Introduction to Computer-based Physical Modeling

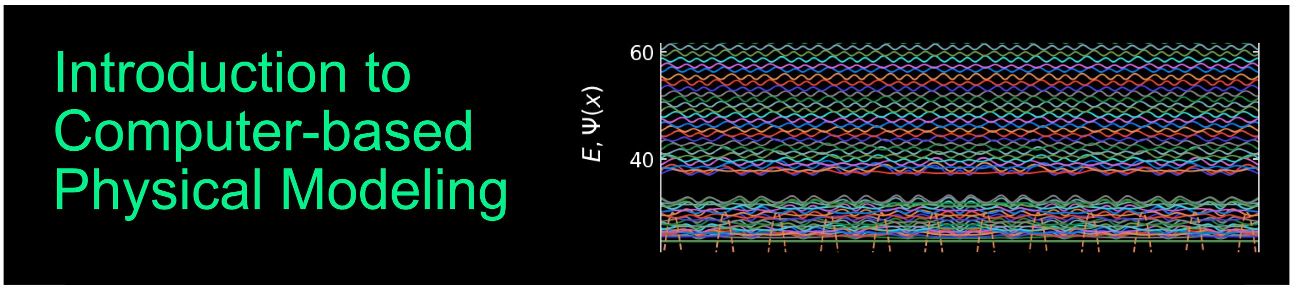
Frank Cichos

2024-08-14

Table of contents

Chapter 1

CBPM 2025



Chapter 2

Welcome to 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.

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.

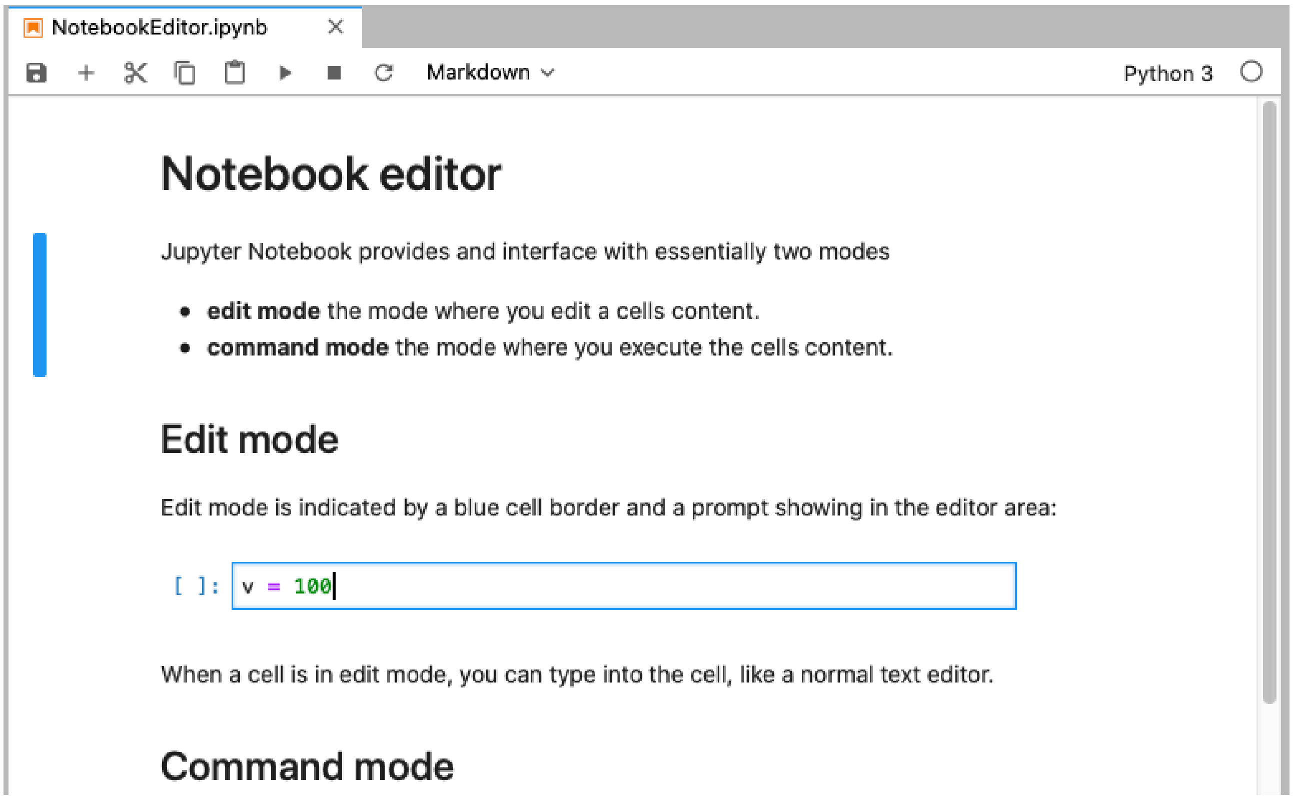
Advanced Notebook Documents Info

- 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



Notebook editor

Jupyter Notebook provides and interface with essentially two modes

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

Edit mode

Edit mode is indicated by a blue cell border and a prompt showing in the editor area:

```
[ ]: v = 100
```

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

Command mode

Figure 4.1: Jupyter Notebook Editor

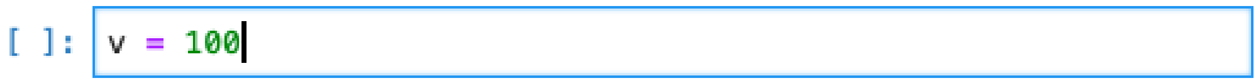
The Jupyter Notebook editor provides an interactive environment for writing code, creating visualizations, and documenting computational workflows. It consists of a web-based interface that allows users to create and edit notebook documents containing code, text, equations, images, and interactive elements.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



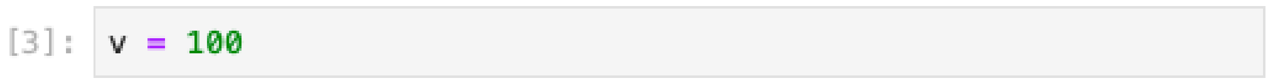
```
[ ]: v = 100
```

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.



```
[3]: v = 100
```

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

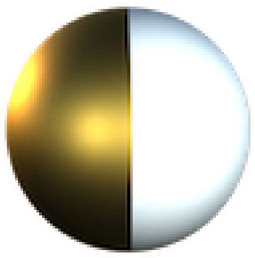


Figure 4.4: alternative text

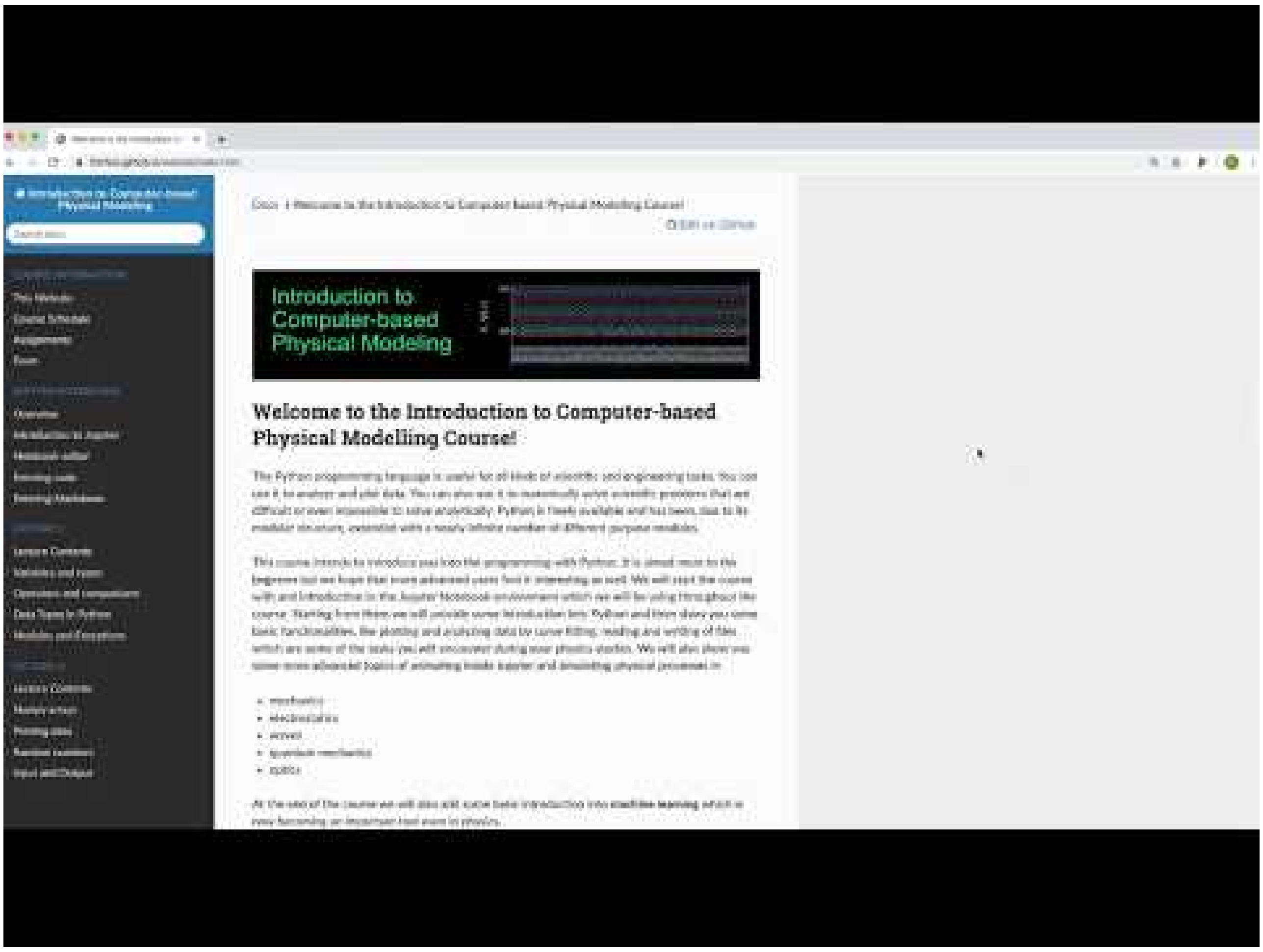
4.4.9 Videos

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!

The Python programming language is useful for all kinds of scientific and engineering tasks. You can use it to analyze and plot data. You can also use it to numerically solve scientific problems that are difficult or even impossible to solve analytically. Python is freely available and has been, due to its modular structure, extended with a nearly infinite number of different purpose modules.

This course intends to introduce you into the programming with Python. It is aimed more to the beginner but we hope that more advanced users find it interesting as well. We will start the course with and introduction to the Jupyter Notebook environment which we will be using throughout the course. Starting from there we will provide some introduction into Python and then show you some basic functionalities, like plotting and analyzing data by curve fitting, reading and writing of files which are some of the tasks you will encounter during your physics studies. We will also show you some more advanced topics of animating inside Jupyter and simulating physical processes in

- mechanics
- electrostatics
- waves
- quantum mechanics
- optics

At the end of the course we will also add some basic introduction into **machine learning** which is now becoming an important tool even in physics.

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

```
# 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

6.2.2 Examples

6.2.3 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
```

6.2.4 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
```

6.2.5 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
```

Part II

Lecture 2

Chapter 7

Data Types

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 Types	Classes	Description
Numeric	int, float, complex	Holds numeric values
String	str	Stores sequence of characters
Sequence	list, tuple, range	Stores ordered collection of items
Mapping	dict	Stores data as key-value pairs
Boolean	bool	Holds either True or False
Set	set, frozenset	Holds collection of unique items

Each of these data types has a number of connected `methods` (functions) which allow you 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 output would be:

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix',
'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

```
#| autorun: false
s = "string"
dir(s)
```

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

7.1 Data Types

7.1.1 Numeric Types

Python supports several numeric data types including **integers**, **floats**, and **complex** numbers.

```
#| autorun: false

x = 10      # integer
y = 3.14    # float
z = 2+3j    # complex number

type(x), type(y), type(z)
```

You can perform various arithmetic operations with numeric types:

```
#| autorun: false
# Basic arithmetic
print(x + y)  # Addition
print(x - y)  # Subtraction
print(x * y)  # Multiplication
print(x / y)  # Division
```

Type conversion works between numeric types:

```
#| autorun: false
# Converting between numeric types
int_num = int(3.9)    # Truncates to 3
float_num = float(5)  # Converts to 5.0

print(int_num, float_num)
```

7.1.2 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
# you may use as well str.isnumeric("975"+"321")
("975"+"321").isnumeric()
```

```
#| 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()
```

7.1.3 Lists

Lists are ordered, mutable collections that can store items of different data types.

```
#| autorun: false
my_list = [1, 2.5, "hello", True]
print(type(my_list))
print(my_list)
```

You can access and modify list elements:

```
#| autorun: false
# Accessing elements
print(my_list[0])       # First element
print(my_list[-1])      # Last element
print(my_list[1:3])     # Slicing

# Modifying elements
my_list[0] = 100
print(my_list)

# Adding elements
my_list.append("new item")
print(my_list)
```

Common list methods:

```
#| autorun: false
sample_list = [3, 1, 4, 1, 5, 9]

sample_list.sort()    # Sort the list in-place
print(sample_list)
```

```
sample_list.reverse() # Reverse the list in-place
print(sample_list)

print(len(sample_list))  # Get the length of the list
```

7.1.4 Tuples

Tuples are ordered, immutable sequences.

```
#| autorun: false
my_tuple = (1, 2, "three", 4.0)
print(type(my_tuple))
print(my_tuple)
```

Tuples are immutable, meaning you cannot change their elements after creation:

```
#| autorun: false
# Accessing tuple elements
print(my_tuple[0])
print(my_tuple[-1])
print(my_tuple[1:3])

# This would cause an error
# my_tuple[0] = 100  # TypeError: 'tuple' object does not support item assignment
```

7.1.5 Dictionaries

Dictionaries store data as key-value pairs. They are mutable and unordered.

```
#| autorun: false
student = {
    "name": "Alice",
    "age": 21,
    "courses": ["Math", "Physics", "Computer Science"],
    "active": True
}

print(type(student))
print(student)
```

Accessing and modifying dictionary elements:

```
#| autorun: false
# Accessing values
print(student["name"])
print(student.get("age"))  # Safer method if key might not exist

# Modifying values
student["age"] = 22
print(student)

# Adding new key-value pair
student["graduation_year"] = 2023
print(student)

# Removing key-value pair
del student["active"]
print(student)
```

Common dictionary methods:

```
#| autorun: false
# Get all keys and values
print(student.keys())
print(student.values())
print(student.items())  # Returns key-value pairs as tuples
```

7.1.6 Boolean

The **Boolean** type has only two possible values: `True` and `False`.

```
#| autorun: false
x = True
y = False
print(type(x), x)
print(type(y), y)
```

Boolean values are commonly used in conditional statements:

```
#| autorun: false
age = 20
is_adult = age >= 18
print(is_adult)

if is_adult:
    print("Person is an adult")
else:
    print("Person is a minor")
```

Boolean operations:

```
#| autorun: false
a = True
b = False

print(a and b)  # Logical AND
print(a or b)   # Logical OR
print(not a)    # Logical NOT
```

7.1.7 Sets

Sets are unordered collections of unique elements.

```
#| autorun: false
my_set = {1, 2, 3, 4, 5}
print(type(my_set))
print(my_set)

# Duplicates are automatically removed
duplicate_set = {1, 2, 2, 3, 4, 4, 5}
print(duplicate_set)  # Still {1, 2, 3, 4, 5}
```

Common set operations:

```
#| autorun: false
set_a = {1, 2, 3, 4, 5}
set_b = {4, 5, 6, 7, 8}
```

```
# Union
print(set_a | set_b)  # or set_a.union(set_b)

# Intersection
print(set_a & set_b)  # or set_a.intersection(set_b)

# Difference
print(set_a - set_b)  # or set_a.difference(set_b)

# Symmetric difference
print(set_a ^ set_b)  # or set_a.symmetric_difference(set_b)
```

Adding and removing elements:

```
#| autorun: false
fruits = {"apple", "banana", "cherry"}

# Adding elements
fruits.add("orange")
print(fruits)

# Removing elements
fruits.remove("banana")  # Raises error if element doesn't exist
print(fruits)

fruits.discard("kiwi")   # No error if element doesn't exist
```

7.2 Type Casting

Type casting is the process of converting a value from one data type to another. Python provides built-in functions for type conversion.

Python offers several built-in functions for type conversion: - `int()`: Converts to integer - `float()`: Converts to float - `str()`: Converts to string - `bool()`: Converts to boolean - `list()`: Converts to list - `tuple()`: Converts to tuple - `set()`: Converts to set - `dict()`: Converts from mappings or iterables of key-value pairs

Let's explore various type conversion examples with practical code demonstrations. These examples show how Python handles conversions between different data types.

Numeric Conversions

When converting between numeric types, it's important to understand how precision and data may change. For example, converting floats to integers removes the decimal portion without rounding.

```
#| autorun: false
# Numeric conversions
print(int(3.7))       # Float to int (truncates decimal part)
print(float(5))       # Int to float
print(complex(3, 4))  # Creating complex number
```

String Conversions

String conversions are commonly used when processing user input or preparing data for output. Python provides straightforward functions for converting between strings and numeric types.

```
#| autorun: false
# String conversions
print(str(123))       # Number to string
print(int("456"))     # String to number
print(float("3.14"))  # String to float
```

Collection Type Conversions

Python allows for easy conversion between different collection types, which is useful for changing the properties of your data structure (like making elements unique with sets).

```
#| autorun: false
# Collection type conversions
print(list("Python"))             # String to list
print(tuple([1, 2, 3]))          # List to tuple
print(set([1, 2, 2, 3, 3, 3]))   # List to set (removes duplicates)
```

Boolean Conversion

Boolean conversion is essential for conditional logic. Python follows specific rules to determine truthiness of values, with certain "empty" or "zero" values converting to False.

When converting to boolean with `bool()`, the following values are considered False: - `0` (integer) - `0.0` (float) - `""` (empty string) - `[]` (empty list) - `()` (empty tuple) - `{}` (empty dictionary) - `set()` (empty set) - `None`

Everything else converts to `True`.

```
#| autorun: false
# Boolean conversions
print(bool(0))      # False
print(bool(1))      # True
print(bool(""))     # False
print(bool("text")) # True
print(bool([]))     # False
print(bool([1, 2])) # True
```

Special Cases and Errors

Type conversion can sometimes fail, especially when the source value cannot be logically converted to the target type. Understanding these limitations helps prevent runtime errors in your code.

Not all type conversions are possible. Python will raise an error when the conversion is not possible.

```
#| autorun: false
# This works
print(int("123"))

# This will cause an error - uncomment to see
# print(int("123.45"))  # ValueError - can't convert string with decimal to int
# print(int("hello"))   # ValueError - can't convert arbitrary string to int
```

To handle potential errors in type conversion, you can use exception handling with try/except blocks:

```
#| autorun: false
try:
    user_input = "abc"
    number = int(user_input)
    print(f"Converted to number: {number}")
except ValueError:
    print(f"Cannot convert '{user_input}' to an integer")
```

Chapter 8

Python Overview

8.1 Control Structures and Functions

Building on our understanding of Python's basic data types and operations, we'll now explore how to control program flow and create reusable code blocks. These structures allow us to write more sophisticated programs that can make decisions, repeat operations, and organize code efficiently.

8.1.1 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.

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
```

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
```

8.1.2 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.

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()
```

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()
```

8.1.3 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`.

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()
```

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()
```

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 current_hour<20:
    print("goodbye")
  else:
    print("good night")

print_hello_or_goodbye()
```

8.2 Exercises

The following exercises will help you practice using functions with conditional logic.

Exercise 1: Temperature Conversion Function

Create a function that converts temperatures between Fahrenheit and Celsius scales. This exercise demonstrates how to define and use functions with conditional logic to perform different types of conversions based on user input.
The conversion formulas are: - Celsius to Fahrenheit: $F = (C \times 9/5) + 32$ - Fahrenheit to Celsius: $C = (F - 32) \times 5/9$
Time estimate: 15-20 minutes

```
#| exercise: temp_convert

def convert_temperature(temp, scale):
    """
    Convert temperature between Celsius and Fahrenheit

    Parameters:
    temp (float): The temperature value to convert
    scale (str): The current scale of the temperature ('C' or 'F')

    Returns:
    tuple: (converted_temp, new_scale)
    """
    # Write your function implementation here
    # If scale is 'C', convert to Fahrenheit
    # If scale is 'F', convert to Celsius
    # Return both the converted temperature and the new scale

    ----

# Test your function with these values
test_cases = [(100, 'C'), (32, 'F'), (0, 'C'), (98.6, 'F')]

for temp, scale in test_cases:
    converted, new_scale = convert_temperature(temp, scale)
    print(f"{temp}°{scale} = {converted:.1f}°{new_scale}")
```

Tip

Use an if-else statement to check the scale parameter. Depending on whether it's 'C' or 'F', apply the appropriate conversion formula. Remember to return both the converted temperature value and the new scale designation (either 'F' or 'C').

Solution.

Note

```python
def convert_temperature(temp, scale):
    """
    Convert temperature between Celsius and Fahrenheit

    Parameters:
    temp (float): The temperature value to convert
    scale (str): The current scale of the temperature ('C' or 'F')

    Returns:
    tuple: (converted_temp, new_scale)
    """
    if scale == 'C':
        # Convert from Celsius to Fahrenheit
        converted_temp = (temp * 9/5) + 32
        new_scale = 'F'
    elif scale == 'F':
        # Convert from Fahrenheit to Celsius
        converted_temp = (temp - 32) * 5/9
        new_scale = 'C'
    else:
        # Handle invalid scale input
        return "Invalid scale. Use 'C' or 'F'.", None

    return converted_temp, new_scale

# Test your function with these values
test_cases = [(100, 'C'), (32, 'F'), (0, 'C'), (98.6, 'F')]

for temp, scale in test_cases:
    converted, new_scale = convert_temperature(temp, scale)
    print(f"{temp}°{scale} = {converted:.1f}°{new_scale}")
```

Exercise 2: Prime Number Checker

Create a function that checks whether a given number is prime. This exercise demonstrates the use of loops, conditional statements, and early return to solve a common mathematical problem.
A prime number is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers.
Time estimate: 15-20 minutes

```python
#| exercise: prime_check

def is_prime(number):
    """
    Check if a number is prime

    Parameters:
    number (int): The number to check

    Returns:
    bool: True if the number is prime, False otherwise
    """
    # Write your function implementation here
    # Remember:
    # - Numbers less than 2 are not prime
    # - A number is prime if it's only divisible by 1 and itself


    ----

# Test the function with various numbers
for num in [2, 7, 10, 13, 15, 23, 24, 29]:
    result = is_prime(num)
    print(f"{num} is {'prime' if result else 'not prime'}")
```

Tip

First, check if the number is less than 2 (not prime). Then, use a loop to check if the number is divisible by any integer from 2 to the square root of the number. If you find a divisor, the number is not prime. If no divisors are found, the number is prime.

Solution.

Note

```
def is_prime(number):
    """
    Check if a number is prime

    Parameters:
    number (int): The number to check

    Returns:
    bool: True if the number is prime, False otherwise
    """
    # Numbers less than 2 are not prime
    if number < 2:
        return False

    # Check for divisibility from 2 to the square root of number
    # We only need to check up to the square root because
    # if number = a*b, either a or b must be   sqrt(number)
    import math
    for i in range(2, int(math.sqrt(number)) + 1):
        if number % i == 0:
            return False

    # If we didn't find any divisors, the number is prime
    return True

# Test the function with various numbers
for num in [2, 7, 10, 13, 15, 23, 24, 29]:
    result = is_prime(num)
    print(f"{num} is {'prime' if result else 'not prime'}")
```

Part III

Lecture 3

Chapter 9

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 program 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

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

print(x)
```

This includes the whole module and makes it available for use later in the program. Note that the functions of the module are accessed using the prefix `math.`, which is the namespace for the module.

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.

9.0.1 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
```

9.0.2 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 from 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 the python website .

9.0.3 Advanced topics

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:

```python
# 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:

```python
# main.py

from mypackage.subpackage import submodule

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

Chapter 10

NumPy Module

Numpy is, besides SciPy, 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.

For physics applications, NumPy is essential because it enables efficient numerical calculations on large datasets, handling of vectors and matrices, and implementation of mathematical models that describe physical phenomena. Whether simulating particle motion, analyzing experimental data, or solving equations of motion, NumPy provides the computational foundation needed for modern physics.

```
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 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

10.1.1 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]])
```

10.1.2 Using array-generating functions

For larger arrays it is impractical 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 equivalent things with logarithmic 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))
```

10.2 Array Attributes

NumPy arrays have several attributes that provide information about their size, shape, and data type. These attributes are essential for understanding and debugging your code.

10.2.1 shape

The `shape` attribute returns a tuple that gives the size of the array along each dimension.

```
#| autorun: false
a = np.array([[1, 2, 3], [4, 5, 6]])
a.shape
```

10.2.2 size

The `size` attribute returns the total number of elements in the array.

```
#| autorun: false
a.size
```

10.2.3 dtype

The `dtype` attribute returns the data type of the array's elements.

```
#| autorun: false
a.dtype
```

```
#| autorun: false
b = np.array([1.0, 2.0, 3.0])
b.dtype
```

These attributes are particularly useful when debugging operations between arrays, as many NumPy functions require arrays of specific shapes or compatible data types.

10.3 Manipulating NumPy arrays

10.3.1 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]
```

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

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

```
#| 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
```

10.3.2 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))
```

10.3.3 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
```

10.3.4 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))
```

10.4 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.

10.4.1 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
```

10.4.2 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])
```

10.4.3 Random Numbers

NumPy provides powerful tools for generating random numbers, which are essential for simulations in statistical physics, quantum mechanics, and other fields:

```
#| autorun: false
# Uniform random numbers between 0 and 1
uniform_samples = np.random.random(5)
```

```
print("Uniform samples:", uniform_samples)

# Normal distribution (Gaussian) with mean 0 and standard deviation 1
gaussian_samples = np.random.normal(0, 1, 5)
print("Gaussian samples:", gaussian_samples)

# Random integers
random_integers = np.random.randint(1, 10, 5)  # Values between 1-9
print("Random integers:", random_integers)
```

These random number generators are particularly useful for Monte Carlo simulations, modeling thermal noise, or simulating quantum mechanical systems.

10.5 Broadcasting

Broadcasting is a powerful mechanism that allows NumPy to work with arrays of different shapes when performing arithmetic operations. The smaller array is "broadcast" across the larger array so that they have compatible shapes.

The rules for broadcasting are:

1. If the arrays don't have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The size in each dimension of the output shape is the maximum of the sizes of the input arrays along that dimension.
3. An input can be used in the calculation if its size in a particular dimension matches the output size or if its value is exactly 1.
4. If an input has a dimension size of 1, the first element is used for all calculations along that dimension.

Let's see some examples:

```
#| autorun: false
# Broadcasting a scalar to an array
a = np.array([1, 2, 3])
a * 2  # 2 is broadcast to [2, 2, 2]
```

```
#| autorun: false
# Broadcasting arrays of different shapes
a = np.array([[1, 2, 3], [4, 5, 6]])  # Shape: (2, 3)
b = np.array([10, 20, 30])  # Shape: (3,)
a + b  # b is broadcast to shape (2, 3)
```

```
#| autorun: false
# A more complex example
a = np.ones((3, 4))
b = np.arange(4)
a + b  # b is broadcast across each row of a
```

Broadcasting enables efficient computation without the need to create copies of arrays, saving memory and computation time.

10.6 Physics Example: Force Calculations

Broadcasting is particularly useful in physics when applying the same operation to multiple objects. For example, when calculating the gravitational force between one massive object and multiple other objects using Newton's law of universal gravitation:

$$F = \frac{GMm}{r^2} \tag{10.2}$$

where F is the gravitational force, G is the gravitational constant, M and m are the masses of the two objects, and r is the distance between them.

```
#| autorun: false
# Gravitational constant
G = 6.67430e-11

# Mass of central object (e.g., Sun) in kg
M = 1.989e30

# Masses of planets in kg (simplified)
planet_masses = np.array([3.3e23, 4.87e24, 5.97e24, 6.42e23])  # Mercury, Venus, Earth, Mars

# Distances from Sun in meters (simplified)
distances = np.array([5.79e10, 1.08e11, 1.5e11, 2.28e11])

# Calculate gravitational forces
# F = G*M*m/r²
forces = G * M * planet_masses / distances**2

print(forces)  # Force in Newtons
```

Chapter 11

Basic Plotting with Matplotlib

Data visualization is an essential skill for analyzing and presenting scientific data effectively. Python itself doesn't include plotting capabilities in its core language, but Matplotlib provides powerful and flexible tools for creating visualizations. Matplotlib is the most widely used plotting library in Python and serves as an excellent starting point for creating basic plots.

Matplotlib works well with NumPy, Python's numerical computing library, to create a variety of plot types including line plots, scatter plots, bar charts, and more. For this document, we've already imported both libraries as you can see in the code below:

```
#| autorun: true

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

We've also set up some default styling parameters to make our plots more readable and professional-looking:

```
#| 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',
                     'figure.figsize': (4, 3),
                     'figure.dpi': 150})

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

These settings configure the appearance of our plots with appropriate font sizes, line widths, and tick marks. The `get_size()` function helps us convert dimensions from centimeters to inches, which is useful when specifying figure sizes. With these preparations complete, we're ready to create various types of visualizations to effectively display our data.

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. Later in this course, we'll briefly touch upon more advanced techniques that provide greater control over plot elements and layout.

11.1 Basic Plotting

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 $[0, 4\pi]$. 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=get_size(8,6))                                        ③
plt.plot(x, y)                                                           ④
plt.tight_layout()                                                       ⑤
plt.show()                                                               ⑥
```

① Create an array of 100 values between 0 and 4π.
② Calculate the sine of each value in the array.
③ create a new figure with a size of (8,6) cm
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=get_size(8,6))
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()
```

11.2 Customizing Plots

11.2.1 Axis Labels

To enhance the clarity and interpretability of our plots, it's crucial to provide context through proper labeling. The following commands add descriptive axis labels to our diagram:

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

Here's an example of adding labels to our sine plot:

```
#| autorun: false

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

plt.figure(figsize=get_size(8,6))
plt.plot(x, y)
plt.xlabel('t')                # set the x-axis label
plt.ylabel('sin(t)')           # set the y-axis label
plt.tight_layout()
plt.show()
```

11.2.2 Legends

When plotting multiple datasets, it's important to include a legend to identify each line. Use these commands:

```
plt.plot(..., label='Label name')
plt.legend(loc='lower left')
```

Here's an example with a legend:

```
#| autorun: false

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

plt.figure(figsize=get_size(8,6))
plt.plot(x, np.sin(x), "ko", markersize=5, label=r"$\delta(t)$")   # define a label
plt.xlabel('t')
plt.ylabel(r'$\sin(t)$')
plt.legend(loc='lower left')                                        # add the legend
plt.tight_layout()
plt.show()
```

11.2.3 Plotting Multiple Lines

You can add multiple lines to the same plot:

```
#| autorun: false
x = np.linspace(0, 2*np.pi, 100)

plt.figure(figsize=get_size(8, 8))
plt.plot(x, np.sin(x), label='sin(x)')         # Add a label for the legend
plt.plot(x, np.cos(x), label='cos(x)')         # Second line
plt.plot(x, np.sin(2*x), label='sin(2x)')      # Third line
plt.legend()                                    # Display the legend
plt.title('Trigonometric Functions')
plt.xlabel('x')
plt.ylabel('y')
plt.tight_layout()
plt.show()
```

11.2.4 Customizing Line Appearance

You can customize the appearance of lines with additional parameters:

```
#| autorun: false
x = np.linspace(0, 2*np.pi, 100)

plt.figure(figsize=get_size(14, 8))
# Format string: color, marker, line style
plt.plot(x, np.sin(x), 'r-', label='sin(x)')      # Red solid line
plt.plot(x, np.cos(x), 'b--', label='cos(x)')     # Blue dashed line
plt.plot(x, np.sin(2*x), 'g.', label='sin(2x)')   # Green dots
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')  # Place legend outside plot
plt.xlabel('x')
plt.ylabel('y')
plt.subplots_adjust(right=0.8)  # Add space for the legend
plt.tight_layout()

plt.show()
```

11.2.5 Plots with Error Bars

When plotting experimental data, it's customary to include error bars that graphically indicate measurement uncertainty. The `errorbar` function can be used to display both vertical and horizontal error bars:

```
plt.errorbar(x, y, xerr=x_errors, yerr=y_errors, fmt='format', label='label')
```

Here's an example of a plot with error bars:

```
#| autorun: false

xdata = np.arange(0.5, 3.5, 0.5)
ydata = 210-40/xdata
yerror = 2e3/ydata

plt.figure(figsize=get_size(8,6))
plt.errorbar(xdata, ydata, fmt="ro", label="data",
             xerr=0.15, yerr=yerror, ecolor="black")
plt.xlabel("x")
plt.ylabel("t-displacement")
plt.legend(loc="lower right")
plt.tight_layout()
plt.show()
```

11.2.6 Visualizing NumPy Arrays

We can visualize 2D arrays created with NumPy:

```
#| autorun: false
# Create a 2D array using mgrid
x, y = np.mgrid[0:5:0.1, 0:5:0.1]
z = np.sin(x) * np.cos(y)

plt.figure(figsize=get_size(12, 8))
plt.pcolormesh(x, y, z, cmap='viridis',edgecolors="none")  # Color mesh plot
plt.colorbar(label='sin(x)cos(y)')      # Add color scale
plt.xlabel('x')
plt.ylabel('y')
plt.tight_layout()
plt.show()
```

11.3 Saving Figures

To save a figure to a file, use the `savefig` method. Matplotlib supports multiple formats including PNG, JPG, EPS, SVG, PGF and PDF:

```
plt.savefig('filename.extension')
```

Here's an example of creating and saving a figure:

```
#| autorun: false

theta = np.linspace(0.01, 10., 100)
ytan = np.sin(2*theta) + np.cos(3.1*theta)

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

For scientific papers, PDF format is recommended whenever possible. LaTeX documents compiled with `pdflatex` can include PDFs using the `includegraphics` command. PGF can also be a good alternative in some cases.

11.4 NumPy with Visualization

The arrays and calculations we've learned in NumPy form the foundation for scientific data visualization. In the next section, we'll explore how to use Matplotlib to create visual representations of NumPy arrays, allowing us to interpret and communicate our physics results more effectively.

For example, we can visualize the planetary force calculations from our broadcasting example:

```
#| autorun: false

# Planet names for our example
planets = ['Mercury', 'Venus', 'Earth', 'Mars']

# Calculate gravitational forces (code from previous example)
G = 6.67430e-11
M = 1.989e30
planet_masses = np.array([3.3e23, 4.87e24, 5.97e24, 6.42e23])
distances = np.array([5.79e10, 1.08e11, 1.5e11, 2.28e11])
forces = G * M * planet_masses / distances**2

# Plotting
plt.figure(figsize=get_size(8, 8))
plt.bar(planets, forces)
plt.ylabel('Gravitational Force (N)')
#plt.title('Gravitational Force from Sun to Planets')
plt.tight_layout()
plt.show()
```

Part IV

Lecture 4

Chapter 12

Classes and Objects

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

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, Rectangle
import matplotlib.patches as mpatches

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',})
```

12.1 Introduction to Object Oriented Programming

Imagine you're simulating a complex physical system—perhaps a collection of interacting particles or cells. Each entity in your simulation has both properties (position, velocity, size) and behaviors (move, interact, divide). How do you organize this complexity in your code?

12.1.1 From Procedural to Object-Oriented Thinking

In previous lectures, we've designed programs using a **procedural approach**—organizing code around functions that operate on separate data structures. While this works for simpler problems, it can become unwieldy as systems grow more complex.

Object-oriented programming (OOP) offers a more intuitive paradigm: it combines data and functionality together into self-contained units called **objects**. Instead of having separate variables and functions, each object maintains its own state and defines its own behaviors.

For computational modeling, this is particularly powerful because:

- Objects can directly represent the entities you're modeling (particles, cells, molecules)
- Code organization mirrors the structure of the real-world system

- Complex systems become easier to build incrementally and modify later

12.2 The Building Blocks: Classes and Objects

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

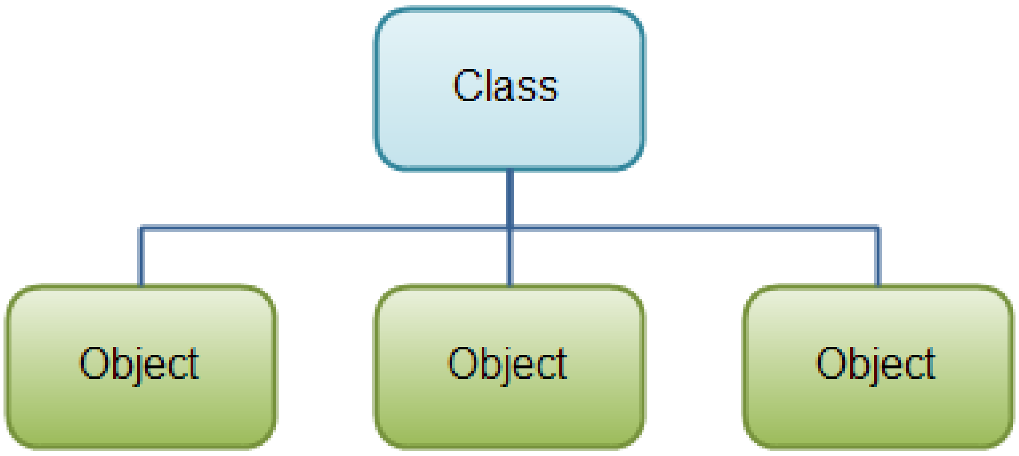


Figure 12.1: Sketch of the relation of classes and objects

12.2.1 Classes: Creating Blueprints

A **class** serves as 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. It specifies:

- What data the object will store (properties)
- What operations the object can perform (methods)

12.2.2 Objects: Creating Instances

An **object** is a specific instance of a class—a concrete realization of that blueprint. When you create an object, you're essentially saying "make me a new thing based on this class design."

Objects 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 and how it manipulates its data

12.2.3 Properties: Storing Data

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 `Colloid` class for a particle simulation:

- Instance variables might include `radius` and `position` (unique to each particle)
- Class variables might include `material_density` (same for all colloids of that type)
- Methods might include `move()` or `calculate_volume()`

12.3 Working with Classes in Python

12.3.1 Creating a Class

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.

Let's start with a minimal example that represents a colloidal particle:

```
#| autorun: false
# Define a minimal empty class for a colloidal particle
class Colloid:
    pass  # 'pass' creates an empty class with no properties or methods

# Create an instance of the Colloid class
particle = Colloid()

# Display the particle object (shows its memory location)
print(particle)
```

Even this empty class is a valid class definition, though it doesn't do anything useful yet. Let's start adding functionality to make it more practical.

12.3.2 Creating Methods

Methods are functions that belong to a class. They define the behaviors and capabilities of your objects.

```
#| autorun: false
# Define a Colloid class with a method
class Colloid:
    # Define a method that identifies the type of colloid
    def type(self):
        print('I am a plastic colloid')

# Create two separate colloid objects
p = Colloid()  # First colloid instance
b = Colloid()  # Second colloid instance

# Call the type method on each object
print("Particle p says:")
p.type()

print("\nParticle b says:")
b.type()
```

Understanding `self` in Python Classes

Every method in a Python class automatically receives a special first parameter, conventionally named `self`. This parameter represents 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 and other methods - 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, Python automatically passes `particle` as the `self` parameter.

12.3.3 The Constructor Method: `__init__`

The `__init__` method is a special method called when a new object is created. It lets you initialize the object's properties with specific values.

```
#| autorun: false
# Define a Colloid class with constructor and a method
class Colloid:
    # Constructor: Initialize a new colloid with a specific radius
    def __init__(self, R):
        # Store the radius as an instance variable (unique to each colloid)
        self.R = R

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

# Create two colloids with different radii
particle1 = Colloid(5)  # Creates a colloid with radius 5 µm
particle2 = Colloid(2)  # Creates a colloid with radius 2 µm

# Get and display the size of each particle
print(f'Particle 1 radius: {particle1.get_size()} µm')
print(f'Particle 2 radius: {particle2.get_size()} µm')

# We can also directly access the R property
print(f'Accessing radius directly: {particle1.R} µm')
```

Note

Python also provides a `__del__` method (destructor) that's called when an object is deleted. This can be useful for cleanup operations or tracking object lifecycles.

12.3.4 String Representation: The `__str__` Method

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

This method helps make your objects more readable and informative:

```
#| autorun: false
# Define a Colloid class with string representation
class Colloid:
    def __init__(self, R):
        self.R = R  # Initialize radius
    def get_size(self):
        return self.R

    # Define how the object should be displayed as text
    def __str__(self):
        return f'Colloid particle with radius {self.R:.1f} µm'

# Create colloids with different radii
particle1 = Colloid(15)
particle2 = Colloid(3.567)

# Print the objects - this automatically calls __str__
print("Particle 1:", particle1)
print("Particle 2:", particle2)
```

Tip

The `.1f` format specification means the radius will be displayed with one decimal place. This helps make your output more readable. You can customize this string representation to show whatever information about your object is most relevant.

12.4 Managing Data in Classes

12.4.1 Class Variables vs. Instance Variables

One of the core features of OOP is how it manages data. Python classes offer two distinct types of variables:

Class Variables: Shared Among All Objects

- **Definition**: Variables defined directly inside the class but outside any method
- **Behavior**: All instances of the class share the **same copy** of these variables
- **Usage**: For properties that should be the same across all instances
- **Access pattern**: Typically accessed as `ClassName.variable_name`

Instance Variables: Unique to Each Object

- **Definition**: Variables defined within methods, typically in `__init__`
- **Behavior**: Each object has its **own separate copy** of these variables
- **Usage**: For properties that can vary between different instances
- **Access pattern**: Typically accessed as `self.variable_name` within methods

Here's a practical example showing both types of variables in action:

```
#| autorun: false
class Colloid:
    # Class variable: shared by all colloids
    material = "polystyrene"  # All colloids made of the same material
    total_particles = 0       # Counter to track total number of particles

    def __init__(self, R, position=(0,0)):
        # Instance variables: each particle has its own
        self.R = R                 # Radius - unique to each particle
        self.position = position   # Position - unique to each particle

        # Update the class variable when a new particle is created
        Colloid.total_particles += 1

    def __del__(self):
        # Update the class variable when a particle is deleted
        Colloid.total_particles -= 1
        print(f"Particle deleted, {Colloid.total_particles} remaining")

    def __str__(self):
        return f"Colloid(R={self.R}, pos={self.position})"

    def change_material(new_material):
        # This changes the material for ALL colloids
        Colloid.material = new_material

# Create some particles with different radii and positions
print("Creating particles...")
p1 = Colloid(3, (0, 0))      # Radius 3, at origin
p2 = Colloid(5, (10, 5))     # Radius 5, at position (10,5)
p3 = Colloid(7, (-5, -5))    # Radius 7, at position (-5,-5)

# Each particle has its own radius and position (instance variables)
print(f"\nInstance variables (unique to each object):")
print(f"p1: radius={p1.R}, position={p1.position}")
print(f"p2: radius={p2.R}, position={p2.position}")
print(f"p3: radius={p3.R}, position={p3.position}")

# All particles share the same material and total_particles count (class variables)
print(f"\nClass variables (shared by all objects):")
print(f"Material for all particles: {Colloid.material}")
print(f"Total particles: {Colloid.total_particles}")

# Change the material - affects all particles
Colloid.material = "silica"
print(f"\nAfter changing material to {Colloid.material}:")
print(f"p1 material: {Colloid.material}")
print(f"p2 material: {Colloid.material}")

# Delete one particle to see the counter decrease
print("\nDeleting p3...")
del p3
print(f"Total particles remaining: {Colloid.total_particles}")
```

12.4.2 When to Use Each Type of Variable

Use Class Variables When:

- A property should be the same for all instances (like physical constants)
- You need to track information about the class as a whole (like counters)
- You want to save memory by not duplicating unchanging values

Use Instance Variables When:

- Objects need their own independent state
- Properties vary between instances (position, size, etc.)
- You're representing unique characteristics of individual objects

Warning

Be careful when modifying class variables! Since they're shared, changes will affect all instances of the class. This can lead to unexpected behavior if not managed carefully.

Chapter 13

Brownian Motion

13.1 Introduction

Brownian motion is a fundamental physical phenomenon that describes the random movement of particles suspended in a fluid. This lecture explores both the physical understanding and computational modeling of Brownian motion using object-oriented programming techniques.

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',})
```

13.2 Brownian Motion

13.2.1 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!

13.2.2 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.

```
//| echo: false
//| label: fig-BM
//| fig-cap: Interactive simulation of Brownian motion. The blue circle represents a larger colloid, wh

brownianMotion = {
  const width = 400;
  const height = 400;

  // Create SVG container
  const svg = d3.create("svg")
    .attr("width", width)
    .attr("height", height)
    .attr("viewBox", [0, 0, width, height])
    .attr("style", "max-width: 100%; height: auto;");

  // Add a border for clarity
  svg.append("rect")
    .attr("width", width)
    .attr("height", height)
    .attr("fill", "none")
    .attr("stroke", "black");

  // Parameters for our simulation - adjusted for better physical representation
  const numSmallParticles = 400;
  const largeParticleRadius = 10;
  const smallParticleRadius = 5;  // Increased scale separation
  const largeParticleColor = "blue";
  const smallParticleColor = "red";
  const trailLength = 300;

  // Physical parameters
  const temperature = 10.0;  // Normalized temperature
  const gamma = 0.1;         // Drag coefficient for large particle

  // Calculate masses based on radius^3 (proportional to volume)
  const largeMass = Math.pow(largeParticleRadius, 3);
  const smallMass = Math.pow(smallParticleRadius, 3);

  // Maxwell-Boltzmann distribution helper
  function maxwellBoltzmannVelocity() {
    // Box-Muller transform for normal distribution
    const u1 = Math.random();
    const u2 = Math.random();
    const mag = Math.sqrt(-2.0 * Math.log(u1)) * Math.sqrt(temperature / smallMass);
    const theta = 2 * Math.PI * u2;
    return {
      vx: mag * Math.cos(theta),
      vy: mag * Math.sin(theta)
    };
  }

  // Initialize large particle in the center
  let largeParticle = {
    x: width / 2,
    y: height / 2,
    vx: 0,
    vy: 0,
    radius: largeParticleRadius,
    mass: largeMass,
    // Store previous positions for trail
    trail: Array(trailLength).fill().map(() => ({
      x: width / 2,
      y: height / 2
    }))
  };

  // Initialize small particles with random positions and thermal velocities
  const smallParticles = Array(numSmallParticles).fill().map(() => {
    const vel = maxwellBoltzmannVelocity();
    return {
      x: Math.random() * width,
      y: Math.random() * height,
      vx: vel.vx * 8,  // Scale for visibility
      vy: vel.vy * 8,  // Scale for visibility
      radius: smallParticleRadius,
      mass: smallMass
    };
  });

  // Create the large particle
  const largeParticleElement = svg.append("circle")
    .attr("cx", largeParticle.x)
    .attr("cy", largeParticle.y)
    .attr("r", largeParticle.radius)
    .attr("fill", largeParticleColor);

  // Create the trail for the large particle
  const trailElements = svg.append("g")
    .selectAll("circle")
    .data(largeParticle.trail)
    .join("circle")
    .attr("cx", d => d.x)
    .attr("cy", d => d.y)
    .attr("r", (_, i) => 1)
    .attr("fill", "rgba(0, 0, 255, 0.2)");

  // Create the small particles
  const smallParticleElements = svg.append("g")
    .selectAll("circle")
    .data(smallParticles)
    .join("circle")
    .attr("cx", d => d.x)
    .attr("cy", d => d.y)
    .attr("r", d => d.radius)
```

```
    .attr("fill", smallParticleColor);

  // Function to update particle positions
  function updateParticles() {
    // Apply drag to large particle (Stokes' law)
    largeParticle.vx *= (1 - gamma);
    largeParticle.vy *= (1 - gamma);

    // Update small particles
    smallParticles.forEach((particle, i) => {
      // Move according to velocity
      particle.x += particle.vx;
      particle.y += particle.vy;

      // Bounce off walls
      if (particle.x < particle.radius || particle.x > width - particle.radius) {
        particle.vx *= -1;
        particle.x = Math.max(particle.radius, Math.min(width - particle.radius, particle.x));
      }
      if (particle.y < particle.radius || particle.y > height - particle.radius) {
        particle.vy *= -1;
        particle.y = Math.max(particle.radius, Math.min(height - particle.radius, particle.y));
      }

      // Check for collision with large particle
      const dx = largeParticle.x - particle.x;
      const dy = largeParticle.y - particle.y;
      const distance = Math.sqrt(dx * dx + dy * dy);

      if (distance < largeParticle.radius + particle.radius) {
        // Physically correct elastic collision

        // Calculate unit normal vector (collision axis)
        const nx = dx / distance;
        const ny = dy / distance;

        // Calculate unit tangent vector (perpendicular to collision)
        const tx = -ny;
        const ty = nx;

        // Project velocities onto normal and tangential axes
        const v1n = largeParticle.vx * nx + largeParticle.vy * ny;
        const v1t = largeParticle.vx * tx + largeParticle.vy * ty;
        const v2n = particle.vx * nx + particle.vy * ny;
        const v2t = particle.vx * tx + particle.vy * ty;

        // Calculate new normal velocities using conservation of momentum and energy
        // Tangential velocities remain unchanged in elastic collision
        const m1 = largeParticle.mass;
        const m2 = particle.mass;

        // One-dimensional elastic collision formula
        const v1nAfter = (v1n * (m1 - m2) + 2 * m2 * v2n) / (m1 + m2);
        const v2nAfter = (v2n * (m2 - m1) + 2 * m1 * v1n) / (m1 + m2);
        // Convert back to x,y velocities
        largeParticle.vx = v1nAfter * nx + v1t * tx;
        largeParticle.vy = v1nAfter * ny + v1t * ty;
        particle.vx = v2nAfter * nx + v2t * tx;
        particle.vy = v2nAfter * ny + v2t * ty;

        // Move particles apart to prevent overlap
        const overlap = largeParticle.radius + particle.radius - distance;
        const massRatio = m2 / (m1 + m2);
        const largeMoveRatio = massRatio;
        const smallMoveRatio = 1 - massRatio;

        // Move particles apart proportional to their masses
        largeParticle.x += overlap * nx * largeMoveRatio;
        largeParticle.y += overlap * ny * largeMoveRatio;
        particle.x -= overlap * nx * smallMoveRatio;
        particle.y -= overlap * ny * smallMoveRatio;
      }

      // Occasionally thermostat small particles to maintain temperature
      if (Math.random() < 0.01) {
        const vel = maxwellBoltzmannVelocity();
        particle.vx = vel.vx * 8;  // Scale for visibility
        particle.vy = vel.vy * 8;  // Scale for visibility
      }

      // Update small particle display
      smallParticleElements.filter((_, j) => i === j)
        .attr("cx", particle.x)
        .attr("cy", particle.y);
    });

    // Update large particle position
    largeParticle.x += largeParticle.vx;
    largeParticle.y += largeParticle.vy;

    // Bounce large particle off walls
    if (largeParticle.x < largeParticle.radius || largeParticle.x > width - largeParticle.radius) {
      largeParticle.vx *= -1;
      largeParticle.x = Math.max(largeParticle.radius, Math.min(width - largeParticle.radius, largePart
    }
    if (largeParticle.y < largeParticle.radius || largeParticle.y > height - largeParticle.radius) {
      largeParticle.vy *= -1;
      largeParticle.y = Math.max(largeParticle.radius, Math.min(height - largeParticle.radius, largePar
    }

    // Update trail
    largeParticle.trail.pop();
    largeParticle.trail.unshift({x: largeParticle.x, y: largeParticle.y});

    // Update large particle display
    largeParticleElement
      .attr("cx", largeParticle.x)
      .attr("cy", largeParticle.y);
    // Update trail display
    trailElements.data(largeParticle.trail)
      .attr("cx", d => d.x)
      .attr("cy", d => d.y);
  }

  // Start animation
  const interval = d3.interval(() => {
    updateParticles();
  }, 30);

  // Clean up on invalidation
  invalidation.then(() => interval.stop());

  return svg.node();
}
```

13.2.3 The Mathematical Model of Brownian Motion

Mathematically, Brownian motion is governed by the Langevin equation, which describes the basic equation of motion:

$$m\frac{d^2\mathbf{r}}{dt^2} = -\gamma\frac{d\mathbf{r}}{dt} + \mathbf{F}_{\text{random}}(t)$$

where:

- m is the particle mass
- $\mathbf{r}$ is the position vector
- γ is the drag coefficient
- $\mathbf{F}_{\text{random}}(t)$ represents random forces from molecular collisions

In the overdamped limit ($m = 0$ which applies to colloidal particles), inertia becomes negligible and the equation simplifies to:

$$\frac{d\mathbf{r}}{dt} = \sqrt{2D}\,\xi(t)$$

Where $\xi(t)$ is Gaussian white noise with a unit variance and D is the diffusion coefficient, the transport coefficient characterizing diffusive transport.

A key observable in Brownian motion is the mean squared displacement (MSD):

$$\langle(\Delta r)^2\rangle = 2dDt$$

with:

- $\langle(\Delta r)^2\rangle$ is the mean squared displacement
- d is the number of dimensions (2 in our simulation)
- D is the diffusion coefficient
- t is the time elapsed

The diffusion coefficient D depends on physical properties according to the Einstein-Stokes relation:

$$D = \frac{k_B T}{6\pi\eta R}$$

Where k_B is Boltzmann's constant, T is temperature, η is fluid viscosity, and R is the particle radius.

13.2.4 Numerical Implementation

In our `Colloid` class simulation, we implement the discretized version of the overdamped Langevin equation. For each time step Δt, the position update is:

$$\Delta x = \sqrt{2D\Delta t} \times \xi$$

Where Δx is the displacement in one direction, and ξ is a random number drawn from a normal distribution with mean 0 and variance 1.

This is implemented directly in the `update()` method of our `Colloid` class:

```
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])
```

In this implementation: - `D` is the diffusion coefficient stored as an instance variable - `dt` is the time step parameter - `np.random.normal` generates the Gaussian random numbers required for the stochastic process

Tip

The choice of time step `dt` is important in our simulation. If too large, it fails to capture the fine details of the motion. If too small, the simulation becomes computationally expensive. The class design allows us to adjust this parameter easily when calling `sim_trajectory()` or `update()`.

```
#| 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:

$$p(x, \Delta t) = \frac{1}{\sqrt{4\pi D\Delta t}} e^{-\frac{x^2}{4D\Delta t}}$$

with:

- 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.

13.3 Object-Oriented Implementation

13.3.1 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.

13.3.2 Class Design

We design a `Colloid` class to simulate particles undergoing Brownian motion. Using object-oriented programming makes physical sense here - in the real world, each colloidal particle is an independent object with its own properties that follows the same physical laws as other particles.

Class-Level Properties (Shared by All Particles)

Our `Colloid` class will store information common to all particles:

1. `number`: A counter tracking how many particles we've created
2. `f = 2.2×10^{-19}`: The physical constant $k_B T/(6\pi\eta)$ in m^3/s
   - This combines Boltzmann's constant (k_B), temperature (T), and fluid viscosity (η)
   - Using this constant simplifies our diffusion calculations

Class Methods (Functions Shared by All Particles)

The class provides these shared behaviors:

1. `how_many()`: Returns the total count of particles created
   - Useful for tracking how many particles exist in our simulation
2. `__str__()`: Returns a human-readable description when we print a particle
   - Shows the particle's radius and current position

Instance Properties (Unique to Each Particle)

Each individual particle will have its own:

1. `R`: Radius in meters
2. `x`, `y`: Lists storing position history (starting with initial position)
3. `index`: Unique ID number for each particle
4. `D`: Diffusion coefficient calculated as $D = f/R$
   - From Einstein-Stokes relation: $D = \frac{k_B T}{6\pi\eta R}$
   - Smaller particles diffuse faster (larger D)

Instance Methods (What Each Particle Can Do)

Each particle object will have these behaviors:

1. `update(dt)`: Performs a single timestep of Brownian motion
   - Takes a timestep `dt` in seconds
   - Adds random displacement based on diffusion coefficient
   - Returns the new position
2. `sim_trajectory(N, dt)`: Simulates a complete trajectory
   - Generates N steps with timestep dt
   - Calls `update()` repeatedly to build the trajectory
3. `get_trajectory()`: Returns the particle's movement history as a DataFrame
   - Convenient for analysis and plotting
4. `get_D()`: Returns the particle's diffusion coefficient
   - Useful for calculations and verification

```
#| 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.

13.4 Simulation and Analysis

13.4.1 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])
```

13.4.2 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()
```

13.4.3 Characterizing the Brownian motion

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

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

$$v(ndt) = \frac{< \sqrt{(x_{i+n} - x_i)^2 + (y_{i+n} - y_i)^2} >}{n\,dt} \tag{13.1}$$

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.

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

$$\langle \Delta r^2(t) \rangle = 2dDt \tag{13.2}$$

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.loglog(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.

13.5 Summary

In this lecture, we have:

1. Explored the physical principles behind Brownian motion and its mathematical description
2. Implemented a computational model using object-oriented programming principles
3. Created a `Colloid` class with properties and methods that simulate realistic particle behavior
4. Generated and visualized multiple particle trajectories
5. Analyzed the simulation results using mean squared displacement calculations
6. Compared our numerical results with theoretical predictions

This exercise demonstrates how object-oriented programming provides an elegant framework for physics simulations, where the objects in our code naturally represent physical entities in the real world.

13.6 Further Reading

- Einstein, A. (1905). "On the Movement of Small Particles Suspended in Stationary Liquids Required by the Molecular-Kinetic Theory of Heat"
- Berg, H.C. (1993). "Random Walks in Biology"
- Chandrasekhar, S. (1943). "Stochastic Problems in Physics and Astronomy"
- Nelson, E. (2001). "Dynamical Theories of Brownian Motion"

Part V

Lecture 5

Chapter 14

Dealing with text files containing data

In physics laboratory experiments, you'll frequently encounter the need to handle data stored in text files. Whether you're collecting measurements from a pendulum experiment, analyzing spectrometer readings, or processing particle collision data, efficiently importing, manipulating, and exporting this data is essential for your analysis. As second-semester physics students, mastering these file handling techniques will save you significant time when processing experimental results and allow you to focus on the physical interpretation rather than data management. This section covers the fundamental approaches to working with data files in Python, from basic file operations to specialized tools in NumPy that are particularly useful for the large datasets common in physics applications.

14.0.1 Input using Python's File Handling

To input or output data to a file you can use Python's built-in file handling, e.g. to write data:

```python
import numpy as np

# Create sample data
x = np.array([1, 2, 3, 4, 5])
y = np.array([10, 20, 30, 40, 50])

# Write data to file using Python's file handling
with open('data.txt', 'w') as file:
    for x_val, y_val in zip(x, y):
        file.write(f"{x_val}, {y_val}\n")

print("Data written to file using Python's file handling")
```

This approach gives you more control over formatting and is useful when dealing with complex data structures or when you need custom formatting. Python's built-in file handling allows you to precisely control how each line is formatted, which is particularly valuable when working with heterogeneous data or when you need to create files that conform to specific format requirements.

Note

The Python **with** statement is a context manager that provides a clean and efficient way to handle resources that need setup and teardown operations, such as file handling, database connections, or network connections.
The basic syntax looks like this:

```python
with expression as variable:
    # code block
```

The `with` statement ensures that resources are properly managed by automatically handling the setup before entering the code block and the cleanup after exiting it, even if exceptions occur within the block. Here's a common example with file operations:

```
with open('file.txt', 'r') as file:
    data = file.read()
    # Process data
# File is automatically closed when exiting the with block
```

The key benefits of using the `with` statement include:

1. Automatic resource management - no need to explicitly call methods like `close()`
2. Exception safety - resources are properly cleaned up even if exceptions occur
3. Cleaner, more readable code compared to try-finally blocks

In physics and electrical engineering contexts, you might use the `with` statement when working with measurement equipment, data acquisition, or when processing large datasets that require temporary file handling.

14.0.2 Text Data Input and Output with NumPy

NumPy provides several functions for reading and writing text data, which can be particularly useful for handling numeric data stored in text files.

Loading Text Data with NumPy

Using `np.loadtxt`

The most common method for loading text data is `np.loadtxt`. This function reads data from a text file and creates a NumPy array with the values:

```
import numpy as np

# Load data from a text file
data = np.loadtxt('data.txt', delimiter=',')  # Add delimiter to parse the comma-separated values
print(f"Loaded data shape: {data.shape}")
print(data)  # Display the loaded data to confirm it matches what was written
```

You can customize how `loadtxt` interprets the file using various parameters. For instance, you can specify a delimiter to handle CSV files, skip header rows that contain metadata, and select only specific columns to read:

```
# Load with specific delimiter, skipping rows, and selecting columns
data = np.loadtxt('data.txt',
                  delimiter=',',   # CSV file
                  skiprows=1,      # Skip header row
                  usecols=(0, 1, 2))  # Use only first three columns
```

Using `np.genfromtxt`

For more flexible loading, especially with missing values, NumPy provides the `genfromtxt` function. This function is particularly useful when dealing with real-world data that may have inconsistencies or missing entries:

```
# Handle missing values with genfromtxt
data = np.genfromtxt('data_with_missing.txt',
                     delimiter=',',
                     filling_values=-999,  # Replace missing values
                     skip_header=1)        # Skip header row
```

The `genfromtxt` function allows you to specify how missing values should be handled, making it more robust for imperfect datasets where some entries might be missing or corrupted.

Saving Text Data with NumPy

Using `np.savetxt`

You can save NumPy arrays to text files using the `savetxt` function. This function allows you to convert your array data into a human-readable text format that can be easily shared or used by other programs:

```
# Create some data
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Save to CSV file
np.savetxt('output.csv', x, delimiter=',', fmt='%d')
```

The `savetxt` function offers numerous formatting options to control exactly how your data is written. You can add headers and footers to provide context, specify the numeric format of your data, and control other aspects of the output file:

```
# Save with header and footer
np.savetxt('output_formatted.csv', x,
           delimiter=',',
           header='col1,col2,col3',  # Column header
           footer='End of data',
           fmt='%.2f',               # Format as float with 2 decimal places
           comments='# ')            # Change comment character
```

These formatting options give you considerable control over how your numerical data is presented in the output file, which can be important for compatibility with other software or for human readability.

Example Workflow

Here's a complete example of reading, processing, and writing text data that demonstrates a typical data analysis workflow using NumPy's I/O capabilities:

```
import numpy as np

# Read data
data = np.loadtxt('input.csv', delimiter=',', skiprows=1)

# Process data (calculate statistics)
row_means = np.mean(data, axis=1)
row_maxes = np.max(data, axis=1)
row_mins = np.min(data, axis=1)

# Combine original data with calculated statistics
result = np.column_stack((data, row_means, row_maxes, row_mins))

# Save processed data
header = "val1,val2,val3,mean,max,min"
np.savetxt('processed_data.csv', result,
           delimiter=',',
           header=header,
           fmt='%.3f')

print("Data processing complete!")
```

This workflow demonstrates how NumPy can efficiently handle text-based data input and output for numerical analysis. The example reads data from a CSV file, performs statistical calculations on each row, combines the original data with the calculated statistics, and then saves the processed results to a new CSV file with appropriate headers. This type of pipeline is common in data analysis and scientific computing, where raw data is imported, transformed, and then exported in a more useful format.

Chapter 15

Numerical Differentiation for Physics

Derivatives form the mathematical backbone of physics. Whether we're calculating velocity from position, acceleration from velocity, or electric field from potential, we're computing derivatives. While calculus provides us with analytical tools to compute derivatives, many real-world physics problems involve functions that are either too complex for analytical solutions or are only known at discrete points (experimental data). This is where numerical differentiation becomes essential for physicists.

```
#| 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})

# default values for plotting
plt.rcParams.update({'font.size': 10,
                     'lines.linewidth': 1,
                     'lines.markersize': 5,
                     'axes.labelsize': 10,
                     '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))
```

15.0.1 The Calculus Foundations

Before diving into numerical methods, let's revisit the calculus definition of a derivative. The derivative of a function $f(x)$ at a point x is defined as the limit of the difference quotient as the interval Δx approaches zero:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

This definition captures the instantaneous rate of change of f with respect to x. In physics, derivatives represent essential physical quantities:

- The derivative of position with respect to time is velocity
- The derivative of velocity with respect to time is acceleration
- The derivative of potential energy with respect to position gives force

However, in computational physics, we cannot take the limit to zero as computers work with discrete values. Instead, we approximate the derivative using finite differences. This is also possible for higher order derivatives, which can be approximated using more complex finite difference formulas such as

$$f^{(n)}(x) = \lim_{\Delta x \to 0} \frac{1}{\Delta x^n} \sum_{k=0}^{n} (-1)^{k+n} \binom{n}{k} f(x + k\Delta x)$$

15.0.2 Finite Difference Approximations

Numerical differentiation methods primarily rely on finite difference approximations derived from Taylor series expansions. Let's explore these systematically.

Forward Difference

The simplest approximation comes directly from the definition, where we look at the change in function value as we move forward from the current point:

$$f_i' \approx \frac{f_{i+1} - f_i}{\Delta x}$$

This is called the *forward difference* method. To understand its accuracy, we can analyze the error using Taylor expansion. The resulting local error δ at each calculation is:

$$\delta = f_{i+1} - f_i - \Delta x f'(x_i) = \frac{1}{2}\Delta x^2 f''(x_i) + O(\Delta x^3)$$

We observe that while the local truncation error is proportional to Δx^2, the accumulated global error is proportional to Δx, making this a first-order accurate method. This means that halving the step size will approximately halve the error in our final derivative approximation.

Local vs. Global Error

Local truncation error refers to the error introduced in a single step of the numerical method due to truncating the Taylor series. For the forward difference method, this error is $O(\Delta x^2)$.
Global accumulated error is the total error that accumulates as we apply the method repeatedly across the domain. For the forward difference method, this accumulated error is $O(\Delta x)$. Global error is generally one order less accurate than the local error due to error propagation through multiple steps.

Central Difference

We can derive a more accurate approximation by using function values on both sides of the point of interest. Using Taylor expansions for $f(x + \Delta x)$ and $f(x - \Delta x)$:

$$f_{i+1} = f_i + \Delta x f_i' + \frac{\Delta x^2}{2!} f_i'' + \frac{\Delta x^3}{3!} f_i^{(3)} + \ldots$$

$$f_{i-1} = f_i - \Delta x f_i' + \frac{\Delta x^2}{2!} f_i'' - \frac{\Delta x^3}{3!} f_i^{(3)} + \ldots$$

Subtracting these equations cancels out the even-powered terms in Δx:

$$f_{i+1} - f_{i-1} = 2\Delta x f_i' + O(\Delta x^3)$$

Solving for f_i':

$$f_i' \approx \frac{f_{i+1} - f_{i-1}}{2\Delta x}$$

This *central difference* formula has an error proportional to Δx^2, making it second-order accurate—significantly more precise than the forward difference method.

Higher-Order Approximations

We can extend this approach to derive higher-order approximations by including more points in our calculation. A common fourth-order accurate formula for the first derivative is:

$$f_i' = \frac{1}{12\Delta x}(-f_{i-2} + 8f_{i-1} - 8f_{i+1} + f_{i+2})$$

This formula provides even better accuracy but requires function values at four points.

Comparison of Methods

The following table summarizes the key finite difference methods for first derivatives:

Method	Formula	Order of Accuracy	Points Required
Forward Difference	$\frac{f_{i+1}-f_i}{\Delta x}$	$O(\Delta x)$	2
Backward Difference	$\frac{f_i-f_{i-1}}{\Delta x}$	$O(\Delta x)$	2
Central Difference	$\frac{f_{i+1}-f_{i-1}}{2\Delta x}$	$O(\Delta x^2)$	3
Fourth-Order Central	$\frac{-f_{i+2}+8f_{i+1}-8f_{i-1}+f_{i-2}}{12\Delta x}$	$O(\Delta x^4)$	5

Higher-order methods generally provide more accurate results but require more computational resources and handle boundaries less efficiently.

15.0.3 Implementation in Python

Let's implement these numerical differentiation methods in Python, starting with a central difference function:

```python
#| autorun: false
def central_difference(f, x, h=1.e-5, *params):
    """Compute the first derivative using central difference"""
    return (f(x+h, *params)-f(x-h, *params))/(2*h)

def fourth_order_central(f, x, h=1.e-3, *params):
    """Compute the first derivative using fourth-order central difference"""
    return (-f(x+2*h, *params) + 8*f(x+h, *params) - 8*f(x-h, *params) + f(x-2*h, *params))/(12*h)
```

We can test these functions with $\sin(x)$, whose derivative is $\cos(x)$:

```
#| autorun: false
def f(x):
    return np.sin(x)

def analytical_derivative(x):
    return np.cos(x)

x_values = np.linspace(0, 2*np.pi, 100)

# Calculate derivatives using different methods
h_value = 0.01
forward_diff = [(f(x+h_value) - f(x))/h_value for x in x_values]
central_diff = [central_difference(f, x, h_value) for x in x_values]
fourth_order = [fourth_order_central(f, x, h_value) for x in x_values]
analytical = analytical_derivative(x_values)

# Plotting
plt.figure(figsize=get_size(12,8))
plt.plot(x_values, analytical, 'k-', label=r'$\cos(x)$')
plt.plot(x_values, forward_diff, 'r--', label='forward difference')
plt.plot(x_values, central_diff, 'g-.', label='central difference')
plt.plot(x_values, fourth_order, 'b:', label='4th-order central')
plt.xlabel('x')
plt.ylabel(r'derivative of $\sin(x)$')
plt.legend()
plt.show()
```

Error Analysis

Let's examine how the error in our numerical derivative varies with the step size $\Delta x$:

```
#| autorun: false
delta_x_values = np.logspace(-10, 0, 20)  # Step sizes from 10^-10 to 10^0
x0 = np.pi/4  # Test point

# Calculate errors for different methods
forward_errors = [abs((f(x0+dx) - f(x0))/dx - analytical_derivative(x0)) for dx in delta_x_values]
central_errors = [abs((f(x0+dx) - f(x0-dx))/(2*dx) - analytical_derivative(x0)) for dx in delta_x_value
fourth_order_errors = [abs((-f(x0+2*dx) + 8*f(x0+dx) - 8*f(x0-dx) + f(x0-2*dx))/(12*dx) - analytical_de

# Plotting errors
plt.figure(figsize=get_size(12,8))
plt.loglog(delta_x_values, forward_errors, 'ro-', label='Forward difference')
plt.loglog(delta_x_values, central_errors, 'go-', label='Central difference')
plt.loglog(delta_x_values, fourth_order_errors, 'bo-', label='4th-order central')
plt.loglog(delta_x_values, delta_x_values, 'k--', label=r'$O(\Delta x)$')
plt.loglog(delta_x_values, [dx**2 for dx in delta_x_values], 'k-.', label=r'$O(\Delta x^2)$')
plt.loglog(delta_x_values, [dx**4 for dx in delta_x_values], 'k:', label=r'$O(\Delta x^4)$')
plt.xlabel(r'step size ($\Delta x$)')
plt.ylabel('absolute error')
plt.legend()

plt.show()
```

This visualization demonstrates how error behaves with step size for different methods. For very small step sizes, roundoff errors become significant (observe the upturn in error for tiny $\Delta x$ values), while for larger steps, truncation error dominates.

15.0.4 Matrix Representation of Derivatives

An elegant approach to numerical differentiation involves representing the differentiation operation as a matrix multiplication. This representation is particularly valuable when solving differential equations numerically.

15.0.5 First Derivative Matrix

For a uniformly spaced grid of points x_i, we can represent the first derivative operation as a matrix:

$$f' = \frac{1}{\Delta x} \begin{bmatrix} -1 & 1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 1 & 0 & \cdots & 0 \\ 0 & 0 & -1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -1 & 1 \\ 0 & 0 & 0 & \cdots & 0 & -1 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \\ f_N \end{bmatrix}$$

This matrix implements the forward difference scheme. For a central difference scheme, the matrix would have entries on both sides of the diagonal.

Second Derivative Matrix

Similarly, the second derivative can be represented as a tridiagonal matrix:

$$f'' = \frac{1}{\Delta x^2} \begin{bmatrix} 1 & -2 & 1 & 0 & \cdots & 0 & \\ 0 & 1 & -2 & 1 & \cdots & 0 & \\ 0 & 0 & 1 & -2 & \cdots & 0 & \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \\ 0 & 0 & 0 & \cdots & 1 & -2 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \\ f_N \end{bmatrix}$$

The boundary conditions affect the structure of these matrices, especially the first and last rows.

Implementation with SciPy

SciPy provides tools to efficiently construct and work with these differentiation matrices:

```python
#| autorun: false
from scipy.sparse import diags

# Define a grid and a function to differentiate
N = 100
x = np.linspace(-5, 5, N)
dx = x[1] - x[0]
y = np.sin(x)

# First derivative matrix (forward difference)
D1_forward = diags([-1, 1], [0, 1], shape=(N, N)) / dx

# First derivative matrix (central difference)
# Corrected central difference implementation
D1_central = diags([-1, 0, 1], [-1, 0, 1], shape=(N, N))
# The central difference formula is (f(x+h) - f(x-h))/(2h)
# So we need -1 at offset -1, 0 at offset 0, and 1 at offset 1
D1_central.setdiag(0, 0)  # Center diagonal should be 0 for central difference
D1_central = D1_central / (2 * dx)
```

```
# Second derivative matrix
D2 = diags([1, -2, 1], [-1, 0, 1], shape=(N, N)) / dx**2

# Compute derivatives
dy_forward = D1_forward @ y
dy_central = D1_central @ y
d2y = D2 @ y

# Plot the results
plt.figure(figsize=get_size(16,12))

plt.subplot(3, 1, 1)
plt.plot(x, y, 'k-', label=r'$f(x) = \sin(x)$')
plt.legend()


plt.subplot(3, 1, 2)
plt.plot(x[:-1], dy_forward[:-1], 'r--', label='forward difference')
plt.plot(x, dy_central, 'g-', label='central difference')
plt.plot(x, np.cos(x), 'k:', label=r'$\cos(x)$')
plt.ylim(-1.05,1.05)
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(x[1:-1], d2y[1:-1], 'b-', label='num. 2nd derivative')
plt.plot(x, -np.sin(x), 'k:', label=r'$-\sin(x)$')
plt.legend()


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

15.0.6 Boundary Conditions

A critical consideration in numerical differentiation is how to handle the boundaries of the domain. Different approaches include:

1. **One-sided differences**: Using forward differences at the left boundary and backward differences at the right boundary.
2. **Extrapolation**: Extending the domain by extrapolating function values beyond the boundaries.
3. **Periodic boundaries**: For periodic functions, using values from the opposite end of the domain.
4. **Ghost points**: Introducing additional points outside the domain whose values are determined by the boundary conditions.

The choice of boundary treatment depends on the physical problem and can significantly impact the accuracy of the solution.

15.0.7 Applications in Physics

Numerical differentiation is foundational to computational physics. Let's explore some specific applications:

1. Solving Differential Equations

Many physics problems are formulated as differential equations. For example, the one-dimensional time-dependent Schrödinger equation:

$$i\hbar\frac{\partial}{\partial t}\Psi(x,t) = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\Psi(x,t) + V(x)\Psi(x,t)$$

Numerical differentiation allows us to approximate the spatial derivatives, reducing this to a system of ordinary differential equations in time.

2. Analysis of Experimental Data

When working with experimental measurements, we often need to calculate derivatives from discrete data points. For instance, determining the velocity and acceleration of an object from position measurements.

```
#| autorun: false
# Simulated noisy position data (as might come from an experiment)
time = np.linspace(0, 10, 100)
position = 5*time**2 + np.random.normal(0, 5, len(time))  # x = 5t² + noise

# Calculate velocity using central differences
dt = time[1] - time[0]
velocity = np.zeros_like(position)
for i in range(1, len(time)-1):
    velocity[i] = (position[i+1] - position[i-1]) / (2*dt)

# Theoretical velocity: v = 10t
theoretical_velocity = 10 * time

# Plot
plt.figure(figsize=get_size(16,12))

plt.subplot(2, 1, 1)
plt.plot(time, position, 'ko',alpha=0.2, label='experiment')
plt.plot(time, 5*time**2, 'r-', label=r'$x = 5t^2$')
plt.xlabel('time [s]')
plt.ylabel('position [m]')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(time[1:-1], velocity[1:-1], 'bo', label='calculated velocity')
plt.plot(time, theoretical_velocity, 'r-', label=r'theoretical velocity: $v = 10t$')
plt.xlabel('time [s]')
plt.ylabel('velocity [m/s]')
plt.legend()

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

Notice how noise in the position measurements gets amplified in the velocity calculations. This highlights a key challenge in numerical differentiation: sensitivity to noise.

3. Electric Field Calculation

In electrostatics, the electric field $\vec{E}$ is related to the electric potential ϕ by $\vec{E} = -\nabla\phi$. Numerical differentiation allows us to calculate the electric field from a known potential distribution.

```
#| autorun: false
# Create a 2D grid
x = np.linspace(-3, 3, 400)
y = np.linspace(-3, 3, 400)
```

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

# Electric potential due to a point charge at origin
potential = 1 / np.sqrt(X**2 + Y**2 + 0.001)  # Adding 0.01 to avoid division by zero

# Calculate electric field components
dx = x[1] - x[0]
dy = y[1] - y[0]
Ex = np.zeros_like(potential)
Ey = np.zeros_like(potential)

# Use central differences for interior points
for i in range(1, len(x)-1):
    for j in range(1, len(y)-1):
        Ex[j, i] = -(potential[j, i+1] - potential[j, i-1]) / (2*dx)
        Ey[j, i] = -(potential[j+1, i] - potential[j-1, i]) / (2*dy)

# Plot potential and electric field
plt.figure(figsize=get_size(16,6.5))

plt.subplot(1, 2, 1)
contour = plt.contourf(X, Y, potential, 20, cmap='viridis')
plt.colorbar(contour, label='electric potential')
plt.xlabel('x')
plt.ylabel('y')
plt.xlim(-2,2)
plt.ylim(-2,2)


plt.subplot(1, 2, 2)
# Skip some points for clearer visualization
skip = 5
plt.streamplot(X[::skip, ::skip], Y[::skip, ::skip],
               Ex[::skip, ::skip], Ey[::skip, ::skip],
               color='w', density=1.5)
plt.contourf(X, Y, np.sqrt(Ex**2 + Ey**2), 20, cmap='plasma', alpha=0.5)
plt.colorbar(label='electric field magnitude')

plt.xlabel('x')
plt.ylabel('y')
plt.xlim(-2,2)
plt.ylim(-2,2)


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

15.0.8 Practical Considerations and Challenges

1. Step Size Selection

Choosing an appropriate step size is crucial for numerical differentiation. If Δx is too large, the truncation error becomes significant. If Δx is too small, roundoff errors dominate. A general approach is to use:

$$\Delta x \approx \sqrt{\epsilon_{\text{machine}}} \times x$$

where $\epsilon_{\text{machine}}$ is the machine epsilon (approximately 10^{-16} for double precision).

2. Dealing with Noise

Numerical differentiation amplifies noise in the data. Several techniques can help:

- **Smoothing**: Apply a filter to the data before differentiation.
- **Regularization**: Use methods that inherently provide some smoothing.
- **Savitzky-Golay filters**: Combine local polynomial fitting with differentiation.

3. Conservation Properties

In physical simulations, preserving conservation laws (energy, momentum, etc.) is often crucial. Some numerical differentiation schemes conserve these properties better than others.

15.0.9 Using SciPy for Numerical Differentiation

The SciPy library provides convenient functions for numerical differentiation:

```
#| autorun: false
from scipy.misc import derivative

# Calculate the derivative of sin(x) at x =  /4
x0 = np.pi/4

# First derivative with different accuracies
first_deriv = derivative(np.sin, x0, dx=1e-6, n=1, order=3)
print(f"First derivative of sin(x) at x =  /4: {first_deriv}")
print(f"Actual value (cos( /4)): {np.cos(x0)}")

# Second derivative
second_deriv = derivative(np.sin, x0, dx=1e-6, n=2, order=5)
print(f"Second derivative of sin(x) at x =  /4: {second_deriv}")
print(f"Actual value (-sin( /4)): {-np.sin(x0)}")
```

The `order` parameter controls the accuracy of the approximation by using more points in the calculation.

15.0.10 Conclusion

Numerical differentiation is a fundamental technique in computational physics, bridging the gap between theoretical models and practical computations. By understanding the principles, methods, and challenges of numerical differentiation, physicists can effectively analyze data, solve differential equations, and simulate physical systems.

The methods we've explored—from simple finite differences to matrix representations—provide a comprehensive toolkit for tackling a wide range of physics problems. As you apply these techniques, remember that the choice of method should be guided by the specific requirements of your problem: accuracy needs, computational constraints, and the nature of your data.

15.0.11 What to try yourself

1. Implement and compare the accuracy of different numerical differentiation schemes for the function $f(x) = e^{-x^2}$.

2. Investigate how noise in the input data affects the accuracy of numerical derivatives and explore techniques to mitigate this effect.

3. Calculate the electric field around two point charges using numerical differentiation of the electric potential.

4. Analyze experimental data from a falling object to determine its acceleration, and compare with the expected value of gravitational acceleration.

Further Reading

- Numerical Recipes: The Art of Scientific Computing by Press, Teukolsky, Vetterling, and Flannery
- Numerical Methods for Physics by Alejandro Garcia
- Computational Physics by Mark Newman
- Applied Numerical Analysis by Curtis F. Gerald and Patrick O. Wheatley

Part VI

Lecture 6

Chapter 16

Introduction to Numerical Integration in Physics

Numerical integration stands as one of the fundamental computational tools in physics, enabling us to solve problems where analytical solutions are either impossible or impractical. In physics, we frequently encounter integrals when calculating quantities such as:

- Work done by a variable force
- Electric and magnetic fields from complex charge distributions
- Center of mass of irregularly shaped objects
- Probability distributions in quantum mechanics
- Energy levels in quantum wells with arbitrary potentials
- Heat flow in non-uniform materials

The need for numerical integration arises because many physical systems are described by functions that cannot be integrated analytically. For example, the potential energy of a complex molecular system, the trajectory of a spacecraft under multiple gravitational influences, or the behavior of quantum particles in complex potentials.

In this lecture, we'll explore three progressively more accurate numerical integration methods: the Box method, Trapezoid method, and Simpson's method. We'll analyze their accuracy, efficiency, and appropriate applications in physical problems.

```
#| 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)
```

16.0.1 Box Method (Rectangle Method)

Theory and Implementation

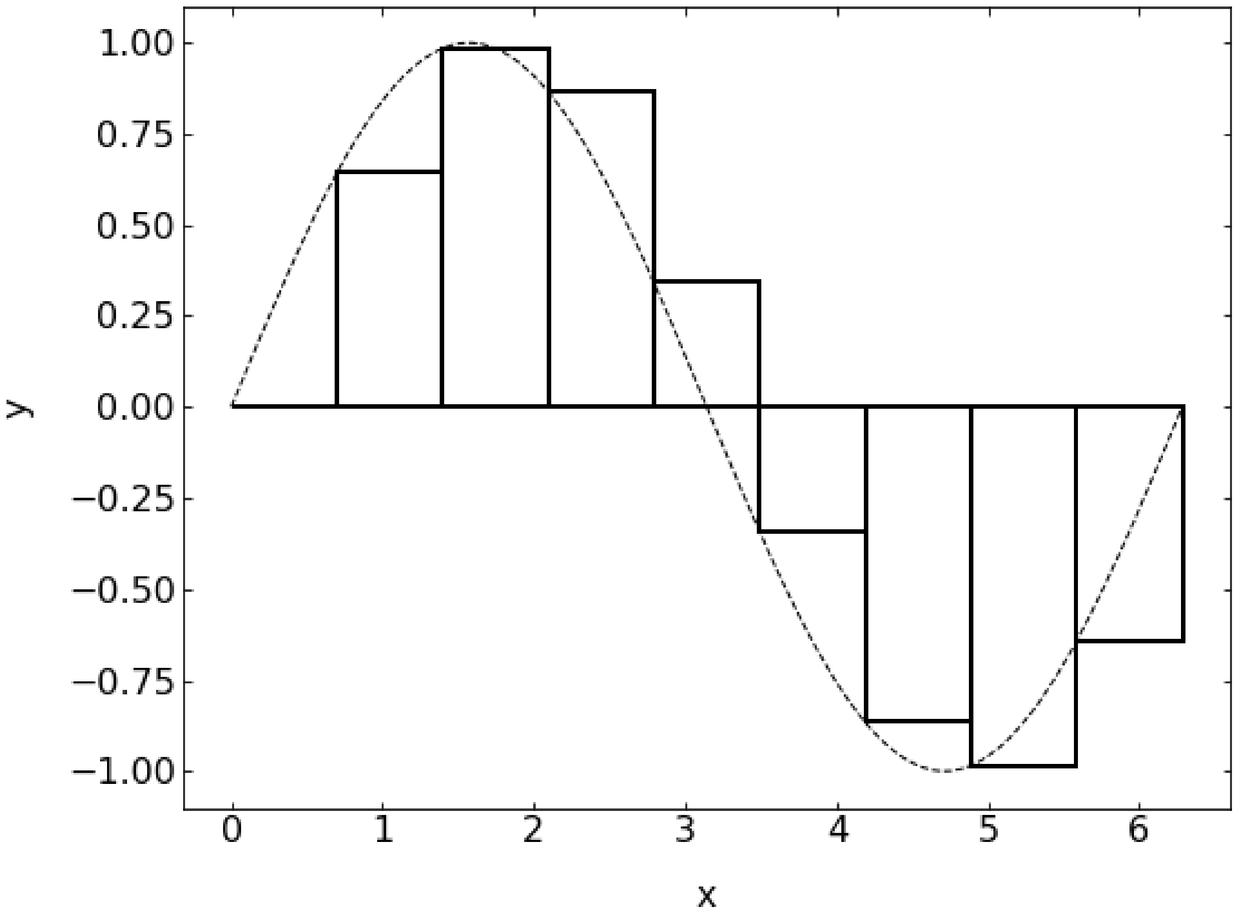


Figure 16.1: Box Method Illustration

The Box method (also known as the Rectangle method) represents the simplest approach for numerical integration. It approximates the function in each interval Δx with a constant value taken at a specific point of the interval—typically the left endpoint, although midpoint or right endpoint variants exist.

Mathematically, the definite integral is approximated as:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} f(x_i)\Delta x \tag{16.1}$$

Where:

- a and b are the integration limits
- N is the number of intervals
- $\Delta x = \frac{b-a}{N}$ is the width of each interval

- x_i represents the left endpoint of each interval

The method gets its name from the visual representation of the approximation as a series of rectangular boxes.

Box Method Applications

16.0.2 Physics Application: Work Calculation

Consider a particle moving along a straight line under a variable force $F(x) = kx^2$ where k is some constant. The work done by this force when moving the particle from position $x = 0$ to $x = L$ is given by:

$$W = \int_0^L F(x)dx = \int_0^L kx^2 dx \tag{16.2}$$

We can approximate this integral using the Box method, particularly useful when the force follows a complex pattern measured at discrete points.

```python
def f(x):
    """Example function to integrate: f(x) = x"""
    return x

def force(x, k=1.0):
    """Force function F(x) = kx^2 for work calculation"""
    return k * x**2

def int_box(f, a, b, N):
    """Box method integration"""
    if N < 2:
        raise ValueError("N must be at least 2")
    x = np.linspace(a, b, N)
    y = f(x)
    dx = (b-a)/(N-1)
    return np.sum(y[:-1] * dx)  # Sum using left endpoints

# Example: Calculate work done by a nonlinear force
L = 2.0  # meters
k = 0.5  # N/m^3
N_points = 100

work = int_box(lambda x: force(x, k), 0, L, N_points)
print(f"Work done by force F(x) = {k}x² from x=0 to x={L} m:")
print(f"Numerical result (Box method): {work:.6f} J")
print(f"Analytical result: {k*L**3/3:.6f} J")

# Demonstrate the method graphically
x_demo = np.linspace(0, 1, 6)
y_demo = x_demo  # Using f(x) = x for demonstration
dx_demo = x_demo[1] - x_demo[0]

plt.figure(figsize=get_size(15, 10))

# Plot the actual function
x_fine = np.linspace(0, 1, 1000)
y_fine = x_fine
plt.plot(x_fine, y_fine, 'b-', label='actual function f(x) = x')

# Plot the boxes
```

```
for i in range(len(x_demo)-1):
    plt.fill_between([x_demo[i], x_demo[i+1]], [y_demo[i], y_demo[i]], alpha=0.3, color='red')
    plt.plot([x_demo[i], x_demo[i]], [0, y_demo[i]], 'r--', alpha=0.5)

plt.plot(x_demo[:-1], y_demo[:-1], 'ro', label='Sample points')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.tight_layout()
plt.show()
```

16.0.3 Convergence Analysis

```
# Convergence demonstration
N_values = np.arange(10, 10000, 100)
box_results = [int_box(f, 0, 1, N) for N in N_values]
exact_value = 0.5  # Exact integral of f(x) = x from 0 to 1

# Calculate absolute errors
box_errors = np.abs(np.array(box_results) - exact_value)

# Fit a power law to error vs N
def power_law(x, a, b):
    return a * x**b

popt, _ = curve_fit(power_law, N_values, box_errors)

plt.figure(figsize=get_size(15, 10))
plt.loglog(N_values, box_errors, 'o', label='Box Method Error')
plt.loglog(N_values, power_law(N_values, *popt), '--',
         label=f'Power Law Fit: error   N^{popt[1]:.2f}')
plt.xlabel('number of points [N]')
plt.ylabel('absolute Error')
plt.legend()
plt.tight_layout()
plt.show()

print(f"Box Method convergence rate: approximately O(N^{popt[1]:.2f})")
```

16.0.4 Trapezoid Method

Theory and Implementation

The Trapezoid method improves upon the Box method by approximating the function with linear segments between consecutive points. Instead of using constant values within each interval, it connects adjacent points with straight lines, forming trapezoids.

The mathematical formula for the Trapezoid method is:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N-1} \frac{f(x_i) + f(x_{i+1})}{2} \Delta x \tag{16.3}$$

Where: - $\Delta x = \frac{b-a}{N-1}$ is the width of each interval - x_i are the sample points

This method is particularly effective for smoothly varying functions, which are common in physical systems.

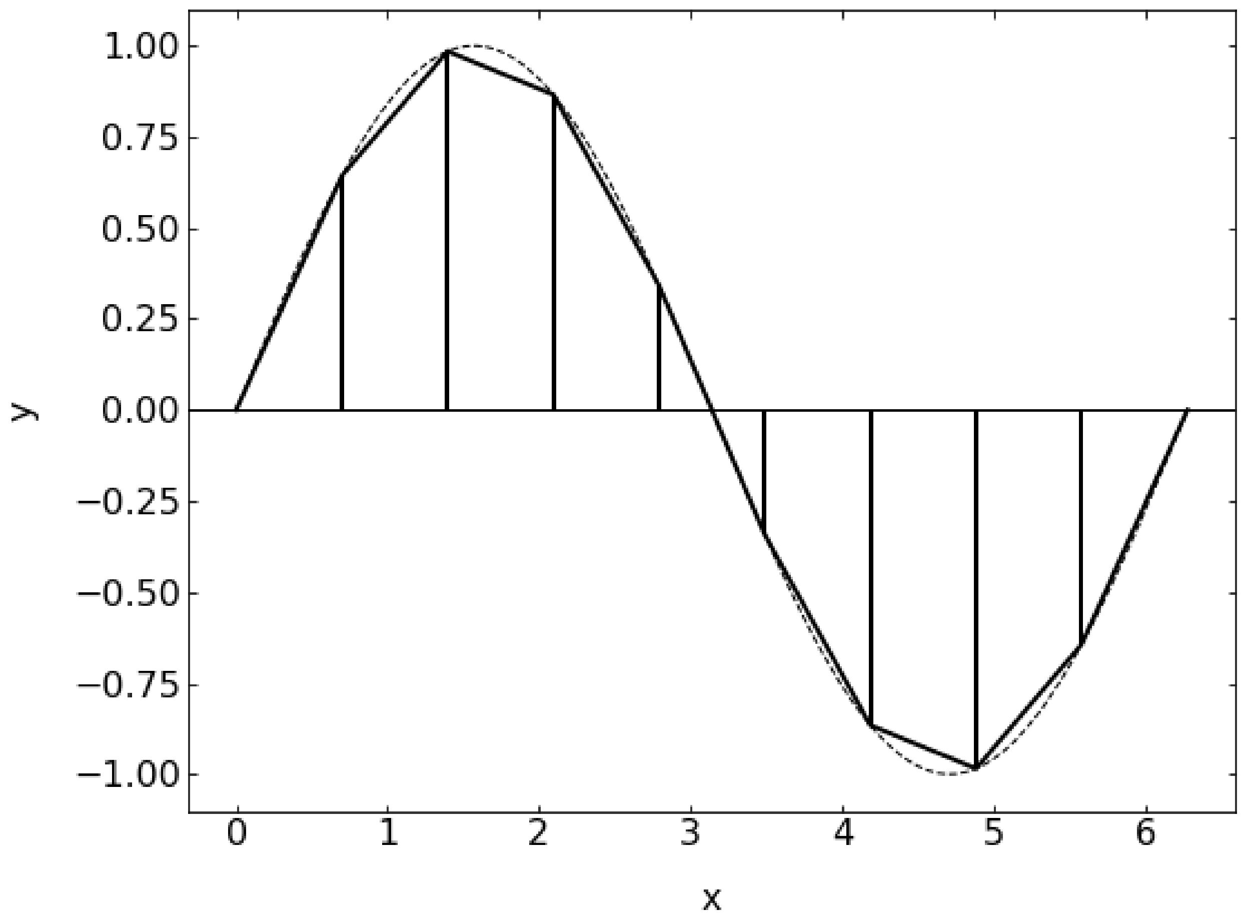


Figure 16.2: Trapezoid Method Illustration

Trapezoid Application

16.0.5 Physics Application: Electric Potential from Charge Distribution

A practical application in electromagnetism involves calculating the electric potential at a point due to a non-uniform charge distribution along a line. For a linear charge density $\lambda(x)$ along the x-axis, the potential at point $(0, d)$ is given by:

$$V(0,d) = \frac{1}{4\pi\epsilon_0}\int_a^b \frac{\lambda(x)}{\sqrt{x^2+d^2}}dx \tag{16.4}$$

The Trapezoid method is well-suited for this calculation, especially when $\lambda(x)$ is provided as experimental data points.

```
def int_trap(f, a, b, N):
    """Trapezoid method integration"""
    if N < 2:
        raise ValueError("N must be at least 2")
    x = np.linspace(a, b, N)
    y = f(x)
    dx = (b-a)/(N-1)
    return np.sum((y[1:] + y[:-1]) * dx/2)

# Simple demonstration
def f_demo(x):
    return x**2  # Using f(x) = x² for demonstration

# Demonstrate the trapezoid method visually
x_demo = np.linspace(0, 1, 6)
y_demo = f_demo(x_demo)

plt.figure(figsize=get_size(15, 8))
# Plot the actual function
x_fine = np.linspace(0, 1, 1000)
y_fine = f_demo(x_fine)
plt.plot(x_fine, y_fine, 'b-', label='actual function f(x) = x²')

# Plot the trapezoids
for i in range(len(x_demo)-1):
    plt.fill_between([x_demo[i], x_demo[i+1]],
                     [y_demo[i], y_demo[i+1]],
                     alpha=0.3, color='green')
    plt.plot([x_demo[i], x_demo[i+1]], [y_demo[i], y_demo[i+1]], 'g-')

plt.plot(x_demo, y_demo, 'go', label='sample points')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.tight_layout()
plt.legend()
plt.show()
```

16.0.6 Convergence Analysis

```
# Convergence demonstration
N_values = np.arange(10, 1000, 10)
trap_results = [int_trap(f_demo, 0, 1, N) for N in N_values]
```

```
exact_value = 1/3  # Exact integral of x² from 0 to 1

# Calculate absolute errors
trap_errors = np.abs(np.array(trap_results) - exact_value)

# Fit a power law to error vs N
def power_law(x, a, b):
    return a * x**b

popt, _ = curve_fit(power_law, N_values, trap_errors)

plt.figure(figsize=get_size(12, 10))
plt.loglog(N_values, trap_errors, 'o', label='trapezoid method error')
plt.loglog(N_values, power_law(N_values, *popt), '--',
         label=f'power law fit: error   N^{popt[1]:.2f}')
plt.xlabel('number of points [N]')
plt.ylabel('absolute error')
plt.legend()
plt.tight_layout()
plt.show()

print(f"Trapezoid Method convergence rate: approximately O(N^{popt[1]:.2f})")
```

16.0.7 Simpson's Method

Theory and Implementation

Simpson's method represents a significant improvement in accuracy over the previous methods by approximating the function with parabolic segments rather than straight lines. This approach is particularly effective for functions with curvature, which are ubiquitous in physics problems.

The mathematical formulation of Simpson's rule is:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{3} \sum_{i=0}^{(N-1)/2} \left(f(x_{2i}) + 4f(x_{2i+1}) + f(x_{2i+2})\right) \tag{16.5}$$

Where: - N is the number of intervals (must be even) - $\Delta x = \frac{b-a}{N}$ is the width of each interval

Simpson's rule is derived from fitting a quadratic polynomial through every three consecutive points and then integrating these polynomials.

Simpson's Nethod Applications

16.0.8 Physics Application: Quantum Mechanical Probability

A fundamental application in quantum mechanics involves calculating the probability of finding a particle in a region of space. For a wavefunction $\psi(x)$, the probability of finding the particle between positions a and b is:

$$P(a \leq x \leq b) = \int_a^b |\psi(x)|^2 dx \tag{16.6}$$

Simpson's method provides high accuracy for these calculations, especially important when dealing with oscillatory wavefunctions.

```
def int_simp(f, a, b, N):
    """Simpson's method integration"""
```

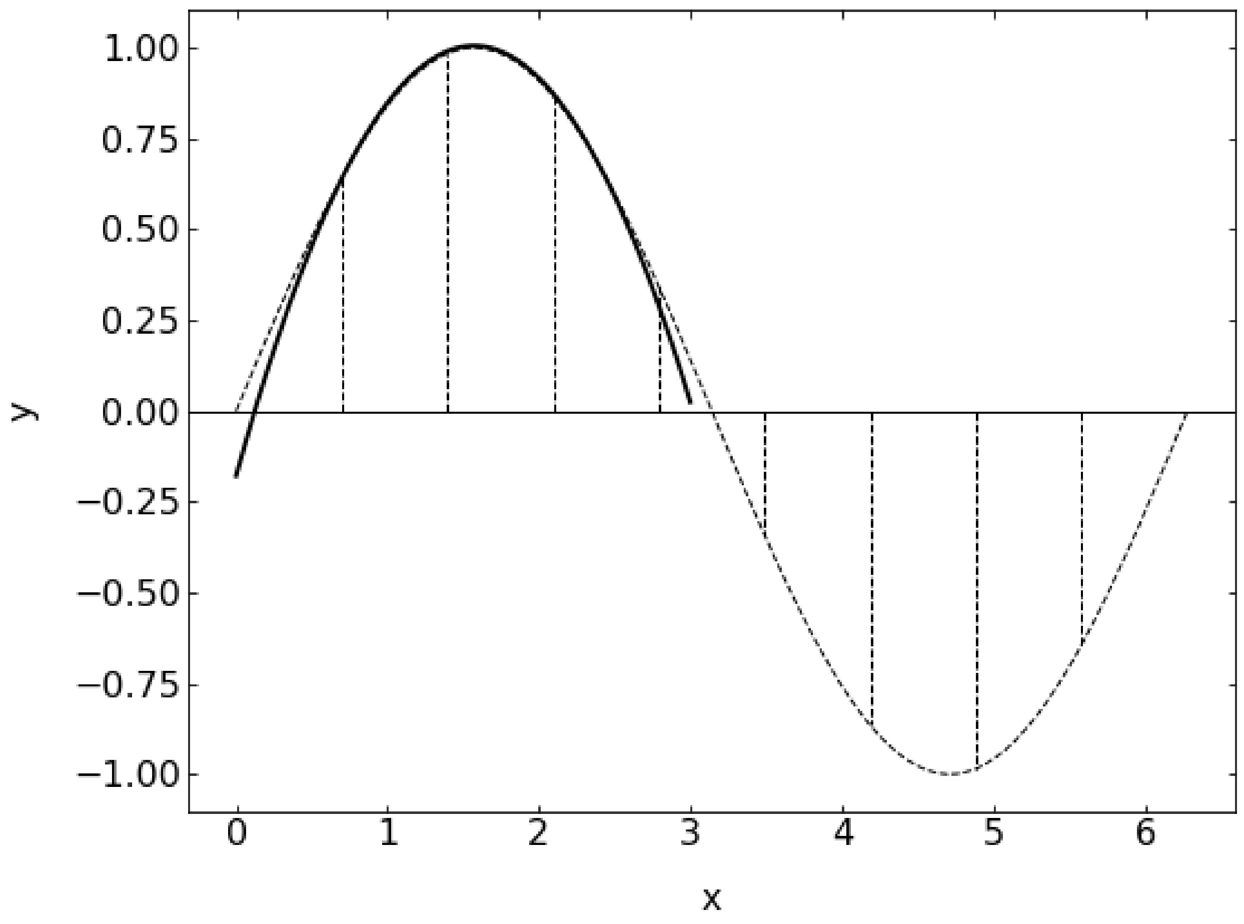


Figure 16.3: Simpson's Method Illustration

```
    if N % 2 == 0:
        N = N + 1  # Ensure N is odd for Simpson's rule

    if N < 3:
        raise ValueError("N must be at least 3 for Simpson's method")

    x = np.linspace(a, b, N)
    y = f(x)
    dx = (b-a)/(N-1)

    # Apply Simpson's formula
    return dx/3 * np.sum(y[0:-2:2] + 4*y[1:-1:2] + y[2::2])

# Demonstrate Simpson's method with a simple example
def f_demo(x):
    """Example function: sin(x)"""
    return np.sin(x)

# Visualize Simpson's method with a few segments
x_demo = np.linspace(0, np.pi, 7)  # 6 intervals
y_demo = f_demo(x_demo)

plt.figure(figsize=get_size(15, 8))
# Plot the actual function
x_fine = np.linspace(0, np.pi, 1000)
y_fine = f_demo(x_fine)
plt.plot(x_fine, y_fine, 'b-', label='f(x) = sin(x)')

# Plot the parabolic segments
for i in range(0, len(x_demo)-2, 2):
    x_segment = np.linspace(x_demo[i], x_demo[i+2], 50)

    # Fit a quadratic polynomial through three points
    x_points = x_demo[i:i+3]
    y_points = y_demo[i:i+3]
    coeffs = np.polyfit(x_points, y_points, 2)
    y_fit = np.polyval(coeffs, x_segment)

    plt.plot(x_segment, y_fit, 'r-', alpha=0.7)
    plt.fill_between(x_segment, 0, y_fit, alpha=0.2, color='purple')

plt.plot(x_demo, y_demo, 'go', label='Sample points')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title("Simpson's Method Visualization")
plt.grid(True)
plt.legend()
plt.show()
```

16.0.9 Convergence Analysis

```
# Convergence demonstration with a simple test function
f_test = lambda x: x**4  # We can integrate x  exactly
exact_value = 1/5  # Exact integral of x  from 0 to 1
```

```python
# Compare accuracy with different numbers of points
N_values = np.arange(3, 101, 2)  # Odd numbers for Simpson's rule
simp_errors = [abs(int_simp(f_test, 0, 1, n) - exact_value) for n in N_values]

# Fit a power law to error vs N
popt, _ = curve_fit(power_law, N_values, simp_errors)

plt.figure(figsize=get_size(12, 10))
plt.loglog(N_values, simp_errors, 'o', label="Simpson's method error")
plt.loglog(N_values, power_law(N_values, *popt), '--',
         label=f'Power law fit: error   N^{popt[1]:.2f}')
plt.xlabel('number of points [N]')
plt.ylabel('absolute error')
plt.legend()
plt.tight_layout()
plt.show()

print(f"Simpson's Method convergence rate: approximately O(N^{popt[1]:.2f})")
```

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:

$$\begin{aligned} f(x_0) &= Ax_0^2 + Bx_0 + C \\ f(x_1) &= Ax_1^2 + Bx_1 + C \\ f(x_2) &= Ax_2^2 + Bx_2 + C \end{aligned}$$

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.

Final Formula

The integration of this polynomial leads to Simpson's Rule:

$$\int_a^b f(x)dx \approx \frac{h}{3}[f(a) + 4f(\frac{a+b}{2}) + f(b)]$$

16.0.10 Error Term

The error in Simpson's Rule is proportional to:

$$-\frac{h^5}{90}f^{(4)}(\xi)$$

for some $\xi \in [a, b]$

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.

16.0.11 When to Use Each Method

Choosing the appropriate numerical integration method for a physics problem requires consideration of several factors:

Method	Error Order	Optimal For	Limitations	Example Physics Applications
Box	$O(N^{-1})$	- Simple, rapid calculations- Step functions- Real-time processing- Discontinuous functions	- Low accuracy- Requires many points for decent results	- Basic data analysis- Signals with sharp transitions- First approximations in mechanics
Trapezoid	$O(N^{-2})$	- Smooth, continuous functions- Moderate accuracy requirements- Periodic functions	- Struggles with sharp peaks- Not ideal for higher derivatives	- Electric and magnetic fields- Orbital mechanics- Path integrals- Work/energy calculations
Simpson	$O(N^{-4})$	- Functions with significant curvature- High precision requirements- Oscillatory integrands	- More computationally intensive- Requires evenly spaced points	- Quantum mechanical probabilities- Wave optics- Statistical mechanics- Thermal physics

Additional Considerations

- **Adaptive methods** can be more efficient when dealing with functions that have varying behavior across the integration range
- **Improper integrals** (with infinite limits or singularities) often require specialized techniques beyond these basic methods

- **Higher-dimensional** integration problems (common in statistical and quantum mechanics) may benefit from Monte Carlo methods rather than these quadrature rules

16.1 Error Analysis

The error behavior of numerical integration methods is crucial for understanding their applicability to physics problems:

- **Box Method**: Error $\propto O(\Delta x) = O(N^{-1})$ (linear convergence)
  - The error is proportional to the step size
  - The dominant error term comes from the first derivative of the function
- **Trapezoid Method**: Error $\propto O(\Delta x^2) = O(N^{-2})$ (quadratic convergence)
  - The error is proportional to the square of the step size
  - The dominant error term involves the second derivative of the function
- **Simpson's Method**: Error $\propto O(\Delta x^4) = O(N^{-4})$ (fourth-order convergence)
  - The error is proportional to the fourth power of the step size
  - The dominant error term involves the fourth derivative of the function

Consequently, if we double the number of points:

- Box method: error reduced by a factor of 2
- Trapezoid method: error reduced by a factor of 4
- Simpson's method: error reduced by a factor of 16

The practical impact of these convergence rates is substantial. For example, to achieve an error of 10^{-6} for a well-behaved function:

- Box method might require millions of points
- Trapezoid method might require thousands of points
- Simpson's method might require only hundreds of points

This explains why higher-order methods are generally preferred for physics applications requiring high precision, such as quantum mechanical calculations, gravitational wave analysis, or computational fluid dynamics.

16.1.1 Conclusion

Numerical integration stands as a cornerstone of computational physics, bridging the gap between theoretical models and practical analysis of real-world systems. In this lecture, we've explored three fundamental methods—Box, Trapezoid, and Simpson's method—that provide different balances of simplicity, accuracy, and computational efficiency.

The key insights from our study include:

1. **Method selection matters**: The choice of integration technique can dramatically impact both accuracy and computational efficiency. For typical physics applications requiring high precision, Simpson's method often provides the optimal balance of accuracy and computation cost.

2. **Error scaling**: Understanding how errors scale with the number of sampling points is crucial for reliable scientific computation. The higher-order convergence of Simpson's method ($O(N^{-4})$) makes it particularly valuable for precision-critical applications in physics.

3. **Physical context**: The nature of the underlying physical system should guide your choice of numerical method. Smoothly varying functions benefit from higher-order methods, while functions with discontinuities may require adaptive or specialized approaches.

4. **Verification**: Always verify numerical results against analytical solutions when possible, or compare results from different numerical methods with increasing resolution to establish confidence in your calculations.

As you progress in your physics education, these numerical integration techniques will become essential tools in your computational toolkit, enabling you to tackle increasingly complex physical systems from quantum mechanics to astrophysics, fluid dynamics, and beyond.

In advanced courses, you'll explore additional techniques such as Gaussian quadrature, Romberg integration, and specialized methods for oscillatory, singular, or multi-dimensional integrals—each designed to address specific challenges encountered in modern physics.

Remember that numerical integration is not merely a computational technique but a powerful approach to understanding physical systems that resist analytical treatment, making it an indispensable skill for the modern physicist.

Advanced Topics in Numerical Integration

16.1.2 Adaptive Integration Methods

The methods we've discussed so far use equal spacing between sampling points. However, most real-world physics problems involve functions that vary dramatically across the integration range. Adaptive methods adjust the point distribution to concentrate more points where the function changes rapidly.

```
# Simple demonstration of adaptive integration concept
def adaptive_demo(f, a, b, tol=1e-5, max_depth=10):
    """Simple demonstration of adaptive integration concept"""
    def recursive_integrate(a, b, depth=0):
        # Compute midpoint
        c = (a + b) / 2

        # Estimate integral using Simpson's rule on entire interval
        I_whole = (b-a)/6 * (f(a) + 4*f(c) + f(b))

        # Estimate integral using Simpson's rule on each half
        mid1 = (a + c) / 2
        mid2 = (c + b) / 2
        I_left = (c-a)/6 * (f(a) + 4*f(mid1) + f(c))
        I_right = (b-c)/6 * (f(c) + 4*f(mid2) + f(b))
        I_parts = I_left + I_right

        # Check error
        error = abs(I_whole - I_parts)

        # If error is small enough or max depth reached, return result
        if error < tol*(b-a) or depth >= max_depth:
            return I_parts, [(a, c, b, error)]

        # Otherwise, recursively integrate each half
        I_left_result, left_regions = recursive_integrate(a, c, depth+1)
        I_right_result, right_regions = recursive_integrate(c, b, depth+1)

        return I_left_result + I_right_result, left_regions + right_regions

    integral, regions = recursive_integrate(a, b)
    return integral, regions

# Define a challenging function with a sharp peak
def challenging_func(x):
    """Function with a sharp peak at x=0.7"""
    return 1 / (0.01 + (x - 0.7)**2)

# Calculate integral using adaptive and non-adaptive methods
a, b = 0, 1
n_points = 101  # Use a fixed number of points for non-adaptive methods

# Calculate using non-adaptive methods
box_result = int_box(challenging_func, a, b, n_points)
trap_result = int_trap(challenging_func, a, b, n_points)
simp_result = int_simp(challenging_func, a, b, n_points)

# Calculate using adaptive method
adaptive_result, regions = adaptive_demo(challenging_func, a, b)

# Calculate a reference solution using a very high number of points
reference = int_simp(challenging_func, a, b, 10001)

# Visualize the function and integration points
x_fine = np.linspace(a, b, 1000)
y_fine = [challenging_func(x) for x in x_fine]

plt.figure(figsize=get_size(15, 10))

# Plot the function
plt.plot(x_fine, y_fine, 'k-', label='f(x)')

# Plot the adaptive integration regions
```

16.1.3 Multi-dimensional Integration

Many physics problems require integration over multiple dimensions, such as calculating mass moments of inertia, electric fields from volume charge distributions, or statistical mechanics partition functions. For 2D integration, we can extend our 1D methods using the concept of iterated integrals:

$$\int_a^b \int_c^d f(x,y)dydx \approx \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} w_i w_j f(x_i, y_j) \tag{16.7}$$

Where w_i and w_j are the weights for the respective 1D methods.

```
def int_2d_trap(f, x_range, y_range, nx, ny):
    """2D integration using the Trapezoid method"""
    x = np.linspace(x_range[0], x_range[1], nx)
    y = np.linspace(y_range[0], y_range[1], ny)
    dx = (x_range[1] - x_range[0]) / (nx - 1)
    dy = (y_range[1] - y_range[0]) / (ny - 1)

    result = 0
    for i in range(nx-1):
        for j in range(ny-1):
            # Average of function values at the four corners of each cell
            f_values = [
                f(x[i], y[j]),
                f(x[i+1], y[j]),
                f(x[i], y[j+1]),
                f(x[i+1], y[j+1])
            ]
            result += sum(f_values) / 4 * dx * dy

    return result

# Example: Electric potential from a square charge distribution
def potential_2d(x, y, z=1):
    """Electric potential at (x,y,z) from a square charge distribution in the x-y plane"""
    # Avoid division by zero
    denominator = (x**2 + y**2 + z**2)**0.5
    if denominator < 1e-10:
        return 0
    return 1 / denominator

# Calculate the electric potential at a point above a charged square
z = 1.0  # height above the plane
potential = int_2d_trap(lambda x, y: potential_2d(x, y, z), [-1, 1], [-1, 1], 51, 51)

# Visualize the potential in the plane above the charge
x_vals = np.linspace(-2, 2, 50)
y_vals = np.linspace(-2, 2, 50)
X, Y = np.meshgrid(x_vals, y_vals)
Z = np.zeros_like(X)

for i in range(len(x_vals)):
    for j in range(len(y_vals)):
        Z[j, i] = potential_2d(X[j, i], Y[j, i], z)

plt.figure(figsize=get_size(15, 10))

# Plot the potential as a contour
contour = plt.contourf(X, Y, Z, 50, cmap='viridis')
plt.colorbar(label='Electric Potential')

# Mark the square charge distribution
plt.plot([-1, 1, 1, -1, -1], [-1, -1, 1, 1, -1], 'r-', linewidth=2)

plt.xlabel('x')
plt.ylabel('y')
plt.axis('equal')

plt.show()

print(f"Electric potential at point (0,0,{z}): {potential:.6f}")
print(f"Analytical value at center: {np.log(1 + np.sqrt(2)):.6f}")
```

16.1.4 Monte Carlo Integration

For higher-dimensional integrals and complex domains, Monte Carlo methods become increasingly efficient. These methods use random sampling to approximate integrals and are particularly valuable in quantum and statistical physics.

```
def monte_carlo_integrate(f, ranges, n_samples):
    """Monte Carlo integration in arbitrary dimensions"""
    # Generate random samples within the integration domain
    dim = len(ranges)
    samples = np.random.uniform(
        low=[r[0] for r in ranges],
        high=[r[1] for r in ranges],
        size=(n_samples, dim)
    )

    # Evaluate function at sample points
    f_values = np.array([f(*sample) for sample in samples])

    # Calculate volume of integration domain
    volume = np.prod([r[1] - r[0] for r in ranges])

    # Estimate integral and error
    integral = volume * np.mean(f_values)
    error = volume * np.std(f_values) / np.sqrt(n_samples)

    return integral, error

# Example: Calculate the volume of a n-dimensional hypersphere
def sphere_indicator(x, y, z):
    """Return 1 if point is inside unit sphere, 0 otherwise"""
    return 1 if x**2 + y**2 + z**2 <= 1 else 0

# Calculate the volume of a 3D sphere using Monte Carlo
n_samples = 100000
ranges = [[-1, 1], [-1, 1], [-1, 1]]  # Cube containing the sphere
volume, error = monte_carlo_integrate(sphere_indicator, ranges, n_samples)

# The exact volume of a unit sphere in 3D is (4/3)
exact_volume = 4/3 * np.pi

# Visualize the convergence
sample_sizes = np.logspace(2, 5, 20, dtype=int)
volumes = []
errors = []

for n in sample_sizes:
    v, e = monte_carlo_integrate(sphere_indicator, ranges, n)
    volumes.append(v)
    errors.append(e)

plt.figure(figsize=get_size(15, 10))
plt.semilogx(sample_sizes, volumes, 'o-', label='Monte Carlo estimate')
plt.fill_between(
    sample_sizes,
    [v - e for v, e in zip(volumes, errors)],
    [v + e for v, e in zip(volumes, errors)],
    alpha=0.3
)
plt.axhline(exact_volume, color='r', linestyle='--', label='Exact value')
plt.xlabel('Number of samples')
plt.ylabel('Volume of unit sphere')
plt.legend()
plt.show()

print(f"Volume of unit sphere (Monte Carlo with {n_samples} samples): {volume:.6f} ± {error:.6f}")
print(f"Exact volume of unit sphere: {exact_volume:.6f}")
print(f"Relative error: {abs(volume - exact_volume)/exact_volume*100:.4f}%")
```

16.1.5 Application to Real Physics Problems

Let's examine two common scenarios in physics that benefit from numerical integration:

1. **Non-uniform Magnetic Field**: When a charged particle moves through a non-uniform magnetic field, the work done can be calculated as:

$$W = q \int_{\vec{r}_1}^{\vec{r}_2} \vec{v} \times \vec{B}(\vec{r}) \cdot d\vec{r}$$

2. **Quantum Tunneling**: The tunneling probability through a potential barrier is given by:

$$T \approx \exp\left(-\frac{2}{\hbar} \int_{x_1}^{x_2} \sqrt{2m(V(x) - E)}\, dx\right)$$

In both cases, the integrals frequently cannot be solved analytically due to the complex spatial dependence of the fields or potentials, making numerical integration indispensable for modern physics.

Chapter 17

Solving Ordinary Differential Equations

17.1 Introduction

In the previous lecture on numerical differentiation, we explored how to compute derivatives numerically using finite difference methods and matrix representations. We learned that:

1. Finite difference approximations allow us to estimate derivatives at discrete points
2. Differentiation can be represented as matrix operations
3. The accuracy of these approximations depends on the order of the method and step size

Building on this foundation, we can now tackle one of the most important applications in computational physics: solving ordinary differential equations (ODEs). Almost all dynamical systems in physics are described by differential equations, and learning how to solve them numerically is essential for modeling physical phenomena.

As second-semester physics students, you've likely encountered differential equations in various contexts:

- Newton's second law: $F = ma = m\frac{d^2x}{dt^2}$
- Simple harmonic motion: $\frac{d^2x}{dt^2} + \omega^2 x = 0$
- RC circuits: $\frac{dQ}{dt} + \frac{1}{RC}Q = 0$
- Heat diffusion: $\frac{\partial T}{\partial t} = \alpha\nabla^2 T$

While analytical solutions exist for some simple cases, real-world physics problems often involve complex systems where analytical solutions are either impossible or impractical to obtain. This is where numerical methods become indispensable tools for the working physicist.

```
#| 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
from scipy.integrate import solve_ivp

# default values for plotting
plt.rcParams.update({'font.size': 12,
                     'lines.linewidth': 1.5,
                     '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))

def set_plot_style():
    plt.style.use('seaborn-v0_8-whitegrid')
    plt.rcParams.update({
        'figure.figsize': (10, 6),
        'axes.grid': True,
        'grid.linestyle': '--',
        'grid.alpha': 0.7,
        'lines.linewidth': 2,
        'font.size': 12,
        'axes.labelsize': 14,
        'axes.titlesize': 16,
        'xtick.labelsize': 12,
        'ytick.labelsize': 12
    })
```

17.2 Types of ODE Solution Methods

There are two main approaches to solving ODEs numerically:

1. **Implicit methods**: These methods solve for all time points simultaneously using matrix operations. They treat the problem as a large system of coupled algebraic equations. Just as you might solve a system of linear equations $Ax = b$ in linear algebra, implicit methods set up and solve a larger matrix equation that represents the entire evolution of the system.

2. **Explicit methods**: These methods march forward in time, computing the solution step by step. Starting from the initial conditions, they use the current state to calculate the next state, similar to how you might use the position and velocity at time t to predict the position and velocity at time $t + \Delta t$.

We'll explore both approaches, highlighting their strengths and limitations. Each has its place in physics: implicit methods often handle "stiff" problems better (problems with vastly different timescales), while explicit methods are typically easier to implement and can handle nonlinear problems more naturally.

17.3 The Harmonic Oscillator: A Prototypical ODE

Physics Interlude: The Harmonic Oscillator

The harmonic oscillator is one of the most fundamental systems in physics, appearing in mechanics, electromagnetism, quantum mechanics, and many other fields. It serves as an excellent test case for ODE solvers due to its simplicity and known analytical solution.
The harmonic oscillator describes any system that experiences a restoring force proportional to displacement. Physically, this occurs in:

- A mass on a spring (Hooke's law: $F = -kx$)
- A pendulum for small angles (where $\sin\theta \approx \theta$)

- An LC circuit with inductors and capacitors
- Molecular vibrations in chemistry
- Phonons in solid state physics

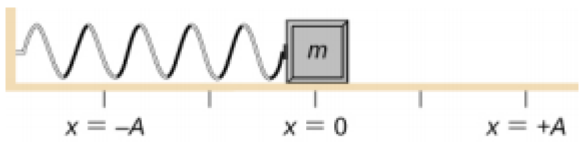


Figure 17.1: A mass suspended from a spring demonstrates a classic harmonic oscillator. When displaced from equilibrium, the spring exerts a restoring force proportional to the displacement, leading to oscillatory motion.

The equation of motion for a classical harmonic oscillator is:

$$\frac{d^2x}{dt^2} + \omega^2 x = 0 \tag{17.1}$$

where $\omega = \sqrt{k/m}$ is the angular frequency, with k being the spring constant and m the mass. In terms of Newton's second law, this is:

$$m\frac{d^2x}{dt^2} = -kx \tag{17.2}$$

This second-order differential equation requires two initial conditions: - Initial position: $x(t = 0) = x_0$ - Initial velocity: $\dot{x}(t = 0) = v_0$
The analytical solution is:

$$x(t) = x_0 \cos(\omega t) + \frac{v_0}{\omega}\sin(\omega t) \tag{17.3}$$

This represents sinusoidal oscillation with a constant amplitude and period $T = 2\pi/\omega$. The total energy of the system (kinetic + potential) remains constant:

$$E = \frac{1}{2}mv^2 + \frac{1}{2}kx^2 \tag{17.4}$$

This energy conservation will be an important test for our numerical methods.

17.4 Implicit Matrix Solution

Using the matrix representation of the second derivative operator from our previous lecture, we can transform the ODE into a linear system that can be solved in one step. This approach treats the entire time evolution as a single mathematical problem.

17.4.1 Setting Up the System

From calculus, we know that the second derivative represents the rate of change of the rate of change. Physically, this corresponds to acceleration in mechanics. In numerical terms, we need to approximate this using finite differences.

Recall that we can represent the second derivative operator as a tridiagonal matrix:

$$D_2 = \frac{1}{\Delta t^2}\begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -2 & 1 \\ 0 & \cdots & 0 & 0 & 1 & -2 \end{bmatrix}$$

This matrix implements the central difference approximation for the second derivative:

$$\frac{d^2x}{dt^2} \approx \frac{x_{i+1} - 2x_i + x_{i-1}}{\Delta t^2} \tag{17.5}$$

Each row in this matrix corresponds to the equation for a specific time point, linking it to its neighbors.

The harmonic oscillator term $\omega^2 x$ represents the restoring force. In a spring system, this is $F = -kx$ divided by mass, giving $a = -\frac{k}{m}x = -\omega^2 x$. This can be represented by a diagonal matrix:

$$V = \omega^2 I = \omega^2 \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

Our equation $\frac{d^2x}{dt^2} + \omega^2 x = 0$ becomes the matrix equation $(D_2 + V)x = 0$, where x is the vector containing the positions at all time points $(x_1, x_2, ..., x_n)$. This is a large system of linear equations that determine the entire trajectory at once.

17.4.2 Incorporating Initial Conditions

To solve this system, we need to incorporate the initial conditions by modifying the first rows of our matrix. This is a crucial step because a second-order differential equation needs two initial conditions to define a unique solution. In physical terms, we need to know both the initial position (displacement) and the initial velocity of our oscillator. We incorporate these conditions directly into our matrix equation:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 1 & 0 & 0 & \cdots & 0 \\ \frac{1}{\Delta t^2} & -\frac{2}{\Delta t^2} + \omega^2 & \frac{1}{\Delta t^2} & 0 & \cdots & 0 \\ 0 & \frac{1}{\Delta t^2} & -\frac{2}{\Delta t^2} + \omega^2 & \frac{1}{\Delta t^2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \frac{1}{\Delta t^2} & -\frac{2}{\Delta t^2} + \omega^2 & \frac{1}{\Delta t^2} \end{bmatrix}$$

The first row (in red) enforces $x(0) = x_0$ (initial position), and the second row (in blue) implements the initial velocity condition. The rest of the matrix represents the differential equation at each time step. By incorporating the initial conditions directly into the matrix, we ensure that our solution satisfies both the differential equation and the initial conditions.

```
#| autorun: false

# Define parameters
k = 15.5  # spring constant (N/m)
m = 0.2   # mass (kg)
x0 = 1    # initial elongation (m)
v0 = 0    # initial velocity (m/s)
omega = np.sqrt(k/m)  # angular frequency (rad/s)
                      # This comes from the physics:  ² = k/m

# Set up time domain
T = 2*np.pi/omega  # natural period of oscillation
L = 3*T            # solve for 3 complete periods
N = 500            # number of data points (time resolution)
t = np.linspace(0, L, N)  # time axis
dt = t[1] - t[0]  # time interval of each step
```

```
# Construct the matrix M that represents (D2 + V)
# This combines the second derivative operator and the potential
M = (diags([-2., 1., 1.], [-1,-2, 0], shape=(N, N))+diags([1], [-1], shape=(N, N))* omega**2*dt**2).tod

# Incorporate initial conditions
M[0,0]=1  # First equation: x(0) = x0 (initial position)
M[1,0]=-1  # Second equation: velocity condition
M[1,1]=1   # These two lines encode v(0) = v0

# Create the right-hand side vector containing the initial conditions
b = np.zeros(N)  # initialize vector of zeros
b[0]=1  # Set initial position (x0 = 1)
b[1]=0  # Set initial velocity (v0 = 0)
b = b.transpose()

# Solve the linear system Mx = b
# This gives us the position at all time points at once
x = np.linalg.solve(M, b)  # this is the solution
```

```
#| autorun: false

# Plot the solution
plt.figure(figsize=get_size(12, 10))
plt.plot(t, x, label='Numerical Solution')
plt.plot(t, x0*np.cos(omega*t) + v0/omega*np.sin(omega*t), '--', label='Analytical Solution')
plt.xlabel('Time (s)')
plt.ylabel('Position x(t)')
plt.legend()
plt.show()
```

This matrix-based approach has several advantages: - It solves for all time points simultaneously, giving the entire trajectory in one operation - It can be very stable for certain types of problems, particularly “stiff” equations - It handles boundary conditions naturally by incorporating them into the matrix structure - It often provides good energy conservation for oscillatory systems

However, it also has limitations: - It requires solving a large linear system, which becomes computationally intensive for long simulations - It’s not suitable for nonlinear problems without modification (e.g., a pendulum with large angles where $\sin\theta \neq \theta$) - Memory requirements grow with the time span (an N×N matrix for N time steps) - Implementing time-dependent forces is more complex than with explicit methods

From a physics perspective, this approach is similar to finding stationary states in quantum mechanics or solving boundary value problems in electrostatics—we’re solving the entire system at once rather than stepping through time.

17.5 Explicit Numerical Integration Methods

An alternative approach is to use explicit step-by-step integration methods. These methods convert the second-order ODE to a system of first-order ODEs and then advance the solution incrementally, similar to how we intuitively understand physical motion: position changes based on velocity, and velocity changes based on acceleration.

17.5.1 Converting to a First-Order System

To convert our second-order ODE to a first-order system, we introduce a new variable $v = dx/dt$ (the velocity):

$$\frac{dx}{dt} = v \tag{17.6}$$
$$\frac{dv}{dt} = -\omega^2 x \tag{17.7}$$

This transformation is common in physics. For example, in classical mechanics, we often convert Newton's second law (a second-order ODE) into phase space equations involving position and momentum. In the case of the harmonic oscillator:

1. The first equation simply states that the rate of change of position is the velocity
2. The second equation comes from $F = ma$ where $F = -kx$, giving $a = \frac{F}{m} = -\frac{k}{m}x = -\omega^2 x$

We can now represent the state of our system as a vector $\vec{y} = [x, v]^T$ and the derivative as $\dot{\vec{y}} = [v, -\omega^2 x]^T$. This representation is called the "phase space" description, and is fundamental in classical mechanics and dynamical systems theory.

17.5.2 Euler Method

The simplest explicit integration method is the Euler method, derived from the first-order Taylor expansion:

$$\vec{y}(t + \Delta t) \approx \vec{y}(t) + \dot{\vec{y}}(t)\Delta t \tag{17.8}$$

This is essentially a linear approximation—assuming the derivative stays constant over the small time step. Physically, it's like assuming constant velocity when updating position, and constant acceleration when updating velocity over each small time interval.

For the harmonic oscillator, this becomes:

$$x_{n+1} = x_n + v_n \Delta t \tag{17.9}$$
$$v_{n+1} = v_n - \omega^2 x_n \Delta t \tag{17.10}$$

The physical interpretation is straightforward: 1. The new position equals the old position plus the displacement (velocity × time) 2. The new velocity equals the old velocity plus the acceleration (- ^{2}x) multiplied by the time step

In practice, the Euler method will cause the total energy of a harmonic oscillator to increase over time, which is physically incorrect. This is because the method doesn't account for the continuous change in acceleration during the time step.

```
#| autorun: false

def euler_method(f, y0, t_span, dt):
    """
    Implements the Euler method for solving ODEs.

    Parameters:
    f: Function that returns the derivative dy/dt = f(t, y)
    y0: Initial condition vector [position, velocity]
    t_span: (t_start, t_end)
    dt: Time step

    Returns:
    t: Array of time points
    y: Array of solution values
    """
    t_start, t_end = t_span
    n_steps = int((t_end - t_start) / dt) + 1
    t = np.linspace(t_start, t_end, n_steps)
    y = np.zeros((n_steps, len(y0)))
    y[0] = y0

    for i in range(1, n_steps):
        y[i] = y[i-1] + dt * f(t[i-1], y[i-1])

    return t, y
```

17.5.3 Euler-Cromer Method

The Euler method often performs poorly for oscillatory systems, as it tends to artificially increase the energy over time. A simple but effective improvement is the Euler-Cromer method (also known as the symplectic Euler method), which uses the updated velocity for the position update:

$$v_{n+1} = v_n - \omega^2 x_n \Delta t \tag{17.11}$$
$$x_{n+1} = x_n + v_{n+1} \Delta t \tag{17.12}$$

Notice the subtle but crucial difference: we use v_{n+1} (the newly calculated velocity) to update the position, rather than v_n.

This small change dramatically improves energy conservation for oscillatory systems. From a physics perspective, this method better preserves the structure of Hamiltonian systems, which include the harmonic oscillator, planetary motion, and many other important physical systems.

The Euler-Cromer method is especially valuable in physics simulations where long-term energy conservation is important, such as: - Orbital mechanics - Molecular dynamics - Plasma physics - N-body simulations

While not perfect, this simple modification makes the method much more useful for real physical systems with oscillatory behavior.

```
#| autorun: false

def euler_cromer_method(f, y0, t_span, dt):
    """
    Implements the Euler-Cromer method for oscillatory systems.

    Parameters:
    f: Function that returns the derivative dy/dt = f(t, y)
    y0: Initial condition vector [position, velocity]
    t_span: (t_start, t_end)
    dt: Time step

    Returns:
    t: Array of time points
    y: Array of solution values
    """
    t_start, t_end = t_span
    n_steps = int((t_end - t_start) / dt) + 1
    t = np.linspace(t_start, t_end, n_steps)
    y = np.zeros((n_steps, len(y0)))
    y[0] = y0

    for i in range(1, n_steps):
        # Calculate derivatives
        derivatives = f(t[i-1], y[i-1])

        # Update velocity first
        y[i, 1] = y[i-1, 1] + dt * derivatives[1]

        # Then update position using the updated velocity
        y[i, 0] = y[i-1, 0] + dt * y[i, 1]

    return t, y
```

17.5.4 Midpoint Method

For higher accuracy, we can use the midpoint method (also known as the second-order Runge-Kutta method or RK2). This evaluates the derivative at the midpoint of the interval, providing a better approximation of the average derivative over the time step.

$$k_1 = f(t_n, y_n) \tag{17.13}$$

$$k_2 = f(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} k_1) \tag{17.14}$$

$$y_{n+1} = y_n + \Delta t \cdot k_2 \tag{17.15}$$

The physical intuition behind this method is: 1. First, calculate the initial derivative k_1 (representing the initial rates of change) 2. Use this derivative to estimate the state at the middle of the time step 3. Calculate a new derivative k_2 at this midpoint 4. Use the midpoint derivative to advance the full step

This is analogous to finding the average velocity by looking at the velocity in the middle of a time interval, rather than just at the beginning. In physical problems with continuously varying forces, this provides a much better approximation than the Euler method.

The midpoint method achieves $O(\Delta t^2)$ accuracy, meaning the error decreases with the square of the step size. This makes it much more accurate than Euler's method for the same computational cost.

```
#| autorun: false

def midpoint_method(f, y0, t_span, dt):
    """
    Implements the midpoint method (RK2) for solving ODEs.

    Parameters:
    f: Function that returns the derivative dy/dt = f(t, y)
    y0: Initial condition vector [position, velocity]
    t_span: (t_start, t_end)
    dt: Time step

    Returns:
    t: Array of time points
    y: Array of solution values
    """
    t_start, t_end = t_span
    n_steps = int((t_end - t_start) / dt) + 1
    t = np.linspace(t_start, t_end, n_steps)
    y = np.zeros((n_steps, len(y0)))
    y[0] = y0
    for i in range(1, n_steps):
        k1 = f(t[i-1], y[i-1])
        k2 = f(t[i-1] + dt/2, y[i-1] + dt/2 * k1)
        y[i] = y[i-1] + dt * k2

    return t, y
```

17.5.5 Comparing the Methods

Let's compare these methods for the harmonic oscillator:

```
#| autorun: false

# Define the harmonic oscillator system
def harmonic_oscillator(t, y, omega=2.0):
    """
    Harmonic oscillator as a system of first-order ODEs.

    Parameters:
    t: time (not used explicitly, but required for compatibility with ODE solvers)
    y: state vector where y[0] is position x and y[1] is velocity v
    omega: angular frequency = sqrt(k/m) where k is spring constant and m is mass

    Returns:
    dydt: derivatives [dx/dt, dv/dt]

    Physics background:
    This implements the coupled differential equations:
    dx/dt = v
    dv/dt = - ²x  (from F = ma where F = -kx)
    """
    dydt = np.zeros_like(y)
    dydt[0] = y[1]                # dx/dt = v
    dydt[1] = -omega**2 * y[0]    # dv/dt = - ²x
    return dydt

# Analytical solution
def analytical_solution(t, x0, v0, omega):
    return x0 * np.cos(omega * t) + v0/omega * np.sin(omega * t)

# Parameters
omega = 2.0
x0 = 1.0
v0 = 0.0
y0 = np.array([x0, v0])
t_span = (0, 10)
dt = 0.1

# Solve using different methods
t_euler, y_euler = euler_method(lambda t, y: harmonic_oscillator(t, y, omega), y0, t_span, dt)
t_cromer, y_cromer = euler_cromer_method(lambda t, y: harmonic_oscillator(t, y, omega), y0, t_span, dt)
t_midpoint, y_midpoint = midpoint_method(lambda t, y: harmonic_oscillator(t, y, omega), y0, t_span, dt)

# Calculate analytical solution
y_analytical = analytical_solution(t_euler, x0, v0, omega)
```

```
#| autorun: false

# Plot comparisons
plt.figure(figsize=get_size(12,10))

# Position comparison
plt.plot(t_euler, y_euler[:, 0], label='Euler')
plt.plot(t_cromer, y_cromer[:, 0], label='Euler-Cromer')
plt.plot(t_midpoint, y_midpoint[:, 0], label='Midpoint')
plt.plot(t_euler, y_analytical, '--', label='Analytical')
plt.xlabel('Time (s)')
plt.ylabel('Position x(t)')

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

```
#| autorun: false

# Energy calculation
def calculate_energy(y, omega):
    """
    Calculate the total energy of the harmonic oscillator.

    Parameters:
    y: state array with positions y[:,0] and velocities y[:,1]
    omega: angular frequency = sqrt(k/m)

    Returns:
    E: total energy at each time point

    Physics:
    E = KE + PE = (1/2)mv² + (1/2)kx²
      = (1/2)v² + (1/2) ²x²  (assuming m=1)

    This energy should be conserved in a perfect harmonic oscillator system.
    The accuracy of energy conservation is a good measure of the
    quality of a numerical method.
    """
    kinetic = 0.5 * y[:, 1]**2       # KE = (1/2)mv² (assuming m=1)
    potential = 0.5 * omega**2 * y[:, 0]**2  # PE = (1/2)kx² = (1/2) ²x²
    return kinetic + potential

# Energy comparison
E_euler = calculate_energy(y_euler, omega)
E_cromer = calculate_energy(y_cromer, omega)
E_midpoint = calculate_energy(y_midpoint, omega)
E_analytical = 0.5 * omega**2 * x0**2  # Constant for this initial condition

plt.figure(figsize=get_size(12,10))
plt.plot(t_euler, E_euler/E_analytical - 1, label='Euler')
plt.plot(t_cromer, E_cromer/E_analytical - 1, label='Euler-Cromer')
plt.plot(t_midpoint, E_midpoint/E_analytical - 1, label='Midpoint')
plt.axhline(y=0, color='k', linestyle='--', alpha=0.3)
plt.xlabel('Time (s)')
plt.ylabel('Relative Energy Error')
plt.legend()


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

17.5.6 Observations:

- **Euler Method**: Simple but least accurate. It systematically increases the total energy of the system, causing the amplitude to grow over time.
- **Euler-Cromer Method**: Much better energy conservation for oscillatory systems with minimal additional computation.
- **Midpoint Method**: Higher accuracy and better energy conservation.

The choice of method depends on the specific requirements of your problem: - Use Euler for simplicity when accuracy is not critical - Use Euler-Cromer for oscillatory systems when computational efficiency is important - Use Midpoint or higher-order methods when accuracy is crucial

17.6 Advanced Methods with SciPy

For practical applications, SciPy provides sophisticated ODE solvers with adaptive step size control, error estimation, and specialized algorithms for different types of problems. These methods represent the state-of-the-art in numerical integration and are what working physicists typically use for research and advanced applications.

17.6.1 What Makes These Methods Advanced?

1. **Adaptive Step Size**: Unlike our fixed-step methods, these solvers can automatically adjust the step size based on the local error estimate, taking smaller steps where the solution changes rapidly and larger steps where it's smooth.

2. **Higher-Order Methods**: These solvers use higher-order approximations (4th, 5th, or even 8th order), achieving much higher accuracy with the same computational effort.

3. **Error Control**: They can maintain the error below a specified tolerance, giving you confidence in the accuracy of your results.

4. **Specialized Methods**: Different methods are optimized for different types of problems (stiff vs. non-stiff, conservative vs. dissipative).

17.6.2 Using solve_ivp for the Harmonic Oscillator

```
#| autorun: false

from scipy.integrate import solve_ivp

# Define the ODE system
def SHO(t, y, omega=2.0):
    """
    Simple Harmonic Oscillator

    This function defines the right-hand side of the ODE system:
    dx/dt = v
    dv/dt = - ²x
    """
    x, v = y   # Unpack the state vector: position and velocity
    dxdt = v   # Rate of change of position equals velocity
    dvdt = -omega**2 * x   # Rate of change of velocity equals acceleration
    return [dxdt, dvdt]

# Parameters
t_span = (0, 10)  # Time interval to solve over
y0 = [1.0, 0.0]   # Initial conditions: [position, velocity]
omega = 2.0       # Angular frequency (rad/s)

# Create evaluation points (for plotting)
t_eval = np.linspace(t_span[0], t_span[1], 200)

# Solve using different methods

# RK45: Explicit Runge-Kutta method of order 5(4)
# This is the default method, good for non-stiff problems
# It uses a 5th-order method with 4th-order error control
solution_RK45 = solve_ivp(
    lambda t, y: SHO(t, y, omega),
    t_span,
    y0,
    method='RK45',
    t_eval=t_eval
)

# BDF: Backward Differentiation Formula
# This is an implicit method designed for stiff problems
# (problems where certain components evolve on much faster timescales than others)
solution_BDF = solve_ivp(
    lambda t, y: SHO(t, y, omega),
    t_span,
    y0,
    method='BDF',
    t_eval=t_eval
)

# DOP853: Explicit Runge-Kutta method of order 8(5,3)
# This is a high-order method for high-precision requirements
# It uses an 8th-order formula for integration with 5th-order error estimation
solution_DOP853 = solve_ivp(
    lambda t, y: SHO(t, y, omega),
    t_span,
    y0,
    method='DOP853',  # Explicit Runge-Kutta method of order 8
    t_eval=t_eval
)

# Plot solutions
plt.figure(figsize=get_size(12, 10))

# Position comparison

plt.plot(solution_RK45.t, solution_RK45.y[0], label='RK45')
plt.plot(solution_BDF.t, solution_BDF.y[0], '--', label='BDF')
plt.plot(solution_DOP853.t, solution_DOP853.y[0], '-.', label='DOP853')
plt.plot(t_eval, analytical_solution(t_eval, y0[0], y0[1], omega), ':', label='Analytical')
plt.xlabel('Time (s)')
plt.ylabel('Position x(t)')
plt.legend()
plt.tight_layout()
plt.show()
```

```
# Phase space plot
plt.figure(figsize=get_size(12, 10))
plt.plot(solution_RK45.y[0], solution_RK45.y[1], label='Phase Space Trajectory')
plt.xlabel('Position x')
plt.ylabel('Velocity v')
plt.axis('equal')

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

17.7 Advanced Example: Damped Driven Pendulum

Let's apply these methods to a more complex system: a damped driven pendulum. This system can exhibit chaotic behavior under certain conditions, making it a fascinating study in nonlinear dynamics.

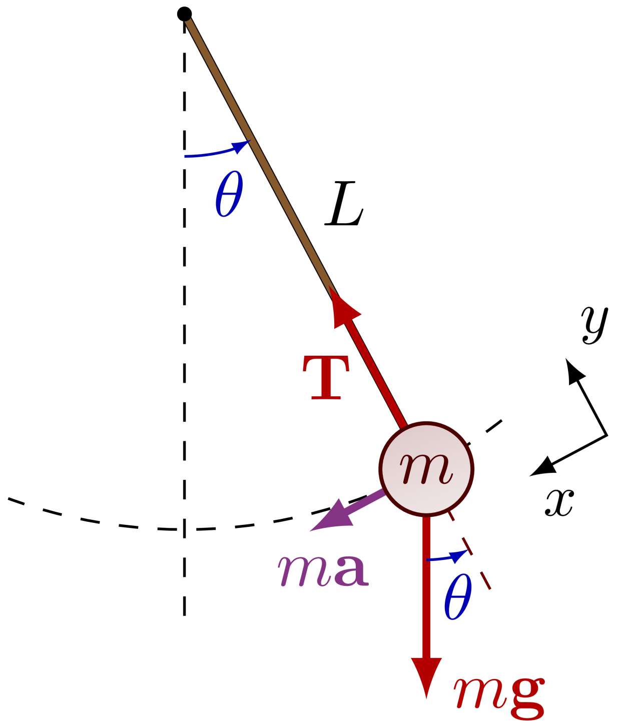


Figure 17.2: The damped driven pendulum. External driving forces and damping (friction) create a system that can exhibit complex behaviors from regular oscillations to chaos.

17.7.1 Physical Context

The damped driven pendulum represents many real physical systems

- A physical pendulum with friction and external driving force
- An RLC circuit with nonlinear components
- Josephson junctions in superconductivity

- Certain types of mechanical and electrical oscillators

The equation of motion is:

$$\frac{d^2\theta}{dt^2} + b\frac{d\theta}{dt} + \omega_0^2 \sin\theta = F_0 \cos(\omega_d t) \tag{17.16}$$

where

- θ is the angle from vertical (radians)
- b is the damping coefficient (represents friction or resistance)
- $\omega_0 = \sqrt{g/L}$ is the natural frequency (where g is gravity and L is pendulum length)
- F_0 is the driving amplitude (strength of the external force)
- ω_d is the driving frequency (how rapidly the external force oscillates)

This equation differs from the harmonic oscillator in three crucial ways:

1. The $\sin\theta$ term (rather than just θ) makes it nonlinear
2. The damping term $b\frac{d\theta}{dt}$ causes energy dissipation
3. The driving term $F_0\cos(\omega_d t)$ adds energy to the system

These additions make the system much more realistic and rich in behavior. Students could explore this system by varying parameters like the damping coefficient (b), driving amplitude (F0), and driving frequency (omega_d) to observe how the pendulum transitions between regular oscillations, period doubling, and chaos.

```
#| autorun: false

def damped_driven_pendulum(t, y, b=0.1, omega0=1.0, F0=0.5, omega_d=0.7):
    """
    Damped driven pendulum ODE system.

    Parameters:
    t: time (seconds)
    y: state vector where y[0] is theta (angle) and y[1] is omega (angular velocity)
    b: damping coefficient (1/second)
    omega0: natural frequency (rad/second) = sqrt(g/L)
    F0: driving amplitude (rad/second²)
    omega_d: driving frequency (rad/second)

    Returns:
    [dtheta_dt, domega_dt]: derivatives of state variables

    Physics:
    This system models a pendulum with:
    - Gravitational restoration torque: -omega0² * sin(theta)
    - Damping (friction) torque: -b * omega
    - External driving torque: F0 * cos(omega_d * t)

    The complete derivation comes from the torque equation:
      = I*  = -mgL*sin( ) - b'*  +  _external

    Dividing by I (moment of inertia) gives this equation.
    """
    theta, omega = y  # Unpack the state vector
    dtheta_dt = omega  # Rate of change of angle equals angular velocity

    # Rate of change of angular velocity equals angular acceleration:
    # (Restoration) + (Damping) + (Driving)
    domega_dt = -omega0**2 * np.sin(theta) - b*omega + F0 * np.cos(omega_d * t)
    return [dtheta_dt, domega_dt]

# Parameters
t_span = (0, 150)     # Time span (seconds)
y0 = [0.1, 0.0]      # Initial conditions: [angle (rad), angular velocity (rad/s)]
b = 0.1              # Damping coefficient (friction) (1/s)
omega0 = 1.0         # Natural frequency = sqrt(g/L) (rad/s)
                     # (equivalent to a pendulum of length L = g/omega0²   9.81 m if omega0 = 1)
F0 = 0.1             # Driving amplitude (rad/s²)
omega_d = 0.7        # Driving frequency (rad/s)
                     # (driving period = 2 /omega_d   8.97 s)

# Solve the ODE system using solve_ivp
# We use RK45 which is well-suited for this type of problem
solution = solve_ivp(
    lambda t, y: damped_driven_pendulum(t, y, b, omega0, F0, omega_d),
    t_span,
    y0,
    method='RK45',
    t_eval=np.linspace(t_span[0], t_span[1], 1000),  # Points for evaluation
    rtol=1e-6,  # Relative tolerance - controls accuracy
    atol=1e-9   # Absolute tolerance - especially important for values near zero
)

# Plot solution
plt.figure(figsize=get_size(12, 10))

# Time series
plt.plot(solution.t, solution.y[0], label='Angle  ')
plt.plot(solution.t, solution.y[1], label='Angular Velocity  ')
plt.xlabel('Time (s)')
plt.ylabel('Value')
plt.legend()
plt.tight_layout()
plt.show()
```

```
#| autorun: false
# Phase space
plt.figure(figsize=get_size(12, 10))
plt.plot(solution.y[0] % (2*np.pi), solution.y[1])
plt.xlabel('Angle   (mod 2 )')
plt.ylabel('Angular Velocity  ')


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

17.7.2 Exploring Chaotic Behavior

Chaotic motion refers to a deterministic system whose behavior appears random and exhibits extreme sensitivity to initial conditions—the famous "butterfly effect." In the damped driven pendulum, chaos emerges when the system's nonlinear restoring force ($\sin\theta$ term) interacts with sufficient driving force. For this system, chaotic behavior typically appears when the driving amplitude F_0 exceeds a critical value (approximately 1.0 in our case) while maintaining moderate damping ($b \approx 0.1$) and a driving frequency ω_d that is not too far from the natural frequency ω_0. In the chaotic regime, the pendulum's trajectory becomes unpredictable over long time scales, despite being governed by deterministic equations. The phase space transforms from closed orbits (regular motion) to a strange attractor with fractal structure, and the system never settles into a periodic oscillation.

Lyapunov Exponents: Quantifying Chaos

The Lyapunov exponent is a powerful mathematical tool for characterizing chaotic systems. It measures the rate at which nearby trajectories in phase space diverge over time. For a system with state vector $\vec{x}$, if two initial conditions differ by a small displacement $\delta\vec{x}_0$, then after time t, this separation grows approximately as:

$$|\delta\vec{x}(t)| \approx e^{\lambda t}|\delta\vec{x}_0|$$

where λ is the Lyapunov exponent.

- A **positive** Lyapunov exponent ($\lambda > 0$) indicates chaos: nearby trajectories diverge exponentially, making long-term prediction impossible. The larger the exponent, the more chaotic the system.
- A **zero** Lyapunov exponent ($\lambda = 0$) suggests a stable limit cycle or quasiperiodic behavior.
- A **negative** Lyapunov exponent ($\lambda < 0$) indicates a stable fixed point, where trajectories converge.

For our damped driven pendulum, we can numerically estimate the Lyapunov exponent by comparing how two slightly different initial conditions evolve over time. For instance, we might begin with two pendulums at nearly the same angle—perhaps (0) = 0.1 and (0) = 0.1001—while keeping their initial angular velocities identical at (0) = (0) = 0. As we simulate both systems, we can track the separation between their trajectories in phase space. This separation represents the difference vector x(t) = [(t) - (t), (t) - (t)]. Initially, this difference is very small, but in a chaotic system, it grows exponentially with time. By measuring this growth rate, we can compute the Lyapunov exponent as   (1/t)ln(| x(t)|/| x(0)|). When examining our pendulum system, which has two dimensions (and), we actually have two Lyapunov exponents forming a spectrum. If either of these exponents is positive, we can conclusively determine that our system exhibits chaotic behavior, indicating fundamental unpredictability despite its deterministic nature.

```python
#| autorun: false

# Function to solve and plot for different driving amplitudes
def solve_for_driving_amplitude(F0):
    """
    Solves the pendulum equation for a given driving amplitude F0.

    Physics note:
    - For small F0, the system behaves like a damped oscillator with a periodic response
    - As F0 increases, the system can undergo period-doubling bifurcations
    - Above a critical value, the system becomes chaotic

    We skip the first 50 seconds (the "transient" behavior) to focus on the
    long-term behavior (the "attractor").
    """
    solution = solve_ivp(
        lambda t, y: damped_driven_pendulum(t, y, b=0.1, omega0=1.0, F0=F0, omega_d=0.7),
        (0, 100),  # Longer time span to observe chaos
        [0.1, 0.0],
        method='RK45',
        t_eval=np.linspace(50, 100, 1000)  # Only plot after transients die out
    )
    return solution

# Try different driving amplitudes to observe the transition to chaos
F0_values = [0.1, 1.0, 1.2]
plt.figure(figsize=get_size(15, 10))
```

```
behavior_types = ["Regular", "Period-Doubled", "Chaotic"]

for i, F0 in enumerate(F0_values):
    solution = solve_for_driving_amplitude(F0)

    # Time series plot
    plt.subplot(2, 3, i+1)
    plt.plot(solution.t, solution.y[0])
    plt.xlabel('Time (s)')
    plt.ylabel('Angle  ')

    # Phase space plot
    plt.subplot(2, 3, i+4)
    plt.plot(solution.y[0] % (2*np.pi), solution.y[1], '.', markersize=1)
    plt.xlabel('Angle   (mod 2 )')
    plt.ylabel('Angular Velocity  ')

    # Add explanatory text for each type of behavior
    #if i == 0:
        #plt.text(1, 0, "Simple attractor\n(periodic motion)", fontsize=9)
    #elif i == 1:
        #plt.text(1, 0, "Period doubling\n(quasi-periodic)", fontsize=9)
    #else:
        #plt.text(1, 0, "Strange attractor\n(chaotic motion)", fontsize=9)

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

17.8 Conclusion

In this lecture, we've built upon our knowledge of numerical differentiation to solve ordinary differential equations. We've explored:

1. **Implicit Matrix Methods**: Using matrices to solve the entire system at once
2. **Explicit Integration Methods**: Step-by-step methods like Euler, Euler-Cromer, and Midpoint
3. **Advanced SciPy Methods**: Leveraging powerful adaptive solvers for complex problems

Each approach has its strengths and is suited to different types of problems. The matrix method is excellent for linear systems, while explicit methods are more versatile for nonlinear problems. SciPy's solvers combine accuracy, stability, and efficiency for practical applications.

17.8.1 Physical Relevance

The methods we've learned are essential tools for computational physics because they: 1. Allow us to study systems with no analytical solutions 2. Provide insights into nonlinear dynamics and chaos 3. Enable the modeling of realistic systems with friction, driving forces, and complex interactions 4. Connect mathematical formulations with observable physical phenomena

As a second-semester physics student, these numerical tools complement your analytical understanding of mechanics, electromagnetism, and other core physics subjects. When analytical methods reach their limits, these numerical approaches allow you to continue exploring and modeling physical reality.

17.9 Exercises

Here are two simple exercises with solutions where you can train your programming skills. In these examples you should use the `odeint` method from SciPy's `integrate` module.

Self-Exercise 1: Simple Harmonic Oscillator

Write a program to solve the equation of motion for a simple harmonic oscillator. This example demonstrates how to solve a second-order differential equation using scipy's `odeint`.
The equation of motion is: $\frac{d^2x}{dt^2} + \omega^2 x = 0$
This represents an idealized spring-mass system or pendulum with small oscillations.

```
#| exercise: ex_1

import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def oscillator(state, t, omega):
    # your code here
    # ypir code here

# Set parameters

# Solve ODE and plot solution
----
```

Use `odeint(oscillator, initial_state, t, args=(omega,))` to solve the system. The solution will have two columns: position ([:,0]) and velocity ([:,1]). Create a plot showing position vs time using matplotlib. Remember to label your axes and add a title.
Solution.

Solution

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def oscillator(state, t, omega):
    x, v = state
    return [v, -omega**2 * x]

omega = 2.0
t = np.linspace(0, 10, 1000)
initial_state = [1, 0]

solution = odeint(oscillator, initial_state, t, args=(omega,))

plt.figure(figsize=(10, 4))
plt.plot(t, solution[:, 0])
plt.xlabel('Time (s)')
plt.ylabel('Position (m)')
plt.title('Simple Harmonic Motion')
plt.grid(True)
plt.show()
```

Self-Exercise 2: Damped Driven Harmonic Oscillator using Matrix Method

Model a damped harmonic oscillator with an external driving force using the implicit matrix method. This exercise demonstrates how to solve a second-order differential equation by constructing and solving a matrix system that represents the entire time evolution.
The equation of motion is: $m\frac{d^2x}{dt^2} + b\frac{dx}{dt} + kx = F_0\cos(\omega t)$
Where m is mass, b is damping coefficient, k is spring constant, F_0 is driving amplitude, and ω is driving frequency.

```
#| exercise: ex_2

import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import diags

# Define system parameters
m = 1.0      # mass (kg)
k = 16.0     # spring constant (N/m)
b = 0.5      # damping coefficient (kg/s)
F0 = 1.0     # driving force amplitude (N)
omega_d = 4.0  # driving frequency (rad/s)
omega0 = np.sqrt(k/m)  # natural frequency

# Setup time domain
T = 2*np.pi/omega0  # natural period
L = 10*T           # solve for 10 complete periods
N = 1000           # number of data points
t = np.linspace(0, L, N)  # time axis
dt = t[1] - t[0]   # time step

# Set initial conditions
x0 = 1.0    # initial position
v0 = 0.0    # initial velocity

# Step 1: Construct the matrix representing the left side of the equation
# This needs to include:
# - Second derivative operator (D2)
# - First derivative operator for damping (D1)
# - Potential term (diagonal with k/m)
# - Initial conditions

# Your code here

# Step 2: Define the right side of the equation with the driving force
# Your code here

# Step 3: Solve the system
# Your code here

# Step 4: Plot the solution
# Your code here
```

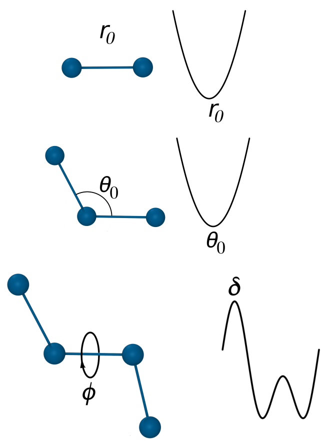
First, construct a tridiagonal matrix for the second derivative operator D_2 with elements $\frac{1}{\Delta t^2}[1, -2, 1]$ along the diagonals.
For the first derivative (needed for damping), use a central difference approximation: $\frac{dx}{dt} \approx \frac{x_{i+1} - x_{i-1}}{2\Delta t}$, which gives a matrix D_1 with elements $\frac{1}{2\Delta t}[-1, 0, 1]$ along the diagonals.

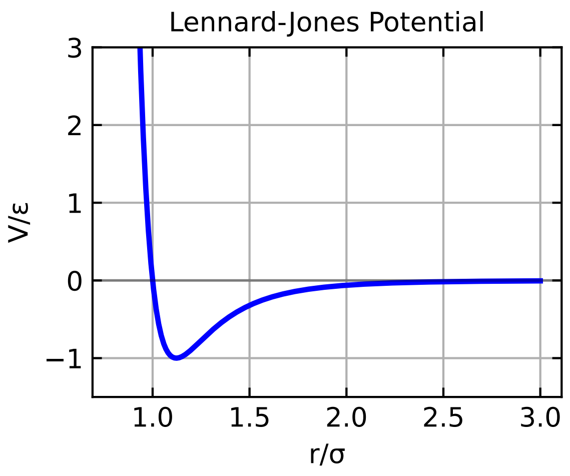
The full matrix equation is $(D_2 + \frac{b}{m}D_1 + \frac{k}{m}I)x = \frac{F_0}{m}\cos(\omega t)$
To incorporate initial conditions, modify the first two rows of your matrix and the corresponding entries in your right-hand side vector: - First row: $x_0 = x(0)$ (initial position) - Second row: Approximation for initial velocity using forward difference
After solving, compare your solution with the analytical solution for the steady-state response of a driven harmonic oscillator.
Solution.

Solution

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import diags

# Define system parameters
m = 1.0      # mass (kg)
k = 16.0     # spring constant (N/m)
b = 0.5      # damping coefficient (kg/s)
F0 = 1.0     # driving force amplitude (N)
omega_d = 4.0  # driving frequency (rad/s)
omega0 = np.sqrt(k/m)  # natural frequency

# Setup time domain
T = 2*np.pi/omega0  # natural period
L = 10*T            # solve for 10 complete periods
N = 1000            # number of data points
t = np.linspace(0, L, N)  # time axis
dt = t[1] - t[0]    # time step

# Set initial conditions
x0 = 1.0    # initial position
v0 = 0.0    # initial velocity

# Step 1: Construct the matrix for the left side of the equation
# Second derivative operator D2 (tridiagonal with [1, -2, 1]/dt²)
diag_vals = np.ones(N) * (-2.0/(dt**2))
upper_diag = np.ones(N-1) * (1.0/(dt**2))
lower_diag = np.ones(N-1) * (1.0/(dt**2))
D2 = diags([diag_vals, upper_diag, lower_diag], [0, 1, -1], shape=(N, N)).toarray()

# First derivative operator D1 for damping (tridiagonal with [-1, 0, 1]/(2dt))
diag_vals_d1 = np.zeros(N)
upper_diag_d1 = np.ones(N-1) * (1.0/(2*dt))
lower_diag_d1 = np.ones(N-1) * (-1.0/(2*dt))
D1 = diags([diag_vals_d1, upper_diag_d1, lower_diag_d1], [0, 1, -1], shape=(N, N)).toarray()

# Create the coefficient matrix: M = m*D2 + b*D1 + k*I
I = np.eye(N)  # Identity matrix
M = m*D2 + b*D1 + k*I

# Incorporate initial conditions (modify the first two rows)
M[0, :] = 0  # Clear first row for initial position
M[0, 0] = 1  # Set x_0 = x0

M[1, :] = 0  # Clear second row for initial velocity
M[1, 0] = -1  # Simple forward difference to encode initial velocity
M[1, 1] = 1
M[1, 2] = 0

# Step 2: Define the right side of the equation with the driving force
b_vector = np.zeros(N)
b_vector[2:] = F0 * np.cos(omega_d * t[2:])  # Driving force (skip first two points for initial

# Set initial conditions in the right-hand side vector
b_vector[0] = x0  # Initial position
b_vector[1] = v0 * dt  # Initial velocity (scaled by dt for the difference approximation)

# Step 3: Solve the system
x = np.linalg.solve(M, b_vector)

# Analytical steady-state solution for comparison
phase_angle = np.arctan2(b*omega_d, (k-m*omega_d**2))
amplitude = F0 / np.sqrt((k-m*omega_d**2)**2 + (b*omega_d)**2)
```

-
-
- scipy.integrate.solve_ivp
-
-
-
-
-
- Scipy Lecture Notes
- Matplotlib Gallery


r_0
r_0
θ_0
θ_0
δ
ϕ

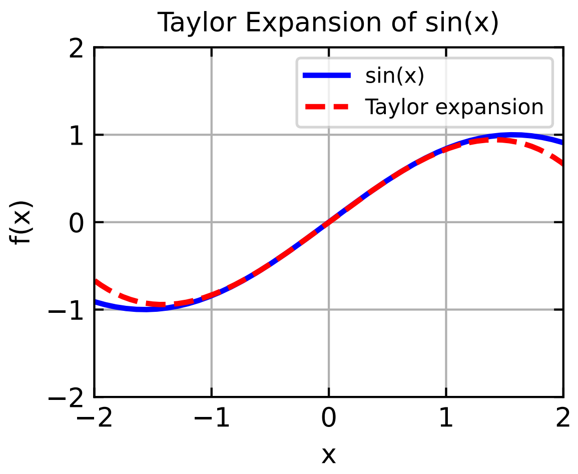


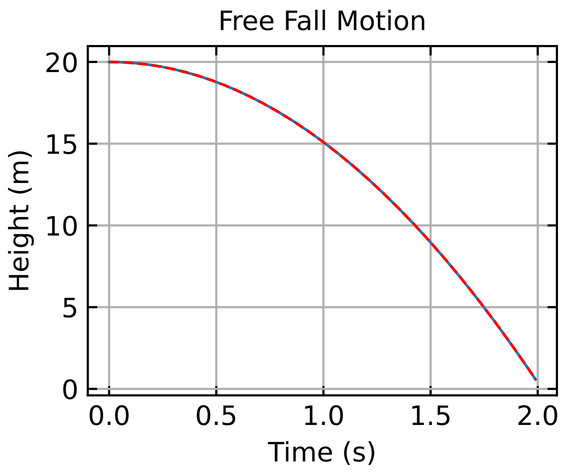
Values for atomic hydrogen

For atomic hydrogen (H), typical Lennard-Jones parameters are:

- $\sigma \approx 2.38$ Å $= 2.38 \times 10^{-10}$ meters
- $\epsilon \approx 0.0167$ kcal/mol $= 1.16 \times 10^{-21}$ joules

1.
2.
3.


Taylor Expansion of sin(x)
sin(x)
Taylor expansion
f(x)
x
2
1
0
−1
−2
−2
−1
0
1
2


Free Fall Motion
Height (m)
Time (s)
20
15
10
5
0
0.0
0.5
1.0
1.5
2.0

-
-

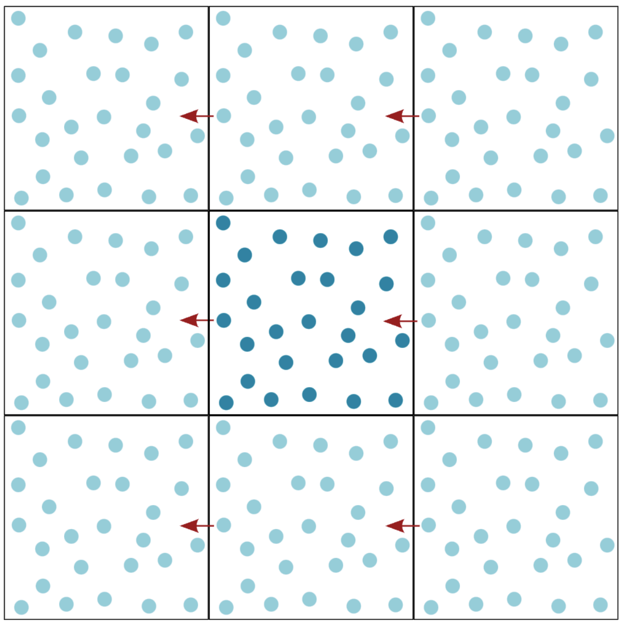


Figure 19.1: Perdiodic Boundary Conditions

The Minimum Image Convention in Molecular Dynamics

When we simulate particles in a box with periodic boundary conditions (meaning particles that leave on one side reappear on the opposite side), we need to calculate the forces between them correctly. Imagine two particles near opposite edges of the box: one at position x=1 and another at x=9 in a box of length 10. Without the minimum image convention, we would calculate their distance as 8 units (9-1). However, due to the periodic boundaries, these particles could actually interact across the boundary, with a shorter distance of just 2 units! The minimum image convention automatically finds this shortest distance, ensuring that we calculate the forces between particles correctly. It's like taking a shortcut across the periodic boundary instead of walking the longer path through the box.

-
-
-
-
-

-
-

Maxwell-Boltzmann Velocities in 3D

The probability distribution for the velocity magnitude in 3D is:

$$f(v) = 4\pi v^2 \left(\frac{m}{2\pi k_B T}\right)^{3/2} \exp\left(-\frac{mv^2}{2k_B T}\right)$$

- Mean velocity magnitude:

$$\langle v \rangle = \sqrt{\frac{8k_B T}{\pi m}}$$

- Most probable velocity (peak of distribution):

$$v_{mp} = \sqrt{\frac{2k_BT}{m}}$$

- Root mean square velocity:

$$v_{rms} = \sqrt{\frac{3k_BT}{m}}$$

These velocities can also be expressed in terms of the kinetic energy of the particles. The average kinetic energy per particle is:

$$\langle E_{kin} \rangle = \frac{3}{2}k_BT$$

Then we can express the velocities as:

- Mean velocity magnitude:

$$\langle v \rangle = \sqrt{\frac{8\langle E_{kin} \rangle}{3\pi m}}$$

- Most probable velocity:

$$v_{mp} = \sqrt{\frac{4\langle E_{kin} \rangle}{3m}}$$

- Root mean square velocity:

$$v_{rms} = \sqrt{\frac{2\langle E_{kin} \rangle}{m}}$$

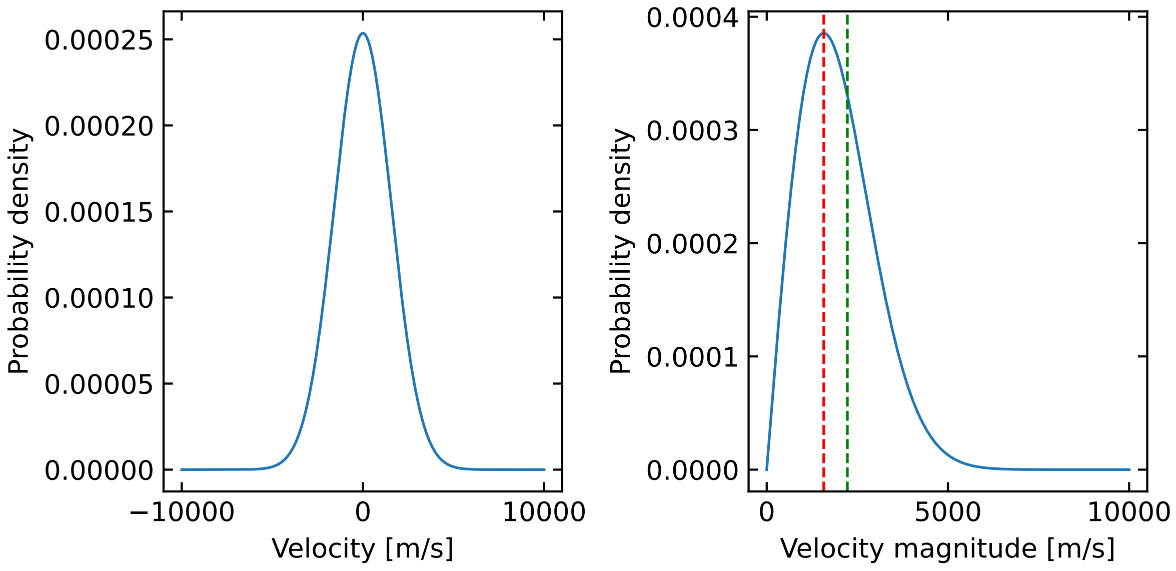


Figure 19.2: Maxwell Boltzmann distribution of speeds for one component of the velocity and the magnitude of the velocity vector.

-
-
-

```
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
from scipy.spatial.distance import cdist

n_side =2

x = np.linspace(0.05, 0.95, n_side)                                                          ①
y = np.linspace(0.05, 0.95, n_side)
xx, yy = np.meshgrid(x, y)                                                                   ②
particles = np.vstack([xx.ravel(), yy.ravel()]).T                                            ③

velocities = np.random.normal(scale=0.005, size=(n_side**2, 2))

radius = 0.0177
fig, ax = plt.subplots(figsize=(9,9))

n_steps = 200

for _ in range(n_steps):                                                                     ④
    clear_output(wait=True)                                                                  ⑤

    # Update particle positions based on their velocities
    particles += velocities
    # Apply periodic boundary conditions in x direction (wrap around at 0 and 1)
    particles[:, 0] = particles[:, 0] % 1
    # Apply periodic boundary conditions in y direction (wrap around at 0 and 1)
    particles[:, 1] = particles[:, 1] % 1
    # Calculate distances between all pairs of particles
    distances = cdist(particles, particles)

    # Calculate collisions using the upper triangle of the distance matrix
    # distances < 2*radius gives a boolean matrix where True means collision
    # np.triu takes only the upper triangle to avoid counting collisions twice
    collisions = np.triu(distances < 2*radius, 1)

    # Handle collisions between particles
    for i, j in zip(*np.nonzero(collisions)):
        # Exchange velocities between colliding particles (elastic collision)
        velocities[i], velocities[j] = velocities[j], velocities[i].copy()        ⑥

        # Calculate how much particles overlap
        overlap = 2*radius - distances[i, j]

        # Calculate unit vector pointing from j to i
        direction = particles[i] - particles[j]
        direction /= np.linalg.norm(direction)

        # Move particles apart to prevent overlap
        particles[i] += 0.5 * overlap * direction
        particles[j] -= 0.5 * overlap * direction

    ax.scatter(particles[:, 0], particles[:, 1], s=100, edgecolors='r', facecolors='none')


    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    ax.axis("off")

    display(fig)
    plt.pause(0.01)

    # Clear the current plot to prepare for next frame
    ax.clear()
```

①
②
③
④
⑤
⑥

-
-

```
from IPython.display import clear_output
from scipy.spatial.distance import cdist
from scipy.stats import maxwell

# Increase number of particles for better statistics
n_side = 10  # Creates 100 particles
x = np.linspace(0.05, 0.95, n_side)
y = np.linspace(0.05, 0.95, n_side)
xx, yy = np.meshgrid(x, y)
particles = np.vstack([xx.ravel(), yy.ravel()]).T
```

```python
# Initialize with normal distribution
velocities = np.random.normal(scale=0.005, size=(n_side**2, 2))
radius = 0.0177

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=get_size(15, 8))
n_steps = 500

# Store velocity magnitudes for analysis
velocity_history = []

for step in range(n_steps):
    clear_output(wait=True)
    # Update particle positions based on their velocities
    particles += velocities

    # Apply periodic boundary conditions
    particles = particles % 1

    # Calculate distances between all pairs of particles
    distances = cdist(particles, particles)

    # Calculate collisions using the upper triangle of the distance matrix
    collisions = np.triu(distances < 2*radius, 1)

    # Handle collisions between particles
    for i, j in zip(*np.nonzero(collisions)):
        # Get particle positions and velocities
        pos_i, pos_j = particles[i], particles[j]
        vel_i, vel_j = velocities[i], velocities[j]

        # Calculate relative position vector (line of centers)
        r_ij = pos_i - pos_j
        dist = np.linalg.norm(r_ij)
        r_ij_normalized = r_ij / dist if dist > 0 else np.array([1, 0])

        # *** CORRECTED COLLISION PHYSICS ***
        # Split velocities into components parallel and perpendicular to collision axis
        v_i_parallel = np.dot(vel_i, r_ij_normalized) * r_ij_normalized
        v_i_perp = vel_i - v_i_parallel

        v_j_parallel = np.dot(vel_j, r_ij_normalized) * r_ij_normalized
        v_j_perp = vel_j - v_j_parallel

        # Exchange parallel components (proper elastic collision)
        velocities[i] = v_i_perp + v_j_parallel
        velocities[j] = v_j_perp + v_i_parallel

        # Move particles apart to prevent overlap
        overlap = 2*radius - dist
        particles[i] += 0.5 * overlap * r_ij_normalized
        particles[j] -= 0.5 * overlap * r_ij_normalized

    # Every 5 steps, record velocity magnitudes
    if step % 5 == 0:
        speeds = np.sqrt(np.sum(velocities**2, axis=1))
        velocity_history.extend(speeds)

    # Every 20 steps, update the visualization
    if step % 20 == 0:
        # Clear the current plot to prepare for next frame
        ax1.clear()
        ax2.clear()

        # Plot particles
        ax1.scatter(particles[:, 0], particles[:, 1], s=100, edgecolors='r', facecolors='none')
        ax1.set_xlim(0, 1)
        ax1.set_ylim(0, 1)
        ax1.set_title(f"Step {step}")
        ax1.axis("off")

        # Plot velocity distribution
        if velocity_history:
            # Plot histogram of speeds
            hist, bins = np.histogram(velocity_history, bins=30, density=True)
            bin_centers = 0.5 * (bins[1:] + bins[:-1])
            ax2.bar(bin_centers, hist, width=bins[1]-bins[0], alpha=0.7, label='Simulation')

            # Plot Maxwell-Boltzmann distribution for comparison
            # For 2D Maxwell-Boltzmann, use Rayleigh distribution parameters
            scale = np.std(velocity_history) / np.sqrt(1 - 2/np.pi)
            x = np.linspace(0, max(velocity_history), 100)

            # In 2D, speed follows Rayleigh distribution
            rayleigh_pdf = (x/scale**2) * np.exp(-x**2/(2*scale**2))
            ax2.plot(x, rayleigh_pdf, 'r-', lw=2, label='Maxwell-Boltzmann (2D)')

            ax2.set_title("Speed Distribution")
            ax2.set_xlabel("Speed")
            ax2.set_ylabel("Probability Density")
            ax2.legend()

        display(fig)
        plt.pause(0.01)

# Final velocity distribution
plt.figure(figsize=get_size(12, 8))
hist, bins = np.histogram(velocity_history, bins=30, density=True)
bin_centers = 0.5 * (bins[1:] + bins[:-1])
plt.bar(bin_centers, hist, width=bins[1]-bins[0], alpha=0.7, label='Simulation')

# Plot ideal Maxwell-Boltzmann distribution for 2D (Rayleigh)
scale = np.std(velocity_history) / np.sqrt(1 - 2/np.pi)
x = np.linspace(0, max(velocity_history), 100)
rayleigh_pdf = (x/scale**2) * np.exp(-x**2/(2*scale**2))
plt.plot(x, rayleigh_pdf, 'r-', lw=2, label='MB (2D)')


plt.xlabel(r"speed $v$")
plt.ylabel("probability density")
plt.legend()
plt.show()
```

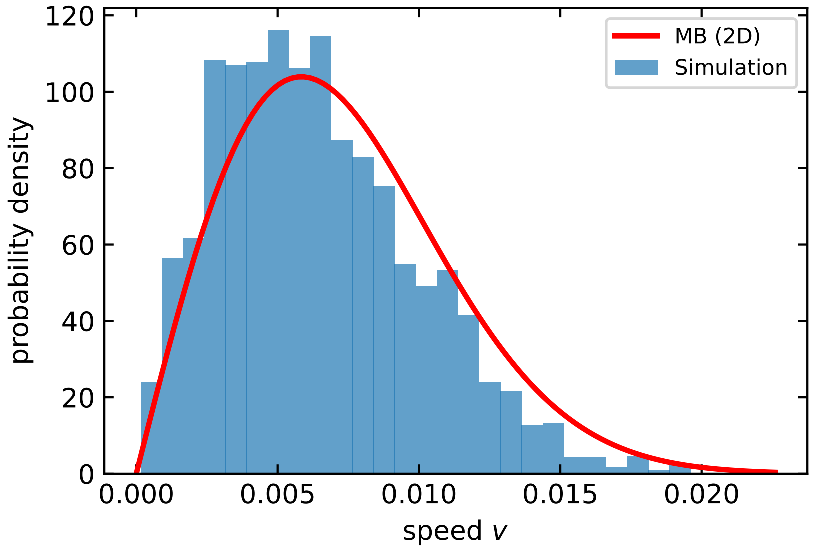


Figure 19.3: Simulated and analytica for the Maxwell-Boltzmann distribution.

```python
class Atom:
    dimension = 2

    def __init__(self, atom_id, atom_type, position, velocity=None, mass=None):
        self.id = atom_id
        self.type = atom_type
        self.position = position
        self.velocity = velocity if velocity is not None else np.zeros(dimension)
        self.mass = mass
        self.force = np.zeros(dimension)
```

-
-
-
-
-
-

```python
def add_force(self, force):
    """Add force contribution to total force on atom"""
    self.force += force
```

```python
def reset_force(self):
    """Reset force to zero at start of each step"""
    self.force = np.zeros(dimension)
```

```python
def update_position(self, dt):
    """First step of velocity Verlet: update position"""
    self.position += self.velocity * dt + 0.5 * (self.force/self.mass) * dt**2
```

```python
def update_velocity(self, dt, new_force):
    """Second step of velocity Verlet: update velocity using average force"""
    self.velocity += 0.5 * (new_force + self.force)/self.mass * dt
    self.force = new_force
```

```python
def apply_periodic_boundaries(self, box_size):
        """Apply periodic boundary conditions"""
        self.position = self.position % box_size
```

Complete Atom class

```
class Atom:
    def __init__(self, atom_id, atom_type, position, velocity=None, mass=None):
        self.id = atom_id
        self.type = atom_type
        self.position = position
        self.velocity = velocity if velocity is not None else np.random.randn(2)*20
        self.mass = mass
        self.force = np.zeros(2)


    def add_force(self, force):
        """Add force contribution to total force on atom"""
        self.force += force

    def reset_force(self):
        """Reset force to zero at start of each step"""
        self.force = np.zeros(2)

    def update_position(self, dt):
        """First step of velocity Verlet: update position"""
        self.position += self.velocity * dt + 0.5 * (self.force/self.mass) * dt**2

    def update_velocity(self, dt, new_force):
        """Second step of velocity Verlet: update velocity using average force"""
        self.velocity += 0.5 * (new_force + self.force)/self.mass * dt
        self.force = new_force

    def apply_periodic_boundaries(self, box_size):
            """Apply periodic boundary conditions"""
            self.position = self.position % box_size
```

```
#| autorun: true
#| edit: false
#| echo: false
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 8,
                     'lines.linewidth': 1,
                     'lines.markersize': 10,
                     'axes.labelsize': 10,
                     'axes.titlesize': 10,
                     '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))
```

```
class Atom:
    def __init__(self, atom_id, atom_type, position, velocity=None, mass=None):
        self.id = atom_id
        self.type = atom_type
        self.position = position
        self.velocity = velocity if velocity is not None else np.random.randn(2)*20
        self.mass = mass
        self.force = np.zeros(2)


    def add_force(self, force):
        """Add force contribution to total force on atom"""
        self.force += force

    def reset_force(self):
        """Reset force to zero at start of each step"""
        self.force = np.zeros(2)

    def update_position(self, dt):
        """First step of velocity Verlet: update position"""
        self.position += self.velocity * dt + 0.5 * (self.force/self.mass) * dt**2

    def update_velocity(self, dt, new_force):
        """Second step of velocity Verlet: update velocity using average force"""
        self.velocity += 0.5 * (new_force + self.force)/self.mass * dt
        self.force = new_force

    def apply_periodic_boundaries(self, box_size):
            """Apply periodic boundary conditions"""
            self.position = self.position % box_size
```

```
atoms = [
    Atom(0, 'C', np.array([0.0, 0.0]), velocity=np.array([1.0, 1.0]), mass=1.0),
    Atom(1, 'C', np.array([2.0, 2.0]), velocity=np.array([-1.0, 1.0]), mass=1.0)
]

# Visualize positions
plt.figure(figsize=(6,6))
for atom in atoms:
    plt.plot(atom.position[0], atom.position[1], 'o')
    # Add velocity arrows
    plt.arrow(atom.position[0], atom.position[1],
             atom.velocity[0], atom.velocity[1],
             head_width=0.1)

plt.axis('equal')
plt.show()
```

```
#| autorun: true
#| edit: false
#| echo: false
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 8,
                     'lines.linewidth': 1,
                     'lines.markersize': 10,
                     'axes.labelsize': 10,
                     'axes.titlesize': 10,
                     '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))
class Atom:
    def __init__(self, atom_id, atom_type, position, velocity=None, mass=None):
        self.id = atom_id
        self.type = atom_type
        self.position = position
        self.velocity = velocity if velocity is not None else np.random.randn(2)*20
        self.mass = mass
        self.force = np.zeros(2)


    def add_force(self, force):
        """Add force contribution to total force on atom"""
        self.force += force

    def reset_force(self):
        """Reset force to zero at start of each step"""
        self.force = np.zeros(2)
    def update_position(self, dt):
        """First step of velocity Verlet: update position"""
        self.position += self.velocity * dt + 0.5 * (self.force/self.mass) * dt**2

    def update_velocity(self, dt, new_force):
        """Second step of velocity Verlet: update velocity using average force"""
        self.velocity += 0.5 * (new_force + self.force)/self.mass * dt
        self.force = new_force

    def apply_periodic_boundaries(self, box_size):
            """Apply periodic boundary conditions"""
            self.position = self.position % box_size
```

-
-

```
class ForceField:
    def __init__(self):
        self.parameters = {
            'C': {'epsilon': 1.615, 'sigma': 1.36},
            'H': {'epsilon': 1.0, 'sigma': 1.0 },
            'O': {'epsilon': 1.846, 'sigma': 3.0},
        }
        self.box_size = None  # Will be set when initializing the simulation
```

-
-

-
-
-

```python
def get_pair_parameters(self, type1, type2):
    # Apply mixing rules when needed
    eps1 = self.parameters[type1]['epsilon']
    eps2 = self.parameters[type2]['epsilon']
    sig1 = self.parameters[type1]['sigma']
    sig2 = self.parameters[type2]['sigma']

    # Lorentz-Berthelot mixing rules
    epsilon = np.sqrt(eps1 * eps2)
    sigma = (sig1 + sig2) / 2

    return epsilon, sigma
```

```python
def minimum_image_distance(self, pos1, pos2):
    """Calculate minimum image distance between two positions"""
    delta = pos1 - pos2
    # Apply minimum image convention
    delta = delta - self.box_size * np.round(delta / self.box_size)
    return delta
```

```python
def calculate_lj_force(self, atom1, atom2):
    epsilon, sigma = self.get_pair_parameters(atom1.type, atom2.type)
    r = self.minimum_image_distance(atom1.position, atom2.position)
    r_mag = np.linalg.norm(r)

    # Add cutoff distance for stability
    if r_mag > 2.5*sigma:
        return np.zeros(2)

    force_mag = 24 * epsilon * (
        2 * (sigma/r_mag)**13
        - (sigma/r_mag)**7
    )
    force = force_mag * r/r_mag
    return force
```

Complete ForceField class

```
class ForceField:
    def __init__(self):
        self.parameters = {
            'C': {'epsilon': 1.615, 'sigma': 1.36},
            'H': {'epsilon': 1.0, 'sigma': 1.0 },
            'O': {'epsilon': 1.846, 'sigma': 3.0},
        }
        self.box_size = None  # Will be set when initializing the simulation

    def get_pair_parameters(self, type1, type2):
        # Apply mixing rules when needed
        eps1 = self.parameters[type1]['epsilon']
        eps2 = self.parameters[type2]['epsilon']
        sig1 = self.parameters[type1]['sigma']
        sig2 = self.parameters[type2]['sigma']

        # Lorentz-Berthelot mixing rules
        epsilon = np.sqrt(eps1 * eps2)
        sigma = (sig1 + sig2) / 2

        return epsilon, sigma

    def minimum_image_distance(self, pos1, pos2):
        """Calculate minimum image distance between two positions"""
        delta = pos1 - pos2
        # Apply minimum image convention
        delta = delta - self.box_size * np.round(delta / self.box_size)
        return delta

    def calculate_lj_force(self, atom1, atom2):
        epsilon, sigma = self.get_pair_parameters(atom1.type, atom2.type)
        r = self.minimum_image_distance(atom1.position, atom2.position)
        r_mag = np.linalg.norm(r)

        # Add cutoff distance for stability
        if r_mag > 2.5*sigma:
            return np.zeros(2)

        force_mag = 24 * epsilon * (
            2 * (sigma/r_mag)**13
            - (sigma/r_mag)**7
        )
        force = force_mag * r/r_mag
        return force
```

```python
class MDSimulation:
    def __init__(self, atoms, forcefield, timestep, box_size):
        self.atoms = atoms
        self.forcefield = forcefield
        self.forcefield.box_size = box_size  # Set box size in forcefield
        self.timestep = timestep
        self.box_size = np.array(box_size)
        self.initial_energy = None
        self.energy_history = []
```

```python
    def calculate_forces(self):
        # Reset all forces
        for atom in self.atoms:
            atom.reset_force()

        # Calculate forces between all pairs
        for i, atom1 in enumerate(self.atoms):
            for atom2 in self.atoms[i+1:]:
                force = self.forcefield.calculate_lj_force(atom1, atom2)
                atom1.add_force(force)
                atom2.add_force(-force)  # Newton's third law
```

```python
    def update_positions_and_velocities(self):
        # First step: Update positions using current forces
        for atom in self.atoms:
            atom.update_position(self.timestep)
            # Apply periodic boundary conditions
            atom.apply_periodic_boundaries(self.box_size)

        # Store current forces for velocity update
        old_forces = {atom.id: atom.force.copy() for atom in self.atoms}

        # Recalculate forces with new positions
        self.calculate_forces()

        # Second step: Update velocities using average of old and new forces
        for atom in self.atoms:
            atom.update_velocity(self.timestep, atom.force)
```

Complete MDSimulation class

-
-
-
-