Introduction to Computer-based Physical Modeling

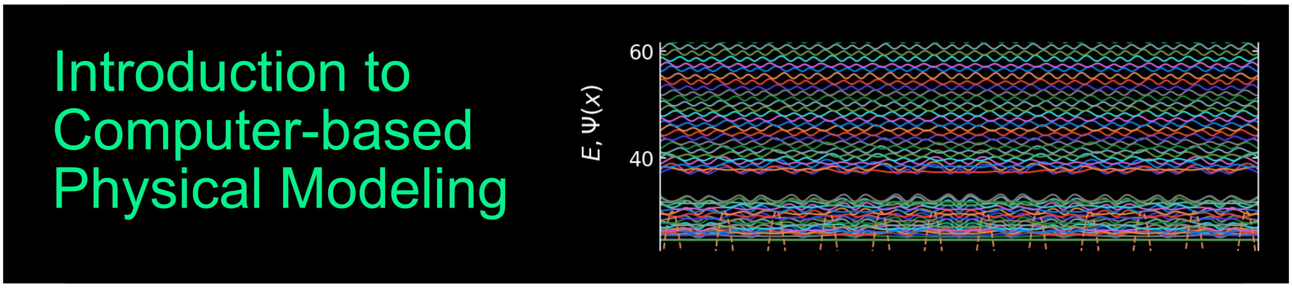
Frank Cichos

2024-08-14

Table of contents

Chapter 1

CBPM 2025



Chapter 2

Welcome to the Course Computer-Based Physical Modelling!

The programming language Python is useful for all kinds of scientific and technical tasks. You can use it to analyze and visualize data. You can also use it to numerically solve scientific problems that are difficult or impossible to solve analytically. Python is freely available and, due to its modular structure, has been expanded with an almost infinite number of modules for various purposes.

This course aims to introduce you to programming with Python. It is primarily aimed at beginners, but we hope it will also be interesting for advanced users. We begin the course with an introduction to the Jupyter Notebook environment, which we will use throughout the entire course. Afterward, we will provide an introduction to Python and show you some basic functions, such as plotting and analyzing data through curve fitting, reading and writing files, which are some of the tasks you will encounter during your physics studies. We will also show you some advanced topics such as animation in Jupyter and the simulation of physical processes in

- Mechanics
- Electrostatics
- Waves
- Optics

If there is time left at the end of the course, we will also take a look at machine learning methods, which have become an important tool in physics as well.

We will not present a comprehensive list of numerical simulation schemes, but rather use the examples to spark your curiosity. Since there are slight differences in the syntax of the various Python versions, we will always refer to the Python 3 standard in the following.

The course will be held in German. However, the web pages that you will be provided for an overview of Python will be in English. Exercise assignments will be given in German.

Part I

Lecture 1

Chapter 3

Programming Background Questionnaire

Please complete this short questionnaire to help tailor the course to your needs. Your responses are anonymous and will be used only to adapt the teaching to your level of experience.

Chapter 4

Jupyter Notebooks

Throughout this course we will have to create and edit python code. We will primarily use this webpage for convenience, but for day-to-day work in the laboratory, it's beneficial to utilize a code editor or a notebook environment like JupyterLab. JupyterLab is a robust platform that enables you to develop and modify notebooks within a web browser, while also offering comprehensive capabilities for analyzing and visualizing data.

4.1 What is a 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.

4.1.1 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.

4.1.2 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

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.

4.1.3 JupyterLab Example

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

4.1.4 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.

4.2 Using the Notebook Editor

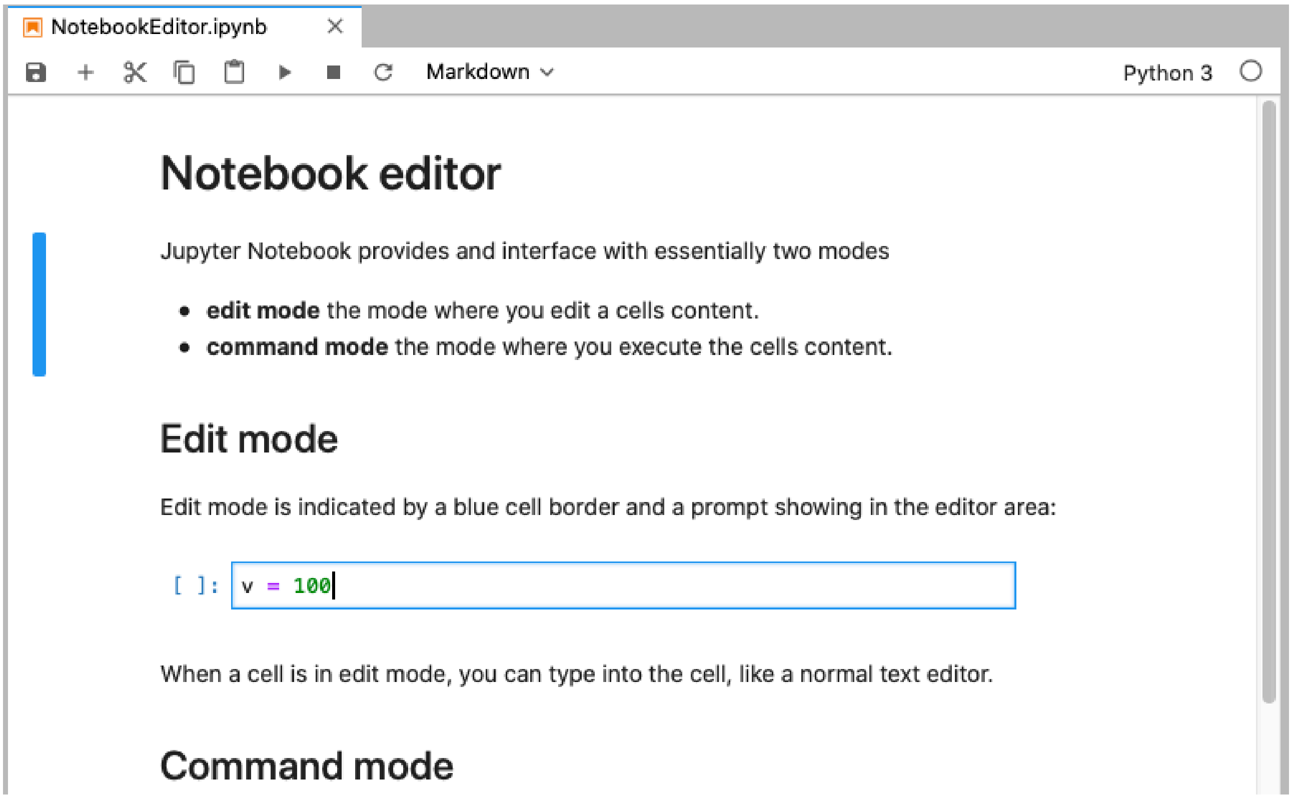


Figure 4.1: Jupyter Notebook Editor

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.

4.2.1 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.

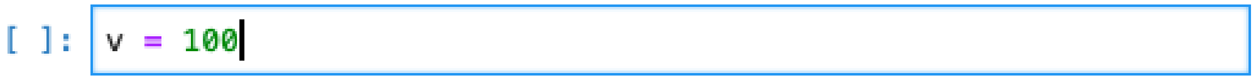


Figure 4.2: Edit Mode

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

4.2.2 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.

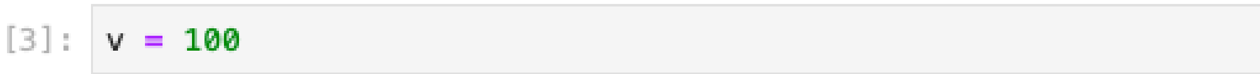


Figure 4.3: Command Mode

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.

4.2.3 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**

4.2.4 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.

4.3 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".

4.4 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.

4.4.1 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:

4.4.2 Markdown lists example

- 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

4.4.3 Blockquote example

> 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!

4.4.4 Web links example

[Jupyter's website]

4.4.5 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
```

4.4.6 Embedded code

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

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

4.4.7 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 $$:

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

4.4.8 Images

Images may be also directly integrated into a Markdown block.

To include images use

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

for example

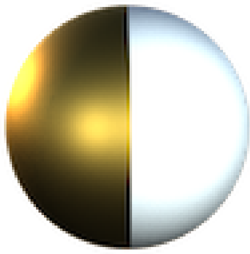


Figure 4.4: alternative text

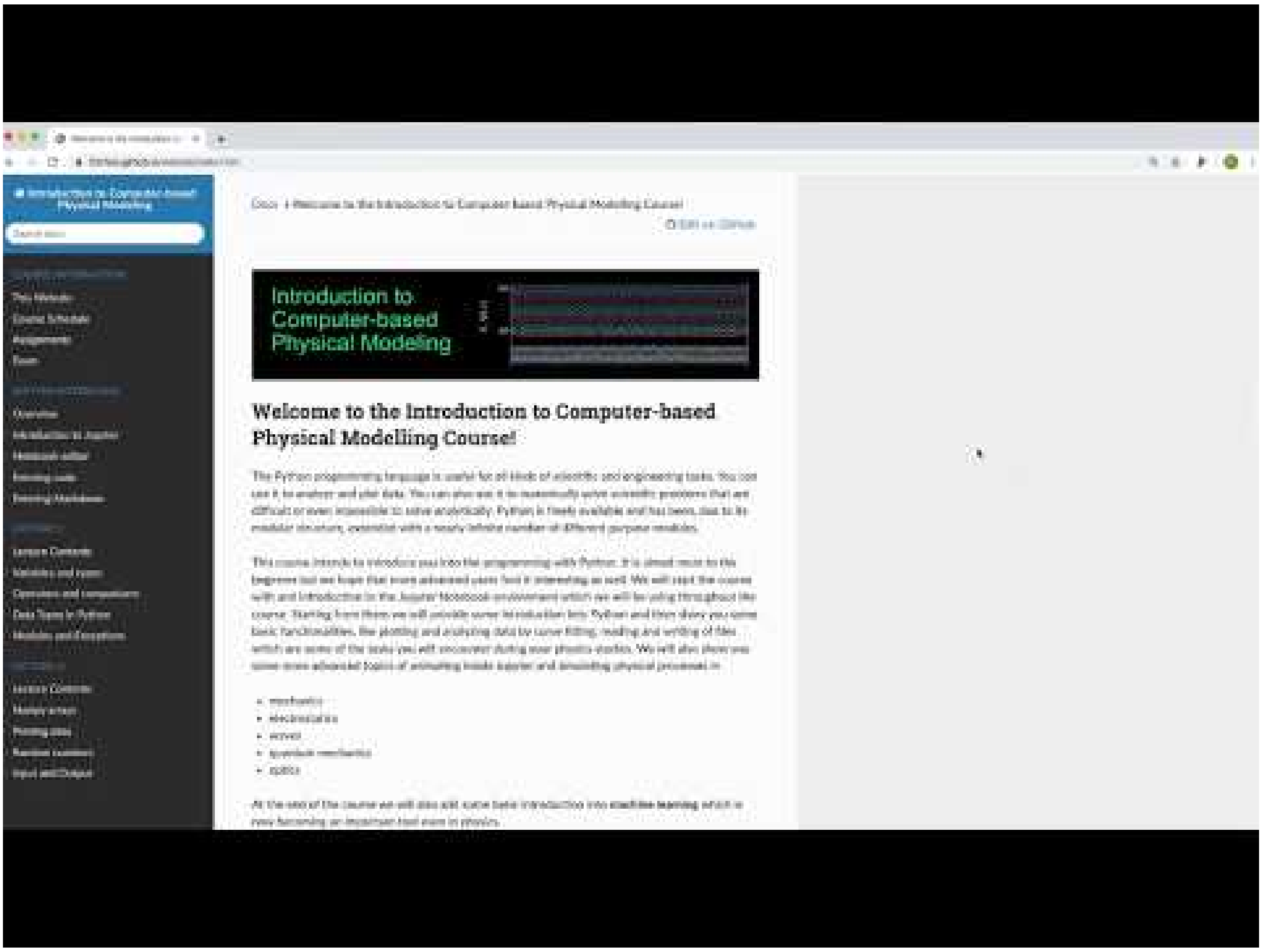
4.4.9 Videos

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 5

Python & Anatomy of a Python Program

5.1 What is Python?

Python is a high-level, interpreted programming language known for its readability and simplicity. Created by Guido van Rossum in 1991, it emphasizes code readability with its clear syntax and use of indentation. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming. It comes with a comprehensive standard library and has a vast ecosystem of third-party packages, making it suitable for various applications such as web development, data analysis, artificial intelligence, scientific computing, and automation. Python's "batteries included" philosophy and gentle learning curve have contributed to its popularity among beginners and experienced developers alike.

For physics students specifically, Python has become the language of choice for data analysis, simulation, and visualization in scientific research. Libraries like NumPy, SciPy, and Matplotlib provide powerful tools for solving physics problems, from basic mechanics to quantum mechanics.

5.2 Anatomy of a Python Program

Understanding the basic structure of a Python program is essential for beginners. Let's break down the fundamental elements that make up a typical Python program.

5.2.1 Basic Elements

Element	Description	Example
Statements	Individual instructions that Python executes	`x = 10`
Expressions	Combinations of values, variables, and operators that evaluate to a value	`x + 5`
Blocks	Groups of statements indented at the same level	Function bodies, loops
Functions	Reusable blocks of code that perform specific tasks	`def calculate_area(radius):`
Comments	Notes in the code that are ignored by the interpreter	`# This is a comment`
Imports	Statements that give access to external modules	`import numpy as np`

5.2.2 Visual Structure of a Python Program

```python
# 1. Import statements (external libraries)
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint  # For solving differential equations

# 2. Constants and global variables
GRAVITY = 9.81  # m/s^2
PLANCK_CONSTANT = 6.626e-34  # J·s
ELECTRON_MASS = 9.109e-31  # kg

# 3. Function definitions
def calculate_kinetic_energy(mass, velocity):
    """
    Calculate the kinetic energy of an object.

    Parameters:
        mass (float): Mass of the object in kg
        velocity (float): Velocity of the object in m/s

    Returns:
        float: Kinetic energy in Joules
    """
    return 0.5 * mass * velocity**2

def spring_force(k, displacement):
    """
    Calculate the force exerted by a spring.

    Parameters:
        k (float): Spring constant in N/m
        displacement (float): Displacement from equilibrium in m

    Returns:
        float: Force in Newtons (negative for restoring force)
    """
    return -k * displacement

# 4. Class definitions (if applicable)
class Particle:
    def __init__(self, mass, position, velocity):
        self.mass = mass
        self.position = position
        self.velocity = velocity

    def update_position(self, time_step):
        # Simple Euler integration
        self.position += self.velocity * time_step

    def potential_energy(self, height, g=GRAVITY):
        """Calculate gravitational potential energy"""
        return self.mass * g * height

    def momentum(self):
        """Calculate momentum"""
```

```
        return self.mass * self.velocity

# 5. Main execution code
if __name__ == "__main__":
    # Create objects or variables
    particle = Particle(1.0, np.array([0.0, 0.0]), np.array([1.0, 2.0]))

    # Set up simulation parameters
    time_step = 0.01  # seconds
    total_time = 1.0  # seconds
    n_steps = int(total_time / time_step)

    # Arrays to store results
    positions = np.zeros((n_steps, 2))
    times = np.zeros(n_steps)

    # Process data/perform calculations - simulate motion
    for i in range(n_steps):
        particle.update_position(time_step)
        positions[i] = particle.position
        times[i] = i * time_step

    # Output results
    print(f"Final position: {particle.position}")
    print(f"Final kinetic energy: {calculate_kinetic_energy(particle.mass, np.linalg.norm(particle.velo

    # Visualize results (if applicable)
    plt.figure(figsize=(10, 6))
    plt.subplot(1, 2, 1)
    plt.plot(positions[:, 0], positions[:, 1], 'r-')
    plt.xlabel('X position (m)')
    plt.ylabel('Y position (m)')
    plt.title('Particle Trajectory')
    plt.grid(True)

    plt.subplot(1, 2, 2)
    plt.plot(times, positions[:, 0], 'b-', label='x-position')
    plt.plot(times, positions[:, 1], 'g-', label='y-position')
    plt.xlabel('Time (s)')
    plt.ylabel('Position (m)')
    plt.title('Position vs Time')
    plt.legend()
    plt.grid(True)

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

5.2.3 Key Concepts

1. **Modularity**: Python programs are typically organized into functions and classes that encapsulate specific functionality.

2. **Indentation**: Python uses indentation (typically 4 spaces) to define code blocks, unlike other languages that use braces `{}`.

3. **Documentation**: Good Python code includes docstrings (triple-quoted strings) that explain what func-

tions and classes do.

4. **Main Block**: The `if __name__ == "__main__":` block ensures code only runs when the file is executed directly, not when imported.

5. **Readability**: Python emphasizes code readability with clear variable names and logical organization.

6. **Physics Modeling**: For physics problems, we typically model physical systems as objects with properties (mass, position, etc.) and behaviors (update_position, calculate_energy, etc.).

7. **Numerical Integration**: Many physics problems require solving differential equations numerically using methods like Euler integration or Runge-Kutta.

8. **Units**: Always include appropriate SI units in your comments and documentation to ensure clarity in physics calculations.

Best Practices

- Keep functions short and focused on a single task
- Use meaningful variable and function names
- Include comments to explain *why* rather than *what* (the code should be self-explanatory)
- Follow PEP 8 style guidelines for consistent formatting
- Structure larger programs into multiple modules (files)
- For physics simulations, validate your code against known analytical solutions when possible
- Remember to handle units consistently throughout your calculations
- Consider the appropriate numerical methods for the physical system you're modeling

Physics-Specific Python Libraries

- **NumPy**: Provides array operations and mathematical functions
- **SciPy**: Scientific computing tools including optimization, integration, and differential equations
- **Matplotlib**: Plotting and visualization
- **SymPy**: Symbolic mathematics for analytical solutions
- **Pandas**: Data manipulation and analysis
- **astropy**: Astronomy and astrophysics
- **scikit-learn**: Machine learning for data analysis
- **PyMC**: Probabilistic programming for statistical analysis

Chapter 6

Variables & Numbers

6.1 Variables in Python

6.1.1 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
>
> Python has keywords that cannot be used as variable names. The most common ones you'll encounter in physics programming are:
> `if`, `else`, `for`, `while`, `return`, `and`, `or`, `lambda`
> Note that `lambda` is particularly relevant as it could naturally appear in physics code, but since it's reserved for anonymous functions in Python, it cannot be used as a variable name.

6.1.2 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)
```

6.2 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.

6.2.1 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

6.2.2 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
```

6.2.3 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
```

6.2.4 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)
```

6.3 Operators

Python provides a variety of operators for performing operations on variables and values. Here we'll cover the most common operators used in scientific programming.

6.3.1 Arithmetic Operators

These operators perform basic mathematical operations:

Operator	Name	Example	Result
`+`	Addition	`5 + 3`	8
`-`	Subtraction	`5 - 3`	2
`*`	Multiplication	`5 * 3`	15
`/`	Division	`5 / 3`	1.6666...
`//`	Floor Division	`5 // 3`	1
`%`	Modulus (remainder)	`5 % 3`	2
`**`	Exponentiation	`5 ** 3`	125

```
#| autorun: false
# Examples of arithmetic operators
print(f"Addition: 5 + 3 = {5 + 3}")
print(f"Division: 5 / 3 = {5 / 3}")
print(f"Floor Division: 5 // 3 = {5 // 3}")
print(f"Exponentiation: 5 ** 3 = {5 ** 3}")
```

6.3.2 Comparison Operators

These operators are used to compare values:

Operator	Description	Example
`==`	Equal to	`x == y`
`!=`	Not equal to	`x != y`
`>`	Greater than	`x > y`
`<`	Less than	`x < y`
`>=`	Greater than or equal to	`x >= y`
`<=`	Less than or equal to	`x <= y`

```
#| autorun: false
# Examples of comparison operators
x, y = 5, 3
print(f"x = {x}, y = {y}")
print(f"x == y: {x == y}")
print(f"x > y: {x > y}")
print(f"x <= y: {x <= y}")
```

6.3.3 Logical Operators

Used to combine conditional statements:

Operator	Description	Example
`and`	Returns True if both statements are true	`x > 0 and x < 10`
`or`	Returns True if one of the statements is true	`x < 0 or x > 10`
`not`	Reverses the result, returns False if the result is true	`not(x > 0 and x < 10)`

```
#| autorun: false
# Examples of logical operators
x = 7
print(f"x = {x}")
print(f"x > 0 and x < 10: {x > 0 and x < 10}")
print(f"x < 0 or x > 10: {x < 0 or x > 10}")
print(f"not(x > 0): {not(x > 0)}")
```

6.3.4 Assignment Operators

Python provides shorthand operators for updating variables:

Operator	Example	Equivalent to
`=`	`x = 5`	`x = 5`
`+=`	`x += 3`	`x = x + 3`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 3`	`x = x * 3`
`/=`	`x /= 3`	`x = x / 3`
`//=`	`x //= 3`	`x = x // 3`
`%=`	`x %= 3`	`x = x % 3`
`**=`	`x **= 3`	`x = x ** 3`

```
#| autorun: false
# Examples of assignment operators
```

```
x = 10
print(f"Initial x: {x}")

x += 5
print(f"After x += 5: {x}")

x *= 2
print(f"After x *= 2: {x}")
```

Operator Precedence

Python follows the standard mathematical order of operations (PEMDAS):

1. Parentheses
2. Exponentiation (`**`)
3. Multiplication and Division (`*`, `/`, `//`, `%`)
4. Addition and Subtraction (`+`, `-`)

When operators have the same precedence, they are evaluated from left to right.

```
#| autorun: false
# Operator precedence example
result = 2 + 3 * 4 ** 2
print(f"2 + 3 * 4 ** 2 = {result}")  # 2 + 3 * 16 = 2 + 48 = 50

# Using parentheses to change precedence
result = (2 + 3) * 4 ** 2
print(f"(2 + 3) * 4 ** 2 = {result}")  # 5 * 16 = 80
```