Introduction to Computer-based Physical Modeling

Frank Cichos

2024-08-14

This book is a collection of lecture notes for the course "Introduction to Computer-based Physical Modeling" at the University of Leipzig. The course is part of the Bachelor's program in Physics and is intended for all students. The course covers the basics of computer-based physical modeling, including numerical methods, simulation techniques, and data analysis. The book is designed to be used as a reference for students taking the course, as well as for anyone interested in learning more about computer-based physical modeling.

Table of contents

1 My Document

Welcome

Welcome ot the Introduction to Computation Physics Course.

This document has been created with Quarto. To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

Course Info

2 Course Information

Part II

Exam

3 Exam

Part III

Lecture Contents

4 Lecture 1

5 Introduction to Python and Basic Calculations

6 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.

6.1 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:

6.1.1 Notebook Editor

The Notebook editor is an interactive web-based application for creating and editing notebook documents. It allows users to write and run code interactively, as well as add rich text and multimedia content. When running Jupyter on a server, you'll typically use either the classic Jupyter Notebook interface or JupyterLab, which is a more advanced and feature-rich version of the notebook editor.

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.

6.1.2 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.

6.1.3 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.

i 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.

6.2 Using the Notebook Editor

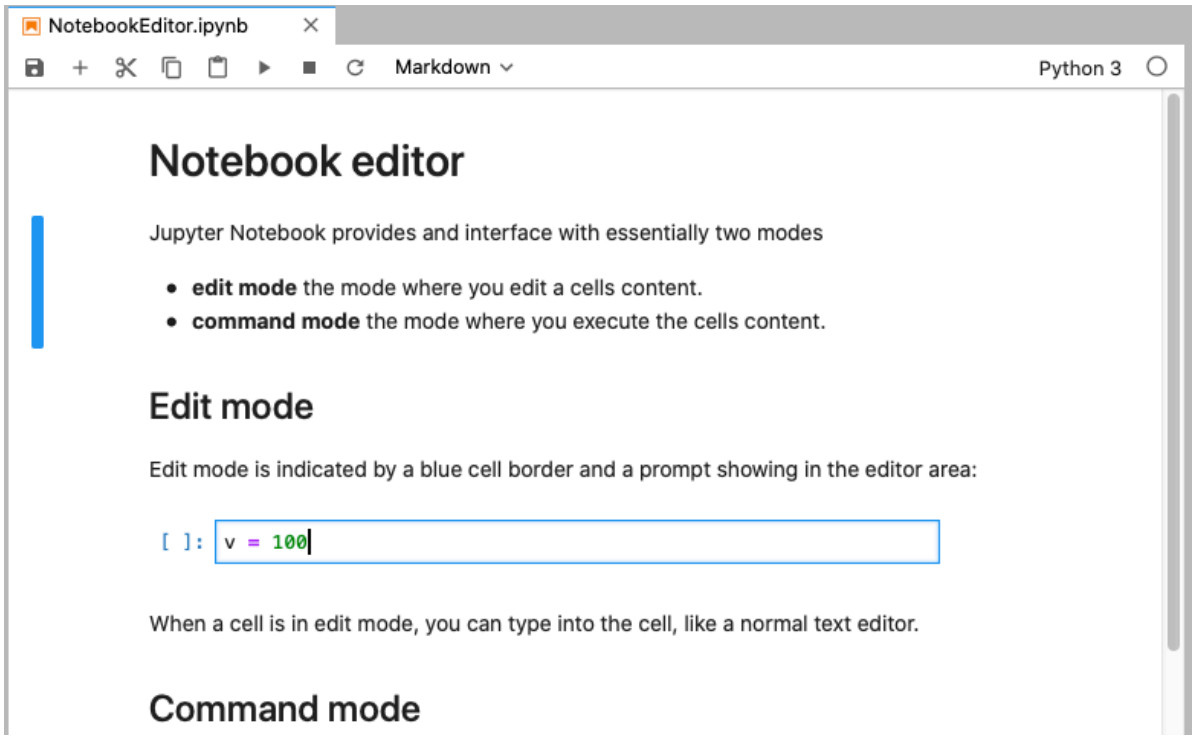

Figure 6.1: Jupyter Notebook Editor

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.

6.2.1 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 6.2: Edit Mode

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

6.2.2 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.

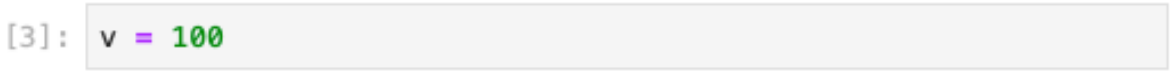

Figure 6.3: Command Mode

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.

6.2.3 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**

6.2.4 Running code in your notebook

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.

```
v = 23752636
print(v)
```

```
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.

6.3 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".

Entering code is pretty easy. You just have to click into a cell and type the commands you want to type. If you have multiple lines of code, just press **enter** at the end of the line and start a new one.

- **code blocks** Python identifies blocks of codes belonging together by its identation. This will become important if you write longer code in a cell later. To indent the block, you may use either *whitespaces* or *tabs*.
- **comments** Comments can be added to annotate the code, such that you or someone else can understand the code.
  - Comments in a single line are started with the `#` character at in front of the comment.
  - Comments over multiple lines can be started with `'''`and end with the same `'''`. They are used as `docstrings` to provide a help text.

```
# typical function

def function(x):
    ''' function to calculate a function
    arguments:
        x ... float or integer value
    returns:
        y ... two times the integer value
    '''
    y=2*x # don't forget the identation of the block
    return(y)

help(function)
```

```
Help on function function in module __main__:

function(x)
    function to calculate a function
    arguments:
        x ... float or integer value
    returns:
        y ... two times the integer value
```

6.4 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://daringfireball.net/projects/markdown/

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.

6.4.1 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

6.4.2 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
```

6.4.3 Embedded code

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

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

6.4.4 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 `$$`:

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

6.4.5 Images

Images may be also directly integrated into a Markdown block.

To include images use

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

for example

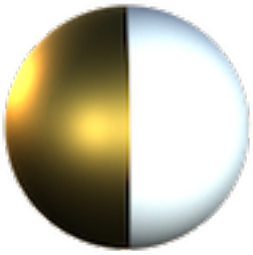

Figure 6.4: alternative text

6.4.6 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.

```
from IPython.display import YouTubeVideo
YouTubeVideo('QlLx32juGzI',width=600)
```

```
<IPython.lib.display.YouTubeVideo at 0x12e408380>
```

6.5 Variables in Python

6.5.1 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.

6.5.2 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:

```
# variable assignments
x = 1.0
my_favorite_variable = 12.2
x
```

```
1.0
```

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.

```
type(x)
```

```
float
```

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

```
x = 1
```

```
type(x)
```

```
int
```

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

```
#print(g)
```

6.6 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.

6.6.1 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

6.6.2 Integers

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

```
x = 1
type(x)
```

```
int
```

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

```
0b1010111110  # Binary
0x0F          # Hexadecimal
```

```
15
```

6.6.3 Floating Point Numbers

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

```
x = 3.141
type(x)
```

```
float
```

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

```
1.7976931348623157e+308 * 2  # Output: inf
```

```
inf
```

6.6.4 Complex Numbers

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

```
c = 2 + 4j
type(c)
```

```
complex
```

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

```
print(c.real)
print(c.imag)
```

```
2.0
4.0
```

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

```
c = c.conjugate()
print(c)
```

```
(2-4j)
```

6.7 Type Conversion

6.7.1 Implicit Type Conversion

Python automatically converts types during operations involving mixed types. The result is a type that can accommodate all the values involved in the operation. This is known as implicit type conversion. For example, adding an integer and a float results in a float.

```
integer_number = 123
float_number = 1.23
new_number = integer_number + float_number
type(new_number)
```

```
float
```

6.7.2 Explicit Type Conversion

Use functions like `int()`, `float()`, and `str()` to explicitly convert types from one to another. Explicit type conversion is also known as type casting. For example, converting a string to an integer.

```
num_string = "12"
num_string = int(num_string)
type(num_string)
```

`int`

Converting a float to an integer truncates the decimal part.

```
x = 5 / 2
x = int(x)
z = complex(x)
```

Handling Complex Numbers

Complex numbers cannot be directly converted to floats or integers; extract the real or imaginary part first.

```
y = bool(z.real)
print(z.real, " -> ", y, type(y))
```

```
2.0  ->  True <class 'bool'>
```

6.8 Application

The following code snippets demonstrate the use of variables and basic arithmetic operations in Python. The code snippets include examples of variable assignments, type conversion, and arithmetic operations.

```
# 1. Converting Units of Distance
# Distance in kilometers
distance_km = 5.0

# Conversion factor from kilometers to meters
conversion_factor = 1000

# Convert distance to meters
distance_meters = distance_km * conversion_factor

# 2. Calculating Time from Distance and Speed
# Given distance in meters
distance = 1000.0  # meters

# Given speed in meters per second
speed = 5.0  # meters per second
```

```
# Calculate time in seconds
time = distance / speed

# 3. Energy Calculation Using Kinetic Energy Formula
# Mass in kilograms
mass = 70.0  # kg

# Velocity in meters per second
velocity = 10.0  # m/s

# Calculate kinetic energy
kinetic_energy = 0.5 * mass * velocity ** 2

# 4. Temperature Conversion (Celsius to Fahrenheit)
# Temperature in Celsius
temp_celsius = 25.0  # degrees Celsius

# Convert to Fahrenheit
temp_fahrenheit = (temp_celsius * 9/5) + 32

# 5. Power Calculation Using Work and Time
# Work done in joules
work_done = 500.0  # joules

# Time taken in seconds
time_taken = 20.0  # seconds

# Calculate power
power_output = work_done / time_taken

# 6. Calculating Force Using Newton's Second Law
# Mass in kilograms
mass = 10.0  # kg

# Acceleration in meters per second squared
acceleration = 9.8  # m/s^2

# Calculate force
force = mass * acceleration

# Output Results
print(f"Distance in meters: {distance_meters} m")
```

```
print(f"Time to travel {distance} meters at {speed} m/s: {time} seconds")
print(f"Kinetic energy: {kinetic_energy} joules")
print(f"Temperature in Fahrenheit: {temp_fahrenheit} °F")
print(f"Power output: {power_output} watts")
print(f"Force: {force} newtons")
```

```
Distance in meters: 5000.0 m
Time to travel 1000.0 meters at 5.0 m/s: 200.0 seconds
Kinetic energy: 3500.0 joules
Temperature in Fahrenheit: 77.0 °F
Power output: 25.0 watts
Force: 98.0 newtons
```

- Simple calculations relevant to physics, such as converting units or calculating simple quantities (e.g., distance = speed × time).
  - Homework: Basic practice problems to reinforce Python syntax and operations.

7 Lecture 2

8 Kinematics and Python

8.1 Introduction to 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.

8.1.1 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:

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

8.1.2 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:

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

```
5
```

8.2 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.

8.2.1 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:

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

print_numbers()
```

```
1
2
3
4
5
6
7
8
9
10
```

8.2.2 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:

```
# 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()
```

```
1
2
3
4
5
6
7
8
9
10
```

8.3 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`.

8.3.1 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:

```
# 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()
```

```
hello
```

8.3.2 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:

```
# 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()
```

```
hello
```

8.4 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:

```
import math
import numpy

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

print(x)
```

```
2.5066282746310002
```

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:

```
from math import *

x = cos(2 * pi)

print(x)
```

```
1.0
```

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.

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:

8.4.0.1 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:

```
# mymodule.py

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

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

8.4.0.2 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:

```
# main.py

import mymodule

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

8.4.1 Importing Specific Functions

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

```
# main.py

from mymodule import greet, add

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

8.4.2 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:

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

8.4.3 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.

8.4.4 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.

```
import math as m

x = m.sqrt(2)

print(x)
```

```
1.4142135623730951
```

You may also only import specific functions of a module.

```
from math import sinh as mysinh
```

8.4.5 Contents of a module

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

```
import math

print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh'
```

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).

```
help(math.log)
```

```
Help on built-in function log in module math:

log(...)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base is not specified, returns the natural logarithm (base e) of x.
```

```
math.log(10)
```

```
2.302585092994046
```

```
math.log(8, 2)
```

```
3.0
```

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/ .

Namespaces in Your Modules

8.4.6 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())
```

Python Application

8.5 Function Plotting

- Writing a Python function to calculate and plot the position vs. time for an object moving with constant velocity or constant acceleration.
- Visualization: Use `matplotlib` to plot simple kinematic graphs (position vs. time, velocity vs. time).
- Homework: Extend the kinematic function to handle different initial conditions and plot the results.

9 Lecture 3

10 Dynamics and Simple Motion

- Lists and arrays (introduction to `numpy` for numerical operations).
- Basic vector operations using `numpy`.

> **Numpy Array**
>
> 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. For example, they might be all integers or all floating point numbers. NumPy arrays are more efficient than Python lists for storing and manipulating data.

```
#| slideshow: {slide_type: fragment}
import numpy as np
```

10.1 Creating Numpy Arrays

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

- a Python list or tuples using `np.array`
- 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 of this course.

10.1.1 From lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function. This is demonstrated in the following cells:

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

type(a)
```

```
list
```

```
b=np.array(a,dtype=float)
type(b)
```

```
numpy.ndarray
```

10.1.2 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 and shapes. Some of the more common are:

```
#| slideshow: {slide_type: fragment}
# create a range

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

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

```
#| slideshow: {slide_type: fragment}
x = np.arange(-5, -2, 0.1)
x
```

```
array([-5. , -4.9, -4.8, -4.7, -4.6, -4.5, -4.4, -4.3, -4.2, -4.1, -4. ,
       -3.9, -3.8, -3.7, -3.6, -3.5, -3.4, -3.3, -3.2, -3.1, -3. , -2.9,
       -2.8, -2.7, -2.6, -2.5, -2.4, -2.3, -2.2, -2.1])
```

> **linspace**
>
> 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. The function `linspace` always includes the end points.

```
#| slideshow: {slide_type: fragment}
# using linspace, both end points ARE included
np.linspace(0,10,25)
```

```
array([ 0.        ,  0.41666667,  0.83333333,  1.25      ,  1.66666667,
        2.08333333,  2.5       ,  2.91666667,  3.33333333,  3.75      ,
        4.16666667,  4.58333333,  5.        ,  5.41666667,  5.83333333,
        6.25      ,  6.66666667,  7.08333333,  7.5       ,  7.91666667,
        8.33333333,  8.75      ,  9.16666667,  9.58333333, 10.        ])
```

i logspace

`logspace` is doing equivelent things with logaritmic spacing. The function `logspace` generates an array of N points between decades 10^start and 10^stop. The form of the function is logspace(start, stop, N). If the third argument N is omitted, then N=50. The function `logspace` always includes the end points.

```
#| slideshow: {slide_type: fragment}
np.logspace(0, 10, 10, base=np.e)
```

```
array([1.00000000e+00, 3.03773178e+00, 9.22781435e+00, 2.80316249e+01,
       8.51525577e+01, 2.58670631e+02, 7.85771994e+02, 2.38696456e+03,
       7.25095809e+03, 2.20264658e+04])
```

Other types of array creation techniques are listed below. Try around with these commands to get a feeling what they do.

i mgrid

`mgrid` generates a multi-dimensional matrix with increasing value entries, for example in columns and rows. The arguments are similar to `arange` and `linspace`.

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

```
x
```

```
array([[0. , 0. , 0. , 0. , 0. ],
       [0.1, 0.1, 0.1, 0.1, 0.1],
       [0.2, 0.2, 0.2, 0.2, 0.2],
       [0.3, 0.3, 0.3, 0.3, 0.3],
       [0.4, 0.4, 0.4, 0.4, 0.4],
       [0.5, 0.5, 0.5, 0.5, 0.5],
       [0.6, 0.6, 0.6, 0.6, 0.6],
       [0.7, 0.7, 0.7, 0.7, 0.7],
       [0.8, 0.8, 0.8, 0.8, 0.8],
       [0.9, 0.9, 0.9, 0.9, 0.9]])
```

```
y
```

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

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

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

i diag

diag generates a diagonal matrix with the list supplied to it as the diagonal values. The values can be also offset from the main diagonal by using the optional argument **k**. If **k** is positive, the diagonal is above the main diagonal, if negative, below the main diagonal.

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

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

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

```
array([[0, 0, 0, 0],
       [1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0]])
```

> **i** zeros and ones
>
> `zeros` and `ones` creates a matrix with the dimensions given in the argument and filled with 0 or 1. The argument is a tuple with the dimensions of the matrix. For example, `np.zeros((3,3))` creates a 3x3 matrix filled with zeros.

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

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

```
np.ones((3,3))
```

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

10.2 Manipulating NumPy arrays

10.2.1 Slicing

Slicing is the name for extracting part of an array by the syntax `M[lower:upper:step]`. When any of these are unspecified, they default to the values `lower=0`, `upper=size of dimension`, `step=1`. We can also use negative indices to count from the end of the array. Here are some examples:

```
A = np.array([1,2,3,4,5])
A
```

```
array([1, 2, 3, 4, 5])
```

```
A[1:4]
```

```
array([2, 3, 4])
```

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

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

```
array([1, 2, 3, 4, 5])
```

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

```
array([1, 3, 5])
```

Negative indices counts from the end of the array (positive index from the begining) and can be used in any of the three slicing parameters. Here are some examples:

```
A = np.array([1,2,3,4,5])
```

```
A[-1] # the last element in the array
```

```
np.int64(5)
```

```
A[2:] # the last three elements
```

```
array([3, 4, 5])
```

Index slicing works exactly the same way for multidimensional arrays. We can slice along each axis independently. Here are some examples:

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
```

```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

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

```
array([[11, 12, 13],
       [21, 22, 23]])
```

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 the formula

$$v_i = \frac{y_i - y_{i-1}}{t_i - t_{i-1}} \tag{10.1}$$

```
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 ])
```

```
v = (y[1:]-y[:-1])/(t[1:]-t[:-1])
v
```

```
array([ 2.65306122,  7.25490196, 11.8       , 13.79310345, 20.85106383,
       17.38461538])
```

10.2.2 Reshaping

Arrays can be reshaped into any form, which contains the same number of elements. For example, a 4-element array can be reshaped into a 2x2 array, or a 2x2 array can be reshaped into a 4-element array. Here are some examples:

```
a=np.zeros(4)
a
```

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

```
np.reshape(a,(2,2))
```

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

10.2.3 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. Here are some examples:

```
#| slideshow: {slide_type: fragment}
v = np.array([1,2,3])
v
```

```
array([1, 2, 3])
```

```
#| slideshow: {slide_type: fragment}
v.shape
```

```
(3,)
```

```
#| slideshow: {slide_type: fragment}
# make a column matrix of the vector v
v[:, np.newaxis]
```

```
array([[1],
       [2],
       [3]])
```

```
#| slideshow: {slide_type: fragment}
# column matrix
v[:,np.newaxis].shape
```

```
(3, 1)
```

```
#| slideshow: {slide_type: fragment}
# row matrix
v[np.newaxis,:].shape
```

```
(1, 3)
```

10.2.4 Stacking and repeating arrays

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

10.2.4.1 Tile and repeat

```
#| slideshow: {slide_type: skip}
a = np.array([[1, 2], [3, 4]])
a
```

```
array([[1, 2],
       [3, 4]])
```

```
#| slideshow: {slide_type: skip}
# repeat each element 3 times
np.repeat(a, 3)
```

```
array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])
```

```
#| slideshow: {slide_type: skip}
# tile the matrix 3 times
np.tile(a, 3)
```

```
array([[1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4]])
```

10.2.4.2 Concatenate

Concatenate joins arrays along an existing axis. Here are some examples:

```
#| slideshow: {slide_type: skip}
a = np.array([[1, 2], [3, 4]])
a
```

```
array([[1, 2],
       [3, 4]])
```

```
#| slideshow: {slide_type: skip}
b = np.array([[5, 6]])
```

```
#| slideshow: {slide_type: skip}
np.concatenate((a, b), axis=0)
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

```
#| slideshow: {slide_type: skip}
np.concatenate((a, b.T), axis=1)
```

```
array([[1, 2, 5],
       [3, 4, 6]])
```

10.2.4.3 Hstack and vstack

hstack and vstack stack arrays horizontally and vertically. Here are some examples:

```
#| slideshow: {slide_type: skip}
a = np.array([1, 2, 3])
b = np.array([2, 3, 4])
```

```
#| slideshow: {slide_type: skip}
np.vstack((a,b))
```

```
array([[1, 2, 3],
       [2, 3, 4]])
```

```
#| slideshow: {slide_type: skip}
np.hstack((a,b.T))
```

```
array([1, 2, 3, 2, 3, 4])
```

10.3 Applying mathematical functions

All kinds of mathematical operations can be carried out on arrays. Typically these operation act element wise as seen from the examples below where `a` is an array of numbers from 0 to 9.

10.3.1 Operation involving one array

```
#| slideshow: {slide_type: fragment}
a=np.arange(0, 10, 1.5)
a
```

```
array([0. , 1.5, 3. , 4.5, 6. , 7.5, 9. ])
```

```
#| slideshow: {slide_type: fragment}
a/2
```

```
array([0.  , 0.75, 1.5 , 2.25, 3.  , 3.75, 4.5 ])
```

```
#| slideshow: {slide_type: fragment}
a**2
```

```
array([ 0.  ,  2.25,  9.  , 20.25, 36.  , 56.25, 81.  ])
```

```
#| slideshow: {slide_type: fragment}
np.sin(a)
```

```
array([ 0.        ,  0.99749499,  0.14112001, -0.97753012, -0.2794155 ,
        0.93799998,  0.41211849])
```

```
#| slideshow: {slide_type: fragment}
np.exp(-a)
```

```
array([1.00000000e+00, 2.23130160e-01, 4.97870684e-02, 1.11089965e-02,
       2.47875218e-03, 5.53084370e-04, 1.23409804e-04])
```

```
#| slideshow: {slide_type: fragment}
(a+2)/3
```

```
array([0.66666667, 1.16666667, 1.66666667, 2.16666667, 2.66666667,
       3.16666667, 3.66666667])
```

10.3.2 Operations involving multiple arrays

Operation between multiple vectors allow in particular very quick operations. The operations address then elements of the same index. These operations are called vector operations since the concern the whole array at the same time. The product between two vectors results therefore not in a dot product, which gives one number but in an array of multiplied elements.

```
#| ExecuteTime: {end_time: '2017-04-20T21:16:34.978152+02:00', start_time: '2017-04-20T21:1
#| slideshow: {slide_type: fragment}
a = np.array([34., -12, 5.,1.2])
b = np.array([68., 5.0, 20.,40.])
```

```
#| ExecuteTime: {end_time: '2017-04-20T21:16:41.810170+02:00', start_time: '2017-04-20T21:1
#| slideshow: {slide_type: fragment}
a + b
```

```
array([102. ,  -7. ,  25. ,  41.2])
```

```
#| ExecuteTime: {end_time: '2017-04-20T21:16:48.002366+02:00', start_time: '2017-04-20T21:1
#| slideshow: {slide_type: fragment}
#| tags: []
2*b
```

```
array([136.,  10.,  40.,  80.])
```

xxw

```
#| ExecuteTime: {end_time: '2017-04-20T21:20:19.373091+02:00', start_time: '2017-04-20T21:2
#| slideshow: {slide_type: fragment}
a*np.exp(-b)
```

```
array([ 9.98743918e-29, -8.08553640e-02,  1.03057681e-08,  5.09802511e-18])
```

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

10.4 Application

- Simulating and plotting the trajectory of a projectile under the influence of gravity (2D motion).
- Introduction to vector addition and resolving vectors into components.
- Visualization: Plotting the path of the projectile and velocity vectors.
- Homework: Simulate projectile motion with air resistance (optional for advanced students).

11 Lecture 4

11.1 Work, Power, Energy

11.1.1 Python Concepts

- Introduction to loops for numerical integration (e.g., trapezoidal rule).
- Functions for calculating work, power, and energy.

11.1.2 Application

- Writing Python code to calculate the work done by a variable force (e.g., spring force) using numerical integration.
- Simulating energy conservation in a closed system (e.g., pendulum).
- Visualization: Plotting energy vs. time for the system.
- Homework: Modify the code to simulate a different system, such as a mass-spring system.

12 Lecture 5

12.1 Angular Momentum and Rotational Motion

12.1.1 Python Concepts

- Introduction to classes in Python (optional, for organizing code).
- Rotational kinematics and dynamics (moment of inertia, angular momentum).

12.1.2 Application

- Simulating the motion of a rotating object (e.g., a spinning disk) and calculating its angular momentum.
- Visualizing the effect of torque on the object's rotation.
- Homework: Extend the simulation to include the effect of external forces, such as friction.

13 Lecture 6

13.1 Collisions and Conservation Laws

13.1.1 Python Concepts

- Advanced conditionals and loops for simulating complex scenarios.
- Introduction to basic data structures (dictionaries) for handling multiple objects.

13.1.2 Application

- Simulating elastic and inelastic collisions in one and two dimensions.
- Applying conservation laws (momentum and energy) to check the validity of the simulation.
- Visualization: Plotting the trajectories and velocities of colliding objects.
- Homework: Simulate a multi-object collision scenario.

14 Lecture 7

14.1 Gravitation and Orbital Mechanics

14.1.1 Python Concepts

- Solving differential equations using Python (Euler's method for numerical integration).
- Using `scipy` for more advanced numerical methods.

14.1.2 Application

- Simulating gravitational forces and orbital motion (e.g., a planet orbiting a star).
- Visualizing the orbit and calculating parameters like orbital period and eccentricity.
- Homework: Modify the simulation to include multiple bodies (e.g., a three-body problem).

15 Lecture 8

15.1 Rigid Bodies and Torque

15.1.1 Python Concepts

- Introduction to 2D arrays and matrix operations in `numpy`.
- Using matrix methods to solve systems of equations related to rigid body dynamics.

15.1.2 Application

- Simulating the motion of a rigid body under the influence of forces and torques.
- Visualizing the rotation and translation of the body.
- Homework: Simulate a rigid body with varying moments of inertia (e.g., a non-uniform object).

16 Lecture 9

16.1 Deformation of Solids

16.1.1 Python Concepts

- Basic file handling for reading and writing data (e.g., stress-strain data).
- Curve fitting using `numpy` or `scipy` to analyze experimental data.

16.1.2 Application

- Simulating the deformation of a solid under load (e.g., Hooke's law for springs).
- Plotting stress-strain curves and fitting them to experimental data.
- Homework: Extend the simulation to include plastic deformation or fracture.

17 Lecture 10

17.1 Fluid Dynamics and Surface Tension

17.1.1 Python Concepts

- Introduction to flow simulation and basic fluid mechanics principles (continuity equation, Bernoulli's equation).
- Using `matplotlib` for more advanced visualizations (e.g., vector fields).

17.1.2 Application

- Simulating fluid flow in simple scenarios (e.g., flow through a pipe, around an object).
- Visualizing pressure and velocity fields.
- Homework: Simulate and visualize a more complex fluid system, such as flow over a wing.

18 Lecture 11

18.1 Oscillations and Waves

18.1.1 Python Concepts

- Solving coupled differential equations for oscillatory systems (e.g., damped harmonic oscillators).
- Using Fourier analysis to study waveforms.

18.1.2 Application

- Simulating simple harmonic motion and damped oscillations.
- Visualizing the motion and analyzing the frequency components of the oscillation.
- Homework: Extend the simulation to include coupled oscillators or waves on a string.

19 Lecture 12

19.1 Advanced Waves and Final Project Introduction

19.1.1 Python Concepts

- Introduction to more advanced simulation techniques (e.g., finite difference methods).
- Structuring a Python project (modules, documentation, testing).

19.1.2 Application

- Simulating wave propagation, reflection, and transmission in different media.
- Visualization: Animating wave motion and energy transfer.

19.1.2.1 Final Project Overview

```
- Introduction to the final project, where students choose a physics problem to model and so
- Discussion of project expectations, timelines, and resources.
- Homework: Start working on the final project by selecting a topic and outlining the approa
```

```
## Ideal Gas Simulation with Pressure, Gravity, and Collisions

Use the sliders below to adjust the speed of the gas particles and the gravitational force a
```