np.newaxis].shape
```

```
#| autorun: false
# row matrix
v[np.newaxis,:].shape
```

i Stacking and repeating arrays

Using function `repeat`, `tile`, `vstack`, `hstack`, and `concatenate` we can create larger vectors and matrices from smaller ones. Please try the individual functions yourself in your notebook. We wont discuss them in detail.

Tile and repeat

```
#| autorun: false
a = np.array([[1, 2], [3, 4]])
a
```

```
#| autorun: false
# repeat each element 3 times
np.repeat(a, 3)
```

```
#| autorun: false
# tile the matrix 3 times
np.tile(a, 3)
```

Concatenate

```
#| autorun: false
b = np.array([[5, 6]])
```

```
#| autorun: false
np.concatenate((a, b), axis=0)
```

```
#| autorun: false
np.concatenate((a, b.T), axis=1)
```

Hstack and vstack

```
#| autorun: false
np.vstack((a,b))
```

```
#| autorun: false
np.hstack((a,b.T))
```

8.3 Applying mathematical functions

All kinds of mathematica operations can be carried out on arrays. Typically these operation act element wise as seen from the examples below.

8.3.1 Operation involving one array

```
#| autorun: false
a=np.arange(0, 10, 1.5)
a
```

```
#| autorun: false
a/2
```

```
#| autorun: false
a**2
```

```
#| autorun: false
 np.sin(a)
```

```
#| autorun: false
np.exp(-a)
```

```
#| autorun: false
(a+2)/3
```

8.3.2 Operations involving multiple arrays

Vector operations enable efficient element-wise calculations where corresponding elements at matching positions are processed simultaneously. Instead of handling elements one by one, these operations work on entire arrays at once, making them particularly fast. When multiplying two vectors using these operations, the result is not a single number (as in a dot product) but rather a new array where each element is the product of the corresponding elements from the input vectors. This element-wise multiplication is just one example of vector operations, which can include addition, subtraction, and other mathematical functions.

```
#| autorun: false
a = np.array([34., -12, 5.,1.2])
```

```
b = np.array([68., 5.0, 20.,40.])
```

```
#| autorun: false
a + b
```

```
#| autorun: false
2*b
```

```
#| autorun: false
a*np.exp(-b)
```

```
#| autorun: false
v1=np.array([1,2,3])
v2=np.array([4,2,3])
```

Chapter 9

Plotting

Data visualization through plotting is a crucial tool for analyzing and interpreting scientific data and theoretical predictions. While plotting capabilities are not built into Python's core, they are available through various external library modules. Matplotlib is widely recognized as the de facto standard for plotting in Python. However, several other powerful plotting libraries exist, including PlotLy, Seaborn, and Bokeh, each offering unique features and capabilities for data visualization.

As Matplotlib is an external library (actually a collection of libraries), it must be imported into any script that uses it. While Matplotlib relies heavily on NumPy, importing NumPy separately is not always necessary for basic plotting. However, for most scientific applications, you'll likely use both. To create 2D plots, you typically start by importing Matplotlib's pyplot module:

`import matplotlib.pyplot as plt`

This import introduces the implicit interface of `pyplot` for creating figures and plots. Matplotlib offers two main interfaces:

- An **implicit** "pyplot" interface that maintains the state of the current Figure and Axes, automatically adding plot elements (Artists) as it interprets the user's intentions.
- An **explicit** "Axes" interface, also known as the "object-oriented" interface, which allows users to methodically construct visualizations by calling methods on Figure or Axes objects to create other Artists.

We will use most of the the the `pyplot` interface as in the examples below. The section *Additional Plotting* will refer to the explicit programming of figures.

```
#| autorun: true

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

We can set some of the parameters for the appearance of graphs globally. In case you still want to modify a part of it, you can set individual parameters later during plotting. The command used here is the

`plt.rcParams.update()`

function, which takes a dictionary with the specific parameters as key.

```
#| autorun: true

plt.rcParams.update({'font.size': 12,
                     'lines.linewidth': 1,
                     'lines.markersize': 10,
                     'axes.labelsize': 11,
                     'xtick.labelsize' : 10,
                     'ytick.labelsize' : 10,
```

```
                    'xtick.top' : True,
                    'xtick.direction' : 'in',
                    'ytick.right' : True,
                    'ytick.direction' : 'in',})

def get_size(w,h):
      return((w/2.54,h/2.54))
```

9.1 Simple Plotting

Matplotlib offers multiple levels of functionality for creating plots. Throughout this section, we'll primarily focus on using commands that leverage default settings. This approach simplifies the process, as Matplotlib automatically handles much of the graph layout. These high-level commands are ideal for quickly creating effective visualizations without delving into intricate details. At the end of this section, we'll briefly touch upon more advanced techniques that provide greater control over plot elements and layout.

9.1.1 Anatomy of a Line Plot

To create a basic line plot, use the following command:

```
plt.plot(x, y)
```

By default, this generates a line plot. However, you can customize the appearance by adjusting various parameters within the `plot()` function. For instance, you can modify it to resemble a scatter plot by changing certain arguments. The versatility of this command allows for a range of visual representations beyond simple line plots.

Let's create a simple line plot of the sine function over the interval $[0, 4]$. We'll use NumPy to generate the x-values and calculate the corresponding y-values. The following code snippet demonstrates this process:

```
x = np.linspace(0, 4.*np.pi, 100)                    ①
y = np.sin(x)                                        ②

plt.figure(figsize=(4,3))                            ③
plt.plot(x, y)                                       ④
plt.tight_layout()                                   ⑤
plt.show()                                           ⑥
```

① Create an array of 100 values between 0 and 4 .
② Calculate the sine of each value in the array.
③ create a new figure
④ plot the data
⑤ automatically adjust the layout
⑥ show the figure

Here is the code in a Python cell:

```
#| autorun: true

x = np.linspace(0, 4.*np.pi, 100)
y = np.sin(x)

plt.figure(figsize=(4,3))
plt.plot(x, y)
plt.tight_layout()
plt.show()
```

Try to change the values of the `x` and `y` arrays and see how the plot changes.

Why use plt.tight_layout()

`plt.tight_layout()` is a very useful function in Matplotlib that automatically adjusts the spacing between plot elements to prevent overlapping and ensure that all elements fit within the figure area. Here's what it does:

1. Padding Adjustment: It adjusts the padding between and around subplots to prevent overlapping of axis labels, titles, and other elements.
2. Subplot Spacing: It optimizes the space between multiple subplots in a figure.
3. Text Accommodation: It ensures that all text elements (like titles, labels, and legends) fit within the figure without being cut off.
4. Margin Adjustment: It adjusts the margins around the entire figure to make sure everything fits neatly.
5. Automatic Resizing: If necessary, it can slightly resize subplot areas to accommodate all elements.
6. Legend Positioning: It takes into account the presence and position of legends when adjusting layouts.

Key benefits of using `plt.tight_layout()`:

- It saves time in manual adjustment of plot elements.
- It helps create more professional-looking and readable plots.
- It's particularly useful when creating figures with multiple subplots or when saving figures to files.

You typically call `plt.tight_layout()` just before `plt.show()` or `plt.savefig()`. For example:

```
plt.figure()
# ... (your plotting code here)
plt.tight_layout()
plt.show()
```

Axis Labels

To enhance the clarity and interpretability of our plots, it's crucial to provide context through proper labeling. Let's add descriptive axis labels to our diagram, a practice that significantly improves the readability and comprehension of the data being presented.

```
plt.xlabel('x-label')
plt.ylabel('y-label')
```

```
#| autorun: false

plt.figure(figsize=(4,3))
plt.plot(x,np.sin(x)) # using x-axis
plt.xlabel('t') # set the x-axis label
plt.ylabel('sin(t)') # set the y-axis label
plt.tight_layout()
plt.show()
```

Legends

```
plt.plot(..., label=r'$\sin(x)$')
plt.legend(loc='lower left')
```

```
#| autorun: false


plt.figure(figsize=(4,3))
plt.plot(x,np.sin(x),"ko",markersize=5,label=r"$\delta(t)$") #define an additional label
plt.xlabel('t')
plt.ylabel(r'$\sin(t)$')

plt.legend(loc='lower left') #add the legend
```

```
plt.tight_layout()
plt.show()
```

Plots with error bars

When plotting experimental data it is customary to include error bars that indicate graphically the degree of uncertainty that exists in the measurement of each data point. The MatPlotLib function errorbar plots data with error bars attached. It can be used in a way that either replaces or augments the plot function. Both vertical and horizontal error bars can be displayed. The figure below illustrates the use of error bars.

```
#| autorun: false

import numpy as np
import matplotlib.pyplot as plt

plt.figure(1, figsize = (8,6) )
xdata=np.arange(0.5,3.5,0.5)
ydata=210-40/xdata

yerror=2e3/ydata

plt.errorbar(xdata, ydata, fmt="ro", label="data",
             xerr=0.15, yerr=yerror, ecolor="black")

plt.xlabel("x")
plt.ylabel("transverse displacement")
plt.legend(loc="lower right")
plt.tight_layout()

plt.show()
```

Saving figures

To save a figure to a file we can use the `savefig` method in the `Figure` class. Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF. For scientific papers, I recommend using PDF whenever possible. (LaTeX documents compiled with `pdflatex` can include PDFs using the `includegraphics` command). In some cases, PGF can also be good alternative.

```
#| autorun: false


theta = np.arange(0.01, 10., 0.04)
ytan = np.sin(2*theta)+np.cos(3.1*theta)

plt.figure(figsize=(8,6))
plt.plot(theta, ytan)
plt.xlabel(r'$\theta$')
plt.ylabel('y')
plt.tight_layout()
plt.savefig('filename.pdf')
plt.show()
```

9.2 Other Plot Types

9.2.1 Scatter plot

If you prefer to use symbols for plotting just use the

`plt.scatter(x,y)`

command of pylab. Note that the scatter command requires a x and y values and you can set the marker symbol (see an overview of the marker symbols).

```
#| autorun: false

plt.figure(figsize=(4,3))
plt.scatter(x,np.cos(x),marker='o')
plt.xlabel('t') # set the x-axis label
plt.ylabel('y') # set the y-axis label
plt.tight_layout()
plt.show()
```

9.2.2 Histograms

A very useful plotting command is also the *hist* command. It generates a histogram of the data provided. A histogram is a graphical representation of the distribution of numerical data. It is an estimate of the probability distribution of a continuous variable. To construct a histogram, the first step is to "bin" the range of values—that is, divide the entire range of values into a series of intervals—and then count how many values fall into each interval. The bins are usually specified as consecutive, non-overlapping intervals of a variable. The bins must be adjacent, and are often (but not required to be) of equal size.

When using the histogram function, you have flexibility in how the data is grouped. If you only provide the dataset, the function will automatically determine appropriate bins. However, you can also specify custom bins by passing an array of intervals using the syntax `hist(data, bins=b)`, where `b` is your custom array of bin edges. To normalize the histogram so that the total area under it equals 1, you can set the `density` parameter to `True`. It's worth noting that the histogram function doesn't just create a visual representation; it also returns useful information such as the count of data points in each bin and the bin edges themselves.

i Physics Interlude- Probability density for finding an oscillating particle

Let's integrate histogram plotting with a fundamental physics concept: the simple harmonic oscillator in one dimension. This system is described by a specific equation of motion:

$$\ddot{x}(t) = -\omega^2 x(t) \tag{9.1}$$

For an initial elongation Δx at $t = 0$, the solution is:

$$x(t) = \Delta x \cos(\omega t) \tag{9.2}$$

To calculate the probability of finding the spring at a certain elongation, we need to consider the time spent at different positions. The time dt spent in the interval $[x(t), x(t) + dx]$ depends on the speed:

$$v(t) = \frac{dx}{dt} = -\omega \Delta x \sin(\omega t) \tag{9.3}$$

The probability of finding the oscillator in a certain interval is the fraction of time spent in this interval, normalized by half the oscillation period $T/2$:

$$\frac{dt}{T/2} = \frac{1}{T/2}\frac{dx}{v(t)} = \frac{1}{T/2}\frac{-dx}{\omega \Delta x \sin(\omega t)} \tag{9.4}$$

Given that $\omega = 2\pi/T$, we can derive the probability density:

$$p(x)dx = \frac{1}{\pi\Delta x}\frac{dx}{\sqrt{1-\left(\frac{x(t)}{\Delta x}\right)^2}} \tag{9.5}$$

This probability density reveals that the spring is more likely to be found at elongations where its speed is low. This principle extends to non-equilibrium physics, where entities moving with variable speed are more likely to be found in locations where they move slowly.
We can visualize this using the histogram function. By evaluating the position at equidistant times using the equation of motion and creating a histogram of these positions, we can represent the probability of finding the oscillator at certain positions. When properly normalized, this histogram will reflect the theoretical probability density we derived.

```
#| autorun: false

t=np.linspace(0,np.pi,10000)
y=np.cos(t)

#ymin=np.mean(y)-2*np.std(y)
#ymax=np.mean(y)+2*np.std(y)
b=np.linspace(-1,1,10)

# plot the histogram
n, bins, _ =plt.hist(y,bins=b,density=True,color='lightgreen',label='histogram')

#plot the analytical solution
x=np.linspace(-0.99,0.99,100)
plt.plot(x,1/(np.pi*np.sqrt(1-x**2)),'k--',label='analytic')

plt.xlabel('y') # set the x-axis label
plt.ylabel('occurrence') # set the y-axis label
plt.legend()
plt.show()
```

9.2.3 Setting plotting limits and excluding data

If you want to zoom in to s specific region of a plot you can set the limits of the individual axes.

```
#| autorun: false


import numpy as np
import matplotlib.pyplot as plt
theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)
plt.figure(figsize=(8,6))
plt.plot(theta, ytan)
plt.xlim(0, 4)
plt.ylim(-8, 8) # restricts range of y axis from -8 to +8 plt.axhline(color="gray", zorder=-1)
plt.show()
```

9.2.4 Masked arrays

Sometimes you encounter situations, when you wish to mask some of the data of your plot, because they are not showing real data as the vertical lines in the plot above. For this purpose, you can mask the data arrays in various ways to not show up. The example below uses the

`np.ma.masked_where()`

function of NumPy, which takes a condition as the first argument and what should be returned if that condition is fulfilled.

```
#| autorun: false


import numpy as np
import matplotlib.pyplot as plt
theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)

#exclude specific data from plotting
ytanM = np.ma.masked_where(np.abs(ytan)>20., ytan)

plt.figure(figsize=(8,6))
plt.plot(theta, ytanM)
plt.ylim(-8, 8)
#plt.axhline(color="gray", zorder=-1)
plt.show()
```

If you look at the resulting array, you will find, that the entries have not been removed but replaced by `--`, so the values are not existent and thefore not plotted.

i Logarithmic plots

Data sets can span many orders of magnitude from fractional quantities much smaller than unity to values much larger than unity. In such cases it is often useful to plot the data on logarithmic axes.

9.2.5 Semi-log plots

For data sets that vary exponentially in the independent variable, it is often useful to use one or more logarithmic axes. Radioactive decay of unstable nuclei, for example, exhibits an exponential decrease in the number of particles emitted from the nuclei as a function of time.
MatPlotLib provides two functions for making semi-logarithmic plots, semilogx and semilogy, for creating plots with logarithmic x and y axes, with linear y and x axes, respectively. We illustrate their use in the program below, which made the above plots.

```
#| autorun: false

# create theoretical curve
tau = 20.2 # Phosphorus-32 half life = 14 days; tau = t_half/ln(2)
N0 = 8200. # Initial count rate (per second)
t = np.linspace(0, 180, 30)
N = N0 * np.exp(-t/tau)

# create plot
plt.figure(1, figsize = (6,5) )

plt.semilogy(t, N, 'bo', label="theory")
plt.xlabel('time (days)')
plt.ylabel('counts per second')
plt.legend(loc='upper right')

plt.tight_layout()
plt.show()
```

9.2.6 Log-log plots

MatPlotLib can also make log-log or double-logarithmic plots using the function loglog. It is useful when both the x and y data span many orders of magnitude. Data that are described by a power law $y = Ax^b$, where A and b are constants, appear as straight lines when plotted on a log-log plot. Again, the loglog function works just like the plot function but with logarithmic axes.

```
#| autorun: false

# create theoretical fitting curve
tau = 20.2 # Phosphorus-32 half life = 14 days; tau = t_half/ln(2)
N0 = 8200. # Initial count rate (per second)
t = np.linspace(0.1, 180, 128)
N = N0 * t**(-3)
N1 = N0 * t**(-2)

# create plot
plt.figure(1, figsize = (3,3),dpi=150 )
plt.loglog(t, N, 'b-', label="theory")
plt.loglog(t, N1, 'r-', label="theory")
#plt.plot(time, counts, 'ro', label="data")
plt.xlabel('time (days)')
plt.ylabel('counts per second')
plt.legend(loc='upper right')
plt.show()
```

9.2.7 Combined plots

You can combine multiple data with the same axes by stacking multiple plots.

```
#| autorun: false


# create x and y arrays for theory
x = np.linspace(-10., 10., 100)
y = np.cos(x) * np.exp(-(x/4.47)**2)

# create plot
plt.figure(figsize = (7,5))

plt.plot(x, y, 'b-', label='theory')
plt.scatter(x, y,  label="data")

plt.axhline(color = 'gray', linewidth=5,zorder=-1)
plt.axvline(ls='--',color = 'gray', zorder=-1)


plt.xlabel('x')
plt.ylabel('transverse displacement')
plt.legend(loc='upper right')

plt.tight_layout()
plt.show()
```

9.2.8 Arranging multiple plots

Often you want to create two or more graphs and place them next to one another, generally because they are related to each other in some way.

```
#| autorun: false

theta = np.arange(0.01, 8., 0.04)
y = np.sqrt((8./theta)**2-1.)
ytan = np.tan(theta)
ytan = np.ma.masked_where(np.abs(ytan)>20., ytan)
ycot = 1./np.tan(theta)

ycot = np.ma.masked_where(np.abs(ycot)>20., ycot)
```

```
#| autorun: false

plt.figure(1,figsize=(8,8))
plt.subplot(2, 2, 2)

plt.plot(theta, ytan)
plt.ylim(-8, 8)
plt.xlabel("theta")
plt.ylabel("tan(theta)")

plt.subplot(2, 2, 3)
plt.plot(theta, -y)
plt.plot(theta, ycot)
plt.ylim(-8, 8)
plt.xlabel(r'$\theta$')
plt.ylabel(r'cot($\theta$)')

plt.tight_layout()
plt.show()
```

Animations

Matplotlib can also be used to create animations. The `FuncAnimation` class makes it easy to create animations by repeatedly calling a function to update the plot. The following example shows a simple pendulum animation.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

# Pendulum parameters
L = 1.0  # length of pendulum
g = 9.81  # gravitational acceleration
theta0 = np.pi/3  # initial angle

# Time array
t = np.linspace(0, 10, 1000)
omega = np.sqrt(g/L)
theta = theta0 * np.cos(omega * t)

# Create figure
fig, ax = plt.subplots()
ax.set_xlim(-1.2, 1.2)
ax.set_ylim(-1.2, 1.2)
ax.set_aspect('equal')

# Initialize pendulum line and bob
line, = ax.plot([], [], 'k-', lw=2)
bob, = ax.plot([], [], 'ko', markersize=20)

# Animation initialization function
def init():
    line.set_data([], [])
    bob.set_data([], [])
    return line, bob

# Animation update function
def update(frame):
    x = L * np.sin(theta[frame])
    y = -L * np.cos(theta[frame])
    line.set_data([0, x], [0, y])
    bob.set_data([x], [y])
    return line, bob

# Create animation
anim = FuncAnimation(fig, update, frames=len(t),
                     init_func=init, blit=True,
                     interval=20)

# Display animation
HTML(anim.to_jshtml())
```

9.2.9 Simple contour plot

Physics Interlude

9.3 Contour and Density Plots

A contour plots are useful tools to study two dimensional data, meaning $Z(X, Y)$. A contour plots the lines of constant value of the function Z.

9.4 Understanding Wave Interference

Imagine throwing two stones into a pond. Each stone creates circular waves that spread out. When these waves meet, they create interesting patterns - this is called interference. Let's explore this using physics and Python!

9.4.1 What is a Wave?

A wave can be described mathematically. For our example, we'll look at spherical waves (like those in the pond). Each wave has: - An amplitude (how tall the wave is) - A wavelength (distance between wave peaks) - A frequency (how fast it oscillates)

9.4.2 Mathematical Description

For a single wave source, we can write:

$$U(r) = e^{-i\,kr} \tag{9.6}$$

Where: - k is related to the wavelength ($k = 2\pi/\lambda$) - r is the distance from the source - We've simplified by ignoring how the wave gets smaller as it travels ($1/r$ term)

9.4.3 Two Wave Sources

When we have two wave sources (like two stones dropped in the pond): 1. Each source creates its own wave 2. The waves combine where they meet 3. The total wave is the sum of both waves

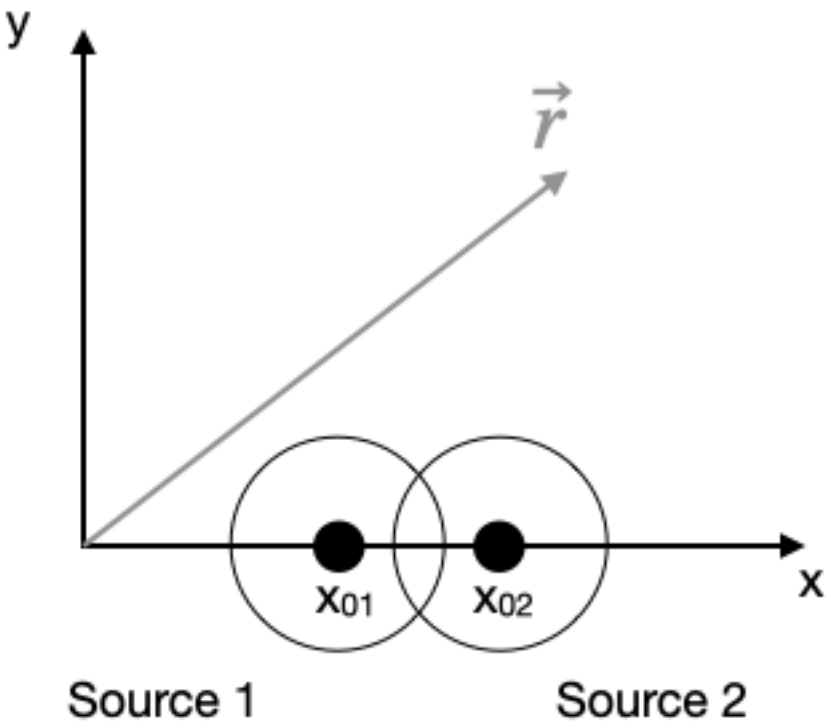


Figure 9.1: interference

Mathematically:

$$U_{total} = e^{-ikr_1} + e^{-ikr_2} \tag{9.7}$$

Where r_1 and r_2 are the distances from each source.

9.4.4 What We See (Intensity)

What we actually see is the intensity of the combined waves:

$$\text{Intensity} \propto |U_{total}|^2 \tag{9.8}$$

This will show us where the waves:

- Add up (bright regions - constructive interference)
- Cancel out (dark regions - destructive interference)

Let's create a Python program to visualize this!

```
#| autorun: false

lmda = 2 # defines the wavelength
x01=1.5*np.pi # location of the first source, y01=0
x02=2.5*np.pi # location of the second source, y02=0
x = np.linspace(0, 4*np.pi, 100)
y = np.linspace(0, 4*np.pi, 100)


X, Y = np.meshgrid(x, y)
I= np.abs( np.exp(-1j*np.sqrt((X-x01)**2+Y**2)*2*np.pi/lmda)
          +np.exp(-1j*np.sqrt((X-x02)**2+Y**2)*2*np.pi/lmda))**2
```

```
#| autorun: false

plt.figure(1,figsize=(6,6))

#contour plot
plt.contour(X, Y, I, 20)
plt.colorbar()
plt.show()
```

9.4.5 Color contour plot

```
#| autorun: false

plt.figure(1,figsize=(7,6))
plt.contourf(X, Y, I, 10, cmap='gray')
plt.colorbar()
plt.show()
```

9.4.6 Image plot

```
#| autorun: false

plt.figure(1,figsize=(7,6))
plt.imshow( I,extent=[10,0,0,10],cmap='gray');
plt.ylim(10,0)
plt.colorbar()
plt.show()
```

Advanced Plotting - Explicit Version

9.5 Advanced Plotting - Explicit Version

While we have so far largely relied on the default setting and the automatic arrangement of plots, there is also a way to precisely design your plot. Python provides the tools of object oriented programming and thus modules provide classes which can be instanced into objects. This explicit interfaces allows you to control all details without the automatisms of `pyplot`.
The figure below, which is taken from the matplotlib documentation website shows the sets of commands and the objects in the figure, the commands refer to. It is a nice reference, when creating a figure.

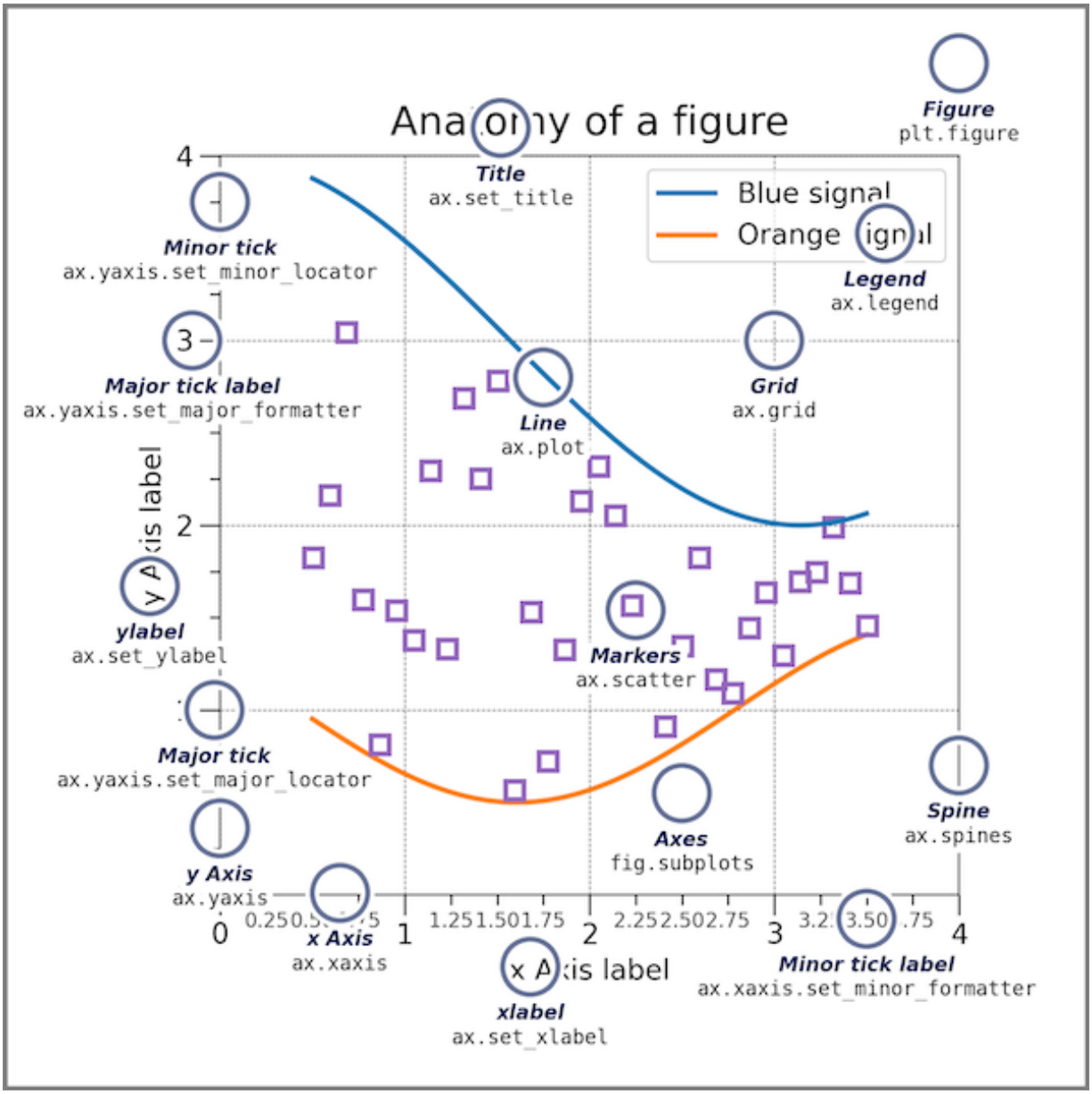


Figure 9.2: anatomy of a figure

9.5.1 Plots with Multiple Spines

Sometimes it is very useful to plot different quantities in the same plot with the same x-axis but with different y-axes. Here is some example, where each line plot has its own y-axis.

```
#| autorun: false

fig, ax = plt.subplots()

fig.subplots_adjust(right=0.75)
fig.subplots_adjust(right=0.75)

twin1 = ax.twinx()
twin2 = ax.twinx()

# Offset the right spine of twin2.  The ticks and label have already been
# placed on the right by twinx above.
twin2.spines.right.set_position(("axes", 1.2))

time=np.linspace(0,10,100)
omega=2
p1, = ax.plot(time,np.cos(omega*time), "C0", label="position")
p2, = twin1.plot(time,-omega*np.sin(omega*time), "C1", label="velocity")
p3, = twin2.plot(time, -omega**2*np.cos(omega*time), "C2", label="acceleration")

ax.set(xlim=(0, 10), ylim=(-2, 2), xlabel="time", ylabel="position")
twin1.set(ylim=(-4, 4), ylabel="velocity")
twin2.set(ylim=(-6, 6), ylabel="acceleration")

ax.yaxis.label.set_color(p1.get_color())
twin1.yaxis.label.set_color(p2.get_color())
twin2.yaxis.label.set_color(p3.get_color())

ax.tick_params(axis='y', colors=p1.get_color())
twin1.tick_params(axis='y', colors=p2.get_color())
twin2.tick_params(axis='y', colors=p3.get_color())

#ax.legend(handles=[p1, p2, p3])

plt.show()
```

9.5.2 Insets

Insets are plots within plots using their own axes. We therefore need to create two axes systems, if we want to have a main plot and and inset.

```
#| autorun: false

x = np.linspace(0, 5, 10)
y = x ** 2
```

```
#| autorun: false

fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.9, 0.9]) # main axes
axes2 = fig.add_axes([0.35, 0.2, 0.4, 0.3]) # inset axes

# main figure
axes1.plot(x, y, 'r')
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('title')

# insert
axes2.plot(y, x, 'g')
axes2.set_xlabel('y')
axes2.set_ylabel('x')
axes2.set_title('insert title')
plt.show()
```

9.5.3 Spine axis

```
#| autorun: false

fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))   # set position of y spine to y=0

xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```

9.5.4 Polar plot

```
#| autorun: false

# polar plot using add_axes and polar projection
fig = plt.figure()
ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
t = np.linspace(0, 2 * np.pi, 100)
ax.plot(t, t, color='blue', lw=3);
```

9.5.5 Text annotation

Annotating text in matplotlib figures can be done using the `text` function. It supports LaTeX formatting just like axis label texts and titles:

```
#| autorun: false

fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)

ax.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="C0")
ax.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="C1");
```

9.5.6 3D Plotting

Matplotlib was initially designed with only two-dimensional plotting in mind. Around the time of the 1.0 release, some three-dimensional plotting utilities were built on top of Matplotlib's two-dimensional display, and the result is a convenient (if somewhat limited) set of tools for three-dimensional data visualization. Three-dimensional plots are enabled by importing the mplot3d toolkit, included with the main Matplotlib installation:

```
#| autorun: false

from mpl_toolkits import mplot3d
```

Once this submodule is imported, a three-dimensional axes can be created by passing the keyword projection='3d' to any of the normal axes creation routines:

Projection Scence

```
#| autorun: false

fig=plt.figure(figsize=(6,6))
ax = plt.axes(projection='3d')
#ax.set_proj_type('ortho')
ax.set_proj_type('persp', focal_length=0.2)
ax.view_init(elev=40., azim=10)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.show()
```

With this three-dimensional axes enabled, we can now plot a variety of three-dimensional plot types. Three-dimensional plotting is one of the functionalities that benefits immensely from viewing figures interactively rather than statically in the notebook; recall that to use interactive figures, you can use %matplotlib notebook rather than %matplotlib inline when running this code.

Line Plotting in 3D

from sets of (x, y, z) triples. In analogy with the more common two-dimensional plots discussed earlier, these can be created using the `ax.plot3D` and `ax.scatter3D` functions. The call signature for these is nearly identical to that of their two-dimensional counterparts, so you can refer to Simple Line Plots and Simple Scatter Plots for more information on controlling the output. Here we'll plot a trigonometric spiral, along with some points drawn randomly near the line:

```
#| autorun: false
plt.figure(figsize=(10,6))

#create the axes
ax = plt.axes(projection='3d')

# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'blue')

# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens')
plt.show()
```

Notice that by default, the scatter points have their transparency adjusted to give a sense of depth on the page. While the three-dimensional effect is sometimes difficult to see within a static image, an interactive view can lead to some nice intuition about the layout of the points. Use the `scatter3D` or the `plot3D` method to plot a random walk in 3-dimensions in your exercise.

Surface Plotting

A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon. Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized:

```
#| autorun: false

x = np.linspace(-6, 6, 50)
y = np.linspace(-6, 6, 60)

X, Y = np.meshgrid(x, y)
Z=np.sin(X)*np.sin(Y)
```

```
#| autorun: false

np.shape(Z)
```

```
#| autorun: false

plt.figure(figsize=(10,6))
ax = plt.axes(projection='3d')
ax.view_init(30, 50)
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.set_title('surface')
plt.show()
```

Part V

Lecture 5

Chapter 10

Classes and Objects

```
#| edit: false
#| echo: false
# include the required modules

import numpy as np
import matplotlib.pyplot as plt

plt.rcParams.update({'font.size': 12,
                     'lines.linewidth': 1,
                     'lines.markersize': 10,
                     'axes.labelsize': 11,
                     'xtick.labelsize' : 10,
                     'ytick.labelsize' : 10,
                     'xtick.top' : True,
                     'xtick.direction' : 'in',
                     'ytick.right' : True,
                     'ytick.direction' : 'in',})
```

10.1 Object oriented programming

A very useful programming concept is object oriented programming. In all the programs we wrote till now, we have designed our program around functions i.e. blocks of statements which manipulate data. This is called the procedure-oriented way of programming.

There is another way of organizing your program which is to combine data and functionality and wrap it inside something called an object. This is called the object oriented programming paradigm, which will be useful especially for larger programs.

10.1.1 Classes and Objects

Object-oriented programming is built upon two fundamental concepts: classes and objects.

- A class is a blueprint or template that defines a new type of object. Think of it as a mold that creates objects with specific characteristics and behaviors.
- Objects are specific instances of a class. They have two main components:
  - **Properties** (also called attributes or fields): Variables that store data within the object
  - **Methods**: Functions that define what the object can do
- Properties come in two varieties:

  - **Instance variables**: Unique to each object instance (each object has its own copy)
  - **Class variables**: Shared among all instances of the class (one copy for the entire class)

For example, if you had a `Car` class: - Instance variables might include `color` and `mileage` (unique to each car) - Class variables might include `number_of_wheels` (same for all cars) - Methods might include `start_engine()` or `brake()`

10.1.2 Creating Classes

To define a class in Python, we use this basic syntax:

```
class ClassName:
    # Class content goes here
```

The definition starts with the `class` keyword, followed by the class name, and a colon. The class content is indented and contains all properties and methods of the class.

Here's a minimal example:

```
#| autorun: false
class Colloid:
    pass  # 'pass' creates an empty class with no properties or methods
```

To create an object (an instance) of this class:

```
#| autorun: false
particle = Colloid()
particle  # Prints the object's location in memory
```

10.2 Class Methods

Methods are functions that belong to a class. They define the behavior of the class and can operate on the class's properties.

> Understanding `self` in Python Classes
>
> Every method in a Python class automatically receives a special first parameter, conventionally named `self`. This parameter refers to the specific instance of the class that calls the method.
> Key points about `self`: - It's automatically passed by Python when you call a method - It gives the method access to the instance's properties - By convention, we name it `self` (though technically you could use any valid name) - You don't include it when calling the method
> Example:
>
> ```
> class Colloid:
>     def type(self):  # self is automatically provided
>         print('I am a plastic colloid')
>
> # Usage:
> particle = Colloid()
> particle.type()  # Notice: no argument needed for self
> ```
>
> In this example, even though `type()` appears to take no arguments when called, it actually receives the `particle` object as `self`.

```
#| autorun: false
class Colloid:
    def type(self):
        print('I am a plastic colloid')
```

```
p = Colloid()
p.type()

b=Colloid()
b.type()
```

10.2.1 The Constructor Method: `__init__`

The `__init__` method (called the constructor) is a special method that initializes a new object when it's created. It allows you to: - Set up initial values for the object's properties - Perform any setup the object needs when it's created

The name has double underscores (dunders) at both ends: `__init__`

Here's an example:

```
#| autorun: false
class Colloid:
    def __init__(self, R):
        self.R = R  # Stores the radius as an instance variable

    def get_size(self):
        return self.R  # Method to retrieve the radius
```

Using the class:

```
#| autorun: false
# Create two colloids with different radii
particle1 = Colloid(5)  # radius = 5
particle2 = Colloid(2)  # radius = 2

# Get the size of particle1
print(f'Colloid radius is {particle1.get_size()} μm')
```

> **Note**
>
> Python also provides a `__del__` method (destructor) that's called when an object is deleted. We'll see this in action later.

10.2.2 The String Representation: `__str__` Method

The `__str__` method defines how an object should be represented as a string. Python automatically calls this method when: - You use `print(object)` - You convert the object to a string using `str(object)`

Here's an example:

```
#| autorun: false
class Colloid:
    def __init__(self, R):
        self.R = R  # Initialize radius

    def get_size(self):
        return self.R

    def __str__(self):
        # Define how the object should be displayed as text
        return f'I am a plastic colloid of radius {self.R:.1f}'
```

> **Tip**
>
> The `.1f` format specification means the radius will be displayed with one decimal place. You can customize this string representation to show whatever information about your object is most relevant.

Let's see it in action:

```
#| autorun: false
# Create a colloid with radius 15
particle = Colloid(15)

# Print the object - this automatically calls __str__
print(particle)
```

10.3 Understanding Class and Instance Variables

In Python classes, we can have two types of variables that store data:

10.3.1 Class Variables (Shared Data)

- Shared among all instances of a class
- Defined inside the class but outside any method
- All objects share the same copy of these variables
- Changes affect all instances
- Useful for tracking data common to all instances

10.3.2 Instance Variables (Individual Data)

- Unique to each instance/object
- Usually defined in `__init__`
- Each object has its own copy
- Changes only affect that specific instance
- Useful for object-specific properties

Here's a practical example:

```
#| autorun: false
class Colloid:
    # Class variable: tracks total number of particles
    total_particles = 0

    def __init__(self, R):
        # Instance variable: each particle has its own radius
        self.R = R
        # Increment counter when new particle is created
        Colloid.total_particles += 1

    def __del__(self):
        # Decrement counter when particle is deleted
        Colloid.total_particles -= 1
```

Let's see how it works:

```
#| autorun: false
# Create two particles
p1 = Colloid(3)   # Particle with radius 3
p2 = Colloid(12)  # Particle with radius 12
```

```
# Each particle has its own radius (instance variable)
print(f"Particle radii: p1 = {p1.R}, p2 = {p2.R}")

# Both share the same total_particles count (class variable)
print(f"Total particles: {Colloid.total_particles}")

# Delete one particle
del p2
print(f"After deletion, total particles: {Colloid.total_particles}")
```

Tip

Common uses for class variables:

- Counters (like tracking total instances)
- Constants shared by all instances
- Configuration values for all objects

Chapter 11

Brownian Motion

We will apply our newly acquired knowledge about classes to simulate Brownian motion. This task aligns perfectly with the principles of object-oriented programming, as each Brownian particle (or colloid) can be represented as an object instantiated from the same class, albeit with different properties. For instance, some particles might be larger while others are smaller. We have already touched on some aspects of this in previous lectures.

```
#| autorun: true
#| edit: false
#| echo: false
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

plt.rcParams.update({'font.size': 12,
                     'lines.linewidth': 1,
                     'lines.markersize': 10,
                     'axes.labelsize': 11,
                     'xtick.labelsize' : 10,
                     'ytick.labelsize' : 10,
                     'xtick.top' : True,
                     'xtick.direction' : 'in',
                     'ytick.right' : True,
                     'ytick.direction' : 'in',})
```

11.1 Brownian Motion

11.1.1 What is Brownian Motion?

Imagine a dust particle floating in water. If you look at it under a microscope, you'll see it moving in a random, zigzag pattern. This is Brownian motion!

11.1.2 Why Does This Happen?

When we observe Brownian motion, we're seeing the effects of countless molecular collisions. Water isn't just a smooth, continuous fluid - it's made up of countless tiny molecules that are in constant motion. These water molecules are continuously colliding with our particle from all directions. Each individual collision causes the particle to move just a tiny bit, barely noticeable on its own. However, when millions of these tiny collisions happen every second from random directions, they create the distinctive zigzag motion we observe.

11.1.3 The Simplified Math Behind It

When our particle moves:

1. Each step is random in direction
2. The size of each step depends on:
    - Temperature (warmer = more movement)
    - Time between steps
    - A property called the "diffusion coefficient" (D)

11.1.4 How We Can Simulate This?

In Python, we can simulate these random steps using random number. These random numbers can be generated with the numpy library. Numpy provides a number of different functions that provide random numbers from different distributions. For Brownian motion, we use a special distribution called the "normal distribution".

```
step_size = np.sqrt(2 * D * time_step)
dx = random_number * step_size  # Random step in x direction
dy = random_number * step_size  # Random step in y direction

new_x = old_x + dx
new_y = old_y + dy
```

Where:

- `D` is how easily the particle moves (diffusion coefficient)
- `time_step` is how often we update the position
- `random_number` is chosen from a special "normal distribution"

> **Tip**
>
> When simulating Brownian motion, we use `np.random.normal` to generate random steps following this distribution. The normal distribution is characterized by two parameters: the mean and the standard deviation. The mean is the average value, and the standard deviation is a measure of how spread out the values are. For Brownian motion, we use a standard deviation that depends on the diffusion coefficient and the time step. The standard deviation $\sigma = \sqrt{2D\Delta t}$ determines the typical step size, which we can use as a parameter in the normal distribution.

```
#| autorun: false

# some space to test out some of the random numbers
```

> **Advanced Mathematical Details**
>
> The Brownian motion of a colloidal particle results from collisions with surrounding solvent molecules. These collisions lead to a probability distribution described by:
>
> $$p(x, \Delta t) = \frac{1}{\sqrt{4\pi D \Delta t}} e^{-\frac{x^2}{4D\Delta t}}$$
>
> where: - D is the diffusion coefficient - Δt is the time step - The variance is $\sigma^2 = 2D\Delta t$
> This distribution emerges from the **central limit theorem**, as shown by Lindenberg and Lévy, when considering many infinitesimally small random steps.
> The evolution of the probability density function $p(x, t)$ is governed by the diffusion equation:

$$\frac{\partial p}{\partial t} = D\frac{\partial^2 p}{\partial x^2}$$

This partial differential equation, also known as Fick's second law, describes how the concentration of particles evolves over time due to diffusive processes. The Gaussian distribution above is the fundamental solution (Green's function) of this diffusion equation, representing how an initially localized distribution spreads out over time.
The connection between the microscopic random motion and the macroscopic diffusion equation was first established by Einstein in his 1905 paper on Brownian motion, providing one of the earliest quantitative links between statistical mechanics and thermodynamics.

11.2 Why Use a Class?

A class is perfect for this physics simulation because each colloidal particle:

1. Has specific properties
   - Size (radius)
   - Current position
   - Movement history
   - Diffusion coefficient
2. Follows certain behaviors
   - Moves randomly (Brownian motion)
   - Updates its position over time
   - Keeps track of where it's been
3. Can exist alongside other particles
   - Many particles can move independently
   - Each particle keeps track of its own properties
   - Particles can have different sizes
4. Needs to track its state over time
   - Remember previous positions
   - Calculate distances moved
   - Maintain its own trajectory

This natural mapping between real particles and code objects makes classes an ideal choice for our simulation.

11.3 Class Design

Let's design a Python class to simulate colloidal particles undergoing Brownian motion. This object-oriented approach will help us manage multiple particles with different properties and behaviors.

11.3.1 Class-Level Properties

The `Colloid` class will maintain information shared by all particles:

1. A counter for the total number of particles
2. The physical constant $k_B T/(6\pi\eta) = 2.2 \times 10^{-19}$ (combining temperature and fluid properties)

11.3.2 Class Methods

The class will provide these shared functions:

1. `how_many()`: Reports the total number of particles
2. `__str__`: Creates a readable description of a particle's properties

11.3.3 Instance Properties

Each individual particle object will have:

1. Radius (R)
2. Position history (x and y coordinates)
3. Unique identifier (index)
4. Diffusion coefficient ($D = k_B T/(6\pi\eta R)$)

11.3.4 Instance Methods

Each particle will be able to:

1. `sim_trajectory()`: Generate a complete motion path
2. `update(dt)`: Calculate one step of Brownian motion
3. `get_trajectory()`: Return its movement history
4. `get_D()`: Provide its diffusion coefficient

```
#| autorun: false
# Class definition
class Colloid:

    # A class variable, counting the number of Colloids
    number = 0
    f = 2.2e-19 # this is k_B T/(6 pi eta) in m^3/s

    # constructor
    def __init__(self,R, x0=0, y0=0):
        # add initialisation code here
        self.R=R
        self.x=[x0]
        self.y=[y0]
        Colloid.number=Colloid.number+1
        self.index=Colloid.number
        self.D=Colloid.f/self.R

    def get_D(self):
        return(self.D)

    def sim_trajectory(self,N,dt):
        for i in range(N):
            self.update(dt)

    def update(self,dt):
        self.x.append(self.x[-1]+np.random.normal(0.0, np.sqrt(2*self.D*dt)))
        self.y.append(self.y[-1]+np.random.normal(0.0, np.sqrt(2*self.D*dt)))
        return(self.x[-1],self.y[-1])

    def get_trajectory(self):
        return(pd.DataFrame({'x':self.x,'y':self.y}))

    # class method accessing a class variable
    @classmethod
    def how_many(cls):
        return(Colloid.number)

    # insert something that prints the particle position in a formatted way when printing
    def __str__(self):
        return("I'm a particle with radius R={0:0.3e} at x={1:0.3e},y={2:0.3e}.".format(self.R, self.x[
```

> **Note**
>
> Note that the function `sim_trajectory` is actually calling the function `update` of the same object to generate the whole trajectory at once.

11.4 Simulating

With the help of this Colloid class, we would like to carry out simulations of Brownian motion of multiple particles. The simulations shall

- take n=200 particles
- have N=200 trajectory points each
- start all at 0,0
- particle objects should be stored in a list p_list

```
#| autorun: false
N=200 # the number of trajectory points
n=200 # the number of particles

p_list=[]
dt=0.05

# creating all objects
for i in range(n):
    p_list.append(Colloid(1e-6))


for (index,p) in enumerate(p_list):
    p.sim_trajectory(N,dt)
```

```
#| autorun: false
print(p_list[42])
```

11.5 Plotting the trajectories

The next step is to plot all the trajectories.

```
#| autorun: false
# we take real world diffusion coefficients so scale up the data to avoid nasty exponentials
scale=1e6

plt.figure(figsize=(4,4))

[plt.plot(np.array(p.x[:])*scale,np.array(p.y[:])*scale,'k-',alpha=0.1,lw=1) for p in p_list]
plt.xlim(-10,10)
plt.ylim(-10,10)
plt.xlabel('x [µm]')
plt.ylabel('y [µm]')
plt.tight_layout()
plt.show()
```

11.6 Characterizing the Brownian motion

Now that we have a number of trajectories, we can analyze the motion of our Brownian particles.

11.6.1 Calculate the particle speed

One way is to calculate its speed by measuring how far it traveled within a certain time $n\,dt$, where dt is the timestep of out simulation. We can do that as

$$v(ndt) = \frac{< \sqrt{(x_{i+n} - x_i)^2 + (y_{i+n} - y_i)^2} >}{n\,dt} \tag{11.1}$$

The angular brackets on the top take care of the fact that we can measure the distance traveled within a certain time $n\,dt$ several times along a trajectory.

These values can be used to calculate a mean speed. Note that there is not an equal amount of data pairs for all separations available. For $n = 1$ there are 5 distances available. For $n = 5$, however, only 1. This changes the statistical accuracy of the mean.

```
#| autorun: false
time=np.array(range(1,N))*dt

plt.figure(figsize=(4,4))
for j in range(100):
    t=p_list[j].get_trajectory()
    md=[np.mean(np.sqrt(t.x.diff(i)**2+t.y.diff(i)**2)) for i in range(1,N)]
    md=md/time
    plt.plot(time,md,alpha=0.4)

plt.ylabel('speed [m/s]')
plt.xlabel('time [s]')
plt.tight_layout()
plt.show()
```

The result of this analysis shows, that each particle has an apparent speed which seems to increase with decreasing time of observation or which decreases with increasing time. This would mean that there is some friction at work, which slows down the particle in time, but this is apparently not true. Also an infinite speed at zero time appears to be unphysical. The correct answer is just that the speed is no good measure to characterize the motion of a Brownian particle.

11.6.2 Calculate the particle mean squared displacement

A better way to characterize the motion of a Brownian particle is the mean squared displacement, as we have already mentioned it in previous lectures. We may compare our simulation now to the theoretical prediction, which is

$$\langle \Delta r^2(t) \rangle = 2dDt \tag{11.2}$$

where d is the dimension of the random walk, which is $d = 2$ in our case.

```
#| autorun: false
time=np.array(range(1,N))*dt

plt.figure(figsize=(4,4))
for j in range(100):
    t=p_list[j].get_trajectory()
    msd=[np.mean(t.x.diff(i).dropna()**2+t.y.diff(i).dropna()**2) for i in range(1,N)]
    plt.plot(time,msd,alpha=0.4)


plt.plot(time, 4*p_list[0].D*time,'k--',lw=2,label='theory')
plt.legend()
```

```
plt.xlabel('time [s]')
plt.ylabel('msd $[m^2/s]$')
plt.tight_layout()
plt.show()
```

The results show that the mean squared displacement of the individual particles follows *on average* the theoretical predictions of a linear growth in time. That means, we are able to read the diffusion coefficient from the slope of the MSD of the individual particles if recorded in a simulation or an experiment.

Yet, each individual MSD is deviating strongly from the theoretical prediction especially at large times. This is due to the fact mentioned earlier that our simulation (or experimental) data only has a limited number of data points, while the theoretical prediction is made for the limit of infinite data points.

> ⚠ Analysis of MSD data
>
> Single particle tracking, either in the experiment or in numerical simulations can therefore only deliver an estimate of the diffusion coefficient and care should be taken when using the whole MSD to obtain the diffusion coefficient. One typically uses only a short fraction of the whole MSD data at short times.

Part VI

Lecture 6

Chapter 12

Input and output

To try out all of the functions of todays notebook, we will need to use the embedded JupyterLite notebook. To use the notebook, click ob `File -> Open from URL` and paste the following link into the input field:

`https://raw.githubusercontent.com/fcichos/EMPP24/refs/heads/main/seminars/1_input_output.ipynb`

To download the data files, click ob `File -> Open from URL` and paste the following links into the input field:

`https://raw.githubusercontent.com/fcichos/EMPP24/refs/heads/main/seminars/MyData.txt`

`https://raw.githubusercontent.com/fcichos/EMPP24/refs/heads/main/seminars/2018-04-11_sds011_sensor_1225`

`https://raw.githubusercontent.com/fcichos/EMPP24/refs/heads/main/seminars/2018-04-12_sds011_sensor_1225`

You should then have 3 data files and one notebook. You can then go into fullscreen mode.

Chapter 13

Curve fitting

Let's take a break from physics-related topics and explore another crucial area: curve fitting. We'll focus on demonstrating how to apply the least-squares method to fit a quadratic function with three parameters to experimental data. It's worth noting that this approach can be applied to more complex functions or even simpler linear models.

Before diving into the fitting process, it's essential to consider how to best estimate your model parameters. In some cases, you may be able to derive explicit estimators for the parameters, which can simplify the fitting procedure. Therefore, it's advisable to carefully consider your approach before beginning the actual fitting process.

For those who want to delve deeper into this subject, you might find it interesting to explore concepts like maximum likelihood estimation. This method offers an alternative approach to parameter estimation and can provide valuable insights in certain scenarios.

```
#| edit: false
#| echo: false
#| execute: true

import numpy as np
import io
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
plt.rcParams.update({'font.size': 18})
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

# default values for plotting
plt.rcParams.update({'font.size': 12,
                     'lines.linewidth': 1,
                     'lines.markersize': 5,
                     'axes.labelsize': 11,
                     'xtick.labelsize' : 10,
                     'ytick.labelsize' : 10,
                     'xtick.top' : True,
                     'xtick.direction' : 'in',
                     'ytick.right' : True,
                     'ytick.direction' : 'in',})

def get_size(w,h):
      return((w/2.54,h/2.54))
```

```
<IPython.core.display.HTML object>
```

```
#| edit: false
#| echo: false
#| execute: true

import pandas as pd

data = {
    'x': [0.00000000000000000e+00, 1.11111111111111049e-01, 2.22222222222222099e-01, 3.3333333333333
    'y': [9.91683920405706742 5e-01, 1.18366784044016171 2e+00, 1.31086214896105701 7e+00, 1.19317486726 59
    'error': [5.33281879919848197 9e-02, 5.33955964674283842 2e-02, 5.36678332638267224 8e-02, 5.43509749 3
}
df = pd.DataFrame(data)

x_data=df.x.values
y_data=df.y.values
err=df.error.values
```

13.1 Idea

In experimental physics, we often collect data points to understand the underlying physical phenomena. This process involves fitting a mathematical model to the experimental data.

The data typically comes as a series of paired points:

x-data	y-data
x_1	y_1
x_2	y_2
...	...
x_N	y_N

Each point $\{x_i, y_i\}$ may represent the result of multiple independent measurements. For instance, y_1 could be the mean of several measurements $y_{1,j}$:

$$y_1 = \frac{1}{N}\sum_{j=1}^{N} y_{1,j}$$

When these measurements have an uncertainty σ for individual readings, the sum of all measurements has a variance of $N\sigma^2$ and a standard deviation of $\sqrt{N}\sigma$. Consequently, the mean value has an associated error (standard deviation) known as the Standard Error of the Mean (SEOM):

$$\sigma_{SEOM} = \frac{\sigma}{\sqrt{N}}$$

This SEOM is crucial in physics measurements.

It's also important to note the definition of variance:

$$\sigma_1^2 = \frac{1}{N}\sum_{j=1}^{N}(y_{1,j} - y_1)^2$$

This statistical framework forms the basis for analyzing experimental data and fitting mathematical models to understand the underlying physics.

13.2 Least squares

In experimental physics, we often collect data points to understand the underlying physical phenomena. To make sense of this data, we fit a mathematical model to it. One common method for fitting data is the least squares method.

13.2.1 Why use least squares fitting?

The goal of least squares fitting is to find the set of parameters for our model that best describes the data. This is done by minimizing the differences (or residuals) between the observed data points and the model's predictions.

13.2.2 Gaussian uncertainty and probability

When we take measurements, there is always some uncertainty. Often, this uncertainty can be modeled using a Gaussian (normal) distribution. This distribution is characterized by its mean (average value) and standard deviation (a measure of the spread of the data).

If we describe our data with a model function, which delivers a function value $f(x_i, a)$ for a set of parameters a at the position x_i, the Gaussian uncertainty dictates a probability of finding a data value y_i:

$$p_{y_i} = \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(y_i - f(x_i, a))^2}{2\sigma_i^2}\right) \tag{13.1}$$

Here, σ_i represents the uncertainty in the measurement y_i.

13.2.3 Combining probabilities for multiple data points

To understand how well our model fits all the data points, we need to consider the combined probability of observing all the data points. This is done by multiplying the individual probabilities:

$$p(y_1, \ldots, y_N) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(y_i - f(x_i, a))^2}{2\sigma_i^2}\right) \tag{13.2}$$

13.2.4 Maximizing the joint probability

The best fit of the model to the data is achieved when this joint probability is maximized. To simplify the calculations, we take the logarithm of the joint probability:

$$\ln(p(y_1, \ldots, y_N)) = -\frac{1}{2}\sum_{i=1}^{N}\left(\frac{y_i - f(x_i, a)}{\sigma_i}\right)^2 - \sum_{i=1}^{N}\ln\left(\sigma_i\sqrt{2\pi}\right) \tag{13.3}$$

The first term on the right side (except the factor 1/2) is the least squared deviation, also known as χ^2:

$$\chi^2 = \sum_{i=1}^{N}\left(\frac{y_i - f(x_i, a)}{\sigma_i}\right)^2 \tag{13.4}$$

The second term is just a constant value given by the uncertainties of our experimental data.

13.3 Data

Let's have a look at the meaning of this equation. Let's assume we measure the trajectory of a ball that has been thrown at an angle α with an initial velocity v_0. We have collected data points by measuring the height of the ball above the ground at equally spaced distances from the throwing person.

The table above shows the measured data points y_i at the position x_i with the associated uncertainties σ_i.

We can plot the data and expect, of course, a parabola. Therefore, we model our experimental data with a parabola like

$$y = ax^2 + bx + c \tag{13.5}$$

where the parameter a must be negative since the parabola is inverted.

I have created an interactive plot with an interact widget, as this allows you to play around with the parameters. The value of χ^2 is also included in the legend, so you can get an impression of how good your fit of the data is.

```
//| echo: false
viewof aSlider = Inputs.range([-4, 0], { label: "a", step: 0.01, value: -1.7 });
viewof bSlider = Inputs.range([-2, 2], { label: "b", step: 0.01, value: 1.3 });
viewof cSlider = Inputs.range([-2, 2], { label: "c", step: 0.01, value: 1.0 });
```

```
//| echo: false
//| fig-align: center
filtered = transpose(data);
// Create the plot

xValues = Array.from({ length: 100 }, (_, i) => i / 100);
parabolaData = xValues.map(x => ({ x, y: parabola(x, aSlider, bSlider, cSlider) }));


parabola = (x, a, b, c) => a * x**2 + b * x + c

calculateChiSquared = (data, a, b, c) => {
  let chisq = 0
  let x= data.map(d => d.x)
  let y= data.map(d => d.y)
  let err= data.map(d => d.error)
  for (let i = 0; i < x.length; i++) {
    let y_model = parabola(x[i], a, b, c)
    chisq += ((y[i] - y_model) / err[i])**2
  }
  return chisq
}

chisq = calculateChiSquared(filtered, aSlider, bSlider, cSlider)

Plot.plot({
  marks: [
    Plot.dot(filtered, { x: "x", y: "y" }),
    Plot.ruleY(filtered, { x: "x", y1: d => d.y - d.error, y2: d => d.y + d.error }),
    Plot.line(parabolaData, { x: "x", y: "y" }),
    Plot.text([{ x: 0.8, y: 1.5, label: `χ²: ${chisq.toFixed(2)}` }], {
          x: "x",
          y: "y",
          text: "label",
          dy: -10, // Adjust vertical position if needed
```

```
          fill: "black", // Set text color
          fontSize: 16
        }),
    Plot.frame()
  ],
  x: {
    label: "X Axis",
    labelAnchor: "center",
    labelOffset: 35,
    grid: true,
    tickFormat: ".2f", // Format ticks to 2 decimal places
    domain: [0, 1]
  },
  y: {
    label: "Y Axis",
    grid: true,
    tickFormat: ".2f", // Format ticks to 2 decimal places
    labelAnchor: "center",  // Center the label on its axis
    labelAngle: -90,
    labelOffset: 60,
    domain: [0, 2],
  },
  width: 400,
  height: 400,
  marginLeft: 100,
  marginBottom: 40,
  style: {
    fontSize: "14px",          // This sets the base font size
    "axis.label": {
      fontSize: "18px",        // This sets the font size for axis labels
      fontWeight: "bold"       // Optionally make it bold
    },
    "axis.tick": {
      fontSize: "14px"         // This sets the font size for tick labels
    }
  },
})
```

```
#| autorun: false
#| fig-align: center
import numpy as np
import io
import pandas as pd
import matplotlib.pyplot as plt
plt.figure(figsize=get_size(8,8))
plt.errorbar(df.x,df.y,yerr=df.error,marker='o',fmt="none",color='k')

plt.scatter(df.x,df.y,marker='o',color='k')
plt.xlabel('x- position')
plt.ylabel('y- position')
plt.tight_layout()
plt.show()
```

We have that troubling point at the right edge with a large uncertainty. However, since the value of χ^2 divides the deviation by the uncertainty σ_i, the weight for this point overall in the χ^2 is smaller than for the other points.

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - f(x_i, a)}{\sigma_i} \right)^2 \tag{13.6}$$

You may simply check the effect by changing the uncertainty of the last data points in the error array.

13.4 Least square fitting

To find the best fit of the model to the experimental data, we use the least squares method. This method minimizes the sum of the squared differences between the observed data points and the model's predictions.

Mathematically, we achieve this by minimizing the least squares, i.e., finding the parameters a that minimize the following expression:

$$\frac{d\chi^2}{da} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \frac{df(x_i, a)}{da} [y_i - f(x_i, a)] = 0 \tag{13.7}$$

This kind of least squares minimization is done by fitting software using different types of algorithms.

13.4.1 Fitting with SciPy

Let's do some fitting using the `SciPy` library, which is a powerful tool for scientific computing in Python. We will use the `curve_fit` method from the `optimize` sub-module of `SciPy`.

First, we need to define the model function we would like to fit to the data. In this case, we will use our parabola function:

```
#| autorun: false
def parabola(x, a, b, c):
    return a * x**2 + b * x + c
```

Next, we need to provide initial guesses for the parameters. These initial guesses help the fitting algorithm start the search for the optimal parameters:

```
#| autorun: false
init_guess = [-1, 1, 1]
```

We then call the `curve_fit` function to perform the fitting:

```
#| autorun: false
fit = curve_fit(parabola, x_data, y_data, sigma=err, p0=init_guess, absolute_sigma=True)
```

> **`curve_fit` Function**
>
> The `curve_fit` function is used to fit a model function to data. It finds the optimal parameters for the model function that minimize the sum of the squared residuals between the observed data and the model's predictions.

13.4.2 Parameters

1. `parabola`:
   - This is the model function that you want to fit to the data. In this case, `parabola` is a function that represents a quadratic equation of the form (y = ax^2 + bx + c).
2. `x_data`:
   - This is the array of independent variable data points (the x-values).
3. `y_data`:
   - This is the array of dependent variable data points (the y-values).

4. `sigma=err`:
   - This parameter specifies the uncertainties (standard deviations) of the y-data points. The `err` array contains the uncertainties for each y-data point. These uncertainties are used to weight the residuals in the least squares optimization.
5. `p0=init_guess`:
   - This parameter provides the initial guesses for the parameters of the model function. The `init_guess` array contains the initial guesses for the parameters (a), (b), and (c). Providing good initial guesses can help the optimization algorithm converge more quickly and accurately.
6. `absolute_sigma=True`:
   - This parameter indicates whether the provided `sigma` values are absolute uncertainties. If `absolute_sigma` is set to `True`, the `sigma` values are treated as absolute uncertainties. If `absolute_sigma` is set to `False`, the `sigma` values are treated as relative uncertainties, and the covariance matrix of the parameters will be scaled accordingly.

13.4.3 Return Value

The `curve_fit` function returns two values:

1. `popt`:
   - An array containing the optimal values for the parameters of the model function that minimize the sum of the squared residuals.
2. `pcov`:
   - The covariance matrix of the optimal parameters. The diagonal elements of this matrix provide the variances of the parameter estimates, and the off-diagonal elements provide the covariances between the parameter estimates.

The `fit` variable contains the results of the fitting process. It is composed of various results, which we can split into the fitted parameters and the covariance matrix:

```
#| autorun: false
ans, cov = fit
fit_a, fit_b, fit_c = ans
```

The `ans` variable contains the fitted parameters `fit_a`, `fit_b`, and `fit_c`, while the `cov` variable contains the covariance matrix. Let's have a look at the fit and the χ^2 value first:

```
#| autorun: false
fit_a, fit_b, fit_c
```

We can then plot the fitted curve along with the original data points and the χ^2 value:

```
#| autorun: false
plt.figure(figsize=get_size(8,8))
chisq = (((y_data - parabola(x_data, fit_a, fit_b, fit_c)) / err)**2).sum()
plt.plot(x_data, parabola(x_data, fit_a, fit_b, fit_c), label=r'$\chi^2$={0:6.3f}'.format(chisq))
plt.errorbar(x_data, y_data, yerr=err, marker='o', fmt="none", color='k')
plt.scatter(x_data, y_data, marker='.', color='k')
plt.xlabel('x-position')
plt.ylabel('y-position')
plt.legend()
plt.tight_layout()
plt.show()
```

13.4.4 χ-squared value

The value of χ^2 gives you a measure of the quality of the fit. We can judge the quality by calculating the expectation value of χ^2:

$$\langle \chi^2 \rangle = \sum_{i=1}^{N} \frac{\langle (y_i - f(x_i, a))^2 \rangle}{\sigma_i^2} = \sum_{i=1}^{N} \frac{\sigma_i^2}{\sigma_i^2} = N \tag{13.8}$$

So, the mean of the least squared deviation increases with the number of data points. Therefore:

- $\chi^2 \gg N$ means that the fit is bad.
- $\chi^2 < N$ means that the uncertainties are wrong.

The first case may occur if you don't have a good fit to your data, for example, if you are using the wrong model. The second case typically occurs if you don't have accurate estimates of the uncertainties and you assume all uncertainties to be constant.

It is really important to have a good estimate of the uncertainties and to include them in your fit. If you include the uncertainties in your fit, it is called a `weighted fit`. If you don't include the uncertainties (meaning you keep them constant), it is called an `unweighted fit`.

For our fit above, we obtain a χ^2 which is on the order of $N = 10$, which tells you that I have created the data with reasonable accuracy.

13.4.5 Residuals

Another way to assess the quality of the fit is by looking at the residuals. Residuals are defined as the deviation of the data from the model for the best fit:

$$r_i = y_i - f(x_i, a) \tag{13.9}$$

The residuals can also be expressed as the percentage of the deviation of the data from the fit:

$$r_i = 100 \left(\frac{y_i - f(x_i, a)}{y_i} \right) \tag{13.10}$$

13.4.6 Importance of Residuals

Residuals are important because they provide insight into how well the model fits the data. If the residuals show only statistical fluctuations around zero, then the fit and likely also the model are good. However, if there are systematic patterns in the residuals, it may indicate that the model is not adequately capturing the underlying relationship in the data.

13.4.7 Visualizing Residuals

Let's visualize the residuals to better understand their distribution. We will plot the residuals as a function of the independent variable x.

```
#| autorun: false
plt.figure(figsize=get_size(8,8))
chisq = (((y_data - parabola(x_data, fit_a, fit_b, fit_c)) / err)**2).sum()
plt.scatter(x_data, 100 * (y_data - parabola(x_data, fit_a, fit_b, fit_c)) / y_data, marker='o', color=
plt.xlabel('x-position')
plt.ylabel('residuals [%]')
plt.tight_layout()
plt.show()
```

> **i** Common Patterns in Residuals
>
> **Random Fluctuations Around Zero**:
> - If the residuals are randomly scattered around zero, it suggests that the model is a good fit for the data.

Systematic Patterns:
- If the residuals show a systematic pattern (e.g., a trend or periodicity), it may indicate that the model is not capturing some aspect of the data. This could suggest the need for a more complex model.

Increasing or Decreasing Trends:
- If the residuals increase or decrease with x, it may indicate heteroscedasticity (non-constant variance) or that a different functional form is needed.

13.5 Covariance Matrix

In the previous sections, we discussed how to fit a model to experimental data and assess the quality of the fit using residuals. Now, let's take a closer look at the uncertainties in the fit parameters and how they are related to each other. This is where the covariance matrix comes into play.

13.5.1 Purpose of the Covariance Matrix

The covariance matrix provides important information about the uncertainties in the fit parameters and how these uncertainties are related to each other. It helps us understand the precision of the parameter estimates and whether the parameters are independent or correlated.

13.5.2 Understanding Covariance

Covariance is a measure of how much two random variables change together. If the covariance between two variables is positive, it means that they tend to increase or decrease together. If the covariance is negative, it means that one variable tends to increase when the other decreases. If the covariance is zero, it means that the variables are independent.

13.5.3 Covariance Matrix in Curve Fitting

When we fit a model to data, we obtain estimates for the parameters of the model. These estimates have uncertainties due to the measurement errors in the data. The covariance matrix quantifies these uncertainties and the relationships between them.

For a model with three parameters (a, b, c), the covariance matrix is a 3×3 matrix that looks like this:

$$\text{cov}(p_i, p_j) = \begin{bmatrix} \sigma_{aa}^2 & \sigma_{ab}^2 & \sigma_{ac}^2 \\ \sigma_{ba}^2 & \sigma_{bb}^2 & \sigma_{bc}^2 \\ \sigma_{ca}^2 & \sigma_{cb}^2 & \sigma_{cc}^2 \end{bmatrix} \tag{13.11}$$

The diagonal elements provide the variances (squared uncertainties) of the fit parameters, while the off-diagonal elements describe the covariances between the parameters.

13.5.4 Example

Let's calculate the covariance matrix for our fitted model and interpret the results.

```
#| autorun: false
# Calculate the covariance matrix
fit = curve_fit(parabola, x_data, y_data, sigma=err, p0=init_guess, absolute_sigma=True)
ans, cov = fit

# Print the covariance matrix
print("Covariance matrix:")
print(cov)
```

13.5.5 Interpreting the Covariance Matrix

The covariance matrix provides valuable information about the uncertainties in the fit parameters:

- **Diagonal Elements**: The diagonal elements represent the variances of the parameters. The square root of these values gives the standard deviations (uncertainties) of the parameters.
- **Off-Diagonal Elements**: The off-diagonal elements represent the covariances between the parameters. If these values are large, it indicates that the parameters are correlated.

13.5.6 Generating Synthetic Data

To better understand the covariance matrix, let's generate synthetic data and fit the model to each dataset. This will help us visualize the uncertainties in the parameters.

```
#| autorun: false
def data(x, a, b, c):
    y = a * x**2 + b * x + c
    err = [np.random.normal() for _ in range(len(x))]
    err = y * err * 0.05
    return y + err
```

```
#| autorun: false
x = np.linspace(0, 1, 10)
ym = np.zeros(10)
plt.figure(figsize=get_size(8, 8))

for _ in range(10):
    y = data(x, -2.52, 1.6971755, 1)
    ym += y
    p, cov = curve_fit(parabola, x, y, sigma=err, p0=init_guess, absolute_sigma=True)
    plt.scatter(x, y, color='k', alpha=0.1)
    xf = np.linspace(0, 1, 100)
    plt.plot(xf, parabola(xf, p[0], p[1], p[2]), alpha=0.2)

plt.scatter(x, ym / 10, color='b')
plt.xlabel('x-position')
plt.ylabel('y-position')
plt.tight_layout()
plt.show()
```

13.5.7 Correlation Matrix

To better understand the relationships between the parameters, we can normalize the covariance matrix to obtain the correlation matrix. The correlation matrix has values between -1 and 1, where 1 indicates perfect positive correlation, -1 indicates perfect negative correlation, and 0 indicates no correlation.

```
#| autorun: false
# Calculate the correlation matrix
s = np.diag(cov)
R = np.zeros([3, 3])
for i in range(3):
    for j in range(3):
        R[i, j] = cov[i, j] / np.sqrt(s[i] * s[j])

# Print the correlation matrix
print("Correlation matrix:")
print(R)
```

13.5.8 Visualizing the Covariance and Correlation

Let's visualize the covariance and correlation between the parameters using scatter plots.

```
#| autorun: false
# Generate synthetic data and fit the model
a = []
b = []
c = []
x = np.linspace(0, 1, 10)
for _ in range(100):
    y = data(x, -2.52, 1.6971755, 1)
    p, cov = curve_fit(parabola, x, y, sigma=err, p0=init_guess, absolute_sigma=True)
    a.append(p[0])
    b.append(p[1])
    c.append(p[2])

# Plot the correlation between parameters a and b
plt.figure(figsize=get_size(8, 8))
plt.scatter(a, b, alpha=0.2)
plt.xlabel('Parameter a')
plt.ylabel('Parameter b')
plt.title('Correlation between Parameters a and b')
plt.tight_layout()
plt.show()
```

By examining the covariance and correlation matrices, we can gain a deeper understanding of the uncertainties in the fit parameters and how they are related to each other.

13.5.9 Improving the Model

If we find that the parameters are highly correlated, we might want to find a better model containing more independent parameters. For example, we can write down a different model:

$$y = a(x - b)^2 + c \tag{13.12}$$

This model also contains three parameters, but the parameter b directly refers to the maximum of our parabola, while the parameter a denotes its curvature.

```
#| autorun: false
def newmodel(x, a, b, c):
    return a * (x - b)**2 + c
```

```
#| autorun: false
fit = curve_fit(newmodel, x_data, y_data, sigma=err, p0=init_guess, absolute_sigma=True)
```

```
#| autorun: false
ans, cov = fit
```

```
#| autorun: false
s = np.diag(cov)
R = np.zeros([3, 3])
for i in range(3):
    for j in range(3):
        R[i, j] = cov[i, j] / np.sqrt(s[i] * s[j])
```

We see from the covariance matrix that the new model has a smaller correlation of the parameters with each other.

```
#| autorun: false
print(cov)
```

This is also expressed by our correlation matrix.

```
#| autorun: false
print(R)
```

By examining the covariance and correlation matrices, we can gain valuable insights into the uncertainties in the fit parameters and how to improve our model.

Part VII

Lecture 7

Chapter 14

Curve fitting

Let's take a break from physics-related topics and explore another crucial area: curve fitting. We'll focus on demonstrating how to apply the least-squares method to fit a quadratic function with three parameters to experimental data. It's worth noting that this approach can be applied to more complex functions or even simpler linear models.

Before diving into the fitting process, it's essential to consider how to best estimate your model parameters. In some cases, you may be able to derive explicit estimators for the parameters, which can simplify the fitting procedure. Therefore, it's advisable to carefully consider your approach before beginning the actual fitting process.

For those who want to delve deeper into this subject, you might find it interesting to explore concepts like maximum likelihood estimation. This method offers an alternative approach to parameter estimation and can provide valuable insights in certain scenarios.

```
#| edit: false
#| echo: false
#| execute: true

import numpy as np
import io
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
plt.rcParams.update({'font.size': 18})
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

# default values for plotting
plt.rcParams.update({'font.size': 12,
                     'lines.linewidth': 1,
                     'lines.markersize': 5,
                     'axes.labelsize': 11,
                     'xtick.labelsize' : 10,
                     'ytick.labelsize' : 10,
                     'xtick.top' : True,
                     'xtick.direction' : 'in',
                     'ytick.right' : True,
                     'ytick.direction' : 'in',})

def get_size(w,h):
      return((w/2.54,h/2.54))
```

```
<IPython.core.display.HTML object>
```

```
#| edit: false
#| echo: false
#| execute: true

import pandas as pd

data = {
    'x': [0.00000000000000000e+00, 1.11111111111111049e-01, 2.22222222222222099e-01, 3.3333333333333
    'y': [9.91683920405706742 5e-01, 1.18366784044016171 2e+00, 1.31086214896105701 7e+00, 1.19317486726 59
    'error': [5.33281879919848197 9e-02, 5.33955964674283842 2e-02, 5.36678332638267224 8e-02, 5.43509749 3
}
df = pd.DataFrame(data)

x_data=df.x.values
y_data=df.y.values
err=df.error.values
```

14.1 Idea

In experimental physics, we often collect data points to understand the underlying physical phenomena. This process involves fitting a mathematical model to the experimental data.

The data typically comes as a series of paired points:

x-data	y-data
x_1	y_1
x_2	y_2
...	...
x_N	y_N

Each point $\{x_i, y_i\}$ may represent the result of multiple independent measurements. For instance, y_1 could be the mean of several measurements $y_{1,j}$:

$$y_1 = \frac{1}{N}\sum_{j=1}^{N} y_{1,j}$$

When these measurements have an uncertainty σ for individual readings, the sum of all measurements has a variance of $N\sigma^2$ and a standard deviation of $\sqrt{N}\sigma$. Consequently, the mean value has an associated error (standard deviation) known as the Standard Error of the Mean (SEOM):

$$\sigma_{SEOM} = \frac{\sigma}{\sqrt{N}}$$

This SEOM is crucial in physics measurements.

It's also important to note the definition of variance:

$$\sigma_1^2 = \frac{1}{N}\sum_{j=1}^{N}(y_{1,j} - y_1)^2$$

This statistical framework forms the basis for analyzing experimental data and fitting mathematical models to understand the underlying physics.

14.2 Least squares

In experimental physics, we often collect data points to understand the underlying physical phenomena. To make sense of this data, we fit a mathematical model to it. One common method for fitting data is the least squares method.

14.2.1 Why use least squares fitting?

The goal of least squares fitting is to find the set of parameters for our model that best describes the data. This is done by minimizing the differences (or residuals) between the observed data points and the model's predictions.

14.2.2 Gaussian uncertainty and probability

When we take measurements, there is always some uncertainty. Often, this uncertainty can be modeled using a Gaussian (normal) distribution. This distribution is characterized by its mean (average value) and standard deviation (a measure of the spread of the data).

If we describe our data with a model function, which delivers a function value $f(x_i, a)$ for a set of parameters a at the position x_i, the Gaussian uncertainty dictates a probability of finding a data value y_i:

$$p_{y_i} = \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(y_i - f(x_i, a))^2}{2\sigma_i^2}\right) \tag{14.1}$$

Here, σ_i represents the uncertainty in the measurement y_i.

14.2.3 Combining probabilities for multiple data points

To understand how well our model fits all the data points, we need to consider the combined probability of observing all the data points. This is done by multiplying the individual probabilities:

$$p(y_1, \ldots, y_N) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(y_i - f(x_i, a))^2}{2\sigma_i^2}\right) \tag{14.2}$$

14.2.4 Maximizing the joint probability

The best fit of the model to the data is achieved when this joint probability is maximized. To simplify the calculations, we take the logarithm of the joint probability:

$$\ln(p(y_1, \ldots, y_N)) = -\frac{1}{2}\sum_{i=1}^{N}\left(\frac{y_i - f(x_i, a)}{\sigma_i}\right)^2 - \sum_{i=1}^{N}\ln\left(\sigma_i\sqrt{2\pi}\right) \tag{14.3}$$

The first term on the right side (except the factor 1/2) is the least squared deviation, also known as χ^2:

$$\chi^2 = \sum_{i=1}^{N}\left(\frac{y_i - f(x_i, a)}{\sigma_i}\right)^2 \tag{14.4}$$

The second term is just a constant value given by the uncertainties of our experimental data.

14.3 Data

Let's have a look at the meaning of this equation. Let's assume we measure the trajectory of a ball that has been thrown at an angle α with an initial velocity v_0. We have collected data points by measuring the height of the ball above the ground at equally spaced distances from the throwing person.

The table above shows the measured data points y_i at the position x_i with the associated uncertainties σ_i.

We can plot the data and expect, of course, a parabola. Therefore, we model our experimental data with a parabola like

$$y = ax^2 + bx + c \tag{14.5}$$

where the parameter a must be negative since the parabola is inverted.

I have created an interactive plot with an interact widget, as this allows you to play around with the parameters. The value of χ^2 is also included in the legend, so you can get an impression of how good your fit of the data is.

```
//| echo: false
viewof aSlider = Inputs.range([-4, 0], { label: "a", step: 0.01, value: -1.7 });
viewof bSlider = Inputs.range([-2, 2], { label: "b", step: 0.01, value: 1.3 });
viewof cSlider = Inputs.range([-2, 2], { label: "c", step: 0.01, value: 1.0 });
```

```
//| echo: false
//| fig-align: center
filtered = transpose(data);
// Create the plot

xValues = Array.from({ length: 100 }, (_, i) => i / 100);
parabolaData = xValues.map(x => ({ x, y: parabola(x, aSlider, bSlider, cSlider) }));


parabola = (x, a, b, c) => a * x**2 + b * x + c

calculateChiSquared = (data, a, b, c) => {
  let chisq = 0
  let x= data.map(d => d.x)
  let y= data.map(d => d.y)
  let err= data.map(d => d.error)
  for (let i = 0; i < x.length; i++) {
    let y_model = parabola(x[i], a, b, c)
    chisq += ((y[i] - y_model) / err[i])**2
  }
  return chisq
}

chisq = calculateChiSquared(filtered, aSlider, bSlider, cSlider)

Plot.plot({
  marks: [
    Plot.dot(filtered, { x: "x", y: "y" }),
    Plot.ruleY(filtered, { x: "x", y1: d => d.y - d.error, y2: d => d.y + d.error }),
    Plot.line(parabolaData, { x: "x", y: "y" }),
    Plot.text([{ x: 0.8, y: 1.5, label: `χ²: ${chisq.toFixed(2)}` }], {
          x: "x",
          y: "y",
          text: "label",
          dy: -10, // Adjust vertical position if needed
```

```
          fill: "black", // Set text color
          fontSize: 16
        }),
    Plot.frame()
  ],
  x: {
    label: "X Axis",
    labelAnchor: "center",
    labelOffset: 35,
    grid: true,
    tickFormat: ".2f", // Format ticks to 2 decimal places
    domain: [0, 1]
  },
  y: {
    label: "Y Axis",
    grid: true,
    tickFormat: ".2f", // Format ticks to 2 decimal places
    labelAnchor: "center",  // Center the label on its axis
    labelAngle: -90,
    labelOffset: 60,
    domain: [0, 2],
  },
  width: 400,
  height: 400,
  marginLeft: 100,
  marginBottom: 40,
  style: {
    fontSize: "14px",          // This sets the base font size
    "axis.label": {
      fontSize: "18px",        // This sets the font size for axis labels
      fontWeight: "bold"       // Optionally make it bold
    },
    "axis.tick": {
      fontSize: "14px"         // This sets the font size for tick labels
    }
  },
})
```

```
#| autorun: false
#| fig-align: center
import numpy as np
import io
import pandas as pd
import matplotlib.pyplot as plt
plt.figure(figsize=get_size(8,8))
plt.errorbar(df.x,df.y,yerr=df.error,marker='o',fmt="none",color='k')

plt.scatter(df.x,df.y,marker='o',color='k')
plt.xlabel('x- position')
plt.ylabel('y- position')
plt.tight_layout()
plt.show()
```

We have that troubling point at the right edge with a large uncertainty. However, since the value of χ^2 divides the deviation by the uncertainty σ_i, the weight for this point overall in the χ^2 is smaller than for the other points.

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - f(x_i, a)}{\sigma_i} \right)^2 \tag{14.6}$$

You may simply check the effect by changing the uncertainty of the last data points in the error array.

14.4 Least square fitting

To find the best fit of the model to the experimental data, we use the least squares method. This method minimizes the sum of the squared differences between the observed data points and the model's predictions.

Mathematically, we achieve this by minimizing the least squares, i.e., finding the parameters a that minimize the following expression:

$$\frac{d\chi^2}{da} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \frac{df(x_i, a)}{da} [y_i - f(x_i, a)] = 0 \tag{14.7}$$

This kind of least squares minimization is done by fitting software using different types of algorithms.

14.4.1 Fitting with SciPy

Let's do some fitting using the `SciPy` library, which is a powerful tool for scientific computing in Python. We will use the `curve_fit` method from the `optimize` sub-module of `SciPy`.

First, we need to define the model function we would like to fit to the data. In this case, we will use our parabola function:

```
#| autorun: false
def parabola(x, a, b, c):
    return a * x**2 + b * x + c
```

Next, we need to provide initial guesses for the parameters. These initial guesses help the fitting algorithm start the search for the optimal parameters:

```
#| autorun: false
init_guess = [-1, 1, 1]
```

We then call the `curve_fit` function to perform the fitting:

```
#| autorun: false
fit = curve_fit(parabola, x_data, y_data, sigma=err, p0=init_guess, absolute_sigma=True)
```

> **`curve_fit` Function**
>
> The `curve_fit` function is used to fit a model function to data. It finds the optimal parameters for the model function that minimize the sum of the squared residuals between the observed data and the model's predictions.

14.4.2 Parameters

1. `parabola`:
   - This is the model function that you want to fit to the data. In this case, `parabola` is a function that represents a quadratic equation of the form (y = ax^2 + bx + c).
2. `x_data`:
   - This is the array of independent variable data points (the x-values).
3. `y_data`:
   - This is the array of dependent variable data points (the y-values).

4. `sigma=err`:
   - This parameter specifies the uncertainties (standard deviations) of the y-data points. The `err` array contains the uncertainties for each y-data point. These uncertainties are used to weight the residuals in the least squares optimization.
5. `p0=init_guess`:
   - This parameter provides the initial guesses for the parameters of the model function. The `init_guess` array contains the initial guesses for the parameters (a), (b), and (c). Providing good initial guesses can help the optimization algorithm converge more quickly and accurately.
6. `absolute_sigma=True`:
   - This parameter indicates whether the provided `sigma` values are absolute uncertainties. If `absolute_sigma` is set to `True`, the `sigma` values are treated as absolute uncertainties. If `absolute_sigma` is set to `False`, the `sigma` values are treated as relative uncertainties, and the covariance matrix of the parameters will be scaled accordingly.

14.4.3 Return Value

The `curve_fit` function returns two values:

1. `popt`:
   - An array containing the optimal values for the parameters of the model function that minimize the sum of the squared residuals.
2. `pcov`:
   - The covariance matrix of the optimal parameters. The diagonal elements of this matrix provide the variances of the parameter estimates, and the off-diagonal elements provide the covariances between the parameter estimates.

The `fit` variable contains the results of the fitting process. It is composed of various results, which we can split into the fitted parameters and the covariance matrix:

```
#| autorun: false
ans, cov = fit
fit_a, fit_b, fit_c = ans
```

The `ans` variable contains the fitted parameters `fit_a`, `fit_b`, and `fit_c`, while the `cov` variable contains the covariance matrix. Let's have a look at the fit and the χ^2 value first:

```
#| autorun: false
fit_a, fit_b, fit_c
```

We can then plot the fitted curve along with the original data points and the χ^2 value:

```
#| autorun: false
plt.figure(figsize=get_size(8,8))
chisq = (((y_data - parabola(x_data, fit_a, fit_b, fit_c)) / err)**2).sum()
plt.plot(x_data, parabola(x_data, fit_a, fit_b, fit_c), label=r'$\chi^2$={0:6.3f}'.format(chisq))
plt.errorbar(x_data, y_data, yerr=err, marker='o', fmt="none", color='k')
plt.scatter(x_data, y_data, marker='.', color='k')
plt.xlabel('x-position')
plt.ylabel('y-position')
plt.legend()
plt.tight_layout()
plt.show()
```

14.4.4 χ-squared value

The value of χ^2 gives you a measure of the quality of the fit. We can judge the quality by calculating the expectation value of χ^2:

$$\langle \chi^2 \rangle = \sum_{i=1}^{N} \frac{\langle (y_i - f(x_i, a))^2 \rangle}{\sigma_i^2} = \sum_{i=1}^{N} \frac{\sigma_i^2}{\sigma_i^2} = N \tag{14.8}$$

So, the mean of the least squared deviation increases with the number of data points. Therefore:

- $\chi^2 \gg N$ means that the fit is bad.
- $\chi^2 < N$ means that the uncertainties are wrong.

The first case may occur if you don't have a good fit to your data, for example, if you are using the wrong model. The second case typically occurs if you don't have accurate estimates of the uncertainties and you assume all uncertainties to be constant.

It is really important to have a good estimate of the uncertainties and to include them in your fit. If you include the uncertainties in your fit, it is called a `weighted fit`. If you don't include the uncertainties (meaning you keep them constant), it is called an `unweighted fit`.

For our fit above, we obtain a χ^2 which is on the order of $N = 10$, which tells you that I have created the data with reasonable accuracy.

14.4.5 Residuals

Another way to assess the quality of the fit is by looking at the residuals. Residuals are defined as the deviation of the data from the model for the best fit:

$$r_i = y_i - f(x_i, a) \tag{14.9}$$

The residuals can also be expressed as the percentage of the deviation of the data from the fit:

$$r_i = 100 \left(\frac{y_i - f(x_i, a)}{y_i} \right) \tag{14.10}$$

14.4.6 Importance of Residuals

Residuals are important because they provide insight into how well the model fits the data. If the residuals show only statistical fluctuations around zero, then the fit and likely also the model are good. However, if there are systematic patterns in the residuals, it may indicate that the model is not adequately capturing the underlying relationship in the data.

14.4.7 Visualizing Residuals

Let's visualize the residuals to better understand their distribution. We will plot the residuals as a function of the independent variable x.

```
#| autorun: false
plt.figure(figsize=get_size(8,8))
chisq = (((y_data - parabola(x_data, fit_a, fit_b, fit_c)) / err)**2).sum()
plt.scatter(x_data, 100 * (y_data - parabola(x_data, fit_a, fit_b, fit_c)) / y_data, marker='o', color=
plt.xlabel('x-position')
plt.ylabel('residuals [%]')
plt.tight_layout()
plt.show()
```

> **i** Common Patterns in Residuals
>
> **Random Fluctuations Around Zero**:
> - If the residuals are randomly scattered around zero, it suggests that the model is a good fit for the data.

Systematic Patterns:
- If the residuals show a systematic pattern (e.g., a trend or periodicity), it may indicate that the model is not capturing some aspect of the data. This could suggest the need for a more complex model.

Increasing or Decreasing Trends:
- If the residuals increase or decrease with x, it may indicate heteroscedasticity (non-constant variance) or that a different functional form is needed.

14.5 Covariance Matrix

In the previous sections, we discussed how to fit a model to experimental data and assess the quality of the fit using residuals. Now, let's take a closer look at the uncertainties in the fit parameters and how they are related to each other. This is where the covariance matrix comes into play.

14.5.1 Purpose of the Covariance Matrix

The covariance matrix provides important information about the uncertainties in the fit parameters and how these uncertainties are related to each other. It helps us understand the precision of the parameter estimates and whether the parameters are independent or correlated.

14.5.2 Understanding Covariance

Covariance is a measure of how much two random variables change together. If the covariance between two variables is positive, it means that they tend to increase or decrease together. If the covariance is negative, it means that one variable tends to increase when the other decreases. If the covariance is zero, it means that the variables are independent.

14.5.3 Covariance Matrix in Curve Fitting

When we fit a model to data, we obtain estimates for the parameters of the model. These estimates have uncertainties due to the measurement errors in the data. The covariance matrix quantifies these uncertainties and the relationships between them.

For a model with three parameters (a, b, c), the covariance matrix is a 3×3 matrix that looks like this:

$$\text{cov}(p_i, p_j) = \begin{bmatrix} \sigma_{aa}^2 & \sigma_{ab}^2 & \sigma_{ac}^2 \\ \sigma_{ba}^2 & \sigma_{bb}^2 & \sigma_{bc}^2 \\ \sigma_{ca}^2 & \sigma_{cb}^2 & \sigma_{cc}^2 \end{bmatrix} \tag{14.11}$$

The diagonal elements provide the variances (squared uncertainties) of the fit parameters, while the off-diagonal elements describe the covariances between the parameters.

14.5.4 Example

Let's calculate the covariance matrix for our fitted model and interpret the results.

```
#| autorun: false
# Calculate the covariance matrix
fit = curve_fit(parabola, x_data, y_data, sigma=err, p0=init_guess, absolute_sigma=True)
ans, cov = fit

# Print the covariance matrix
print("Covariance matrix:")
print(cov)
```

14.5.5 Interpreting the Covariance Matrix

The covariance matrix provides valuable information about the uncertainties in the fit parameters:

- **Diagonal Elements**: The diagonal elements represent the variances of the parameters. The square root of these values gives the standard deviations (uncertainties) of the parameters.
- **Off-Diagonal Elements**: The off-diagonal elements represent the covariances between the parameters. If these values are large, it indicates that the parameters are correlated.

14.5.6 Generating Synthetic Data

To better understand the covariance matrix, let's generate synthetic data and fit the model to each dataset. This will help us visualize the uncertainties in the parameters.

```
#| autorun: false
def data(x, a, b, c):
    y = a * x**2 + b * x + c
    err = [np.random.normal() for _ in range(len(x))]
    err = y * err * 0.05
    return y + err
```

```
#| autorun: false
x = np.linspace(0, 1, 10)
ym = np.zeros(10)
plt.figure(figsize=get_size(8, 8))

for _ in range(10):
    y = data(x, -2.52, 1.6971755, 1)
    ym += y
    p, cov = curve_fit(parabola, x, y, sigma=err, p0=init_guess, absolute_sigma=True)
    plt.scatter(x, y, color='k', alpha=0.1)
    xf = np.linspace(0, 1, 100)
    plt.plot(xf, parabola(xf, p[0], p[1], p[2]), alpha=0.2)

plt.scatter(x, ym / 10, color='b')
plt.xlabel('x-position')
plt.ylabel('y-position')
plt.tight_layout()
plt.show()
```

14.5.7 Correlation Matrix

To better understand the relationships between the parameters, we can normalize the covariance matrix to obtain the correlation matrix. The correlation matrix has values between -1 and 1, where 1 indicates perfect positive correlation, -1 indicates perfect negative correlation, and 0 indicates no correlation.

```
#| autorun: false
# Calculate the correlation matrix
s = np.diag(cov)
R = np.zeros([3, 3])
for i in range(3):
    for j in range(3):
        R[i, j] = cov[i, j] / np.sqrt(s[i] * s[j])

# Print the correlation matrix
print("Correlation matrix:")
print(R)
```

14.5.8 Visualizing the Covariance and Correlation

Let's visualize the covariance and correlation between the parameters using scatter plots.

```
#| autorun: false
# Generate synthetic data and fit the model
a = []
b = []
c = []
x = np.linspace(0, 1, 10)
for _ in range(100):
    y = data(x, -2.52, 1.6971755, 1)
    p, cov = curve_fit(parabola, x, y, sigma=err, p0=init_guess, absolute_sigma=True)
    a.append(p[0])
    b.append(p[1])
    c.append(p[2])

# Plot the correlation between parameters a and b
plt.figure(figsize=get_size(8, 8))
plt.scatter(a, b, alpha=0.2)
plt.xlabel('Parameter a')
plt.ylabel('Parameter b')
plt.title('Correlation between Parameters a and b')
plt.tight_layout()
plt.show()
```

By examining the covariance and correlation matrices, we can gain a deeper understanding of the uncertainties in the fit parameters and how they are related to each other.

14.5.9 Improving the Model

If we find that the parameters are highly correlated, we might want to find a better model containing more independent parameters. For example, we can write down a different model:

$$y = a(x - b)^2 + c \tag{14.12}$$

This model also contains three parameters, but the parameter b directly refers to the maximum of our parabola, while the parameter a denotes its curvature.

```
#| autorun: false
def newmodel(x, a, b, c):
    return a * (x - b)**2 + c
```

```
#| autorun: false
fit = curve_fit(newmodel, x_data, y_data, sigma=err, p0=init_guess, absolute_sigma=True)
```

```
#| autorun: false
ans, cov = fit
```

```
#| autorun: false
s = np.diag(cov)
R = np.zeros([3, 3])
for i in range(3):
    for j in range(3):
        R[i, j] = cov[i, j] / np.sqrt(s[i] * s[j])
```

We see from the covariance matrix that the new model has a smaller correlation of the parameters with each other.

```
#| autorun: false
print(cov)
```

This is also expressed by our correlation matrix.

```
#| autorun: false
print(R)
```

By examining the covariance and correlation matrices, we can gain valuable insights into the uncertainties in the fit parameters and how to improve our model.

Chapter 15

Numerical Differentiation

While we did introduce derivatives shortly already when exploring the slicing of arrays, we will now look at the numerical differentiation in more detail. This will require again a little bit of math.

```
#| edit: false
#| echo: false
#| execute: true

import numpy as np
import io
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
plt.rcParams.update({'font.size': 18})
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

# default values for plotting
plt.rcParams.update({'font.size': 12,
                     'lines.linewidth': 1,
                     'lines.markersize': 5,
                     'axes.labelsize': 11,
                     'xtick.labelsize' : 10,
                     'ytick.labelsize' : 10,
                     'xtick.top' : True,
                     'xtick.direction' : 'in',
                     'ytick.right' : True,
                     'ytick.direction' : 'in',})

def get_size(w,h):
      return((w/2.54,h/2.54))
```

15.1 First Order Derivative

Our previous method of finding the derivative was based on the definition of the derivative itself. The derivative of a function $f(x)$ at a point x is defined as the limit of the difference quotient as the interval Δx goes to zero:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

If we do not take the limit, we can approximate the derivative by:

$$f_i' \approx \frac{f_{i+1} - f_i}{\Delta x}$$

Here, we look to the right of the current position i and divide by the interval Δx. It is not difficult to see that the resulting local error δ at each step is given by:

$$\delta = f_{i+1} - f_i - \Delta x f'(x_i) = \frac{1}{2}\Delta x^2 f''(x_i) + O(\Delta x^3)$$

It can be seen that the error is proportional to the **square** of the interval Δx. This is the reason why the method is called first order accurate. The error is of the order of Δx^2.

A better expression can be found using the Taylor expansion around the position x_0:

$$f(x) = f(x_0) + (x - x_0)f'(x) + \frac{(x - x_0)^2}{2!}f''(x) + \frac{(x - x_0)^3}{3!}f^{(3)}(x) + ...$$

In discrete notation, this gives:

$$f_{i+1} = f_i + \Delta x f_i' + \frac{\Delta x^2}{2!}f_i'' + \frac{\Delta x^3}{3!}f_i^{(3)} + ...$$

The same can be done to obtain the function value at $i - 1$:

$$f_{i-1} = f_i - \Delta x f_i' + \frac{\Delta x^2}{2!}f_i'' - \frac{\Delta x^3}{3!}f_i^{(3)} + ...$$

Subtracting these two equations, we get:

$$f_{i+1} - f_{i-1} = 2\Delta x f_i' + O(\Delta x^3)$$

such that the second order term in Δx disappears. Neglecting the higher-order terms, we have

$$f_i' \approx \frac{f_{i+1} - f_{i-1}}{2\Delta x}$$

an thus have a first order derivative which is even more accurate than the one obtained from the definition of the derivative.

We can continue that type of derivation now to obtain higher order approximation of the first derivative with better accuracy. For that purpose you may calculate now $f_{i\pm2}$ and combining that with $f_{i+1} - f_{i-1}$ will lead to

$$f_i' = \frac{1}{12\Delta x}(f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}) \tag{15.1}$$

This can be used to give even better values for the first derivative.

Let‘s try out one of the formulas in the following code cell. We will write a function that calculates the derivative of a given function at a given position x. The function will take the function f as and argument, which is new to us. We will also introduce a small interval $h = \Delta x$ which will be used to calculate the derivative. The function will return the derivative of the function at the given position x.

```
#| autorun: false
def D(f, x, h=1.e-12, *params):
    return (f(x+h, *params)-f(x-h, *params))/(2*h)
```

Note that the definition contains additional parameters `*params` which are passed to the function `f`. This is a general way to pass additional parameters to the function `f` which is used in the definition of the derivative.

We will try to calculate the derivative of the $\sin(x)$ function:

```
#| autorun: false
def f(x):
    return(np.sin(x))
```

We can plot this and nicely obtain our cosine function

```
#| autorun: false

x=np.linspace(0.01,np.pi*4,1000)

plt.plot(x,D(f,x))
```

15.1.1 Matrix Version of the First Derivative

If we supply the above function with an array of positions x_i at which we would like to calculate the derivative, we obtain an array of derivative values. We can also write this procedure in a different way, which will be helpful for solving differential equations later.

If we consider the above finite difference formulas for a set of positions x_i, we can represent the first derivative at these positions by a matrix operation as well:

$$f' = \frac{1}{\Delta x}\begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}\begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{bmatrix} = \begin{bmatrix} \frac{f_2-f_1}{\Delta x} \\ \frac{f_3-f_2}{\Delta x} \\ \frac{f_4-f_3}{\Delta x} \\ \frac{f_5-f_4}{\Delta x} \\ \frac{f_6-f_5}{\Delta x} \\ \frac{0-f_6}{\Delta x} \end{bmatrix}$$

Note that here we took the derivative only to the right side! Each row of the matrix, when multiplied by the vector containing the function values, gives the derivative of the function f at the corresponding position x_i. The resulting vector represents the derivative in a certain position region.

We will demonstrate how to generate such a matrix with the `SciPy` module below.

15.2 Second order derivative

While we did before calculate the first derivative, we can also calculate the second derivative of a function. In the previous calculations we evaluated $f_{i+1} - f_{i-1}$. We can now also use the sum of both to arrive at

$$f_i'' \approx \frac{f_{i-1} - 2f_i + f_{i+1}}{\Delta x^2} \tag{15.2}$$

which gives the basic equation for calculating the second order derivative and the next order may be obtained from

$$f_i'' \approx \frac{1}{12\Delta x^2}(-f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+2}) \tag{15.3}$$

which is again better than our previous formula, yet needs more function values to be calculated.

15.3 SciPy Module

Of course, we are not the first to define some functions for calculating the derivative of functions numerically. This is already implemented in different modules. One module is the above mentioned `SciPy` module.

The `SciPy` module provides the method `derivative`, which we can call with

`derivative(f,x,dx=1.0,n=1):`

This will calculate the nth derivative of the function f at the position x with a intervall $dx = 1.0$ (default value).

```
#| autorun: false
## the derivative method is hidden in the `misc` sub-module of `SciPy`.
from scipy.misc import derivative
```

We also have the option to define the order parameter, which is not the order of the derivative but rather the number of points used to calculate the derivative according to our scheme earlier.

```
#| autorun: false
derivative(np.sin,np.pi,dx=0.000001,n=2,order=5)
```

15.3.1 Matrix Version

The `SciPy` module allows us to construct matrices as mentioned above. We will need the `diags` method from the `SciPy` module for that purpose.

```
#| autorun: false
from scipy.sparse import diags
```

Let's assume we want to calculate the derivative of the `sin` function at certain positions.

```
#| autorun: false
N = 100
x = np.linspace(-5, 5, N)
y = np.sin(x)
```

The `diags` function uses a set of numbers that should be distributed along the diagonals of the matrix. If you supply a list like in the example below, the numbers are distributed using the offsets as defined in the second list. The `shape` keyword defines the shape of the matrix. Try the example in the next cell with the `.todense()` suffix. This converts the otherwise unreadable sparse output to a readable matrix form.

```
#| autorun: false
m = diags([-1, 1], [0, 1], shape=(10, 10)).todense()
print(m)
```

To comply with our previous definition of $N = 100$ data points and the interval Δx, we define:

```
#| autorun: false
dx = x[1] - x[0]
m = diags([-1, 1], [0, 1], shape=(N, N)) / dx
```

The derivative is then simply a matrix-vector multiplication, which is done either by `np.dot(m,y)` or just by the `@` operator.

```
#| autorun: false
diff = m @ y
```

Let's plot the original function and its numerical derivative.

```
#| autorun: false
plt.figure(figsize=get_size(14,8))
plt.plot(x[:-1], diff[:-1], label=r'$f^{\prime}(x)$')
plt.plot(x, y, label=r'$f(x)=\sin(x)$')
```

```
plt.xlabel('x')
plt.ylabel(r'$f$, $f^{\prime}$')
plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
plt.ylim(-1, 1)
plt.tight_layout()
plt.show()
```

Check for yourself that the following line of code will calculate the second derivative.

```
#| autorun: false
m = diags([1, -2, 1], [-1, 0, 1], shape=(N, N)) / dx**2
second_diff = m @ y
```

Let's plot the original function and its second numerical derivative.

```
#| autorun: false
plt.figure(figsize=get_size(14,8))
plt.plot(x[1:-1], second_diff[1:-1], label=r'$f^{\prime\prime}(x)$')
plt.plot(x, y, label=r'$f(x)=\sin(x)$')
plt.xlabel('x')
plt.ylabel(r'$f$, $f^{\prime\prime}$')
plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
plt.ylim(-1, 1)
plt.tight_layout()
plt.show()
```

This demonstrates how to use the `SciPy` module to construct matrices for numerical differentiation and how to apply these matrices to compute first and second derivatives.

Applications of the Matrix Method

The matrix method for computing derivatives is particularly useful in several contexts, especially in numerical analysis and computational mathematics. Here are some key applications:

1. **Solving Differential Equations**:
   - **Ordinary Differential Equations (ODEs)**: The matrix method can be used to discretize ODEs, transforming them into a system of linear equations that can be solved using linear algebra techniques.
   - **Partial Differential Equations (PDEs)**: Similarly, PDEs can be discretized using finite difference methods, where derivatives are approximated by matrix operations. This is essential in fields like fluid dynamics, heat transfer, and electromagnetics.
2. **Numerical Differentiation**:
   - The matrix method provides a systematic way to approximate derivatives of functions given discrete data points. This is useful in data analysis, signal processing, and any application where you need to estimate the rate of change from sampled data.
3. **Stability and Accuracy Analysis**:
   - By representing derivative operations as matrices, it becomes easier to analyze the stability and accuracy of numerical schemes. This is crucial for ensuring that numerical solutions to differential equations are reliable.
4. **Optimization Problems**:
   - In optimization, especially in gradient-based methods, the matrix method can be used to compute gradients and Hessians efficiently. This is important in machine learning, operations research, and various engineering disciplines.
5. **Finite Element Analysis (FEA)**:
   - In FEA, the matrix method is used to approximate derivatives and integrals over complex geometries. This is widely used in structural engineering, biomechanics, and materials science.
6. **Control Theory**:

- In control theory, especially in the design and analysis of control systems, the matrix method can be used to model and simulate the behavior of dynamic systems.

Part VIII

Lecture 8

Chapter 16

Numerical Integration

This lecture covers numerical integration methods, which are essential for computing definite integrals of functions. We'll explore three different methods with increasing accuracy: the Box method, Trapezoid method, and Simpson's method.

```
#| edit: false
#| echo: false
#| execute: true

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Set default plotting parameters
plt.rcParams.update({
    'font.size': 12,
    'lines.linewidth': 1,
    'lines.markersize': 5,
    'axes.labelsize': 11,
    'xtick.labelsize': 10,
    'ytick.labelsize': 10,
    'xtick.top': True,
    'xtick.direction': 'in',
    'ytick.right': True,
    'ytick.direction': 'in',
})

def get_size(w, h):
    return (w/2.54, h/2.54)
```

16.1 Box Method (Rectangle Method)

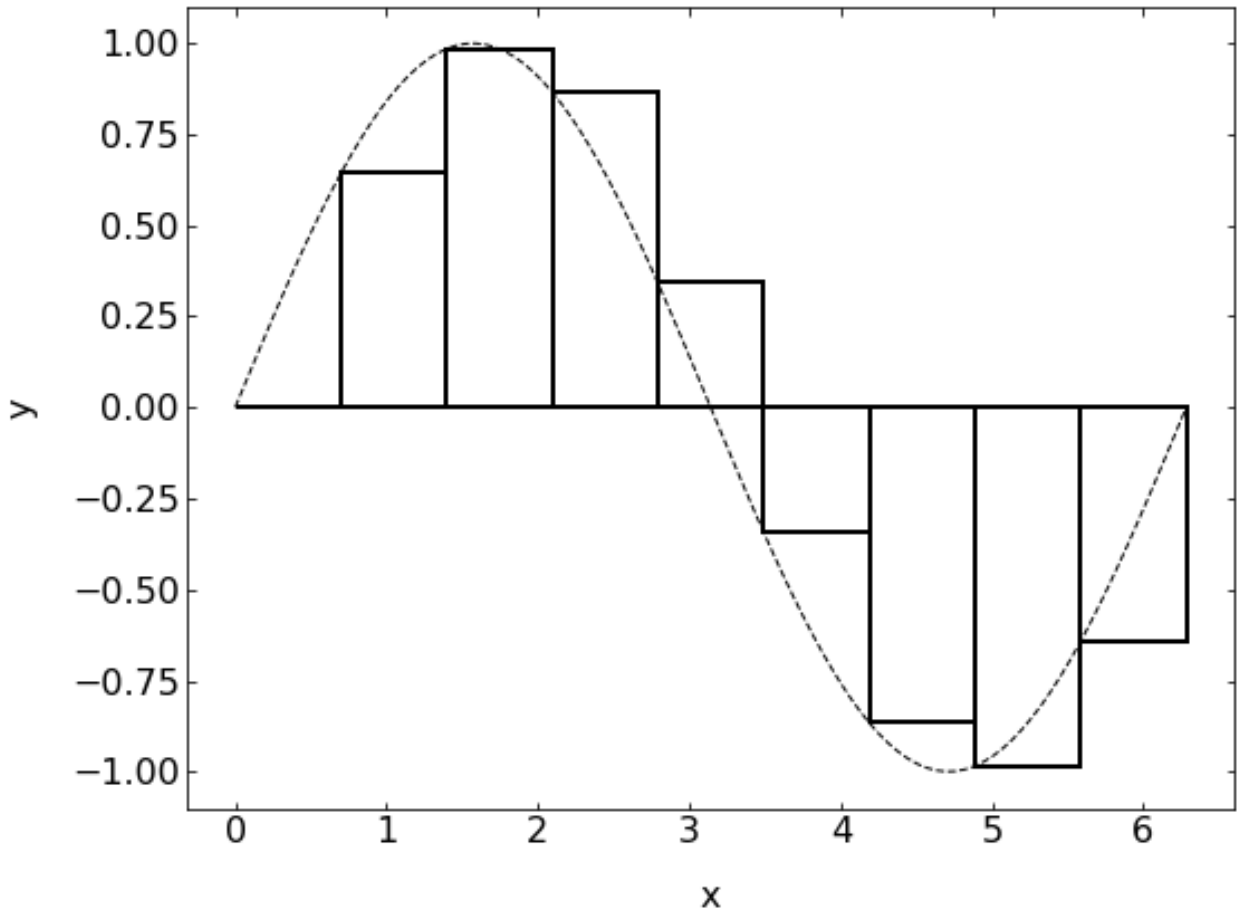


Figure 16.1: Box Method Illustration

The Box method is the simplest approach for numerical integration. It approximates the function in each interval Δx with a constant value taken at the left endpoint of the interval.

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} f(x_i)\Delta x \tag{16.1}$$

```
def f(x):
    """Example function to integrate: f(x) = x"""
    return x

def int_box(f, a, b, N):

    if N < 2:
        raise ValueError("N must be at least 2")
    x = np.linspace(a, b, N)
    y = f(x)
    return np.sum((x[1]-x[0])*y)

# Example calculation and convergence demonstration
```

```
N_values = np.arange(10, 10000, 100)
box_results = [int_box(f, 0, 1, N) for N in N_values]

plt.figure(figsize=get_size(15, 10))
plt.plot(N_values, box_results, label='Box Method')
plt.xlabel('Number of points (N)')
plt.ylabel('Integral value')
plt.title('Convergence of Box Method')
plt.grid(True)
plt.legend()
plt.show()
```

16.2 Trapezoid Method

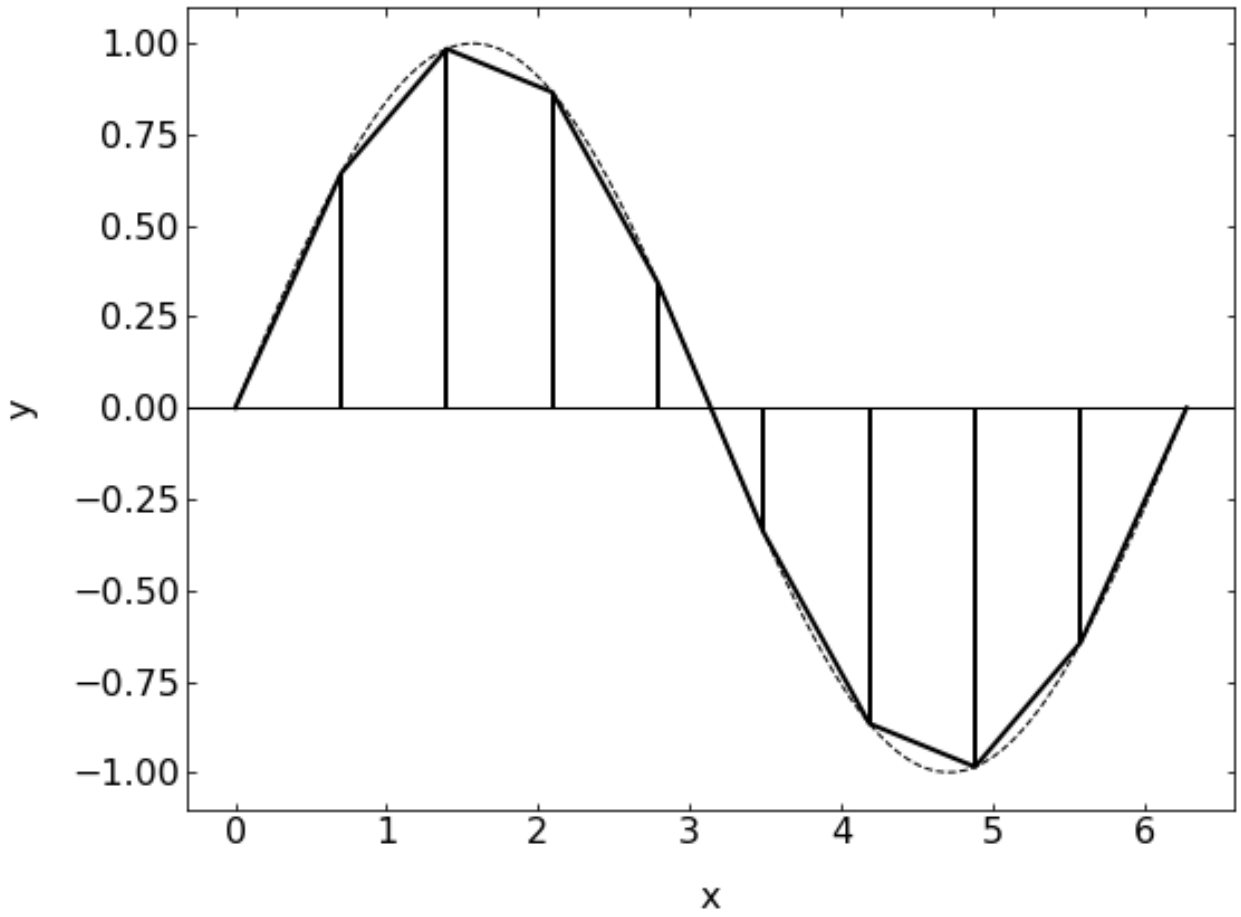


Figure 16.2: Trapezoid Method Illustration

The Trapezoid method improves upon the Box method by approximating the function with linear segments between points.

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} \frac{f(x_i) + f(x_{i-1})}{2} \Delta x \tag{16.2}$$

```
def int_trap(f, a, b, N):
    if N < 2:
        raise ValueError("N must be at least 2")
    x = np.linspace(a, b, N)
    y = f(x)
    return np.sum((y[1:] + y[:-1]) * (x[1]-x[0])/2)
```

16.3 Simpson's Method

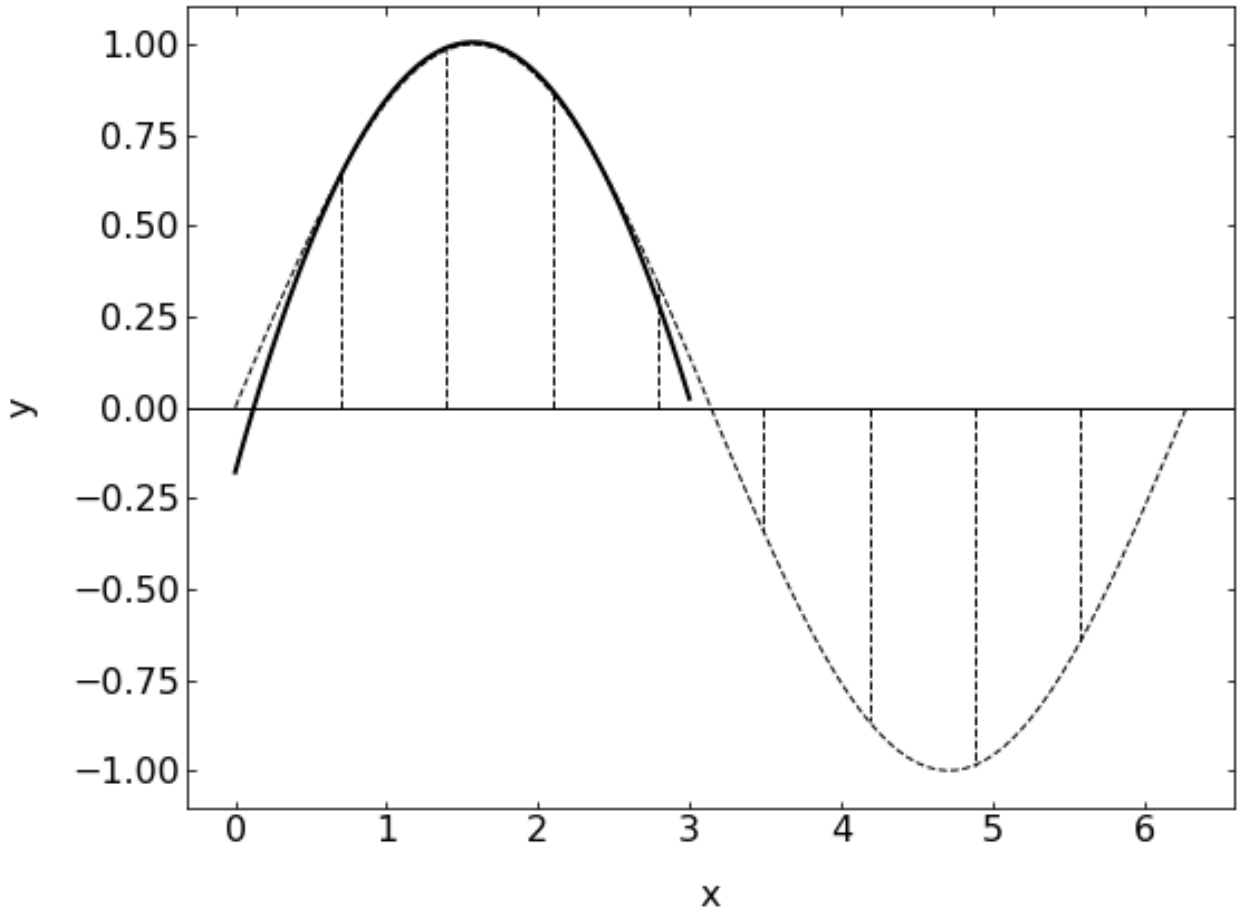


Figure 16.3: Simpson's Method Illustration

Simpson's method provides higher accuracy by approximating the function with parabolic segments.

$$\int_a^b f(x)dx \approx \frac{\Delta x}{3} \sum_{i=1}^{(N-1)/2} (f(x_{i-1}) + 4f(x_i) + f(x_{i+1})) \tag{16.3}$$

```
def int_simp(f, a, b, N):
    if N % 2 == 0:
        raise ValueError("N must be odd for Simpson's method")
    if N < 3:
        raise ValueError("N must be at least 3")
```

```
    x = np.linspace(a, b, N)
    y = f(x)
    return np.sum((y[0:-2:2] + 4*y[1:-1:2] + y[2::2]) * (x[1]-x[0])/3)
```

i Simpson's Rule for Numerical Integration

Simpson's Rule is a method for numerical integration that approximates the definite integral of a function by using quadratic polynomials.

1) For an integral $\int_a^b f(x)dx$, Simpson's Rule fits a quadratic function through three points:
   - $f(a)$
   - $f(\frac{a+b}{2})$
   - $f(b)$
2) Let's define:
   - $h = \frac{b-a}{2}$
   - $x_0 = a$
   - $x_1 = \frac{a+b}{2}$
   - $x_2 = b$
3) The quadratic approximation has the form:
$$P(x) = Ax^2 + Bx + C$$
4) This polynomial must satisfy:
$$f(x_0) = Ax_0^2 + Bx_0 + C$$
$$f(x_1) = Ax_1^2 + Bx_1 + C$$
$$f(x_2) = Ax_2^2 + Bx_2 + C$$
5) Using Lagrange interpolation:
$$P(x) = f(x_0)L_0(x) + f(x_1)L_1(x) + f(x_2)L_2(x)$$
   where L_0, L_1, L_2 are the Lagrange basis functions.

16.3.1 Final Formula

The integration of this polynomial leads to Simpson's Rule:

$$\int_a^b f(x)dx \approx \frac{h}{3}[f(a) + 4f(\frac{a+b}{2}) + f(b)]$$

16.3.2 Error Term

The error in Simpson's Rule is proportional to:

$$-\frac{h^5}{90}f^{(4)}(\xi)$$

for some $\xi \in [a, b]$

16.3.3 Composite Simpson's Rule

For better accuracy, we can divide the interval into n subintervals (where n is even):

$$\int_a^b f(x)dx \approx \frac{h}{3}[f(x_0) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=1}^{n/2-1} f(x_{2i}) + f(x_n)]$$

where $h = \frac{b-a}{n}$
The method is particularly effective for integrating functions that can be well-approximated by quadratic polynomials over small intervals.

16.4 Comparison of Methods

Let's compare the accuracy of all three methods:

```
def compare_methods(f, a, b, N_range):
    """Compare the accuracy of all three integration methods."""
    exact = 0.5  # Exact value of integral of f(x)=x from 0 to 1

    errors = {'Box': [], 'Trapezoid': [], 'Simpson': []}
    N_values = []

    for N in N_range:
        if N % 2 == 1:  # Only use odd N for fair comparison
            N_values.append(N)
            errors['Box'].append(abs(int_box(f, a, b, N) - exact))
            errors['Trapezoid'].append(abs(int_trap(f, a, b, N) - exact))
            errors['Simpson'].append(abs(int_simp(f, a, b, N) - exact))

    plt.figure(figsize=get_size(15, 10))
    plt.loglog(N_values, errors['Box'], 'o-', label='Box Method')
    plt.loglog(N_values, errors['Trapezoid'], 's-', label='Trapezoid Method')
    plt.loglog(N_values, errors['Simpson'], '^-', label='Simpson Method')
    plt.xlabel('Number of points (N)')
    plt.ylabel('Absolute Error')
    plt.title('Convergence Comparison of Integration Methods')
    plt.grid(True)
    plt.legend()
    plt.show()

# Compare methods for N from 3 to 99 (odd numbers only)
compare_methods(f, 0, 1, range(3, 100, 2))
```

16.5 Error Analysis

The three methods have different convergence rates:

- Box Method: Error Δx (linear convergence)
- Trapezoid Method: Error Δx^2 (quadratic convergence)
- Simpson's Method: Error Δx^4 (fourth-order convergence)

This explains why Simpson's method typically achieves higher accuracy with fewer points. For example, doubling the number of points in Simpson's method reduces the error by a factor of 16

Chapter 17

Solving ODEs

All the stuff we have defined in the previous sections is useful for solving ordinary differential equations. This will bring us closer to solving out physics problems now.

```
#| edit: false
#| echo: false
#| execute: true

import numpy as np
import io
import pandas as pd
import matplotlib.pyplot as plt
from scipy.sparse import diags

# default values for plotting
plt.rcParams.update({'font.size': 12,
                     'lines.linewidth': 1,
                     'lines.markersize': 5,
                     'axes.labelsize': 11,
                     'xtick.labelsize' : 10,
                     'ytick.labelsize' : 10,
                     'xtick.top' : True,
                     'xtick.direction' : 'in',
                     'ytick.right' : True,
                     'ytick.direction' : 'in',})

def get_size(w,h):
      return((w/2.54,h/2.54))
```

17.1 Harmonic Oscillator

> **Physics Interlude: The harmonic oscillator**
>
> We are going to tackle as a first very simple problem, the harmonic oscillator and we will demonstrate that with the matrix (Crank-Nicholson method or implicit scheme), the Euler type integration method and using some 'unknown' integrator in the module `SciPy`.
> The equation of motion for a classical harmonic oscillator is given

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + \omega^2 x = 0 \tag{17.1}$$

This is a second order differential equation which requires for its solution two initial conditions. The first initial condition is the initial elongation $x(t=0) = x_0$ and the second the initial velocity $\dot{x}(t=0) = v_0$.

17.2 Implicit Solution - Crank Nicholson

Lets start with the matrix appraoch we have just learned about. Using the matrix version, we can transform the above equation into a system of coupled equations, which we can solve with some standard methods available from e.g. the `SciPy` module.

17.2.1 Define Matrices

Our matrix will consist of two parts. The first containing the second derivative and the second just the elongation. Suppose we want to calculate the position $x(t)$ at 6 instances in time t_i then the matrix version of the second derivative reads as

$(x_1 = x(t_1), ...)$.

$$T = \frac{d^2x}{dt^2} = \frac{1}{\delta t^2} \begin{bmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

The second term in the equation of motion is a multiplication of the elongation $x(t_i)$ by ω^2 and can be written as

$$V = \omega^2 x = \begin{bmatrix} \omega^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \omega^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \omega^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \omega^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \omega^2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

The left hand side of the would threfore contain a sum of the two matrices $M = T + V$ multiplied by the vector x. We have therfore almost all things together to solve this differential equation with the help of an implicit scheme. What we have ignored so far are the initial conditions.

17.2.2 Use Initial Conditions

The matrix given for the second detivative actually implies already some initial (bounary) conditions. You probably noticed that the matrix contains incomplete coefficients for the second derivative in the first and last line. The first line contains $(-2, 1)$, but the second derivative should contain $(1, -2, 1)$. This $(-2, 1)$ thus always includes the boundary condition that $x_0 = 0$. To include our own initial/boundary conditions, we have to construct the matrix for the second derivative slightly differently and modify the differential equation to

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + \omega^2 x = b \tag{17.2}$$

where the vector b takes care of the initial conditions.

If we have N positions in time at which we calculate the elongation x, we have a $N \times N$ matrix of for the second derivatives. The lower $N-2$ lines will contain the the coefficients for the second derivative $(1, -2, 1)$. The first two lines supply the initial/boundary conditions.

The initial condition for the elongation $x(t=0) = x_0$ is obtained when the first element of the first line is a **1**. The matrix multiplication $M\,x = b$ for yields thus in the first line $x_1 = b_1$ and we set $b_1 = x_0$. The second line shall give the initial velocity. So the matrix entries of the second line contain a first derivative $(-1, 1)$. The matrix multiplication thus yields $x_2 - x_1 = b_2$. We can therefore need to set $b_2 = v_0 \delta t$. All of the other entries of b shall be set to zero according to the differential equation of the harmonic oscillator.

Our final problem $M\,x = b$ will thus have the following shape

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2+\omega^2 * \delta t^2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2+\omega^2 * \delta t^2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2+\omega^2 * \delta t^2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2+\omega^2 * \delta t^2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} x_0 \\ v_0 \delta t \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{17.3}$$

17.2.3 Solution

This is the final system of coupled equations which we can supply to any matrix solver. We will use a solver from the `scipy.linalg` module. Lets have a look at the details below.

```
N=10

(diags([-2., 1., 1.], [-1,-2, 0],
    shape=(N, N))+diags([1], [-1], shape=(N, N))* omega**2*dt**2)
```

```
#| autorun: false
from scipy.sparse import diags

k = 15.5 # spring constant
m = 0.2 # mass
omega=np.sqrt(k/m) # frequency of the oscillator

L = np.pi # time period over which we solve the ODE
N = 500 # number of data points
t = np.linspace(0, L, N) # time axis

b = np.zeros(N) # initial conditions vector
b[0]=1 # initial elongation
b[1]=0 # initial velocity

x = np.zeros(N) # solution vector
dt = t[1] - t[0] # time intervall of each step

# construct the matrix
T= diags([-2., 1., 1.], [-1,-2, 0], shape=(N, N)).todense()
V= diags([1], [-1], shape=(N, N)).todense()
M= T/dt**2 + V*omega**2

M[0,0]=1 # initial condition for amplitude, x1=1
M[1,0]=-1 # initial condition for velocity, dx/dt=0
M[1,1]=1

# initial condition vector
b=b.transpose()

x= np.linalg.solve(M, b) # this is the solution
```

```
#| autorun: false
#| fig-align: center
plt.figure(figsize=get_size(8,6))
plt.plot(t,x)
plt.xlabel('time t')
plt.ylabel('elongation x(t)')
plt.tight_layout()
plt.show()
```

17.3 Explicit Solution - Numerical Integration

Before implementing explicit numerical schemes, let's develop a standardized approach for solving ODEs. This framework will allow us to solve different problems using various methods with minimal code modification.

Let's examine the free fall problem as an example:

$$\ddot{x} = -g \tag{17.4}$$

This second-order equation can be transformed into a system of two first-order equations:

$$\dot{x} = v \tag{17.5}$$
$$\dot{v} = -g \tag{17.6}$$

Using the Euler method, these equations become:

$$x_{i+1} = x_i + v_i \Delta t \tag{17.7}$$
$$v_{i+1} = v_i - g\Delta t \tag{17.8}$$

Note: The original equations had $\dot{x}$ and $\dot{v}$ in the right-hand side, which should be replaced with their actual values (v and $-g$ respectively).

These equations can be written more compactly in vector form:

$$\vec{y}_{i+1} = \vec{y}_i + \dot{\vec{y}}_i \Delta t \tag{17.9}$$

where

$$\vec{y} = \begin{bmatrix} x \\ v \end{bmatrix} \tag{17.10}$$

and

$$\dot{\vec{y}} = \begin{bmatrix} v \\ -g \end{bmatrix} \tag{17.11}$$

This vector formulation allows us to separate: 1. Problem definition (specifying $\dot{\vec{y}}$ as a function of $\vec{y}$ and t) 2. Solution method (implementing the numerical integration scheme)

We'll explore three numerical methods:

- **Euler Method**: First-order accurate

- **Euler-Cromer Method**: Modified Euler method, better for oscillatory systems
- **Midpoint Method**: Second-order accurate

More sophisticated methods like the Runge-Kutta family offer higher accuracy but are not covered here.

17.3.1 Euler Method

The **Euler method** is derived from the Taylor expansion of the solution $\vec{y}(t)$ around the current time t:

$$\vec{y}(t+\Delta t) = \vec{y}(t) + \dot{\vec{y}}(t)\Delta t + \frac{1}{2}\ddot{\vec{y}}(t)\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{17.12}$$

The Euler method approximates this by truncating after the first-order term:

$$\vec{y}(t+\Delta t) \approx \vec{y}(t) + \dot{\vec{y}}(t)\Delta t \tag{17.13}$$

For our free fall example, this becomes:

$$\begin{bmatrix} x_{i+1} \\ v_{i+1} \end{bmatrix} = \begin{bmatrix} x_i \\ v_i \end{bmatrix} + \begin{bmatrix} v_i \\ -g \end{bmatrix} \Delta t \tag{17.14}$$

Error Analysis: The method has two distinct types of errors. The local truncation error, which represents the error made in a single step, is of order $\mathcal{O}(\Delta t^2)$. This corresponds to the first term omitted in the Taylor expansion. The global truncation error, which accumulates over the entire integration interval $[0, \tau]$, is of order $\mathcal{O}(\Delta t)$. This can be understood by considering that we take $N = \tau/\Delta t$ steps, each contributing an error proportional to Δt^2. The total error thus scales as $N \cdot \Delta t^2 = \tau \Delta t$.

Limitations and Extensions: The method is directly applicable only to first-order systems of the form $\dot{\vec{y}} = \vec{f}(\vec{y}, t)$. However, this is not a fundamental limitation as higher-order equations can be converted to systems of first-order equations. For example, a second-order equation $\ddot{x} = f(x, \dot{x}, t)$ can be transformed into a system of two first-order equations by introducing the velocity as an additional variable. The resulting system becomes:

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v \\ f(x, v, t) \end{bmatrix} \tag{17.15}$$

This transformation allows us to apply the method to a wider class of problems while maintaining its fundamental characteristics.

17.3.2 Euler-Cromer Method

The **Euler-Cromer method** (also known as the semi-implicit Euler method) modifies the basic Euler method by using the updated velocity when calculating the position. For a system described by position and velocity:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= f(x, v, t) \end{aligned} \tag{17.16}$$

The integration steps are:

$$\begin{aligned} v_{i+1} &= v_i + f(x_i, v_i, t_i)\Delta t \\ x_{i+1} &= x_i + v_{i+1}\Delta t \end{aligned} \tag{17.17}$$

For our free fall example:

$$\begin{aligned} v_{i+1} &= v_i - g\Delta t \\ x_{i+1} &= x_i + v_{i+1}\Delta t \end{aligned} \tag{17.18}$$

Energy Behavior: The method shows improved energy conservation for oscillatory systems compared to the standard Euler method. While the Euler method typically increases energy over time, the Euler-Cromer method exhibits small energy oscillations around the correct value.

Error Analysis: The method maintains a local truncation error of $\mathcal{O}(\Delta t^2)$ and a global truncation error of $\mathcal{O}(\Delta t)$. Despite having the same order of accuracy as the Euler method, it provides more stable solutions for oscillatory systems.

Advantages: The Euler-Cromer method represents a simple modification of the Euler method that achieves better stability for oscillatory systems without requiring additional function evaluations.

Limitations: The method remains first-order accurate globally and is not symmetric in time. While it performs well for certain types of problems, particularly oscillatory systems, it may not be suitable for all differential equations.

Comparison with Euler Method:

```
# Euler Method
v[i+1] = v[i] + f(x[i],v[i],t[i])*dt
x[i+1] = x[i] + v[i]*dt       # Uses old velocity

# Euler-Cromer Method
v[i+1] = v[i] + f(x[i],v[i],t[i])*dt
x[i+1] = x[i] + v[i+1]*dt     # Uses new velocity
```

17.3.3 Midpoint Method

The **Midpoint Method** (also known as the second-order Runge-Kutta method) improves upon both the Euler and Euler-Cromer methods by using the average of the derivatives at the current point and an estimated midpoint.

For a system of first-order differential equations:

$$\dot{\vec{y}} = \vec{f}(\vec{y}, t) \tag{17.19}$$

The algorithm proceeds in two steps:

1. Calculate an intermediate point using an Euler step to the midpoint:

$$\vec{k}_1 = \vec{f}(\vec{y}_i, t_i) \tag{17.20}$$

$$\vec{y}_{i+1/2} = \vec{y}_i + \frac{\Delta t}{2}\vec{k}_1 \tag{17.21}$$

2. Use the derivative at this midpoint for the full step:

$$\vec{k}_2 = \vec{f}(\vec{y}_{i+1/2}, t_i + \Delta t/2) \tag{17.22}$$

$$\vec{y}_{i+1} = \vec{y}_i + \Delta t\vec{k}_2 \tag{17.23}$$

For our free fall example, this becomes:

$$\begin{aligned} v_{i+1/2} &= v_i - \frac{g\Delta t}{2} \\ x_{i+1/2} &= x_i + v_i\frac{\Delta t}{2} \\ v_{i+1} &= v_i - g\Delta t \\ x_{i+1} &= x_i + v_{i+1/2}\Delta t \end{aligned} \tag{17.24}$$

Error Analysis: The method achieves higher accuracy than both Euler and Euler-Cromer methods with:

- Local truncation error: $\mathcal{O}(\Delta t^3)$
- Global truncation error: $\mathcal{O}(\Delta t^2)$

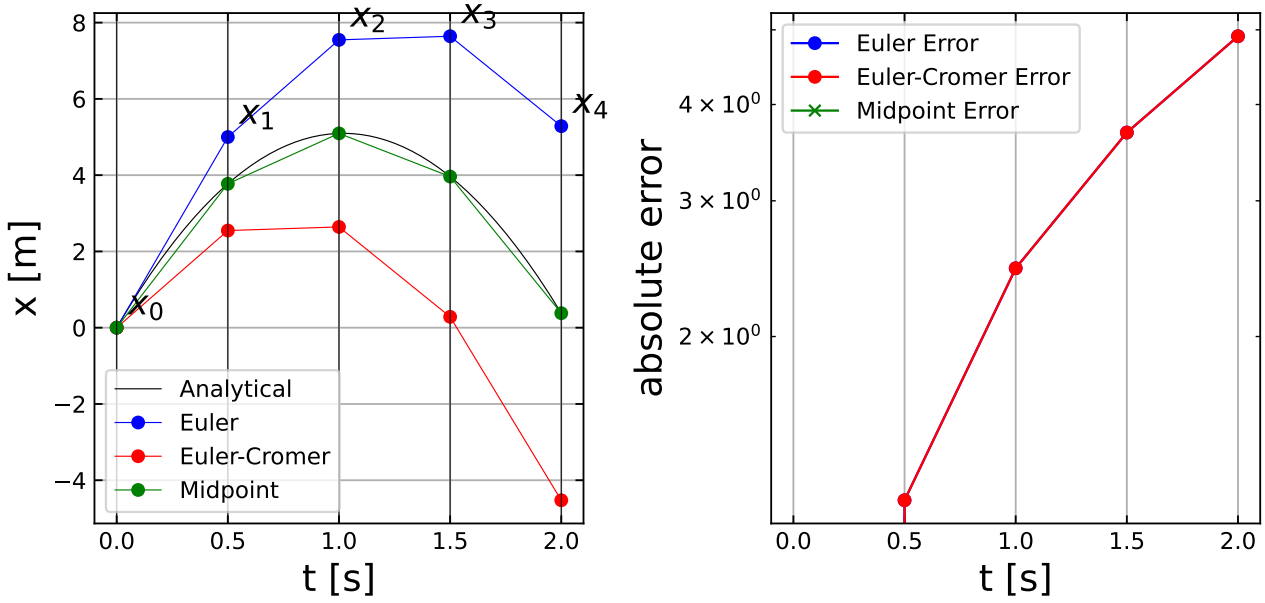
Implementation:

```
def midpoint_step(y, t, dt, f):
    # Calculate k1
    k1 = f(y, t)

    # Calculate midpoint
    y_mid = y + 0.5 * dt * k1

    # Calculate k2 at midpoint
    k2 = f(y_mid, t + 0.5*dt)

    # Full step using midpoint derivative
    return y + dt * k2
```



17.3.4 Putting it all together

Now we can implement our numerical solution by combining our understanding of both the physical system and numerical methods. This implementation consists of two main parts: defining the differential equation and solving it numerically.

The Definition of the Problem

For the simple harmonic oscillator, we start with the second-order differential equation:

$$\frac{d^2x}{dt^2} + \omega^2 x = 0 \tag{17.25}$$

To solve this numerically, we convert it to a system of first-order equations using our state vector $\vec{y} = [x, v]^T$:

$$\frac{d}{dt}\begin{bmatrix} x \\ v \end{bmatrix} = \begin{bmatrix} v \\ -\omega^2 x \end{bmatrix} \tag{17.26}$$

This is implemented as: ~~~ def SHO(state, time): “ “ ” Define the harmonic oscillator system. state[0] : position x state[1] : velocity v returns : [dx/dt, dv/dt] “ “ ” g0 = state[1] # dx/dt = v g1 = -k/m*state[0] # dv/dt = - ²x return np.array([g0, g1]) ~~~

This function defines our physical system by returning the derivatives of our state variables at any given point.

Solving the Problem

With our system defined, we can implement the numerical solution using Euler's method. The basic algorithm takes the current state and advances it by one time step:

```
def euler(y, t, dt, derivs):
    """
    Perform one step of the Euler method.
    y      : current state [x, v]
    t      : current time
    dt     : time step
    derivs : function returning derivatives
    """
    y_next = y + derivs(y, t) * dt
    return y_next
```

This simple structure allows us to solve different physical problems by just changing the derivative function. For example, we can solve the free fall problem with initial conditions $x_0 = 0$ and $v_0 = 10$, or the harmonic oscillator with specified spring constant k and mass m.

The key advantage of this structure lies in its flexibility. We can change the physical system by providing a different derivative function, implement various numerical methods by modifying the integration step, and explore the system behavior by adjusting parameters and initial conditions. This modular approach allows us to study a wide range of physical systems using the same basic numerical framework.

```
#| autorun: false
# Parameters
N = 2000  # number of steps
tau = 4*np.pi  # time period
xo = 1.0  # initial position
vo = 0.0  # initial velocity

k = 3.5
m = 0.2
gravity = 9.8

dt = tau/float(N-1)
time = np.linspace(0, tau, N)

y = np.zeros([N, 2])
y[0, 0] = xo
y[0, 1] = vo


##  defining the problem
def free_fall(state , time):
    g0 = state[1]
    g1 = -gravity
    return(np.array([g0, g1]))


def SHO(y, t, b=0, k=3.5):
    x, v = y
```

```
    dydt = [v, -b*v - k*x]
    return np.array(dydt)


def MMM(state, time):
    g0 = state[1]-1.1*state[1]
    g1 = -k/m * state[0]-12
    return(np.array([g0, g1]))

##  solving the problem with euler
def euler(y, t, dt, derivs):
    y_next = y + derivs(y,t)* dt
    return(y_next)

def runge_kutta2(y, time, dt, derivs):
    k0 = dt * derivs(y, time)
    k1 = dt * derivs(y + k0, time + dt)
    y_next = y + 0.5 * (k0 + k1)
    return y_next

# Solve the differential equation
for j in range(N-1):
    y[j+1] = euler(y[j], time[j], dt, SHO)

# Plot results
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

ax1.plot(time, y[:, 0])
ax1.set_title("Position")
ax1.set_xlabel('time [s]', fontsize=16)
ax1.set_ylabel('position x [m]', fontsize=16)
ax1.tick_params(labelsize=14)

ax2.plot(time, y[:, 1])
ax2.set_title("Velocity")
ax2.set_xlabel('time [s]', fontsize=16)
ax2.set_ylabel('velocity v [m/s]', fontsize=16)
ax2.tick_params(labelsize=14)

plt.tight_layout()
plt.show()
```

17.4 Solving the Harmonic Oscillator with SciPy

Having explored basic numerical integration methods, we can now utilize more sophisticated tools available in SciPy. The `scipy.integrate.odeint()` function provides a robust and accurate integration method with several advantages over our simple implementations.

To use SciPy's integrator:

```
from scipy.integrate import odeint
```

The basic syntax is:

```
solution = odeint(derivative_function, initial_conditions, time_points)
```

where:

- `derivative_function` defines the system (like our `SHO` function)
- `initial_conditions` is a vector containing $[x_0, v_0]$
- `time_points` is an array of times at which to compute the solution

The `odeint` function offers several significant advantages over our simple implementations. It features adaptive step size control, which automatically adjusts the integration step size based on the local error. The function performs continuous error estimation and correction to maintain accuracy throughout the integration. It also provides various integration methods that can be selected based on the problem's requirements. The function is capable of handling stiff equations, which are particularly challenging for simpler methods, and generally provides better numerical stability across a wide range of problems.

For example, to solve the harmonic oscillator:

```
def SHO(state, t, k=1.0, m=1.0):
    x, v = state
    return [v, -k/m * x]

# Initial conditions
y0 = [1.0, 0.0]  # x  = 1, v  = 0
t = np.linspace(0, 10, 1000)

# Solve the system
solution = odeint(SHO, y0, t)
```

The solution array contains:

- `solution[:, 0]`: position values
- `solution[:, 1]`: velocity values

Having understood the fundamentals of numerical integration through our implementations of Euler and other methods, we can now confidently use this more sophisticated tool for solving differential equations more accurately and efficiently.

17.4.1 Setup

```
#| autorun: false
from scipy.integrate import odeint

N = 1000  # number of steps
xo = 1.0 # initial position
vo = 0.0 # initial velocity
tau = 4*np.pi # time period

k = 3.5
m = 0.2
gravity = 9.8

time = np.linspace(0, tau, N)

y = np.zeros(2)
y[0] = xo
y[1] = vo
```

17.4.2 Definition

```
#| autorun: false
##  defining the problem
def SHO(state, time):
```

```
    g0 = state[1]
    g1 = -k/m * state [0]
    return(np.array([g0, g1]))
```

17.4.3 Solution

```
#| autorun: false

answer = odeint( SHO, y , time )
```

17.4.4 Plotting

```
#| autorun: false
fig=plt.figure(1, figsize = (10,5) )
plt.subplot(1, 2, 1)
plt.plot(time, answer[:,0])
plt.ylabel("position , velocity")
plt.xlabel('time [s]', fontsize=16)
plt.ylabel('position x [m]',fontsize=16)
plt.tick_params(labelsize=14)

plt.subplot(1, 2, 2)
plt.plot(time, answer[:,1])
plt.xlabel('time [s]', fontsize=16)
plt.ylabel('velocity v [m/s]',fontsize=16)
plt.tick_params(labelsize=14)

plt.tight_layout()
plt.show()
```

17.5 Damped Driven Pendulum in SciPy

Write a `derivs` function for a damped driven pendulum:

$$\ddot{\theta} = -\frac{g}{L}\sin(\theta) - b\dot{\theta} + \beta\cos(\omega t) \tag{17.27}$$

Use this *derivs* function with the *SciPy* solver and plot the result for different parameters. Vary the damping parameter b. Observe the contributions of the homogeneous and the particular solution. Plot the amplitude of the stationary solution as a function of frequency!

17.5.1 Setup

```
#| autorun: false
N = 10000  # number of steps
theta_o = 1.0 # initial position
vo = -0.0 # initial velocity
tau = 100.0 # time period

length=10.0
b=0.2
beta=np.pi/2
gravity = 9.8
```

```
omega=np.sqrt(gravity/length)


time = np.linspace(0, tau, N)

y = np.zeros (2)
y[0] = theta_o
y[1] = vo
```

17.5.2 Definition

```
#| autorun: false
def pendulum_def(state , time):
    g0 = state[1]
    g1 = -gravity/length * np.sin(state[0]) - b*state[1] + beta*np.cos(omega * time)
    return(np.array([g0, g1]) )
```

17.5.3 Solution

```
#| autorun: false
answer = odeint( pendulum_def, y , time )
```

17.5.4 Plotting

```
#| autorun: false
fig=plt.figure(1, figsize = (8,6) )
plt.plot(time,beta*np.cos(omega * time),'r--',alpha=0.3)
plt.plot(time, answer[:,0])
plt.xlabel('time [s]', fontsize=16)
plt.ylabel('angular velocity',fontsize=16)
plt.tick_params(labelsize=14)
plt.show()
```