Computational Physics Lecture Notes

Li Zhu

May 19, 2026

CONTENTS

Why do we need to study computational methods and programming?

Computers have become the "mathematics" of the modern era—they are, in essence, the capability to solve problems. In the 17th century, mathematics had its golden age: by mastering math, one could tackle many practical issues in astronomy, physics, and engineering. In other words, mathematics represented "problem-solving ability."

In the 21st century, the types of problems solvable solely by mathematics have basically already been solved. What remain are complex systems and complex engineering challenges. These problems inevitably require computing power, storage capabilities, and AI.

So, why study computational methods? Because in today's world, it is essentially the discipline of "problem-solving." Once you've mastered computational science skills, you can apply them to solve complex problems and handle complex projects across all industries.

About AI/ChatGPT

In today's age of AI and ChatGPT, many people might wonder why we still need to learn programming. After all, these powerful tools seem capable of doing so much already. However, rather than diminishing the importance of programming, AI has actually made it more essential than ever.

Tools like ChatGPT/Gemini/Claude operate within the framework we set for them. To maximize their benefits and customize or extend them to fit our needs, programming skills are crucial. By learning to code, we gain a better understanding of how these technologies work, giving us control rather than simply being passive users.

Programming is a form of creative expression. Even though AI can assist with code generation or problem-solving, truly understanding how to implement features or optimize algorithms still requires engineering thinking and creativity. Mastering programming skills allows for flexible realization of complex ideas.

Technology changes rapidly, with new frameworks, languages, and use cases emerging all the time. Learning to program fundamentally means learning a systematic problem-solving approach. Regardless of how technology evolves, this mindset will help you quickly adapt to new tools and platforms.

In short, if you know how to program, AI will be like adding wings to a tiger—making you even more powerful. But if you don't know how to code and only rely on ChatGPT to solve problems, you'll become increasingly dependent on AI and, over time, lose your competitive edge.

Learning Objectives

By the end of this course, you will be able to:

- Write Python programs to solve physics and engineering problems
- Implement numerical methods for integration, differentiation, and interpolation
- Solve ordinary and partial differential equations computationally
- Apply Monte Carlo methods and stochastic simulations
- Use optimization techniques including global optimization algorithms
- Apply basic machine learning algorithms to scientific data

Prerequisites

- Basic calculus (derivatives, integrals)
- Linear algebra fundamentals (vectors, matrices)

- No prior programming experience required

Location: Smith Hall 242

Schedule: Mondays/Thursdays 1:00 - 2:20 pm

Instructor	Prof. Li Zhu
Email	li.zhu@rutgers.edu
Website	https://zhuli.name
Office	Smith 504
Office hours	by appointment

Course Outline

	Subjects
1	Python basics I
2	Python basics II
3	Python basics III
4	Integrals
5	Derivatives
6	Interpolation
7	Fitting
8	Fourier transform
9	Fast Fourier transform
10	Ordinary differential equations
11	Partial differential equations
12	Stochastic method I
13	Stochastic method II
14	Monte carlo III
15	Optimization I
16	Optimization II
17	Global Optimization I
18	Global Optimization II
19	Global Optimization III
20	Machine Learning I (Algorithms)
21	Machine Learning II (Applications)
22	Machine Learning III (Neural Networks)
23	Machine Learning IV (Graph Neural Networks)

Textbook: ***Computational Physics, M. Newman (not required)***

Additional Resources

- **Python**: Python.org Tutorial
- **NumPy**: NumPy Quickstart
- **Matplotlib**: Pyplot Tutorial

Grade Distribution:

Items	Percentage
Attendance	10%
Homeworks	30%
Midterm Project	30%
Final Project	30%

Course Description

This course is open to both undergraduate and graduate students interested in scientific programming and data analysis. There will be weekly assignments and two projects during the semester.

Please bring your laptop/tablet (iPhone/Android) to class. All practices will be based on Python 3.

We recommend using Colab (https://colab.research.google.com/) as the coding environment. The advantage of Colab is that it can be used on various computational devices, including laptops, tablets, and even smartphones, if you choose.

AI Policy

You are encouraged to use AI tools (ChatGPT, Claude, Copilot) as learning aids. However:

- You must understand and be able to explain any code you submit
- Cite AI assistance in your submissions

Lecture Notes

You can find the lecture notes on Google Drive.

Please make sure to save a copy to your own Google Drive.

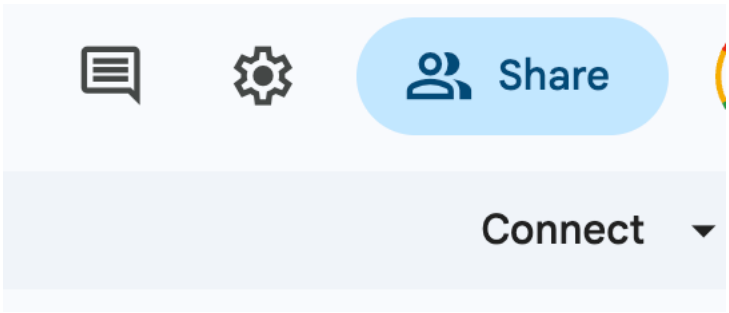
The lecture notes will be uploaded to the course Google Drive prior to each class.

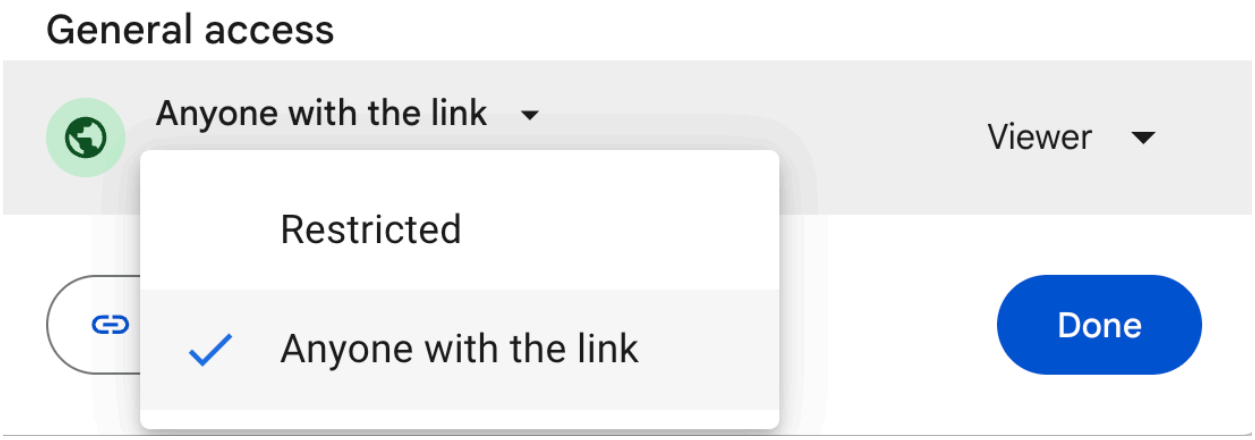
Homework Submission

Assignments will be provided in the form of **Jupyter (`.ipynb`) notebooks**. Please save a copy of each assignment to your own Google Drive and complete it there.

To submit the assignment, follow **both** steps below:

1. **Export the notebook to PDF** (File → Print → "Save as PDF") and submit the PDF file to Canvas.
2. Click **Share** in the top-right corner of the notebook, set access to **"Anyone with the link – Viewer"**, copy the link, and paste it into the **Submission Comments box** in Canvas **along with uploading your PDF file**.





CHAPTER

ONE

PYTHON BASICS I

Welcome to **Python Basics I**, where you'll learn the fundamental building blocks of Python:

- Printing and running Python code
- Basic data types (integers, floats, booleans, strings)
- Basic operators (arithmetic, comparison, logical)
- Input and output
- Control flow (if statements, for/while loops)

By the end of this notebook, you should be comfortable writing simple Python scripts, understanding indentation, and implementing small programs that use conditionals and loops.

1.1 Preliminary Setup

No need to install if you are using Google Colab.

If you want to run Python code on your own devices, JupyterLab Desktop is recommended for this course. You can download it from https://github.com/jupyterlab/jupyterlab-desktop/releases.

Let's get started by printing a simple message to the screen.

1.1.1 Hello, Physics!

This is the traditional "Hello, World!" program, adapted for physics.

```
print("Hello, Physics!")
```

If you run the cell above, you should see `Hello, Physics!` printed below it.

1.2 1. Python Basic Syntax & Indentation

Python uses **indentation** (instead of braces `{}`) to define code blocks. For example:

```
if True:
    print("This is inside the if-statement.")
    print("Note the indentation!")
```

Key Points:

- A code block is typically indented by 4 spaces.
- If indentation is off, Python will raise an `IndentationError`.

1.3 2. Making graphs

First we need to import a couple libraries: **numpy** and **matplotlib**. Libraries are collections of python functions that other people have written. We will frequently need these two libraries for doing physics.

```
# for mathematics and working with arrays
import numpy as np
# for plotting
import matplotlib.pyplot as plt
```

Python completely ignores any line that begins with #. These lines are called 'comments', and they are a critical part of all of your code.

Now let's pick some points to plot. We'll store the x-values in the variable x and the y-values in the variable y.

We store the values as arrays, which are a useful data type for math and plotting and working with large collections of numbers. Notice we use numpy to define the array (through it's nickname "np"), since arrays aren't built into python by default.

```
x = np.array([1,2,3,4,5])
y = x+3
print (y)
```

Let's plot it! We need matplotlib.pyplot to do this, which we nicknamed "plt".

```
plt.plot(x,y,'r*')
```

The graph is ugly, so let's clean it up a bit.

Matplotlib includes all sorts of functions to clean up and label plots. Let's try some of them:

```
plt.plot(x,y,'g+')
plt.title('Title X')
plt.xlabel('Time (days)')
plt.ylabel('Distance (miles)')
plt.xlim(0, 6)
plt.ylim(2, 10);
```

Let's take a moment to talk about what's we've done so far. For starters, x and y are variables. Variables in Python are essentially storage bins: x in this case is an address which points to a memory bin somewhere in the computer that contains an array of 5 numbers. Python variables can point to bins containing just about anything: different types of numbers, lists, files on the hard drive, strings of text characters, true/false values, other bits of Python code, whatever! When any other

line in the Python script refers to a variable, Python looks at the appropriate memory bin and pulls out those contents. When Python gets the line

```
y = x+3
```

It pulls out the x array, adds three to everything in that array, puts the resulting array in another memory bin, and makes y point to that new bin. The plot command `plt.plot(x,y,'rx')` creates a new figure window if none exists, then makes a graph in that window. The first item in parenthesis is the x data, the second is the y data, and the third is a description of how the data should be represented on the graph, in this case red x symbols.

Let's build a fancier plot this time.

```
# this linspace function generates 100 values evenly spaced between 0.0 and 10.0
time = np.linspace(0.0, 10.0, 100)
# use exponential and sine functions from numpy to define the y-values (height) from␣
 ↪the x-values (time)
height = np.exp(-time/3.0) * np.sin(time*3)
# plot time vs height, with magenta triangles for each point, connected by lines
plt.plot(time, height, 'm-^')
# on the same figure, plot time vs a sinusoid with constant height, and connect the␣
 ↪points with green lines
plt.plot(time, 0.7*np.sin(time*3), 'g-')
# add a legend to label the different curves
plt.legend(['damped', 'constant amplitude'], loc='upper right')
# label the x-axis
plt.xlabel('Time (s)')
plt.ylabel('Height')
```

1.3.1 Importing and graphing real data

It's unlikely that you would be particularly excited by the prospect of manually typing in data from every experiment or simulation. The whole point of computers, after all, is to save us effort! Python can read data from text files quite well. We'll discuss this ability more in later, but for now here's a quick way of reading data files for graphing.

Suppose we have a text file with real data, and we want to graph it. We need to find the file, and then we need to import it into python.

First, we need to download the data file 'microphones.txt'. The exclamation mark (!) allows users to run shell commands from inside a Jupyter Notebook code cell. Simply start a line of code with ! and it will run the command in the shell.

Example: !pwd will print the current working directory. Here, we use the command 'wget' to downloand the data file.

```
from pathlib import Path

Path.cwd()
```

```
from pathlib import Path

sorted(item.name for item in Path.cwd().iterdir())[:10]
```

```
from urllib.request import urlretrieve

url = "https://zhuli.name/files/microphones.txt"
urlretrieve(url, "microphones.txt")
```

```
from pathlib import Path

"microphones.txt" in [item.name for item in Path.cwd().iterdir()]
```

```
#frequency, mic1, mic2 = np.loadtxt('microphones.txt', unpack = True)
data = np.loadtxt('microphones.txt', unpack=True)
print (data)
frequency, mic1, mic2 = np.loadtxt('microphones.txt', unpack = True)
```

```
print(mic1)
print(len(mic1))
plt.plot(frequency, mic1)
```

Now let's make a labeled plot, commenting our code for clarity:

```
# plot frequency vs mic1 amplitude with a red curve, and
#      frequency vs mic2 amplitude with a blue curve
plt.plot(frequency, mic1, 'r-', frequency, mic2*(-2.5)+5, 'b-')
# label the plot
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude(arbitrary units)')
plt.legend(['Microphone 1', 'Microphone 2']);
```

1.4 3. Variables and Data Types

1.4.1 3.1 Variables

A **variable** is a name that refers to a value. In Python, you assign variables using =:

```
x = 10        # integer
y = 3.14      # float
name = "Alice"  # string
```

```
# Example:
x = 10
y = 3.14
name = "Alice"
print("x =", x)
print("y =", y)
print("name =", name)

t = False
print (t)
```

1.4.2 3.2 Data Types

Common data types in Python:

1. **Integer (`int`)** – whole numbers (e.g., 42).
2. **Float (`float`)** – decimal or real numbers (e.g., 3.14).
3. **Boolean (`bool`)** – `True` or `False`.
4. **String (`str`)** – sequence of characters in quotes (e.g., "Hello").

Use `type()` to check:

```
a = 5
b = 2.718
c = "Hello"
print(type(a))  # <class 'int'>
print(type(b))  # <class 'float'>
print(type(c))  # <class 'str'>
```

```
# Let's test type() ourselves
a = 5
b = 2.718
c = "Hello"
d = True
e = 1+2j

print("a =", a, ", type:", type(a))
print("b =", b, ", type:", type(b))
print("c =", c, ", type:", type(c))
print("d =", d, ", type:", type(d))
print("e =", e, ", type:", type(e))

print ('real ', e.real)
print ('imag ', e.imag)
```

1.4.3 3.3 String Operations

Strings are sequences of characters. Python provides many useful operations for working with strings:

```
## String concatenation
first = "Computational"
last = "Physics"
full = first + " " + last  # "Computational Physics"

## f-strings (formatted strings) - very useful for physics!
mass = 9.109e-31
print(f"Electron mass: {mass:.3e} kg")  # Electron mass: 9.109e-31 kg

## String methods
text = "  Hello Physics  "
print(text.strip())   # "Hello Physics" (removes whitespace)
print(text.lower())   # "  hello physics  "
print(text.upper())   # "  HELLO PHYSICS  "
```

```
# Try string operations
course = "Computational"
subject = "Physics"
full_name = course + " " + subject
print(full_name)

# f-strings for formatted output (very useful!)
c = 299792458  # speed of light in m/s
print(f"Speed of light: {c} m/s")
print(f"Speed of light: {c:.2e} m/s")  # scientific notation
```

1.4.4 3.4 Type Conversion

You often need to convert between data types. This is especially important when reading user input (which is always a string).

```
## String to numbers
x = int("42")       # string to int: 42
y = float("3.14")   # string to float: 3.14

## Numbers to string
z = str(299792458)  # int to string: "299792458"

## Why this matters for input()
speed = float(input("Enter speed: "))  # Convert input to float
```

```
# Type conversion examples
num_str = "42"
num_int = int(num_str)
print(f"{num_str} as integer: {num_int}, type: {type(num_int)}")

pi_str = "3.14159"
pi_float = float(pi_str)
print(f"{pi_str} as float: {pi_float}, type: {type(pi_float)}")

# Be careful! This will cause an error:
# int("3.14")  # Can't convert float string directly to int
```

1.4.5 3.5 List Basics

Lists store multiple values in a single variable. They are essential for working with data.

```
## Creating lists
measurements = [1.2, 3.4, 5.6, 7.8]
names = ["electron", "proton", "neutron"]
mixed = [1, "two", 3.0, True]  # Can mix types (but usually don't)

## Accessing elements (0-indexed!)
print(measurements[0])   # First element: 1.2
print(measurements[-1])  # Last element: 7.8

## List length
print(len(measurements)) # 4
```

(continues on next page)

(continued from previous page)

```
## Adding elements
measurements.append(9.0) # Add to end
```

```
# List basics
masses = [9.109e-31, 1.673e-27, 1.675e-27]  # electron, proton, neutron masses in kg
particles = ["electron", "proton", "neutron"]

print(f"First particle: {particles[0]}")
print(f"Its mass: {masses[0]:.3e} kg")
print(f"Number of particles: {len(particles)}")

# Add a new particle
particles.append("muon")
masses.append(1.884e-28)
print(f"Updated list: {particles}")
```

1.5 4. Operators

1.5.1 4.1 Arithmetic Operators

- + (addition)
- − (subtraction)
- * (multiplication)
- / (division, float result)
- // (floor division, integer result)
- % (modulus, remainder)
- ** (exponentiation)

Example:

```
x = 5
y = 2
print(x + y)   # 7
print(x - y)   # 3
print(x * y)   # 10
print(x / y)   # 2.5
print(x // y)  # 2
print(x % y)   # 1
print(x ** y)  # 25
```

```
# Let's try arithmetic:
x = 5
y = 2
print("x + y =", x + y)
print("x - y =", x - y)
print("x * y =", x * y)
print("x / y =", x / y)
print("x // y =", x // y)
```

(continues on next page)

(continued from previous page)

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

1.5.2 4.2 Comparison Operators

Compare two values; results are `True` or `False`.

- > (greater than)
- < (less than)
- == (equal to)
- != (not equal to)
- >= (greater or equal)
- <= (less or equal)

Example:

```
x = 5
y = 2
print(x > y)    # True
print(x < y)    # False
print(x == y)   # False
print(x != y)   # True
```

```
x = 5
y = 2
print("x > y?", x > y)
print("x < y?", x < y)
print("x == y?", x == y)
print("x != y?", x != y)
print("x >= y?", x >= y)
print("x <= y?", x <= y)

z = 2
print(x>y and z>y)
print(x>y or z>y)
```

1.5.3 4.3 Logical (Boolean) Operators

- `and`
- `or`
- `not`

Example:

```
p = True
q = False
print(p and q)   # False
print(p or q)    # True
print(not p)     # False
```

```
p = True
q = False
print("p and q =", p and q)
print("p or q  =", p or q)
print("not p   =", not p)
```

1.6 5. Comments

- **Single-line comments** use `#`.
- **Multi-line strings/docstrings** use triple quotes `"""..."""`.

Example:

```
## This is a single-line comment

"""
This is a multi-line string (often used as a docstring).
Not typically used for block commenting in real code.
"""
```

Comments help others (and your future self) understand the code.

1.7 6. Input/Output

1.7.1 6.1 Output with `print()`

We've already been using `print()`, which sends text to the console.

1.7.2 6.2 Reading Input with `input()`

We can prompt the user and read input from the console with `input()`. Note that `input()` returns a **string**, so you must convert it to a number if needed.

```
# Try running this cell. Enter a number at the prompt.
user_input = input("Enter a number: ")
print("You entered:", user_input, "which is of type", type(user_input))
x = float(user_input)  # Convert string to float
print("You entered:", x, "which is of type", type(x))
```

1.8 7. Control Flow

1.8.1 7.1 `if-elif-else` Statements

Used to execute code conditionally.

```
temperature = 15
if temperature < 0:
    print("It's freezing!")
elif temperature < 20:
    print("It's chilly.")
else:
    print("The weather is nice.")
```

```
temperature = 15
if temperature < 0:
    print("It's freezing!")
elif temperature < 20:
    print("It's chilly.")
else:
    print("The weather is nice.")
```

1.8.2 7.2 `for` Loops

Loop over a sequence or range.

```
for i in range(5):
    print("i =", i)
```

You can also iterate over elements in a list or string:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

```
for i in range(5):
    print("i =", i)
print ('------')
for i in [3,5,8,1,5]:
    print("i =", i)
```

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print("Fruit:", fruit)
```

1.8.3 7.2.1 The `range()` Function in Detail

The `range()` function is very flexible:

```
range(stop)              # 0 to stop-1
range(start, stop)       # start to stop-1
range(start, stop, step) # start to stop-1, stepping by step
```

Examples:

```
range(5)         # 0, 1, 2, 3, 4
range(2, 6)      # 2, 3, 4, 5
range(0, 10, 2) # 0, 2, 4, 6, 8 (even numbers)
range(5, 0, -1) # 5, 4, 3, 2, 1 (countdown!)
```

```
# range() examples
print("range(5):")
for i in range(5):
    print(i, end=" ")
print()

print("\nrange(2, 6):")
for i in range(2, 6):
    print(i, end=" ")
print()

print("\nrange(0, 10, 2) - even numbers:")
for i in range(0, 10, 2):
    print(i, end=" ")
print()

print("\nrange(5, 0, -1) - countdown:")
for i in range(5, 0, -1):
    print(i, end=" ")
print("\nBlastoff!")
```

1.8.4 7.2.2 `break` and `continue` Statements

Control loop execution:

- **`break`**: Exit the loop immediately
- **`continue`**: Skip to the next iteration

```
## break example: stop when we find what we want
for n in range(1, 100):
    if n > 10 and n % 2 == 0:
        print(f"First even number > 10: {n}")
        break

## continue example: skip certain values
for x in range(5):
    if x == 2:
        continue  # skip x=2
    print(x)  # prints 0, 1, 3, 4
```

```
# break: find first number divisible by 7 greater than 50
for n in range(1, 100):
    if n > 50 and n % 7 == 0:
        print(f"Found: {n}")
        break

print("---")

# continue: skip negative values in data processing
data = [1.5, -2.3, 3.7, -0.5, 4.2]
print("Processing only positive values:")
for value in data:
    if value < 0:
        continue  # skip negative
    print(f"  {value}")
```

1.8.5 7.3 `while` Loops

Repeat a block of code as long as a condition is `True`.

```
count = 0
while count < 3:
    print("Count is:", count)
    count += 1
```

```
count = 0
while count < 3:
    print("Count is:", count)
    # count += 1
    count = count + 1
```

1.9 8. Mini-Exercises

Try these to reinforce what you've learned.

1.9.1 Exercise: Conditionals – Positive, Negative, or Zero

1. Prompt the user to input a number.
2. Check if the number is positive, negative, or zero.
3. Print a message accordingly.

```
num = float(input("Enter a number: "))

if num > 0:
    print("The number is positive.")
elif num < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

1.10 8.5 Common Beginner Errors

Understanding error messages will save you hours of frustration! Here are the most common errors:

1.10.1 IndentationError

Python uses indentation to define code blocks. Inconsistent indentation causes errors.

```
if True:
print("This will fail!")  # Missing indentation!
```

1.10.2 NameError

Trying to use a variable that doesn't exist.

```
print(undefined_variable)  # NameError: name 'undefined_variable' is not defined
```

1.10.3 TypeError

Operating on incompatible types.

```
result = "5" + 3  # TypeError: can only concatenate str to str
## Fix: result = int("5") + 3  OR  result = "5" + str(3)
```

1.10.4 SyntaxError

Invalid Python syntax (missing colons, parentheses, quotes, etc.).

```
if x > 5     # SyntaxError: missing colon
print("hi"   # SyntaxError: missing closing parenthesis
```

```
# Uncomment each line one at a time to see the errors:

# NameError:
# print(my_undefined_variable)

# TypeError:
# result = "5" + 3

# The fix for TypeError:
result = int("5") + 3
print(f"Fixed result: {result}")
```

1.11 8.6 Physics Examples

Let's apply what we've learned to physics problems!

```
# Kinematics: Calculate final velocity
v0 = 10.0   # initial velocity (m/s)
a = -9.8    # acceleration due to gravity (m/s²)
t = 2.0     # time (s)

v = v0 + a * t
print(f"Initial velocity: {v0} m/s")
print(f"After {t} seconds: {v} m/s")
```

```
# Unit conversion with a loop
distances_km = [1, 5, 10, 42.195]  # marathon!

print("Distance Conversion:")
print("-" * 25)
for d in distances_km:
    miles = d * 0.621371
    print(f"{d:8.3f} km = {miles:10.3f} miles")
```

```
# Calculate kinetic energy for different velocities
mass = 1.0  # kg
velocities = [1, 2, 5, 10, 20]  # m/s

print(f"Kinetic Energy (mass = {mass} kg)")
print("-" * 30)
for v in velocities:
    KE = 0.5 * mass * v**2
    print(f"v = {v:2} m/s  →  KE = {KE:6.1f} J")
```

1.12 9. Additional Practice

Try writing small scripts that:

- Sum the squares of the first 10 integers.
- Check if a given year is a leap year (divisible by 4 but not by 100, unless also by 400).
- Create a multiplication table using nested loops.

1.13 10. Summary & Next Steps

- We covered:
    1. How to **print** in Python.
    2. Basic **data types** and **variables**.
    3. **Operators** (arithmetic, comparison, logical).
    4. **Input/Output**.
    5. **Control flow** (`if-elif-else`, `for`, `while`).

You now have a solid foundation in Python Basics I. In the next lesson, we'll dive deeper into data structures (lists, tuples, dictionaries) and learn about **functions**—the building blocks for writing organized, reusable code.

1.14 10. Using the `math` Module

Python's built-in `math` module provides mathematical functions and constants. This is essential for physics calculations!

```
import math

# Mathematical constants
print(f"π = {math.pi}")
print(f"e = {math.e}")

# Trigonometric functions (input in radians!)
print(f"\ncos(π) = {math.cos(math.pi)}")
print(f"sin(π/2) = {math.sin(math.pi/2)}")

# Other useful functions
print(f"\nsqrt(2) = {math.sqrt(2)}")
print(f"log(e) = {math.log(math.e)}")
print(f"log10(100) = {math.log10(100)}")
```

1.14.1 Getting Help

Use `help()` to learn about any function or module:

```
# Get help on a specific function
help(math.cos)

# Uncomment to see all functions in math module:
# help(math)
```

1.15 11. Summary & Next Steps

In this lecture, we covered:

1. **Making graphs** with NumPy and Matplotlib
2. **Variables and data types** (int, float, bool, str, list)
3. **String operations** and f-strings for formatting
4. **Type conversion** between data types
5. **Operators** (arithmetic, comparison, logical)
6. **Input/Output** with `print()` and `input()`
7. **Control flow** (`if-elif-else`, `for`, `while`, `break`, `continue`)
8. **Common errors** and how to fix them
9. **The `math` module** for mathematical functions

1.15.1 Next Lecture: Python Basics II

- Functions
- More on lists and other data structures
- File I/O
- Error handling with try/except

CHAPTER

TWO

PYTHON BASICS II

Welcome to **Python Basics II**! In this lecture, we'll cover:

1. **Functions —** Writing reusable code
2. **Data Structures —** Lists, tuples, dictionaries, and sets
3. **File I/O —** Reading and writing files
4. **Error Handling —** Making robust code with try/except

By the end of this notebook, you'll be able to write well-organized Python programs that can read data from files, process it with functions, and handle errors gracefully.

2.1 I Functions in Python

2.1.1 1. What is a Function?

In Python, a **function** is a reusable block of code that performs a specific task. It helps you:

- **Organize** code into logical sections.
- **Avoid repetition** by reusing the same logic multiple times.
- **Enhance readability** by making your code self-contained and modular.

Python functions are defined with the `def` keyword. A typical function looks like this:

```
def function_name(parameters):
    """
    Docstring: A description of the function's purpose, inputs, and outputs.
    """
    # function body
    # ...
    return result
```

A Simple Example

```
def greet(name):
    """Return a greeting for the given name."""
    return f"Hello, {name}!"

print(greet("Rutgers"))  # Expect: Hello, Alice!
```

```
Hello, Rutgers!
```

2.1.2 2. Function Parameters and Arguments

Functions can have different types of parameters and arguments to accommodate a variety of use cases.

2.1 Positional Arguments

By default, arguments are matched to parameters by position:

```
def add(x, y):
    """Return the sum of x and y."""
    return x + y

result = add(3, 5)  # x=3, y=5
print(result)       # Expect: 8
```

```
8
```

2.2 Keyword Arguments

You can also match arguments by explicitly naming them:

```
def connect(host, port):
    """Simulate connecting to a network host and port."""
    return f"Connecting to {host} on port {port}"

print(connect(port=8080, host="rutgers.edu"))
# Output: Connecting to localhost on port 8080
print(connect(host="rutgers.edu", port=8888))
```

```
Connecting to rutgers.edu on port 8080
Connecting to rutgers.edu on port 8888
```

2.3 Default Arguments

If a parameter has a default value, it becomes optional for the caller:

```
def power(base, exponent=2):
    """Raise base to a given exponent; default exponent is 2."""
    return base ** exponent

print(power(4))       # exponent defaults to 2 -> 16
print(power(4, 3))    # -> 64
power(5)
```

```
16
64
```

Note: Be cautious when using **mutable default arguments** (e.g., lists or dictionaries) since they can lead to unexpected behavior if modified in-place.

```
def add_item(item, items=[]):
    items.append(item)
    return items

print (add_item(1))
print (add_item(2))
print (add_item(3))

list1 = add_item(,list)

print (add_item(1, [44]))
```

```
[1]
[1, 2]
[1, 2, 3]
[44, 1]
```

```
def add_item2(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items

print (add_item2(1))
print (add_item2(2))
print (add_item2(3))
```

```
[1]
[2]
[3]
```

2.1.3 3. Return Values

1. **Single Return Value**: A function returns a single object—this object can be anything in Python (string, number, list, dictionary, etc.).
2. **Multiple Return Values**: Python's tuple-unpacking allows for multiple outputs.

```
def min_and_max(values):
    """Return both the minimum and maximum of a list of values."""
    return min(values), max(values)

mn, mx = min_and_max([3, 1, 5, 9, 2])
print(mn, mx)  # Expect: 1 9

mm = min_and_max([3, 1, 5, 9, 2])
print (mm)

def somefunc():
    return 4+5

print(somefunc())
```

```
1 9
(1, 9)
9
```

3. **No Explicit Return**: If a function ends without a `return` statement, Python implicitly returns `None`.

2.1.4 4. Documentation with Docstrings

A **docstring** is a specialized string literal that documents a function's purpose, parameters, and return values. Typically enclosed with triple quotes (`"""`). Good docstrings:

- Start with a one-sentence summary.
- Optionally include more detailed explanations or examples in subsequent paragraphs.
- Can use conventions like NumPy Docstrings or Google Python Style.

```
def linear_transform(x, a, b):
    """
    Compute a linear transformation of x: y = a*x + b.

    Parameters
    ----------
    x : float
        The input variable.
    a : float
        The slope of the linear function.
    b : float
        The intercept of the linear function.

    Returns
    -------
    float
        The value of y = a*x + b.
    """
    return a*x + b
```

```
help(linear_transform)
```

```
Help on function linear_transform in module __main__:

linear_transform(x, a, b)
    Compute a linear transformation of x: y = a*x + b.

    Parameters
    ----------
    x : float
        The input variable.
    a : float
        The slope of the linear function.
    b : float
        The intercept of the linear function.

    Returns
    -------
    float
        The value of y = a*x + b.
```

2.1.5 5. Scoping and Namespaces

Variables defined inside a function are local to that function, meaning they are not accessible outside unless explicitly returned. This principle (known as **scope**) protects your variables from unintentional modifications.

```
def foo():
    x = 10  # local variable
    return x

def foo2():
    x = 20
    return x

print(foo())
# A local variable is not available outside the function.
# Running `print(x)` here would raise a NameError.
```

2.1.6 6. An Introduction to Recursion

Though recursion can be powerful, it needs to be used judiciously:

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Be mindful of recursion depth limits in Python.

```
def factorial(n):
    if n <= 1:
        return 1
```

(continues on next page)

(continued from previous page)

```
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

```
120
```

2.1.7 7. Best Practices

- **Keep functions small** and focused.
- **Name functions** clearly and descriptively.
- Use **docstrings** and **type hints** for clarity.
- Validate inputs, watch for edge cases.
- Consider **unit tests** (e.g., `unittest`, `pytest`).

2.1.8 Summary

Functions are fundamental to writing modular, maintainable Python code. For a graduate student working on complex projects—perhaps in computational research, data science, or advanced engineering simulations—understanding how to design, document, and structure functions effectively is essential.

2.2 II Python Containers

This section provides an overview of **lists**, **tuples**, and **dictionaries**, including slicing and key/value manipulation. We'll explore usage and performance considerations, especially in large-scale or research scenarios.

2.3 1. Overview

In Python, a **container** is any object that holds an arbitrary number of other objects. The most commonly used containers are:

1. **Lists**: Ordered, mutable sequences.
2. **Tuples**: Ordered, immutable sequences.
3. **Dictionaries**: Key-value mappings, mutable in nature.

These containers are essential for organizing data in scientific computing and complex research applications.

2.4 2. Lists

2.4.1 2.1 Basic Usage

A Python list is defined using square brackets `[ ]`. It is mutable, meaning you can add, remove, or change elements in-place.

```
# Creating a list
data = [10, 20, 30, 40]

# Accessing elements by index
print(data[0])  # Output: 10
print(data[-1]) # Output: 40
print(data[3])
print(data[-2])

# Modifying elements
data[0] = 100
print('data', data)  # [100, 20, 30, 40]

data2 = data[:]  # data2 = data or data2=data[:]
print('data2', data2)

data2[0] = 1000
print('data', data)
print('data2', data2)

data2.append(10000)
print('data2', data2)
data2.reverse()
print('data2', data2)
```

```
10
40
40
30
data [100, 20, 30, 40]
data2 [100, 20, 30, 40]
data [100, 20, 30, 40]
data2 [1000, 20, 30, 40]
data2 [1000, 20, 30, 40, 10000]
data2 [10000, 40, 30, 20, 1000]
```

Common Methods

- **append(x)**: Add an item to the end.
- **extend(iterable)**: Add multiple items (another list or iterator).
- **insert(i, x)**: Insert `x` at index `i`.
- **pop(i)**: Remove and return the item at index `i` (default last).
- **remove(x)**: Remove the first occurrence of `x`.
- **sort()**: Sort the list in-place.

- **reverse**(): Reverse the list in-place.

Inserting or removing elements from the beginning of a list can be costly (`O(n)`), as other elements shift.

2.4.2 2.2 List Slicing

Slicing allows you to extract a subsequence of a list:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7]

# Basic slice: list[start:end:step]
sub_list = numbers[2:6]       # [2, 3, 4, 5]
step_slice = numbers[1:7:2]   # [1, 3, 5]
reverse_slice = numbers[::-1] # [7, 6, 5, 4, 3, 2, 1, 0]

print("sub_list:", sub_list)
print("step_slice:", step_slice)
print("reverse_slice:", reverse_slice)
```

```
sub_list: [2, 3, 4, 5]
step_slice: [1, 3, 5]
reverse_slice: [7, 6, 5, 4, 3, 2, 1, 0]
```

> **Note:** For large-scale data, slicing can help avoid copying the entire list, especially if you're using libraries like NumPy that implement slicing via views.

2.4.3 2.3 Advanced Considerations: List Comprehensions

A **list comprehension** is a concise way to create lists:

```
squares = [x**2 for x in range(5)]
print(squares)

squares2 = []
for x in range(5):
    squares2.append(x**2)
print(squares2)
```

```
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16]
```

List comprehensions are often more Pythonic and can be faster than building lists with a standard `for` loop in many cases.

2.5 3. Tuples

2.5.1 3.1 Basic Usage

A **tuple** is an **immutable** ordered sequence, typically created using parentheses `( )`. Similar to lists, but you **cannot** modify them in-place.

```
coords = (3.2, 4.1)
print(coords[0])  # 3.2

lcoords = [3.2, 4.1]
print(lcoords[0])

lcoords[0] = 100
print(lcoords[0])

# Tuples are immutable, so this assignment would raise a TypeError:
# coords[0] = 100
```

Trying to modify a tuple element will raise a `TypeError`:

2.5.2 3.2 Performance and Immutability

Tuples can sometimes be more memory-efficient and faster to process than lists, especially for **read-only** data.

2.6 4. Dictionaries

2.6.1 4.1 Basic Usage

A **dictionary** in Python is a **mutable** mapping of **key-value** pairs, created using curly braces `{ }` or the `dict()` constructor:

```
person = {
    "name": "Alice",
    "age": 29,
    "department": "Physics"
}

print(person["name"])  # Accessing values by key

person["age"] = 30      # Modifying values
person

print("shool" in person)
print(person.keys())
print(person.values())
print(person.items())
```

```
Alice
False
dict_keys(['name', 'age', 'department'])
```

(continues on next page)

(continued from previous page)

```
dict_values(['Alice', 30, 'Physics'])
dict_items([('name', 'Alice'), ('age', 30), ('department', 'Physics')])
```

2.6.2 4.2 Key/Value Access and Methods

- Check existence: `if "department" in person:`
- Add/Update: `person["title"] = "Research Associate"`
- Remove entries: `person.pop("title")` or `del person["age"]`

Methods:

- `keys()`, `values()`, `items()`

2.6.3 4.3 Dictionary Comprehensions

Similar to list comprehensions, you can create a dictionary using **dictionary comprehensions**:

```
squares_dict = {x: x**2 for x in range(5)}
squares_dict
```

2.6.4 4.4 Use Cases

1. **Metadata**: Store experiment parameters.
2. **Lookup Tables**: Fast mapping from IDs or keys to values.
3. **Sparse Data**: Represent data that only has a few non-zero entries.

2.7 5. Nested Data Structures

You can nest these containers for more complex data:

```
lab_data = {
    "experiment1": {
        "timestamp": "2025-01-01",
        "results": [1.2, 2.3, 3.4],
    },
    "experiment2": {
        "timestamp": "2025-01-02",
        "results": [2.1, 4.5, 9.0],
    }
}

print(lab_data["experiment1"]["results"][0])  # 1.2
```

```
1.2
```

2.8 6. Performance Considerations

2.8.1 6.1 Lists vs. Tuples

- **Lists**: Good for data that you need to mutate frequently.
- **Tuples**: Lighter and faster if the data is static.

2.8.2 6.2 Large-Scale Data

- Consider **NumPy**, **pandas**, or **SciPy** for specialized or large-scale data.

2.8.3 6.3 Dictionary Optimizations

- **Hash tables** can degrade if collisions become frequent.
- For read-only dictionaries, consider `types.MappingProxyType`.
- `collections.OrderedDict` can preserve insertion order if needed (though normal dicts do in 3.7+).

2.9 7. Best Practices

1. **Start with Built-ins**.
2. **Document** nested containers.
3. **Validate** that keys exist and list lengths match.
4. **Immutable Where Possible**.
5. **Leverage Comprehensions**.
6. **Time Complexity** awareness.

2.9.1 Summary

Python's **lists**, **tuples**, and **dictionaries** form the backbone of most data structures in Python. Understanding their nuances, especially in research or large-scale code, is essential.

2.10 5. Sets

A **set** is an unordered collection of unique elements. Sets are useful for:

- Removing duplicates from a list
- Fast membership testing (O(1) lookup)
- Mathematical set operations (union, intersection, difference)

```
## Creating a set
unique_numbers = {1, 2, 3, 2, 1}  # {1, 2, 3} - duplicates removed!
empty_set = set()  # Note: {} creates an empty dict, not a set

## From a list
numbers = [1, 2, 2, 3, 3, 3]
unique = set(numbers)  # {1, 2, 3}
```

```
# Creating sets
particles = {'electron', 'proton', 'neutron', 'electron'}  # duplicate removed
print(f"Particles: {particles}")
print(f"Number of unique particles: {len(particles)}")

# Fast membership testing
print(f"Is 'proton' in the set? {'proton' in particles}")
print(f"Is 'muon' in the set? {'muon' in particles}")
```

```
Particles: {'proton', 'electron', 'neutron'}
Number of unique particles: 3
Is 'proton' in the set? True
Is 'muon' in the set? False
```

2.10.1 5.1 Set Operations

Sets support mathematical operations:

- **Union** (|): Elements in either set
- **Intersection** (&): Elements in both sets
- **Difference** (−): Elements in first but not second
- **Symmetric Difference** (^): Elements in either but not both

```
# Set operations
fermions = {'electron', 'proton', 'neutron', 'quark'}
leptons = {'electron', 'muon', 'tau', 'neutrino'}

print(f"Fermions: {fermions}")
print(f"Leptons: {leptons}")
print(f"Union (all particles): {fermions | leptons}")
print(f"Intersection (both categories): {fermions & leptons}")
print(f"Fermions but not leptons: {fermions - leptons}")
```

```
Fermions: {'proton', 'electron', 'quark', 'neutron'}
Leptons: {'tau', 'electron', 'neutrino', 'muon'}
Union (all particles): {'tau', 'muon', 'electron', 'quark', 'neutron', 'proton',
 ↪'neutrino'}
Intersection (both categories): {'electron'}
Fermions but not leptons: {'proton', 'neutron', 'quark'}
```

2.10.2 5.2 Common Use Case: Removing Duplicates

```
# Removing duplicates from experimental data
measurements = [1.2, 3.4, 1.2, 5.6, 3.4, 7.8, 1.2]

print(list(set(measurements)))

unique_measurements = list(set(measurements))
print(f"Original: {measurements}")
print(f"Unique values: {unique_measurements}")

# Note: set() doesn't preserve order. Use dict.fromkeys() to preserve order:
ordered_unique = list(dict.fromkeys(measurements))
print(f"Unique (order preserved): {ordered_unique}")
```

```
[1.2, 3.4, 5.6, 7.8]
Original: [1.2, 3.4, 1.2, 5.6, 3.4, 7.8, 1.2]
Unique values: [1.2, 3.4, 5.6, 7.8]
Unique (order preserved): [1.2, 3.4, 5.6, 7.8]
```

2.11 III. File Input/Output

Working with files is essential for any scientific programming. You'll need to:

- Read experimental data from files
- Save results for later analysis
- Process large datasets that don't fit in your code

Python makes file operations straightforward.

2.11.1 1. Opening and Closing Files

The basic pattern for working with files:

```
## Open a file
file = open('filename.txt', 'mode')

## Do something with the file
content = file.read()

## Always close when done!
file.close()
```

File modes:

- `'r'` — Read (default). File must exist.
- `'w'` — Write. Creates new file or **overwrites** existing!
- `'a'` — Append. Adds to end of file.
- `'r+'` — Read and write.

2.11.2 2. The `with` Statement (Context Manager)

Always use `with` when working with files! It automatically closes the file, even if an error occurs:

```
with open('filename.txt', 'r') as file:
    content = file.read()
    # File is automatically closed when we exit this block
```

This is the **recommended way** to handle files in Python.

2.11.3 3. Reading Files

```
# First, let's create a sample file to work with
sample_data = """Temperature Data
Day 1: 23.5 C
Day 2: 24.1 C
Day 3: 22.8 C
Day 4: 25.0 C
Day 5: 23.9 C
"""

with open('temperature.txt', 'w') as f:
    f.write(sample_data)

print("Created temperature.txt")
```

```
Created temperature.txt
```

Reading Methods

- `read()` — Read entire file as one string
- `readline()` — Read one line at a time
- `readlines()` — Read all lines into a list

```
# Method 1: read() - entire file as string
with open('temperature.txt', 'r') as f:
    content = f.read()
print("=== Using read() ===")
print(content)

# Method 2: readlines() - list of lines
with open('temperature.txt', 'r') as f:
    lines = f.readlines()
print("=== Using readlines() ===")
print(lines)
print (len(lines))
```

```
=== Using read() ===
Temperature Data
Day 1: 23.5 C
Day 2: 24.1 C
Day 3: 22.8 C
```

(continues on next page)

(continued from previous page)

```
Day 4: 25.0 C
Day 5: 23.9 C

=== Using readlines() ===
['Temperature Data\n', 'Day 1: 23.5 C\n', 'Day 2: 24.1 C\n', 'Day 3: 22.8 C\n',
 ↪'Day 4: 25.0 C\n', 'Day 5: 23.9 C\n']
6
```

Iterating Over Lines (Most Common Pattern)

```
# Best practice: iterate directly over the file
with open('temperature.txt', 'r') as f:
    for line_number, line in enumerate(f, 1):
        print(f"Line {line_number}: {line.strip()}")
```

```
Line 1: Temperature Data
Line 2: Day 1: 23.5 C
Line 3: Day 2: 24.1 C
Line 4: Day 3: 22.8 C
Line 5: Day 4: 25.0 C
Line 6: Day 5: 23.9 C
```

2.11.4 4. Writing Files

```
# Writing to a file
results = [
    ("Experiment 1", 3.14159),
    ("Experiment 2", 2.71828),
    ("Experiment 3", 1.41421),
]

with open('results.txt', 'w') as f:
    f.write("Experimental Results\n")  # write() for strings
    f.write("=" * 30 + "\n")
    for name, value in results:
        f.write(f"{name}: {value:.4f}\n")

print("Wrote results.txt")

# Verify by reading it back
with open('results.txt', 'r') as f:
    print(f.read())
```

```
Wrote results.txt
Experimental Results
==============================
Experiment 1: 3.1416
Experiment 2: 2.7183
Experiment 3: 1.4142
```

Appending to Files

Use mode `'a'` to add to the end of a file without erasing existing content:

```
# Append more results
with open('results.txt', 'a') as f:
    f.write("Experiment 4: 1.73205\n")
    f.write("Experiment 5: 2.23607\n")

# Check the updated file
with open('results.txt', 'r') as f:
    print(f.read())
```

2.11.5 5. Reading Numerical Data with NumPy

For scientific data, `numpy.loadtxt()` and `numpy.savetxt()` are more convenient:

```
import numpy as np

# Create a numerical data file
data = """# Time (s), Position (m), Velocity (m/s)
0.0, 0.0, 10.0
1.0, 9.5, 9.0
2.0, 18.0, 8.0
3.0, 25.5, 7.0
4.0, 32.0, 6.0
"""

with open('motion_data.csv', 'w') as f:
    f.write(data)

# Read with numpy (skip header row, comma delimiter)
time, position, velocity = np.loadtxt('motion_data.csv',
                                      delimiter=',',
                                      skiprows=1,
                                      unpack=True)

print(f"Time: {time}")
print(f"Position: {position}")
print(f"Velocity: {velocity}")
```

Saving Numerical Data

```
# Calculate acceleration and save results
acceleration = np.gradient(velocity, time)

# Stack columns and save
output_data = np.column_stack((time, position, velocity, acceleration))
np.savetxt('motion_analysis.csv',
           output_data,
           delimiter=',',
           header='Time,Position,Velocity,Acceleration',
           fmt='%.4f',
           comments='')
```

(continues on next page)

(continued from previous page)

```
print("Saved motion_analysis.csv")

# Verify
with open('motion_analysis.csv', 'r') as f:
    print(f.read())
```

2.11.6 File I/O Summary

Task	Method
Open file safely	`with open(filename, mode) as f:`
Read entire file	`f.read()`
Read lines	`for line in f:` or `f.readlines()`
Write string	`f.write(string)`
Read numerical data	`np.loadtxt(filename, delimiter=',')`
Save numerical data	`np.savetxt(filename, data, delimiter=',')`

2.12 IV. Error Handling with try/except

Errors happen. Users enter invalid input. Files don't exist. Networks fail. Good programs **handle errors gracefully** instead of crashing.

Python uses **exceptions** to handle errors. When something goes wrong, Python "raises" an exception. We can "catch" these exceptions and respond appropriately.

2.12.1 1. Common Python Exceptions

Exception	Cause
`ValueError`	Wrong value type (e.g., `int("hello")`)
`TypeError`	Wrong type for operation (e.g., `"5" + 3`)
`ZeroDivisionError`	Division by zero
`FileNotFoundError`	File doesn't exist
`IndexError`	List index out of range
`KeyError`	Dictionary key doesn't exist

2.12.2 2. Basic try/except

```
try:
    # Code that might cause an error
    risky_operation()
except SomeException:
    # Code to run if that error occurs
    handle_error()
```

```
x = 0
if x != 0:
    result = 10 / x
    print(result)
else:
    print("Cannot divide by zero.")
```

```
# Without error handling - this would crash!
# result = 10 / 0  # ZeroDivisionError

# With error handling
try:
    result = 10 / 0
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")

print("Program continues running...")
```

```
Error: Cannot divide by zero!
Program continues running...
```

2.12.3 3. Handling Multiple Exception Types

```
def safe_divide(a, b):
    """Safely divide two numbers with error handling."""
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        print("Error: Division by zero!")
        return None
    except TypeError:
        print("Error: Invalid types for division!")
        return None

# Test cases
print(f"10 / 2 = {safe_divide(10, 2)}")
print(f"10 / 0 = {safe_divide(10, 0)}")
print(f"'10' / 2 = {safe_divide('10', 2)}")
```

```
10 / 2 = 5.0
Error: Division by zero!
10 / 0 = None
Error: Invalid types for division!
'10' / 2 = None
```

2.12.4 4. The `else` and `finally` Clauses

- **`else`**: Runs if NO exception occurred
- **`finally`**: ALWAYS runs, whether or not an exception occurred

```
try:
    risky_operation()
except SomeException:
    handle_error()
else:
    # Runs only if no exception
    success_actions()
finally:
    # Always runs
    cleanup()
```

```
def read_data_file(filename):
    """Read a data file with full error handling."""
    try:
        f = open(filename, 'r')
        data = f.read()
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found!")
        return None
    else:
        print(f"Successfully read {len(data)} characters")
        return data
    finally:
        print("Cleanup: Closing file (if open)")
        try:
            f.close()
        except:
            pass  # File was never opened

# Test with existing and non-existing files
print("=== Reading existing file ===")
result = read_data_file('temperature.txt')

print("\n=== Reading non-existing file ===")
result = read_data_file('nonexistent.txt')
```

```
=== Reading existing file ===
Successfully read 87 characters
Cleanup: Closing file (if open)

=== Reading non-existing file ===
Error: File 'nonexistent.txt' not found!
Cleanup: Closing file (if open)
```

2.12.5 5. Practical Example: Input Validation

A common use case is validating user input:

```
def get_positive_number(prompt):
    """Keep asking until user enters a valid positive number."""
    while True:
        try:
            value = float(input(prompt))
            if value <= 0:
                print("Please enter a positive number!")
                continue
            return value
        except ValueError:
            print("Invalid input! Please enter a number.")

# Uncomment to test interactively in a live notebook:
# mass = get_positive_number("Enter mass (kg): ")
# print(f"You entered: {mass} kg")
```

2.12.6 6. Physics Example: Robust Calculation Function

```
import math

def calculate_orbital_period(semi_major_axis, central_mass):
    """
    Calculate orbital period using Kepler's third law.
    T = 2π * sqrt(a³ / (G * M))

    Parameters:
        semi_major_axis: in meters
        central_mass: in kg
    Returns:
        Period in seconds, or None if invalid input
    """
    G = 6.674e-11  # gravitational constant

    try:
        # Validate inputs
        if semi_major_axis <= 0:
            raise ValueError("Semi-major axis must be positive")
        if central_mass <= 0:
            raise ValueError("Central mass must be positive")

        # Calculate
        period = 2 * math.pi * math.sqrt(semi_major_axis**3 / (G * central_mass))
        return period

    except ValueError as e:
        print(f"Invalid input: {e}")
        return None
    except TypeError as e:
        print(f"Type error: {e}")
        return None

# Earth's orbit around the Sun
```

(continues on next page)

(continued from previous page)

```
a_earth = 1.496e11  # meters
M_sun = 1.989e30    # kg

T = calculate_orbital_period(a_earth, M_sun)
if T:
    print(f"Earth's orbital period: {T:.2e} seconds")
    print(f"That's {T / (24*3600):.1f} days")

# Test error handling
print("\nTesting with invalid input:")
calculate_orbital_period(-1e11, M_sun)  # negative distance
calculate_orbital_period(a_earth, 0)     # zero mass
```

```
Earth's orbital period: 3.16e+07 seconds
That's 365.2 days

Testing with invalid input:
Invalid input: Semi-major axis must be positive
Invalid input: Central mass must be positive
```

```
import math

def calculate_orbital_period(semi_major_axis, central_mass):
    """
    Calculate orbital period using Kepler's third law.
    T = 2π * sqrt(a³ / (G * M))

    Parameters:
        semi_major_axis: in meters
        central_mass: in kg
    Returns:
        Period in seconds, or None if invalid input
    """
    if semi_major_axis <= 0 or central_mass <= 0:
        return None

    G = 6.674e-11  # gravitational constant
    period = 2 * math.pi * math.sqrt(semi_major_axis**3 / (G * central_mass))
    return period

a_earth = 1.496e11  # meters
M_sun = 1.989e30    # kg

T = calculate_orbital_period(a_earth, M_sun)
print(T)
```

2.12.7 7. Raising Your Own Exceptions

Use `raise` to signal errors in your own code:

```
def set_temperature(kelvin):
    """Set temperature, ensuring it's physically valid."""
    if kelvin < 0:
        raise ValueError(f"Temperature {kelvin}K is below absolute zero!")
    print(f"Temperature set to {kelvin}K ({kelvin - 273.15:.1f}°C)")

# Valid temperature
set_temperature(300)

# Invalid temperature - this will raise an exception
try:
    set_temperature(-50)
except ValueError as e:
    print(f"Caught error: {e}")
```

2.12.8 Error Handling Summary

Keyword	Purpose
`try`	Code that might raise an exception
`except`	Handle specific exception types
`else`	Run if no exception occurred
`finally`	Always run (cleanup)
`raise`	Raise your own exception

Best Practices:

- Catch specific exceptions, not bare `except:`
- Use `finally` for cleanup (closing files, connections)
- Validate inputs early with meaningful error messages
- Don't silently ignore errors — at least log them

2.13 V. Summary & Next Steps

In this lecture, we covered:

2.13.1 Functions

- Defining functions with `def`
- Parameters: positional, keyword, default
- Return values and docstrings
- Scope and namespaces

2.13.2 Data Structures

- **Lists**: Ordered, mutable, indexed
- **Tuples**: Ordered, immutable, faster
- **Dictionaries**: Key-value pairs, fast lookup
- **Sets**: Unique elements, set operations

2.13.3 File I/O

- Reading and writing text files
- The `with` statement for safe file handling
- NumPy's `loadtxt` and `savetxt` for numerical data

2.13.4 Error Handling

- `try/except` for catching errors
- `else` and `finally` clauses
- Raising exceptions with `raise`

2.13.5 Next Lecture: Python Basics III

- NumPy arrays in depth
- Advanced plotting with Matplotlib
- Introduction to numerical methods

2.14 VI. Practice Exercises

2.14.1 Exercise 1: Function Practice

Write a function `quadratic_formula(a, b, c)` that:

- Takes coefficients a, b, c of a quadratic equation $ax^2 + bx + c = 0$
- Returns both roots (as a tuple)
- Handles the case where the discriminant is negative (return `None`)

- Includes a docstring

2.14.2 Exercise 2: Data Structure Practice

Create a dictionary called `element_data` that stores:

- Element symbols as keys
- A dictionary with 'name', 'atomic_number', and 'mass' as values
- Include at least 5 elements
- Write a function that takes an element symbol and prints its properties

2.14.3 Exercise 3: File I/O Practice

Write a program that:

1. Creates a file with 10 random numbers (one per line)
2. Reads the file back
3. Calculates and prints the mean and standard deviation
4. Saves the statistics to a new file

2.14.4 Exercise 4: Error Handling Practice

Write a function `safe_sqrt(x)` that:

- Returns the square root of x
- Handles negative numbers gracefully (print warning, return None)
- Handles non-numeric input (print error, return None)
- Works correctly for valid positive numbers

```
# Your solutions here

# Exercise 1: quadratic_formula


# Exercise 2: element_data


# Exercise 3: File I/O


# Exercise 4: safe_sqrt
```

2.15 VII. Advanced Topics (Optional Reading)

The following sections cover more advanced Python features. These are useful for writing sophisticated code but are not essential for the core computational physics content.

- **Variable Arguments** (`*args`, `**kwargs`)
- **Iterators and Generators**
- **Decorators**

Feel free to explore these topics if you're comfortable with the main material.

2.15.1 Variable Arguments: `*args` and `**kwargs`

Sometimes you want a function to accept any number of arguments:

- `*args`: Collects extra positional arguments as a tuple
- `**kwargs`: Collects extra keyword arguments as a dictionary

```
def flexible_function(*args, **kwargs):
    """A function that accepts any arguments."""
    print(f"Positional args: {args}")
    print(f"Keyword args: {kwargs}")

flexible_function(1, 2, 3, name="physics", value=42)
```

2.15.2 Generators

Generators are functions that produce a sequence of values lazily (one at a time), using `yield` instead of `return`. They're memory-efficient for large sequences.

```
def fibonacci(n):
    """Generate first n Fibonacci numbers."""
    a, b = 0, 1
    count = 0
    while count < n:
        yield a  # Return value and pause
        a, b = b, a + b
        count += 1

# Use the generator
print("First 10 Fibonacci numbers:")
for num in fibonacci(10):
    print(num, end=" ")
```

2.15.3 Decorators

Decorators wrap functions to add extra behavior without modifying the original function.

```
import time

def timing_decorator(func):
    """Measure how long a function takes to run."""
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f} seconds")
        return result
    return wrapper

@timing_decorator
def slow_function():
    """A function that takes some time."""
    total = sum(i**2 for i in range(100000))
    return total

result = slow_function()
```

CHAPTER

THREE

PYTHON BASICS III

Welcome to **Python Basics III** — the final Python foundations lecture before numerical methods!

In this notebook, we'll cover:

1. **NumPy Arrays In-Depth** — The foundation of scientific computing in Python
2. **Advanced Matplotlib** — Creating publication-quality figures
3. **Vectorization & Performance** — Writing fast numerical code
4. **Functions as Arguments** — Preparing for numerical methods
5. **Physics Applications** — Putting it all together

By the end of this lecture, you'll be ready to implement numerical integration, differentiation, and other computational methods.

3.1 I. NumPy Arrays In-Depth

NumPy (Numerical Python) is the foundation of scientific computing in Python. The **ndarray** (n-dimensional array) is its core data structure.

Why NumPy?

- **Fast**: Operations are implemented in C, much faster than Python loops
- **Convenient**: Mathematical operations work element-wise
- **Memory efficient**: Stores data in contiguous memory blocks
- **Universal**: Used by virtually all scientific Python libraries

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

# Check version
print(f"NumPy version: {np.__version__}")
```

```
NumPy version: 2.0.2
```

3.1.1 1. Creating Arrays

There are many ways to create NumPy arrays:

```
# From a Python list
a = np.array([1, 2, 3, 4, 5], float)
print(f"From list: {a}")
print(f"Type: {type(a)}")
print(f"Shape: {a.shape}")
print(f"Data type: {a.dtype}")
```

```
From list: [1. 2. 3. 4. 5.]
Type: <class 'numpy.ndarray'>
Shape: (5,)
Data type: float64
```

```
# Zeros, ones, and empty arrays
zeros = np.zeros(5)          # 5 zeros
ones = np.ones(5)            # 5 ones
full = np.full(5, 3.14)      # 5 copies of 3.14

print(f"Zeros: {zeros}")
print(f"Ones: {ones}")
print(f"Full: {full}")
```

```
Zeros: [0. 0. 0. 0. 0.]
Ones: [1. 1. 1. 1. 1.]
Full: [3.14 3.14 3.14 3.14 3.14]
```

```
# linspace and arange - VERY important for physics!

# linspace: N evenly spaced points between start and end (inclusive)
x1 = np.linspace(0, 10, 5)   # 5 points from 0 to 10
print(f"linspace(0, 10, 5): {x1}")

# arange: points with fixed step (like range, but for floats)
x2 = np.arange(0, 10, 2)     # start=0, stop=10, step=2
print(f"arange(0, 10, 2): {x2}")

### range(0,10)

# Common use: time array for simulations
t = np.linspace(0, 1, 101)   # 0 to 1 second, 101 points (dt = 0.01)
print(f"\nTime array: {t[:10]}... (first 10 of {len(t)} points)")
```

```
linspace(0, 10, 5): [ 0.   2.5  5.   7.5 10. ]
arange(0, 10, 2): [0 2 4 6 8]

Time array: [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09]... (first 10 of␣
↪101 points)
```

```
# Random arrays - useful for Monte Carlo simulations
np.random.seed(3)  # For reproducibility

uniform = np.random.random(5)           # Uniform [0, 1)
```

(continues on next page)

(continued from previous page)

```
normal = np.random.randn(5)              # Standard normal (mean=0, std=1)
integers = np.random.randint(1, 10, 5)   # Random integers [1, 10)

print(f"Uniform [0,1): {uniform}")
print(f"Normal: {normal}")
print(f"Integers [1,10): {integers}")
```

```
Uniform [0,1): [0.5507979  0.70814782 0.29090474 0.51082761 0.89294695]
Normal: [-0.46004212 -0.05792084  2.07757414 -0.60131248  0.93923639]
Integers [1,10): [3 2 4 6 9]
```

3.1.2 2. Multi-dimensional Arrays

NumPy arrays can have any number of dimensions. 2D arrays are like matrices.

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

print("Matrix:")
print(matrix)
print(f"Shape: {matrix.shape}")  # (rows, columns)
print(f"Dimensions: {matrix.ndim}")
print(f"Total elements: {matrix.size}")
```

```
Matrix:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Shape: (3, 3)
Dimensions: 2
Total elements: 9
```

```
# Creating 2D arrays
zeros_2d = np.zeros((3, 4))        # 3 rows, 4 columns
ones_2d = np.ones((2, 5))
identity = np.eye(3)               # 3x3 identity matrix

print("Zeros (3x4):")
print(zeros_2d)

print (ones_2d)

print("\nIdentity (3x3):")
print(identity)
```

```
Zeros (3x4):
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

(continues on next page)

(continued from previous page)

```
Identity (3x3):
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

3.1.3 3. Array Operations

NumPy operations work **element-wise** by default. This is incredibly powerful!

```
# Element-wise operations
a = np.array([1, 2, 3, 4, 5])
b = np.array([10, 20, 30, 40, 50])

print(f"a = {a}")
print(f"b = {b}")
print(f"a + b = {a + b}")     # Element-wise addition
print(f"a * b = {a * b}")     # Element-wise multiplication
print(f"a ** 2 = {a ** 2}")   # Square each element
print(f"b / a = {b / a}")     # Element-wise division


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

matrix2 = np.array([[10, 20, 3],
                   [4, 10, 6],
                   [4, 1, 9]])

print (matrix * matrix2)
```

```
a = [1 2 3 4 5]
b = [10 20 30 40 50]
a + b = [11 22 33 44 55]
a * b = [ 10  40  90 160 250]
a ** 2 = [ 1  4  9 16 25]
b / a = [10. 10. 10. 10. 10.]
[[10 40  9]
 [16 50 36]
 [28  8 81]]
```

Broadcasting

NumPy can operate on arrays of different shapes through **broadcasting**:

```
# Scalar + array: scalar is "broadcast" to match array shape
a = np.array([1, 2, 3, 4, 5])

print(f"a = {a}")
print(f"a + 10 = {a + 10}")      # Add 10 to each element
print(f"a * 2 = {a * 2}")        # Multiply each by 2
print(f"a / 2 = {a / 2}")        # Divide each by 2
```

(continues on next page)

(continued from previous page)

```
# Physics example: convert Celsius to Fahrenheit
celsius = np.array([0, 20, 37, 100])
fahrenheit = celsius * 9/5 + 32
print(f"\nCelsius: {celsius}")
print(f"Fahrenheit: {fahrenheit}")

a_list = [1, 2, 3, 4, 5]
print (a_list)
print (a_list + [10])
```

```
a = [1 2 3 4 5]
a + 10 = [11 12 13 14 15]
a * 2 = [ 2  4  6  8 10]
a / 2 = [0.5 1.  1.5 2.  2.5]

Celsius: [  0  20  37 100]
Fahrenheit: [ 32.   68.   98.6 212. ]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 10]
```

Universal Functions (ufuncs)

NumPy provides fast mathematical functions that operate on arrays:

```
        x = np.linspace(0, 2*np.pi, 5)
        print(f"x = {x}")

        # NumPy functions work element-wise on arrays.
        print(f"sin(x) = {np.sin(x)}")
        print(f"cos(x) = {np.cos(x)}")

        # Exponential and logarithm
        y = np.array([1, 2, 3])
        print(f"
y = {y}")
        print(f"exp(y) = {np.exp(y)}")
        print(f"log(y) = {np.log(y)}")
        print(f"sqrt(y) = {np.sqrt(y)}")

        import math

        print(np.sin(x))
        # The standard-library math module expects scalar inputs:
        print(math.sin(float(x[0])))
```

3.1.4 4. Indexing and Slicing

Accessing elements in NumPy arrays is similar to lists, but more powerful.

```
# 1D indexing
a = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90])
print(f"a = {a}")

print(f"a[0] = {a[0]}")         # First element
print(f"a[-1] = {a[-1]}")       # Last element
print(f"a[2:5] = {a[2:5]}")     # Elements 2, 3, 4
print(f"a[::2] = {a[::2]}")     # Every other element
print(f"a[::-1] = {a[::-1]}")   # Reversed
```

```
a = [10 20 30 40 50 60 70 80 90]
a[0] = 10
a[-1] = 90
a[2:5] = [30 40 50]
a[::2] = [10 30 50 70 90]
a[::-1] = [90 80 70 60 50 40 30 20 10]
```

```
# 2D indexing
matrix = np.array([[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]])

print("Matrix:")
print(matrix)

print(f"\nElement [1, 2]: {matrix[1, 2]}")  # Row 1, Column 2 → 7
print(f"Row 0: {matrix[0, :]}")             # First row
print(f"Column 1: {matrix[:, 1]}")          # Second column
print(f"Submatrix [0:2, 1:3]:\n{matrix[0:2, 1:3]}")
```

```
Matrix:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

Element [1, 2]: 7
Row 0: [1 2 3 4]
Column 1: [ 2  6 10]
Submatrix [0:2, 1:3]:
[[2 3]
 [6 7]]
```

Boolean Indexing (Filtering)

Select elements based on conditions — extremely useful for data analysis!

```
# Boolean indexing
data = np.array([1, -2, 3, -4, 5, -6, 7, -8, 9])
print(f"data = {data}")

positive_data = []
for i in range(len(data)):
    if data[i] > 0:
        positive_data.append(data[i])
positive_data = np.array(positive_data)

# Create boolean mask
positive_mask = data > 0
print(f"positive_mask = {positive_mask}")

# Use mask to filter
positive_values = data[positive_mask]
print(f"Positive values: {positive_values}")

# One-liner
print(f"Values > 3: {data[data > 3]}")
print(f"Even values: {data[data % 2 == 0]}")
```

```
data = [ 1 -2  3 -4  5 -6  7 -8  9]
positive_mask = [ True False  True False  True False  True False  True]
Positive values: [1 3 5 7 9]
Values > 3: [5 7 9]
Even values: [-2 -4 -6 -8]
```

```
# Physics example: filter measurements
measurements = np.array([1.2, 3.5, -0.1, 2.8, 5.1, 0.3, -0.5, 4.2])
uncertainties = np.array([0.1, 0.2, 0.1, 0.3, 0.2, 0.1, 0.2, 0.1])

# Keep only positive measurements with uncertainty < 0.2
mask = (measurements > 0) & (uncertainties < 0.2)
clean_data = measurements[mask]
# clean_data = measurements[(measurements > 0) & (uncertainties < 0.2)]
clean_errors = uncertainties[mask]

print(f"Original: {measurements}")
print(f"Filtered: {clean_data}")
```

```
Original: [ 1.2  3.5 -0.1  2.8  5.1  0.3 -0.5  4.2]
Filtered: [1.2 0.3 4.2]
```

3.1.5 5. Array Methods and Functions

NumPy provides many useful functions for analyzing data:

```
data = np.array([2.3, 4.5, 1.2, 6.7, 3.4, 5.6, 2.1])
print(f"data = {data}")

# Basic statistics
print(f"\nSum: {np.sum(data)}")
print(f"Mean: {np.mean(data):.3f}")
print(f"Std Dev: {np.std(data):.3f}")
print(f"Min: {np.min(data)}, Max: {np.max(data)}")

# Index of min/max
print(f"Index of min: {np.argmin(data)} (value: {data[np.argmin(data)]})")
print(f"Index of max: {np.argmax(data)} (value: {data[np.argmax(data)]})")
```

```
data = [2.3 4.5 1.2 6.7 3.4 5.6 2.1]

Sum: 25.799999999999997
Mean: 3.686
Std Dev: 1.856
Min: 1.2, Max: 6.7
Index of min: 2 (value: 1.2)
Index of max: 3 (value: 6.7)
```

```
# Useful for physics computations
x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([0, 1, 4, 9, 16, 25])  # y = x²

# Cumulative sum (useful for integration)
print(f"x = {x}")
print(f"cumsum(x) = {np.cumsum(x)}")

# Difference (useful for derivatives)
print(f"\ny = {y}")
print(f"diff(y) = {np.diff(y)}")  # [1-0, 4-1, 9-4, 16-9, 25-16] = [1, 3, 5, 7, 9]
```

```
x = [0 1 2 3 4 5]
cumsum(x) = [ 0  1  3  6 10 15]

y = [ 0  1  4  9 16 25]
diff(y) = [1 3 5 7 9]
```

3.1.6 6. Reshaping Arrays

```
# Reshaping
a = np.arange(12)  # [0, 1, 2, ..., 11]
print(f"Original: {a}")
print(f"Shape: {a.shape}")

# Reshape to 3x4
b = a.reshape(3, 4)
print(f"\nReshaped to (3, 4):")
```

(continues on next page)

(continued from previous page)

```
print(b)

# Reshape to 2x6
c = a.reshape(2, 6)
print(f"\nReshaped to (2, 6):")
print(c)

# Flatten back to 1D
d = c.flatten()
print(f"\nFlattened: {d}")
```

```
Original: [ 0  1  2  3  4  5  6  7  8  9 10 11]
Shape: (12,)

Reshaped to (3, 4):
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

Reshaped to (2, 6):
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]

Flattened: [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

3.1.7 7. Combining Arrays

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

# Concatenate (join end-to-end)
print(f"concatenate: {np.concatenate([a, b])}")

# Stack vertically and horizontally
print(f"\nvstack (vertical):")
print(np.vstack([a, b]))

print(f"\nhstack (horizontal): {np.hstack([a, b])}")

# column_stack - useful for saving data
x = np.array([1, 2, 3])
y = np.array([10, 20, 30])
data = np.column_stack([x, y])
print(f"\ncolumn_stack:")
print(data)
```

```
concatenate: [1 2 3 4 5 6]

vstack (vertical):
[[1 2 3]
 [4 5 6]]

hstack (horizontal): [1 2 3 4 5 6]
```

(continues on next page)

(continued from previous page)

```
column_stack:
[[ 1 10]
 [ 2 20]
 [ 3 30]]
```

3.2 II. Advanced Matplotlib

You've already used basic Matplotlib plotting. Now let's learn to create **publication-quality figures**.

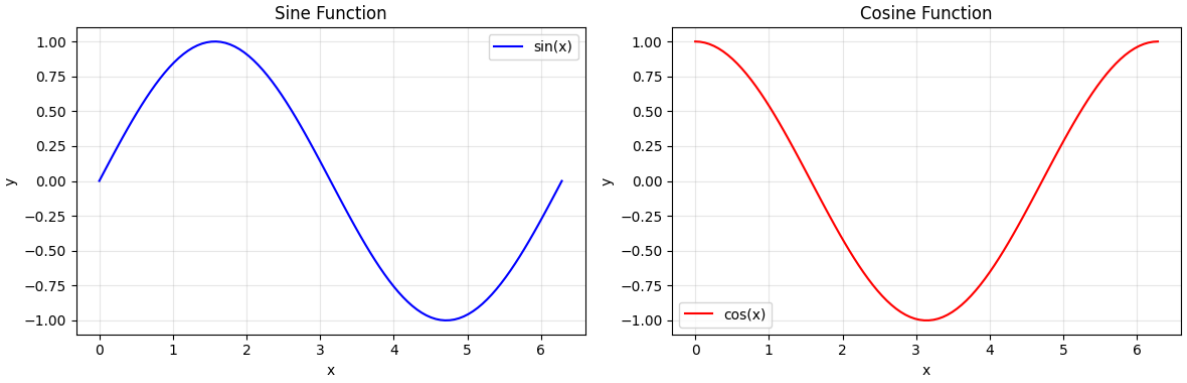
Key skills:

- Multiple subplots
- Different plot types
- Customization and styling
- Saving figures

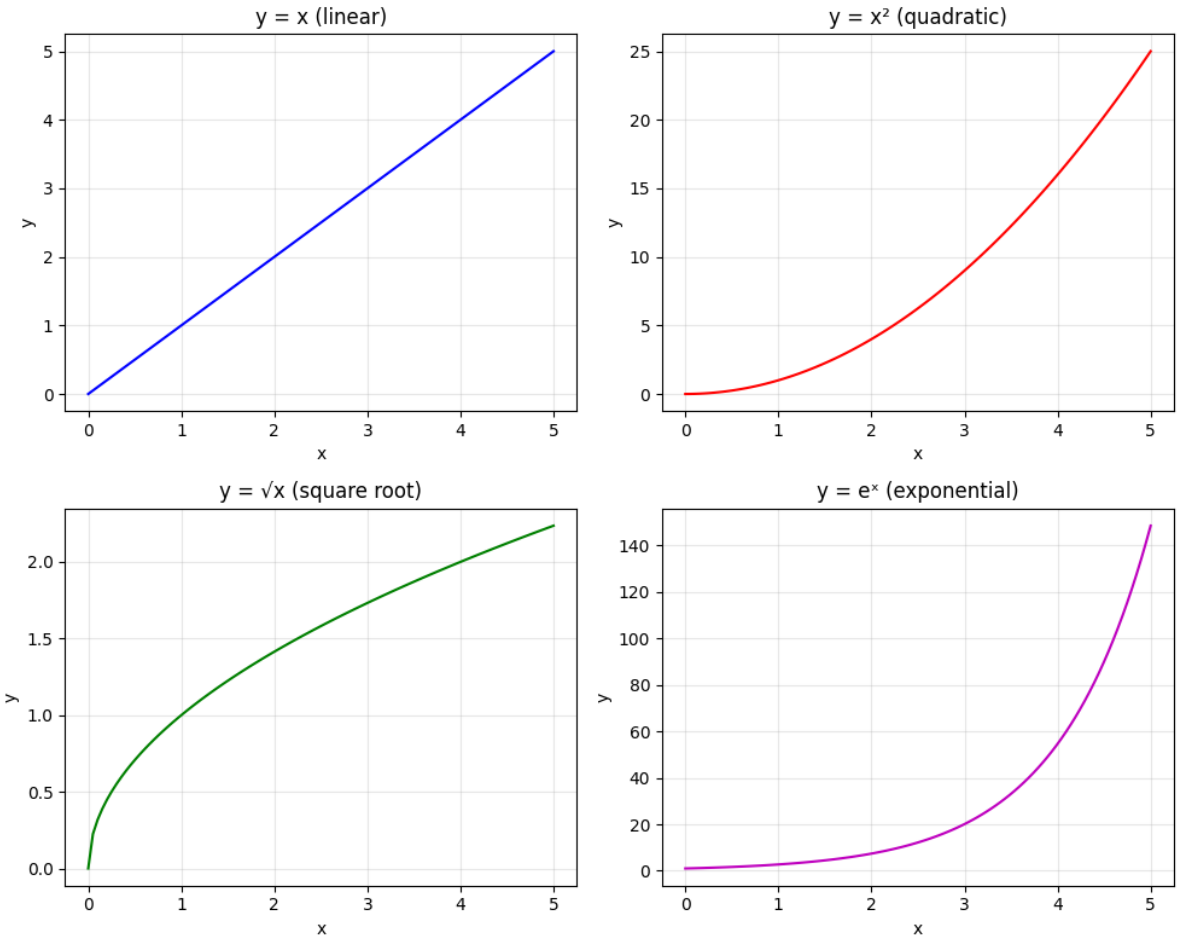
3.2.1 1. Subplots

Use `plt.subplots()` to create multiple plots in one figure:

```
# Create 1 row, 2 columns of subplots
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

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

# First subplot
axes[0].plot(x, np.sin(x), 'b-', label='sin(x)')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].set_title('Sine Function')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Second subplot
axes[1].plot(x, np.cos(x), 'r-', label='cos(x)')
axes[1].set_xlabel('x')
axes[1].set_ylabel('y')
axes[1].set_title('Cosine Function')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()  # Prevent overlap
plt.show()
```

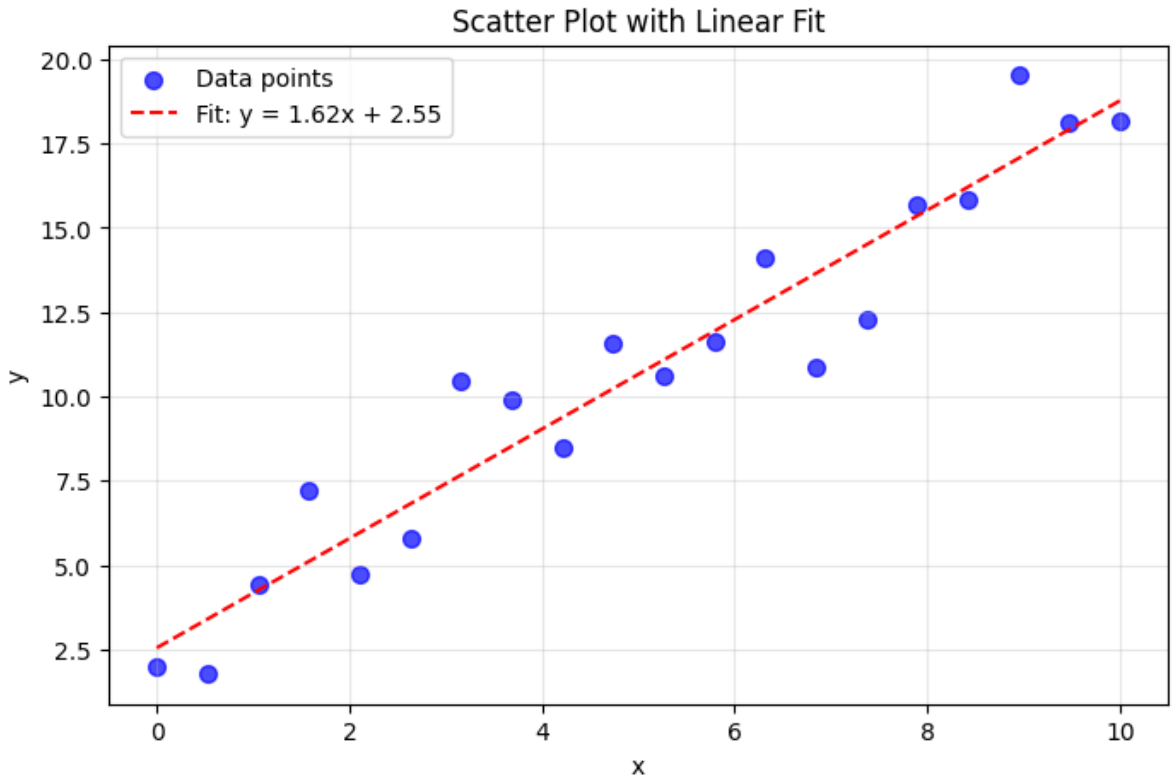



```
# 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

x = np.linspace(0, 5, 100)

# Access with [row, col]
axes[0, 0].plot(x, x, 'b-')
axes[0, 0].set_title('y = x (linear)')

axes[0, 1].plot(x, x**2, 'r-')
axes[0, 1].set_title('y = x² (quadratic)')

axes[1, 0].plot(x, np.sqrt(x), 'g-')
axes[1, 0].set_title('y = √x (square root)')

axes[1, 1].plot(x, np.exp(x), 'm-')
axes[1, 1].set_title('y = eˣ (exponential)')

# Add labels to all
for ax in axes.flat:
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.grid(True, alpha=0.3)

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

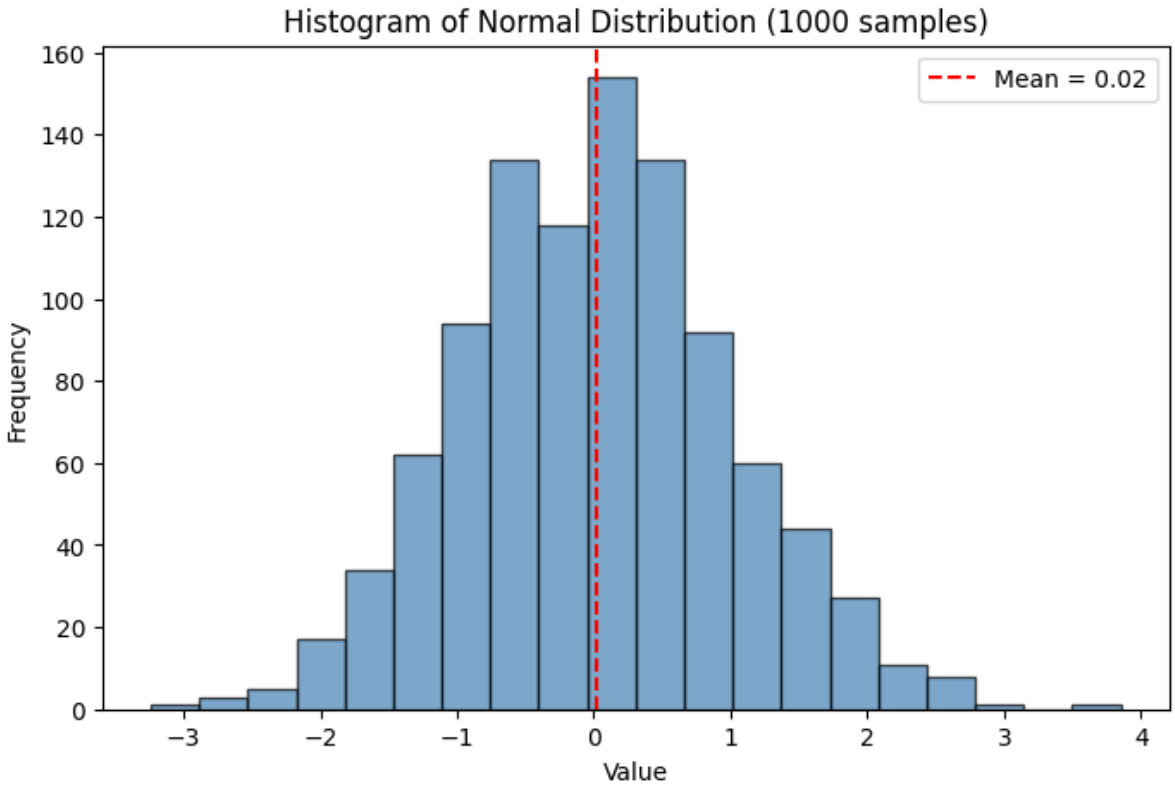


3.2.2 2. Different Plot Types

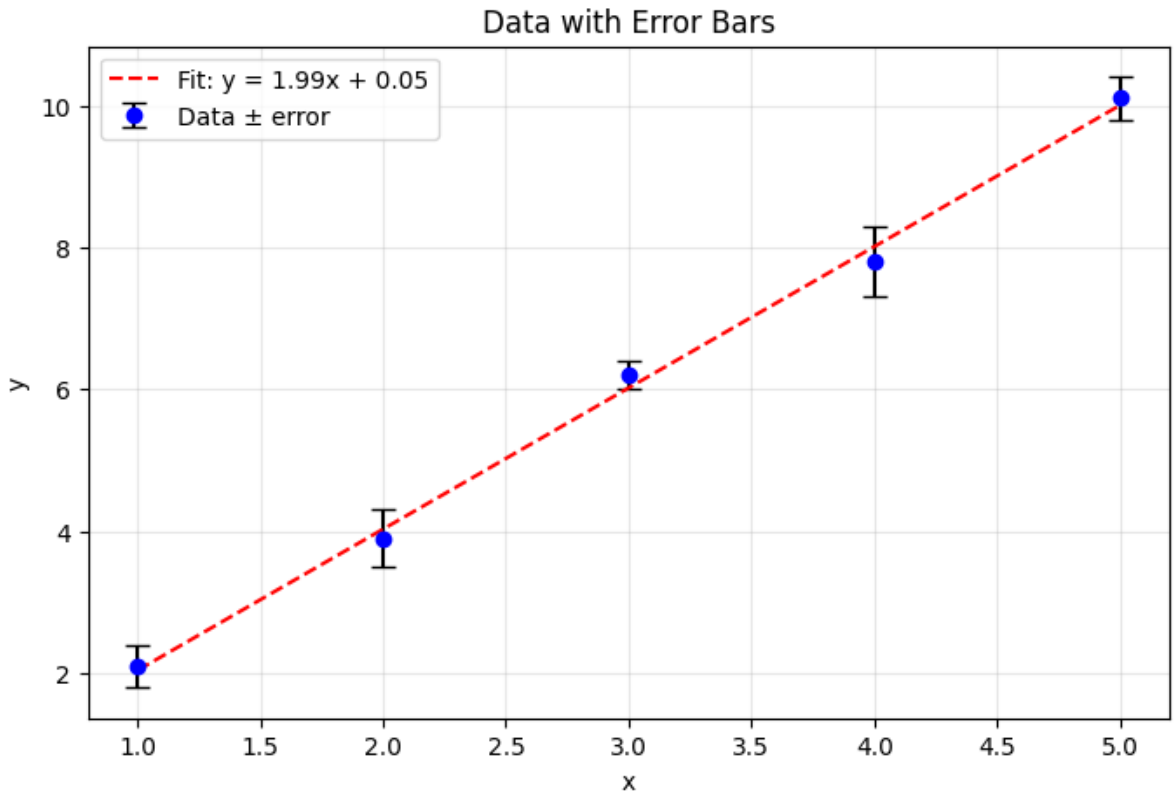
```
# Scatter plot - good for experimental data
np.random.seed(42)
x = np.linspace(0, 10, 20)
y = 2 * x + 1 + np.random.randn(20) * 2  # Linear with noise

plt.figure(figsize=(8, 5))
plt.scatter(x, y, c='blue', s=50, alpha=0.7, label='Data points')

# Add a fit line
coeffs = np.polyfit(x, y, 1)  # Linear fit
fit_line = np.poly1d(coeffs)
plt.plot(x, fit_line(x), 'r--', label=f'Fit: y = {coeffs[0]:.2f}x + {coeffs[1]:.2f}')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Scatter Plot with Linear Fit')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```



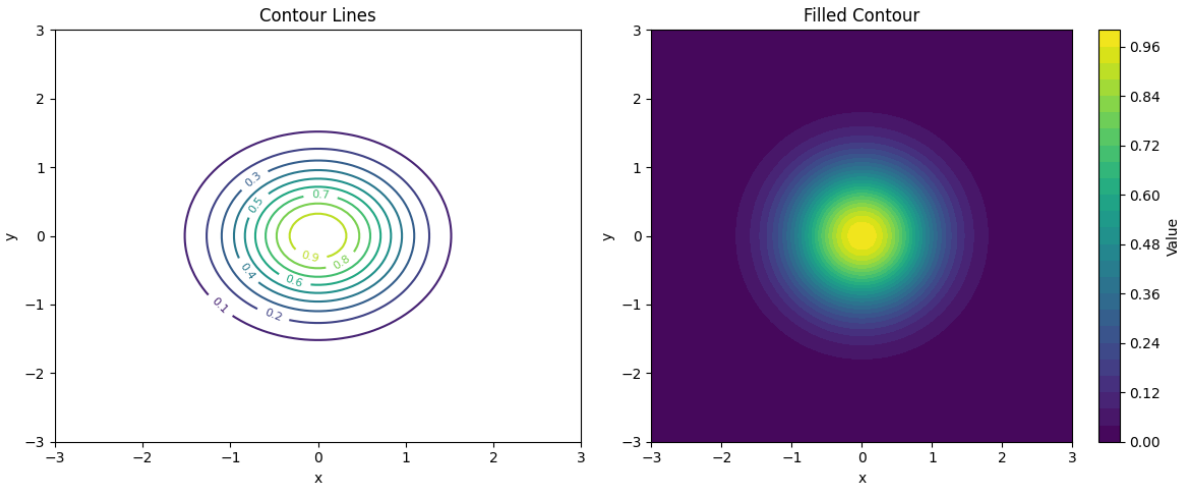

```
# Histogram - distribution of data
np.random.seed(42)
data = np.random.randn(1000)  # 1000 samples from normal distribution

plt.figure(figsize=(8, 5))
plt.hist(data, bins=20, color='steelblue', edgecolor='black', alpha=0.7)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Normal Distribution (1000 samples)')

# Add mean line
plt.axvline(np.mean(data), color='red', linestyle='--',
            label=f'Mean = {np.mean(data):.2f}')
plt.legend()
plt.show()
```




```
# Error bars - essential for experimental physics!
x = np.array([1, 2, 3, 4, 5])
y = np.array([2.1, 3.9, 6.2, 7.8, 10.1])
yerr = np.array([0.3, 0.4, 0.2, 0.5, 0.3])  # Uncertainties

plt.figure(figsize=(8, 5))
plt.errorbar(x, y, yerr=yerr, fmt='o', capsize=5,
             color='blue', ecolor='black', label='Data ± error')

# Fit line
coeffs = np.polyfit(x, y, 1)
plt.plot(x, np.poly1d(coeffs)(x), 'r--', label=f'Fit: y = {coeffs[0]:.2f}x +
 ↪{coeffs[1]:.2f}')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Data with Error Bars')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```



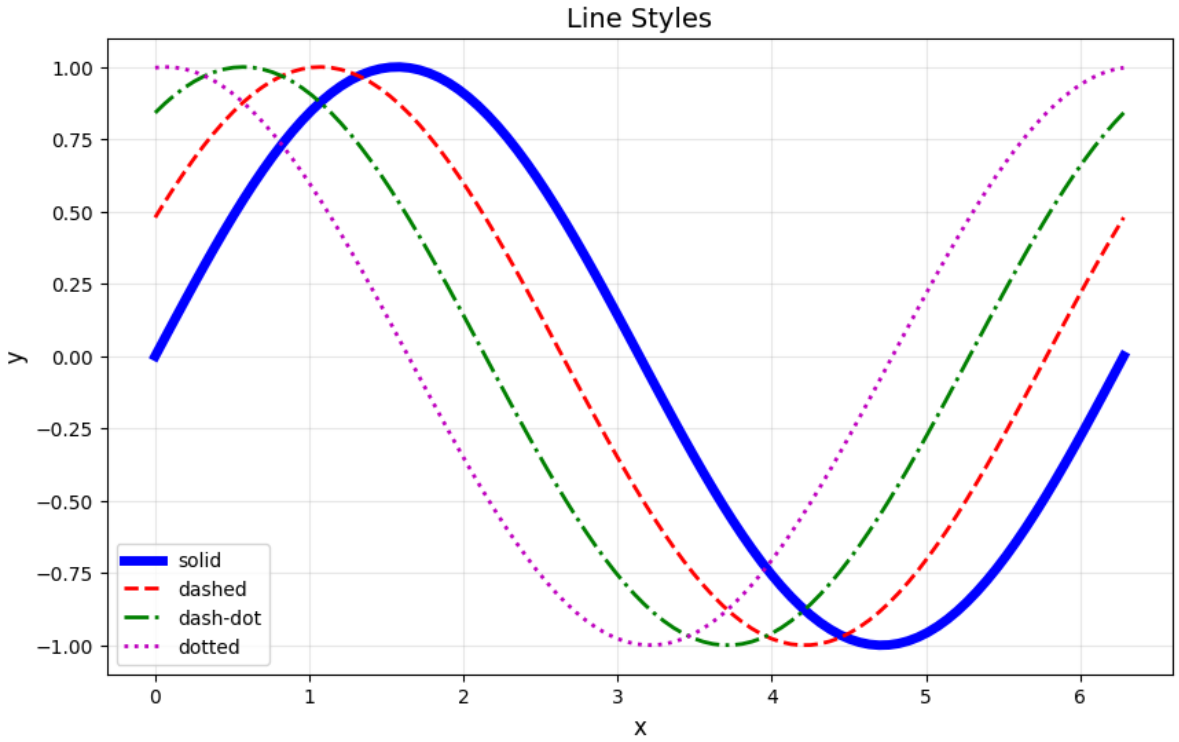

```
# Contour plot - for 2D data (like potential fields)
x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)
X, Y = np.meshgrid(x, y)  # Create 2D grid

# 2D Gaussian
Z = np.exp(-(X**2 + Y**2))

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Contour lines
cs1 = axes[0].contour(X, Y, Z, levels=10, cmap='viridis')
axes[0].clabel(cs1, inline=True, fontsize=8)
axes[0].set_title('Contour Lines')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')

# Filled contour (heatmap)
cs2 = axes[1].contourf(X, Y, Z, levels=30, cmap='viridis')
plt.colorbar(cs2, ax=axes[1], label='Value')
axes[1].set_title('Filled Contour')
axes[1].set_xlabel('x')
axes[1].set_ylabel('y')

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



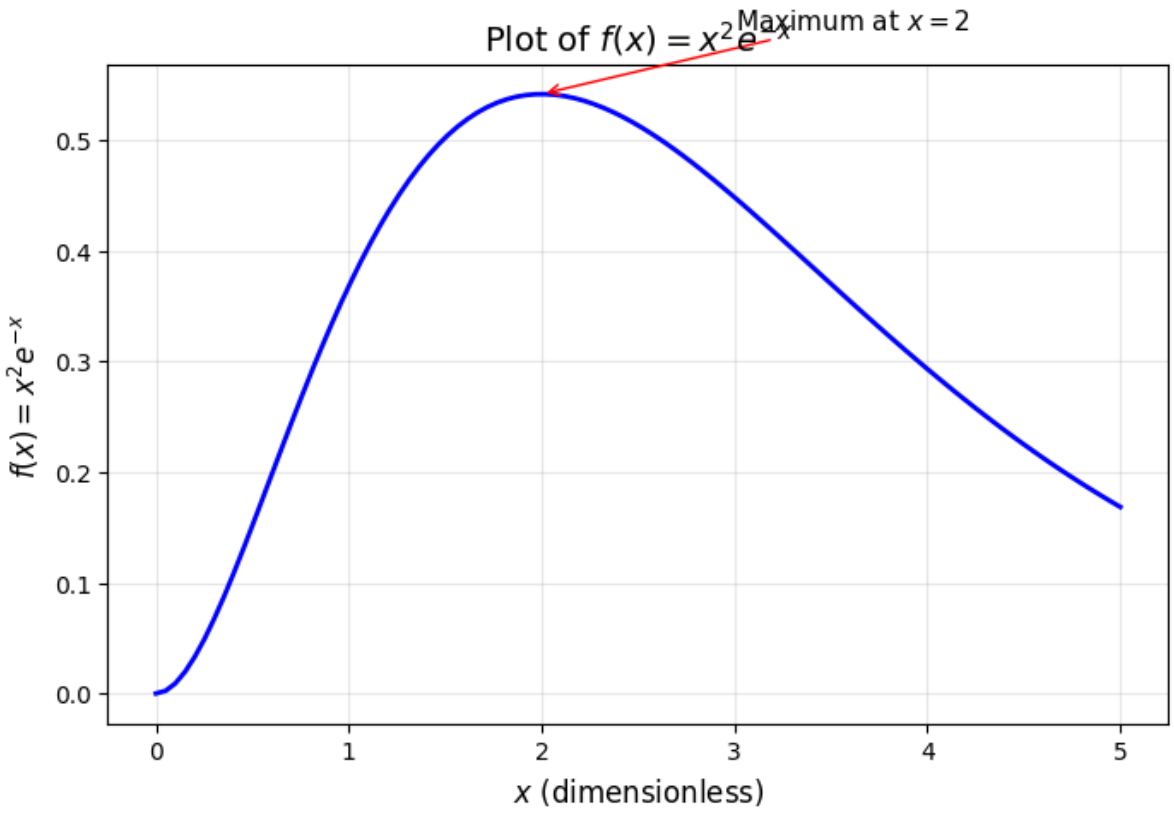
3.2.3 3. Customization

Make your plots publication-ready:

```
# Line styles and colors
x = np.linspace(0, 2*np.pi, 100)

plt.figure(figsize=(10, 6))

# Different line styles
plt.plot(x, np.sin(x), 'b-', linewidth=5, label='solid')
plt.plot(x, np.sin(x + 0.5), 'r--', linewidth=2, label='dashed')
plt.plot(x, np.sin(x + 1.0), 'g-.', linewidth=2, label='dash-dot')
plt.plot(x, np.sin(x + 1.5), 'm:', linewidth=2, label='dotted')

plt.xlabel('x', fontsize=12)
plt.ylabel('y', fontsize=12)
plt.title('Line Styles', fontsize=14)
plt.legend(fontsize=10)
plt.grid(True, alpha=0.3)
plt.show()
```




```
# LaTeX in labels - for proper mathematical notation
x = np.linspace(0, 5, 100)
y = x**2 * np.exp(-x)

plt.figure(figsize=(8, 5))
plt.plot(x, y, 'b-', linewidth=2)

# Use LaTeX notation with r'...' (raw string) and $...$
plt.xlabel(r'$x$ (dimensionless)', fontsize=12)
plt.ylabel(r'$f(x) = x^2 e^{-x}$', fontsize=12)
plt.title(r'Plot of $f(x) = x^2 e^{-x}$', fontsize=14)

# Add annotation with LaTeX
plt.annotate(r'Maximum at $x = 2$',
             xy=(2, 4*np.exp(-2)),
             xytext=(3, 0.6),
             fontsize=11,
             arrowprops=dict(arrowstyle='->', color='red'))

plt.grid(True, alpha=0.3)
plt.show()
```

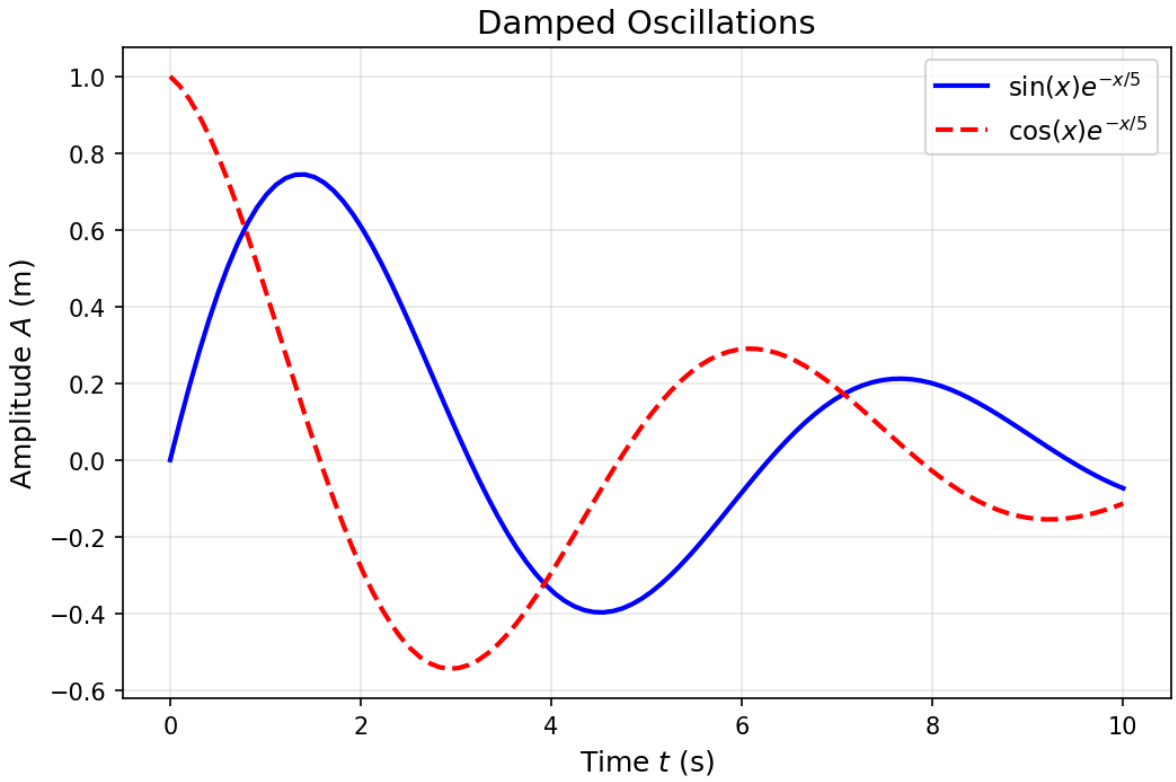


3.2.4 4. Saving Figures

Save your plots for papers, presentations, and reports:

```
# Create a publication-quality figure
x = np.linspace(0, 10, 100)
y1 = np.sin(x) * np.exp(-x/5)
y2 = np.cos(x) * np.exp(-x/5)

fig, ax = plt.subplots(figsize=(8, 5), dpi=150)  # High DPI for quality

ax.plot(x, y1, 'b-', linewidth=2, label=r'$\sin(x) e^{-x/5}$')
ax.plot(x, y2, 'r--', linewidth=2, label=r'$\cos(x) e^{-x/5}$')

ax.set_xlabel(r'Time $t$ (s)', fontsize=12)
ax.set_ylabel(r'Amplitude $A$ (m)', fontsize=12)
ax.set_title('Damped Oscillations', fontsize=14)
ax.legend(fontsize=11)
ax.grid(True, alpha=0.3)

# Save in different formats
fig.savefig('damped_oscillation.png', dpi=300, bbox_inches='tight')
fig.savefig('damped_oscillation.pdf', bbox_inches='tight')  # Vector format

print("Saved: damped_oscillation.png and damped_oscillation.pdf")
plt.show()
```

```
Saved: damped_oscillation.png and damped_oscillation.pdf
```



3.3 III. Vectorization & Performance

One of the most important skills in scientific Python is writing **vectorized** code. This means using NumPy operations instead of Python loops.

Why vectorization matters:

- NumPy operations are implemented in C — 10-100x faster than Python loops
- Code is shorter and more readable
- Essential for large-scale simulations

3.3.1 1. Loop vs Vectorized: Speed Comparison

```
import time

# Task: compute sin(x) for 1 million points
n = 1000000
x = np.linspace(0, 2*np.pi, n)

# Method 1: Python loop (SLOW)
start = time.time()
y_loop = []
```

(continues on next page)

(continued from previous page)

```
for val in x:
    y_loop.append(np.sin(val))
y_loop = np.array(y_loop)
loop_time = time.time() - start
print(f"Python loop: {loop_time:.4f} seconds")

# Method 2: Vectorized (FAST)
start = time.time()
y_vec = np.sin(x)
vec_time = time.time() - start
print(f"Vectorized:  {vec_time:.4f} seconds")

print(f"\nSpeedup: {loop_time/vec_time:.1f}x faster!")
```

```
Python loop: 1.1931 seconds
Vectorized:  0.0140 seconds

Speedup: 85.3x faster!
```

3.3.2 2. Vectorization Examples

Learn to think in arrays, not loops:

```
# Example 1: Compute kinetic energy for many particles
masses = np.array([1.0, 2.0, 0.5, 1.5, 3.0])      # kg
velocities = np.array([10, 5, 20, 8, 4])          # m/s

# Vectorized: all at once!
kinetic_energies = 0.5 * masses * velocities**2

print("Masses (kg):    ", masses)
print("Velocities (m/s):", velocities)
print("KE (J):         ", kinetic_energies)
print(f"Total KE: {np.sum(kinetic_energies):.1f} J")
```

```
Masses (kg):     [1.  2.  0.5 1.5 3. ]
Velocities (m/s): [10  5 20  8  4]
KE (J):          [ 50.  25. 100.  48.  24.]
Total KE: 247.0 J
```

```
# Example 2: Distance between all pairs of points
# (This would require nested loops without vectorization)

# 5 particles in 2D
x = np.array([0, 1, 2, 3, 4])
y = np.array([0, 1, 0, 1, 0])

# n = len(x)
# for i in range(5):
#     for j in range(5):
#         dx = x[i] - x[j]
#         dy = y[i] - y[j]
#         distances = np.sqrt(dx**2 + dy**2)
```

(continues on next page)

(continued from previous page)

```
# Compute pairwise distances using broadcasting
# We need to reshape to enable broadcasting
dx = x[:, np.newaxis] - x[np.newaxis, :]  # 5x5 matrix of x-differences
dy = y[:, np.newaxis] - y[np.newaxis, :]  # 5x5 matrix of y-differences
distances = np.sqrt(dx**2 + dy**2)

print("Particle positions:")
for i in range(len(x)):
    print(f"  Particle {i}: ({x[i]}, {y[i]})")

print("\nDistance matrix:")
print(np.round(distances, 2))
```

```
Particle positions:
  Particle 0: (0, 0)
  Particle 1: (1, 1)
  Particle 2: (2, 0)
  Particle 3: (3, 1)
  Particle 4: (4, 0)

Distance matrix:
[[0.   1.41 2.   3.16 4.  ]
 [1.41 0.   1.41 2.   3.16]
 [2.   1.41 0.   1.41 2.  ]
 [3.16 2.   1.41 0.   1.41]
 [4.   3.16 2.   1.41 0.  ]]
```

3.3.3 3. Common Vectorization Patterns

Loop Version	Vectorized Version
`for x in arr: result.append(f(x))`	`result = f(arr)`
`for i in range(n): total += arr[i]`	`total = np.sum(arr)`
`for i in range(n): if arr[i] > 0: ...`	`arr[arr > 0]`
`for i in range(n): arr[i] = arr[i] * 2`	`arr = arr * 2`

3.4 IV. Functions as Arguments

In Python, functions are **first-class objects** — they can be passed as arguments to other functions. This is essential for numerical methods!

For example, a numerical integrator needs to know **which function** to integrate. You pass the function itself as an argument.

3.4.1 1. Passing Functions to Functions

```
# A function that takes another function as an argument
def apply_twice(func, x):
    """Apply a function twice: func(func(x))"""
    return func(func(x))

def square(x):
    return x ** 2

def add_one(x):
    return x + 1

# Pass different functions
print(f"square applied twice to 2: {apply_twice(square, 2)}")    # (2²)² = 16
print(f"add_one applied twice to 5: {apply_twice(add_one, 5)}") # 5+1+1 = 7
print(f"np.sin applied twice to π/2: {apply_twice(np.sin, np.pi/2):.4f}")  #␣
↪sin(sin(π/2))
```

```
square applied twice to 2: 16
add_one applied twice to 5: 7
np.sin applied twice to π/2: 0.8415
```

3.4.2 2. Preview: Numerical Derivative

This is exactly how we'll implement numerical differentiation in the next lecture:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

```
def numerical_derivative(f, x, h=1e-5):
    """
    Compute the derivative of f at x using finite difference.

    Parameters:
        f: function to differentiate
        x: point at which to compute derivative
        h: step size (smaller = more accurate, but can cause numerical issues)

    Returns:
        Approximate derivative f'(x)
    """
    return (f(x + h) - f(x)) / h

# Test with known derivatives
```

(continues on next page)

(continued from previous page)

```
print("Testing numerical derivative:")
print(f"d/dx[sin(x)] at x=0: {numerical_derivative(np.sin, 0):.6f} (exact: {np.cos(0)}
↪)")
print(f"d/dx[x²] at x=3: {numerical_derivative(lambda x: x**2, 3):.6f} (exact: 6)")
print(f"d/dx[eˣ] at x=1: {numerical_derivative(np.exp, 1):.6f} (exact: {np.exp(1):.6f}
↪)")
```

```
Testing numerical derivative:
d/dx[sin(x)] at x=0: 1.000000 (exact: 1.0)
d/dx[x²] at x=3: 6.000010 (exact: 6)
d/dx[eˣ] at x=1: 2.718295 (exact: 2.718282)
```

3.4.3 3. Lambda Functions

Lambda functions are small, anonymous functions defined in one line. They're convenient when you need a simple function just once:

```
# Lambda syntax: lambda arguments: expression

# Instead of:
def square(x):
    return x ** 2

# You can write:
square_lambda = lambda x: x ** 2

print(f"square(5) = {square(5)}")
print(f"square_lambda(5) = {square_lambda(5)}")

# More examples
add = lambda x, y: x + y
print(f"add(3, 4) = {add(3, 4)}")

def addd(x,y):
    return x+y

# Often used inline:
print(f"Derivative of x³ at x=2: {numerical_derivative(lambda x: x**3, 2)}")
```

```
# Physics example: Compute derivatives of various potentials

# Harmonic oscillator: V(x) = 0.5 * k * x²
# Gravitational: V(r) = -G*M*m / r
# Lennard-Jones: V(r) = 4ε[(σ/r)¹² - (σ/r)⁶]

k = 1.0  # spring constant
harmonic = lambda x: 0.5 * k * x**2

# Force = -dV/dx
x = 2.0
force = -numerical_derivative(harmonic, x)
print(f"Harmonic potential at x={x}: V = {harmonic(x)}")
print(f"Force at x={x}: F = {force:.4f} (exact: {-k*x})")
```

```
Harmonic potential at x=2.0: V = 2.0
Force at x=2.0: F = -2.0000 (exact: -2.0)
```

3.5 V. Physics Applications

Let's put everything together with complete physics examples!

3.5.1 1. Projectile Motion

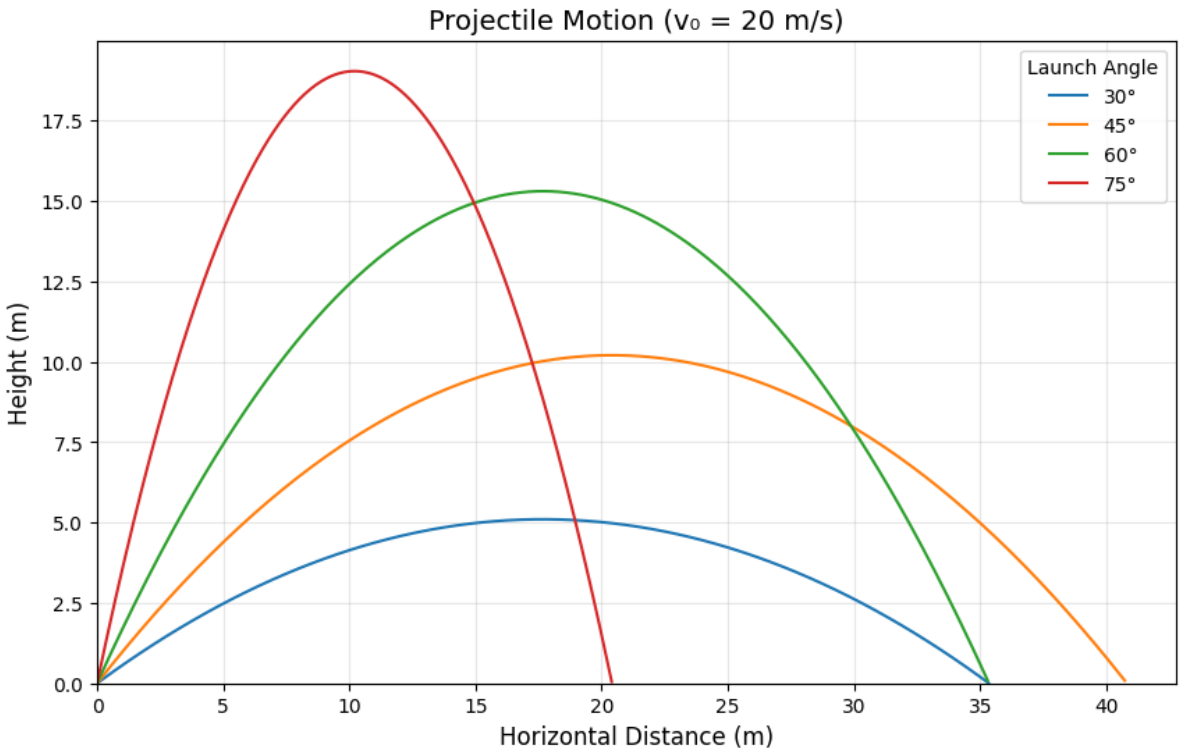
Simulate and visualize projectile motion with different launch angles:

```
def projectile_motion(v0, theta_deg, g=9.8, dt=0.01):
    """
    Simulate projectile motion.

    Parameters:
        v0: initial velocity (m/s)
        theta_deg: launch angle (degrees)
        g: gravitational acceleration (m/s²)
        dt: time step (s)

    Returns:
        t, x, y: time array, x positions, y positions
    """
    theta = np.radians(theta_deg)
    v0x = v0 * np.cos(theta)
    v0y = v0 * np.sin(theta)

    # Time of flight (when y = 0 again)
    t_flight = 2 * v0y / g

    # Create time array
    t = np.arange(0, t_flight, dt)

    # Compute trajectory (vectorized!)
    x = v0x * t
    y = v0y * t - 0.5 * g * t**2

    return t, x, y

# Compare different launch angles
v0 = 20  # m/s
angles = [30, 45, 60, 75]

plt.figure(figsize=(10, 6))

for angle in angles:
    t, x, y = projectile_motion(v0, angle)
    plt.plot(x, y, label=f'{angle}°')

plt.xlabel('Horizontal Distance (m)', fontsize=12)
plt.ylabel('Height (m)', fontsize=12)
plt.title(f'Projectile Motion (v₀ = {v0} m/s)', fontsize=14)
```

(continues on next page)

(continued from previous page)

```
plt.legend(title='Launch Angle')
plt.grid(True, alpha=0.3)
plt.xlim(0, None)
plt.ylim(0, None)
plt.show()
```



3.5.2 2. Simple Harmonic Oscillator

A preview of differential equations — the motion of a mass on a spring:

$$x(t) = A\cos(\omega t + \phi)$$

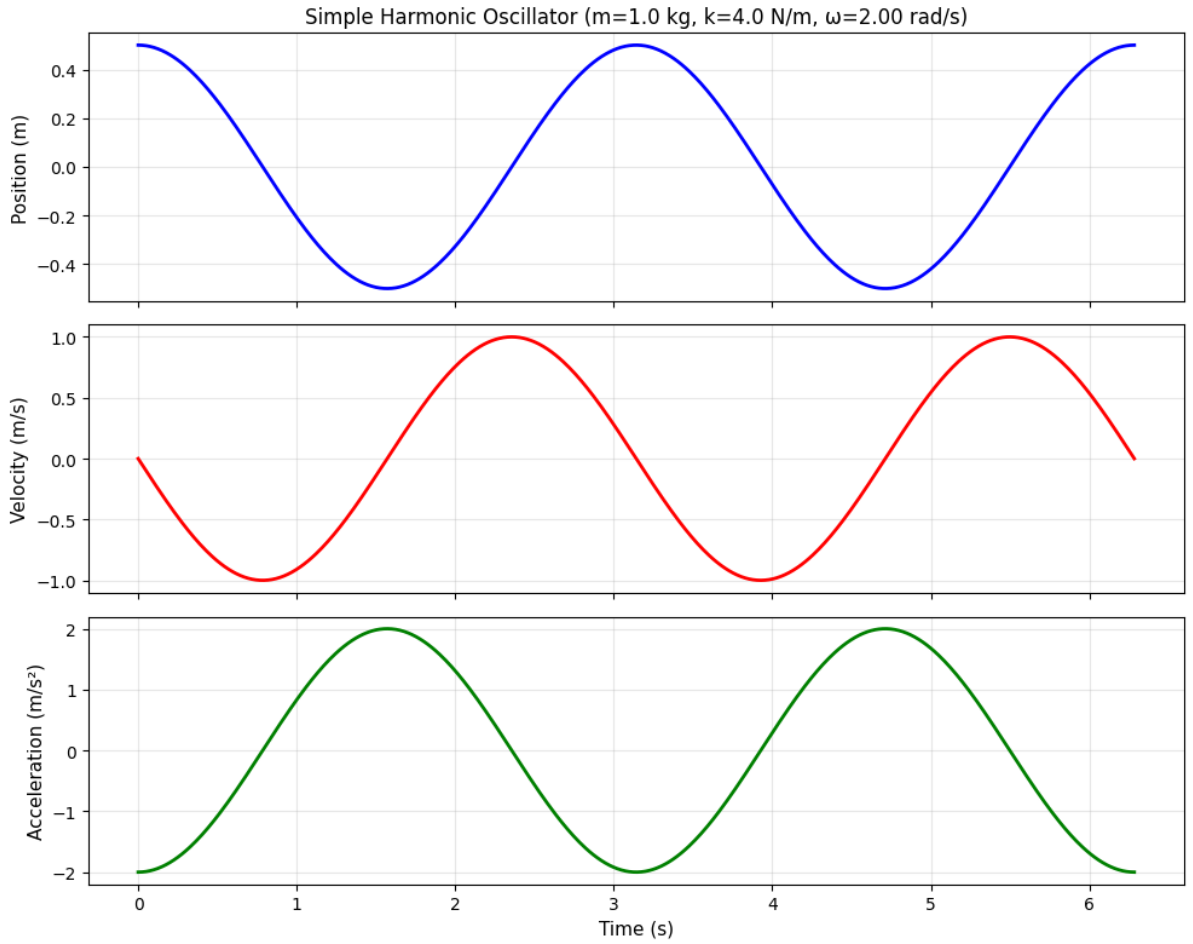
where $\omega = \sqrt{k/m}$

```
def harmonic_oscillator(t, A, omega, phi=0):
    """
    Simple harmonic oscillator solution.

    Parameters:
        t: time array
        A: amplitude
        omega: angular frequency (rad/s)
        phi: phase (rad)

    Returns:
        x: position, v: velocity, a: acceleration
    """
    x = A * np.cos(omega * t + phi)
```

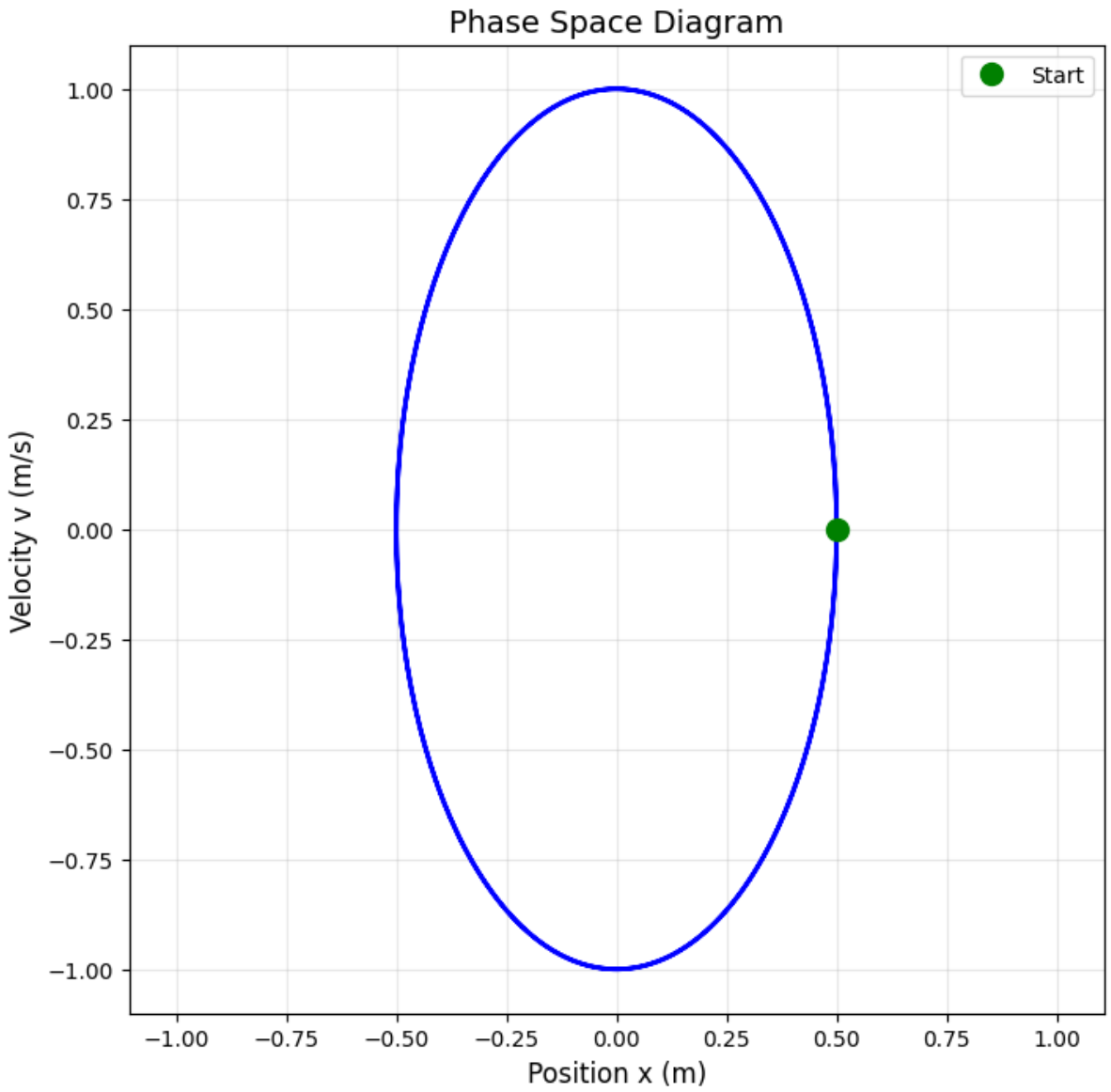
(continues on next page)

(continued from previous page)

```
    v = -A * omega * np.sin(omega * t + phi)
    a = -A * omega**2 * np.cos(omega * t + phi)
    return x, v, a

# Parameters
m = 1.0    # mass (kg)
k = 4.0    # spring constant (N/m)
A = 0.5    # amplitude (m)
omega = np.sqrt(k / m)  # angular frequency

t = np.linspace(0, 4*np.pi/omega, 500)  # 2 complete periods
x, v, a = harmonic_oscillator(t, A, omega)

# Visualization
fig, axes = plt.subplots(3, 1, figsize=(10, 8), sharex=True)

axes[0].plot(t, x, 'b-', linewidth=2)
axes[0].set_ylabel('Position (m)', fontsize=11)
axes[0].set_title(f'Simple Harmonic Oscillator (m={m} kg, k={k} N/m, ω={omega:.2f}␣
 ↪rad/s)', fontsize=12)
axes[0].grid(True, alpha=0.3)

axes[1].plot(t, v, 'r-', linewidth=2)
axes[1].set_ylabel('Velocity (m/s)', fontsize=11)
axes[1].grid(True, alpha=0.3)

axes[2].plot(t, a, 'g-', linewidth=2)
axes[2].set_ylabel('Acceleration (m/s²)', fontsize=11)
axes[2].set_xlabel('Time (s)', fontsize=11)
axes[2].grid(True, alpha=0.3)

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




```
# Phase space diagram (position vs velocity)
plt.figure(figsize=(8, 8))
plt.plot(x, v, 'b-', linewidth=2)
plt.xlabel('Position x (m)', fontsize=12)
plt.ylabel('Velocity v (m/s)', fontsize=12)
plt.title('Phase Space Diagram', fontsize=14)
plt.grid(True, alpha=0.3)
plt.axis('equal')

# Mark the starting point
plt.plot(x[0], v[0], 'go', markersize=10, label='Start')
plt.legend()
plt.show()
```

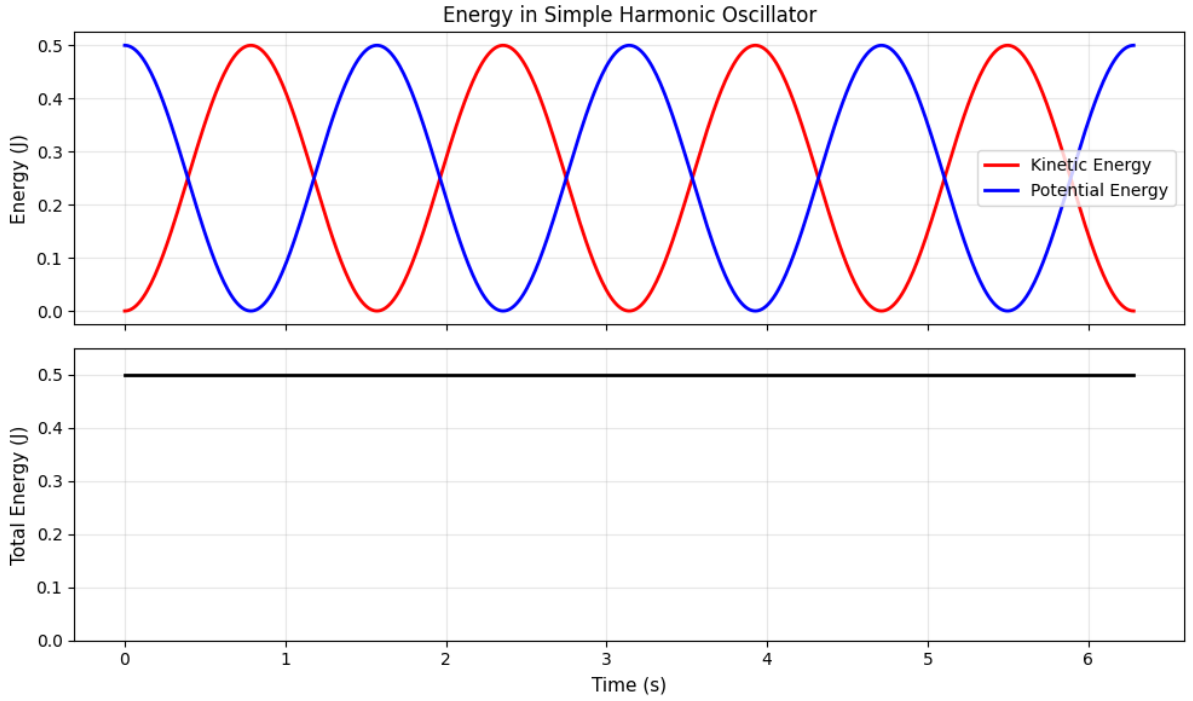



```
# Verify energy conservation
KE = 0.5 * m * v**2           # Kinetic energy
PE = 0.5 * k * x**2           # Potential energy
total_energy = KE + PE

fig, axes = plt.subplots(2, 1, figsize=(10, 6), sharex=True)

axes[0].plot(t, KE, 'r-', label='Kinetic Energy', linewidth=2)
axes[0].plot(t, PE, 'b-', label='Potential Energy', linewidth=2)
axes[0].set_ylabel('Energy (J)', fontsize=11)
axes[0].legend()
axes[0].grid(True, alpha=0.3)
axes[0].set_title('Energy in Simple Harmonic Oscillator', fontsize=12)

axes[1].plot(t, total_energy, 'k-', linewidth=2)
axes[1].set_ylabel('Total Energy (J)', fontsize=11)
axes[1].set_xlabel('Time (s)', fontsize=11)
```

(continues on next page)

(continued from previous page)

```
axes[1].set_ylim([0, 1.1*max(total_energy)])
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Total energy varies by: {(max(total_energy)-min(total_energy))/np.mean(total_
↪energy)*100:.2e}%")
print("(Should be constant - variation is due to numerical precision)")
```




```
Total energy varies by: 4.44e-14%
(Should be constant - variation is due to numerical precision)
```

3.5.3 3. Experimental Data Analysis

A complete workflow: load data, analyze, visualize, save results.

```
# Generate synthetic "experimental" data
np.random.seed(42)
n_points = 50

# True relationship: y = 2.5x + 1.0 with noise
x_data = np.linspace(0, 10, n_points)
y_true = 2.5 * x_data + 1.0
y_noise = np.random.normal(0, 2, n_points)
y_data = y_true + y_noise
y_errors = np.random.uniform(0.5, 2.0, n_points)  # Uncertainties

# Save to file
```

(continues on next page)

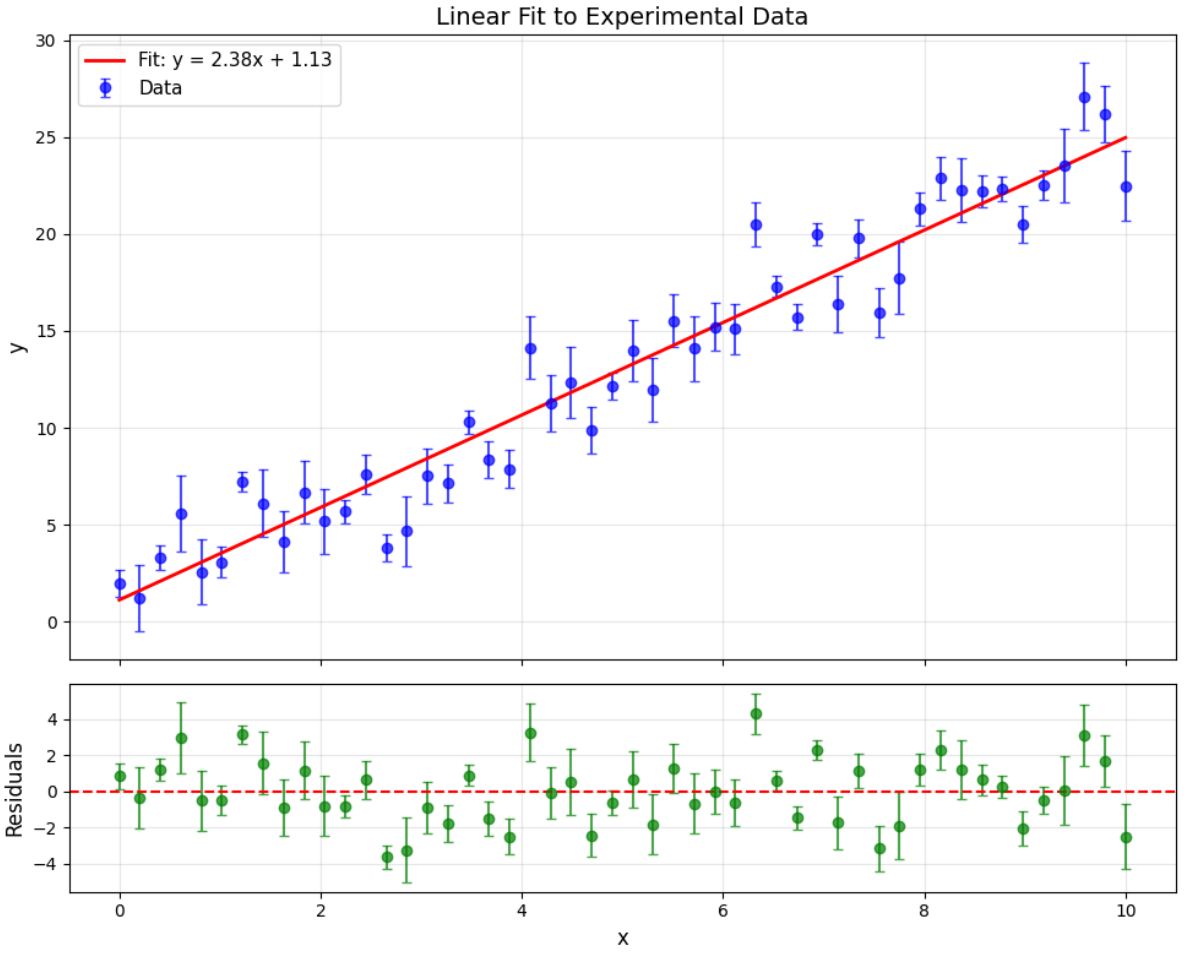
(continued from previous page)

```
np.savetxt('experiment_data.csv',
           np.column_stack([x_data, y_data, y_errors]),
           delimiter=',',
           header='x,y,y_error',
           comments='')
print("Saved: experiment_data.csv")
```

```
Saved: experiment_data.csv
```

```
# Load and analyze the data
data = np.loadtxt('experiment_data.csv', delimiter=',', skiprows=1)
x = data[:, 0]
y = data[:, 1]
yerr = data[:, 2]

# Fit a line
coeffs = np.polyfit(x, y, 1)
slope, intercept = coeffs
y_fit = np.poly1d(coeffs)(x)

# Statistics
residuals = y - y_fit
chi_squared = np.sum((residuals / yerr)**2)
dof = len(y) - 2  # degrees of freedom

print("=== Analysis Results ===")
print(f"Slope: {slope:.3f}")
print(f"Intercept: {intercept:.3f}")
print(f"χ²/dof: {chi_squared/dof:.2f}")

# Publication-quality plot
fig, axes = plt.subplots(2, 1, figsize=(10, 8),
                         gridspec_kw={'height_ratios': [3, 1]}, sharex=True)

# Main plot
axes[0].errorbar(x, y, yerr=yerr, fmt='o', capsize=3,
                 color='blue', alpha=0.7, label='Data')
axes[0].plot(x, y_fit, 'r-', linewidth=2,
             label=f'Fit: y = {slope:.2f}x + {intercept:.2f}')
axes[0].set_ylabel('y', fontsize=12)
axes[0].set_title('Linear Fit to Experimental Data', fontsize=14)
axes[0].legend(fontsize=11)
axes[0].grid(True, alpha=0.3)

# Residuals
axes[1].errorbar(x, residuals, yerr=yerr, fmt='o', capsize=3, color='green', alpha=0.
 ↪7)
axes[1].axhline(y=0, color='r', linestyle='--')
axes[1].set_xlabel('x', fontsize=12)
axes[1].set_ylabel('Residuals', fontsize=12)
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('analysis_result.png', dpi=300, bbox_inches='tight')
print("\nSaved: analysis_result.png")
plt.show()
```

```
=== Analysis Results ===
Slope: 2.384
Intercept: 1.129
χ²/dof: 3.71

Saved: analysis_result.png
```



3.6 VI. Summary

3.6.1 What We Learned Today

NumPy Arrays

- Creating arrays: `np.array()`, `np.zeros()`, `np.linspace()`, `np.arange()`
- Array operations: element-wise math, broadcasting, universal functions
- Indexing: slicing, boolean indexing (filtering)
- Methods: `sum()`, `mean()`, `std()`, `min()`, `max()`, `diff()`, `cumsum()`

Advanced Matplotlib

- Subplots: `plt.subplots(rows, cols)`
- Plot types: `scatter()`, `hist()`, `errorbar()`, `contour()`
- Customization: labels, LaTeX, styles
- Saving: `plt.savefig()`

Vectorization

- NumPy operations are 10-100x faster than Python loops
- Think in arrays, not loops
- Use boolean indexing for filtering

Functions as Arguments

- Functions can be passed to other functions
- Lambda functions for quick, anonymous functions
- Essential for numerical methods (integration, differentiation, root finding)

CHAPTER

FOUR

NUMERICAL INTEGRATION

4.1 Why Numerical Integration?

Many integrals in physics cannot be solved analytically:

- The integrand has no closed-form antiderivative
- The function is defined by data points, not a formula
- The integral is too complex for symbolic computation

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

# For nicer plots
plt.rcParams['figure.figsize'] = [10, 6]
plt.rcParams['font.size'] = 12
```

4.2 I. The Integration Problem

We want to compute the definite integral:

$$I = \int_a^b f(x)\,dx$$

Geometrically, this is the **signed area** under the curve $f(x)$ between $x = a$ and $x = b$.

4.2.1 Example: An "Impossible" Integral

Consider the Gaussian integral:

$$I = \int_0^1 e^{-x^2}\,dx$$

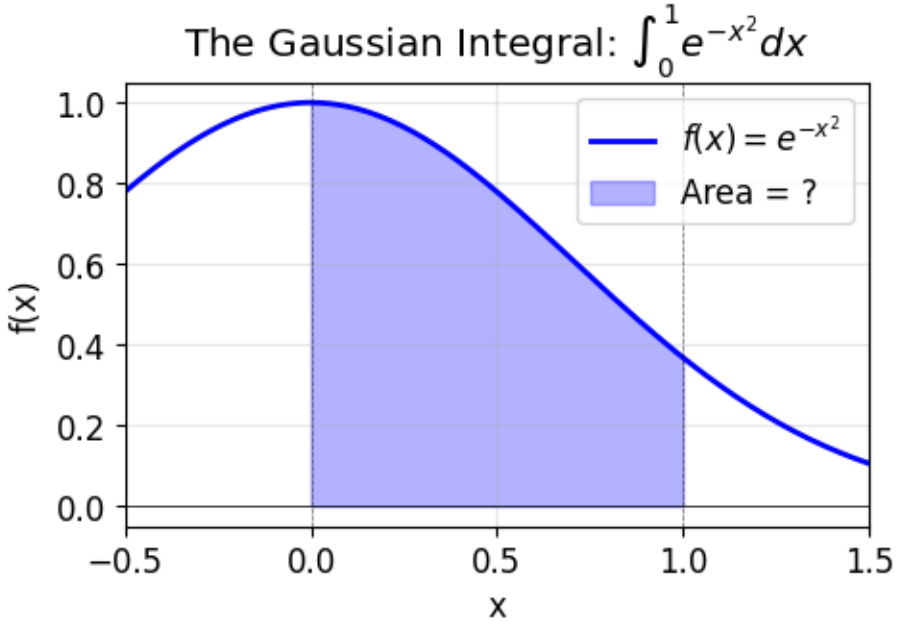
This integral has **no closed-form antiderivative** in terms of elementary functions! But we can compute it numerically.

```
# Visualize the integral we want to compute
def f(x):
    return np.exp(-x**2)
```

(continues on next page)

(continued from previous page)

```
# f = lambda x: x**2

x = np.linspace(-0.5, 1.5, 200)
y = f(x)

# Area to integrate
x_fill = np.linspace(0, 1, 100)
y_fill = f(x_fill)

plt.figure(figsize=(5, 3))
plt.plot(x, y, 'b-', linewidth=2, label=r'$f(x) = e^{-x^2}$')
plt.fill_between(x_fill, y_fill, alpha=0.3, color='blue', label='Area = ?')
plt.axhline(y=0, color='k', linestyle='-', linewidth=0.5)
plt.axvline(x=0, color='k', linestyle='--', linewidth=0.5, alpha=0.5)
plt.axvline(x=1, color='k', linestyle='--', linewidth=0.5, alpha=0.5)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title(r'The Gaussian Integral: $\int_0^1 e^{-x^2} dx$')
plt.legend()
plt.grid(True, alpha=0.3)
plt.xlim(-0.5, 1.5)
plt.show()
```



4.3 II. Riemann Sums

The fundamental idea: **approximate the area with rectangles**.

1. Divide $[a, b]$ into n subintervals of width $h = \frac{b-a}{n}$
2. Approximate the area in each subinterval with a rectangle
3. Sum all the rectangles

4.3.1 Left Riemann Sum

Use the **left endpoint** of each subinterval:

$$I \approx \sum_{i=0}^{n-1} f(x_i) \cdot h$$

where $x_i = a + i \cdot h$

```
def visualize_riemann(f, a, b, n, method='left'):
    """Visualize Riemann sum approximation."""
    h = (b - a) / n
    x_fine = np.linspace(a, b, 200)

    plt.figure(figsize=(5, 3))
    plt.plot(x_fine, f(x_fine), 'b-', linewidth=2, label='f(x)')

    total = 0
    for i in range(n):
        x_left = a + i * h
        x_right = x_left + h

        if method == 'left':
            height = f(x_left)
        elif method == 'right':
            height = f(x_right)
        elif method == 'midpoint':
            height = f((x_left + x_right) / 2)

        total += height * h

        # Draw rectangle
        rect = plt.Rectangle((x_left, 0), h, height,
                             fill=True, alpha=0.3, color='green',
                             edgecolor='green', linewidth=1)
        plt.gca().add_patch(rect)

    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title(f'{method.capitalize()} Riemann Sum (n={n}): I ≈ {total:.6f}')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    return total

# Visualize left Riemann sum with different n values
```

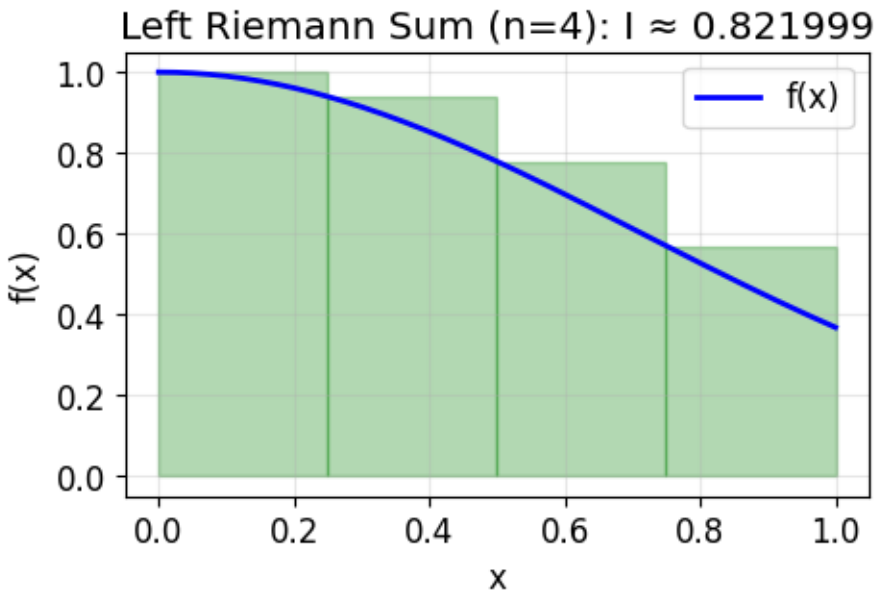
(continues on next page)

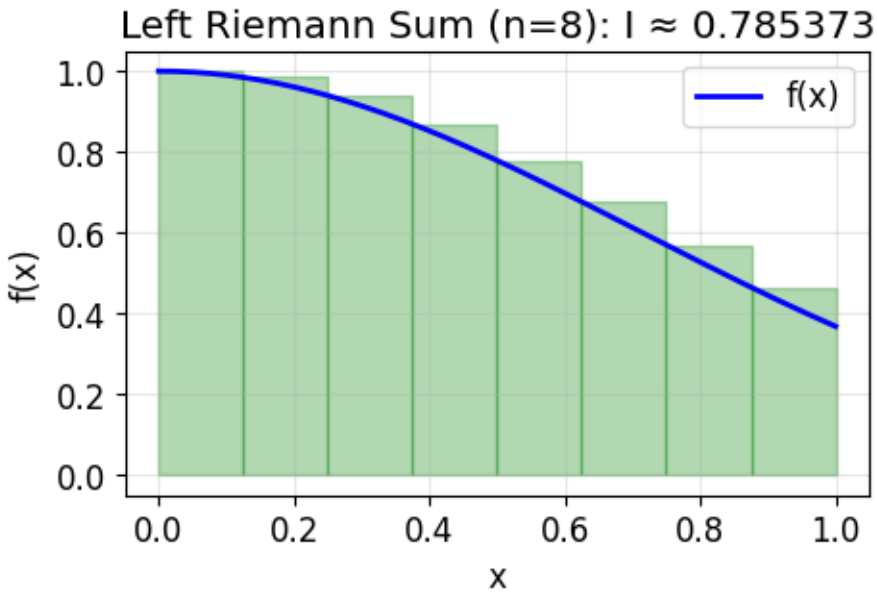
(continued from previous page)

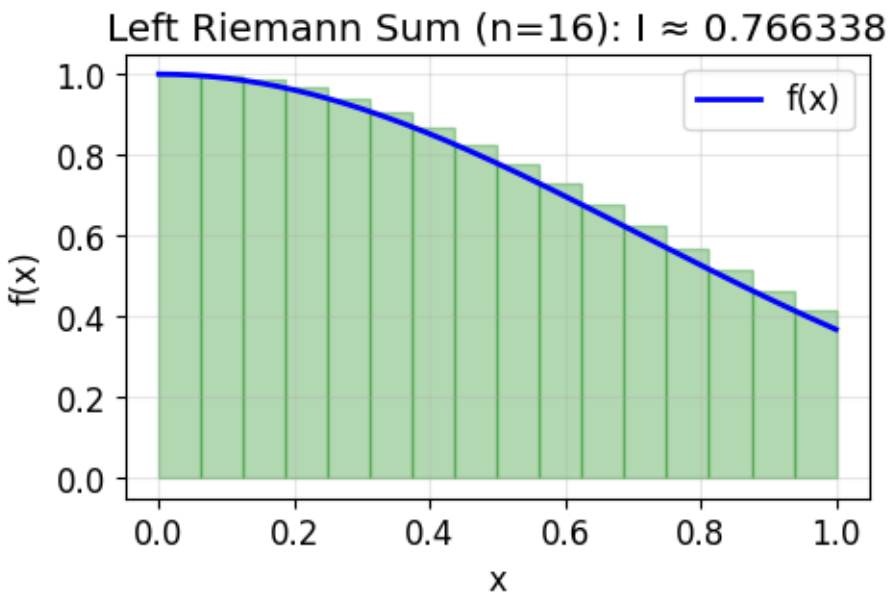
```
print("Left Riemann Sum:")
for n in [4, 8, 16]:
    result = visualize_riemann(f, 0, 1, n, 'left')
```

```
Left Riemann Sum:
```

```
/tmp/ipython-input-1523402423.py:24: UserWarning: Setting the 'color' property␣
↪will override the edgecolor or facecolor properties.
  rect = plt.Rectangle((x_left, 0), h, height,
```







4.3.2 Right and Midpoint Riemann Sums

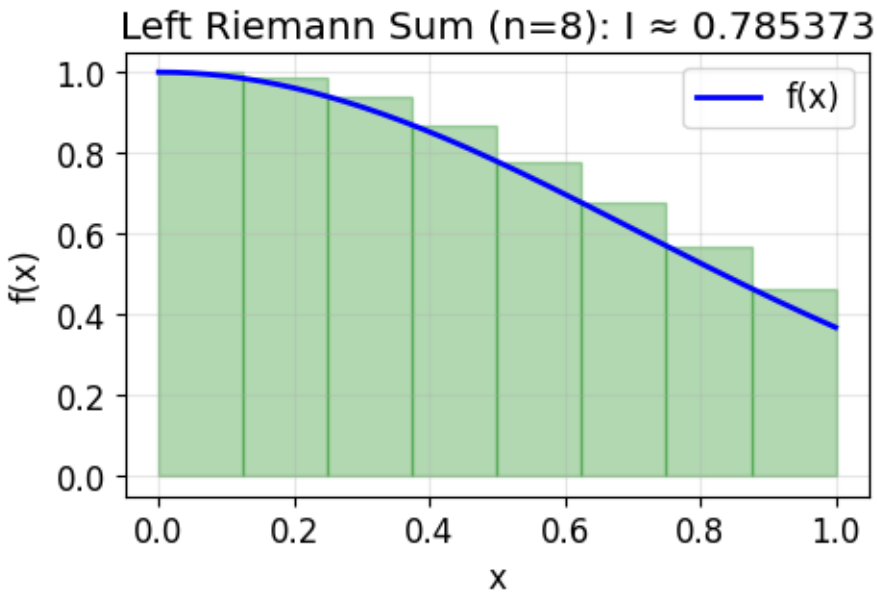
Right Riemann Sum: Use the right endpoint $I \approx \sum_{i=1}^{n} f(x_i) \cdot h$

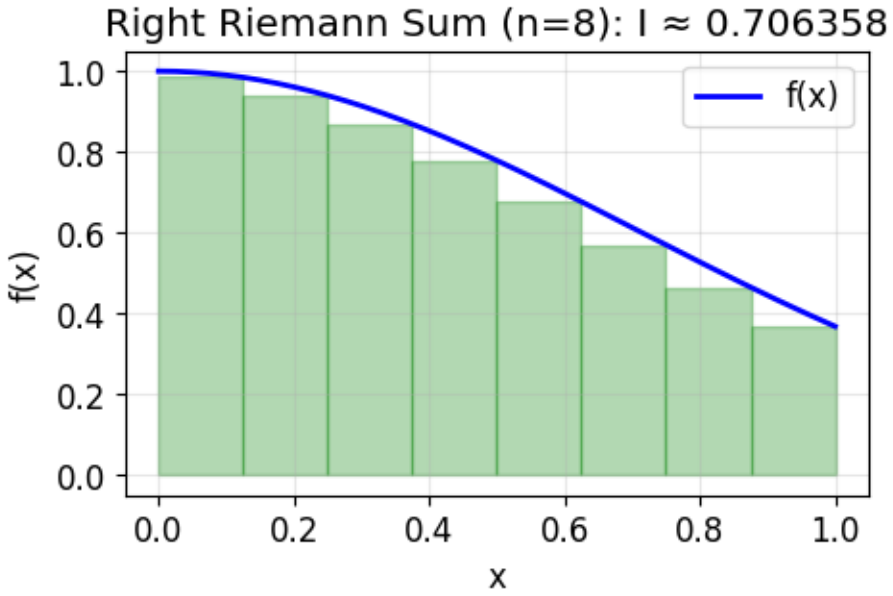
Midpoint Rule: Use the midpoint (often more accurate!) $I \approx \sum_{i=0}^{n-1} f\left(x_i + \frac{h}{2}\right) \cdot h$

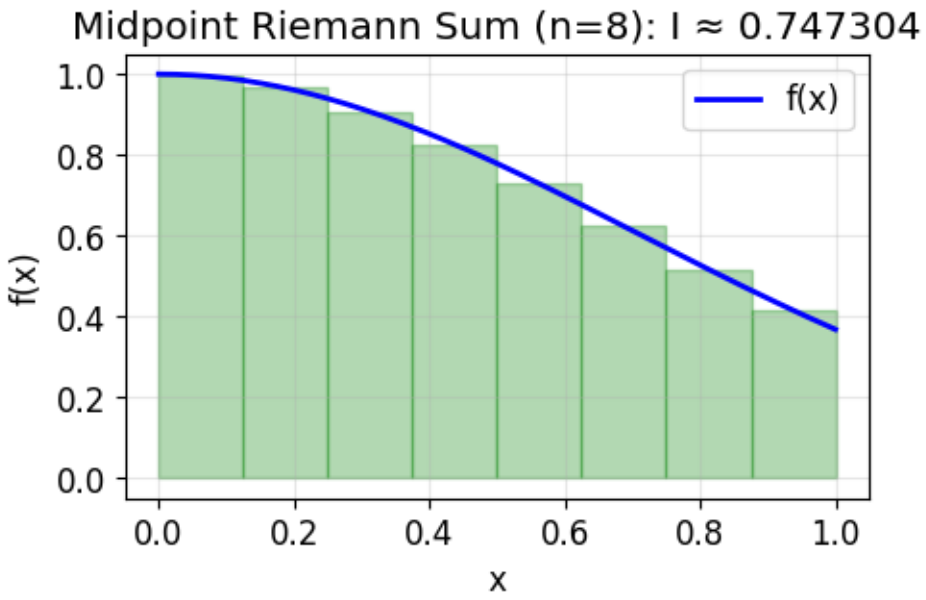
```
# Compare different Riemann sums
print("Comparison of Riemann Sum Methods (n=8):")
print("=" * 40)

for method in ['left', 'right', 'midpoint']:
    result = visualize_riemann(f, 0, 1, 8, method)
```

```
Comparison of Riemann Sum Methods (n=8):
========================================
```

```
/tmp/ipython-input-1523402423.py:24: UserWarning: Setting the 'color' property␣
↪will override the edgecolor or facecolor properties.
  rect = plt.Rectangle((x_left, 0), h, height,
```


Left Riemann Sum (n=8): I ≈ 0.785373
f(x)
1.0
0.8
0.6
0.4
0.2
0.0
f(x)
0.0
0.2
0.4
0.6
0.8
1.0
x


Right Riemann Sum (n=8): I ≈ 0.706358
f(x)
1.0
0.8
0.6
0.4
0.2
0.0
f(x)
0.0
0.2
0.4
0.6
0.8
1.0
x



4.3.3 Implementing Riemann Sums

```python
def riemann_sum(f, a, b, n, method='midpoint'):
    """
    Compute Riemann sum approximation of integral.

    Parameters:
        f: function to integrate
        a, b: integration limits
        n: number of subintervals
        method: 'left', 'right', or 'midpoint'

    Returns:
        Approximate value of the integral
    """
    h = (b - a) / n

    if method == 'left':
        x = np.linspace(a, b - h, n)
    elif method == 'right':
        x = np.linspace(a + h, b, n)
    elif method == 'midpoint':
        x = np.linspace(a + h/2, b - h/2, n)

    return np.sum(f(x)) * h

# Test with increasing n
print("Convergence of Midpoint Rule:")
print(f"{'n':>8} {'Integral':>15} {'Error':>15}")
print("-" * 40)

# True value (from scipy)
true_value, _ = integrate.quad(f, 0, 1)
# true_value = 1/3  # For f(x) = x^2 from 0 to 1
```

(continues on next page)

(continued from previous page)

```
for n in [10, 100, 1000, 10000]:
    result = riemann_sum(f, 0, 1, n, 'midpoint')
    error = abs(result - true_value)
    print(f"{n:>8} {result:>15.10f} {error:>15.2e}")

print(f"\nTrue value: {true_value:.10f}")
```

```
Convergence of Midpoint Rule:
       n        Integral           Error
----------------------------------------
      10    0.7471308777        3.07e-04
     100    0.7468271985        3.07e-06
    1000    0.7468241635        3.07e-08
   10000    0.7468241331        3.07e-10

True value: 0.7468241328
```

4.4 III. Trapezoidal Rule

Instead of rectangles, use **trapezoids** to better approximate the curve.

For each subinterval, connect the endpoints with a straight line:

$$I \approx \frac{h}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)\right]$$

Or equivalently:

$$I \approx h\left[\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(x_i)\right]$$

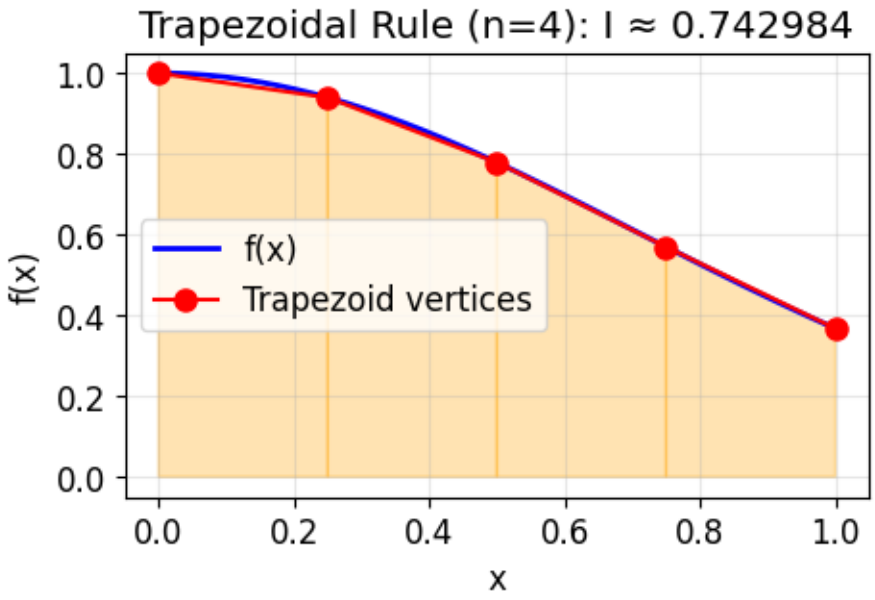
The trapezoidal rule is **exact for linear functions** and has error $O(h^2)$.

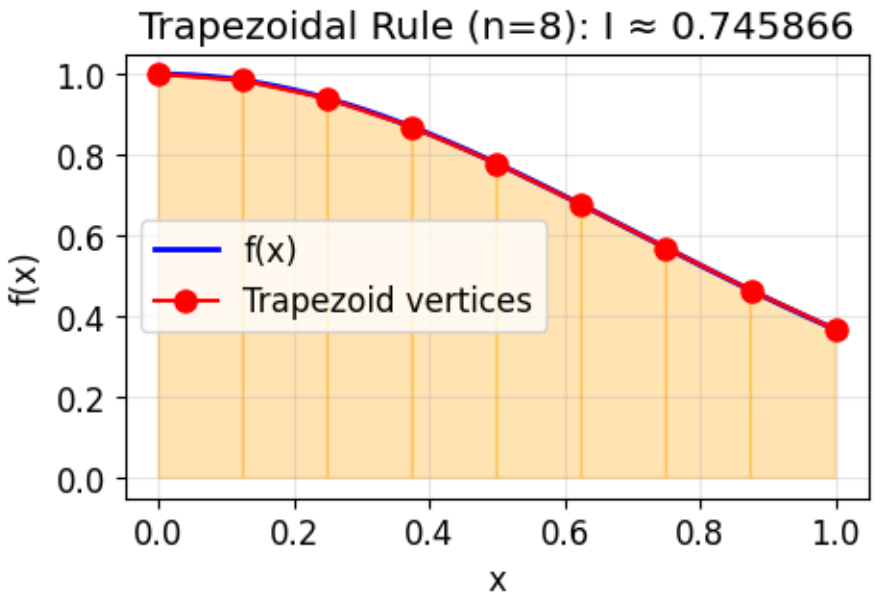
```
def visualize_trapezoid(f, a, b, n):
    """Visualize trapezoidal rule approximation."""
    h = (b - a) / n
    x_points = np.linspace(a, b, n + 1)
    y_points = f(x_points)

    x_fine = np.linspace(a, b, 200)

    plt.figure(figsize=(5, 3))
    plt.plot(x_fine, f(x_fine), 'b-', linewidth=2, label='f(x)')

    # Draw trapezoids
    for i in range(n):
        x_trap = [x_points[i], x_points[i], x_points[i+1], x_points[i+1]]
        y_trap = [0, y_points[i], y_points[i+1], 0]
        plt.fill(x_trap, y_trap, alpha=0.3, color='orange', edgecolor='orange',␣
↪linewidth=1)

    # Plot the approximating line segments
    plt.plot(x_points, y_points, 'ro-', markersize=8, label='Trapezoid vertices')
```

(continues on next page)

(continued from previous page)

```
    # Compute integral
    integral = h * (0.5 * y_points[0] + np.sum(y_points[1:-1]) + 0.5 * y_points[-1])

    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title(f'Trapezoidal Rule (n={n}): I ≈ {integral:.6f}')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    return integral

# Visualize
for n in [4, 8]:
    result = visualize_trapezoid(f, 0, 1, n)
```





4.4.1 Implementing the Trapezoidal Rule

```python
def trapezoidal(f, a, b, n):
    """
    Compute integral using the trapezoidal rule.

    Parameters:
        f: function to integrate
        a, b: integration limits
        n: number of subintervals

    Returns:
        Approximate value of the integral
    """
    h = (b-a)/n
    x = np.linspace(a, b, n+1)
    y = f(x)

    # Trapezoidal formula
    return h * (y[0]*0.5 + y[-1]*0.5 + np.sum(y[1:-1]))

# Test convergence
print("Convergence of Trapezoidal Rule:")
print(f"{'n':>8} {'Integral':>15} {'Error':>15}")
print("-" * 40)

for n in [10, 100, 1000, 10000]:
    result = trapezoidal(f, 0, 1, n)
    error = abs(result - true_value)
    print(f"{n:>8} {result:>15.10f} {error:>15.2e}")

print(f"\nTrue value: {true_value:.10f}")
```

```
Convergence of Trapezoidal Rule:
       n        Integral           Error
----------------------------------------
      10    0.7462107961        6.13e-04
     100    0.7468180015        6.13e-06
    1000    0.7468240715        6.13e-08
   10000    0.7468241322        6.13e-10

True value: 0.7468241328
```

4.5 IV. Simpson's Rule

The trapezoidal rule is simple, taking only a few lines of code as we have seen. And it is often perfectly adequate for calculations where no great accuracy is required. It happens frequently in physics calculations that we don't need an answer accurate to many significant figures and in such cases the ease and simplicity of the trapezoidal rule can make it the method of choice.

However, there are also cases where greater accuracy is required.

As we have seen we can increase the accuracy of the trapezoidal rule by *increasing the number N of steps used in the calculation*.

But in some cases, *the calculation can become slow*.

There are other, more advanced schemes for calculating integrals that can achieve high accuracy while still arriving at an answer quickly.

We can often get a better result if we approximate the function instead with curves of some kind.

Simpson's rule does exactly this, using quadratic curves.

In order to specify a quadratic completely

one needs three points, not just two as with a straight line.

Suppose, as before, that our integrand is denoted f(x) and the spacing of adjacent points is h. And suppose for the purposes of argument that we have three points at x = −h, 0, and +h. If we fit a quadratic $Ax^2 + Bx + C$ through these points, then by definition we will have:

$$f(-h) = Ah^2 - Bh + C$$

$$f(0) = C$$

$$f(h) = Ah^2 + Bh + C$$

Solving these equations gives

$$A = \frac{1}{h^2}[\frac{f(-h)}{2} - f(0) + \frac{f(h)}{2}]$$

$$B = \frac{1}{2h}[f(h) - f(-h)]$$

$$C = f(0)$$

and the area under the curve of f(x) from −h to +h is given approximately by the area under the quadratic:

$$\int_{-h}^{h}(Ax^2 + Bx + C)dx = \frac{2}{3}Ah^3 + 2Ch = \frac{1}{3}h[f(-h) + 4f(0) + f(h)]$$

This is *Simpson's* rule. It gives us an approximation to the area under two adjacent slices of our function. Note that the final formula for the area involves only h and the value of the function at evenly spaced points, just as with the trapezoidal rule. So to use Simpson's rule we don't actually have to worry about the details of fitting a quadratic—we just plug numbers into this formula and it gives us an answer.

Applying Simpson's rule involves dividing the domain of integration into many slices and using the rule to separately estimate the area under successive pairs of slices, then adding the estimates for all pairs to get the final answer. If, as before, we are integrating from x = a to x = b in slices of width h then the three points bounding the first pair of slices fall at $x = a, a + h, a + 2h$, those bounding the second pair at $a + 2h, a + 3h, a + 4h$, and so forth. Then the approximate value of the entire integral is given by

$$
\begin{aligned}
I(a,b) \approx & \frac{1}{3}h[f(a) + 4f(a+h) + f(a+2h)] \\
& \frac{1}{3}h[f(a+2h) + 4f(a+3h) + f(a+4h)] \\
& ... \\
& ... \\
& \frac{1}{3}h[f(a+(N-2)h) + 4f(a+(N-1)h) + f(b)] \\
= & \frac{1}{3}h[f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + 2f(a+4h) + ... + 2f(a+(N-2)h) + 4f(a+(N-1)h) + f(b)]
\end{aligned}
$$

1/3 h (1, 4, 2,4,2,4,2,4….,4,1)

see more details in: https://en.wikipedia.org/wiki/Simpson's_rule

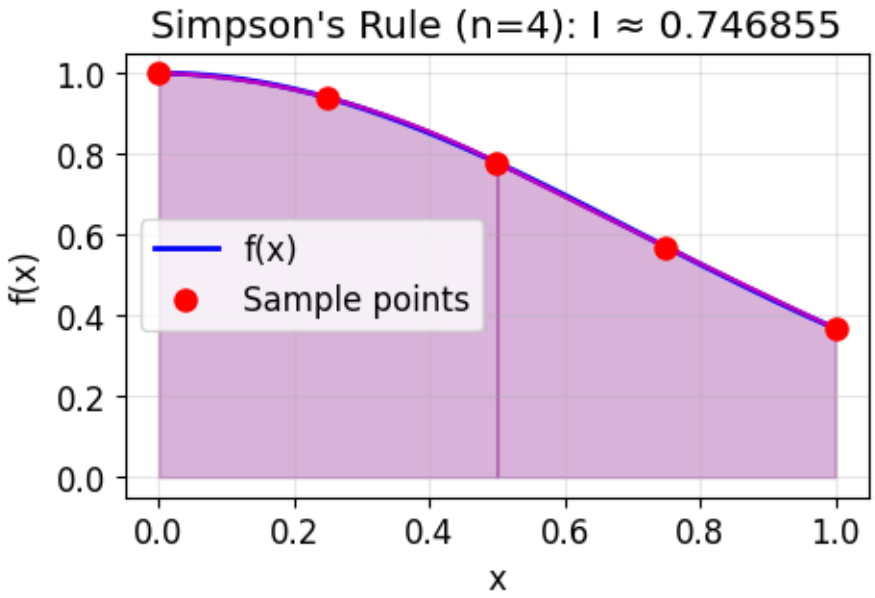
Simpson's rule is **exact for polynomials up to degree 3** and has error $O(h^4)$ — much better than trapezoidal!

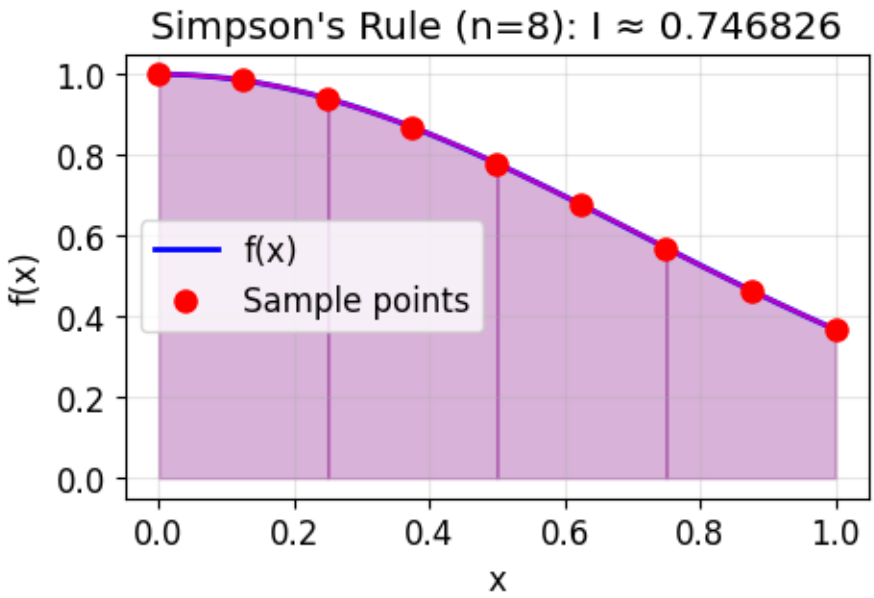
```
def visualize_simpson(f, a, b, n):
    """Visualize Simpson's rule with parabolas."""
    if n % 2 != 0:
        n += 1  # Ensure even

    h = (b - a) / n
    x_points = np.linspace(a, b, n + 1)
    y_points = f(x_points)

    x_fine = np.linspace(a, b, 200)

    plt.figure(figsize=(5, 3))
    plt.plot(x_fine, f(x_fine), 'b-', linewidth=2, label='f(x)')

    # Draw parabolic segments
    for i in range(0, n, 2):
        # Fit parabola through 3 points
        x3 = x_points[i:i+3]
        y3 = y_points[i:i+3]
        coeffs = np.polyfit(x3, y3, 2)

        x_para = np.linspace(x3[0], x3[2], 50)
        y_para = np.polyval(coeffs, x_para)

        plt.fill_between(x_para, y_para, alpha=0.3, color='purple')
        plt.plot(x_para, y_para, 'm-', linewidth=1.5)

    plt.plot(x_points, y_points, 'ro', markersize=8, label='Sample points')

    # Compute integral using Simpson's rule
```

(continues on next page)

(continued from previous page)

```
    coefficients = np.ones(n + 1)
    coefficients[1:-1:2] = 4  # Odd indices
    coefficients[2:-1:2] = 2  # Even indices (except first and last)
    integral = h / 3 * np.sum(coefficients * y_points)

    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title(f"Simpson's Rule (n={n}): I ≈ {integral:.6f}")
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    return integral

# Visualize
for n in [4, 8]:
    result = visualize_simpson(f, 0, 1, n)
```





4.5.1 Implementing Simpson's Rule

```python
def simpson(f, a, b, n):
    """
    Compute integral using Simpson's rule.

    Parameters:
        f: function to integrate
        a, b: integration limits
        n: number of subintervals (must be even)

    Returns:
        Approximate value of the integral
    """
    if n % 2 != 0:
        raise ValueError("n must be even for Simpson's rule")

    h = (b - a) / n
    x = np.linspace(a, b, n + 1)
    y = f(x)

    # Simpson's coefficients: 1, 4, 2, 4, 2, ..., 4, 1
    coefficients = np.ones(n + 1)
    coefficients[1:-1:2] = 4
    coefficients[2:-1:2] = 2

    return  np.sum(coefficients * y) * h/3

# Test convergence
print("Convergence of Simpson's Rule:")
print(f"{'n':>8} {'Integral':>15} {'Error':>15}")
print("-" * 40)

for n in [10, 100, 1000, 10000]:
```

(continues on next page)

(continued from previous page)

```
    result = simpson(f, 0, 1, n)
    error = abs(result - true_value)
    print(f"{n:>8} {result:>15.10f} {error:>15.2e}")

print(f"\nTrue value: {true_value:.10f}")
```

```
Convergence of Simpson's Rule:
       n        Integral           Error
----------------------------------------
      10    0.7468249483        8.15e-07
     100    0.7468241329        8.17e-11
    1000    0.7468241328        7.99e-15
   10000    0.7468241328        0.00e+00

True value: 0.7468241328
```

4.6 V. Using SciPy for Integration

In practice, use **scipy.integrate** — it provides robust, well-tested implementations.

4.6.1 Main Functions

Function	Use Case
`quad()`	General-purpose adaptive integration
`trapezoid()`	Trapezoidal rule for sampled data
`simpson()`	Simpson's rule for sampled data
`dblquad()`	Double integrals
`tplquad()`	Triple integrals

```
# scipy.integrate.quad - Adaptive quadrature
from scipy.integrate import quad

def f(x):
    return np.exp(-x**2)

# quad returns (value, error_estimate)
result, error = quad(f, 0, 1)

print("Using scipy.integrate.quad:")
print(f"  Integral: {result:.15f}")
print(f"  Error estimate: {error:.2e}")

# It can handle infinite limits!
result_inf, error_inf = quad(f, 0, np.inf)
print(f"\n∫₀^∞ e^(-x²) dx = {result_inf:.10f}")
print(f"(Exact: √π/2 = {np.sqrt(np.pi)/2:.10f})")
```

```
Using scipy.integrate.quad:
  Integral: 0.746824132812427
```

(continues on next page)

(continued from previous page)

```
  Error estimate: 8.29e-15

∫₀^∞ e^(-x²) dx = 0.8862269255
(Exact: √π/2 = 0.8862269255)
```

4.6.2 Integrating Sampled Data

When you have data points (not a function), use `trapezoid()` or `simpson()`:

```
from scipy.integrate import trapezoid, simpson

# Sampled data (e.g., from an experiment)
x_data = np.array([0, 0.25, 0.5, 0.75, 1.0])
y_data = np.exp(-x_data**2)  # Simulating measured values

# Integrate using trapezoidal rule
result_trapz = trapezoid(y_data, x_data)

# Integrate using Simpson's rule
result_simp = simpson(y_data, x=x_data)

print("Integrating sampled data:")
print(f"  x = {x_data}")
print(f"  y = {np.round(y_data, 4)}")
print(f"\n  Trapezoidal: {result_trapz:.6f}")
print(f"  Simpson's:   {result_simp:.6f}")
print(f"  True value:  {true_value:.6f}")
```

```
Integrating sampled data:
  x = [0.   0.25 0.5  0.75 1.  ]
  y = [1.     0.9394 0.7788 0.5698 0.3679]

  Trapezoidal: 0.742984
  Simpson's:   0.746855
  True value:  0.746824
```

4.6.3 Multiple Integrals

```
from scipy.integrate import dblquad

# Double integral: ∫∫ x*y dA over the unit square [0,1] × [0,1]
def integrand(y, x):  # Note: y first, then x!
    return x * y

result, error = dblquad(integrand, 0, 1, 0, 1)
print(f"∫∫ xy dA over [0,1]×[0,1] = {result:.6f}")
print(f"Exact: 1/4 = {0.25:.6f}")

# Volume of a hemisphere: ∫∫ √(R² - x² - y²) dA over disk of radius R
R = 1

def hemisphere(y, x):
```

(continues on next page)

(continued from previous page)

```
    r2 = x**2 + y**2
    if r2 < R**2:
        return np.sqrt(R**2 - r2)
    return 0

def y_lower(x):
    return -np.sqrt(max(0, R**2 - x**2))

def y_upper(x):
    return np.sqrt(max(0, R**2 - x**2))

volume, _ = dblquad(hemisphere, -R, R, y_lower, y_upper)
print(f"\nVolume of hemisphere (R=1): {volume:.6f}")
print(f"Exact: (2/3)πR³ = {2/3 * np.pi * R**3:.6f}")
```

```
∫∫ xy dA over [0,1]×[0,1] = 0.250000
Exact: 1/4 = 0.250000

Volume of hemisphere (R=1): 2.094395
Exact: (2/3)πR³ = 2.094395
```

4.7 VI. Physics Applications

Let's apply numerical integration to real physics problems!

4.7.1 1. Work Done by a Variable Force

The work done by a force $F(x)$ moving an object from a to b:

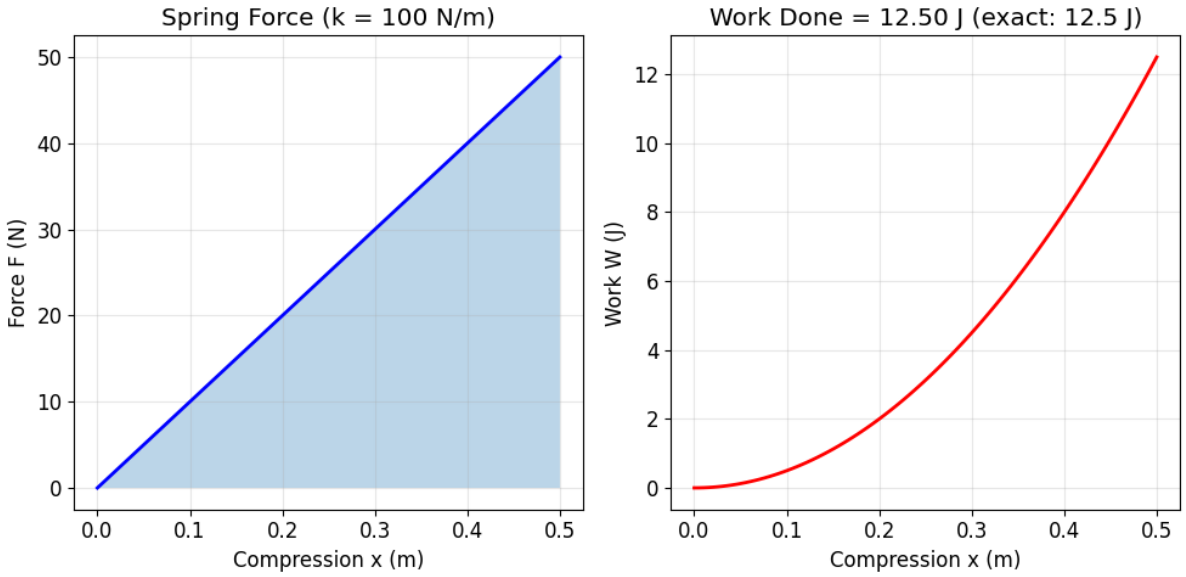
$$W = \int_a^b F(x)\,dx$$

```
# Work to compress a spring (Hooke's Law: F = kx)
k = 100  # N/m (spring constant)

def spring_force(x):
    return k * x

# Compress spring from 0 to 0.5 meters
x = np.linspace(0, 0.5, 100)
F = spring_force(x)

# Compute work
W, _ = quad(spring_force, 0, 0.5)

# Exact: W = (1/2)kx² = 0.5 * 100 * 0.5² = 12.5 J

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(x, F, 'b-', linewidth=2)
plt.fill_between(x, F, alpha=0.3)
plt.xlabel('Compression x (m)')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Force F (N)')
plt.title(f'Spring Force (k = {k} N/m)')
plt.grid(True, alpha=0.3)

plt.subplot(1, 2, 2)
# Cumulative work as function of compression
W_cumulative = [quad(spring_force, 0, xi)[0] for xi in x]
plt.plot(x, W_cumulative, 'r-', linewidth=2)
plt.xlabel('Compression x (m)')
plt.ylabel('Work W (J)')
plt.title(f'Work Done = {W:.2f} J (exact: 12.5 J)')
plt.grid(True, alpha=0.3)

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



4.7.2 2. Center of Mass of a Non-Uniform Rod

For a rod with linear density $\lambda(x)$:

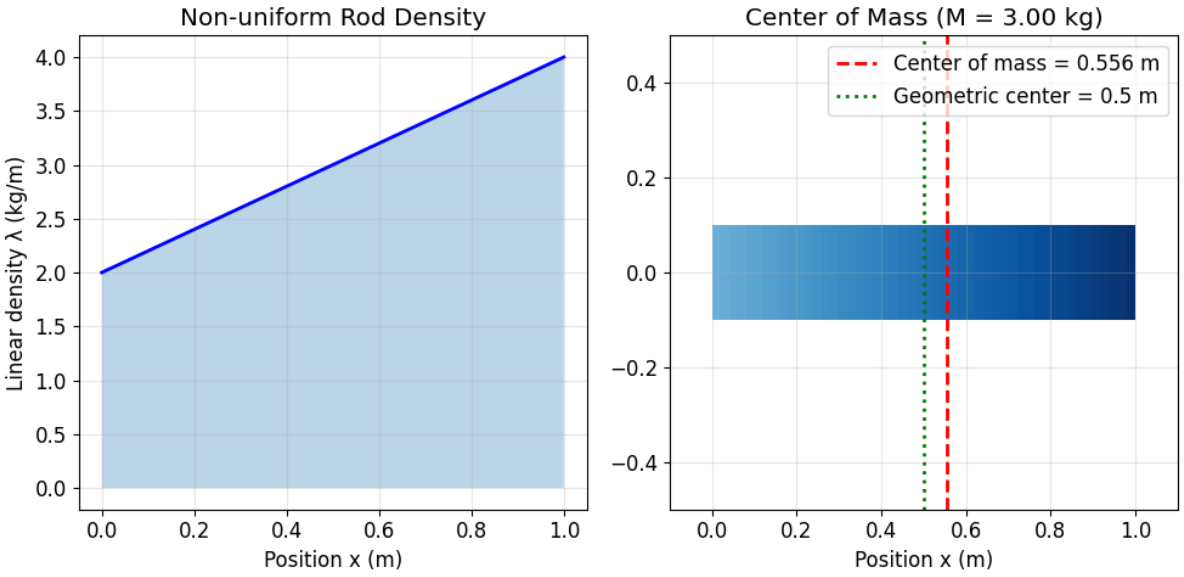
$$x_{cm} = \frac{\int_0^L x \cdot \lambda(x)\, dx}{\int_0^L \lambda(x)\, dx} = \frac{\int_0^L x \cdot \lambda(x)\, dx}{M}$$

```
# Rod with linearly increasing density: λ(x) = λ₀(1 + x/L)
L = 1.0  # Length of rod (m)
lambda0 = 2.0  # kg/m at x=0

def density(x):
    return lambda0 * (1 + x / L)

def x_times_density(x):
    return x * density(x)
```

(continues on next page)

(continued from previous page)

```
# Total mass
M, _ = quad(density, 0, L)

# First moment
moment, _ = quad(x_times_density, 0, L)

# Center of mass
x_cm = moment / M

# Visualize
x = np.linspace(0, L, 100)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(x, density(x), 'b-', linewidth=2)
plt.fill_between(x, density(x), alpha=0.3)
plt.xlabel('Position x (m)')
plt.ylabel('Linear density λ (kg/m)')
plt.title('Non-uniform Rod Density')
plt.grid(True, alpha=0.3)

plt.subplot(1, 2, 2)
# Show rod as colored bar
for i in range(50):
    xi = i * L / 50
    color_intensity = density(xi) / density(L)
    plt.barh(0, L/50, left=xi, height=0.2, color=plt.cm.Blues(color_intensity))
plt.axvline(x_cm, color='red', linewidth=2, linestyle='--', label=f'Center of mass =
 ↪{x_cm:.3f} m')
plt.axvline(L/2, color='green', linewidth=2, linestyle=':', label='Geometric center =␣
 ↪0.5 m')
plt.xlabel('Position x (m)')
plt.xlim(-0.1, 1.1)
plt.ylim(-0.5, 0.5)
plt.legend()
plt.title(f'Center of Mass (M = {M:.2f} kg)')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Total mass: M = {M:.4f} kg")
print(f"Center of mass: x_cm = {x_cm:.4f} m")
print(f"(The heavier end pulls the center of mass toward x = L)")
```




```
Total mass: M = 3.0000 kg
Center of mass: x_cm = 0.5556 m
(The heavier end pulls the center of mass toward x = L)
```

4.7.3 3. Probability from a Distribution

For a probability density function $p(x)$, the probability of x being in $[a, b]$:

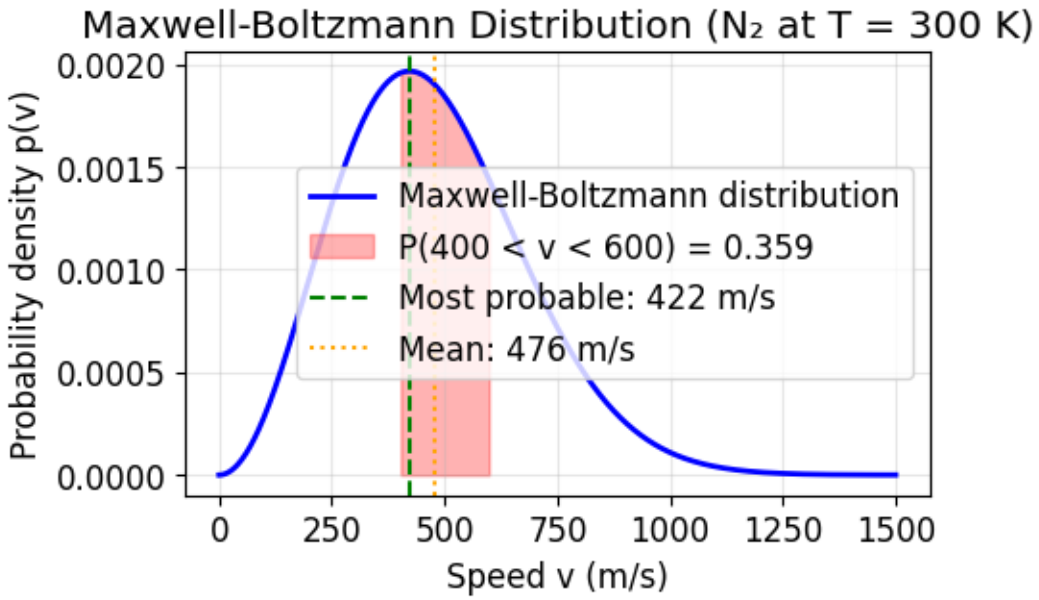
$$P(a \leq x \leq b) = \int_a^b p(x)\,dx$$

```python
# Maxwell-Boltzmann speed distribution
# p(v) = 4π(m/2πkT)^(3/2) * v² * exp(-mv²/2kT)

def maxwell_boltzmann(v, T, m):
    """Maxwell-Boltzmann speed distribution."""
    kB = 1.38e-23  # Boltzmann constant
    a = m / (2 * kB * T)
    return 4 * np.pi * (a / np.pi)**(3/2) * v**2 * np.exp(-a * v**2)

# Nitrogen molecule at room temperature
m_N2 = 28 * 1.66e-27  # kg (N₂ mass)
T = 300  # K

v = np.linspace(0, 1500, 500)  # m/s
p = maxwell_boltzmann(v, T, m_N2)

# Probability that speed is between 400 and 600 m/s
prob, _ = quad(lambda v: maxwell_boltzmann(v, T, m_N2), 400, 600)

# Most probable speed
v_mp = np.sqrt(2 * 1.38e-23 * T / m_N2)

# Mean speed
mean_v, _ = quad(lambda v: v * maxwell_boltzmann(v, T, m_N2), 0, 3000)
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(5, 3))
plt.plot(v, p, 'b-', linewidth=2, label='Maxwell-Boltzmann distribution')
plt.fill_between(v[(v >= 400) & (v <= 600)],
                 p[(v >= 400) & (v <= 600)],
                 alpha=0.3, color='red',
                 label=f'P(400 < v < 600) = {prob:.3f}')
plt.axvline(v_mp, color='green', linestyle='--', label=f'Most probable: {v_mp:.0f} m/s
↪')
plt.axvline(mean_v, color='orange', linestyle=':', label=f'Mean: {mean_v:.0f} m/s')

plt.xlabel('Speed v (m/s)')
plt.ylabel('Probability density p(v)')
plt.title(f'Maxwell-Boltzmann Distribution (N₂ at T = {T} K)')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

print(f"Probability that N₂ molecule has speed between 400-600 m/s: {prob*100:.1f}%")
```




```
Probability that N₂ molecule has speed between 400-600 m/s: 35.9%
```

4.8 VII. Summary

4.8.1 Methods Learned

Method	Order	Best For
Riemann Sum	$O(h)$ or $O(h^2)$	Understanding the concept
Trapezoidal	$O(h^2)$	Quick calculations, sampled data
Simpson's	$O(h^4)$	Smooth functions, high accuracy
`scipy.integrate.quad`	Adaptive	Production code

4.8.2 Key Takeaways

1. **Higher-order methods converge faster —** Simpson's needs far fewer points than trapezoidal for the same accuracy
2. **Use SciPy in practice —** `quad()` handles edge cases and adapts automatically
3. **For sampled data**, use `trapezoid()` or `simpson()` from scipy
4. **Check convergence —** increase n and verify the result doesn't change significantly

4.8.3 Coming Up Next

Numerical Differentiation — computing derivatives from functions or data

CHAPTER

FIVE

NUMERICAL DIFFERENTIATION

5.1 Why Numerical Differentiation?

- The function is only known at discrete data points (experimental measurements)
- The analytical derivative is too complex to compute
- We need to verify analytical results computationally
- Building blocks for solving differential equations

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import UnivariateSpline
from scipy.signal import savgol_filter

# For nicer plots
plt.rcParams['figure.figsize'] = [5, 3]
plt.rcParams['font.size'] = 12
```

5.2 I. The Differentiation Problem

We want to compute the derivative:

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

The challenge: we can't take $h \to 0$ on a computer! We must use a **finite** value of h.

5.2.1 The Fundamental Trade-off

- **Too large h**: Poor approximation (truncation error)
- **Too small h**: Numerical noise dominates (round-off error)

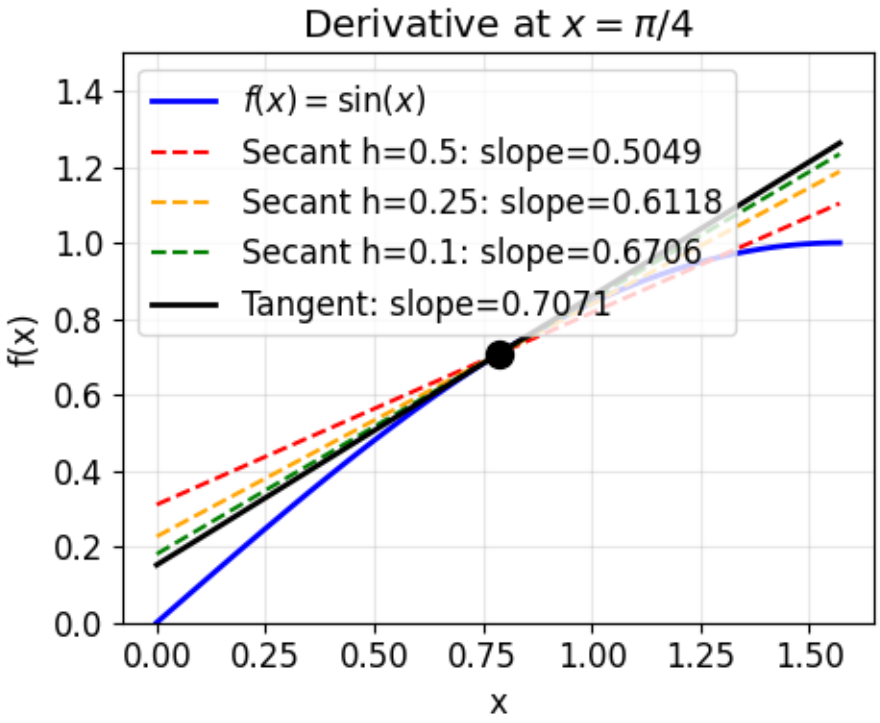
Finding the right h is key to accurate numerical differentiation.

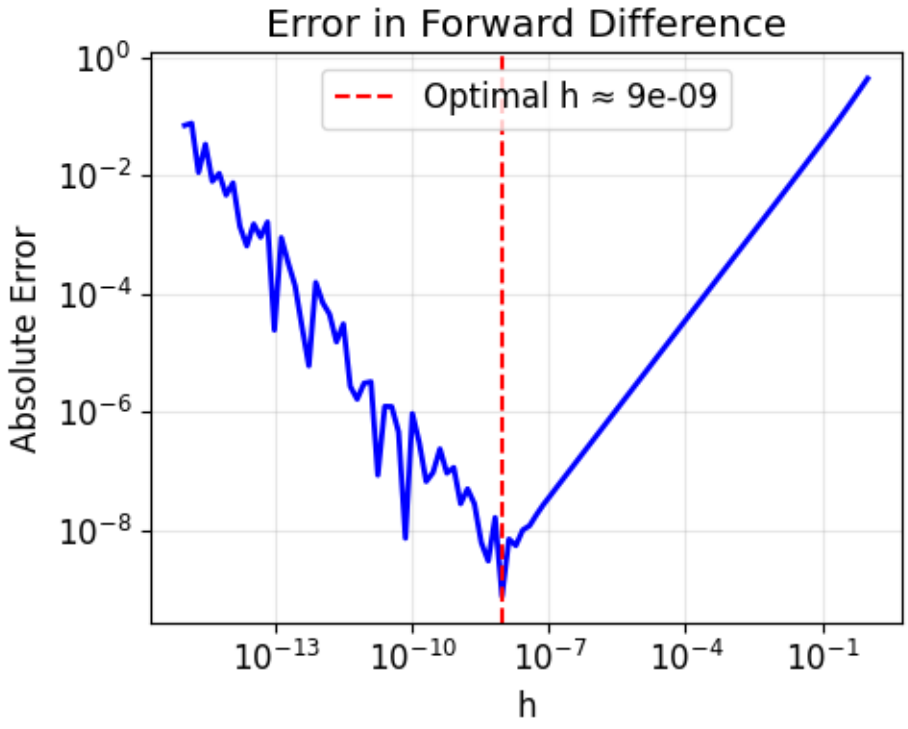
```
# Visualize the derivative as a limit
def f(x):
    return np.sin(x)

def f_prime_exact(x):
```

(continues on next page)

(continued from previous page)

```
    return np.cos(x)

x0 = np.pi/4  # Point where we compute derivative
x = np.linspace(0, np.pi/2, 100)

plt.figure(figsize=(5, 8))

plt.subplot(2, 1, 1)
plt.plot(x, f(x), 'b-', linewidth=2, label=r'$f(x) = \sin(x)$')

# Show secant lines for different h
for h, color in [(0.5, 'red'), (0.25, 'orange'), (0.1, 'green')]:
    slope = (f(x0 + h) - f(x0)) / h
    y_secant = f(x0) + slope * (x - x0)
    plt.plot(x, y_secant, color=color, linestyle='--',
             label=f'Secant h={h}: slope={slope:.4f}')

# Tangent line (true derivative)
slope_true = f_prime_exact(x0)
y_tangent = f(x0) + slope_true * (x - x0)
plt.plot(x, y_tangent, 'k-', linewidth=2, label=f'Tangent: slope={slope_true:.4f}')

plt.plot(x0, f(x0), 'ko', markersize=10)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title(r'Derivative at $x = \pi/4$')
plt.legend(loc='upper left')
plt.grid(True, alpha=0.3)
plt.ylim(0, 1.5)

plt.subplot(2, 1, 2)
# Error as function of h
h_values = np.logspace(-15, 0, 100)
errors = np.abs((f(x0 + h_values) - f(x0)) / h_values - f_prime_exact(x0))

plt.loglog(h_values, errors, 'b-', linewidth=2)
plt.xlabel('h')
plt.ylabel('Absolute Error')
plt.title('Error in Forward Difference')
plt.grid(True, alpha=0.3)

# Mark optimal region
optimal_h = h_values[np.argmin(errors)]
plt.axvline(optimal_h, color='r', linestyle='--', label=f'Optimal h ≈ {optimal_h:.0e}
 ↪')
plt.legend()

plt.tight_layout()
plt.show()

print(f"True derivative at x = π/4: {f_prime_exact(x0):.10f}")
print(f"Optimal h: {optimal_h:.2e}")
print(f"Minimum error: {errors.min():.2e}")
```






```
True derivative at x = π/4: 0.7071067812
Optimal h: 9.33e-09
Minimum error: 7.46e-10
```

5.3 II. Finite Difference Approximations

The three basic finite difference formulas:

5.3.1 Forward Difference

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

5.3.2 Backward Difference

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

5.3.3 Central Difference

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

The central difference is more accurate because errors cancel!

```
def forward_diff(f, x, h):
    """Forward difference approximation of f'(x)."""
    return (f(x + h) - f(x)) / h

def backward_diff(f, x, h):
    """Backward difference approximation of f'(x)."""
    return (f(x) - f(x-h)) / h

def central_diff(f, x, h):
    """Central difference approximation of f'(x)."""
    return (f(x+h) - f(x-h)) / (2*h)
```

```
# Compare the three methods
def f(x):
    return np.exp(x)

def f_prime(x):
    return np.exp(x)  # Derivative of e^x is e^x

x0 = 1.0
true_value = f_prime(x0)

print("Comparison of Finite Difference Methods")
print(f"Function: f(x) = e^x at x = {x0}")
print(f"True derivative: {true_value:.10f}")
print("=" * 70)
print(f"{'h':>12} {'Forward':>15} {'Backward':>15} {'Central':>15}")
print("-" * 70)

for h in [0.1, 0.01, 0.001, 0.0001, 0.00001]:
    fwd = forward_diff(f, x0, h)
    bwd = backward_diff(f, x0, h)
```

(continues on next page)

(continued from previous page)

```
    ctr = central_diff(f, x0, h)

    err_fwd = abs(fwd - true_value)
    err_bwd = abs(bwd - true_value)
    err_ctr = abs(ctr - true_value)

    print(f"{h:>12.0e} {err_fwd:>15.2e} {err_bwd:>15.2e} {err_ctr:>15.2e}")

print("\nObservation: Central difference is MUCH more accurate!")
```

```
Comparison of Finite Difference Methods
Function: f(x) = e^x at x = 1.0
True derivative: 2.7182818285
======================================================================
           h         Forward        Backward         Central
----------------------------------------------------------------------
       1e-01        1.41e-01        1.31e-01        4.53e-03
       1e-02        1.36e-02        1.35e-02        4.53e-05
       1e-03        1.36e-03        1.36e-03        4.53e-07
       1e-04        1.36e-04        1.36e-04        4.53e-09
       1e-05        1.36e-05        1.36e-05        5.86e-11

Observation: Central difference is MUCH more accurate!
```

5.3.4 Why is Central Difference Better?

Using Taylor series expansion:

Forward difference: $$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + O(h^3)$$

$$\frac{f(x+h) - f(x)}{h} = f'(x) + \frac{h}{2}f''(x) + O(h^2)$$

Error is $O(h)$ — **first-order accurate**

Central difference: $$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + O(h^4) f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + O(h^4)$$

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{h^2}{6}f'''(x) + O(h^4)$$

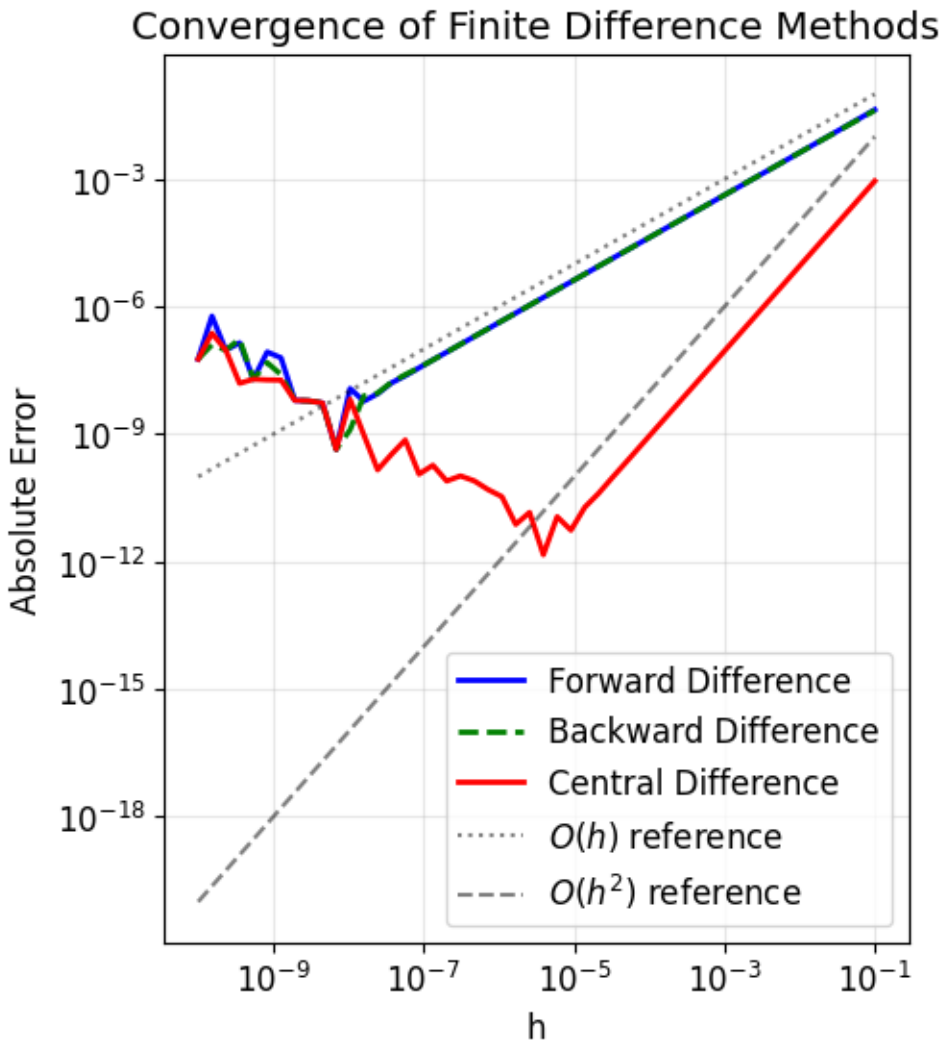
Error is $O(h^2)$ — **second-order accurate**

```
# Visualize convergence rates
def f(x):
    return np.sin(x)

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

x0 = 1.0
true_value = f_prime(x0)

h_values = np.logspace(-10, -1, 50)
errors_fwd = np.abs(forward_diff(f, x0, h_values) - true_value)
```

(continues on next page)

(continued from previous page)

```
errors_bwd = np.abs(backward_diff(f, x0, h_values) - true_value)
errors_ctr = np.abs(central_diff(f, x0, h_values) - true_value)

plt.figure(figsize=(5, 6))
plt.loglog(h_values, errors_fwd, 'b-', linewidth=2, label='Forward Difference')
plt.loglog(h_values, errors_bwd, 'g--', linewidth=2, label='Backward Difference')
plt.loglog(h_values, errors_ctr, 'r-', linewidth=2, label='Central Difference')

# Reference lines
plt.loglog(h_values, h_values, 'k:', alpha=0.5, label=r'$O(h)$ reference')
plt.loglog(h_values, h_values**2, 'k--', alpha=0.5, label=r'$O(h^2)$ reference')

plt.xlabel('h')
plt.ylabel('Absolute Error')
plt.title('Convergence of Finite Difference Methods')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

print("Notice:")
print("- Forward/Backward follow the O(h) line (slope = 1)")
print("- Central follows the O(h²) line (slope = 2)")
print("- At very small h, round-off error takes over!")
```




```
Notice:
- Forward/Backward follow the O(h) line (slope = 1)
- Central follows the O(h²) line (slope = 2)
- At very small h, round-off error takes over!
```

5.4 III. Second Derivatives

For the second derivative, we can use the **central difference formula**:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

This is derived by combining forward and backward differences, or by Taylor expansion.

This formula is $O(h^2)$ accurate.

```python
def second_derivative(f, x, h):
    """Central difference approximation of f''(x)."""
    # finish this function...
```

(continues on next page)

(continued from previous page)

```
    return (f(x+h) - 2*f(x) + f(x-h))/h**2

# Test with f(x) = sin(x), f''(x) = -sin(x)
def f(x):
    return np.sin(x)
    # return np.exp(x)

def f_double_prime(x):
    return -np.sin(x)
    # return np.exp(x)

x0 = np.pi/3
true_value = f_double_prime(x0)

print("Second Derivative Approximation")
print(f"Function: f(x) = sin(x) at x = π/3")
print(f"True f''(x): {true_value:.10f}")
print("=" * 50)
print(f"{'h':>12} {'Approximation':>18} {'Error':>15}")
print("-" * 50)

for h in [0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001]:
    approx = second_derivative(f, x0, h)
    error = abs(approx - true_value)
    print(f"{h:>12.0e} {approx:>18.10f} {error:>15.2e}")

print("\nNote: Very small h causes round-off errors (dividing by h²!)")
```

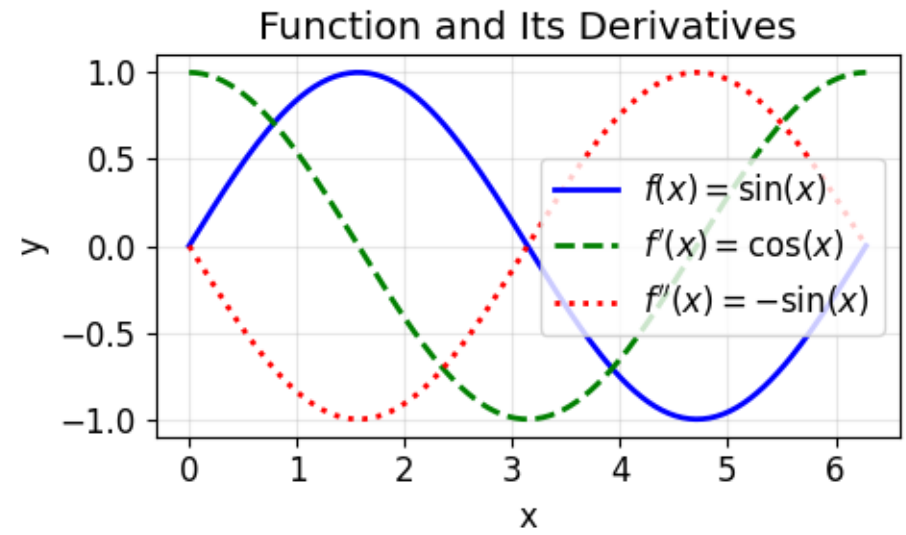
```
Second Derivative Approximation
Function: f(x) = sin(x) at x = π/3
True f''(x): -0.8660254038
==================================================
           h      Approximation           Error
--------------------------------------------------
       1e-01      -0.8653039565        7.21e-04
       1e-02      -0.8660181869        7.22e-06
       1e-03      -0.8660253317        7.21e-08
       1e-04      -0.8660253958        7.94e-09
       1e-05      -0.8660250295        3.74e-07
       1e-06      -0.8659739592        5.14e-05

Note: Very small h causes round-off errors (dividing by h²!)
```

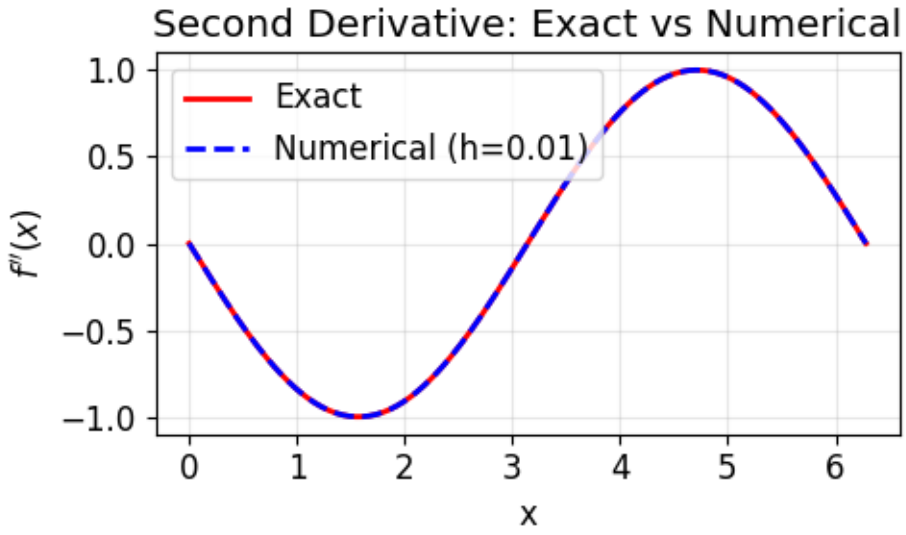
```
# Visualize second derivative calculation
x = np.linspace(0, 2*np.pi, 200)
y = np.sin(x)

# Compute numerical second derivative
h = 0.01
y_double_prime_numerical = second_derivative(np.sin, x, h)
y_double_prime_exact = -np.sin(x)

plt.figure(figsize=(5, 6))

plt.subplot(2, 1, 1)
plt.plot(x, y, 'b-', linewidth=2, label=r"$f(x) = \sin(x)$")
```

(continues on next page)

(continued from previous page)

```
plt.plot(x, np.cos(x), 'g--', linewidth=2, label=r"$f'(x) = \cos(x)$")
plt.plot(x, -np.sin(x), 'r:', linewidth=2, label=r"$f''(x) = -\sin(x)$")
plt.xlabel('x')
plt.ylabel('y')
plt.title('Function and Its Derivatives')
plt.legend()
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
plt.plot(x, y_double_prime_exact, 'r-', linewidth=2, label='Exact')
plt.plot(x, y_double_prime_numerical, 'b--', linewidth=2, label=f'Numerical (h={h})')
plt.xlabel('x')
plt.ylabel(r"$f''(x)$")
plt.title('Second Derivative: Exact vs Numerical')
plt.legend()
plt.grid(True, alpha=0.3)

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





5.5 IV. Error Analysis: The Optimal Step Size

Numerical differentiation faces two competing errors:

5.5.1 1. Truncation Error

From truncating the Taylor series. Decreases as h decreases.

- Forward/Backward: $\sim \frac{h}{2}|f''(x)|$
- Central: $\sim \frac{h^2}{6}|f'''(x)|$

5.5.2 2. Round-off Error

From finite precision arithmetic. Increases as h decreases!

When subtracting nearly equal numbers: $f(x+h) \approx f(x)$ for small h.

Round-off error $\sim \frac{\epsilon|f(x)|}{h}$ where $\epsilon \approx 10^{-16}$ for double precision.

5.5.3 Optimal h

For central difference, balancing truncation and round-off:

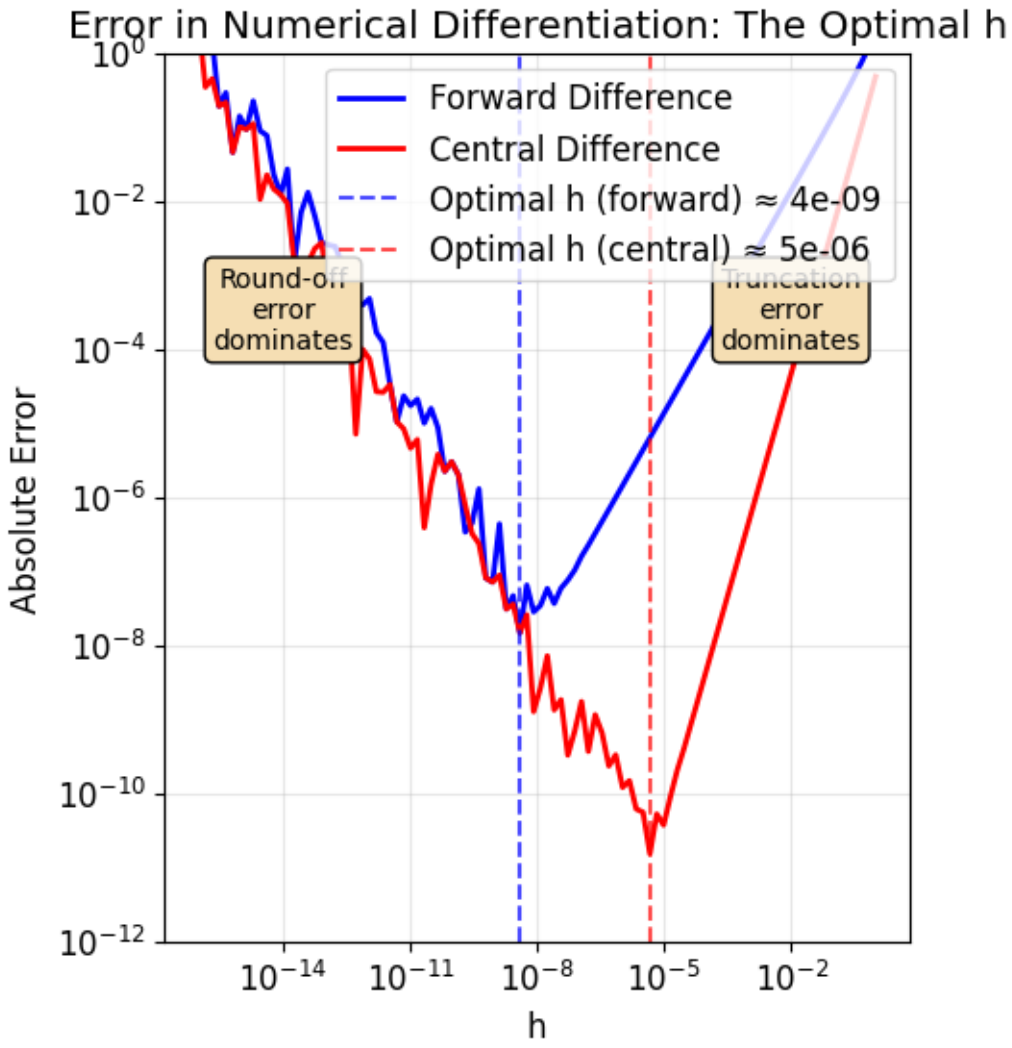
$$h_{opt} \sim \epsilon^{1/3} \approx 10^{-5} \text{ to } 10^{-6}$$

```
# Demonstrate the truncation vs round-off trade-off
def f(x):
    return np.exp(x)

def f_prime(x):
    return np.exp(x)

x0 = 1.0
true_value = f_prime(x0)

h_values = np.logspace(-16, 0, 100)
errors_fwd = np.abs(forward_diff(f, x0, h_values) - true_value)
errors_ctr = np.abs(central_diff(f, x0, h_values) - true_value)

# Find optimal h
opt_h_fwd = h_values[np.argmin(errors_fwd)]
opt_h_ctr = h_values[np.argmin(errors_ctr)]

plt.figure(figsize=(5, 6))
plt.loglog(h_values, errors_fwd, 'b-', linewidth=2, label='Forward Difference')
plt.loglog(h_values, errors_ctr, 'r-', linewidth=2, label='Central Difference')

plt.axvline(opt_h_fwd, color='b', linestyle='--', alpha=0.7,
            label=f'Optimal h (forward) ≈ {opt_h_fwd:.0e}')
plt.axvline(opt_h_ctr, color='r', linestyle='--', alpha=0.7,
            label=f'Optimal h (central) ≈ {opt_h_ctr:.0e}')

# Annotate regions
plt.annotate('Truncation\nerror\ndominates', xy=(1e-2, 1e-4), fontsize=10,
             ha='center', bbox=dict(boxstyle='round', facecolor='wheat'))
plt.annotate('Round-off\nerror\ndominates', xy=(1e-14, 1e-4), fontsize=10,
             ha='center', bbox=dict(boxstyle='round', facecolor='wheat'))

plt.xlabel('h')
plt.ylabel('Absolute Error')
plt.title('Error in Numerical Differentiation: The Optimal h')
plt.legend(loc='upper right')
plt.grid(True, alpha=0.3)
plt.ylim(1e-12, 1)
plt.show()

print(f"Optimal h for forward difference: {opt_h_fwd:.2e}")
print(f"Minimum error: {errors_fwd.min():.2e}")
```

(continues on next page)

(continued from previous page)

```
print(f"\nOptimal h for central difference: {opt_h_ctr:.2e}")
print(f"Minimum error: {errors_ctr.min():.2e}")
print(f"\nCentral difference achieves ~{errors_fwd.min()/errors_ctr.min():.0f}x␣
↪better accuracy!")
```




```
Optimal h for forward difference: 3.94e-09
Minimum error: 1.46e-08

Optimal h for central difference: 4.64e-06
Minimum error: 1.53e-11

Central difference achieves ~960x better accuracy!
```

5.6 V. Higher-Order Formulas

We can improve accuracy by using more points. These are derived from Taylor series or polynomial interpolation.

5.6.1 Fourth-Order Central Difference (5-point stencil)

$$f'(x) \approx \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}$$

Error is $O(h^4)$ — much better than basic central difference!

5.6.2 Fourth-Order Second Derivative

$$f''(x) \approx \frac{-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)}{12h^2}$$

```
def central_diff_4th_order(f, x, h):
    """Fourth-order central difference approximation of f'(x)."""
    return (-f(x + 2*h) + 8*f(x + h) - 8*f(x - h) + f(x - 2*h)) / (12 * h)

def second_deriv_4th_order(f, x, h):
    """Fourth-order central difference approximation of f''(x)."""
    return (-f(x + 2*h) + 16*f(x + h) - 30*f(x) + 16*f(x - h) - f(x - 2*h)) / (12 *␣
 ↪h**2)

# Compare second-order vs fourth-order
def f(x):
    return np.sin(x)

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

x0 = 1.0
true_value = f_prime(x0)

print("Comparison: 2nd-Order vs 4th-Order Central Difference")
print(f"Function: f(x) = sin(x) at x = {x0}")
print(f"True f'(x): {true_value:.15f}")
print("=" * 60)
print(f"{'h':>10} {'2nd-Order Error':>20} {'4th-Order Error':>20}")
print("-" * 60)

for h in [0.1, 0.05, 0.02, 0.01, 0.005]:
    err_2nd = abs(central_diff(f, x0, h) - true_value)
    err_4th = abs(central_diff_4th_order(f, x0, h) - true_value)
    print(f"{h:>10.3f} {err_2nd:>20.2e} {err_4th:>20.2e}")

print("\n4th-order is dramatically more accurate!")
```

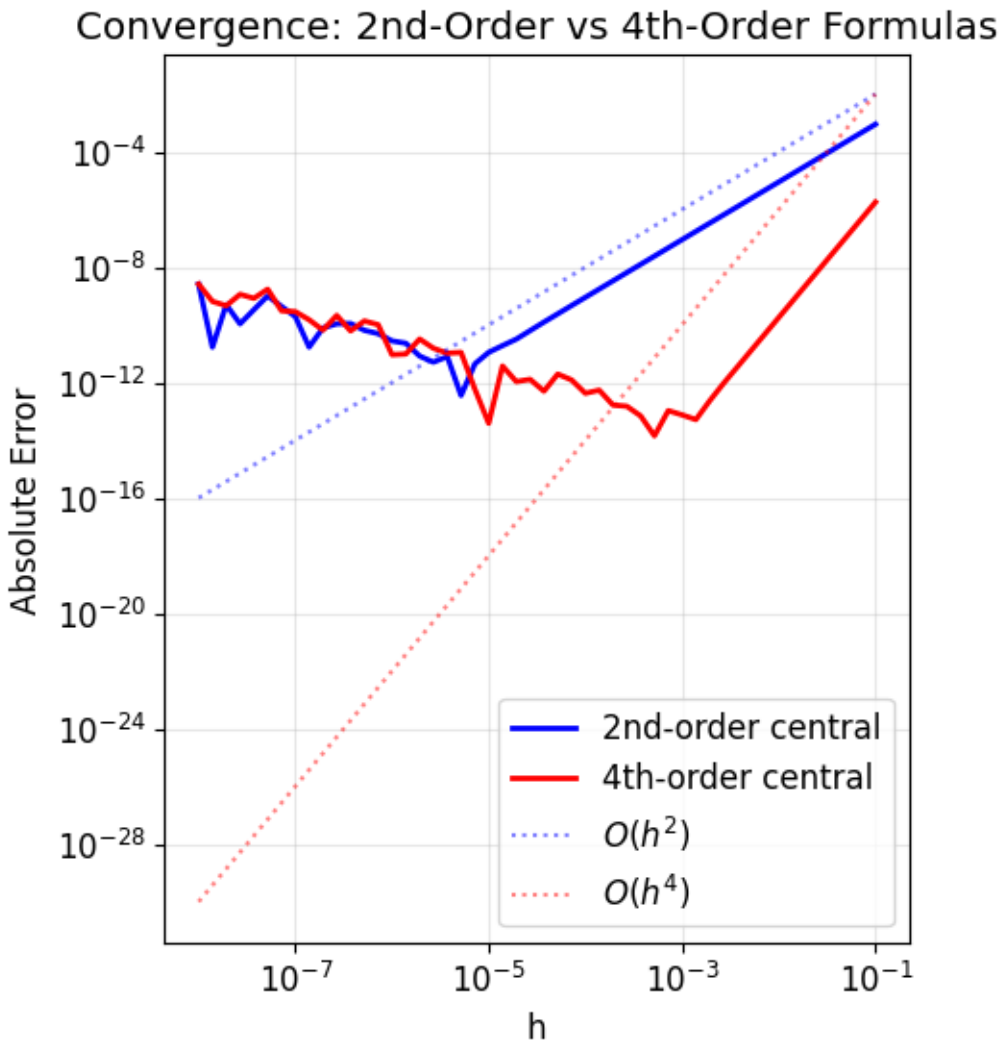
```
Comparison: 2nd-Order vs 4th-Order Central Difference
Function: f(x) = sin(x) at x = 1.0
True f'(x): 0.540302305868140
============================================================
         h      2nd-Order Error      4th-Order Error
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------
     0.100             9.00e-04             1.80e-06
     0.050             2.25e-04             1.13e-07
     0.020             3.60e-05             2.88e-09
     0.010             9.00e-06             1.80e-10
     0.005             2.25e-06             1.13e-11

4th-order is dramatically more accurate!
```

```python
# Visualize convergence of different orders
h_values = np.logspace(-8, -1, 50)

errors_2nd = np.abs(central_diff(f, x0, h_values) - true_value)
errors_4th = np.abs(central_diff_4th_order(f, x0, h_values) - true_value)

plt.figure(figsize=(5, 6))
plt.loglog(h_values, errors_2nd, 'b-', linewidth=2, label='2nd-order central')
plt.loglog(h_values, errors_4th, 'r-', linewidth=2, label='4th-order central')

# Reference lines
plt.loglog(h_values, h_values**2, 'b:', alpha=0.5, label=r'$O(h^2)$')
plt.loglog(h_values, h_values**4 * 100, 'r:', alpha=0.5, label=r'$O(h^4)$')

plt.xlabel('h')
plt.ylabel('Absolute Error')
plt.title('Convergence: 2nd-Order vs 4th-Order Formulas')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

print("Observation:")
print("- 2nd-order: halving h reduces error by 4x")
print("- 4th-order: halving h reduces error by 16x!")
```




```
Observation:
- 2nd-order: halving h reduces error by 4x
- 4th-order: halving h reduces error by 16x!
```

5.7 VI. Derivatives from Data

In experimental physics, we often have data points (x_i, y_i) rather than a function.

5.7.1 Approaches:

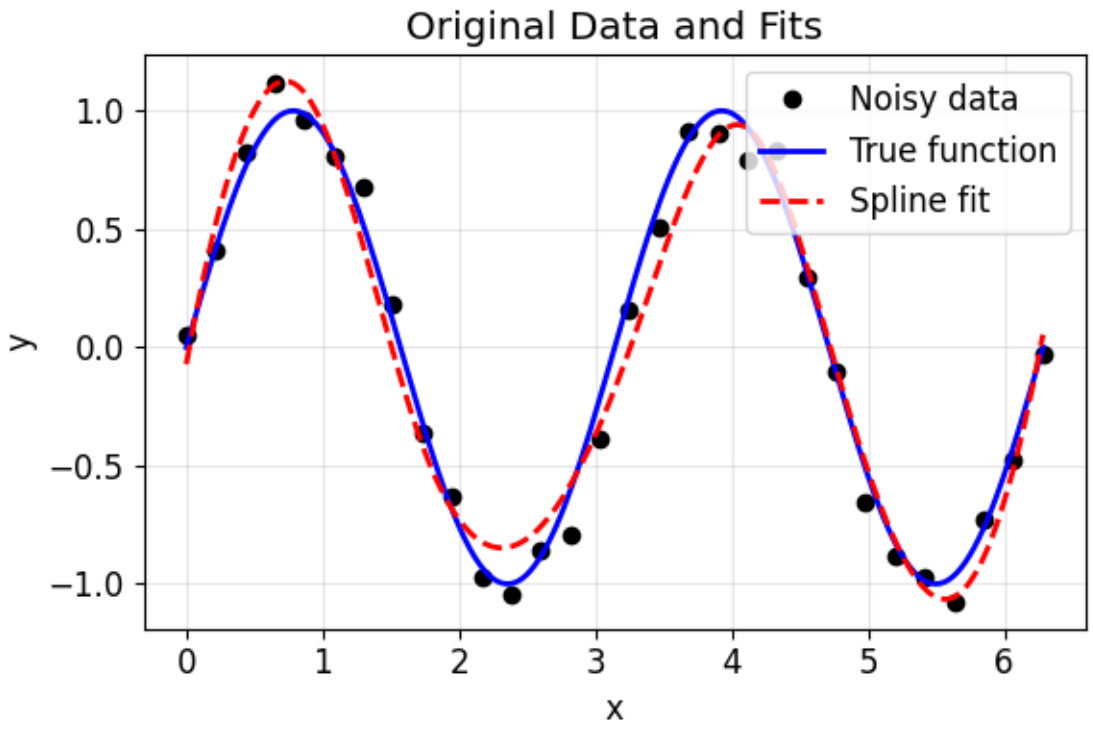
1. **Direct finite differences** — Simple but noisy
2. **Fit a smooth function** — Then differentiate analytically
3. **Spline interpolation** — Smooth and flexible
4. **Savitzky-Golay filter** — Smoothing + differentiation

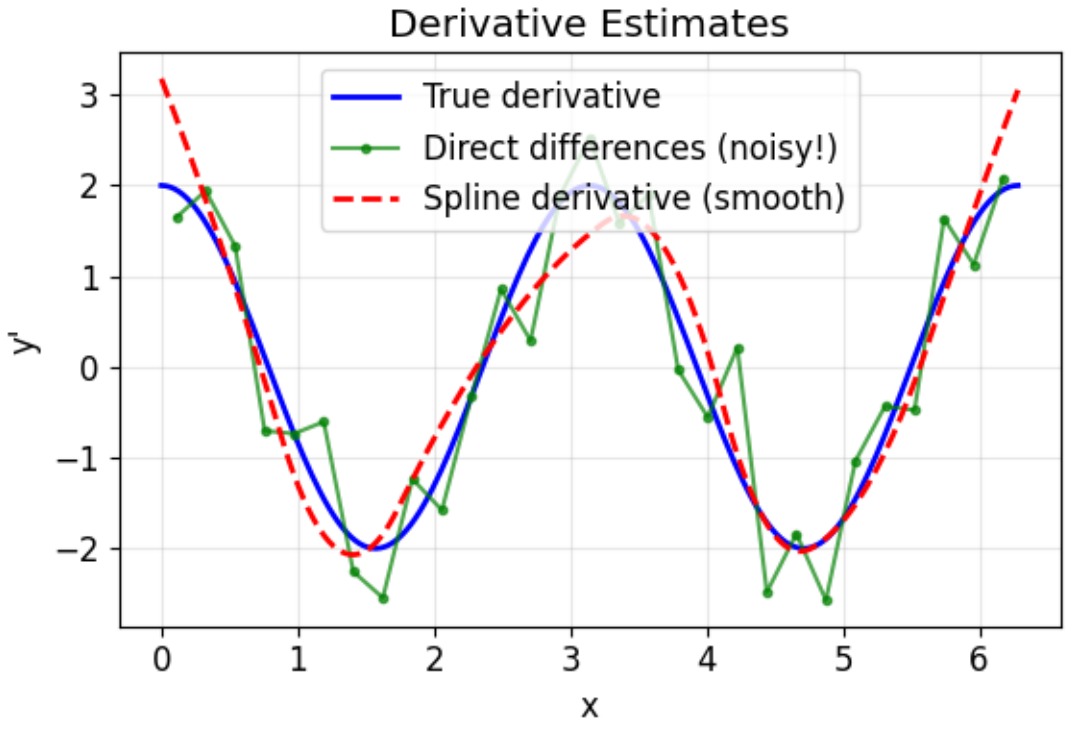
```
# Generate noisy experimental data
np.random.seed(42)

# True function
def true_func(x):
    return np.sin(2 * x)

def true_derivative(x):
    return 2 * np.cos(2 * x)

# Simulated experimental data
x_data = np.linspace(0, 2*np.pi, 30)
y_data = true_func(x_data) + 0.1 * np.random.randn(len(x_data))  # Add noise

# Method 1: Direct finite differences
dy_direct = np.diff(y_data) / np.diff(x_data)
x_direct = (x_data[:-1] + x_data[1:]) / 2  # Midpoints

# Method 2: Central differences (for interior points)
dy_central = (y_data[2:] - y_data[:-2]) / (x_data[2:] - x_data[:-2])
x_central = x_data[1:-1]

# Method 3: Spline interpolation
spline = UnivariateSpline(x_data, y_data, s=0.5)  # s controls smoothing
x_fine = np.linspace(0, 2*np.pi, 200)
dy_spline = spline.derivative()(x_fine)

# Plot results
plt.figure(figsize=(6, 8))

plt.subplot(2, 1, 1)
plt.plot(x_data, y_data, 'ko', markersize=6, label='Noisy data')
plt.plot(x_fine, true_func(x_fine), 'b-', linewidth=2, label='True function')
plt.plot(x_fine, spline(x_fine), 'r--', linewidth=2, label='Spline fit')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Original Data and Fits')
plt.legend()
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
plt.plot(x_fine, true_derivative(x_fine), 'b-', linewidth=2, label='True derivative')
plt.plot(x_direct, dy_direct, 'g.-', alpha=0.7, label='Direct differences (noisy!)')
plt.plot(x_fine, dy_spline, 'r--', linewidth=2, label='Spline derivative (smooth)')
plt.xlabel('x')
plt.ylabel("y'")
plt.title('Derivative Estimates')
plt.legend()
plt.grid(True, alpha=0.3)
```

(continues on next page)

(continued from previous page)

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

print("Key insight: Direct finite differences amplify noise!")
print("Spline interpolation provides a much smoother derivative.")
```






```
Key insight: Direct finite differences amplify noise!
Spline interpolation provides a much smoother derivative.
```

5.7.2 Savitzky-Golay Filter

A popular method that simultaneously smooths data AND computes derivatives.

It fits a polynomial to a moving window and extracts the derivative from the polynomial coefficients.

We will cover this next week.

5.8 VII. Using `np.gradient()` for Array Data

For arrays of data, NumPy's `np.gradient()` is the go-to tool:

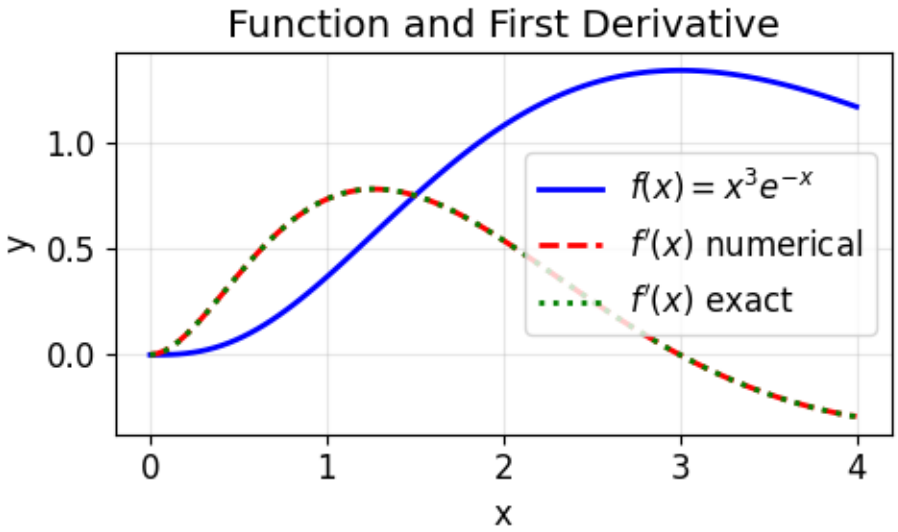
- Computes centered differences for interior points
- Handles edges automatically (uses forward/backward differences)
- Works with non-uniform spacing

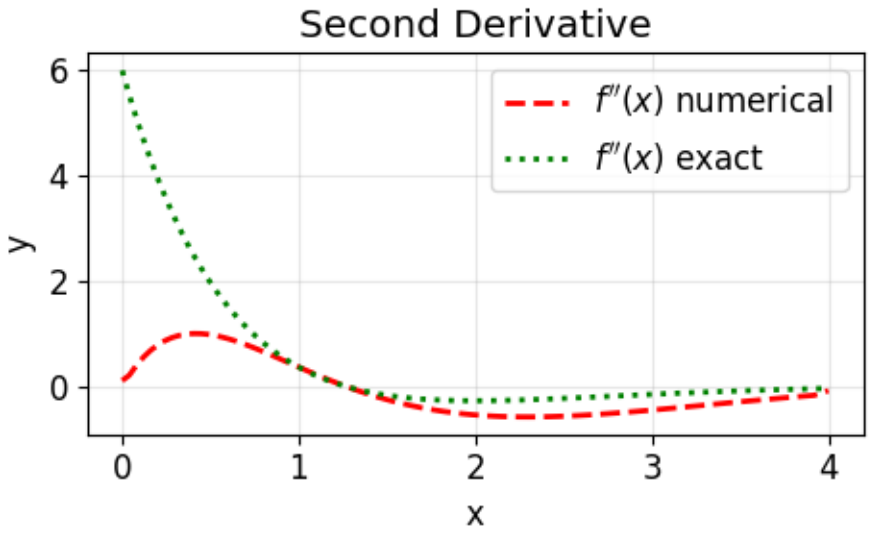
```
# Example: Using np.gradient for derivatives
def f(x):
    return x**3 * np.exp(-x)

x = np.linspace(0, 4, 100)
y = f(x)

# np.gradient computes centered differences, handling edges appropriately
# Syntax: np.gradient(y, x) where x is the coordinate array
dy = np.gradient(y, x)   # First derivative
ddy = np.gradient(dy, x) # Second derivative (apply gradient twice)

# Exact derivatives for comparison
# f(x) = x³e^(-x)
# f'(x) = x²e^(-x)(3-x)
# f''(x) = e^(-x)(x² - 6x + 6)
dy_exact = x**2 * np.exp(-x) * (3 - x)
ddy_exact = np.exp(-x) * (x**2 - 6*x + 6)

plt.figure(figsize=(5, 6))

plt.subplot(2, 1, 1)
plt.plot(x, y, 'b-', linewidth=2, label=r'$f(x) = x^3 e^{-x}$')
plt.plot(x, dy, 'r--', linewidth=2, label=r"$f'(x)$ numerical")
plt.plot(x, dy_exact, 'g:', linewidth=2, label=r"$f'(x)$ exact")
plt.xlabel('x')
plt.ylabel('y')
plt.title('Function and First Derivative')
plt.legend()
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
plt.plot(x, ddy, 'r--', linewidth=2, label=r"$f''(x)$ numerical")
plt.plot(x, ddy_exact, 'g:', linewidth=2, label=r"$f''(x)$ exact")
plt.xlabel('x')
plt.ylabel('y')
plt.title('Second Derivative')
plt.legend()
plt.grid(True, alpha=0.3)
```

(continues on next page)

(continued from previous page)

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

# Show the error
print("Maximum error in first derivative:", np.max(np.abs(dy - dy_exact)))
print("Maximum error in second derivative:", np.max(np.abs(ddy - ddy_exact)))
```






```
Maximum error in first derivative: 0.0030302710209441086
Maximum error in second derivative: 5.889734082400933
```

5.9 VIII. Physics Applications

Let's apply numerical differentiation to real physics problems!

5.9.1 1. Heat Transfer: Temperature Gradient

The heat flux is proportional to the temperature gradient (Fourier's Law):

$$q = -k\frac{dT}{dx}$$

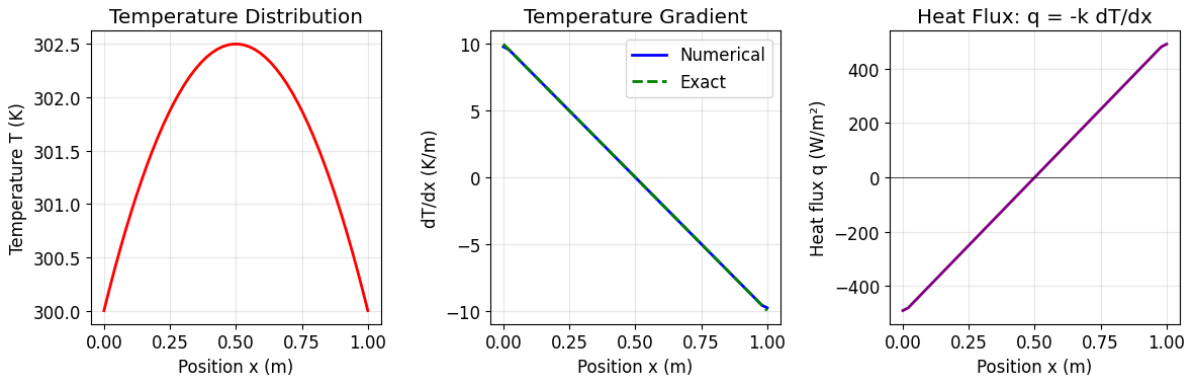
where k is the thermal conductivity.

```
# Temperature distribution in a rod with heat source
# Steady state: k*d²T/dx² + Q = 0
# Solution with T(0)=T(L)=T0: T(x) = T0 + (Q/2k)*x*(L-x)

L = 1.0       # Rod length (m)
T0 = 300      # Boundary temperature (K)
k_thermal = 50  # Thermal conductivity (W/m·K)
Q = 1000      # Heat source (W/m³)

x = np.linspace(0, L, 50)

# Temperature distribution
T = T0 + (Q / (2 * k_thermal)) * x * (L - x)

# Temperature gradient dT/dx
dT_dx = np.gradient(T, x)
dT_dx_exact = (Q / (2 * k_thermal)) * (L - 2*x)

# Heat flux q = -k dT/dx
q = -k_thermal * dT_dx

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)
plt.plot(x, T, 'r-', linewidth=2)
plt.xlabel('Position x (m)')
plt.ylabel('Temperature T (K)')
plt.title('Temperature Distribution')
plt.grid(True, alpha=0.3)

plt.subplot(1, 3, 2)
plt.plot(x, dT_dx, 'b-', linewidth=2, label='Numerical')
plt.plot(x, dT_dx_exact, 'g--', linewidth=2, label='Exact')
plt.xlabel('Position x (m)')
plt.ylabel('dT/dx (K/m)')
plt.title('Temperature Gradient')
plt.legend()
plt.grid(True, alpha=0.3)

plt.subplot(1, 3, 3)
plt.plot(x, q, 'purple', linewidth=2)
plt.axhline(0, color='k', linestyle='-', linewidth=0.5)
plt.xlabel('Position x (m)')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Heat flux q (W/m²)')
plt.title('Heat Flux: q = -k dT/dx')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("Observations:")
print("- Temperature peaks at the center (heat generated throughout rod)")
print("- Gradient is positive on left (heat flows left), negative on right (heat
 ↪flows right)")
print("- Heat flows from center outward toward the cooled boundaries")
```




```
Observations:
- Temperature peaks at the center (heat generated throughout rod)
- Gradient is positive on left (heat flows left), negative on right (heat flows
 ↪right)
- Heat flows from center outward toward the cooled boundaries
```

5.9.2 2. Wave Properties: Group Velocity

The group velocity of a wave packet is:

$$v_g = \frac{d\omega}{dk}$$

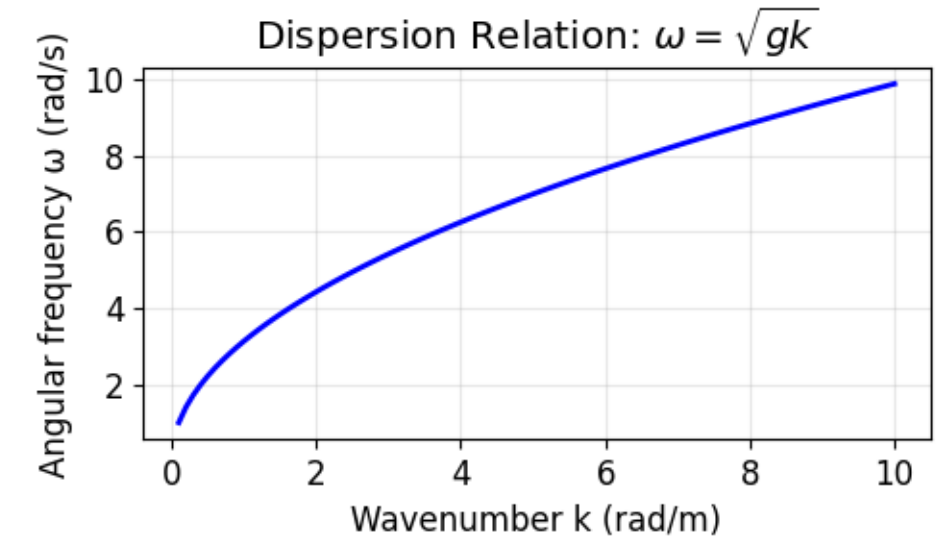
where $\omega(k)$ is the dispersion relation.

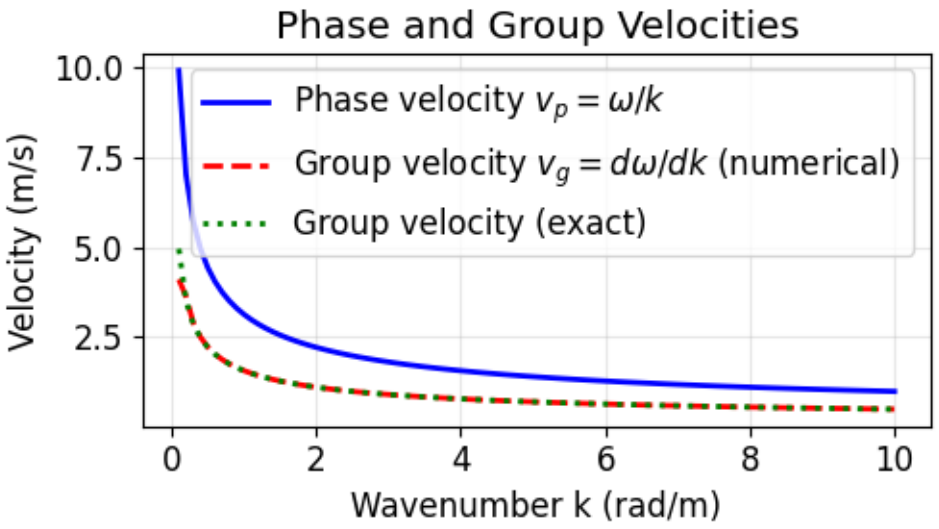
```
# Dispersion relation for deep water waves: ω = √(gk)
g = 9.8  # m/s²

k = np.linspace(0.1, 10, 100)  # Wavenumber (rad/m)
omega = np.sqrt(g * k)  # Angular frequency (rad/s)

# Phase velocity: v_p = ω/k
v_phase = omega / k

# Group velocity: v_g = dω/dk
v_group_numerical = np.gradient(omega, k)

# Exact: v_g = d(√(gk))/dk = (1/2)√(g/k) = v_p/2
```

(continues on next page)

(continued from previous page)

```
v_group_exact = 0.5 * np.sqrt(g / k)

plt.figure(figsize=(5, 6))

plt.subplot(2, 1, 1)
plt.plot(k, omega, 'b-', linewidth=2)
plt.xlabel('Wavenumber k (rad/m)')
plt.ylabel('Angular frequency ω (rad/s)')
plt.title(r'Dispersion Relation: $\omega = \sqrt{gk}$')
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
plt.plot(k, v_phase, 'b-', linewidth=2, label='Phase velocity $v_p = ω/k$')
plt.plot(k, v_group_numerical, 'r--', linewidth=2, label='Group velocity $v_g = dω/dk
 ↪$ (numerical)')
plt.plot(k, v_group_exact, 'g:', linewidth=2, label='Group velocity (exact)')
plt.xlabel('Wavenumber k (rad/m)')
plt.ylabel('Velocity (m/s)')
plt.title('Phase and Group Velocities')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("Key insight: For deep water waves, v_group = v_phase / 2")
print("Energy travels at half the speed of the wave crests!")
```






```
Key insight: For deep water waves, v_group = v_phase / 2
Energy travels at half the speed of the wave crests!
```

CHAPTER

SIX

INTERPOLATION

In this lecture, we'll learn how to **estimate values between known data points** — a fundamental technique in computational physics.

6.1 Why Interpolation?

- Experimental data is measured at discrete points, but physics is continuous
- We need to estimate values between measurements
- Creating smooth curves through data for visualization
- Building functions from tabulated data (e.g., material properties)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate

# For nicer plots
plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 10
```

6.2 I. The Interpolation Problem

Given $n+1$ data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, find a function $f(x)$ such that:

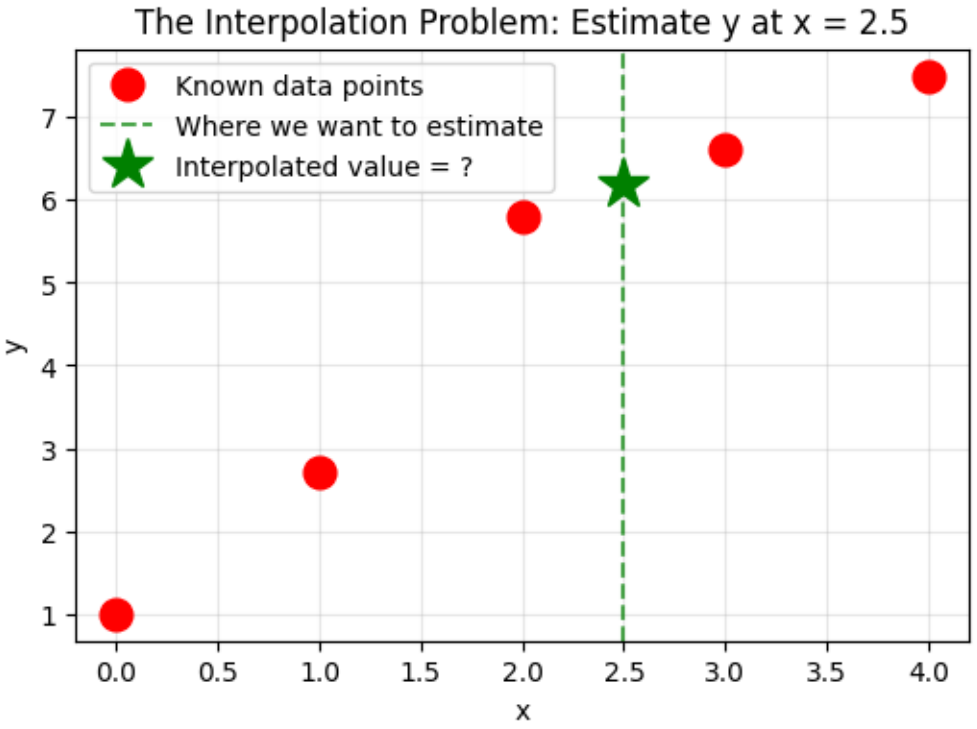
$$f(x_i) = y_i \quad \text{for all } i = 0, 1, \ldots, n$$

The function $f(x)$ **passes exactly through** all data points.

6.2.1 Interpolation vs. Curve Fitting

Interpolation	Curve Fitting (Regression)
Passes through ALL points	Minimizes overall error
No measurement error assumed	Accounts for noise
Good for exact data	Good for noisy data
Can oscillate wildly	Smoother results

```
# Visualize the interpolation problem
# Given these data points, what's the value at x=2.5?

x_data = np.array([0, 1, 2, 3, 4])
y_data = np.array([1, 2.7, 5.8, 6.6, 7.5])

plt.figure(figsize=(6, 4))
plt.plot(x_data, y_data, 'ro', markersize=12, label='Known data points')
plt.axvline(x=2.5, color='g', linestyle='--', alpha=0.7, label='Where we want to␣
↪estimate')
plt.plot(2.5, 6.2, 'g*', markersize=20, label='Interpolated value = ?')

plt.xlabel('x')
plt.ylabel('y')
plt.title('The Interpolation Problem: Estimate y at x = 2.5')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```



6.3 II. Linear Interpolation

The simplest approach: connect adjacent points with straight lines.

For a point x between x_i and x_{i+1}:

$$f(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i)$$

This is just the equation of a line through two points! The slope is:

$$m = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

This is exactly what we used in the trapezoidal rule for integration.

```
# First, let's implement linear interpolation from scratch
def linear_interp(x_data, y_data, xnew):
    """
    Linear interpolation - implemented from scratch.

    Parameters:
        x_data: array of x coordinates of known points
        y_data: array of y coordinates of known points
        xnew: the x value where we want to interpolate

    Returns:
        Interpolated y value at xnew
    """
    # Step 1: Find the interval where xnew falls
    # We need to find i such that x_data[i] <= xnew <= x_data[i+1]
    i = 0
    while xnew > x_data[i+1]:
        i += 1

    # Step 2: Calculate the slope (coefficient a in y = ax + b)
    a = (y_data[i+1] - y_data[i]) / (x_data[i+1] - x_data[i])

    # Step 3: Calculate the intercept
    b = y_data[i] - a*x_data[i]

    # Step 4: Evaluate at the new point
    ynew = a*xnew + b

    return ynew

# Test with simple data: y = x^2
x = [1, 2, 3, 4]
y = [1, 4, 9, 16]

print("Testing our linear interpolation:")
print(f"Known points: x = {x}, y = {y}")
print(f"Interpolated value at x=2.5: {linear_interp(x, y, 2.5)}")
print(f"True value (2.5²): {2.5**2}")
```

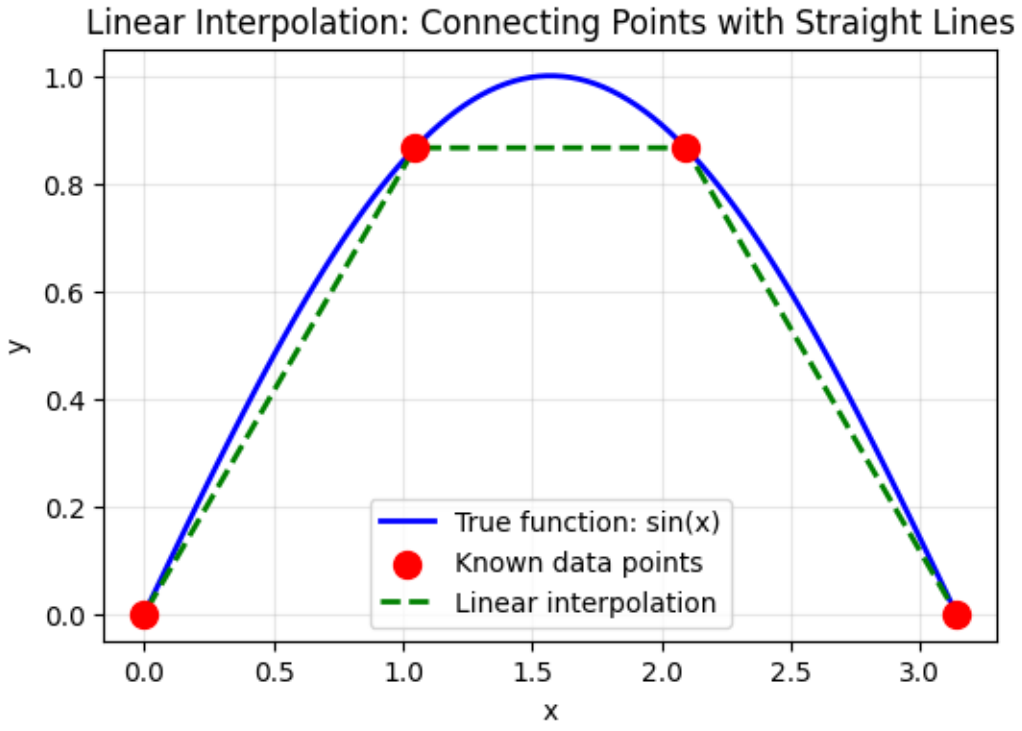
```
Testing our linear interpolation:
Known points: x = [1, 2, 3, 4], y = [1, 4, 9, 16]
Interpolated value at x=2.5: 6.5
True value (2.5²): 6.25
```

```
# Visualize linear interpolation on sin(x)
from math import pi

f = lambda x: np.sin(x)
x_min, x_max = 0, pi
npt_ideal = 100  # Points for ideal function
npt_known = 4    # Available data points
```

(continues on next page)

(continued from previous page)

```
x_ideal = np.linspace(x_min, x_max, npt_ideal)
x_known = np.linspace(x_min, x_max, npt_known)
y_known = f(x_known)

# Use our linear interpolation
x_interp = np.linspace(x_min + 0.01, x_max - 0.01, 50)
y_interp = np.array([linear_interp(x_known, y_known, xi) for xi in x_interp])

plt.figure(figsize=(6, 4))
plt.plot(x_ideal, f(x_ideal), 'b-', linewidth=2, label='True function: sin(x)')
plt.scatter(x_known, y_known, color='r', s=100, zorder=5, label='Known data points')
plt.plot(x_interp, y_interp, 'g--', linewidth=2, label='Linear interpolation')

plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Linear Interpolation: Connecting Points with Straight Lines')
plt.grid(True, alpha=0.3)
plt.show()

print("Notice: Linear interpolation creates 'corners' at data points - it's not␣
↪smooth!")
```




```
Notice: Linear interpolation creates 'corners' at data points - it's not smooth!
```

6.3.1 Pros and Cons of Linear Interpolation

Pros:

- Simple and fast
- Always stable (no oscillations)
- Works well for many practical cases

Cons:

- Not smooth — derivative is discontinuous at data points
- May not capture curved behavior well

6.4 III. Polynomial Interpolation

Fact: Given $n + 1$ distinct points, there exists a **unique** polynomial of degree at most n that passes through all points.

6.4.1 Lagrange Interpolation

The Lagrange form expresses the interpolating polynomial as:

$$P(x) = \sum_{i=0}^{n} y_i \cdot L_i(x)$$

where the **Lagrange basis polynomials** are:

$$L_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Key property: $L_i(x_j) = \delta_{ij}$ (1 if $i = j$, 0 otherwise)

Advantages:

- Simple to understand and implement
- Directly constructs a polynomial through all data points

Disadvantages:

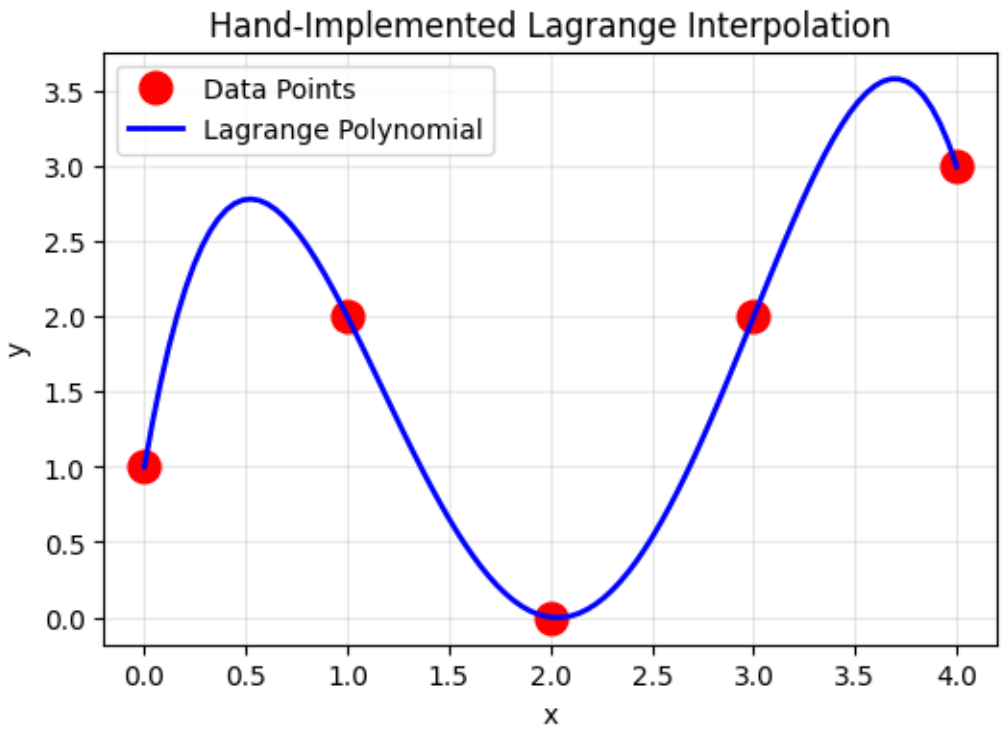
- Computational cost increases with number of points
- Not easily extendable if new points are added

```
# Implement Lagrange interpolation from scratch
def lagrange_interpolation(x_points, y_points, x_value):
    """
    Compute the Lagrange interpolated value at x_value.

    The Lagrange polynomial is:
    P(x) = Σ y_j * L_j(x)

    where L_j(x) = Π (x - x_i)/(x_j - x_i) for i ≠ j

    Parameters:
        x_points: array of x coordinates of data points
```

(continues on next page)

(continued from previous page)

```
        y_points: array of y coordinates of data points
        x_value: the x value at which to evaluate

    Returns:
        Interpolated y value at x_value
    """
    total = 0.0
    n = len(x_points)
    for i in range(n):
        # compuite L_i
        Li = 1.0
        for j in range(n):
            if j != i:
                Li = Li * (x_value - x_points[j]) / (x_points[i] - x_points[j])
        total = total + Li * y_points[i]

    return total

# Example data points
x_points = np.array([0, 1, 2, 3, 4])
y_points = np.array([1, 2, 0, 2, 3])

# Create points for plotting
x_new = np.linspace(min(x_points), max(x_points), 100)
y_new = [lagrange_interpolation(x_points, y_points, xv) for xv in x_new]

# Plotting
plt.figure(figsize=(6, 4))
plt.plot(x_points, y_points, 'ro', markersize=12, label='Data Points')
plt.plot(x_new, y_new, 'b-', linewidth=2, label='Lagrange Polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Hand-Implemented Lagrange Interpolation')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

print(f"Lagrange interpolation at x=2.5: {lagrange_interpolation(x_points, y_points,␣
↪2.5):.4f}")
```



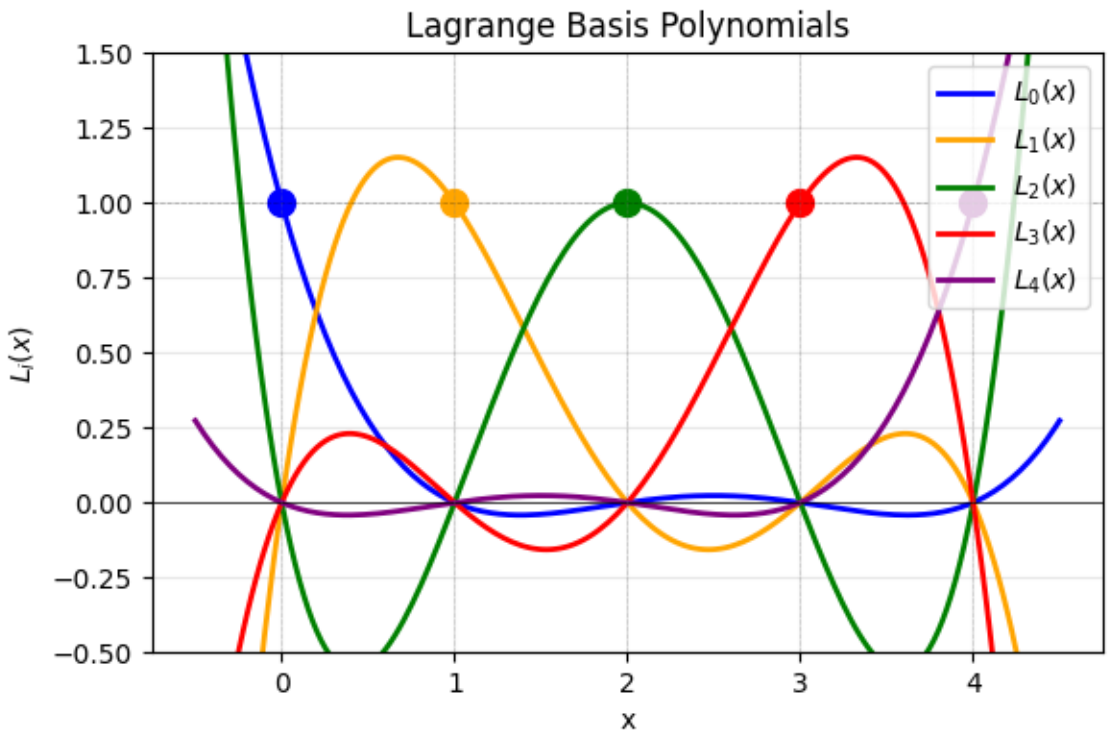

```
Lagrange interpolation at x=2.5: 0.5312
```

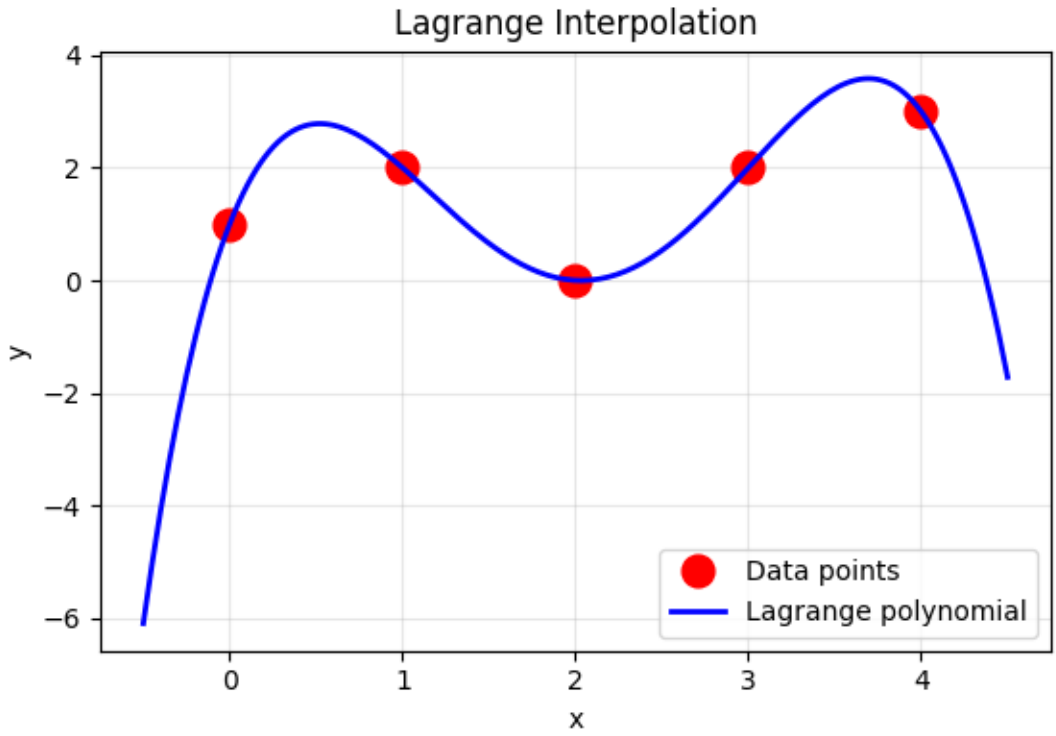
```python
def lagrange_basis(x_data, i, x):
    """
    Compute the i-th Lagrange basis polynomial at x.
    """
    n = len(x_data)
    result = 1.0
    for j in range(n):
        if j != i:
            result *= (x - x_data[j]) / (x_data[i] - x_data[j])
    return result

def lagrange_interp(x_data, y_data, x):
    """
    Lagrange polynomial interpolation.
    """
    n = len(x_data)
    result = 0.0
    for i in range(n):
        result += y_data[i] * lagrange_basis(x_data, i, x)
    return result

# Visualize Lagrange basis polynomials
x_plot = np.linspace(-0.5, 4.5, 200)
x_data = x_points
y_data = y_points

plt.figure(figsize=(6, 8))

plt.subplot(2, 1, 1)
```

(continues on next page)

(continued from previous page)

```
colors = ['blue', 'orange', 'green', 'red', 'purple']
for i in range(len(x_data)):
    L_i = [lagrange_basis(x_data, i, x) for x in x_plot]
    plt.plot(x_plot, L_i, color=colors[i], linewidth=2, label=f'$L_{i}(x)$')
    plt.plot(x_data[i], 1, 'o', color=colors[i], markersize=10)

plt.axhline(y=0, color='k', linewidth=0.5)
plt.axhline(y=1, color='k', linewidth=0.5, linestyle='--', alpha=0.3)
for xi in x_data:
    plt.axvline(x=xi, color='gray', linewidth=0.5, linestyle='--', alpha=0.3)
plt.xlabel('x')
plt.ylabel('$L_i(x)$')
plt.title('Lagrange Basis Polynomials')
plt.legend()
plt.grid(True, alpha=0.3)
plt.ylim(-0.5, 1.5)

plt.subplot(2, 1, 2)
# Interpolation result
y_lagrange = [lagrange_interp(x_data, y_data, x) for x in x_plot]
plt.plot(x_data, y_data, 'ro', markersize=12, label='Data points')
plt.plot(x_plot, y_lagrange, 'b-', linewidth=2, label='Lagrange polynomial')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Lagrange Interpolation')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Lagrange interpolation at x=2.5: {lagrange_interp(x_data, y_data, 2.5):.4f}")
```






```
Lagrange interpolation at x=2.5: 0.5312
```

6.4.2 Newton's Divided Difference Form

An equivalent but more efficient form:

$$P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots$$

where a_i are the **divided differences**:

$$a_0 = f[x_0] = y_0$$

$$a_1 = f[x_0, x_1] = \frac{y_1 - y_0}{x_1 - x_0}$$

$$a_2 = f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

Advantage: Adding a new point only requires computing one new coefficient!

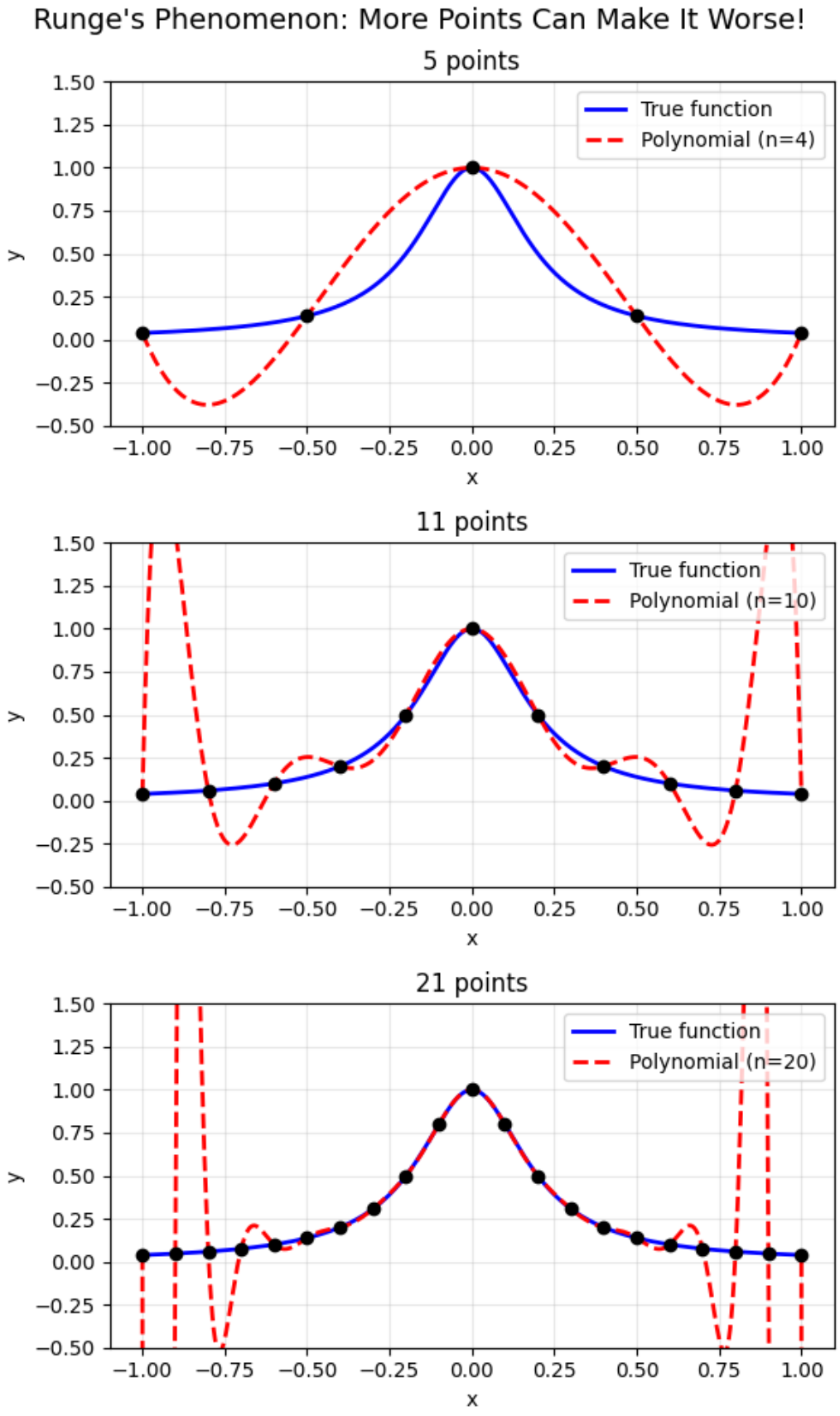
```
def divided_differences(x_data, y_data):
    """
    Compute divided difference coefficients.
    """
    n = len(x_data)
    # Create divided difference table
    dd = np.zeros((n, n))
    dd[:, 0] = y_data

    for j in range(1, n):
        for i in range(n - j):
            dd[i, j] = (dd[i+1, j-1] - dd[i, j-1]) / (x_data[i+j] - x_data[i])

    return dd[0, :]  # First row contains the coefficients

def newton_interp(x_data, y_data, x):
    """
    Newton's divided difference interpolation.
    """
    coeffs = divided_differences(x_data, y_data)
    n = len(x_data)

    # Evaluate polynomial using Horner's method
    result = coeffs[-1]
    for i in range(n-2, -1, -1):
        result = result * (x - x_data[i]) + coeffs[i]

    return result

# Test Newton interpolation
coeffs = divided_differences(x_data, y_data)
print("Divided difference coefficients:")
for i, c in enumerate(coeffs):
    print(f"  a_{i} = {c:.4f}")

print(f"\nNewton interpolation at x=2.5: {newton_interp(x_data, y_data, 2.5):.4f}")
print(f"Lagrange interpolation at x=2.5: {lagrange_interp(x_data, y_data, 2.5):.4f}")
print("(They should be identical!)")
```

```
Divided difference coefficients:
  a_0 = 1.0000
  a_1 = 1.0000
  a_2 = -1.5000
  a_3 = 1.1667
  a_4 = -0.5000

Newton interpolation at x=2.5: 0.5312
Lagrange interpolation at x=2.5: 0.5312
(They should be identical!)
```

6.4.3 The Danger of High-Degree Polynomials: Runge's Phenomenon

High-degree polynomial interpolation can produce **wild oscillations** near the edges, especially for uniformly spaced points.

```python
# Demonstrate Runge's phenomenon
def runge_function(x):
    """The Runge function: 1/(1 + 25x²)"""
    return 1 / (1 + 25 * x**2)

x_fine = np.linspace(-1, 1, 500)
y_true = runge_function(x_fine)

plt.figure(figsize=(6, 10))

for idx, n_points in enumerate([5, 11, 21]):
    plt.subplot(3, 1, idx + 1)

    # Uniformly spaced points
    x_uniform = np.linspace(-1, 1, n_points)
    y_uniform = runge_function(x_uniform)

    # Polynomial interpolation
    y_interp = [lagrange_interp(x_uniform, y_uniform, x) for x in x_fine]

    plt.plot(x_fine, y_true, 'b-', linewidth=2, label='True function')
    plt.plot(x_fine, y_interp, 'r--', linewidth=2, label=f'Polynomial (n={n_points-1})
↪')
    plt.plot(x_uniform, y_uniform, 'ko', markersize=6)

    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(f'{n_points} points')
    plt.legend(loc='upper right')
    plt.ylim(-0.5, 1.5)
    plt.grid(True, alpha=0.3)

plt.suptitle("Runge's Phenomenon: More Points Can Make It Worse!", fontsize=14)
plt.tight_layout()
plt.show()

print("Notice: With 21 points, the interpolation is WORSE near the edges!")
print("This is Runge's phenomenon — a fundamental limitation of polynomial␣
↪interpolation.")
```


Runge's Phenomenon: More Points Can Make It Worse!
5 points
True function
Polynomial (n=4)
11 points
True function
Polynomial (n=10)
21 points
True function
Polynomial (n=20)
x
y

```
Notice: With 21 points, the interpolation is WORSE near the edges!
This is Runge's phenomenon — a fundamental limitation of polynomial interpolation.
```

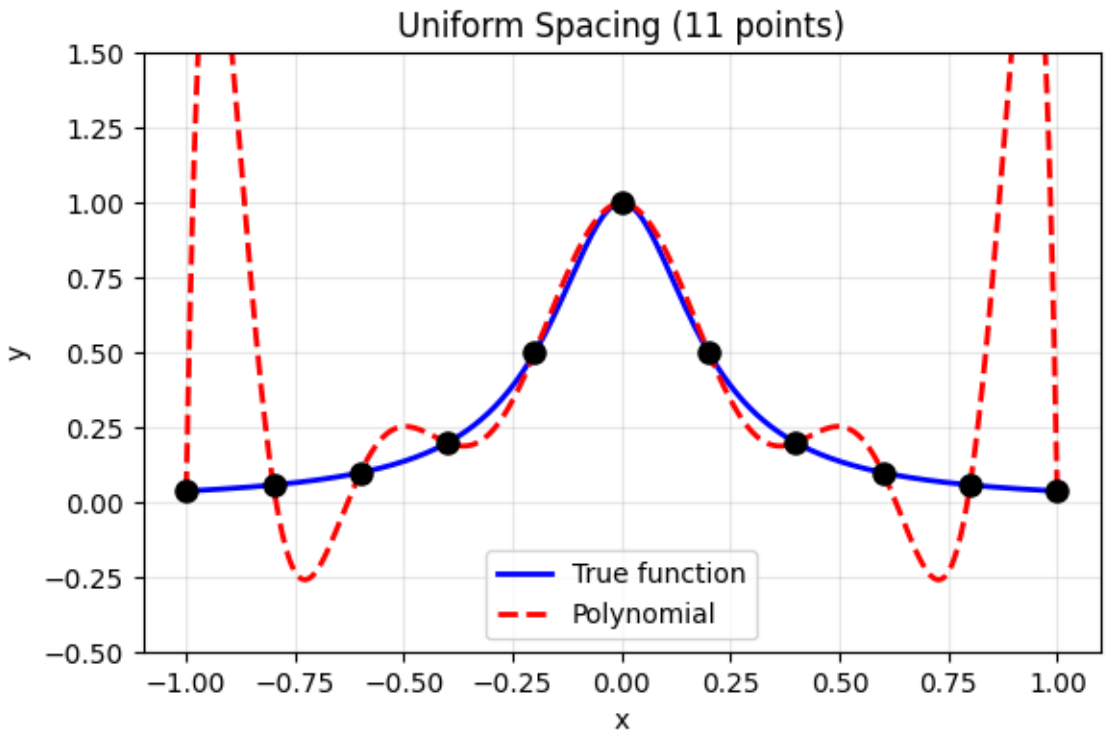
6.4.4 Solution: Chebyshev Nodes

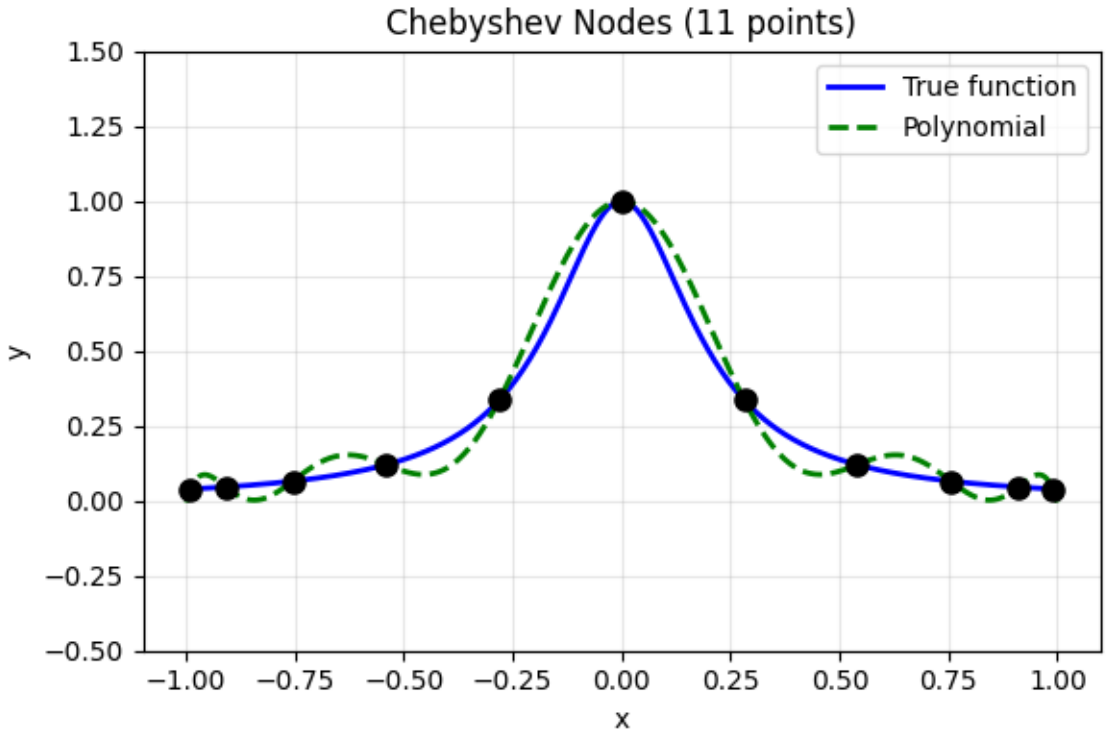
Using **non-uniform** point spacing can eliminate Runge's phenomenon:

$$x_k = \cos\left(\frac{2k+1}{2n+2}\pi\right), \quad k = 0, 1, \ldots, n$$

These cluster more points near the edges.

```python
# Compare uniform vs Chebyshev nodes
n_points = 11

# Uniform spacing
x_uniform = np.linspace(-1, 1, n_points)
y_uniform = runge_function(x_uniform)

# Chebyshev nodes
k = np.arange(n_points)
x_chebyshev = np.cos((2*k + 1) / (2*n_points) * np.pi)
x_chebyshev = np.sort(x_chebyshev)  # Sort for plotting
y_chebyshev = runge_function(x_chebyshev)

# Interpolate
y_interp_uniform = [lagrange_interp(x_uniform, y_uniform, x) for x in x_fine]
y_interp_chebyshev = [lagrange_interp(x_chebyshev, y_chebyshev, x) for x in x_fine]

plt.figure(figsize=(6, 8))

plt.subplot(2, 1, 1)
plt.plot(x_fine, y_true, 'b-', linewidth=2, label='True function')
plt.plot(x_fine, y_interp_uniform, 'r--', linewidth=2, label='Polynomial')
plt.plot(x_uniform, y_uniform, 'ko', markersize=8)
plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Uniform Spacing ({n_points} points)')
plt.legend()
plt.ylim(-0.5, 1.5)
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
plt.plot(x_fine, y_true, 'b-', linewidth=2, label='True function')
plt.plot(x_fine, y_interp_chebyshev, 'g--', linewidth=2, label='Polynomial')
plt.plot(x_chebyshev, y_chebyshev, 'ko', markersize=8)
plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Chebyshev Nodes ({n_points} points)')
plt.legend()
plt.ylim(-0.5, 1.5)
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("Chebyshev nodes cluster near the edges, reducing oscillations!")
```






```
Chebyshev nodes cluster near the edges, reducing oscillations!
```

6.5 IV. Spline Interpolation

Instead of one high-degree polynomial, use **piecewise low-degree polynomials** joined smoothly.

Linear spline is actually the simplest version of spline interpolation: $$S_i(x) = y_i + \frac{y_{i+1}-y_i}{x_{i+1}-x_i}(x - x_i)$$

But it's not smooth at the data points (the derivative is discontinuous).

6.5.1 Quadratic Spline Interpolation

To make a smoother curve, we can use quadratic polynomials between each pair of points:

$$S_i(x) = a_i + b_i x + c_i x^2$$

Example: Consider three points: $(-1, 0), (0, 1), (1, 3)$

We want to find two quadratic pieces: $$p_1(x) = a_1 + b_1 x + c_1 x^2 \quad \text{on } [-1, 0] p_2(x) = a_2 + b_2 x + c_2 x^2 \quad \text{on } [0, 1]$$

Conditions:

1. $p_1(-1) = 0 \rightarrow a_1 - b_1 + c_1 = 0$
2. $p_1(0) = 1 \rightarrow a_1 = 1$
3. $p_2(0) = 1 \rightarrow a_2 = 1$
4. $p_2(1) = 3 \rightarrow a_2 + b_2 + c_2 = 3$
5. Smoothness: $p_1'(0) = p_2'(0) \rightarrow b_1 = b_2$

That's 5 equations for 6 unknowns! We need one more condition.

Boundary condition: Set $p_1'(-1) = 0 \rightarrow b_1 - 2c_1 = 0$

Solving: $p_1(x) = 1 + 2x + x^2$ and $p_2(x) = 1 + 2x$

6.5.2 General Quadratic Spline Formula

For a series of points, the quadratic spline can be written as:

$$S_i(x) = y_i + z_i(x - x_i) + \frac{z_{i+1} - z_i}{2(x_{i+1} - x_i)}(x - x_i)^2$$

where the z values (related to derivatives) are computed recursively:

$$z_{i+1} = -z_i + 2\frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

and z_0 is the derivative at the first point (a boundary condition).

```
# Implement quadratic spline from scratch
def quad_spline(x_data, y_data, xnew, z0=0.0):
    """
    Quadratic spline interpolation - implemented from scratch.

    Parameters:
        x_data: array of x coordinates of known points
        y_data: array of y coordinates of known points
        xnew: the x value where we want to interpolate
```

(continues on next page)

(continued from previous page)

```
        z0: derivative at the first point (boundary condition)

    Returns:
        Interpolated y value at xnew
    """

    x_data = np.array(x_data)
    y_data = np.array(y_data)
    n = len(x_data)

    dx = np.diff(x_data) # x[i+1] - x[i]
    dy = np.diff(y_data) # y[i+1] - y[i]

    # calcualte z values recursively
    z = np.zeros(n)
    z[0] = z0

    for i in range(n-1):
        z[i+1] = -z[i] + 2 * dy[i] / dx[i]

    # find the interal where xnew falls
    i = 0
    while i < n-2 and xnew > x_data[i+1]:
        i+=1

    # get ynew
    ynew = y_data[i] + z[i] * (xnew - x_data[i]) + (z[i+1] - z[i]) * (xnew - x_
↪data[i])**2/(2*dx[i])

    return ynew

# Test quadratic spline
x_test = np.array([0, 1, 2, 3, 4])
y_test = np.array([1, 2, 0, 2, 3])

print(f"Quadratic spline at x=2.5: {quad_spline(x_test, y_test, 2.5, z0=2.0)}")
```

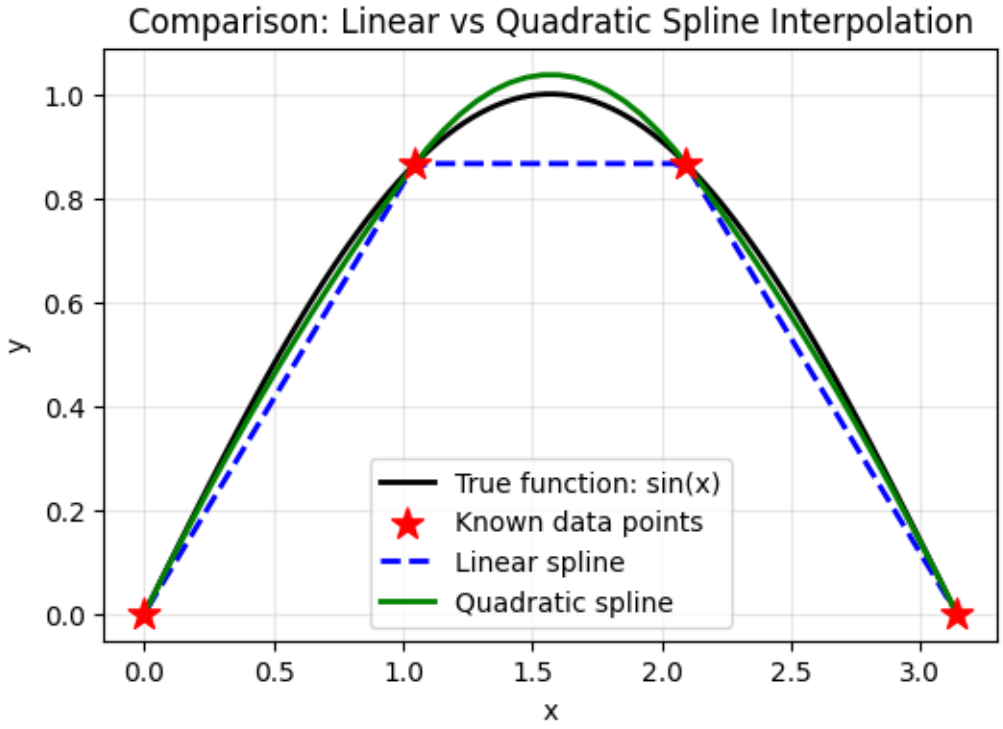
```
Quadratic spline at x=2.5: -0.5
```

```
# Compare linear vs quadratic spline on sin(x)
f = lambda x: np.sin(x)
x_min, x_max = 0, np.pi
npt_known = 4

x_ideal = np.linspace(x_min, x_max, 100)
x_known = np.linspace(x_min, x_max, npt_known)
y_known = f(x_known)

# Interpolate using our hand-implemented methods
x_interp = np.linspace(x_min + 0.01, x_max - 0.01, 50)
y_linear = np.array([linear_interp(x_known, y_known, xi) for xi in x_interp])
y_quad = np.array([quad_spline(x_known, y_known, xi, z0=1.0) for xi in x_interp])

plt.figure(figsize=(6, 4))
plt.plot(x_ideal, f(x_ideal), 'k-', linewidth=2, label='True function: sin(x)')
plt.scatter(x_known, y_known, color='r', s=150, marker='*', zorder=5, label='Known␣
```

(continues on next page)

(continued from previous page)

```
↪data points')
plt.plot(x_interp, y_linear, 'b--', linewidth=2, label='Linear spline')
plt.plot(x_interp, y_quad, 'g-', linewidth=2, label='Quadratic spline')

plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Comparison: Linear vs Quadratic Spline Interpolation')
plt.grid(True, alpha=0.3)
plt.show()

print("Quadratic spline is smoother and follows the curve better!")
```




```
Quadratic spline is smoother and follows the curve better!
```

6.5.3 Cubic Splines: The Most Popular Choice

A **cubic spline** uses cubic polynomials between each pair of adjacent points:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

Why cubic? Consider what we need for a "smooth" curve:

Condition	What it means	Spline degree needed
$S_i(x_{i+1}) = S_{i+1}(x_{i+1})$	Continuous curve	Linear (degree 1)
$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$	Continuous slope	Quadratic (degree 2)
$S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$	Continuous curvature	**Cubic (degree 3)**

Cubic is the **minimum degree** that gives continuous curvature — this makes the curve look natural and physically realistic.

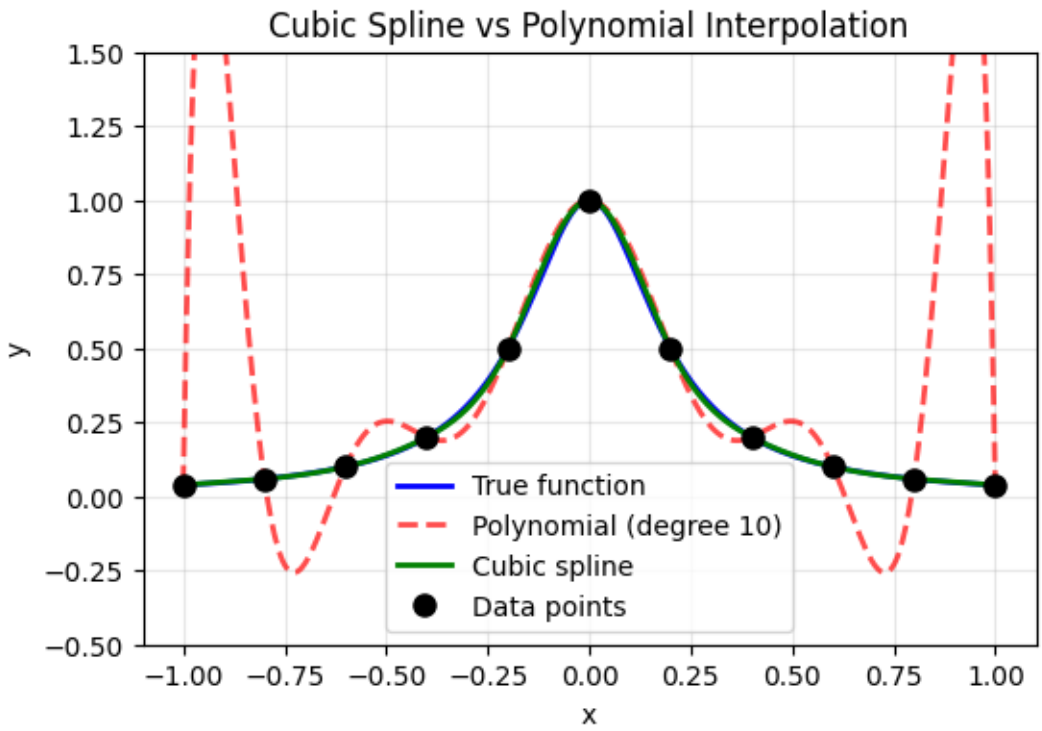
Higher-order splines (quartic, quintic) give even smoother curves, but with diminishing returns and increased computational cost.

6.5.4 Using SciPy for Cubic Splines

Implementing cubic splines from scratch requires solving a tridiagonal system of equations — doable but tedious. This is where libraries like SciPy shine!

> **Philosophy:** Understand the principle first, then use well-tested libraries.
>
> It's risky to use a function without understanding what it does!

```
from scipy.interpolate import CubicSpline

# Compare polynomial vs cubic spline on Runge's function
n_points = 11
x_data_runge = np.linspace(-1, 1, n_points)
y_data_runge = runge_function(x_data_runge)

# Cubic spline
cs = CubicSpline(x_data_runge, y_data_runge)
y_spline = cs(x_fine)

# Polynomial
y_poly = [lagrange_interp(x_data_runge, y_data_runge, x) for x in x_fine]

plt.figure(figsize=(6, 4))
plt.plot(x_fine, y_true, 'b-', linewidth=2, label='True function')
plt.plot(x_fine, y_poly, 'r--', linewidth=2, alpha=0.7, label='Polynomial (degree 10)
 ↪')
plt.plot(x_fine, y_spline, 'g-', linewidth=2, label='Cubic spline')
plt.plot(x_data_runge, y_data_runge, 'ko', markersize=8, label='Data points')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Cubic Spline vs Polynomial Interpolation')
plt.legend()
plt.ylim(-0.5, 1.5)
plt.grid(True, alpha=0.3)
plt.show()

print("The cubic spline is much better behaved than the polynomial!")
```




```
The cubic spline is much better behaved than the polynomial!
```

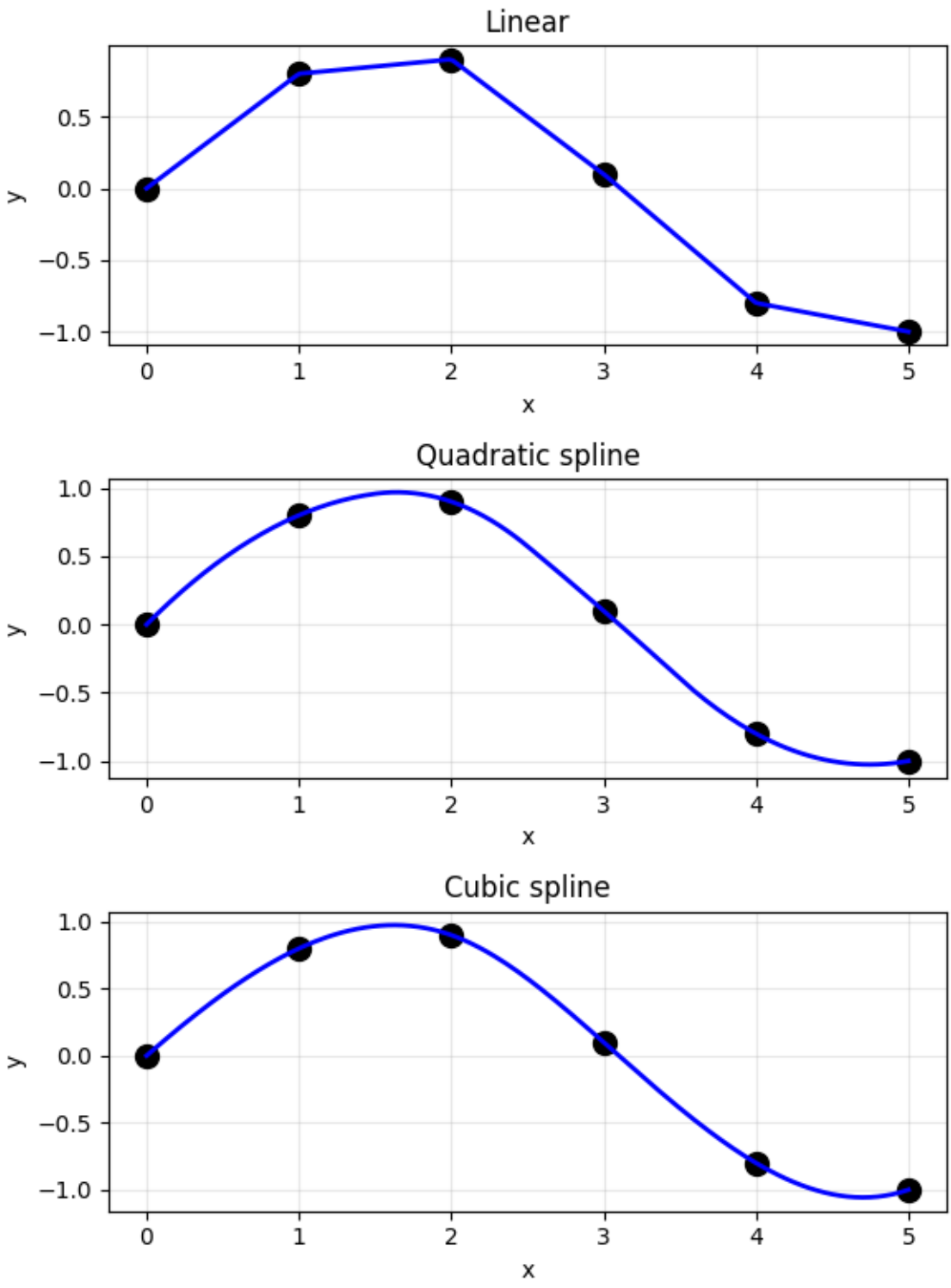
6.5.5 Types of Splines

```
from scipy.interpolate import interp1d

# Compare different interpolation methods
x_data_demo = np.array([0, 1, 2, 3, 4, 5])
y_data_demo = np.array([0, 0.8, 0.9, 0.1, -0.8, -1.0])
x_fine_demo = np.linspace(0, 5, 200)

# Different interpolation methods
methods = [
    ('linear', 'Linear'),
    ('quadratic', 'Quadratic spline'),
    ('cubic', 'Cubic spline')
]

plt.figure(figsize=(6, 8))

for idx, (method, name) in enumerate(methods):
    plt.subplot(3, 1, idx + 1)

    f = interp1d(x_data_demo, y_data_demo, kind=method)
    y_interp = f(x_fine_demo)

    plt.plot(x_data_demo, y_data_demo, 'ko', markersize=10)
    plt.plot(x_fine_demo, y_interp, 'b-', linewidth=2)
    plt.xlabel('x')
```

(continues on next page)

(continued from previous page)

```
    plt.ylabel('y')
    plt.title(name)
    plt.grid(True, alpha=0.3)

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

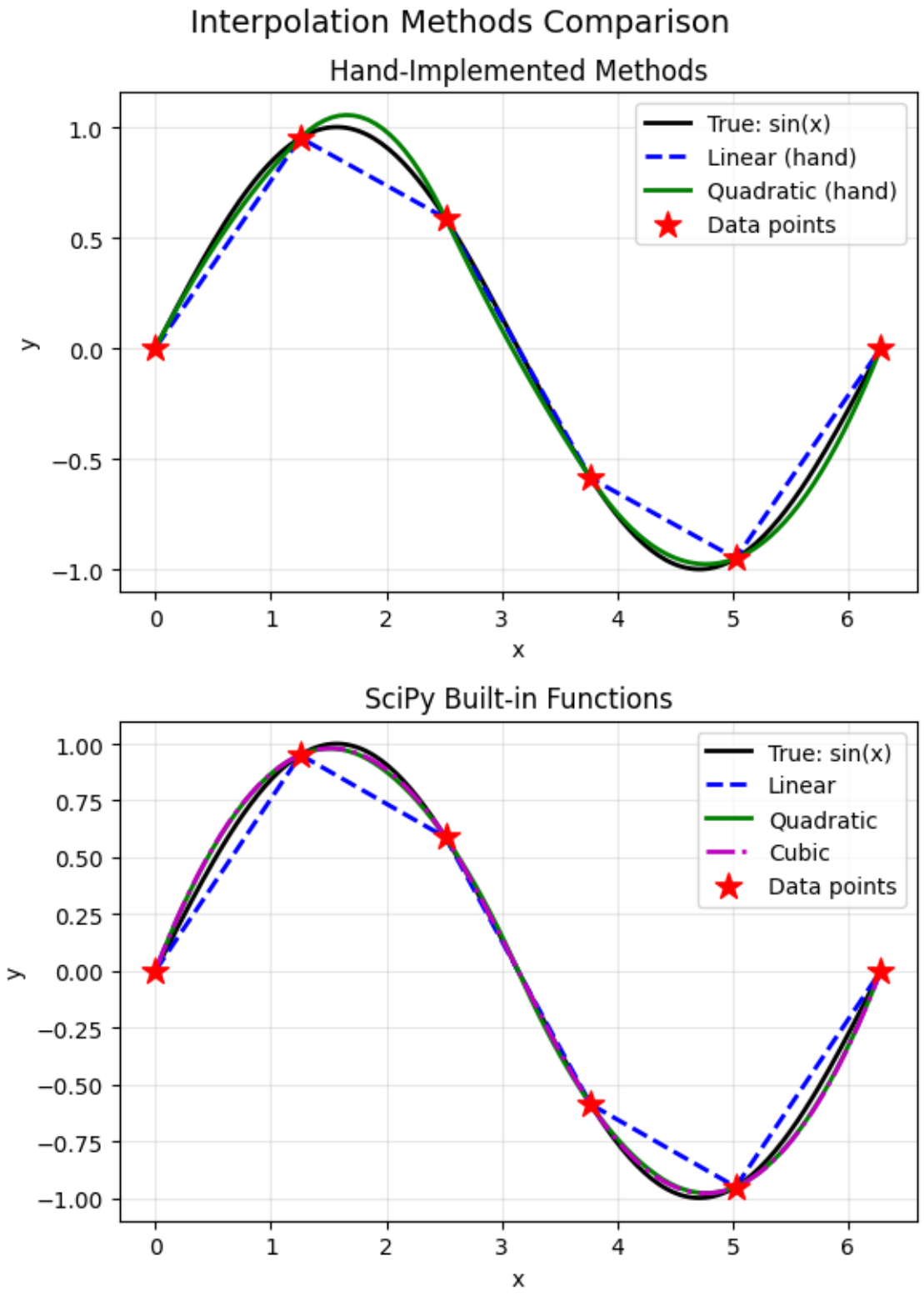



```
# Comprehensive comparison: Hand-implemented vs SciPy
from scipy.interpolate import interp1d

f = lambda x: np.sin(x)
x_min, x_max = 0, 2*pi
```

(continues on next page)

(continued from previous page)

```
npt_known = 6

x_ideal = np.linspace(x_min, x_max, 200)
x_known = np.linspace(x_min, x_max, npt_known)
y_known = f(x_known)

# SciPy interpolation functions
f_linear_scipy = interp1d(x_known, y_known, kind='linear')
f_quad_scipy = interp1d(x_known, y_known, kind='quadratic')
f_cubic_scipy = interp1d(x_known, y_known, kind='cubic')

plt.figure(figsize=(6, 8))

plt.subplot(2, 1, 1)
plt.title('Hand-Implemented Methods')
plt.plot(x_ideal, f(x_ideal), 'k-', linewidth=2, label='True: sin(x)')

# Our implementations
x_interp = np.linspace(x_min + 0.01, x_max - 0.01, 100)
y_linear_hand = np.array([linear_interp(x_known, y_known, xi) for xi in x_interp])
y_quad_hand = np.array([quad_spline(x_known, y_known, xi, z0=1.0) for xi in x_interp])

plt.plot(x_interp, y_linear_hand, 'b--', linewidth=2, label='Linear (hand)')
plt.plot(x_interp, y_quad_hand, 'g-', linewidth=2, label='Quadratic (hand)')
plt.scatter(x_known, y_known, c='r', s=150, marker='*', zorder=5, label='Data points')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
plt.title('SciPy Built-in Functions')
plt.plot(x_ideal, f(x_ideal), 'k-', linewidth=2, label='True: sin(x)')
plt.plot(x_ideal, f_linear_scipy(x_ideal), 'b--', linewidth=2, label='Linear')
plt.plot(x_ideal, f_quad_scipy(x_ideal), 'g-', linewidth=2, label='Quadratic')
plt.plot(x_ideal, f_cubic_scipy(x_ideal), 'm-.', linewidth=2, label='Cubic')
plt.scatter(x_known, y_known, c='r', s=150, marker='*', zorder=5, label='Data points')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.suptitle('Interpolation Methods Comparison', y=1.02, fontsize=14)
plt.show()

print("Key insight: Understanding the hand-implemented versions helps you choose")
print("the right scipy function for your application!")
```



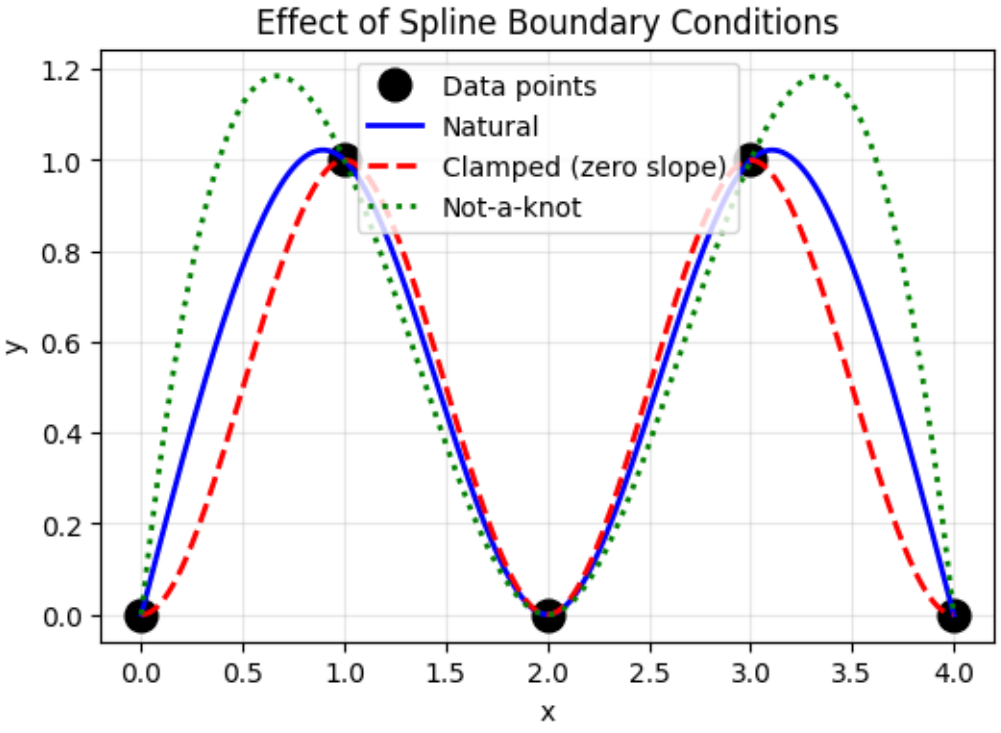

```
Key insight: Understanding the hand-implemented versions helps you choose
the right scipy function for your application!
```

6.5.6 Spline Boundary Conditions

The cubic spline has degrees of freedom at the boundaries. Common choices:

- **Natural spline**: $S''(x_0) = S''(x_n) = 0$ (second derivative is zero at ends)
- **Clamped spline**: Specify $S'(x_0)$ and $S'(x_n)$
- **Not-a-knot**: Use same cubic polynomial for first two and last two intervals

```
# Compare different boundary conditions
x_bc = np.array([0, 1, 2, 3, 4])
y_bc = np.array([0, 1, 0, 1, 0])
x_fine_bc = np.linspace(0, 4, 200)

# Different boundary conditions
cs_natural = CubicSpline(x_bc, y_bc, bc_type='natural')
cs_clamped = CubicSpline(x_bc, y_bc, bc_type=((1, 0), (1, 0)))  # zero slope at ends
cs_notaknot = CubicSpline(x_bc, y_bc, bc_type='not-a-knot')

plt.figure(figsize=(6, 4))
plt.plot(x_bc, y_bc, 'ko', markersize=12, label='Data points')
plt.plot(x_fine_bc, cs_natural(x_fine_bc), 'b-', linewidth=2, label='Natural')
plt.plot(x_fine_bc, cs_clamped(x_fine_bc), 'r--', linewidth=2, label='Clamped (zero␣
↪slope)')
plt.plot(x_fine_bc, cs_notaknot(x_fine_bc), 'g:', linewidth=2, label='Not-a-knot')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Effect of Spline Boundary Conditions')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```



6.6 V. Using SciPy for Interpolation

SciPy provides robust, well-tested interpolation tools.

```
# Summary of SciPy interpolation functions
print("Key SciPy Interpolation Functions:")
print("=" * 60)
print()
print("1D Interpolation:")
print("  np.interp()          - Simple linear interpolation")
print("  interp1d()           - Flexible 1D interpolation")
print("  CubicSpline()        - Cubic spline with boundary control")
print("  UnivariateSpline()   - Spline with smoothing option")
print()
print("2D Interpolation:")
print("  interp2d()           - 2D interpolation on grid")
print("  RectBivariateSpline()- 2D spline on rectangular grid")
print("  griddata()           - Interpolate scattered data to grid")
```

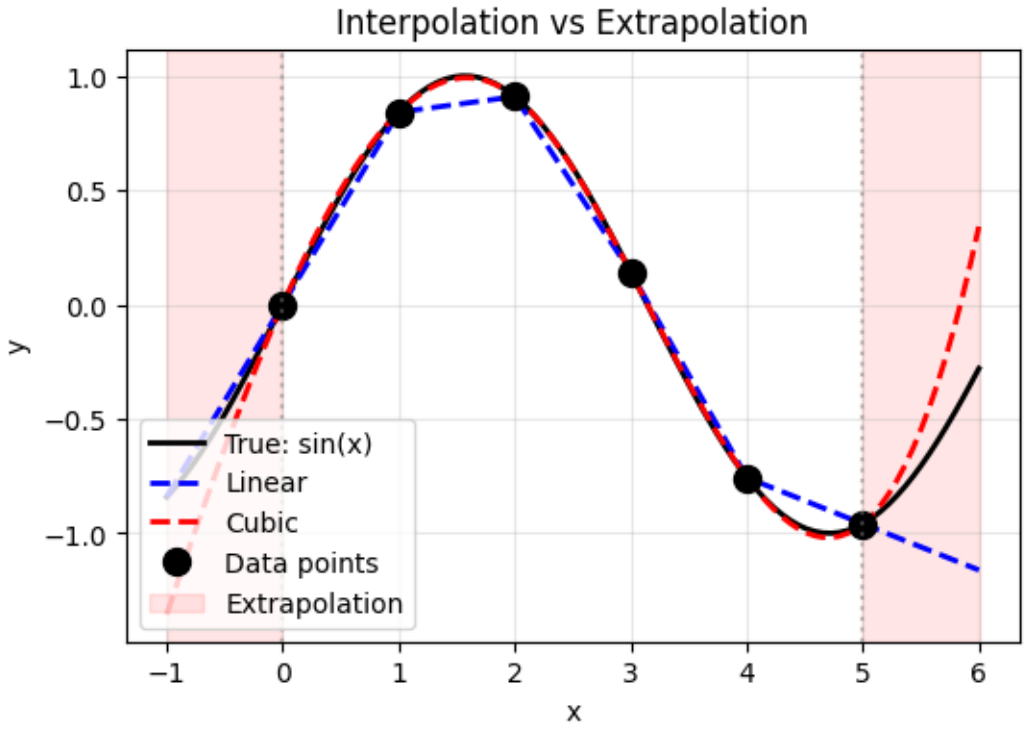
```
Key SciPy Interpolation Functions:
============================================================

1D Interpolation:
  np.interp()          - Simple linear interpolation
  interp1d()           - Flexible 1D interpolation
  CubicSpline()        - Cubic spline with boundary control
  UnivariateSpline()   - Spline with smoothing option

2D Interpolation:
  interp2d()           - 2D interpolation on grid
  RectBivariateSpline()- 2D spline on rectangular grid
  griddata()           - Interpolate scattered data to grid
```

```
# Practical example: interp1d with extrapolation
from scipy.interpolate import interp1d

x = np.array([0, 1, 2, 3, 4, 5])
y = np.sin(x)

# Create interpolation functions
f_linear = interp1d(x, y, kind='linear', fill_value='extrapolate')
f_cubic = interp1d(x, y, kind='cubic', fill_value='extrapolate')

# Evaluate (including extrapolation)
x_eval = np.linspace(-1, 6, 100)
y_true = np.sin(x_eval)

plt.figure(figsize=(6, 4))
plt.plot(x_eval, y_true, 'k-', linewidth=2, label='True: sin(x)')
plt.plot(x_eval, f_linear(x_eval), 'b--', linewidth=2, label='Linear')
plt.plot(x_eval, f_cubic(x_eval), 'r--', linewidth=2, label='Cubic')
plt.plot(x, y, 'ko', markersize=10, label='Data points')

# Mark interpolation vs extrapolation regions
plt.axvline(x=0, color='gray', linestyle=':', alpha=0.5)
plt.axvline(x=5, color='gray', linestyle=':', alpha=0.5)
```

(continues on next page)

(continued from previous page)

```
plt.axvspan(-1, 0, alpha=0.1, color='red', label='Extrapolation')
plt.axvspan(5, 6, alpha=0.1, color='red')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Interpolation vs Extrapolation')
plt.legend(loc='lower left')
plt.grid(True, alpha=0.3)
plt.show()

print("Warning: Extrapolation (outside data range) can be unreliable!")
```




```
Warning: Extrapolation (outside data range) can be unreliable!
```

6.7 VI. 2D Interpolation

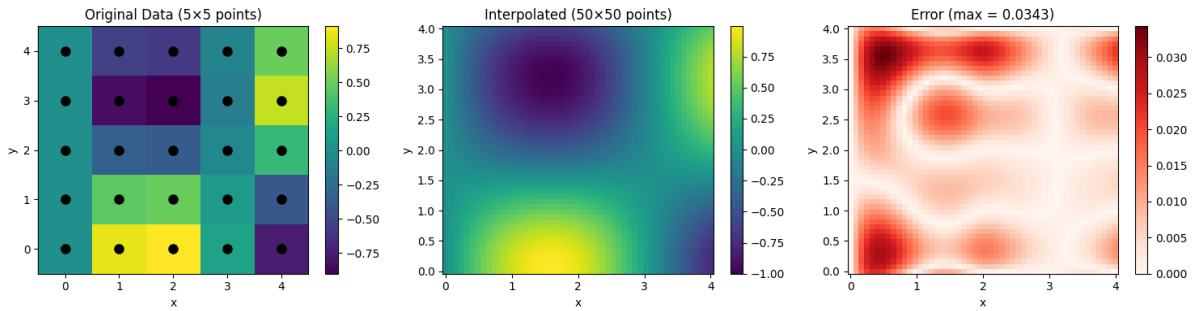
Extend interpolation to surfaces: given (x_i, y_j, z_{ij}), find $z(x, y)$.

```
from scipy.interpolate import RectBivariateSpline

# Create sample 2D data
x_2d = np.linspace(0, 4, 5)
y_2d = np.linspace(0, 4, 5)
X, Y = np.meshgrid(x_2d, y_2d)

# True function: z = sin(x) * cos(y)
Z_data = np.sin(X) * np.cos(Y)
```

(continues on next page)

(continued from previous page)

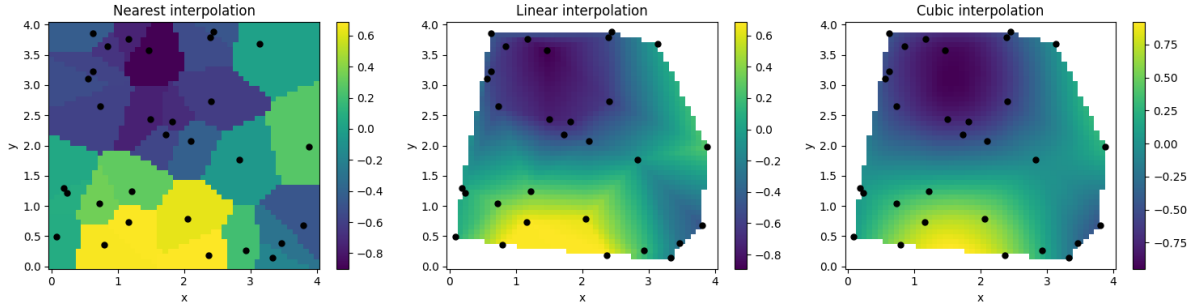
```
# Create spline interpolator
spline_2d = RectBivariateSpline(x_2d, y_2d, Z_data.T)  # Note: transpose needed

# Evaluate on fine grid
x_fine_2d = np.linspace(0, 4, 50)
y_fine_2d = np.linspace(0, 4, 50)
Z_interp = spline_2d(x_fine_2d, y_fine_2d).T

# True values on fine grid
X_fine, Y_fine = np.meshgrid(x_fine_2d, y_fine_2d)
Z_true = np.sin(X_fine) * np.cos(Y_fine)

# Plot
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Data points
ax1 = axes[0]
c1 = ax1.pcolormesh(X, Y, Z_data, shading='auto', cmap='viridis')
ax1.plot(X.flatten(), Y.flatten(), 'ko', markersize=8)
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('Original Data (5×5 points)')
plt.colorbar(c1, ax=ax1)

# Interpolated
ax2 = axes[1]
c2 = ax2.pcolormesh(X_fine, Y_fine, Z_interp, shading='auto', cmap='viridis')
ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.set_title('Interpolated (50×50 points)')
plt.colorbar(c2, ax=ax2)

# Error
ax3 = axes[2]
error = np.abs(Z_interp - Z_true)
c3 = ax3.pcolormesh(X_fine, Y_fine, error, shading='auto', cmap='Reds')
ax3.set_xlabel('x')
ax3.set_ylabel('y')
ax3.set_title(f'Error (max = {error.max():.4f})')
plt.colorbar(c3, ax=ax3)

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



6.7.1 Interpolating Scattered Data

For data not on a regular grid, use `griddata()`.

```
from scipy.interpolate import griddata

# Generate scattered data points
np.random.seed(42)
n_scatter = 30
x_scatter = np.random.uniform(0, 4, n_scatter)
y_scatter = np.random.uniform(0, 4, n_scatter)
z_scatter = np.sin(x_scatter) * np.cos(y_scatter)

# Create regular grid for interpolation
x_grid = np.linspace(0, 4, 50)
y_grid = np.linspace(0, 4, 50)
X_grid, Y_grid = np.meshgrid(x_grid, y_grid)

# Interpolate using different methods
methods = ['nearest', 'linear', 'cubic']

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

for ax, method in zip(axes, methods):
    Z_grid = griddata((x_scatter, y_scatter), z_scatter,
                      (X_grid, Y_grid), method=method)

    c = ax.pcolormesh(X_grid, Y_grid, Z_grid, shading='auto', cmap='viridis')
    ax.plot(x_scatter, y_scatter, 'ko', markersize=5)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title(f'{method.capitalize()} interpolation')
    plt.colorbar(c, ax=ax)

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

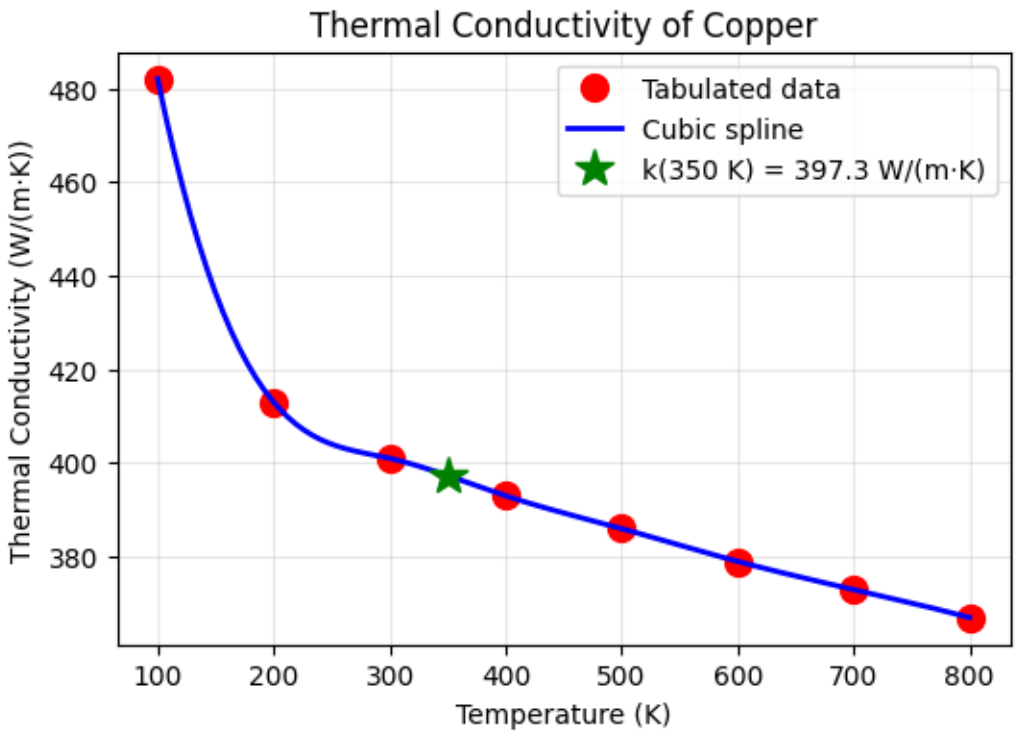


6.8 VII. Physics Applications

6.8.1 1. Material Properties from Tables

Many material properties are tabulated. Interpolation lets us use values at any temperature or pressure.

```
# Thermal conductivity of copper vs temperature (simplified data)
T_data = np.array([100, 200, 300, 400, 500, 600, 700, 800])  # K
k_data = np.array([482, 413, 401, 393, 386, 379, 373, 367])  # W/(m·K)

# Create interpolation function
k_interp = CubicSpline(T_data, k_data)

# Evaluate at desired temperatures
T_fine = np.linspace(100, 800, 200)
k_fine = k_interp(T_fine)

# Specific query
T_query = 350  # K
k_query = k_interp(T_query)

plt.figure(figsize=(6, 4))
plt.plot(T_data, k_data, 'ro', markersize=10, label='Tabulated data')
plt.plot(T_fine, k_fine, 'b-', linewidth=2, label='Cubic spline')
plt.plot(T_query, k_query, 'g*', markersize=15,
         label=f'k({T_query} K) = {k_query:.1f} W/(m·K)')

plt.xlabel('Temperature (K)')
plt.ylabel('Thermal Conductivity (W/(m·K))')
plt.title('Thermal Conductivity of Copper')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

print(f"Thermal conductivity at T = {T_query} K: {k_query:.2f} W/(m·K)")
```




```
Thermal conductivity at T = 350 K: 397.32 W/(m·K)
```

6.8.2 2. Temperature Field Interpolation

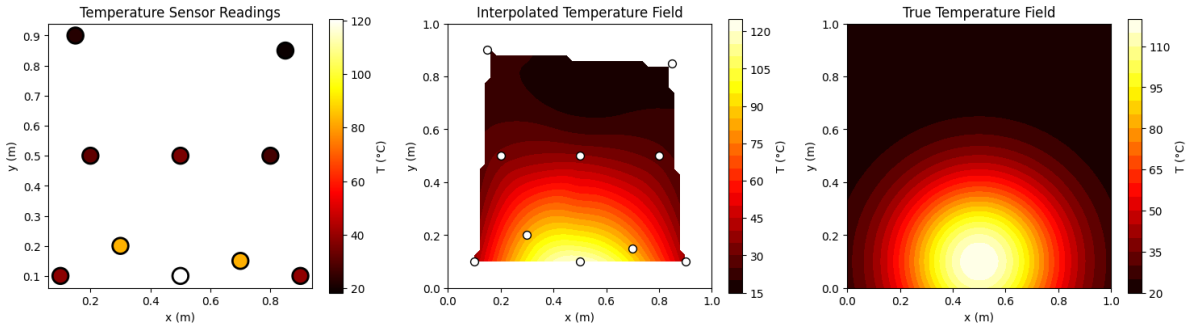
Create a smooth temperature field from sensor measurements.

```
# Temperature sensors at various locations
np.random.seed(123)
sensor_x = np.array([0.1, 0.3, 0.5, 0.7, 0.9, 0.2, 0.5, 0.8, 0.15, 0.85])
sensor_y = np.array([0.1, 0.2, 0.1, 0.15, 0.1, 0.5, 0.5, 0.5, 0.9, 0.85])

# Temperature readings (simulating a heat source at center-bottom)
def true_temp(x, y):
    return 100 * np.exp(-((x-0.5)**2 + (y-0.1)**2) / 0.1) + 20

sensor_T = true_temp(sensor_x, sensor_y) + np.random.randn(len(sensor_x)) * 2

# Create grid for interpolation
grid_x = np.linspace(0, 1, 50)
grid_y = np.linspace(0, 1, 50)
Grid_X, Grid_Y = np.meshgrid(grid_x, grid_y)

# Interpolate temperature field
Grid_T = griddata((sensor_x, sensor_y), sensor_T, (Grid_X, Grid_Y), method='cubic')

# True temperature field
True_T = true_temp(Grid_X, Grid_Y)

fig, axes = plt.subplots(1, 3, figsize=(15, 4))
```

(continues on next page)

(continued from previous page)

```
# Sensor locations
ax1 = axes[0]
scatter = ax1.scatter(sensor_x, sensor_y, c=sensor_T, s=200, cmap='hot',
                      edgecolors='black', linewidths=2)
ax1.set_xlabel('x (m)')
ax1.set_ylabel('y (m)')
ax1.set_title('Temperature Sensor Readings')
ax1.set_aspect('equal')
plt.colorbar(scatter, ax=ax1, label='T (°C)')

# Interpolated field
ax2 = axes[1]
c2 = ax2.contourf(Grid_X, Grid_Y, Grid_T, levels=20, cmap='hot')
ax2.scatter(sensor_x, sensor_y, c='white', s=50, edgecolors='black')
ax2.set_xlabel('x (m)')
ax2.set_ylabel('y (m)')
ax2.set_title('Interpolated Temperature Field')
ax2.set_aspect('equal')
plt.colorbar(c2, ax=ax2, label='T (°C)')

# True field
ax3 = axes[2]
c3 = ax3.contourf(Grid_X, Grid_Y, True_T, levels=20, cmap='hot')
ax3.set_xlabel('x (m)')
ax3.set_ylabel('y (m)')
ax3.set_title('True Temperature Field')
ax3.set_aspect('equal')
plt.colorbar(c3, ax=ax3, label='T (°C)')

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



6.9 VIII. Summary

6.9.1 Methods Comparison

Method	Pros	Cons	Best For
Linear	Simple, stable, fast	Not smooth	Quick estimates, large datasets
Polynomial	Exact through all points	Runge's phenomenon	Few points, Chebyshev nodes
Cubic Spline	Smooth, stable	Requires solving system	Most applications
2D griddata	Handles scattered data	Can be slow	Irregular sensor layouts

CHAPTER

SEVEN

CURVE FITTING AND REGRESSION

In this lecture, we'll learn how to find the **best mathematical function** to describe noisy experimental data.

7.1 Fitting vs. Interpolation

Interpolation	Curve Fitting
Passes through ALL points	Finds the best overall trend
Assumes data is exact	Accounts for noise and errors
Can oscillate wildly	Smoother, more robust
Good for tabulated data	Good for experimental data

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

# For nicer plots
plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9
```

7.2 I. The Fitting Problem

Given noisy data points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, we want to find a function $f(x)$ that **best describes** the underlying trend.

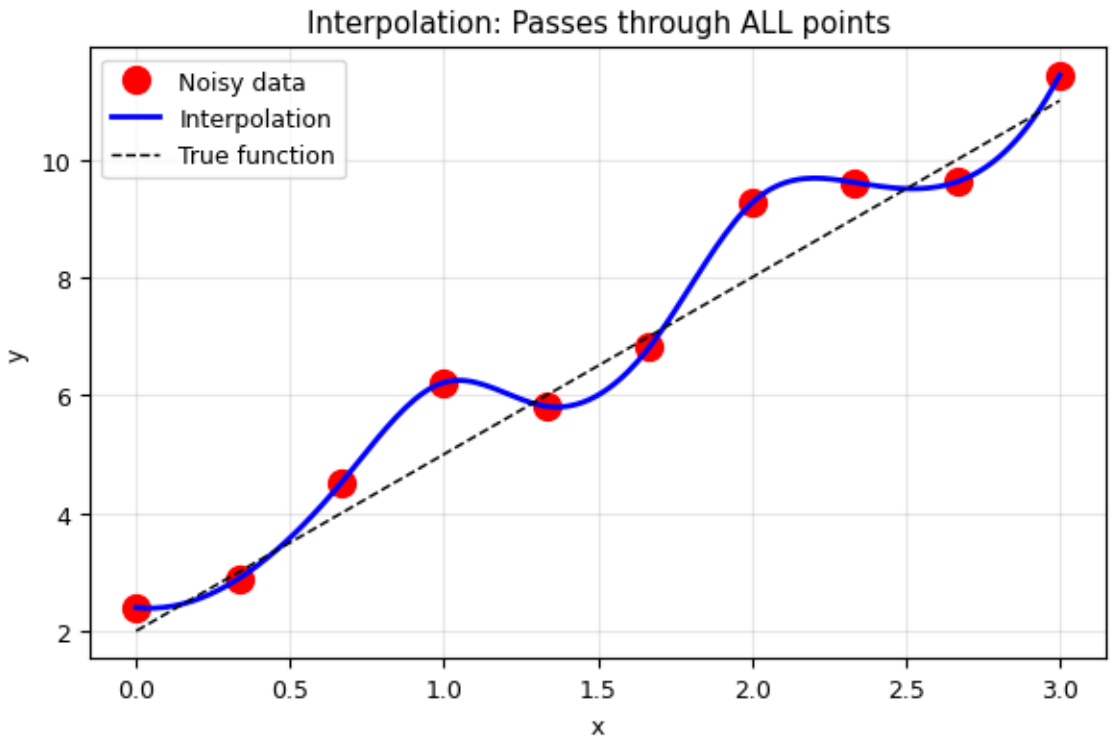
Unlike interpolation, the function **does NOT need to pass through** the data points. Instead, we minimize the overall error between the model and the data.

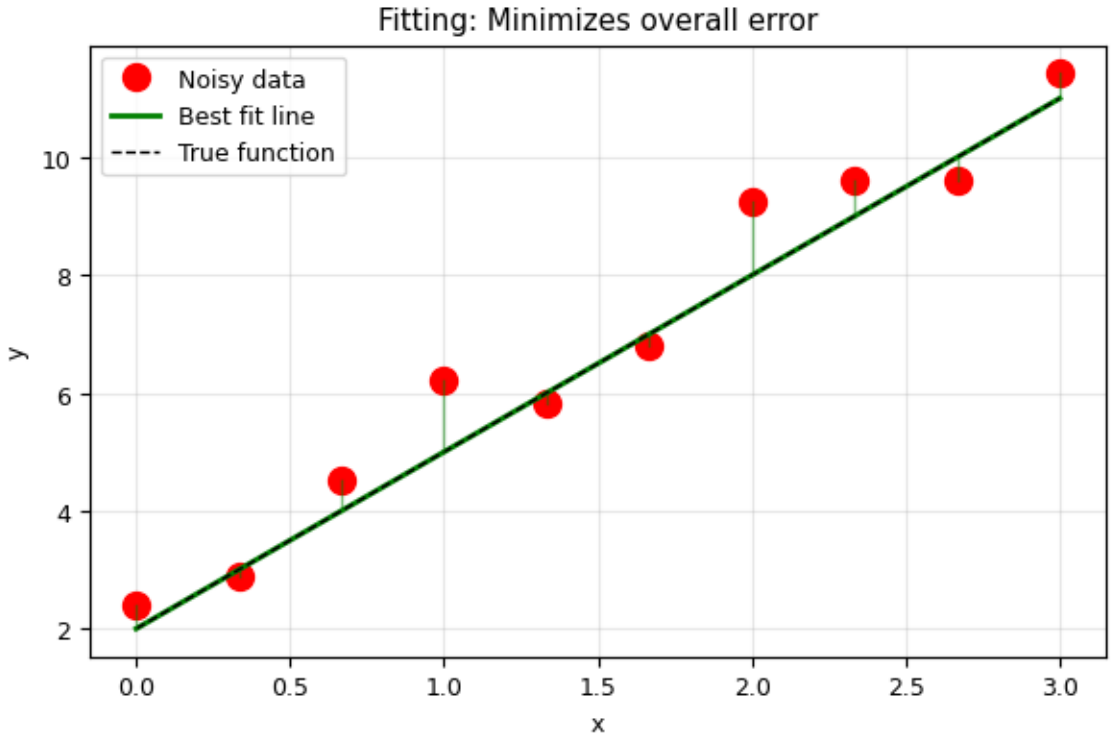
```
# Visualize the difference between interpolation and fitting
from scipy.interpolate import interp1d

# True underlying function
f_true = lambda x: 3*x + 2

# Generate noisy data
np.random.seed(42)
x_data = np.linspace(0, 3, 10)
y_data = f_true(x_data) + np.random.randn(10) * 0.8

# Interpolation (passes through all points)
```

(continues on next page)

(continued from previous page)

```
f_interp = interp1d(x_data, y_data, kind='cubic')
x_fine = np.linspace(0, 3, 200)

plt.figure(figsize=(6, 8))

plt.subplot(2, 1, 1)
plt.plot(x_data, y_data, 'ro', markersize=10, label='Noisy data')
plt.plot(x_fine, f_interp(x_fine), 'b-', linewidth=2, label='Interpolation')
plt.plot(x_fine, f_true(x_fine), 'k--', linewidth=1, label='True function')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Interpolation: Passes through ALL points')
plt.legend()
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
plt.plot(x_data, y_data, 'ro', markersize=10, label='Noisy data')
plt.plot(x_fine, f_true(x_fine), 'g-', linewidth=2, label='Best fit line')
plt.plot(x_fine, f_true(x_fine), 'k--', linewidth=1, label='True function')
# Show residuals
for i in range(len(x_data)):
    plt.plot([x_data[i], x_data[i]], [y_data[i], f_true(x_data[i])],
             'g-', alpha=0.5, linewidth=1)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Fitting: Minimizes overall error')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print('Interpolation follows every bump in the noise.')
print('Fitting captures the TRUE underlying trend.')
```






```
Interpolation follows every bump in the noise.
Fitting captures the TRUE underlying trend.
```

7.3 II. The Least Squares Principle

7.3.1 What is the "best" fit?

We need a criterion. The most common is **least squares**: minimize the sum of squared residuals.

Given data (x_i, y_i) and a model $f(x)$, the **error** (or cost function) is:

$$\text{err} = \sum_{i=1}^{n} \left(y_i - f(x_i)\right)^2$$

The best fit is the one that **minimizes** this error.

7.3.2 Why squared differences?

- Positive and negative errors don't cancel out
- Larger errors are penalized more heavily
- The math works out nicely (differentiable, has unique minimum)
- Related to maximum likelihood estimation for Gaussian noise

7.4 III. Linear Fitting (from Scratch)

7.4.1 The Simplest Case: $f(x) = ax + b$

We want to find the slope a and intercept b that minimize:

$$\text{err}(a, b) = \sum_{i=1}^{n} \left(y_i - (ax_i + b)\right)^2$$

7.4.2 Deriving the Solution

At the minimum, the partial derivatives must be zero:

$$\frac{\partial \text{err}}{\partial a} = -2 \sum_{i=1}^{n} x_i \left(y_i - ax_i - b\right) = 0$$

$$\frac{\partial \text{err}}{\partial b} = -2 \sum_{i=1}^{n} \left(y_i - ax_i - b\right) = 0$$

These give us a system of two linear equations:

$$a \sum x_i^2 + b \sum x_i = \sum x_i y_i$$

$$a \sum x_i + b \cdot n = \sum y_i$$

7.4.3 Matrix Form

This can be written as $\mathbf{A}\vec{p} = \vec{c}$:

$$\begin{bmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} \begin{bmatrix} b \\ a \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \end{bmatrix}$$

This is just solving $\mathbf{A}\vec{p} = \vec{c}$!

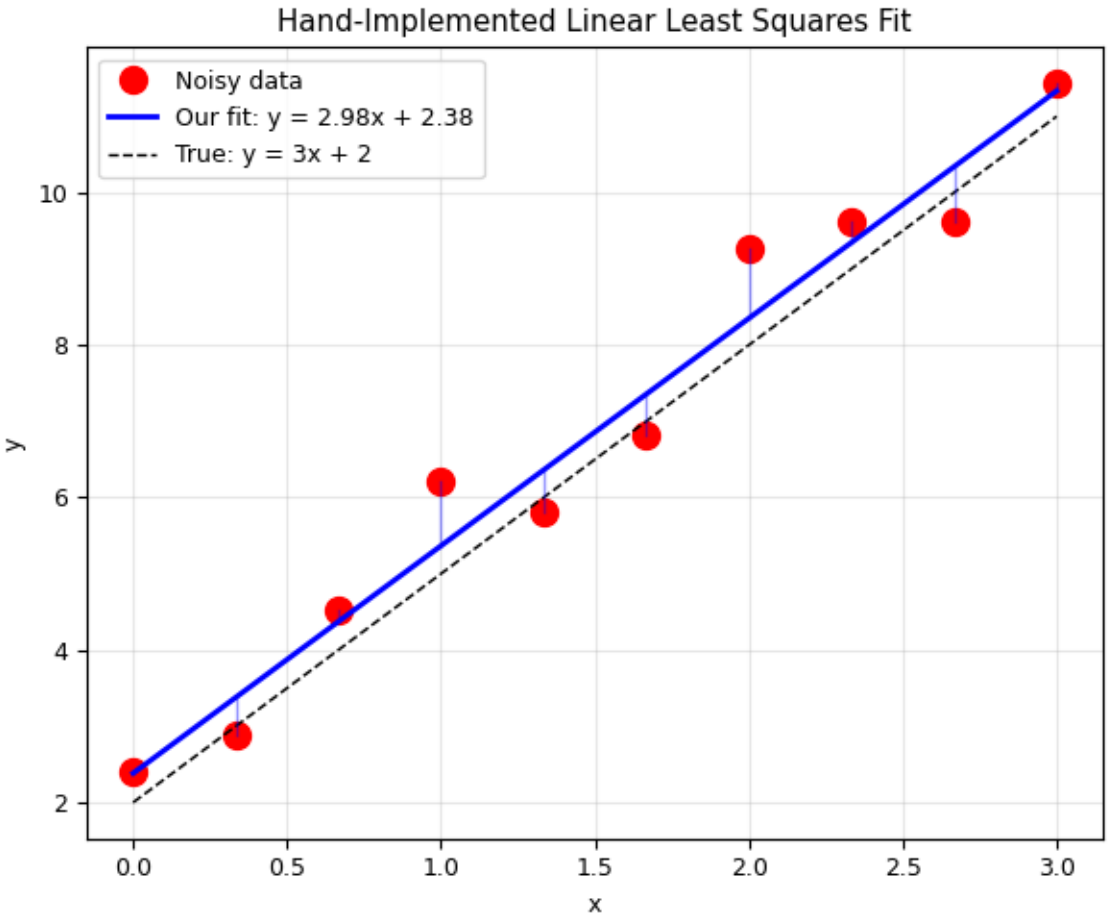
```
# Implement linear fitting from scratch
def linear_fit(x, y):
    """
    Linear least squares fit: y = a*x + b
    Implemented from scratch using the normal equations.

    Parameters:
        x, y: arrays of data points

    Returns:
        a (slope), b (intercept)
    """
    n = len(x)

    # Compute the sums we need
    sum_x = np.sum(x)
    sum_y = np.sum(y)
    sum_x2 = np.sum(x**2)
    sum_xy = np.sum(x*y)

    # Build the matrix equation: A @ [b, a] = c
    A = np.array([ [n, sum_x],
                   [sum_x, sum_x2]
    ])

    c = np.array([sum_y, sum_xy])

    # Solve A @ p = c
    p = np.linalg.solve(A, c)

    b, a = p  # p = [b, a]
    return a, b

# Test with our noisy data
a_fit, b_fit = linear_fit(x_data, y_data)

print(f'True parameters:   a = 3.000, b = 2.000')
print(f'Fitted parameters: a = {a_fit:.3f}, b = {b_fit:.3f}')

# Plot
plt.figure(figsize=(6, 5))
plt.plot(x_data, y_data, 'ro', markersize=10, label='Noisy data')
plt.plot(x_fine, a_fit * x_fine + b_fit, 'b-', linewidth=2,
         label=f'Our fit: y = {a_fit:.2f}x + {b_fit:.2f}')
plt.plot(x_fine, f_true(x_fine), 'k--', linewidth=1, label='True: y = 3x + 2')

# Show residuals
for i in range(len(x_data)):
    y_pred = a_fit * x_data[i] + b_fit
```

(continues on next page)

(continued from previous page)

```
    plt.plot([x_data[i], x_data[i]], [y_data[i], y_pred],
             'b-', alpha=0.4, linewidth=1)

plt.xlabel('x')
plt.ylabel('y')
plt.title('Hand-Implemented Linear Least Squares Fit')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
True parameters:   a = 3.000, b = 2.000
Fitted parameters: a = 2.983, b = 2.383
```



7.4.4 Alternative: Using `scipy.optimize.minimize`

We can also find the fit by directly minimizing the error function numerically:

```
# Method 2: Using scipy.optimize.minimize
from scipy.optimize import minimize

# Define the model function
def fit_func(x, a, b):
    return a * x + b

# Define the error (cost) function
def err_func(params, x, y):
    """Sum of squared residuals."""
    a, b = params
    y_pred = fit_func(x, a, b)
    err = np.sum((y - y_pred)**2)
    return err

# Minimize!
param_init = [1.0, 1.0]  # Initial guess
result = minimize(err_func, param_init, args=(x_data, y_data))

a_opt, b_opt = result.x
print(f'Our analytic solution:  a = {a_fit:.4f}, b = {b_fit:.4f}')
print(f'scipy.optimize result:  a = {a_opt:.4f}, b = {b_opt:.4f}')
print(f'\nThey match! Both methods give the same answer.')
print(f'\nMinimum error: {result.fun:.4f}')
print(f'Optimization converged: {result.success}')
```

```
Our analytic solution:  a = 2.9834, b = 2.3833
scipy.optimize result:  a = 2.9834, b = 2.3833

They match! Both methods give the same answer.

Minimum error: 3.0085
Optimization converged: True
```

7.5 IV. Polynomial Fitting

7.5.1 Generalizing to Higher-Order Polynomials

For a polynomial of degree j: $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_j x^j$

The least squares condition leads to the **normal equations** in matrix form:

$$\begin{pmatrix} n & \sum x_i & \sum x_i^2 & \cdots & \sum x_i^j \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \cdots & \sum x_i^{j+1} \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \cdots & \sum x_i^{j+2} \\ \vdots & & & \ddots & \vdots \\ \sum x_i^j & \sum x_i^{j+1} & \sum x_i^{j+2} & \cdots & \sum x_i^{2j} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_j \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \\ \vdots \\ \sum x_i^j y_i \end{pmatrix}$$

This is again $\mathbf{A}\vec{p} = \vec{c}$, where $\mathbf{A}$ is a **(j+1) × (j+1)** matrix.

7.5.2 Note

The matrix **A** is called the **Vandermonde matrix** and can become ill-conditioned for high polynomial degrees.

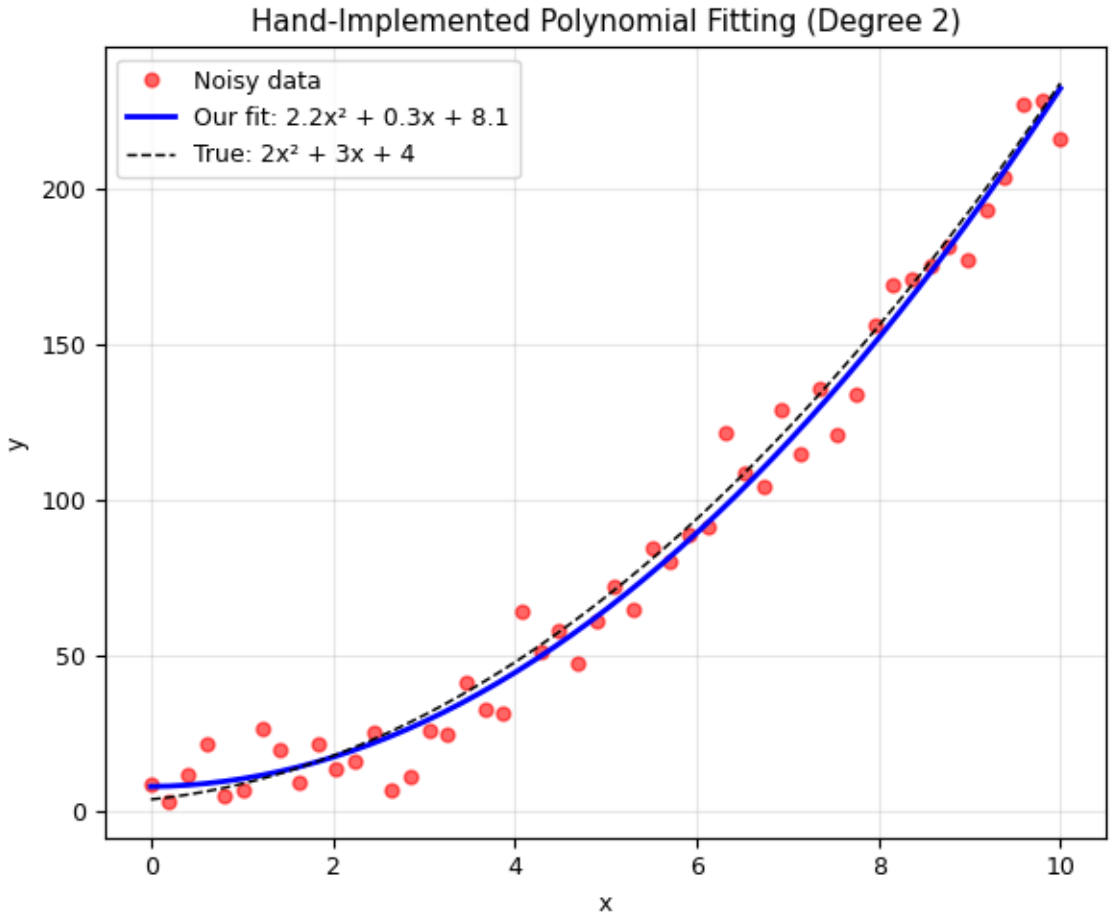
```
# Implement polynomial fitting from scratch
def poly_fit_manual(x, y, degree):
    """
    Polynomial least squares fit - implemented from scratch.

    Parameters:
        x, y: arrays of data points
        degree: degree of the polynomial

    Returns:
        coefficients [a0, a1, a2, ...]
    """
    n = degree + 1  # Number of coefficients

    # Build the normal equation matrix A
    A = np.zeros((n, n))
    for row in range(n):
        for col in range(n):
            # code..
            # A[row, col] = ...
            A[row, col] = np.sum(x**(row+col))

    # Build the right-hand side vector c
    c = np.zeros(n)
    for row in range(n):
        # code ..
        # c[row] = ...
        c[row] = np.sum(y*x**row)


    # Solve for coefficients
    coeffs = np.linalg.solve(A, c)
    return coeffs  # [a0, a1, a2, ...]

def poly_eval(coeffs, x):
    """Evaluate polynomial given coefficients [a0, a1, a2, ...]."""
    result = np.zeros_like(x, dtype=float)
    for i, a in enumerate(coeffs):
        result += a * x**i
    return result

# Generate quadratic data with noise
np.random.seed(42)
x_poly = np.linspace(0, 10, 50)
y_poly = 2*x_poly**2 + 3*x_poly + 4 + np.random.normal(0, 10, 50)

# Fit with our hand-implemented function
coeffs = poly_fit_manual(x_poly, y_poly, 2)
print(f'True coefficients:   a0 = 4, a1 = 3, a2 = 2')
print(f'Fitted coefficients: a0 = {coeffs[0]:.2f}, a1 = {coeffs[1]:.2f}, a2 =
 ↪{coeffs[2]:.2f}')

# Plot
x_plot = np.linspace(0, 10, 200)
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(6, 5))
plt.plot(x_poly, y_poly, 'ro', markersize=5, alpha=0.6, label='Noisy data')
plt.plot(x_plot, poly_eval(coeffs, x_plot), 'b-', linewidth=2,
         label=f'Our fit: {coeffs[2]:.1f}x² + {coeffs[1]:.1f}x + {coeffs[0]:.1f}')
plt.plot(x_plot, 2*x_plot**2 + 3*x_plot + 4, 'k--', linewidth=1,
         label='True: 2x² + 3x + 4')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Hand-Implemented Polynomial Fitting (Degree 2)')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
True coefficients:   a0 = 4, a1 = 3, a2 = 2
Fitted coefficients: a0 = 8.14, a1 = 0.28, a2 = 2.21
```



7.5.3 Using `np.polyfit` from NumPy

NumPy provides `np.polyfit(x, y, degree)` for polynomial fitting, and `np.polyval(p, x)` for evaluation.

```
# Compare our result with numpy's polyfit
p_numpy = np.polyfit(x_poly, y_poly, 2)  # Returns [a2, a1, a0] (highest first!)

print('Our implementation: a0={:.2f}, a1={:.2f}, a2={:.2f}'.format(*coeffs))
print('np.polyfit:         a0={:.2f}, a1={:.2f}, a2={:.2f}'.format(p_numpy[2], p_
 ↪numpy[1], p_numpy[0]))
print('\nNote: np.polyfit returns coefficients in REVERSE order (highest degree␣
 ↪first)!')
```

```
Our implementation: a0=8.14, a1=0.28, a2=2.21
np.polyfit:         a0=8.14, a1=0.28, a2=2.21

Note: np.polyfit returns coefficients in REVERSE order (highest degree first)!
```

7.5.4 Choosing the Right Polynomial Degree

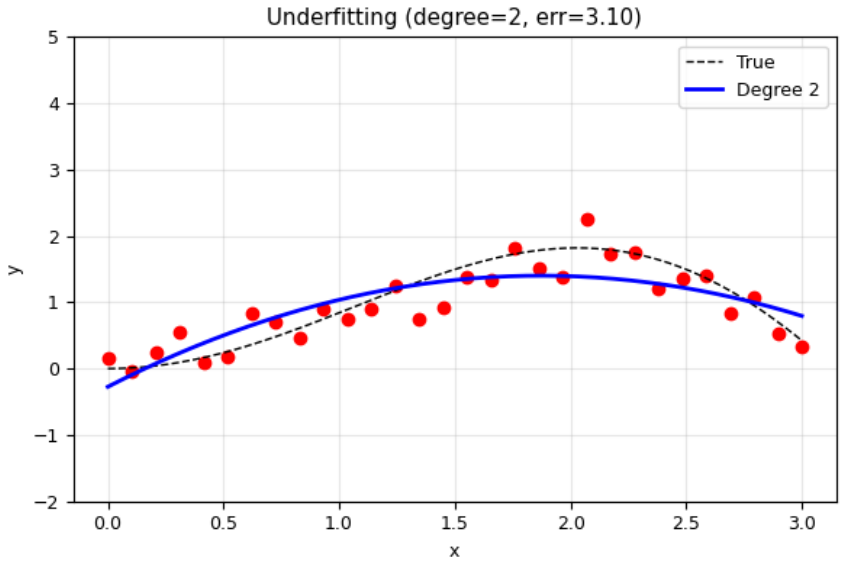
Quiz: Try different polynomial degrees. Which one gives the best fit?

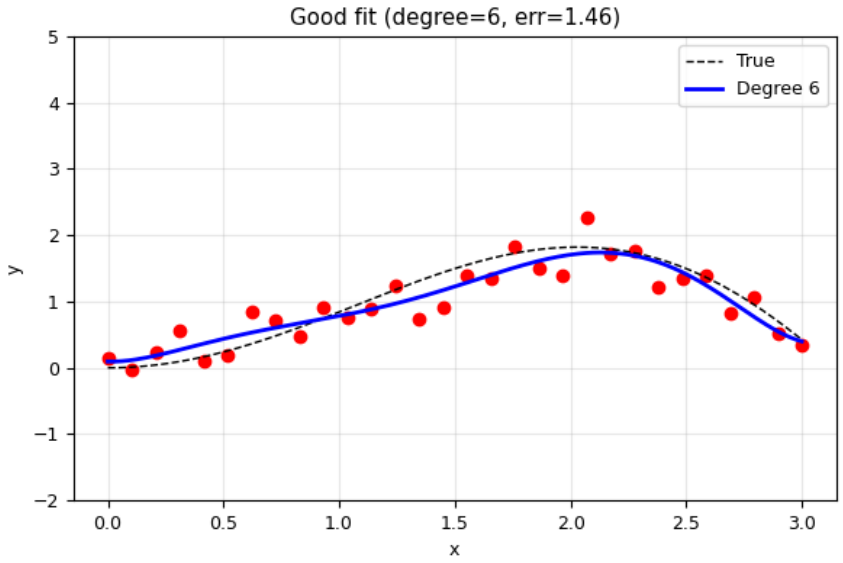
```
# Effect of polynomial degree: underfitting vs overfitting
f_complex = lambda x: x * np.sin(x)

np.random.seed(42)
x_demo = np.linspace(0, 3, 30)
y_demo = f_complex(x_demo) + 0.3 * np.random.randn(30)
x_fine_demo = np.linspace(0, 3, 200)

plt.figure(figsize=(6, 12))

for idx, order in enumerate([2, 6, 14]):
    plt.subplot(3, 1, idx + 1)

    fit = np.polyfit(x_demo, y_demo, order)
    y_pred = np.polyval(fit, x_fine_demo)

    # Compute residual
    y_pred_data = np.polyval(fit, x_demo)
    residual = np.sum((y_demo - y_pred_data)**2)

    plt.plot(x_demo, y_demo, 'ro', markersize=6)
    plt.plot(x_fine_demo, f_complex(x_fine_demo), 'k--', linewidth=1, label='True')
    plt.plot(x_fine_demo, y_pred, 'b-', linewidth=2, label=f'Degree {order}')
    plt.xlabel('x')
    plt.ylabel('y')

    title = ['Underfitting', 'Good fit', 'Overfitting'][idx]
    plt.title(f'{title} (degree={order}, err={residual:.2f})')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.ylim(-2, 5)

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

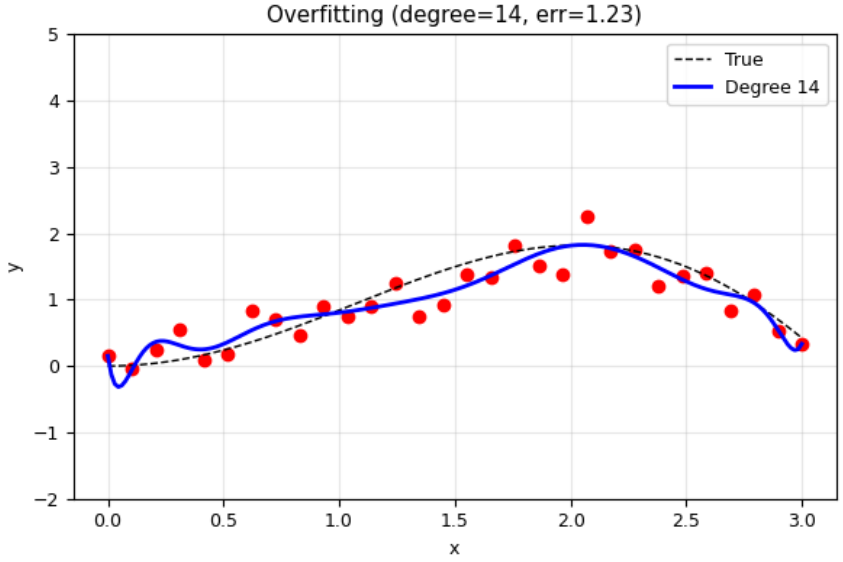
(continues on next page)

(continued from previous page)

```
print('Underfitting: Model too simple — misses the trend')
print('Overfitting:  Model too complex — fits the noise, not the signal')
print('Good fit:     Right balance — captures the trend, ignores the noise')
```


Underfitting (degree=2, err=3.10)
True
Degree 2
y
x


Good fit (degree=6, err=1.46)
True
Degree 6
y
x


Overfitting (degree=14, err=1.23)
True
Degree 14
y
x

```
Underfitting: Model too simple — misses the trend
Overfitting:  Model too complex — fits the noise, not the signal
Good fit:     Right balance — captures the trend, ignores the noise
```

7.6 V. General Curve Fitting

What if the model is not a polynomial? For example:

- Exponential decay: $f(x) = a\, e^{-bx} + c$
- Gaussian: $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/(2\sigma^2)}$
- Power law: $f(x) = a\, x^b$
- Sinusoidal: $f(x) = A \sin(\omega x + \phi)$

7.6.1 Two Approaches

1. **`scipy.optimize.minimize`** — Define an error function and minimize it (we already know this!)
2. **`scipy.optimize.curve_fit`** — A convenient wrapper designed specifically for fitting

7.6.2 Method 1: Using `scipy.optimize.minimize` (by hand)

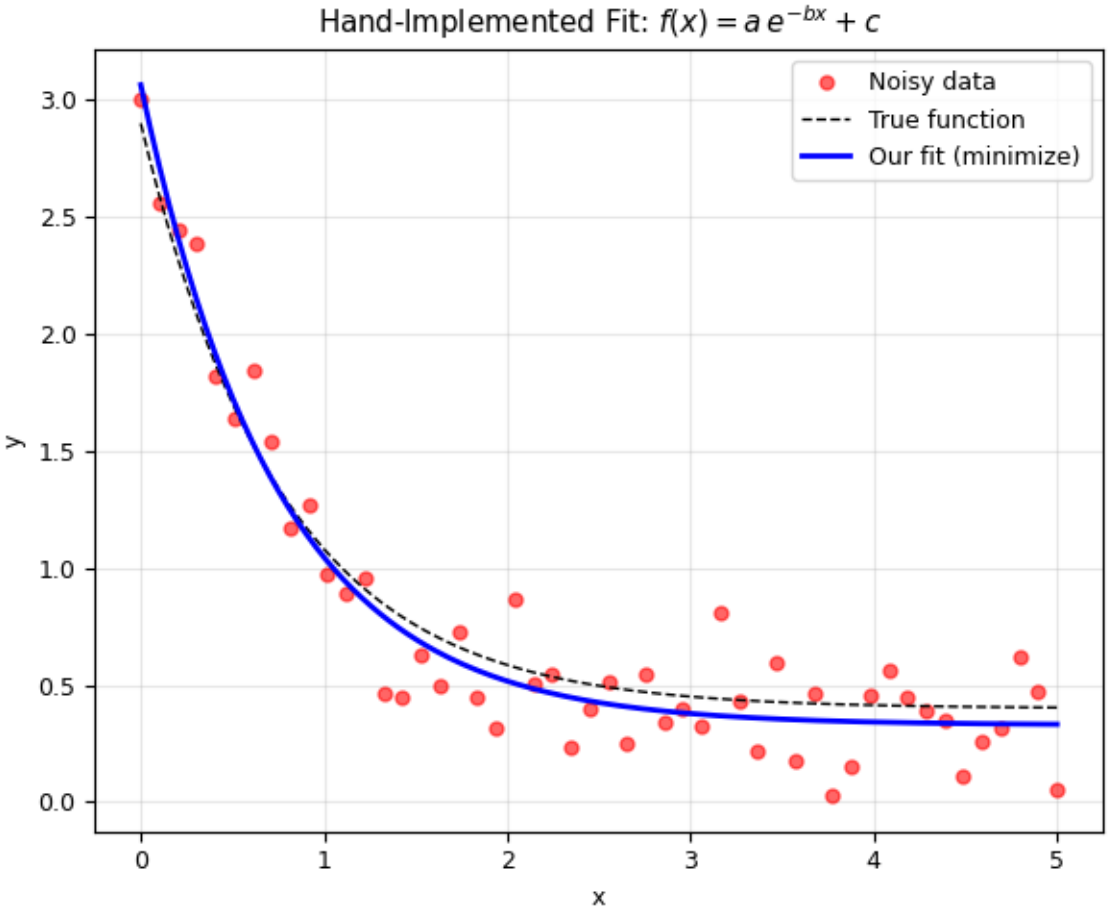
```
# Example: Fit an exponential decay f(x) = a * exp(-b*x) + c
from scipy.optimize import minimize

# Generate data
np.random.seed(42)
a_true, b_true, c_true = 2.5, 1.3, 0.4
x_exp = np.linspace(0, 5, 50)
y_exp = a_true * np.exp(-b_true * x_exp) + c_true + 0.2 * np.random.randn(50)

# Step 1: Define the model function
def exp_model(x, a, b, c):
    return a * np.exp(-b * x) + c

# Step 2: Define the error function
def err_exp(params, x, y):
    a, b, c = params
    y_pred = exp_model(x, a, b, c)
    return np.sum((y - y_pred)**2)

# Step 3: Minimize!
param_init = [1.0, 1.0, 0.0]  # Initial guess
result = minimize(err_exp, param_init, args=(x_exp, y_exp))
a_fit, b_fit, c_fit = result.x

print('True parameters:    a={:.3f}, b={:.3f}, c={:.3f}'.format(a_true, b_true, c_
↪true))
print('Fitted parameters:  a={:.3f}, b={:.3f}, c={:.3f}'.format(a_fit, b_fit, c_fit))

# Plot
```

(continues on next page)

(continued from previous page)

```
x_plot = np.linspace(0, 5, 200)
plt.figure(figsize=(6, 5))
plt.plot(x_exp, y_exp, 'ro', markersize=5, alpha=0.6, label='Noisy data')
plt.plot(x_plot, exp_model(x_plot, a_true, b_true, c_true), 'k--', linewidth=1, label=
 ↪'True function')
plt.plot(x_plot, exp_model(x_plot, a_fit, b_fit, c_fit), 'b-', linewidth=2, label=
 ↪'Our fit (minimize)')
plt.xlabel('x')
plt.ylabel('y')
plt.title(r'Hand-Implemented Fit: $f(x) = a \, e^{-bx} + c$')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
True parameters:    a=2.500, b=1.300, c=0.400
Fitted parameters:  a=2.736, b=1.339, c=0.329
```



7.6.3 Method 2: Using `scipy.optimize.curve_fit`

`curve_fit` is a convenient function designed specifically for fitting. It:

- Takes the model function, x data, and y data
- Returns optimal parameters AND their covariance matrix (uncertainty!)

```
from scipy.optimize import curve_fit

# curve_fit expects f(x, param1, param2, ...)
# It returns (optimal_params, covariance_matrix)
params_cf, covariance = curve_fit(exp_model, x_exp, y_exp, p0=[1, 1, 0])

# Extract parameter uncertainties from covariance matrix
uncertainties = np.sqrt(np.diag(covariance))

print('curve_fit results:')
print(f'  a = {params_cf[0]:.4f} +/- {uncertainties[0]:.4f}')
print(f'  b = {params_cf[1]:.4f} +/- {uncertainties[1]:.4f}')
print(f'  c = {params_cf[2]:.4f} +/- {uncertainties[2]:.4f}')
print(f'\nTrue values: a={a_true}, b={b_true}, c={c_true}')
print('\nAdvantage of curve_fit: it gives you uncertainties automatically!')
```

```
curve_fit results:
  a = 2.7355 +/- 0.1188
  b = 1.3390 +/- 0.1132
  c = 0.3289 +/- 0.0396

True values: a=2.5, b=1.3, c=0.4

Advantage of curve_fit: it gives you uncertainties automatically!
```

7.6.4 Example: Gaussian Fitting

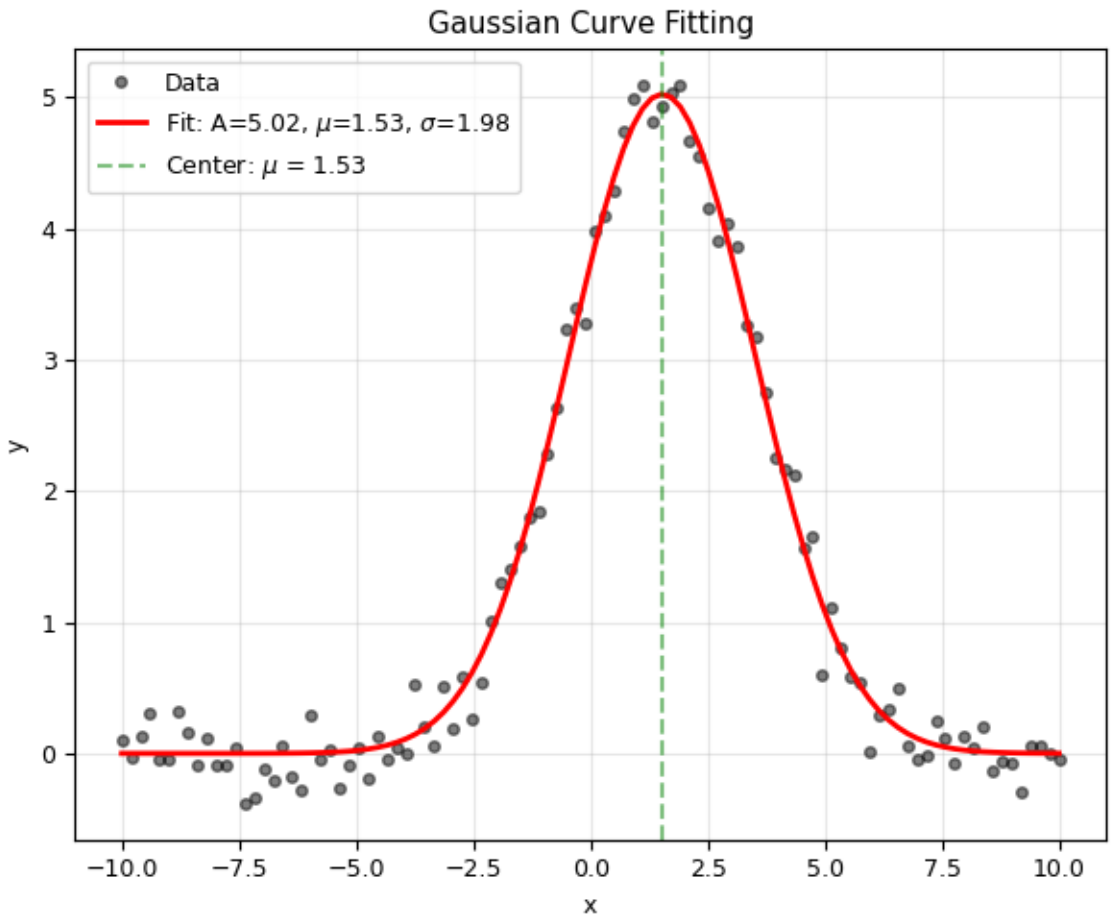
Fitting a Gaussian peak is very common in spectroscopy, particle physics, and signal processing.

```
# Gaussian fitting example
def gaussian(x, A, mu, sigma):
    """Gaussian function with amplitude A, center mu, width sigma."""
    return A * np.exp(-(x - mu)**2 / (2 * sigma**2))

# Generate data
np.random.seed(42)
x_gauss = np.linspace(-10, 10, 100)
y_gauss = gaussian(x_gauss, 5.0, 1.5, 2.0) + 0.2 * np.random.randn(100)

# Fit
params_gauss, cov_gauss = curve_fit(gaussian, x_gauss, y_gauss, p0=[4, 0, 1])
A_fit, mu_fit, sigma_fit = params_gauss
errs = np.sqrt(np.diag(cov_gauss))

# Plot
plt.figure(figsize=(6, 5))
plt.plot(x_gauss, y_gauss, 'ko', markersize=4, alpha=0.5, label='Data')
plt.plot(x_gauss, gaussian(x_gauss, *params_gauss), 'r-', linewidth=2,
```

(continues on next page)

(continued from previous page)

```
        label=f'Fit: A={A_fit:.2f}, $\\mu$={mu_fit:.2f}, $\\sigma$={sigma_fit:.2f}')
plt.axvline(mu_fit, color='g', linestyle='--', alpha=0.5, label=f'Center: $\\mu$ =
 ↪{mu_fit:.2f}')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Gaussian Curve Fitting')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f'Fitted: A = {A_fit:.3f} +/- {errs[0]:.3f}')
print(f'        mu = {mu_fit:.3f} +/- {errs[1]:.3f}')
print(f'        sigma = {sigma_fit:.3f} +/- {errs[2]:.3f}')
print(f'True:   A = 5.000, mu = 1.500, sigma = 2.000')
```




```
Fitted: A = 5.024 +/- 0.054
        mu = 1.531 +/- 0.024
        sigma = 1.980 +/- 0.024
True:   A = 5.000, mu = 1.500, sigma = 2.000
```

7.7 VI. Goodness of Fit

How do we know if our fit is good? Several metrics:

7.7.1 Residual Sum of Squares (RSS)

$$\text{RSS} = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

7.7.2 Coefficient of Determination (R^2)

$$R^2 = 1 - \frac{\text{RSS}}{\text{TSS}} = 1 - \frac{\sum (y_i - f(x_i))^2}{\sum (y_i - \bar{y})^2}$$

where TSS = Total Sum of Squares.

- $R^2 = 1$: Perfect fit
- $R^2 = 0$: Model no better than the mean
- $R^2 < 0$: Model is worse than the mean!

7.7.3 Reduced Chi-Squared (χ^2_ν)

$$\chi^2_\nu = \frac{1}{n-p} \sum_{i=1}^{n} \frac{(y_i - f(x_i))^2}{\sigma_i^2}$$

where p = number of parameters, σ_i = measurement uncertainty.

- $\chi^2_\nu \approx 1$: Good fit
- $\chi^2_\nu \gg 1$: Poor fit (or underestimated uncertainties)
- $\chi^2_\nu \ll 1$: Overfitting (or overestimated uncertainties)

```
def compute_r_squared(y_data, y_pred):
    """Compute R² (coefficient of determination)."""
    ss_res = np.sum((y_data - y_pred)**2)  # Residual sum of squares
    ss_tot = np.sum((y_data - np.mean(y_data))**2)  # Total sum of squares
    return 1 - ss_res / ss_tot

# Compare R² for different polynomial degrees
print('R² for different polynomial degrees:')
print('=' * 40)
for degree in range(1, 8):
    p = np.polyfit(x_demo, y_demo, degree)
    y_pred = np.polyval(p, x_demo)
    r2 = compute_r_squared(y_demo, y_pred)
    print(f'  Degree {degree}: R² = {r2:.6f}')

print('\nNote: R² always increases with degree (more parameters)!')
print('But higher R² does NOT always mean a better model.')
print('Use adjusted R² or cross-validation to avoid overfitting.')
```

```
R² for different polynomial degrees:
========================================
  Degree 1: R² = 0.315799
  Degree 2: R² = 0.679083
  Degree 3: R² = 0.827932
  Degree 4: R² = 0.834795
  Degree 5: R² = 0.841264
  Degree 6: R² = 0.848789
  Degree 7: R² = 0.852316

Note: R² always increases with degree (more parameters)!
But higher R² does NOT always mean a better model.
Use adjusted R² or cross-validation to avoid overfitting.
```

7.7.4 Residual Analysis

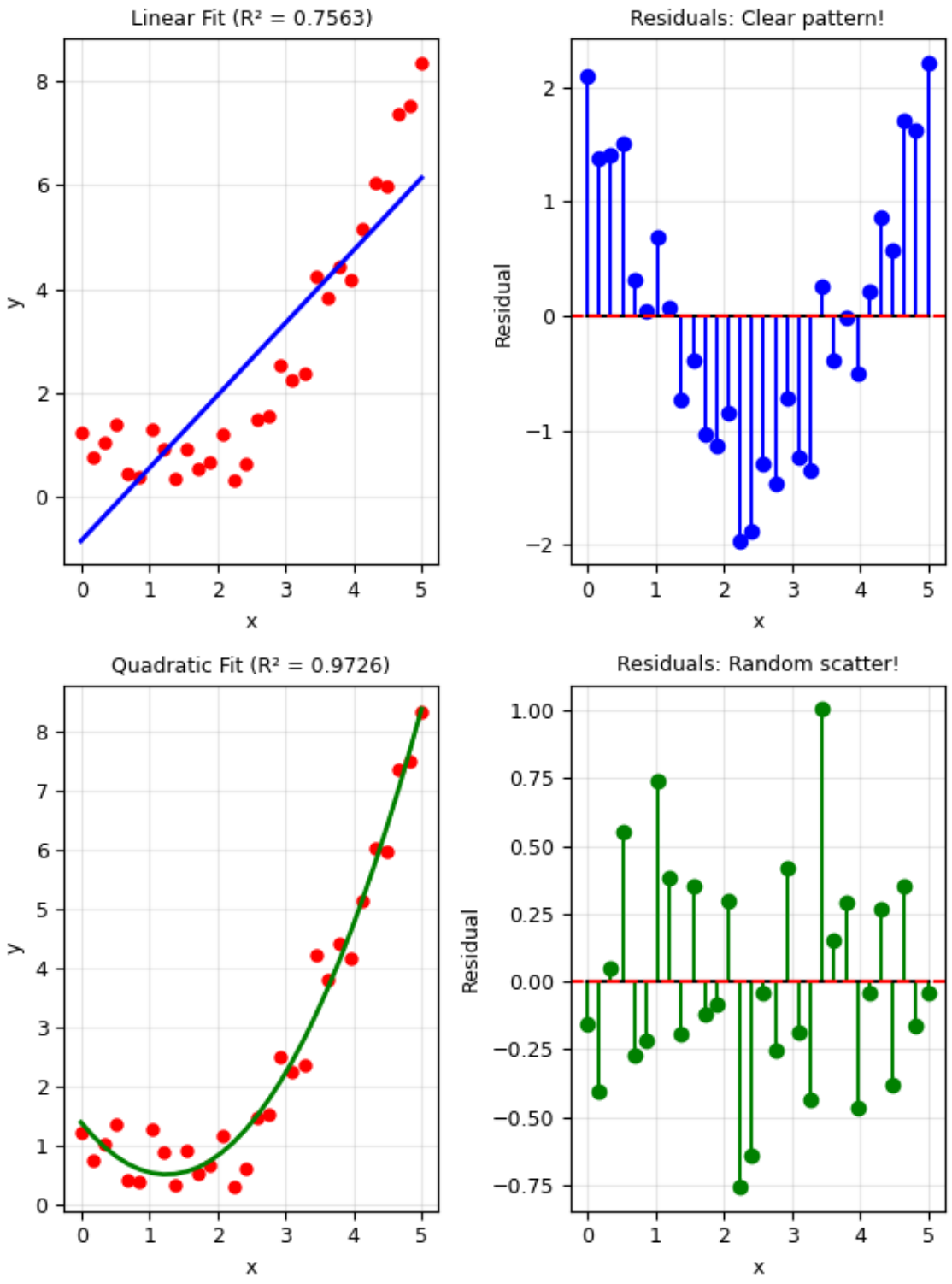
A good fit should have **random** residuals. Patterns in residuals indicate a poor model.

```
# Residual analysis: linear vs quadratic fit on quadratic data
np.random.seed(42)
x_res = np.linspace(0, 5, 30)
y_res = 0.5 * x_res**2 - x_res + 1 + 0.5 * np.random.randn(30)

# Linear fit
p_lin = np.polyfit(x_res, y_res, 1)
residuals_lin = y_res - np.polyval(p_lin, x_res)

# Quadratic fit
p_quad = np.polyfit(x_res, y_res, 2)
residuals_quad = y_res - np.polyval(p_quad, x_res)

fig, axes = plt.subplots(2, 2, figsize=(6, 8))

# Linear fit
axes[0, 0].plot(x_res, y_res, 'ro', markersize=5)
axes[0, 0].plot(x_res, np.polyval(p_lin, x_res), 'b-', linewidth=2)
axes[0, 0].set_title(f'Linear Fit (R² = {compute_r_squared(y_res, np.polyval(p_lin, x_
↪res)):.4f})', fontsize=9)
axes[0, 0].set_xlabel('x')
axes[0, 0].set_ylabel('y')
axes[0, 0].grid(True, alpha=0.3)

axes[0, 1].stem(x_res, residuals_lin, linefmt='b-', markerfmt='bo', basefmt='k-')
axes[0, 1].axhline(0, color='r', linestyle='--')
axes[0, 1].set_title('Residuals: Clear pattern!', fontsize=9)
axes[0, 1].set_xlabel('x')
axes[0, 1].set_ylabel('Residual')
axes[0, 1].grid(True, alpha=0.3)

# Quadratic fit
axes[1, 0].plot(x_res, y_res, 'ro', markersize=5)
axes[1, 0].plot(x_res, np.polyval(p_quad, x_res), 'g-', linewidth=2)
axes[1, 0].set_title(f'Quadratic Fit (R² = {compute_r_squared(y_res, np.polyval(p_
↪quad, x_res)):.4f})', fontsize=9)
axes[1, 0].set_xlabel('x')
axes[1, 0].set_ylabel('y')
```

(continues on next page)

(continued from previous page)

```
axes[1, 0].grid(True, alpha=0.3)

axes[1, 1].stem(x_res, residuals_quad, linefmt='g-', markerfmt='go', basefmt='k-')
axes[1, 1].axhline(0, color='r', linestyle='--')
axes[1, 1].set_title('Residuals: Random scatter!', fontsize=9)
axes[1, 1].set_xlabel('x')
axes[1, 1].set_ylabel('Residual')
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print('Linear fit: Residuals show a pattern -> wrong model!')
print('Quadratic fit: Residuals are random -> good model!')
```




```
Linear fit: Residuals show a pattern -> wrong model!
Quadratic fit: Residuals are random -> good model!
```

7.8 VII. Distance Metrics: Beyond Least Squares

The least squares (L_2 norm) is the most common, but not the only option:

7.8.1 L_p Distances

$$\text{err}_{L_p} = \left(\sum_{i=1}^{n} |y_i - f(x_i)|^p \right)^{1/p}$$

7.8.2 Distance Metrics

Distance	Properties
L_1 (Manhattan)	Robust to outliers
L_2 (Euclidean)	Most common, smooth
L_∞ (Chebyshev)	Minimizes worst case

L_1 (Manhattan)

$$\sum_i |y_i - f(x_i)|$$

L_2 (Euclidean)
$$\sqrt{\sum_i (y_i - f(x_i))^2}$$

L_∞ (Chebyshev)

$$\max_i |y_i - f(x_i)|$$

7.8.3 When to use which?

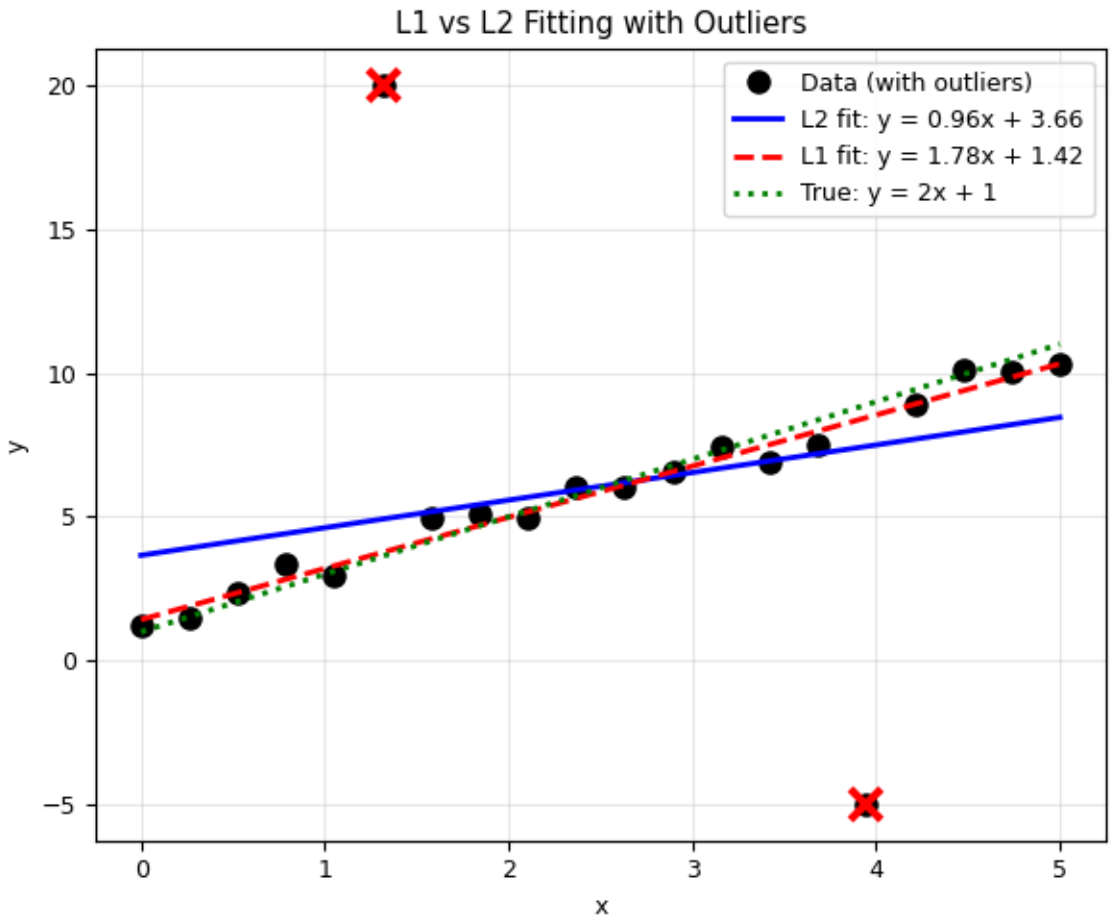
- L_2: Default choice, best for Gaussian noise
- L_1: When data has outliers (more robust)
- L_∞: When worst-case error matters

```
# Demonstrate L1 vs L2 with outliers
np.random.seed(42)
x_out = np.linspace(0, 5, 20)
y_out = 2 * x_out + 1 + 0.5 * np.random.randn(20)

# Add outliers
y_out[5] = 20   # Far from trend
y_out[15] = -5   # Far from trend

# L2 fit (standard least squares)
p_L2 = np.polyfit(x_out, y_out, 1)

# L1 fit (minimize absolute deviations)
def err_L1(params, x, y):
```

(continues on next page)

(continued from previous page)

```
    a, b = params
    return np.sum(np.abs(y - (a * x + b)))

result_L1 = minimize(err_L1, [1, 1], args=(x_out, y_out), method='Nelder-Mead')
a_L1, b_L1 = result_L1.x

plt.figure(figsize=(6, 5))
plt.plot(x_out, y_out, 'ko', markersize=8, label='Data (with outliers)')
plt.plot(x_out[5], y_out[5], 'rx', markersize=12, markeredgewidth=3)
plt.plot(x_out[15], y_out[15], 'rx', markersize=12, markeredgewidth=3)
plt.plot(x_out, np.polyval(p_L2, x_out), 'b-', linewidth=2,
         label=f'L2 fit: y = {p_L2[0]:.2f}x + {p_L2[1]:.2f}')
plt.plot(x_out, a_L1 * x_out + b_L1, 'r--', linewidth=2,
         label=f'L1 fit: y = {a_L1:.2f}x + {b_L1:.2f}')
plt.plot(x_out, 2 * x_out + 1, 'g:', linewidth=2, label='True: y = 2x + 1')

plt.xlabel('x')
plt.ylabel('y')
plt.title('L1 vs L2 Fitting with Outliers')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print('L2 is pulled toward the outliers (sensitive to large errors).')
print('L1 is more robust — closer to the true line despite outliers.')
```




```
L2 is pulled toward the outliers (sensitive to large errors).
L1 is more robust — closer to the true line despite outliers.
```

7.9 VIII. Physics Applications

7.9.1 1. Radioactive Decay

Radioactive decay follows: $N(t) = N_0\, e^{-\lambda t}$

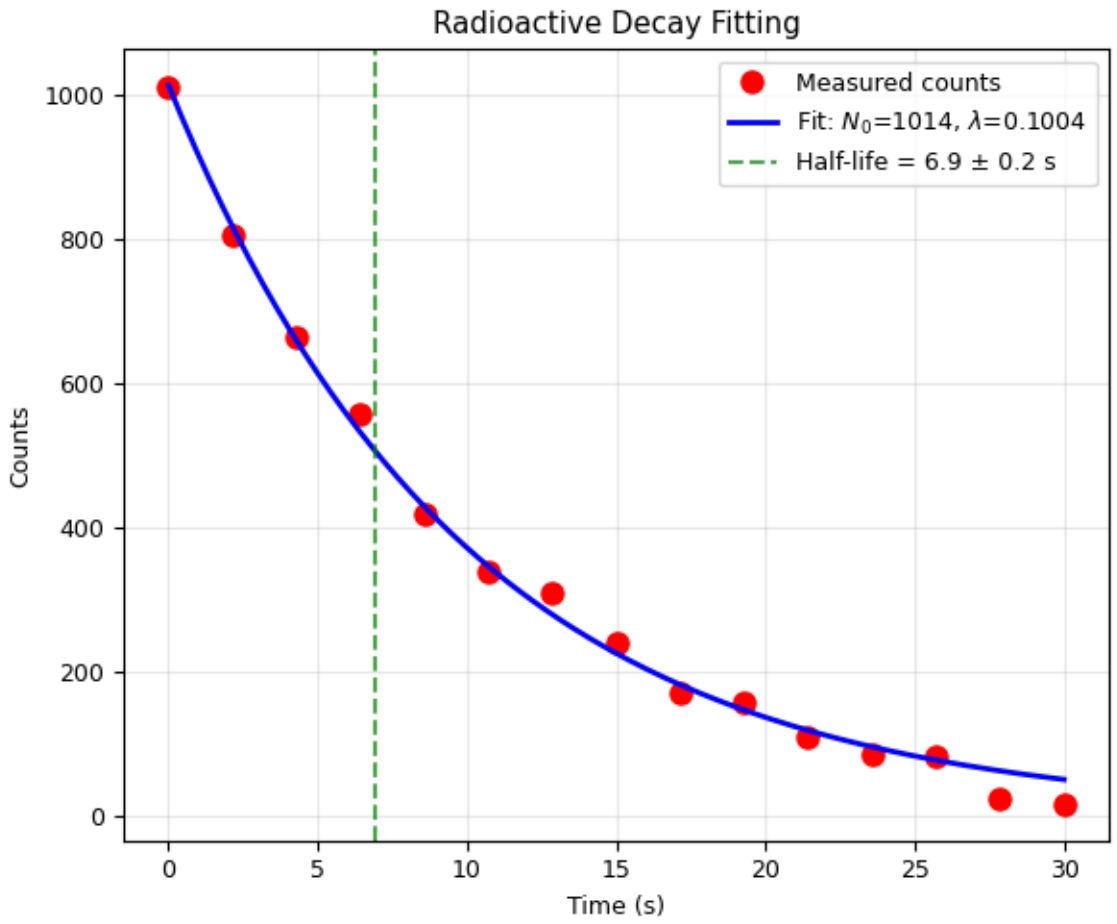
where λ is the decay constant and $t_{1/2} = \ln 2/\lambda$.

```python
# Radioactive decay fitting
def decay_model(t, N0, lam):
    return N0 * np.exp(-lam * t)

# Simulate "experimental" data
np.random.seed(42)
N0_true, lam_true = 1000, 0.1  # True parameters
t_data = np.linspace(0, 30, 15)
N_data = decay_model(t_data, N0_true, lam_true) + 20 * np.random.randn(15)
N_data = np.maximum(N_data, 0)  # Can't have negative counts
```

(continues on next page)

(continued from previous page)

```
# Fit
params_decay, cov_decay = curve_fit(decay_model, t_data, N_data, p0=[800, 0.05])
N0_fit, lam_fit = params_decay
errs_decay = np.sqrt(np.diag(cov_decay))

# Calculate half-life
half_life = np.log(2) / lam_fit
half_life_err = np.log(2) / lam_fit**2 * errs_decay[1]  # Error propagation

# Plot
t_fine = np.linspace(0, 30, 200)
plt.figure(figsize=(6, 5))
plt.plot(t_data, N_data, 'ro', markersize=8, label='Measured counts')
plt.plot(t_fine, decay_model(t_fine, *params_decay), 'b-', linewidth=2,
         label=f'Fit: $N_0$={N0_fit:.0f}, $\\lambda$={lam_fit:.4f}')
plt.axvline(half_life, color='g', linestyle='--', alpha=0.7,
            label=f'Half-life = {half_life:.1f} ± {half_life_err:.1f} s')
plt.xlabel('Time (s)')
plt.ylabel('Counts')
plt.title('Radioactive Decay Fitting')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f'True:   N0 = {N0_true}, lambda = {lam_true}, half-life = {np.log(2)/lam_true:.
 ↪2f} s')
print(f'Fitted: N0 = {N0_fit:.1f} +/- {errs_decay[0]:.1f}')
print(f'        lambda = {lam_fit:.4f} +/- {errs_decay[1]:.4f}')
print(f'        half-life = {half_life:.2f} +/- {half_life_err:.2f} s')
```




```
True:   N0 = 1000, lambda = 0.1, half-life = 6.93 s
Fitted: N0 = 1013.6 +/- 15.2
        lambda = 0.1004 +/- 0.0024
        half-life = 6.91 +/- 0.17 s
```

7.9.2 2. Damped Oscillation

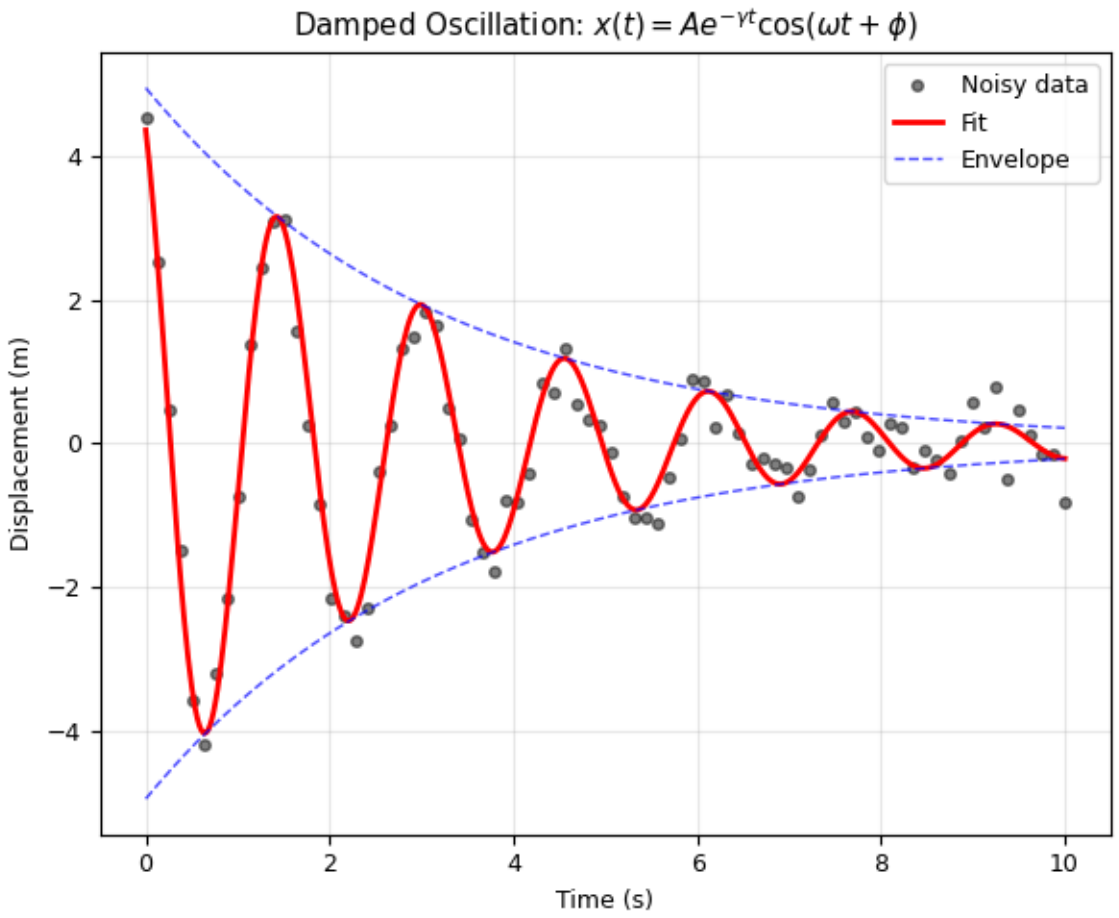
A damped oscillator: $x(t) = A\,e^{-\gamma t}\cos(\omega t + \phi)$

```python
# Damped oscillation fitting
def damped_oscillation(t, A, gamma, omega, phi):
    return A * np.exp(-gamma * t) * np.cos(omega * t + phi)

# Simulate data
np.random.seed(42)
A_true, gamma_true, omega_true, phi_true = 5.0, 0.3, 4.0, 0.5
t_osc = np.linspace(0, 10, 80)
x_osc = damped_oscillation(t_osc, A_true, gamma_true, omega_true, phi_true)
x_osc_noisy = x_osc + 0.3 * np.random.randn(len(t_osc))

# Fit
params_osc, cov_osc = curve_fit(damped_oscillation, t_osc, x_osc_noisy,
                                p0=[4, 0.2, 3.5, 0])
```

(continues on next page)

(continued from previous page)

```
A_fit, gamma_fit, omega_fit, phi_fit = params_osc

# Plot
t_fine = np.linspace(0, 10, 500)
plt.figure(figsize=(6, 5))
plt.plot(t_osc, x_osc_noisy, 'ko', markersize=4, alpha=0.5, label='Noisy data')
plt.plot(t_fine, damped_oscillation(t_fine, *params_osc), 'r-', linewidth=2, label=
↪'Fit')
plt.plot(t_fine, A_fit * np.exp(-gamma_fit * t_fine), 'b--', linewidth=1,
         alpha=0.7, label='Envelope')
plt.plot(t_fine, -A_fit * np.exp(-gamma_fit * t_fine), 'b--', linewidth=1, alpha=0.7)
plt.xlabel('Time (s)')
plt.ylabel('Displacement (m)')
plt.title(r'Damped Oscillation: $x(t) = A e^{-\gamma t} \cos(\omega t + \phi)$')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f'Fitted parameters:')
print(f'  Amplitude A = {A_fit:.3f} (true: {A_true})')
print(f'  Damping gamma = {gamma_fit:.3f} s^-1 (true: {gamma_true})')
print(f'  Frequency omega = {omega_fit:.3f} rad/s (true: {omega_true})')
print(f'  Phase phi = {phi_fit:.3f} rad (true: {phi_true})')
print(f'\n  Quality factor Q = omega/(2*gamma) = {omega_fit/(2*gamma_fit):.2f}')
```




```
Fitted parameters:
  Amplitude A = 4.941 (true: 5.0)
  Damping gamma = 0.314 s^-1 (true: 0.3)
  Frequency omega = 4.014 rad/s (true: 4.0)
  Phase phi = 0.493 rad (true: 0.5)

  Quality factor Q = omega/(2*gamma) = 6.40
```

7.9.3 3. Power Law: Kepler's Third Law

Kepler's Third Law: $T^2 = a^3$ (in appropriate units)

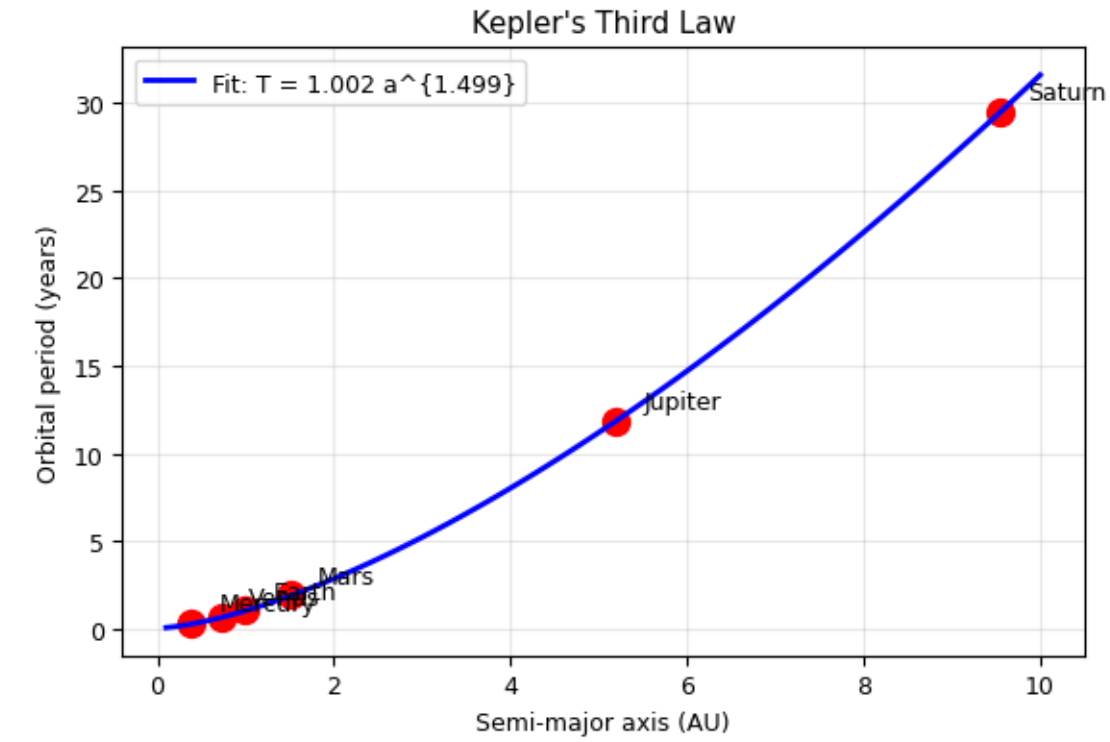
Or equivalently: $T = C \cdot a^{3/2}$, which is a power law!

```
# Kepler's Third Law
def power_law(x, C, n):
    return C * x**n

# Planetary data
planet_names = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn']
a_AU = np.array([0.39, 0.72, 1.00, 1.52, 5.20, 9.54])  # Semi-major axis (AU)
T_yr = np.array([0.24, 0.62, 1.00, 1.88, 11.86, 29.46])  # Period (years)
```

(continues on next page)

(continued from previous page)

```
# Fit power law T = C * a^n
params_kepler, cov_kepler = curve_fit(power_law, a_AU, T_yr, p0=[1, 1.5])
C_fit, n_fit = params_kepler
errs_kepler = np.sqrt(np.diag(cov_kepler))

fig, axes = plt.subplots(2, 1, figsize=(6, 8))

# Linear plot
a_fine = np.linspace(0.1, 10, 100)
axes[0].plot(a_AU, T_yr, 'ro', markersize=10)
for i, name in enumerate(planet_names):
    axes[0].annotate(name, (a_AU[i], T_yr[i]), textcoords='offset points',
                     xytext=(10, 5), fontsize=9)
axes[0].plot(a_fine, power_law(a_fine, *params_kepler), 'b-', linewidth=2,
             label=f'Fit: T = {C_fit:.3f} a^{{{n_fit:.3f}}}')
axes[0].set_xlabel('Semi-major axis (AU)')
axes[0].set_ylabel('Orbital period (years)')
axes[0].set_title("Kepler's Third Law")
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Log-log plot (power law becomes a straight line!)
axes[1].loglog(a_AU, T_yr, 'ro', markersize=10)
axes[1].loglog(a_fine, power_law(a_fine, *params_kepler), 'b-', linewidth=2,
               label=f'Slope = {n_fit:.3f} (expected: 1.500)')
axes[1].set_xlabel('Semi-major axis (AU)')
axes[1].set_ylabel('Orbital period (years)')
axes[1].set_title('Log-Log Plot: Power Law = Straight Line')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Fitted: T = {C_fit:.4f} * a^{n_fit:.4f}")
print(f"Expected (Kepler): T = 1.000 * a^1.500")
print(f"\nFitted exponent: {n_fit:.4f} +/- {errs_kepler[1]:.4f}")
print(f"Deviation from 3/2: {abs(n_fit - 1.5):.4f}")
print(f"\nKepler's Third Law confirmed!")
```




```
Fitted: T = 1.0020 * a^1.4990
Expected (Kepler): T = 1.000 * a^1.500

Fitted exponent: 1.4990 +/- 0.0006
Deviation from 3/2: 0.0010
```

(continues on next page)

(continued from previous page)

```
Kepler's Third Law confirmed!
```

7.9.4 4. Econophysics: Bitcoin Power Law

Econophysics applies methods from statistical physics to financial systems. It's a recognized branch of physics studied at institutions worldwide since the 1990s.

Power laws appear throughout econophysics:

- Wealth distribution (Pareto's Law)
- Price return distributions (fat tails)
- Network growth effects (Metcalfe's Law)

A remarkable empirical observation: Bitcoin's price appears to follow a **power law** in time:

$$P(t) = a \cdot t^n$$

where t is time (days since Bitcoin's creation). On a log-log plot, this should appear as a straight line.

Key question: Is this a fundamental "law" like Kepler's, or just an empirical trend?

```
# Econophysics: Bitcoin Power Law Fitting
# Historical Bitcoin price data (monthly averages, selected points)
# Source: publicly available Bitcoin price history

# Days since Bitcoin genesis block (Jan 3, 2009)
# Selected data points spanning 2010-2024
btc_days = np.array([
    365,    # Jan 2010
    545,    # Jul 2010
    730,    # Jan 2011
    912,    # Jul 2011
    1095,   # Jan 2012
    1460,   # Jan 2013
    1642,   # Jul 2013
    1825,   # Jan 2014
    2190,   # Jan 2015
    2555,   # Jan 2016
    2920,   # Jan 2017
    3100,   # Jul 2017
    3285,   # Jan 2018
    3650,   # Jan 2019
    4015,   # Jan 2020
    4380,   # Jan 2021
    4562,   # Jul 2021
    4745,   # Jan 2022
    5110,   # Jan 2023
    5475,   # Jan 2024
    5650,   # Jul 2024
])

# Approximate price (USD) at those dates
btc_price = np.array([
    0.05,     # Jan 2010
```

(continues on next page)

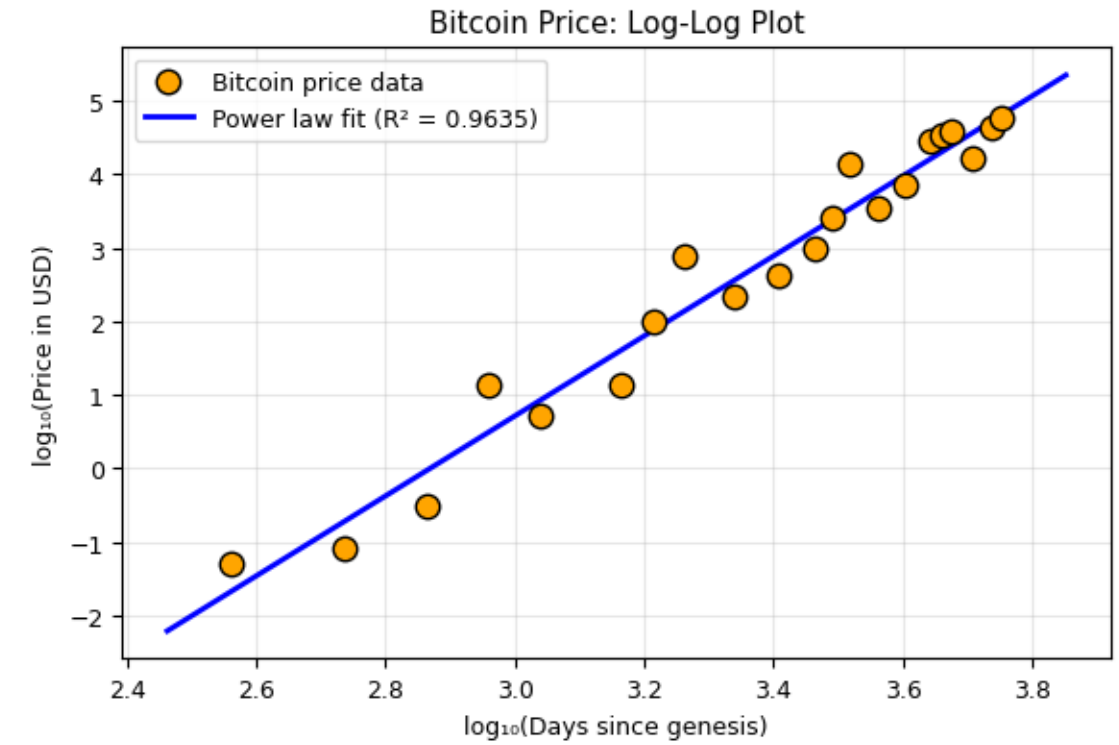
(continued from previous page)

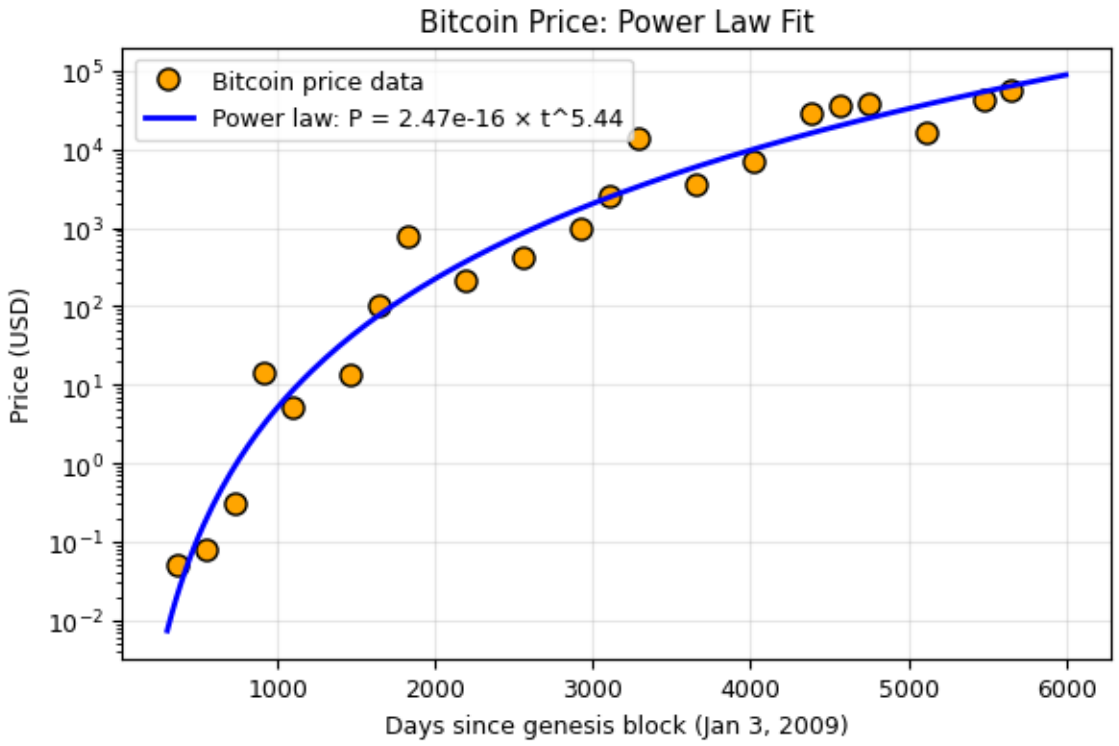
```
    0.08,      # Jul 2010
    0.30,      # Jan 2011
    14.0,      # Jul 2011
    5.3,       # Jan 2012
    13.3,      # Jan 2013
    100.0,     # Jul 2013
    770.0,     # Jan 2014
    215.0,     # Jan 2015
    430.0,     # Jan 2016
    960.0,     # Jan 2017
    2500.0,    # Jul 2017
    13400.0,   # Jan 2018
    3500.0,    # Jan 2019
    7200.0,    # Jan 2020
    29000.0,   # Jan 2021
    35000.0,   # Jul 2021
    38500.0,   # Jan 2022
    16500.0,   # Jan 2023
    42000.0,   # Jan 2024
    57000.0,   # Jul 2024
])

# Fit power law: P = a * t^n
# Use log-log linearization for robustness
log_days = np.log10(btc_days)
log_price = np.log10(btc_price)

# Linear fit in log-log space: log(P) = log(a) + n*log(t)
p_btc = np.polyfit(log_days, log_price, 1)
n_btc = p_btc[0]
a_btc = 10**p_btc[1]

# Also fit directly with curve_fit
def power_law(x, a, n):
    return a * x**n

params_btc, cov_btc = curve_fit(power_law, btc_days, btc_price,
                                p0=[1e-10, 5], maxfev=10000)
a_btc_cf, n_btc_cf = params_btc

# R² in log space
r2_btc = compute_r_squared(log_price, np.polyval(p_btc, log_days))

fig, axes = plt.subplots(2, 1, figsize=(6, 8))

# Log-log plot
axes[0].scatter(log_days, log_price, c='orange', s=80, edgecolors='black',
                linewidths=1, zorder=5, label='Bitcoin price data')
log_days_fine = np.linspace(np.min(log_days) - 0.1, np.max(log_days) + 0.1, 100)
axes[0].plot(log_days_fine, np.polyval(p_btc, log_days_fine), 'b-', linewidth=2,
             label=f'Power law fit (R² = {r2_btc:.4f})')
axes[0].set_xlabel('log₁₀(Days since genesis)')
axes[0].set_ylabel('log₁₀(Price in USD)')
axes[0].set_title('Bitcoin Price: Log-Log Plot')
axes[0].legend()
axes[0].grid(True, alpha=0.3)
```

(continues on next page)

(continued from previous page)

```
# Regular scale with fit
days_fine = np.linspace(300, 6000, 200)
axes[1].semilogy(btc_days, btc_price, 'o', color='orange', markersize=8,
                 markeredgecolor='black', label='Bitcoin price data')
axes[1].semilogy(days_fine, a_btc * days_fine**n_btc, 'b-', linewidth=2,
                 label=f'Power law: P = {a_btc:.2e} × t^{n_btc:.2f}')
axes[1].set_xlabel('Days since genesis block (Jan 3, 2009)')
axes[1].set_ylabel('Price (USD)')
axes[1].set_title('Bitcoin Price: Power Law Fit')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f'Power law fit (log-log linear regression):')
print(f'  P(t) = {a_btc:.2e} × t^{n_btc:.2f}')
print(f'  Exponent n = {n_btc:.3f}')
print(f'  R² (in log space) = {r2_btc:.4f}')
print(f'\nPower law fit (direct curve_fit):')
print(f'  P(t) = {a_btc_cf:.2e} × t^{n_btc_cf:.2f}')
```





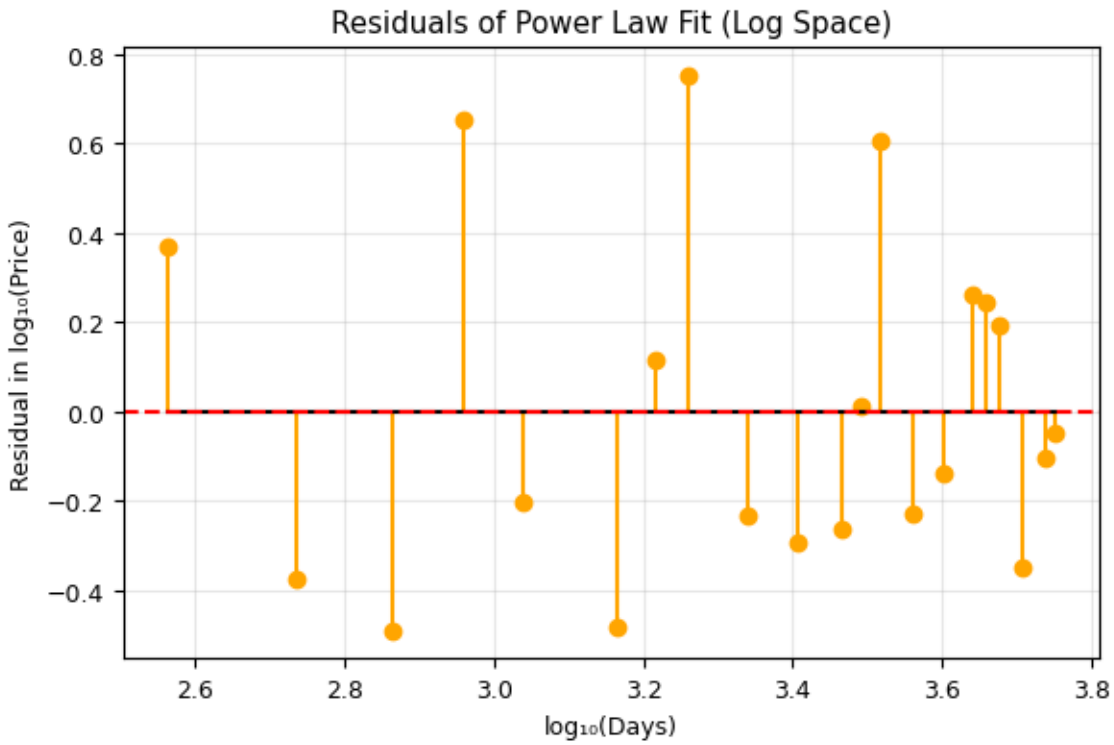

```
Power law fit (log-log linear regression):
  P(t) = 2.47e-16 × t^5.44
  Exponent n = 5.440
  R² (in log space) = 0.9635
```

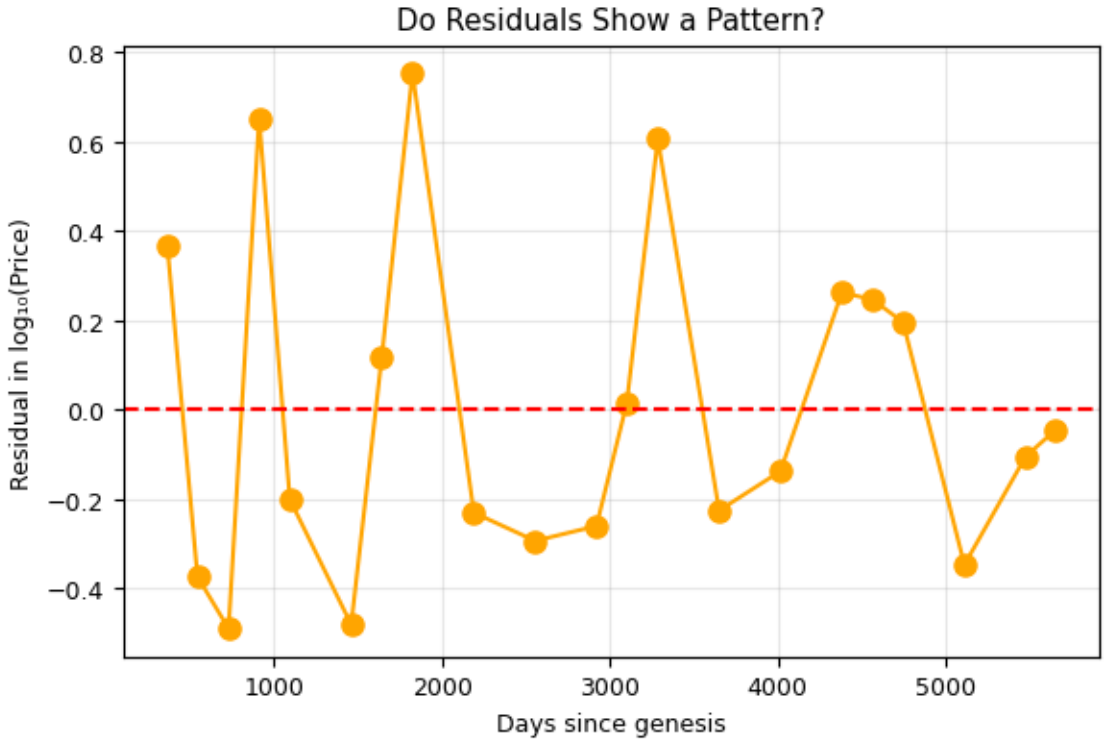
(continues on next page)

(continued from previous page)

```
Power law fit (direct curve_fit):
  P(t) = 1.59e-09 × t^3.60
```

```
# Critical analysis: examine the residuals in log space
residuals_btc = log_price - np.polyval(p_btc, log_days)

fig, axes = plt.subplots(2, 1, figsize=(6, 8))

# Residuals
axes[0].stem(log_days, residuals_btc, linefmt='orange', markerfmt='o', basefmt='k-')
axes[0].axhline(0, color='r', linestyle='--')
axes[0].set_xlabel('log₁₀(Days)')
axes[0].set_ylabel('Residual in log₁₀(Price)')
axes[0].set_title('Residuals of Power Law Fit (Log Space)')
axes[0].grid(True, alpha=0.3)

# Residuals vs time to check for patterns
axes[1].plot(btc_days, residuals_btc, 'o-', color='orange', markersize=8)
axes[1].axhline(0, color='r', linestyle='--')
axes[1].set_xlabel('Days since genesis')
axes[1].set_ylabel('Residual in log₁₀(Price)')
axes[1].set_title('Do Residuals Show a Pattern?')
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print('Discussion questions:')
print('1. Do the residuals look random, or is there a pattern (e.g., cycles)?')
print('2. The oscillations around the power law may relate to Bitcoin halving cycles␣
 ↪(~4 years)')
print('3. How is this different from Kepler\'s Third Law?')
print('   - Kepler: derived from fundamental physics (gravity)')
print('   - Bitcoin: empirical observation from a complex system (no first-principles␣
 ↪derivation)')
print('4. Can we use this fit to PREDICT future prices? Why or why not?')
print('5. What would falsify the power law model?')
```






```
Discussion questions:
1. Do the residuals look random, or is there a pattern (e.g., cycles)?
2. The oscillations around the power law may relate to Bitcoin halving cycles (~4␣
↪years)
3. How is this different from Kepler's Third Law?
```

(continues on next page)

(continued from previous page)

```
   - Kepler: derived from fundamental physics (gravity)
   - Bitcoin: empirical observation from a complex system (no first-principles␣
↪derivation)
4. Can we use this fit to PREDICT future prices? Why or why not?
5. What would falsify the power law model?
```

7.10 IX. Advanced Topics (Brief Overview)

7.10.1 Maximum Likelihood Estimation (MLE)

Finds parameters that maximize the probability of observing the data. For Gaussian noise, MLE is equivalent to least squares!

7.10.2 Bayesian Fitting

Uses Bayes' theorem to compute the **posterior probability** distribution of parameters:

$$P(\text{params}|\text{data}) \propto P(\text{data}|\text{params}) \times P(\text{params})$$

This gives not just the best parameters, but their full probability distribution!

7.10.3 Weighted Least Squares

When data points have different uncertainties σ_i:

$$\text{err} = \sum_i \frac{(y_i - f(x_i))^2}{\sigma_i^2}$$

Points with smaller error bars contribute more to the fit.

```
# Weighted vs unweighted fitting
np.random.seed(42)
x_w = np.linspace(0, 5, 15)
y_true_w = 2 * x_w + 1

# Different uncertainties: first few points very precise, last few noisy
sigma_w = np.concatenate([0.1 * np.ones(8), 2.0 * np.ones(7)])
y_w = y_true_w + sigma_w * np.random.randn(15)

# Unweighted fit
p_unw = np.polyfit(x_w, y_w, 1)

# Weighted fit
p_weighted = np.polyfit(x_w, y_w, 1, w=1/sigma_w)

plt.figure(figsize=(6, 5))
plt.errorbar(x_w, y_w, yerr=sigma_w, fmt='ko', markersize=6, capsize=5, label='Data')
plt.plot(x_w, np.polyval(p_unw, x_w), 'b-', linewidth=2,
         label=f'Unweighted: y = {p_unw[0]:.2f}x + {p_unw[1]:.2f}')
plt.plot(x_w, np.polyval(p_weighted, x_w), 'r--', linewidth=2,
```

(continues on next page)

(continued from previous page)

```
         label=f'Weighted: y = {p_weighted[0]:.2f}x + {p_weighted[1]:.2f}')
plt.plot(x_w, y_true_w, 'g:', linewidth=2, label='True: y = 2x + 1')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Weighted vs Unweighted Fitting')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print('Weighted fit trusts precise points more -> closer to true line!')
```

CHAPTER

EIGHT

THE FOURIER TRANSFORM

Any signal — no matter how complex — can be decomposed into a sum of simple sine and cosine waves.

8.1 Why Fourier Analysis?

Time Domain	Frequency Domain
What happens at each moment	What frequencies are present
Natural for measurement	Natural for understanding
Convolution is hard	Convolution becomes multiplication
Differential equations are hard	Become algebraic equations

Note: We focus on understanding the *mathematics* and *implementation* of the DFT. The Fast Fourier Transform (FFT) algorithm will be covered in the next lecture.

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

# For nicer plots (projector-friendly)
plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9
```

8.2 I. The Big Idea: Building Signals from Sines

Joseph Fourier (1807) showed that **any** periodic function can be written as a sum of sines and cosines:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{2\pi n t}{T}\right) + b_n \sin\left(\frac{2\pi n t}{T}\right) \right]$$

where T is the period. This is the **Fourier Series**.

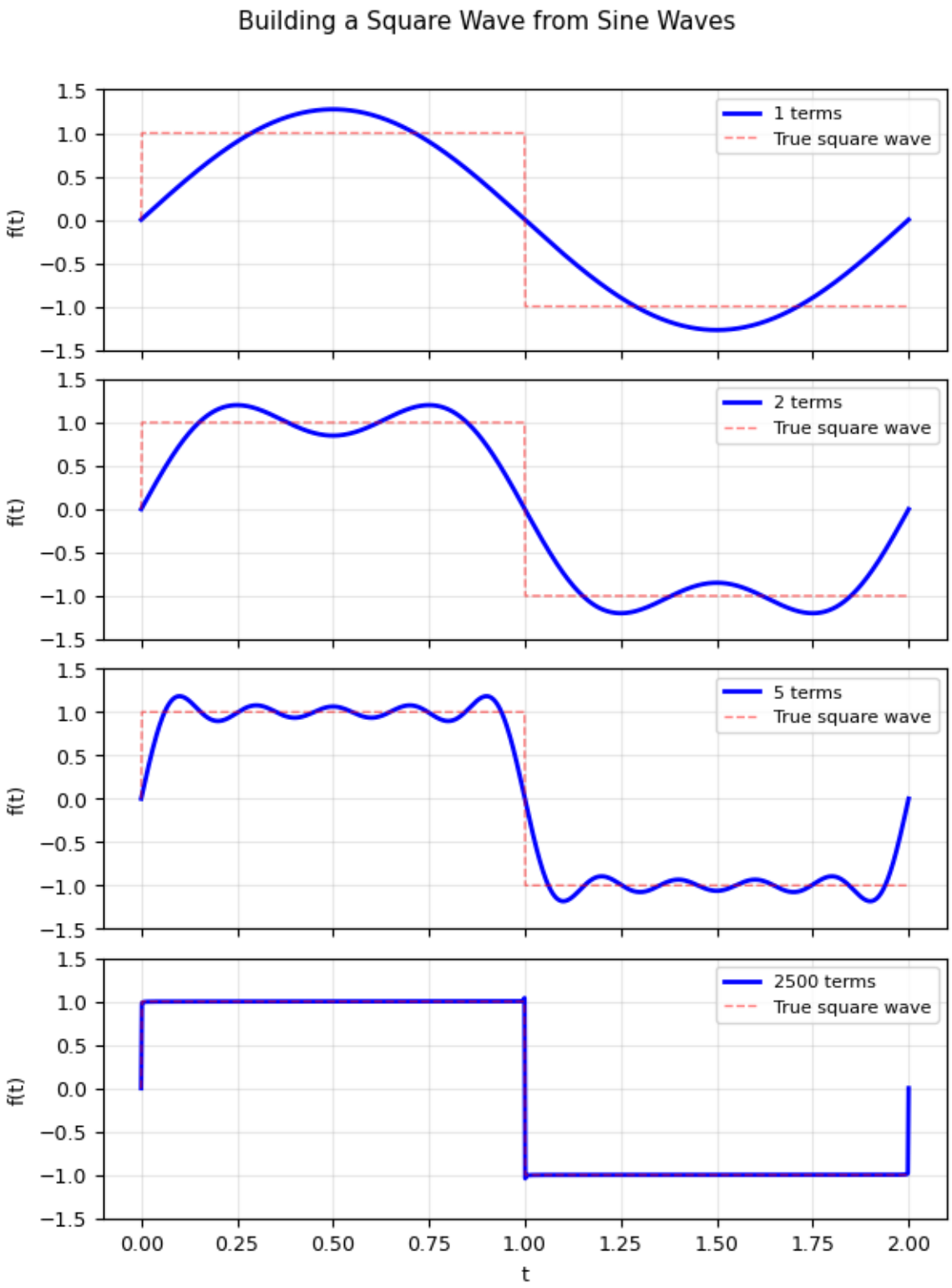
Let's see this in action — building a square wave from sines.

```
# Build a square wave from sine waves!
t = np.linspace(0, 2, 1000)  # 2 periods

# Square wave Fourier series: f(t) = (4/pi) * sum[ sin(n*pi*t) / n ] for odd n
plt.figure(figsize=(6, 8))
```

(continues on next page)

(continued from previous page)

```
n_terms_list = [1, 3, 9, 5000]
for idx, N in enumerate(n_terms_list):
    plt.subplot(4, 1, idx + 1)

    # Build the partial sum
    f = np.zeros_like(t)
    for n in range(1, N + 1, 2):  # odd n only
        f += (4 / np.pi) * np.sin(2 * np.pi * n * t / 2) / n

    plt.plot(t, f, 'b-', linewidth=2, label=f'{(N+1)//2} terms')

    # True square wave for reference
    square = np.sign(np.sin(2 * np.pi * t / 2))
    plt.plot(t, square, 'r--', linewidth=1, alpha=0.5, label='True square wave')

    plt.ylim(-1.5, 1.5)
    plt.ylabel('f(t)')
    plt.legend(loc='upper right', fontsize=8)
    plt.grid(True, alpha=0.3)
    if idx < 3:
        plt.tick_params(labelbottom=False)

plt.xlabel('t')
plt.suptitle('Building a Square Wave from Sine Waves', y=1.01)
plt.tight_layout()
plt.show()

print('More terms → better approximation!')
print('Notice the overshoot at the edges — this is the Gibbs phenomenon.')
```




```
More terms → better approximation!
Notice the overshoot at the edges — this is the Gibbs phenomenon.
```

8.3 II. Fourier Series Coefficients

For a periodic function $f(t)$ with period T, the Fourier coefficients are:

8.3.1 Cosine coefficients

$$a_n = \frac{2}{T} \int_0^T f(t) \cos\left(\frac{2\pi n t}{T}\right) dt$$

8.3.2 Sine coefficients

$$b_n = \frac{2}{T} \int_0^T f(t) \sin\left(\frac{2\pi n t}{T}\right) dt$$

8.3.3 DC component (average value)

$$a_0 = \frac{2}{T} \int_0^T f(t)\, dt$$

8.3.4 Why does this work?

Orthogonality! Sine and cosine functions are orthogonal over one period:

$$\int_0^T \sin\left(\frac{2\pi m t}{T}\right) \sin\left(\frac{2\pi n t}{T}\right) dt = \begin{cases} T/2 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases}$$

This is just like dot products of orthogonal vectors — we "project" the signal onto each basis function.

```
# Hand-implement Fourier coefficient computation
# using numerical integration (trapezoidal rule from Lecture 04!)

def fourier_coefficients(f_values, t, T, N_max):
    """
    Compute Fourier series coefficients numerically.

    Parameters:
        f_values: array of function values over one period
        t: array of time points over one period
        T: period of the function
        N_max: number of harmonics to compute

    Returns:
        a: array of cosine coefficients [a0, a1, a2, ...]
        b: array of sine coefficients [0, b1, b2, ...]
    """
    a = np.zeros(N_max + 1)
    b = np.zeros(N_max + 1)

    for n in range(N_max + 1):
```

(continues on next page)

(continued from previous page)

```
        # Compute a_n using trapezoidal integration np.trapz()
        # using the formula: a_n = (2/T) * ∫ f(t) * cos(2πnt/T) dt
        integrand_a = f_values * np.cos(2 * np.pi * n * t / T)
        a[n] = (2/T) * np.trapz(integrand_a, t)

        # Compute b_n using trapezoidal integration np.trapz()
        # using the formula: b_n = (2/T) * ∫ f(t) * sin(2πnt/T) dt
        integrand_b = f_values * np.sin(2 * np.pi * n * t / T)
        b[n] = (2/T) * np.trapz(integrand_b, t)

    return a, b

# Test: compute coefficients of a square wave
T = 2.0  # period
t_period = np.linspace(0, T, 1000, endpoint=False)
square_wave = np.sign(np.sin(2 * np.pi * t_period / T))

N_max = 10
a_coeffs, b_coeffs = fourier_coefficients(square_wave, t_period, T, N_max)

print('Fourier coefficients of a square wave:')
print('=' * 45)
print(f'{"n":>3s}  {"a_n":>10s}  {"b_n":>10s}  {"expected b_n":>12s}')
print('-' * 45)
for n in range(N_max + 1):
    expected = 4 / (np.pi * n) if (n > 0 and n % 2 == 1) else 0
    print(f'{n:3d}  {a_coeffs[n]:10.4f}  {b_coeffs[n]:10.4f}  {expected:12.4f}')

print('\nAll a_n ≈ 0 (square wave is odd → no cosine terms)')
print('b_n = 4/(nπ) for odd n, 0 for even n')
```

```
Fourier coefficients of a square wave:
=============================================
  n         a_n         b_n  expected b_n
---------------------------------------------
  0      0.0030      0.0000        0.0000
  1     -0.0010      1.2732        1.2732
  2      0.0030     -0.0000        0.0000
  3     -0.0010      0.4244        0.4244
  4      0.0030     -0.0000        0.0000
  5     -0.0010      0.2546        0.2546
  6      0.0030     -0.0000        0.0000
  7     -0.0010      0.1818        0.1819
  8      0.0030     -0.0001        0.0000
  9     -0.0010      0.1414        0.1415
 10      0.0030     -0.0001        0.0000

All a_n ≈ 0 (square wave is odd → no cosine terms)
b_n = 4/(nπ) for odd n, 0 for even n
```

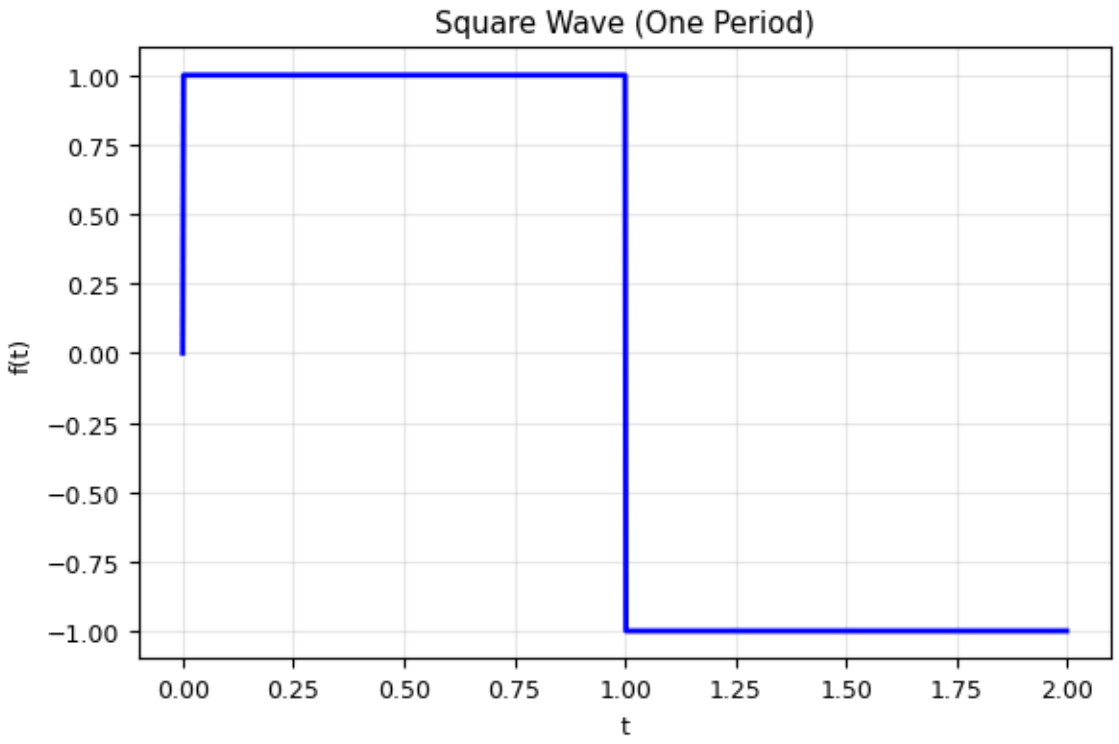
```
/tmp/ipython-input-1056247226.py:25: DeprecationWarning: `trapz` is deprecated.␣
↪Use `trapezoid` instead, or one of the numerical integration functions in `scipy.
↪integrate`.
  a[n] = (2/T) * np.trapz(integrand_a, t)
/tmp/ipython-input-1056247226.py:30: DeprecationWarning: `trapz` is deprecated.␣
↪Use `trapezoid` instead, or one of the numerical integration functions in `scipy.
```

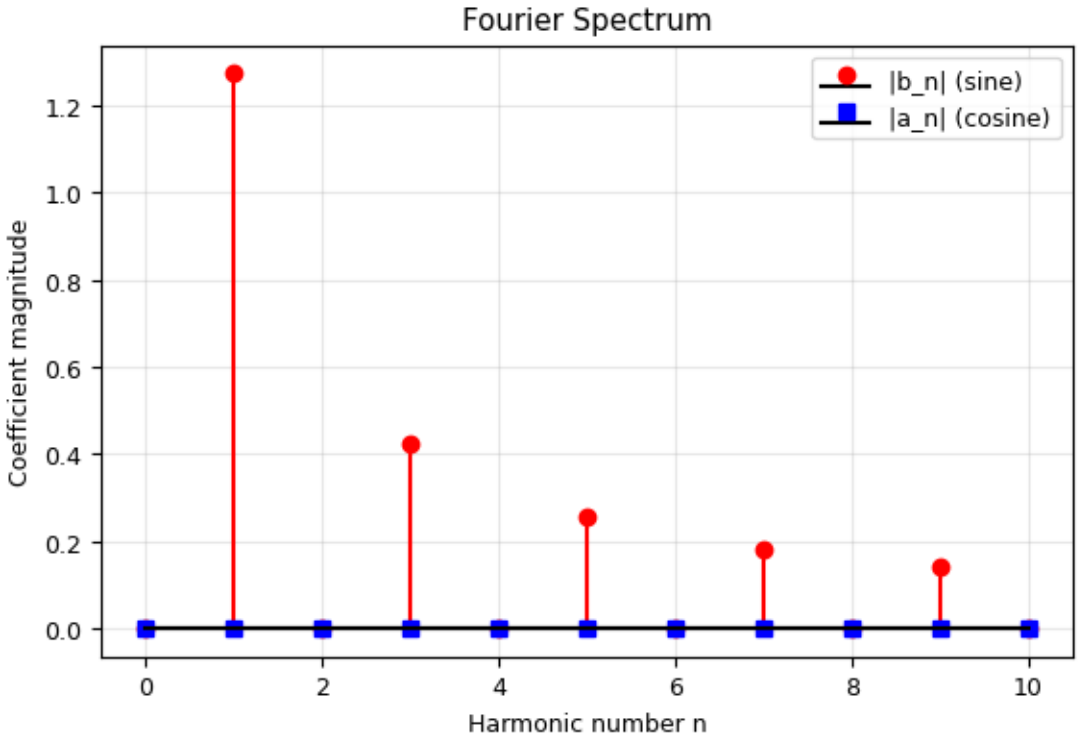
(continues on next page)

(continued from previous page)

```
    ↪integrate`.
     b[n] = (2/T) * np.trapz(integrand_b, t)
```

```
# Visualize the Fourier spectrum
plt.figure(figsize=(6, 8))

# Top: the signal
plt.subplot(2, 1, 1)
plt.plot(t_period, square_wave, 'b-', linewidth=2)
plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Square Wave (One Period)')
plt.grid(True, alpha=0.3)

# Bottom: the frequency spectrum
plt.subplot(2, 1, 2)
n_vals = np.arange(N_max + 1)
plt.stem(n_vals, np.abs(b_coeffs), linefmt='r-', markerfmt='ro', basefmt='k-',
         label='|b_n| (sine)')
plt.stem(n_vals, np.abs(a_coeffs), linefmt='b-', markerfmt='bs', basefmt='k-',
         label='|a_n| (cosine)')
plt.xlabel('Harmonic number n')
plt.ylabel('Coefficient magnitude')
plt.title('Fourier Spectrum')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print('The spectrum tells us WHICH frequencies are present and HOW MUCH of each.')
```



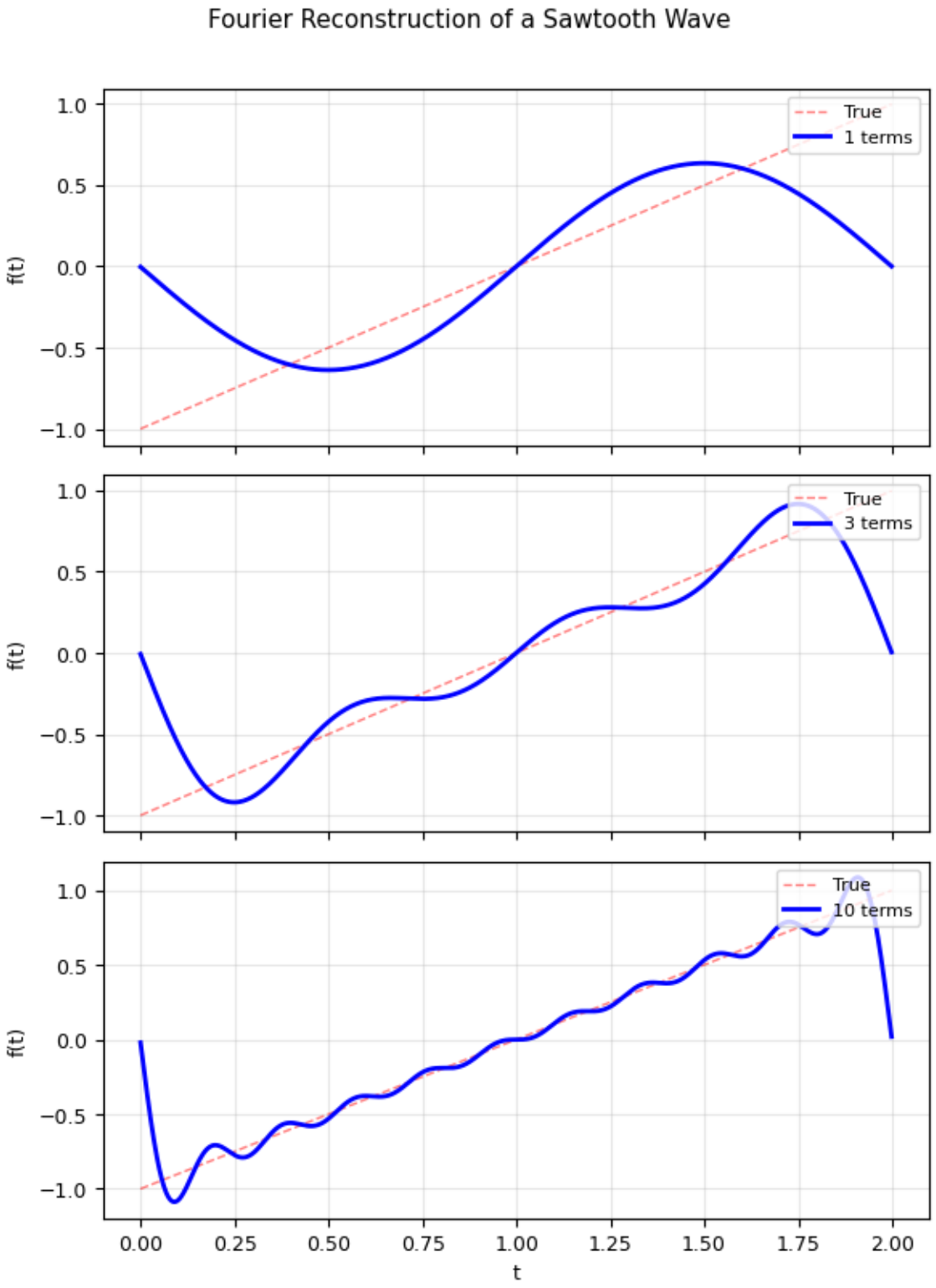



```
The spectrum tells us WHICH frequencies are present and HOW MUCH of each.
```

8.3.5 Example: Sawtooth Wave

```
# Sawtooth wave: f(t) = t/T on [0, T), periodic
sawtooth = 2 * (t_period / T) - 1  # Range [-1, 1]

a_saw, b_saw = fourier_coefficients(sawtooth, t_period, T, N_max)

# Reconstruct with increasing number of terms
plt.figure(figsize=(6, 8))

for idx, N in enumerate([1, 3, 10]):
    plt.subplot(3, 1, idx + 1)

    # Reconstruct
    f_approx = a_saw[0] / 2
    for n in range(1, N + 1):
        f_approx += a_saw[n] * np.cos(2 * np.pi * n * t_period / T)
        f_approx += b_saw[n] * np.sin(2 * np.pi * n * t_period / T)

    plt.plot(t_period, sawtooth, 'r--', linewidth=1, alpha=0.5, label='True')
    plt.plot(t_period, f_approx, 'b-', linewidth=2, label=f'{N} terms')
    plt.ylabel('f(t)')
    plt.legend(loc='upper right', fontsize=8)
    plt.grid(True, alpha=0.3)
    if idx < 2:
        plt.tick_params(labelbottom=False)

plt.xlabel('t')
plt.suptitle('Fourier Reconstruction of a Sawtooth Wave', y=1.01)
plt.tight_layout()
plt.show()
```

```
/tmp/ipython-input-1056247226.py:25: DeprecationWarning: `trapz` is deprecated.␣
↪Use `trapezoid` instead, or one of the numerical integration functions in `scipy.
↪integrate`.
  a[n] = (2/T) * np.trapz(integrand_a, t)
/tmp/ipython-input-1056247226.py:30: DeprecationWarning: `trapz` is deprecated.␣
↪Use `trapezoid` instead, or one of the numerical integration functions in `scipy.
↪integrate`.
  b[n] = (2/T) * np.trapz(integrand_b, t)
```


Fourier Reconstruction of a Sawtooth Wave
True
1 terms
True
3 terms
True
10 terms
f(t)
t
1.0
0.5
0.0
−0.5
−1.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

8.4 III. Complex Fourier Series

Using Euler's formula $e^{i\theta} = \cos\theta + i\sin\theta$, we can write the Fourier series more compactly:

$$f(t) = \sum_{n=-\infty}^{\infty} c_n \, e^{i2\pi nt/T}$$

where the **complex coefficients** are:

$$c_n = \frac{1}{T}\int_0^T f(t)\, e^{-i2\pi nt/T}\, dt$$

8.4.1 Relationship to a_n and b_n

$$c_n = \frac{1}{2}(a_n - i\, b_n) \quad \text{for } n > 0$$

$$c_0 = \frac{a_0}{2} \quad \text{(DC component)}$$

The **amplitude spectrum** is $|c_n|$ and the **power spectrum** is $|c_n|^2$.

8.4.2 Why complex?

- More compact notation
- Easier to manipulate mathematically
- Natural for the DFT and FFT
- Negative frequencies have physical meaning (rotation direction)

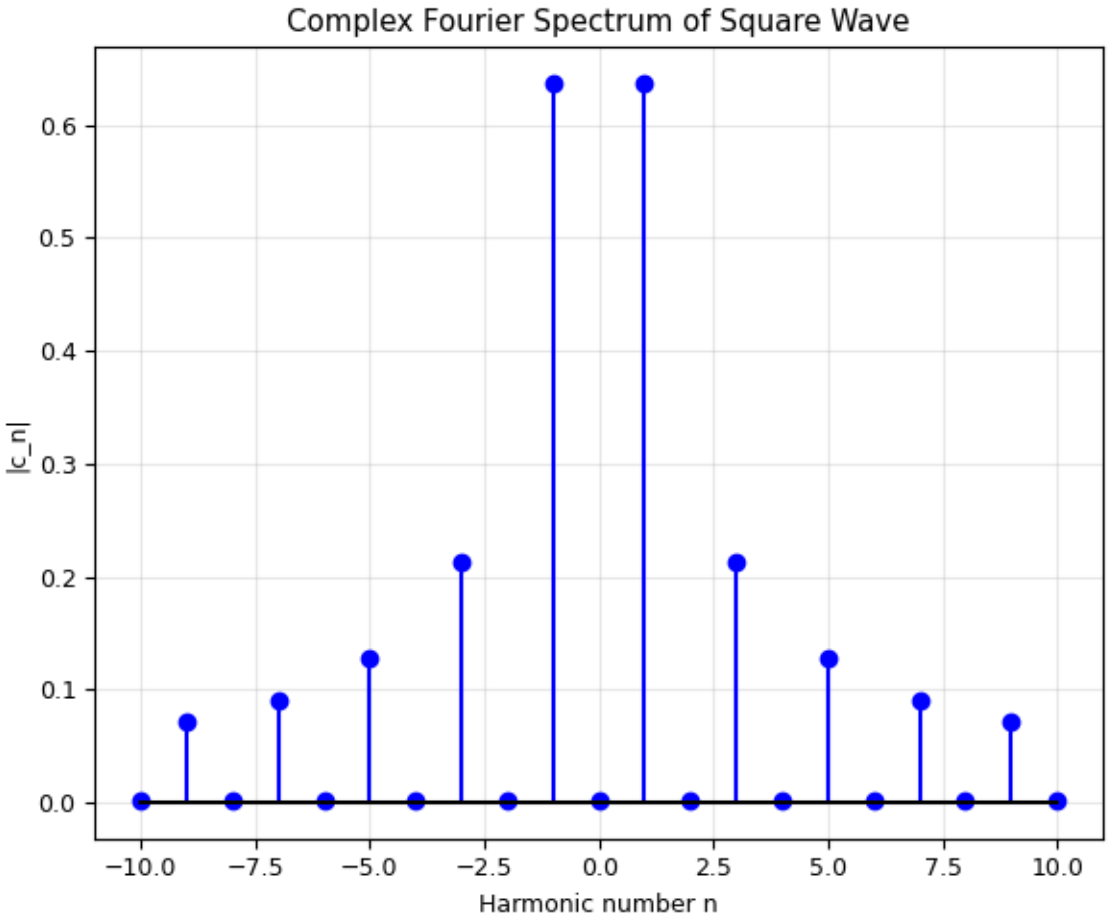
```
# Hand-implement complex Fourier coefficients
def complex_fourier_coefficients(f_values, t, T, N_max):
    """
    Compute complex Fourier coefficients c_n.

    Parameters:
        f_values: function values over one period
        t: time array over one period
        T: period
        N_max: compute c_{-N_max} to c_{N_max}

    Returns:
        n_vals: array of n values from -N_max to N_max
        c_n: complex coefficients
    """
    n_vals = np.arange(-N_max, N_max + 1)
    c_n = np.zeros(len(n_vals), dtype=complex)

    for i, n in enumerate(n_vals):
        # c_n = (1/T) * integral of f(t) * exp(-i*2*pi*n*t/T) dt
        integrand = f_values * np.exp(-1j * 2 * np.pi * n * t / T)
        c_n[i] = (1 / T) * np.trapz(integrand, t)

    return n_vals, c_n
```

(continues on next page)

(continued from previous page)

```
# Compute for the square wave
n_vals, c_n = complex_fourier_coefficients(square_wave, t_period, T, N_max)

# Plot amplitude spectrum
plt.figure(figsize=(6, 5))
plt.stem(n_vals, np.abs(c_n), linefmt='b-', markerfmt='bo', basefmt='k-')
plt.xlabel('Harmonic number n')
plt.ylabel('|c_n|')
plt.title('Complex Fourier Spectrum of Square Wave')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print('Note: the spectrum is symmetric — |c_n| = |c_{-n}| for real signals.')
print('This symmetry means negative frequencies are redundant (for real signals).')
```

```
/tmp/ipython-input-727021825.py:22: DeprecationWarning: `trapz` is deprecated. Use␣
↪`trapezoid` instead, or one of the numerical integration functions in `scipy.
↪integrate`.
  c_n[i] = (1 / T) * np.trapz(integrand, t)
```



```
Note: the spectrum is symmetric — |c_n| = |c_{-n}| for real signals.
This symmetry means negative frequencies are redundant (for real signals).
```

8.5 IV. From Fourier Series to Fourier Transform

The Fourier Series works for **periodic** functions. What about **non-periodic** functions?

Key idea: Let the period $T \to \infty$. Then:

- The discrete harmonics n/T become a continuous frequency f
- The sum becomes an integral
- The coefficients c_n become a continuous function $\hat{f}(\nu)$

8.5.1 The Fourier Transform Pair

Forward Transform (time → frequency):

$$\hat{f}(\nu) = \int_{-\infty}^{\infty} f(t)\, e^{-i2\pi\nu t}\, dt$$

Inverse Transform (frequency → time):

$$f(t) = \int_{-\infty}^{\infty} \hat{f}(\nu)\, e^{i2\pi\nu t}\, d\nu$$

8.5.2 Alternative convention (angular frequency $\omega = 2\pi\nu$)

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t)\, e^{-i\omega t}\, dt \qquad f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{f}(\omega)\, e^{i\omega t}\, d\omega$$

Warning: Different textbooks use different conventions for the 2π factor! Always check which convention is being used.

8.5.3 Example: Fourier Transform of a Gaussian

A Gaussian pulse: $f(t) = e^{-\alpha t^2}$

Its Fourier transform is also a Gaussian:

$$\hat{f}(\nu) = \sqrt{\frac{\pi}{\alpha}}\, e^{-\pi^2\nu^2/\alpha}$$

Key insight: A narrow pulse in time → a wide spectrum in frequency (and vice versa). This is the **uncertainty principle**!

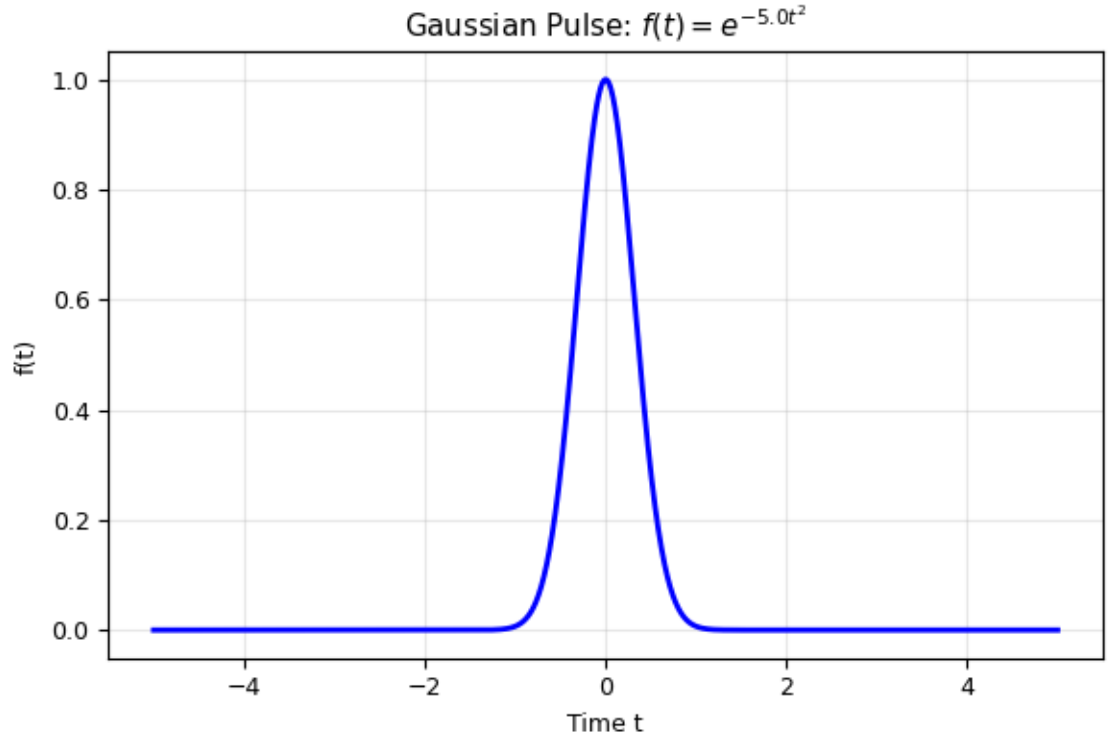
```python
# Fourier Transform of a Gaussian — numerical vs analytical
def numerical_fourier_transform(f_values, t, freqs):
    """
    Compute the continuous Fourier transform numerically
    using trapezoidal integration.

    Parameters:
```

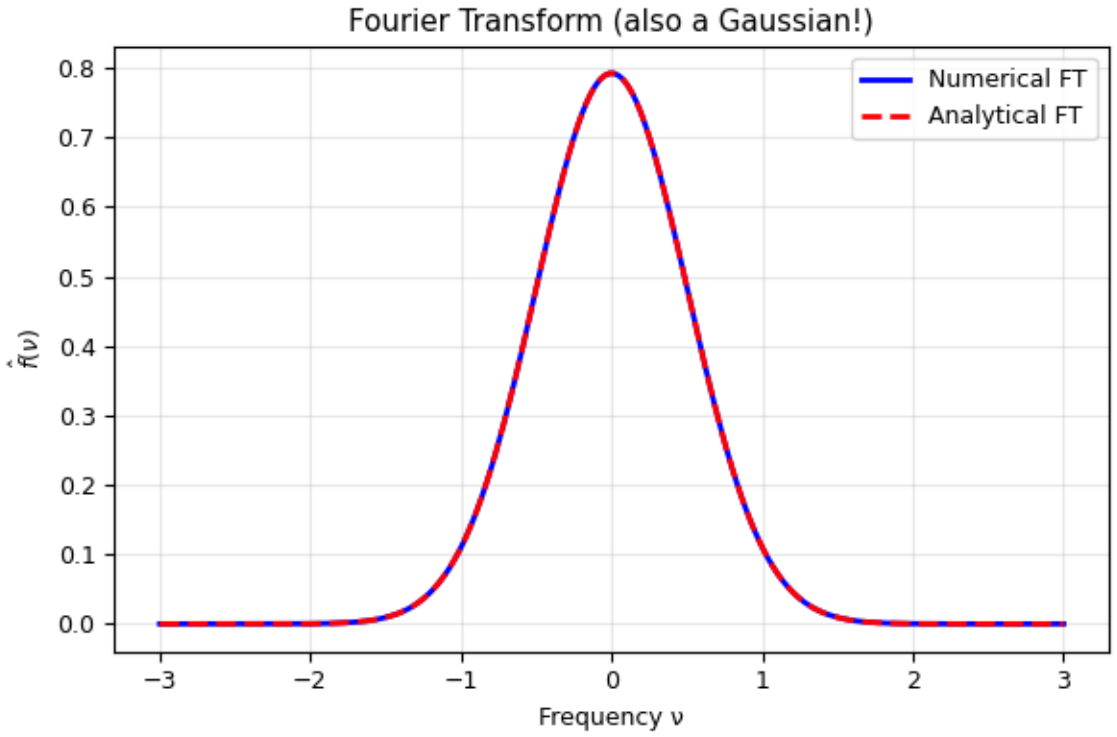
(continues on next page)

(continued from previous page)

```
        f_values: function values at time points t
        t: time array
        freqs: frequency array to evaluate at

    Returns:
        F: complex Fourier transform values
    """
    F = np.zeros(len(freqs), dtype=complex)
    for i, nu in enumerate(freqs):
        integrand = f_values * np.exp(-1j * 2 * np.pi * nu * t)
        F[i] = np.trapz(integrand, t)
    return F

# Gaussian pulse
alpha = 5.0
t_ft = np.linspace(-5, 5, 2000)
gaussian_pulse = np.exp(-alpha * t_ft**2)

# Numerical FT
freqs = np.linspace(-3, 3, 200)
F_numerical = numerical_fourier_transform(gaussian_pulse, t_ft, freqs)

# Analytical FT
F_analytical = np.sqrt(np.pi / alpha) * np.exp(-np.pi**2 * freqs**2 / alpha)

plt.figure(figsize=(6, 8))

plt.subplot(2, 1, 1)
plt.plot(t_ft, gaussian_pulse, 'b-', linewidth=2)
plt.xlabel('Time t')
plt.ylabel('f(t)')
plt.title(f'Gaussian Pulse: $f(t) = e^{{-{alpha}t^2}}$')
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
plt.plot(freqs, np.real(F_numerical), 'b-', linewidth=2, label='Numerical FT')
plt.plot(freqs, F_analytical, 'r--', linewidth=2, label='Analytical FT')
plt.xlabel('Frequency ν')
plt.ylabel('$\\hat{f}(\\nu)$')
plt.title('Fourier Transform (also a Gaussian!)')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f'Max error between numerical and analytical: {np.max(np.abs(np.real(F_
 ↪numerical) - F_analytical)):.6f}')
print('The Fourier transform of a Gaussian is a Gaussian!')
```

```
/tmp/ipython-input-3070628031.py:18: DeprecationWarning: `trapz` is deprecated.␣
 ↪Use `trapezoid` instead, or one of the numerical integration functions in `scipy.
 ↪integrate`.
  F[i] = np.trapz(integrand, t)
```





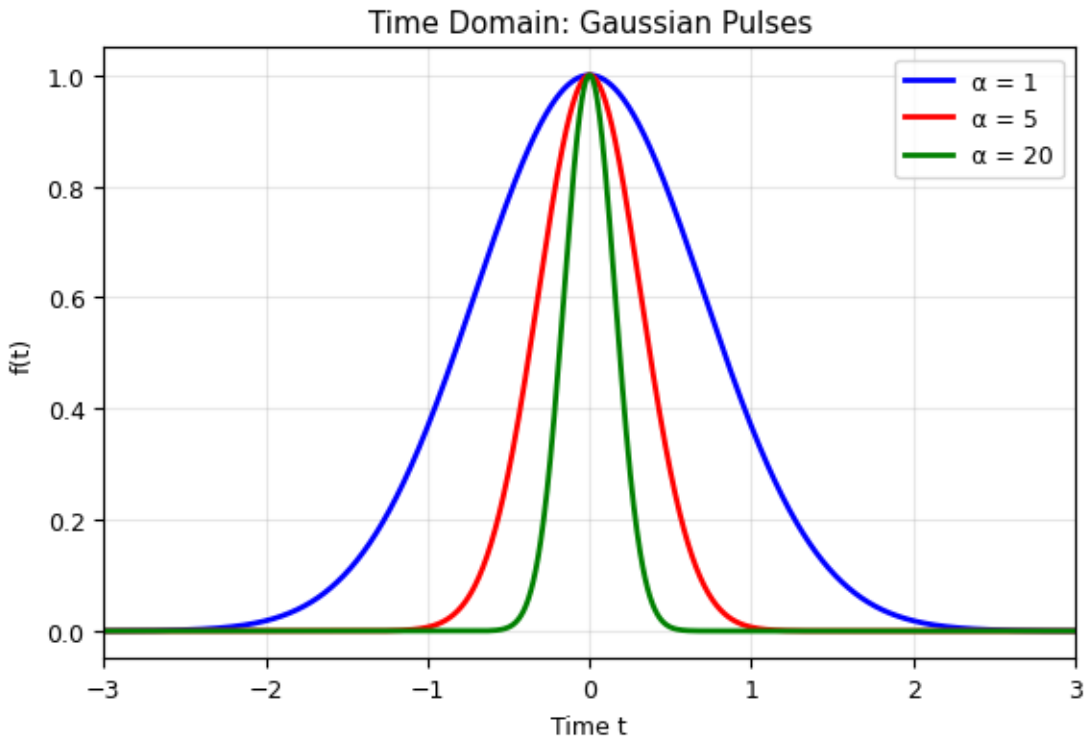

```
Max error between numerical and analytical: 0.000000
The Fourier transform of a Gaussian is a Gaussian!
```

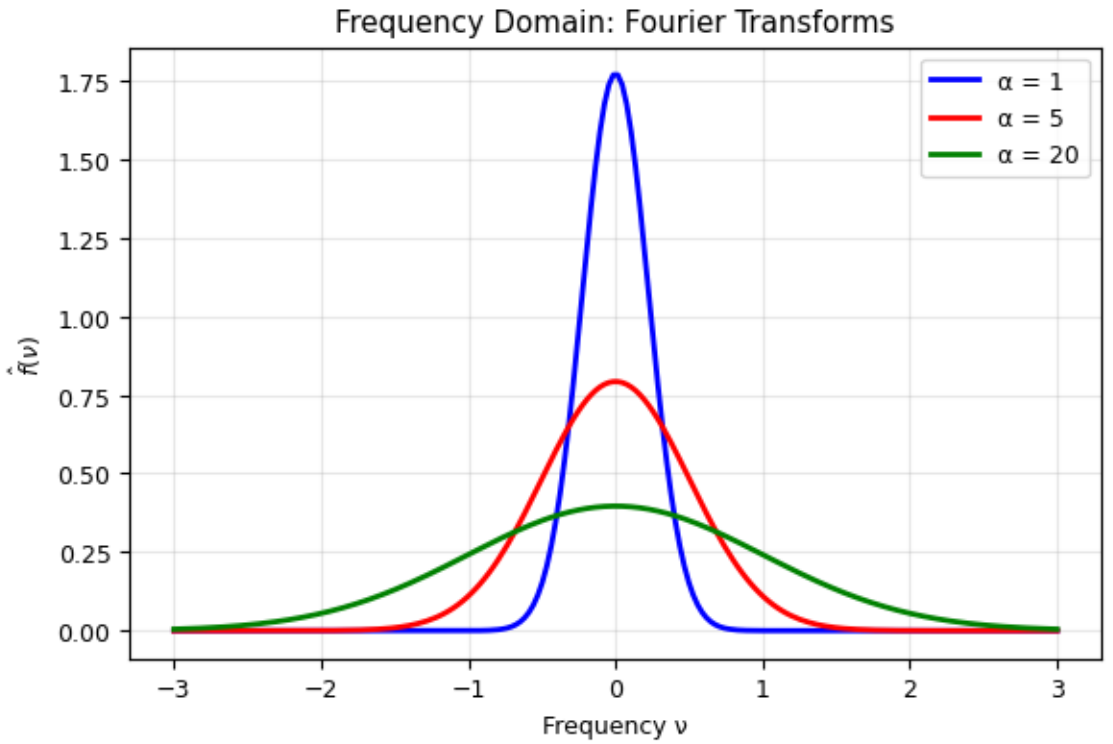
```
# The Uncertainty Principle: narrow in time ↔ wide in frequency
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(6, 8))

alphas = [1, 5, 20]
colors = ['b', 'r', 'g']

plt.subplot(2, 1, 1)
for alpha_val, c in zip(alphas, colors):
    pulse = np.exp(-alpha_val * t_ft**2)
    plt.plot(t_ft, pulse, c + '-', linewidth=2, label=f'α = {alpha_val}')
plt.xlabel('Time t')
plt.ylabel('f(t)')
plt.title('Time Domain: Gaussian Pulses')
plt.legend()
plt.grid(True, alpha=0.3)
plt.xlim(-3, 3)

plt.subplot(2, 1, 2)
for alpha_val, c in zip(alphas, colors):
    F_anal = np.sqrt(np.pi / alpha_val) * np.exp(-np.pi**2 * freqs**2 / alpha_val)
    plt.plot(freqs, F_anal, c + '-', linewidth=2, label=f'α = {alpha_val}')
plt.xlabel('Frequency ν')
plt.ylabel('$\\hat{f}(\\nu)$')
plt.title('Frequency Domain: Fourier Transforms')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print('Narrow in time (large α) → Wide in frequency')
print('Wide in time (small α) → Narrow in frequency')
print('\nThis is the Fourier uncertainty principle: Δt · Δν ≥ 1/(4π)')
print('(closely related to the Heisenberg uncertainty principle in quantum mechanics!)
 ↪')
```






```
Narrow in time (large α) → Wide in frequency
Wide in time (small α) → Narrow in frequency

This is the Fourier uncertainty principle: Δt · Δν ≥ 1/(4π)
(closely related to the Heisenberg uncertainty principle in quantum mechanics!)
```

8.6 V. The Discrete Fourier Transform (DFT)

In practice, we work with **sampled** (discrete) data, not continuous functions. Given N samples:

$$f_0, f_1, f_2, \cdots, f_{N-1}$$

sampled at times $t_k = k\Delta t$ where Δt is the sampling interval.

8.6.1 The DFT Formula

$$F_n = \sum_{k=0}^{N-1} f_k \, e^{-i2\pi nk/N} \qquad n = 0, 1, \ldots, N-1$$

8.6.2 The Inverse DFT

$$f_k = \frac{1}{N} \sum_{n=0}^{N-1} F_n \, e^{i2\pi nk/N} \qquad k = 0, 1, \ldots, N-1$$

8.6.3 Key Parameters

Parameter	Formula	Meaning
Sampling rate	$f_s = 1/\Delta t$	Samples per second
Frequency resolution	$\Delta\nu = f_s/N = 1/(N\Delta t)$	Smallest detectable frequency difference
Nyquist frequency	$\nu_{\text{max}} = f_s/2$	Highest frequency we can detect
Frequency bins	$\nu_n = n \cdot \Delta\nu$	The discrete frequencies

8.6.4 Computational cost

The DFT as written above requires $O(N^2)$ operations (N frequencies × N time points). The **FFT** algorithm reduces this to $O(N \log N)$ — that's for the next lecture!

```
# Hand-implement the DFT!
def dft(f):
    """
    Compute the Discrete Fourier Transform from scratch.

    Parameters:
        f: array of N sample values

    Returns:
        F: array of N complex DFT coefficients

    Formula: F_n = sum_{k=0}^{N-1} f_k * exp(-i*2*pi*n*k/N)
    """
    N = len(f)
    F = np.zeros(N, dtype=complex)
    for n in range(N):
```

(continues on next page)

(continued from previous page)

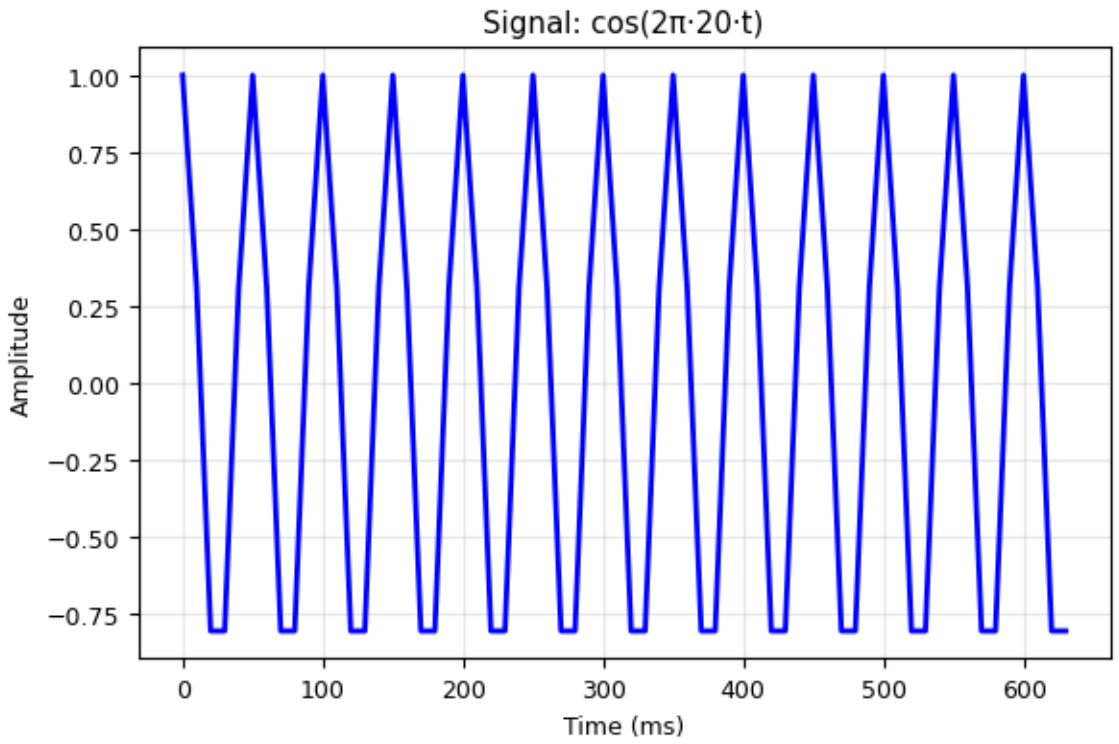
```
        for k in range(N):
            F[n] += f[k] * np.exp(-1j * 2 * np.pi * n * k / N)

    return F

def idft(F):
    """
    Compute the Inverse DFT from scratch.

    Parameters:
        F: array of N complex DFT coefficients

    Returns:
        f: array of N sample values (reconstructed)
    """
    N = len(F)
    f = np.zeros(N, dtype=complex)


    # Formula: f_k = (1/N) * sum_{n=0}^{N-1} F_n * exp(i*2*pi*n*k/N)
    for k in range(N):
        for n in range(N):
            f[k] += F[n] * np.exp(1j * 2 * np.pi * n * k/N)

    return f / N

# Test: DFT of a simple cosine
N = 64
dt = 0.01  # sampling interval (s)
t_dft = np.arange(N) * dt

# Signal: a 20 Hz cosine
freq_signal = 20  # Hz
signal = np.cos(2 * np.pi * freq_signal * t_dft)

# Compute DFT
F = dft(signal)

# Verify against numpy's FFT
F_numpy = np.fft.fft(signal)

print(f'Max difference between our DFT and np.fft.fft: {np.max(np.abs(F - F_numpy)):.
 ↪2e}')
print('Our implementation matches numpy!')

# Frequency axis
freqs_dft = np.arange(N) / (N * dt)  # Hz

plt.figure(figsize=(6, 8))

plt.subplot(2, 1, 1)
plt.plot(t_dft * 1000, signal, 'b-', linewidth=2)
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude')
plt.title(f'Signal: cos(2π·{freq_signal}·t)')
plt.grid(True, alpha=0.3)
```

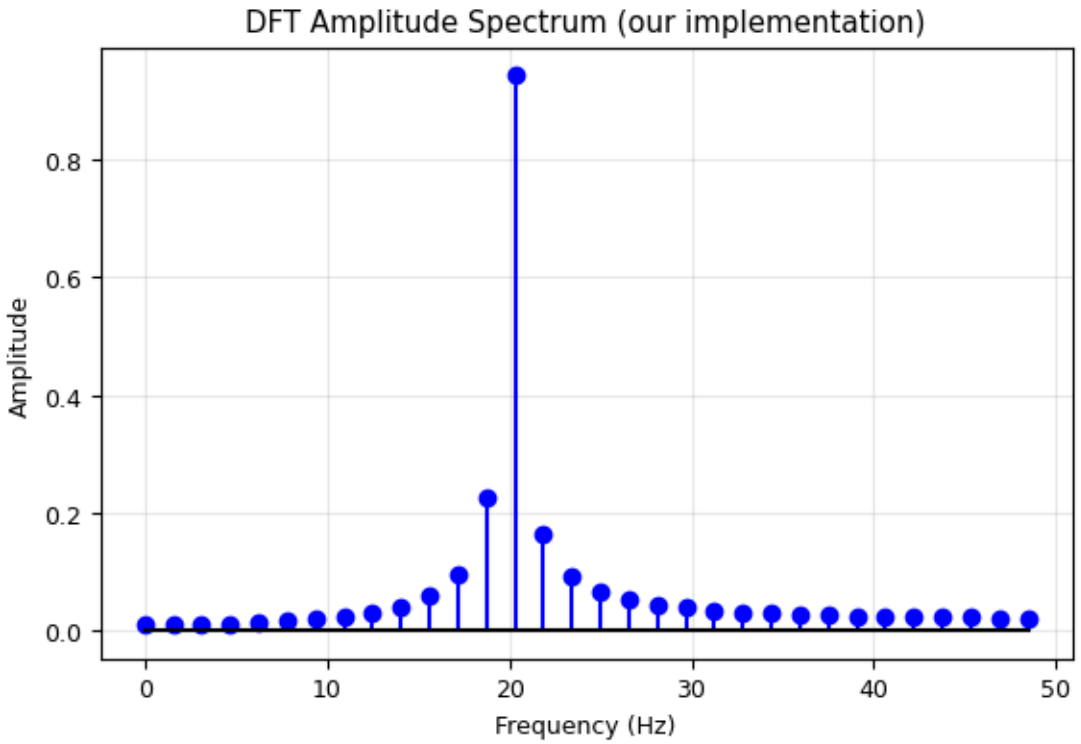
(continues on next page)

(continued from previous page)

```
plt.subplot(2, 1, 2)
plt.stem(freqs_dft[:N//2], np.abs(F[:N//2]) / N * 2, linefmt='b-', markerfmt='bo',␣
↪basefmt='k-')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude')
plt.title('DFT Amplitude Spectrum (our implementation)')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f'\nThe peak is at {freqs_dft[np.argmax(np.abs(F[:N//2]))]} Hz — correct!')
```

```
Max difference between our DFT and np.fft.fft: 3.32e-13
Our implementation matches numpy!
```

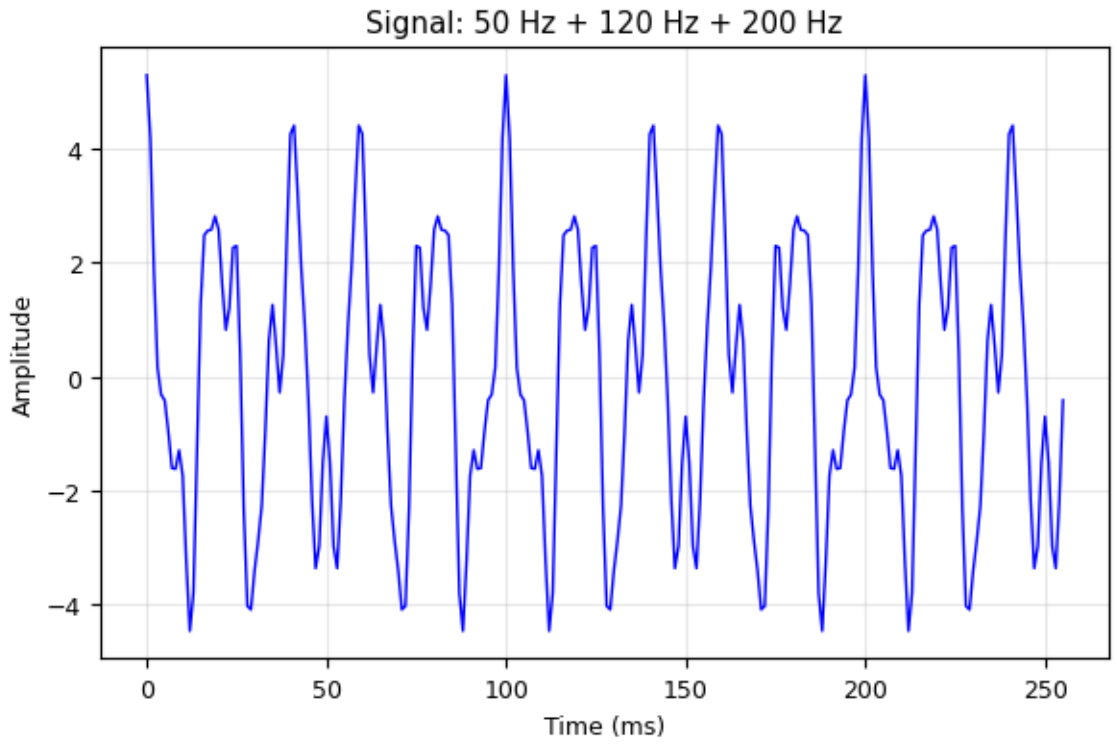


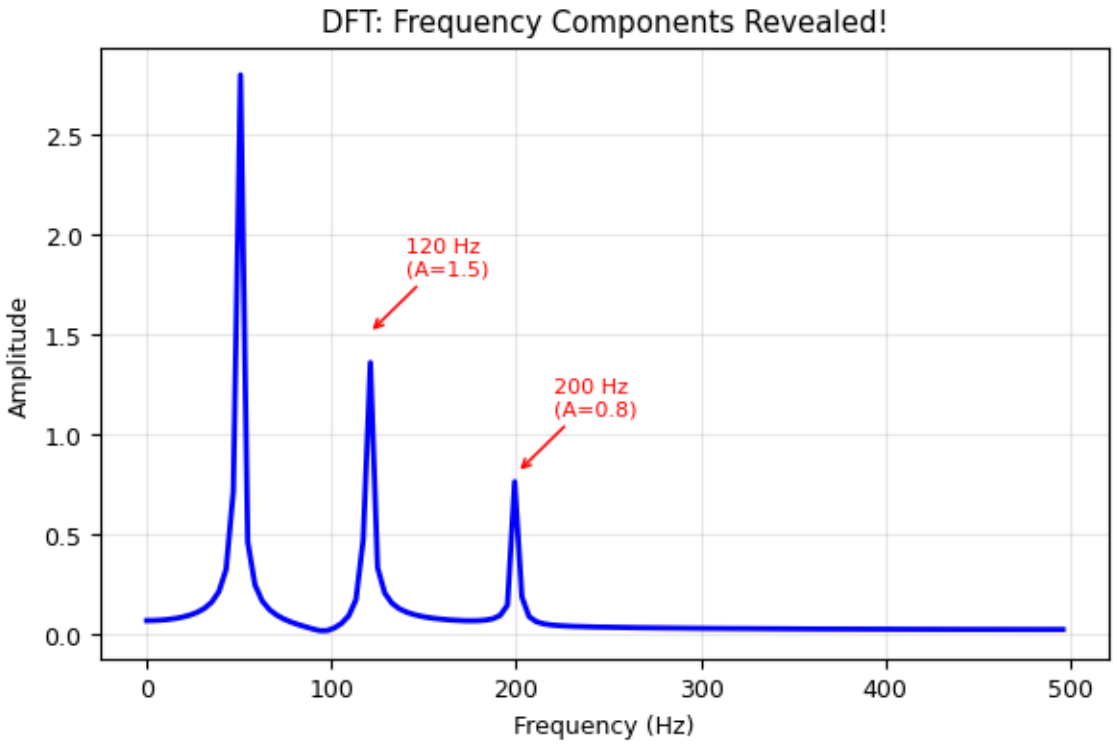



```
The peak is at 20.3125 Hz — correct!
```

8.6.5 DFT of a Multi-Frequency Signal

```
# Multi-frequency signal
N = 256
dt = 0.001  # 1 kHz sampling
t_multi = np.arange(N) * dt

# Three frequencies with different amplitudes
signal_multi = (3.0 * np.cos(2 * np.pi * 50 * t_multi) +    # 50 Hz, amplitude 3
                1.5 * np.cos(2 * np.pi * 120 * t_multi) +   # 120 Hz, amplitude 1.5
                0.8 * np.cos(2 * np.pi * 200 * t_multi))    # 200 Hz, amplitude 0.8

# Use our DFT (slow but educational!)
# For N=256, this takes a moment...
F_multi = dft(signal_multi)

# Frequency axis
freqs_multi = np.arange(N) / (N * dt)

plt.figure(figsize=(6, 8))

plt.subplot(2, 1, 1)
plt.plot(t_multi * 1000, signal_multi, 'b-', linewidth=1)
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude')
plt.title('Signal: 50 Hz + 120 Hz + 200 Hz')
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
amplitudes = np.abs(F_multi[:N//2]) / N * 2
plt.plot(freqs_multi[:N//2], amplitudes, 'b-', linewidth=2)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude')
plt.title('DFT: Frequency Components Revealed!')
plt.grid(True, alpha=0.3)

# Mark the peaks
for freq, amp in [(50, 3.0), (120, 1.5), (200, 0.8)]:
    plt.annotate(f'{freq} Hz\n(A={amp})', xy=(freq, amp),
                 xytext=(freq + 20, amp + 0.3),
                 arrowprops=dict(arrowstyle='->', color='red'),
                 fontsize=8, color='red')

plt.tight_layout()
plt.show()

print('The DFT perfectly identifies all three frequency components!')
print('The amplitudes match too: 3.0, 1.5, and 0.8')
```

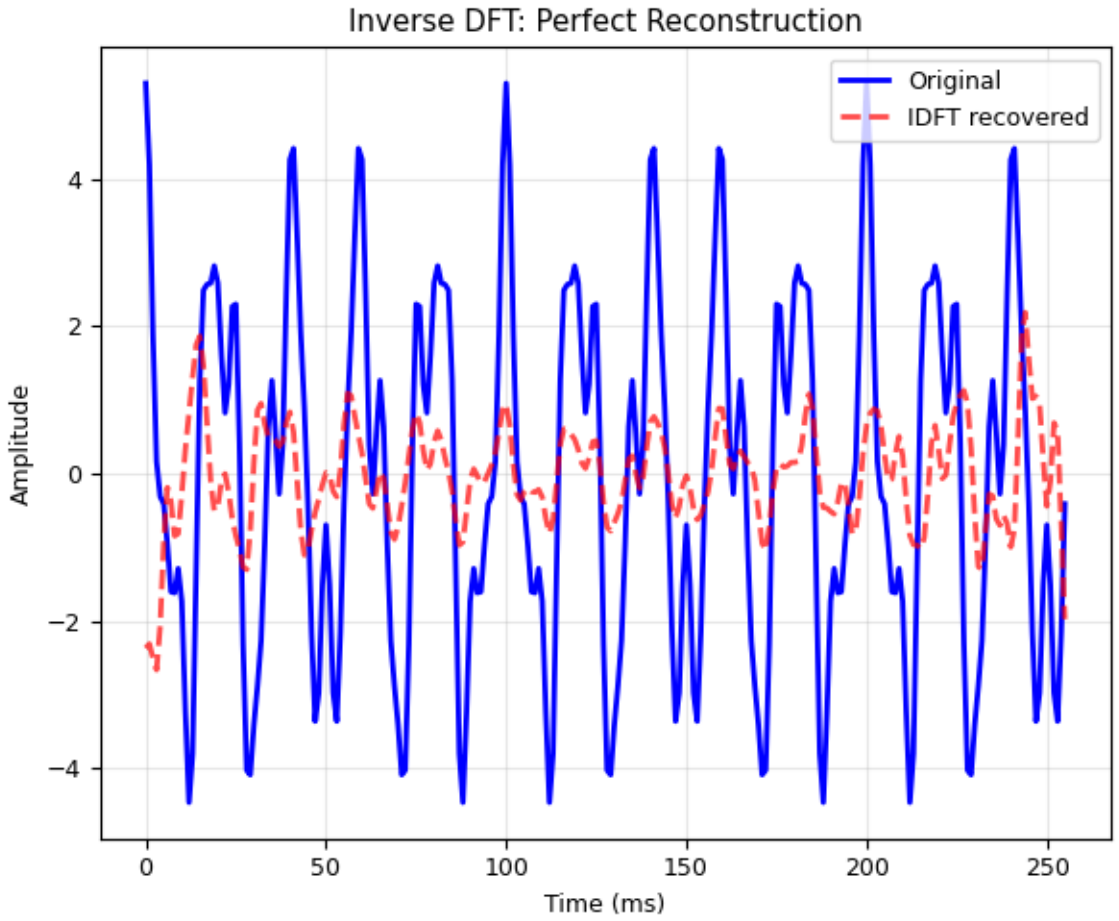





```
The DFT perfectly identifies all three frequency components!
The amplitudes match too: 3.0, 1.5, and 0.8
```

8.6.6 Verify: Inverse DFT recovers the signal

```
# Verify that IDFT recovers the original signal
# print (F_multi)
for i in range(len(F_multi)):
    if F_multi[i] > 2.0:
        F_multi[i] = 0.0
signal_recovered = idft(F_multi)

plt.figure(figsize=(6, 5))
plt.plot(t_multi * 1000, signal_multi, 'b-', linewidth=2, label='Original')
plt.plot(t_multi * 1000, np.real(signal_recovered), 'r--', linewidth=2,
         alpha=0.7, label='IDFT recovered')
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude')
plt.title('Inverse DFT: Perfect Reconstruction')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

max_err = np.max(np.abs(signal_multi - np.real(signal_recovered)))
print(f'Max reconstruction error: {max_err:.2e}')
print('The DFT is perfectly invertible (up to floating point precision)!')
```



```
Max reconstruction error: 7.67e+00
The DFT is perfectly invertible (up to floating point precision)!
```

8.6.7 The Physical interpretation of FT

How do we understand the outcome of Fourier transform?

The Fourier transform breaks a function down into a set of real or complex sinusoidal waves. Each term in a sum represents one wave with its own well-defined frequency. If the function $f(x)$ is a function in space then we have spatial frequencies; say like musical notes. Saying that any function can be expressed as a sume of waves of given frequencies, and the Fourier transform tells us what that sum is for any particular function. The coefficients of the transform tell us exactly how much of each frequency we have in the sum.

Thus, by looking at the output of our Fourier transform, we can get a picture of what the frequency breakdown of a signal is. For example, consider the signal shown above. As we see, the signal consists of three basic waves that go up and down with different frequency. But there is all some noise in the data as well, visiable as smaller wiggles in the line. If one were to listen to this signal as sound one would hear a constant note at the frequency of the main wave, accompanied by a background hiss that comes from the noise.

Since the coefficients retured by the transform in the array c are in general complex, which could be expressed by the absolute values and angles. The absoulte values give us a measure of the amplitude of each waves and the angle gives the phase of each Fourier series. The advantage of Fourier transform is to conveniently capture the waves with significant contributions to the measurement and the noise will only appear as a uniform random background.

8.6.8 Parseval's Theorem: Energy Conservation

```python
# Parseval's theorem: energy in time domain = energy in frequency domain
# For DFT: sum |f_k|^2 = (1/N) * sum |F_n|^2

energy_time = np.sum(np.abs(signal_multi)**2)
energy_freq = np.sum(np.abs(F_multi)**2) / N

print("Parseval's Theorem Verification")
print('=' * 40)
print(f'Energy in time domain:      {energy_time:.4f}')
print(f'Energy in frequency domain:  {energy_freq:.4f}')
print(f'Ratio: {energy_freq / energy_time:.6f}')
print('\nEnergy is conserved between domains!')
```

```
Parseval's Theorem Verification
========================================
Energy in time domain:      1527.9145
Energy in frequency domain:  134.3685
Ratio: 0.087942

Energy is conserved between domains!
```

8.7 VI. Physics Applications

8.7.1 1. Frequency Analysis of a Noisy Signal

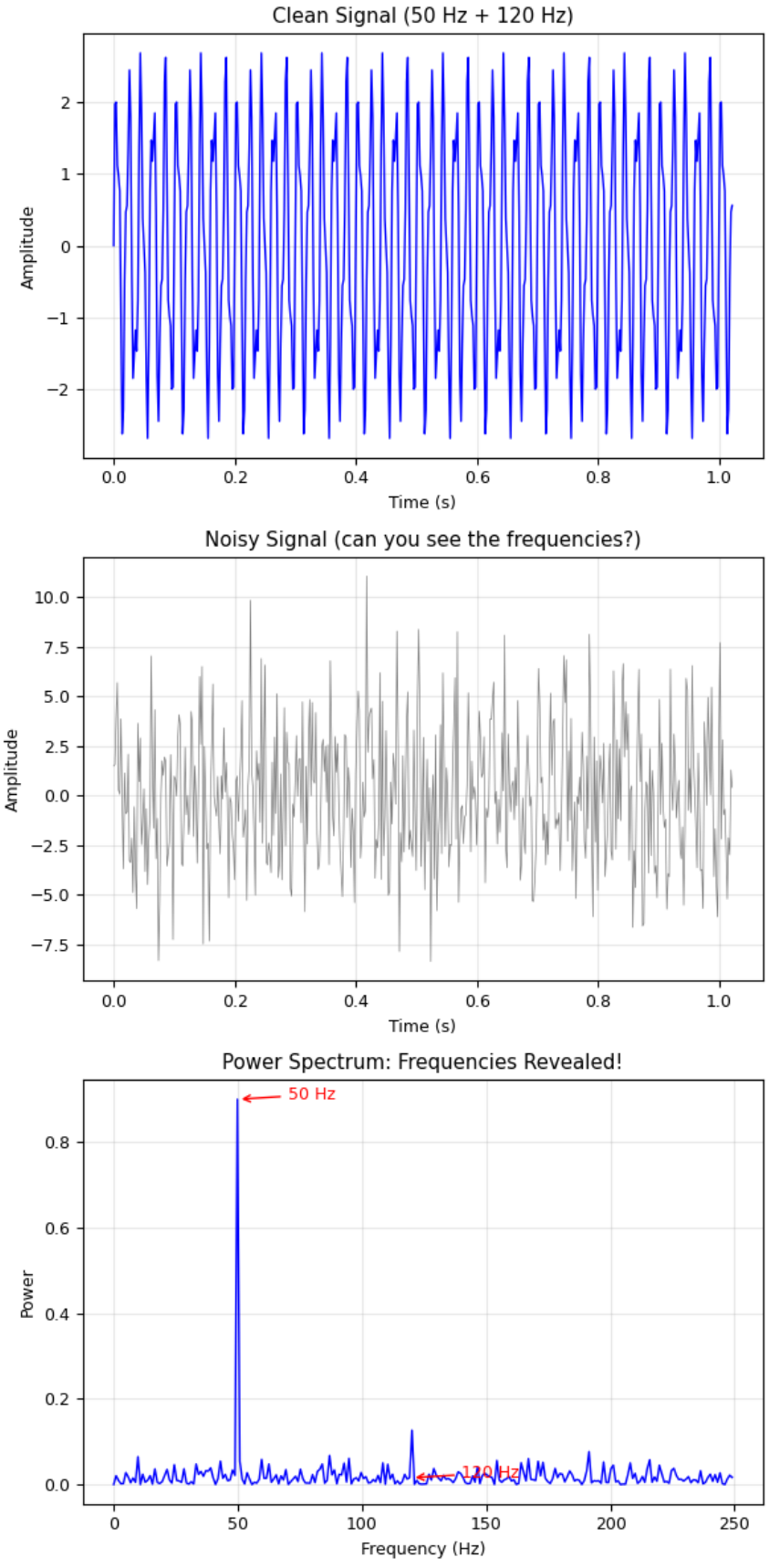
A common task: given a noisy measurement, find the hidden frequencies.

```
# Hidden frequencies in noise
np.random.seed(42)
N = 512
fs = 500  # Hz
dt = 1 / fs
t_noisy = np.arange(N) * dt

# Hidden signal: two frequencies buried in noise
clean_signal = (2.0 * np.sin(2 * np.pi * 50 * t_noisy) +
                0.8 * np.sin(2 * np.pi * 120 * t_noisy))
noise = 3.0 * np.random.randn(N)
noisy_signal = clean_signal + noise

# DFT (using numpy for speed)
F_noisy = np.fft.fft(noisy_signal)
freqs_noisy = np.fft.fftfreq(N, dt)

# Power spectrum
power = np.abs(F_noisy[:N//2])**2 / N**2

plt.figure(figsize=(6, 12))

plt.subplot(3, 1, 1)
plt.plot(t_noisy, clean_signal, 'b-', linewidth=1)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Clean Signal (50 Hz + 120 Hz)')
plt.grid(True, alpha=0.3)

plt.subplot(3, 1, 2)
plt.plot(t_noisy, noisy_signal, 'gray', linewidth=0.5)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Noisy Signal (can you see the frequencies?)')
plt.grid(True, alpha=0.3)

plt.subplot(3, 1, 3)
plt.plot(freqs_noisy[:N//2], power, 'b-', linewidth=1)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power')
plt.title('Power Spectrum: Frequencies Revealed!')
plt.grid(True, alpha=0.3)
plt.annotate('50 Hz', xy=(50, power[int(50*N/fs)]),
             xytext=(70, power[int(50*N/fs)]),
             arrowprops=dict(arrowstyle='->', color='red'),
             fontsize=9, color='red')
plt.annotate('120 Hz', xy=(120, power[int(120*N/fs)]),
             xytext=(140, power[int(120*N/fs)]),
             arrowprops=dict(arrowstyle='->', color='red'),
             fontsize=9, color='red')
```

(continues on next page)

(continued from previous page)

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

print('Even though the noise is LARGER than the signal,',)
print('the Fourier transform clearly identifies both frequencies!')
```

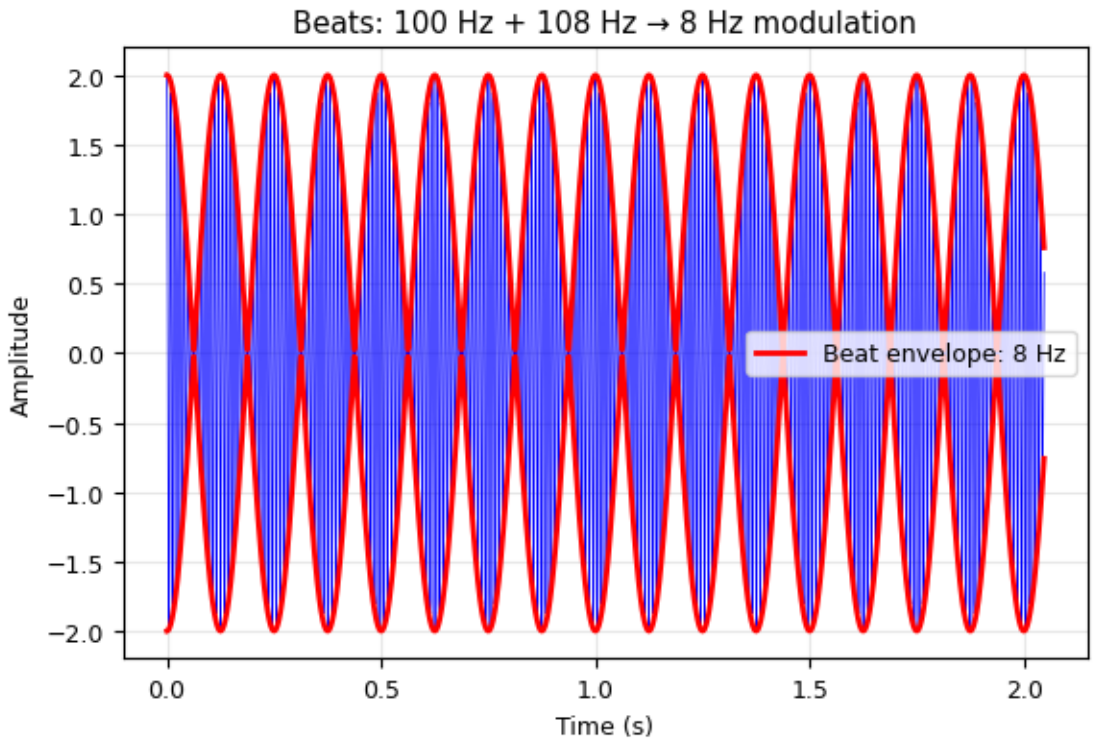

Clean Signal (50 Hz + 120 Hz)
Amplitude
Time (s)
Noisy Signal (can you see the frequencies?)
Amplitude
Time (s)
Power Spectrum: Frequencies Revealed!
50 Hz
120 Hz
Power
Frequency (Hz)

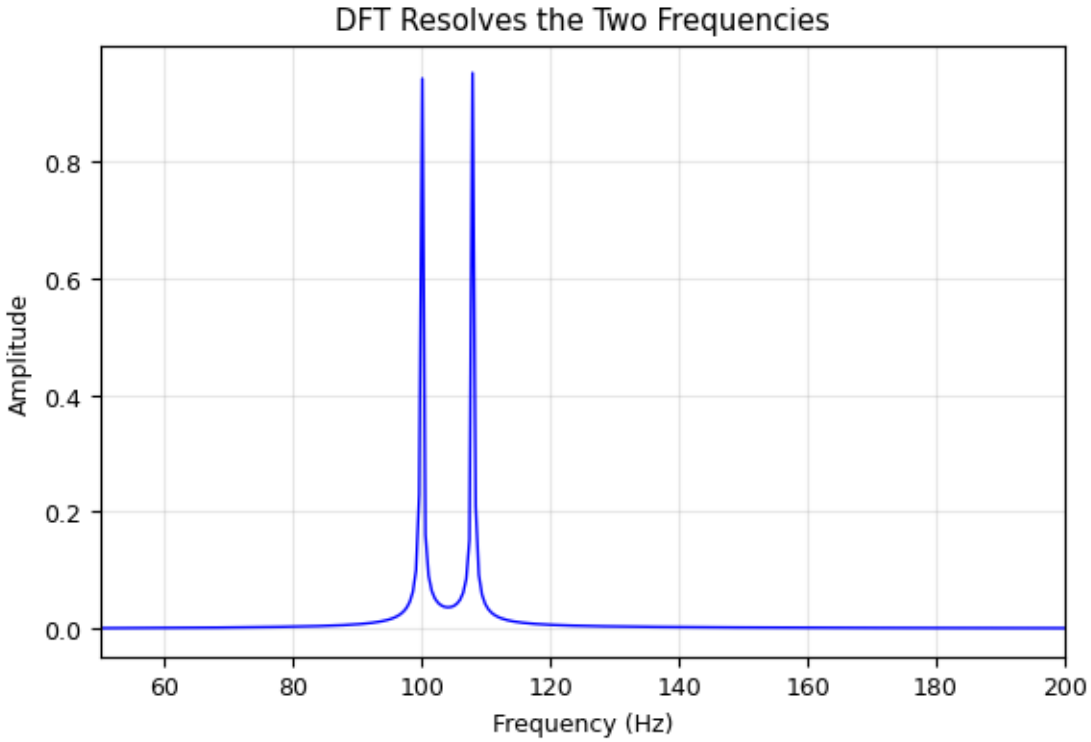
```
Even though the noise is LARGER than the signal,
the Fourier transform clearly identifies both frequencies!
```

8.7.2 2. Beats: Superposition of Close Frequencies

When two close frequencies ν_1 and ν_2 are added, we hear **beats** at frequency $|\nu_1 - \nu_2|$.

```
# Beats: two close frequencies
N = 2048
fs = 1000  # Hz
dt = 1 / fs
t_beat = np.arange(N) * dt

f1, f2 = 100, 108  # Close frequencies (Hz)
beat_signal = np.cos(2 * np.pi * f1 * t_beat) + np.cos(2 * np.pi * f2 * t_beat)

# DFT
F_beat = np.fft.fft(beat_signal)
freqs_beat = np.fft.fftfreq(N, dt)

plt.figure(figsize=(6, 8))

plt.subplot(2, 1, 1)
plt.plot(t_beat, beat_signal, 'b-', linewidth=0.5)
# Envelope
envelope = 2 * np.abs(np.cos(np.pi * (f2 - f1) * t_beat))
plt.plot(t_beat, envelope, 'r-', linewidth=2, label=f'Beat envelope: {f2-f1} Hz')
plt.plot(t_beat, -envelope, 'r-', linewidth=2)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title(f'Beats: {f1} Hz + {f2} Hz → {f2-f1} Hz modulation')
plt.legend()
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
amplitudes_beat = np.abs(F_beat[:N//2]) / N * 2
plt.plot(freqs_beat[:N//2], amplitudes_beat, 'b-', linewidth=1)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude')
plt.title('DFT Resolves the Two Frequencies')
plt.xlim(50, 200)
plt.grid(True, alpha=0.3)
plt.annotate(f'{f1} Hz', xy=(f1, 1.0), xytext=(f1-20, 1.3),
             arrowprops=dict(arrowstyle='->', color='red'),
             fontsize=9, color='red')
plt.annotate(f'{f2} Hz', xy=(f2, 1.0), xytext=(f2+10, 1.3),
             arrowprops=dict(arrowstyle='->', color='red'),
             fontsize=9, color='red')

plt.tight_layout()
plt.show()

print(f'Two frequencies {f1} Hz and {f2} Hz create beats at {f2-f1} Hz.')
print('Our ears hear the slow modulation — this is how you tune a guitar!')
```






```
Two frequencies 100 Hz and 108 Hz create beats at 8 Hz.
Our ears hear the slow modulation — this is how you tune a guitar!
```

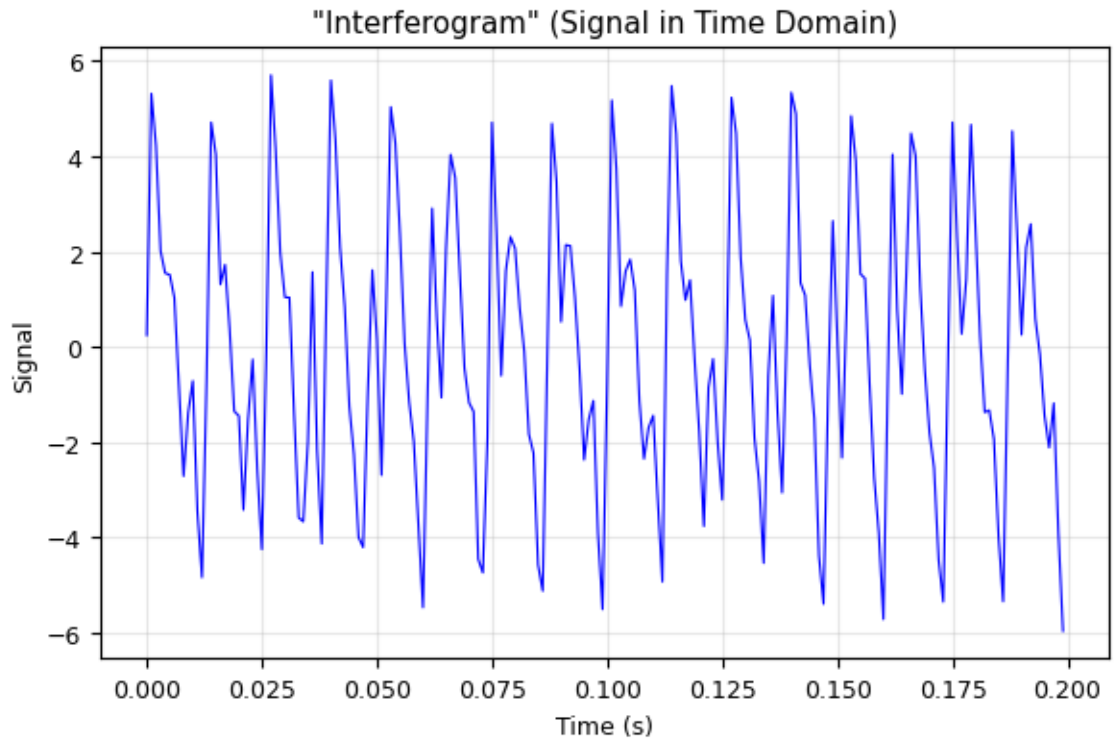
8.7.3 3. Spectroscopy: Identifying Spectral Lines

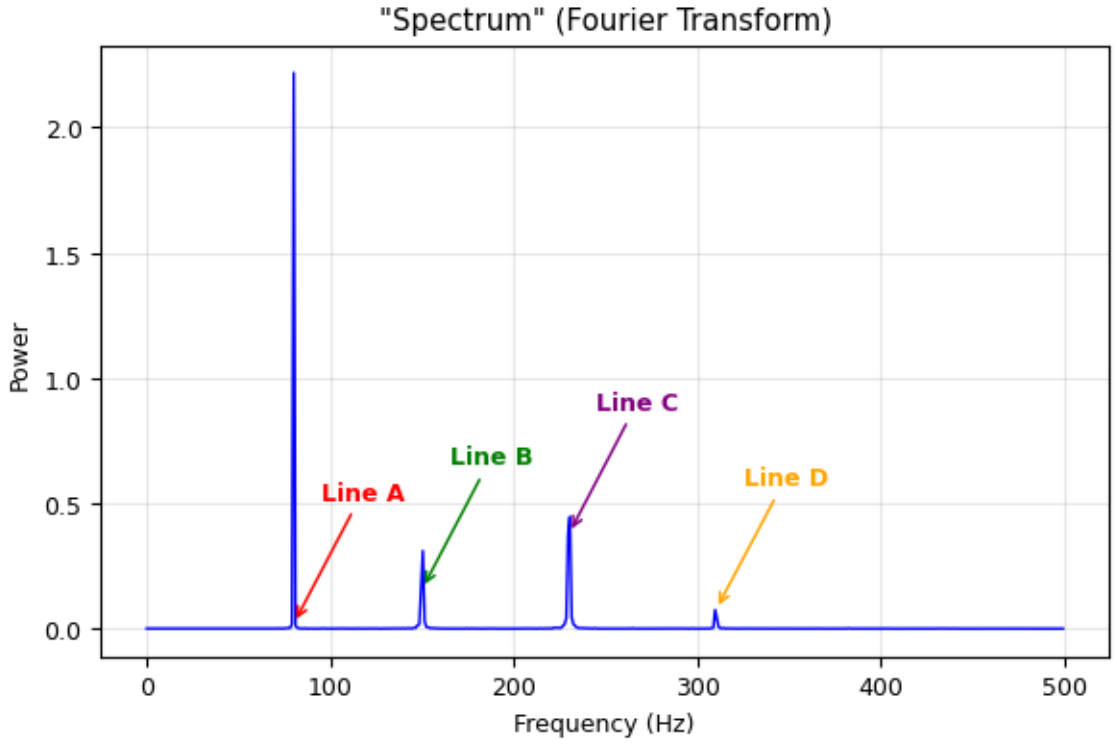
```
# Simulate spectroscopy: signal with multiple frequency components + noise
# (analogy to atomic emission/absorption lines)
np.random.seed(42)
N = 1024
fs = 1000
dt = 1 / fs
t_spec = np.arange(N) * dt

# "Emission lines" at specific frequencies
spectral_lines = {
    'Line A': (80, 3.0),    # frequency (Hz), amplitude
    'Line B': (150, 1.5),
    'Line C': (230, 2.0),
    'Line D': (310, 0.8),
}

signal_spec = np.zeros(N)
for name, (freq, amp) in spectral_lines.items():
    signal_spec += amp * np.sin(2 * np.pi * freq * t_spec)

# Add noise
signal_spec += 0.5 * np.random.randn(N)

# DFT
F_spec = np.fft.fft(signal_spec)
freqs_spec = np.fft.fftfreq(N, dt)
power_spec = np.abs(F_spec[:N//2])**2 / N**2

plt.figure(figsize=(6, 8))

plt.subplot(2, 1, 1)
plt.plot(t_spec[:200], signal_spec[:200], 'b-', linewidth=0.8)
plt.xlabel('Time (s)')
plt.ylabel('Signal')
plt.title('"Interferogram" (Signal in Time Domain)')
plt.grid(True, alpha=0.3)

plt.subplot(2, 1, 2)
plt.plot(freqs_spec[:N//2], power_spec, 'b-', linewidth=1)

# Label the peaks
colors_spec = ['red', 'green', 'purple', 'orange']
for (name, (freq, amp)), c in zip(spectral_lines.items(), colors_spec):
    idx = int(freq * N / fs)
    plt.annotate(name, xy=(freq, power_spec[idx]),
                 xytext=(freq + 15, power_spec[idx] + 0.5),
                 arrowprops=dict(arrowstyle='->', color=c),
                 fontsize=9, color=c, fontweight='bold')

plt.xlabel('Frequency (Hz)')
plt.ylabel('Power')
plt.title('"Spectrum" (Fourier Transform)')
plt.grid(True, alpha=0.3)

plt.tight_layout()
```

(continues on next page)

(continued from previous page)

```
plt.show()

print('This is exactly how Fourier Transform Infrared Spectroscopy (FTIR) works!')
print('1. Measure a signal in time (interferogram)')
print('2. Fourier transform → frequency spectrum')
print('3. Identify spectral lines → identify the material')
```






```
This is exactly how Fourier Transform Infrared Spectroscopy (FTIR) works!
1. Measure a signal in time (interferogram)
2. Fourier transform → frequency spectrum
3. Identify spectral lines → identify the material
```

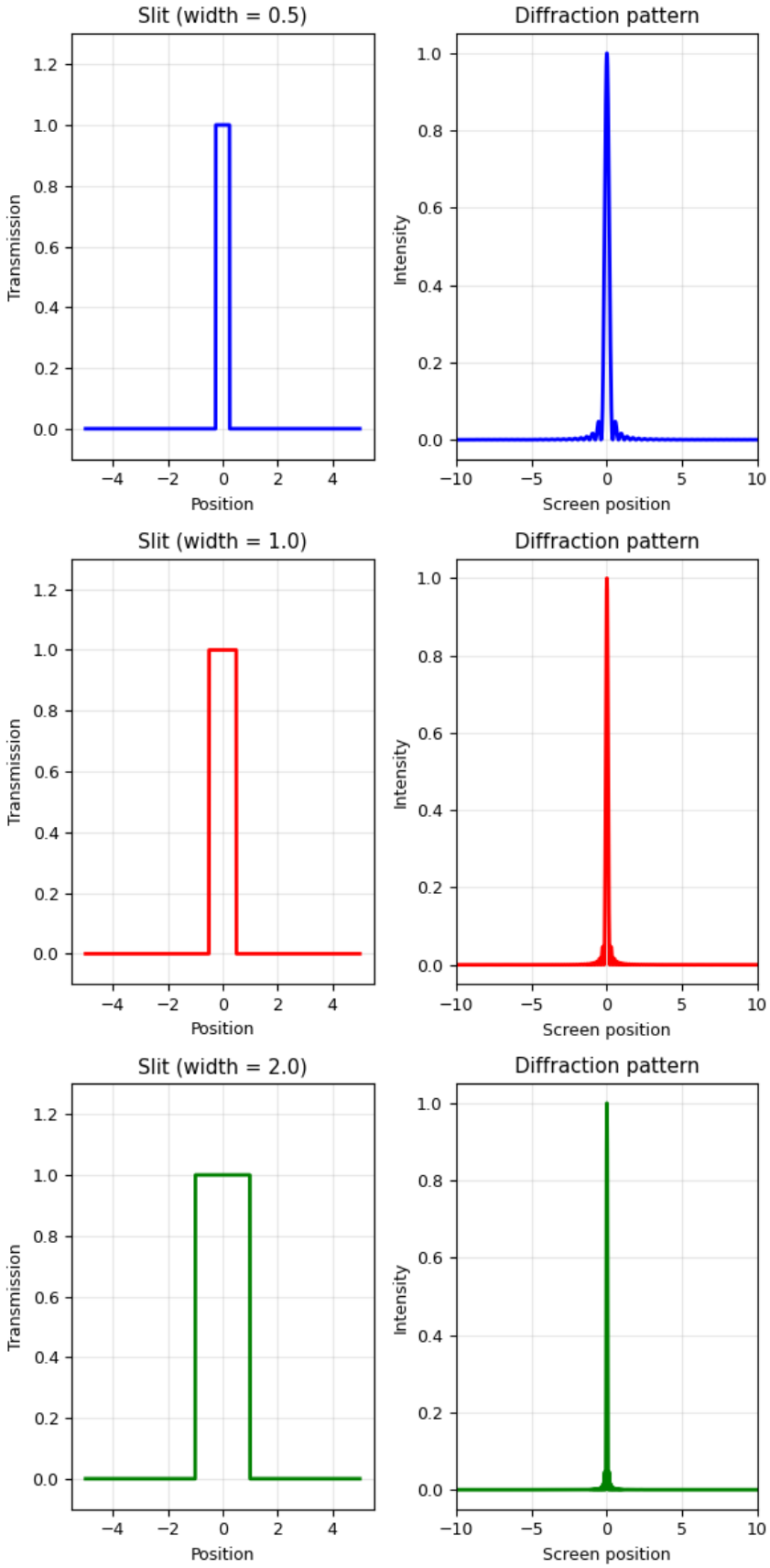
8.7.4 4. Diffraction: Single Slit

The diffraction pattern from a single slit is the **Fourier transform** of the slit aperture function!

For a slit of width a: the aperture is a rectangular function, and its Fourier transform is a **sinc** function:

$$I(\theta) \propto \text{sinc}^2\left(\frac{\pi a \sin\theta}{\lambda}\right)$$

```
# Single-slit diffraction as a Fourier Transform
N = 1024

# Aperture function: rectangular slit
x = np.linspace(-5, 5, N)

plt.figure(figsize=(6, 12))

slit_widths = [0.5, 1.0, 2.0]
colors = ['b', 'r', 'g']

for idx, (width, c) in enumerate(zip(slit_widths, colors)):
    # Rectangular aperture
    aperture = np.zeros(N)
    aperture[np.abs(x) < width / 2] = 1.0

    # Fourier transform → diffraction pattern
    F_diff = np.fft.fftshift(np.fft.fft(np.fft.fftshift(aperture)))
    intensity = np.abs(F_diff)**2
    intensity /= np.max(intensity)  # Normalize

    # Plot aperture
    plt.subplot(3, 2, 2 * idx + 1)
    plt.plot(x, aperture, c + '-', linewidth=2)
    plt.xlabel('Position')
    plt.ylabel('Transmission')
    plt.title(f'Slit (width = {width})')
    plt.ylim(-0.1, 1.3)
    plt.grid(True, alpha=0.3)

    # Plot diffraction pattern
    screen = np.linspace(-10, 10, N)
    plt.subplot(3, 2, 2 * idx + 2)
    plt.plot(screen, intensity, c + '-', linewidth=2)
    plt.xlabel('Screen position')
    plt.ylabel('Intensity')
    plt.title(f'Diffraction pattern')
    plt.xlim(-10, 10)
    plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print('Wider slit → narrower diffraction pattern (and vice versa).')
print('This is the uncertainty principle again: Δx · Δk ≥ 1/2')
```


Slit (width = 0.5)
Diffraction pattern
Transmission
Intensity
Position
Screen position
Slit (width = 1.0)
Diffraction pattern
Transmission
Intensity
Position
Screen position
Slit (width = 2.0)
Diffraction pattern
Transmission
Intensity
Position
Screen position

```
Wider slit → narrower diffraction pattern (and vice versa).
This is the uncertainty principle again: Δx · Δk ≥ 1/2
```

CHAPTER

NINE

LECTURE 09: THE FAST FOURIER TRANSFORM (FFT)

Computational Physics — Spring 2026

9.1 Recap from Lecture 08

We implemented the **Discrete Fourier Transform (DFT)** from scratch:

$$F_n = \sum_{k=0}^{N-1} f_k \, e^{-2\pi i \, nk/N}$$

This required **two nested loops** → $O(N^2)$ operations.

9.1.1 Today's Goal

The **Fast Fourier Transform (FFT)** computes the *exact same* DFT, but in $O(N \log N)$ operations.

	DFT (Lecture 08)	FFT (Today)
Result	F_n	Same F_n
Complexity	$O(N^2)$	$O(N \log N)$
N = 1024	~1,000,000 ops	~10,000 ops
N = 10^6	~10^{12} ops (~hours)	~2×10^7 ops (~seconds)

The FFT is often called **one of the most important algorithms of the 20th century** (Cooley & Tukey, 1965).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
import time

# Projector-friendly settings
plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

print("Ready!")
```

```
Ready!
```

9.2 I. Why Do We Need FFT?

Our hand-coded DFT from Lecture 08 works, but it's **painfully slow** for large N.

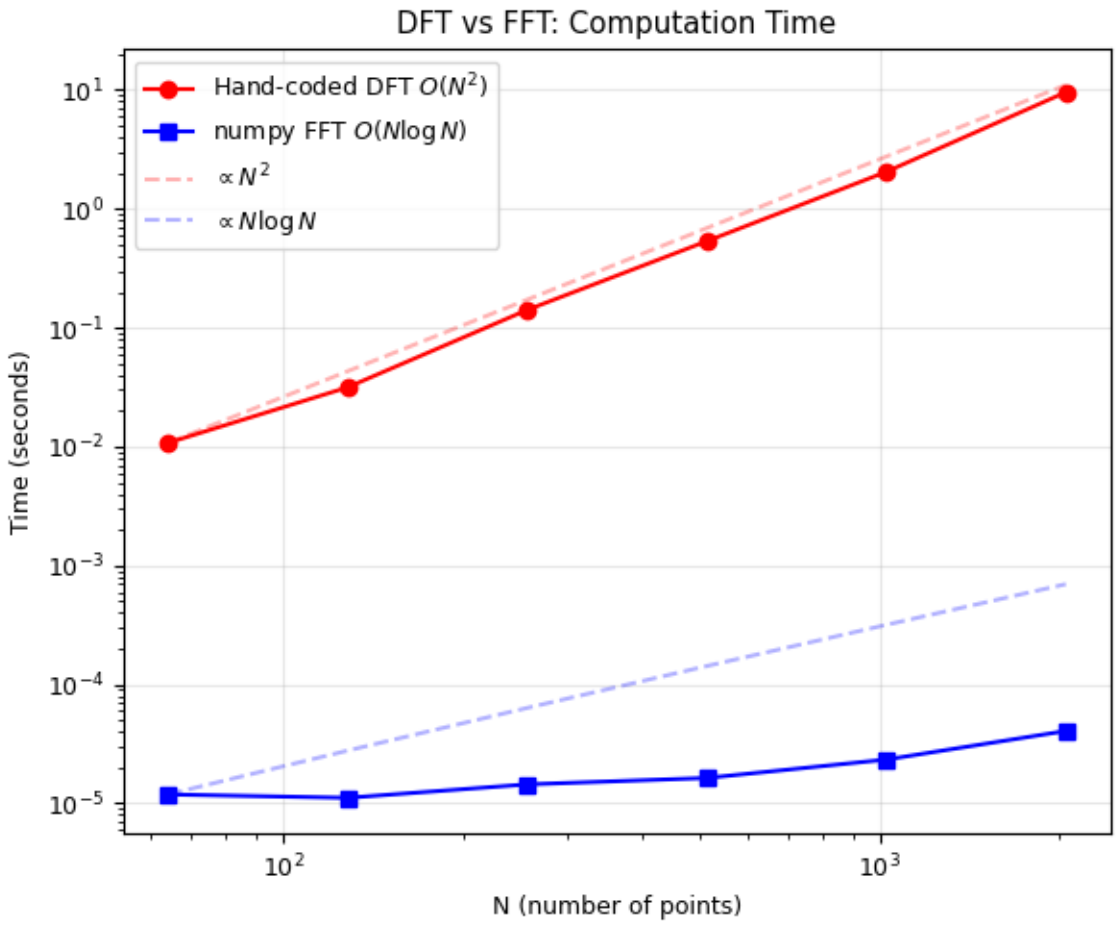
Let's measure it.

```
# Our hand-coded DFT from Lecture 08
def dft(f):
    """Compute the Discrete Fourier Transform (naive O(N^2) method).

    Parameters
    ----------
    f : array_like
        Input signal of length N.

    Returns
    -------
    F : ndarray
        DFT of the input signal.
    """
    N = len(f)
    F = np.zeros(N, dtype=complex)
    for n in range(N):
        for k in range(N):
            F[n] += f[k] * np.exp(-1j * 2 * np.pi * n * k / N)
    return F

# Timing comparison: DFT vs numpy FFT
N_values = [64, 128, 256, 512, 1024, 2048]
dft_times = []
fft_times = []

for N in N_values:
    x = np.random.randn(N)

    # Time our DFT
    t0 = time.time()
    F_dft = dft(x)
    dft_times.append(time.time() - t0)

    # Time numpy's FFT
    t0 = time.time()
    for _ in range(1000):  # Run 1000 times to get measurable time
        F_fft = np.fft.fft(x)
    fft_times.append((time.time() - t0) / 1000)

    # Verify they give the same answer!
    assert np.allclose(F_dft, F_fft), f"Mismatch at N={N}!"
    print(f"N={N:5d}:  DFT = {dft_times[-1]:.4f}s,  FFT = {fft_times[-1]:.6f}s,  "
          f"speedup = {dft_times[-1]/fft_times[-1]:.0f}x")
```

```
N=   64:  DFT = 0.0107s,  FFT = 0.000012s,  speedup = 903x
N=  128:  DFT = 0.0315s,  FFT = 0.000011s,  speedup = 2825x
N=  256:  DFT = 0.1410s,  FFT = 0.000014s,  speedup = 9755x
N=  512:  DFT = 0.5338s,  FFT = 0.000016s,  speedup = 32616x
N= 1024:  DFT = 2.0467s,  FFT = 0.000023s,  speedup = 87569x
N= 2048:  DFT = 9.5672s,  FFT = 0.000041s,  speedup = 235287x
```

```
# Plot timing comparison
fig, ax = plt.subplots(figsize=(6, 5))

ax.loglog(N_values, dft_times, 'ro-', label='Hand-coded DFT $O(N^2)$', markersize=6)
ax.loglog(N_values, fft_times, 'bs-', label='numpy FFT $O(N \log N)$', markersize=6)

# Theoretical scaling lines
N_arr = np.array(N_values, dtype=float)
ax.loglog(N_arr, dft_times[0] * (N_arr/N_arr[0])**2, 'r--', alpha=0.3, label='$\
↪propto N^2$')
ax.loglog(N_arr, fft_times[0] * (N_arr*np.log2(N_arr))/(N_arr[0]*np.log2(N_arr[0])),
          'b--', alpha=0.3, label='$\propto N \log N$')

ax.set_xlabel('N (number of points)')
ax.set_ylabel('Time (seconds)')
ax.set_title('DFT vs FFT: Computation Time')
ax.legend()
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
<>:5: SyntaxWarning: invalid escape sequence '\l'
<>:9: SyntaxWarning: invalid escape sequence '\p'
<>:11: SyntaxWarning: invalid escape sequence '\p'
<>:5: SyntaxWarning: invalid escape sequence '\l'
<>:9: SyntaxWarning: invalid escape sequence '\p'
<>:11: SyntaxWarning: invalid escape sequence '\p'
/tmp/ipython-input-654999808.py:5: SyntaxWarning: invalid escape sequence '\l'
  ax.loglog(N_values, fft_times, 'bs-', label='numpy FFT $O(N \log N)$',␣
↪markersize=6)
/tmp/ipython-input-654999808.py:9: SyntaxWarning: invalid escape sequence '\p'
  ax.loglog(N_arr, dft_times[0] * (N_arr/N_arr[0])**2, 'r--', alpha=0.3, label='$\
↪propto N^2$')
/tmp/ipython-input-654999808.py:11: SyntaxWarning: invalid escape sequence '\p'
  'b--', alpha=0.3, label='$\propto N \log N$')
```



9.2.1 The Speedup in Practice

N	DFT operations	FFT operations	Speedup
10^3	10^6	10^4	100×
10^4	10^8	1.3×10^5	750×
10^6	10^{12}	2×10^7	50,000×
10^8	10^{16}	2.7×10^9	4,000,000×

At $N = 10^6$, the DFT would take **hours**. The FFT takes **milliseconds**.

This is not a different algorithm giving an *approximation* — it computes the **exact same DFT**, just smarter.

9.3 II. The Cooley-Tukey Algorithm

9.3.1 The Key Idea: Divide and Conquer

Split the sum into **even-indexed** and **odd-indexed** terms:

$$F_n = \sum_{k=0}^{N-1} f_k \, W_N^{nk} \quad \text{where } W_N = e^{-2\pi i/N}$$

Separate even ($k = 2m$) and odd ($k = 2m+1$) terms:

$$F_n = \sum_{m=0}^{N/2-1} f_{2m} \, W_N^{n \cdot 2m} + \sum_{m=0}^{N/2-1} f_{2m+1} \, W_N^{n(2m+1)}$$

Using $W_N^2 = W_{N/2}$:

$$\boxed{F_n = E_n + W_N^n \cdot O_n}$$

where:

- E_n = DFT of even-indexed elements (size $N/2$)
- O_n = DFT of odd-indexed elements (size $N/2$)
- $W_N^n = e^{-2\pi i n/N}$ are called **twiddle factors**

9.3.2 The Symmetry Trick

Since E_n and O_n have period $N/2$:

$$F_n = E_n + W_N^n \cdot O_n \quad \text{for } n = 0, 1, \ldots, N/2 - 1$$

$$F_{n+N/2} = E_n - W_N^n \cdot O_n \quad (\text{because } W_N^{n+N/2} = -W_N^n)$$

This is the **butterfly operation**: one complex multiply + one add/subtract gives **two** output values!

9.3.3 Recursion: Keep Splitting!

Each half-size DFT can be split again:

$$N \xrightarrow{\text{split}} 2 \times \frac{N}{2} \xrightarrow{\text{split}} 4 \times \frac{N}{4} \xrightarrow{\text{split}} \cdots \xrightarrow{\text{split}} N \times 1$$

- Number of levels: $\log_2 N$
- Work per level: $O(N)$ (butterfly operations)
- **Total:** $O(N \log N)$

Requirement: N must be a power of 2 (for radix-2 FFT).

In practice, we **zero-pad** to the next power of 2.

```python
def fft_recursive(x):
    """Compute FFT using the Cooley-Tukey radix-2 algorithm (recursive).

    Parameters
    ----------
    x : array_like
        Input signal. Length must be a power of 2.

    Returns
    -------
    F : ndarray
        FFT of the input signal.
    """
    N = len(x)

    # Base case: DFT of a single element is itself
    if N == 1:
        return np.array(x, dtype=complex)

    if N % 2 != 0:
        raise ValueError("Length must be a power of 2")

    # Recursively compute FFT of even and odd parts
    E = fft_recursive(x[0::2])   # Even-indexed: x[0], x[2], x[4], ...
    O = fft_recursive(x[1::2])   # Odd-indexed:  x[1], x[3], x[5], ...

    # Twiddle factors
    n = np.arange(N // 2)
    W = np.exp(-2j * np.pi * n / N)

    # Butterfly: combine even and odd
    F = np.zeros(N, dtype=complex)
    F[:N//2] = E + W * O       # First half
    F[N//2:] = E - W * O       # Second half (symmetry!)

    return F

# Test it!
x_test = np.random.randn(256)

F_ours = fft_recursive(x_test)
F_numpy = np.fft.fft(x_test)

print(f"Max difference: {np.max(np.abs(F_ours - F_numpy)):.2e}")
print(f"Match: {np.allclose(F_ours, F_numpy)}")
```

```
Max difference: 1.78e-14
Match: True
```

```python
# Time our recursive FFT vs DFT vs numpy
N = 1024
x = np.random.randn(N)

t0 = time.time()
F1 = dft(x)
t_dft = time.time() - t0
```

(continues on next page)

(continued from previous page)

```
t0 = time.time()
F2 = fft_recursive(x)
t_ours = time.time() - t0

t0 = time.time()
for _ in range(1000):
    F3 = np.fft.fft(x)
t_numpy = (time.time() - t0) / 1000

print(f"N = {N}")
print(f"Hand-coded DFT:    {t_dft:.4f} s")
print(f"Our recursive FFT: {t_ours:.4f} s  ({t_dft/t_ours:.1f}x faster than DFT)")
print(f"numpy FFT:         {t_numpy:.6f} s  ({t_dft/t_numpy:.0f}x faster than DFT)")
print(f"\nnumpy is {t_ours/t_numpy:.0f}x faster than our FFT")
print("(numpy uses optimized C code with FFTW-style algorithms)")
```

```
N = 1024
Hand-coded DFT:    2.0269 s
Our recursive FFT: 0.0106 s  (191.1x faster than DFT)
numpy FFT:         0.000038 s  (52911x faster than DFT)

numpy is 277x faster than our FFT
(numpy uses optimized C code with FFTW-style algorithms)
```

9.3.4 The Butterfly Diagram

For N=8, the FFT computation flow looks like:

```
Input order:   x[0]  x[4]  x[2]  x[6]  x[1]  x[5]  x[3]  x[7]
               (bit-reversed order)

Stage 1:       x---x  x---x  x---x  x---x     (N/2 = 4 butterflies, size 2)
Stage 2:       x---+--x     x---+--x          (N/2 = 4 butterflies, size 4)
               x---+--x     x---+--x
Stage 3:       x---+--+--+--x                 (N/2 = 4 butterflies, size 8)
               x---+--+--+--x
               x---+--+--+--x
               x---+--+--+--x

Output:        F[0] F[1] F[2] F[3] F[4] F[5] F[6] F[7]
```

Each **butterfly** computes:

- $a + W \cdot b$ (top output)
- $a - W \cdot b$ (bottom output)

Total: $\log_2(8) = 3$ stages × $N/2 = 4$ butterflies = **12 complex multiplications**

Compare: DFT needs $N^2 = 64$ multiplications → **5× savings** even at $N = 8$!

9.4 III. Using numpy/scipy FFT in Practice

Now that we understand *how* the FFT works, let's use the optimized library functions.

9.4.1 Key Functions

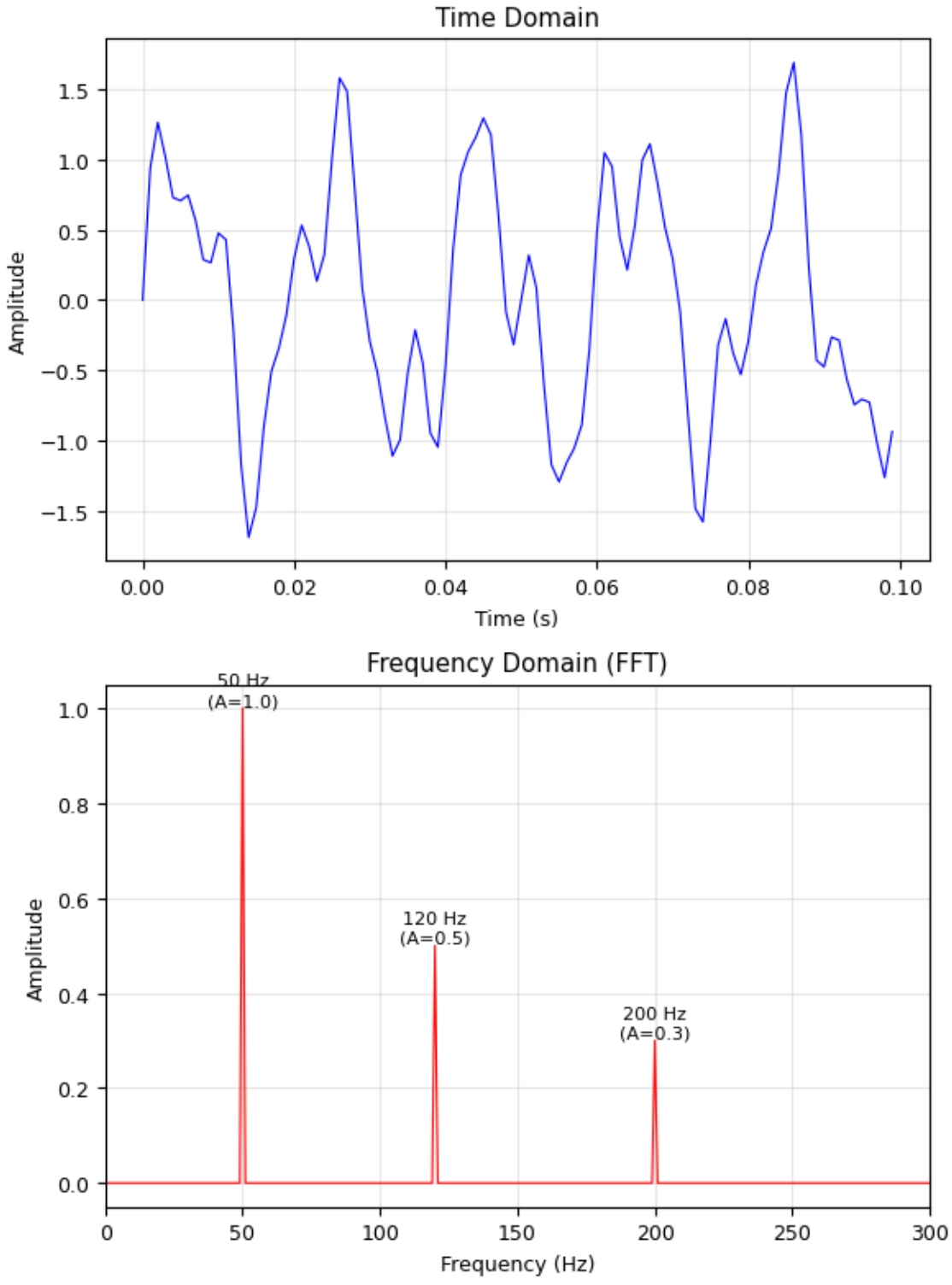
Function	Purpose
`np.fft.fft(x)`	Forward FFT
`np.fft.ifft(F)`	Inverse FFT
`np.fft.fftfreq(N, d)`	Frequency axis (d = sample spacing)
`np.fft.rfft(x)`	FFT for real signals (returns only positive frequencies)
`np.fft.fftshift(F)`	Shift zero-frequency to center
`np.fft.fft2(x)`	2D FFT (for images)

```
# Create a signal with known frequencies
fs = 1000           # Sampling rate (Hz)
T = 1.0             # Duration (s)
N = int(fs * T)     # Number of samples
t = np.linspace(0, T, N, endpoint=False)

# Signal: 50 Hz + 120 Hz + 200 Hz
x = 1.0 * np.sin(2 * np.pi * 50 * t) + \
    0.5 * np.sin(2 * np.pi * 120 * t) + \
    0.3 * np.sin(2 * np.pi * 200 * t)

# Compute FFT
F = np.fft.fft(x)
freqs = np.fft.fftfreq(N, d=1/fs)  # Frequency axis

# Plot
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 8))

# Time domain
ax1.plot(t[:100], x[:100], 'b-', linewidth=0.8)
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Amplitude')
ax1.set_title('Time Domain')
ax1.grid(True, alpha=0.3)

# Frequency domain (only positive frequencies)
positive = freqs > 0
ax2.plot(freqs[positive],  2/N*np.abs(F[positive]), 'r-', linewidth=0.8)
ax2.set_xlabel('Frequency (Hz)')
ax2.set_ylabel('Amplitude')
ax2.set_title('Frequency Domain (FFT)')
ax2.set_xlim(0, 300)
ax2.grid(True, alpha=0.3)

# Annotate peaks
for freq, amp in [(50, 1.0), (120, 0.5), (200, 0.3)]:
    ax2.annotate(f'{freq} Hz\n(A={amp})', xy=(freq, amp), fontsize=8,
                 ha='center', va='bottom')
```

(continues on next page)

(continued from previous page)

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



```
# Understanding fftfreq
print("np.fft.fftfreq(N, d=1/fs) explained:")
print(f"  N = {N} samples")
print(f"  d = 1/fs = {1/fs} s (time between samples)")
print(f"  Frequency resolution: Δf = fs/N = {fs/N} Hz")
print(f"  Max frequency (Nyquist): fs/2 = {fs/2} Hz")
print()
print("First 10 frequencies:", freqs[:10])
print("Last 5 frequencies: ", freqs[-5:])
print("\nNote: negative frequencies appear in the second half!")
print("Use fftshift() to center them, or rfft() for real signals.")
```

```
np.fft.fftfreq(N, d=1/fs) explained:
  N = 1000 samples
  d = 1/fs = 0.001 s (time between samples)
  Frequency resolution: Δf = fs/N = 1.0 Hz
  Max frequency (Nyquist): fs/2 = 500.0 Hz

First 10 frequencies: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Last 5 frequencies:  [-5. -4. -3. -2. -1.]

Note: negative frequencies appear in the second half!
Use fftshift() to center them, or rfft() for real signals.
```

```
# rfft: efficient FFT for real signals
# For real input, F[-n] = F[n]* (conjugate symmetry)
# So we only need the positive half!

F_full = np.fft.fft(x)        # Returns N complex values
F_real = np.fft.rfft(x)       # Returns N/2 + 1 complex values
freqs_real = np.fft.rfftfreq(N, d=1/fs)

print(f"fft output length:  {len(F_full)}")
print(f"rfft output length: {len(F_real)}  (about half!)")
print(f"\nFor real signals, rfft is ~2x faster and uses ~half the memory.")

# They give the same result for positive frequencies
print(f"Same result? {np.allclose(F_full[:N//2+1], F_real)}")
```

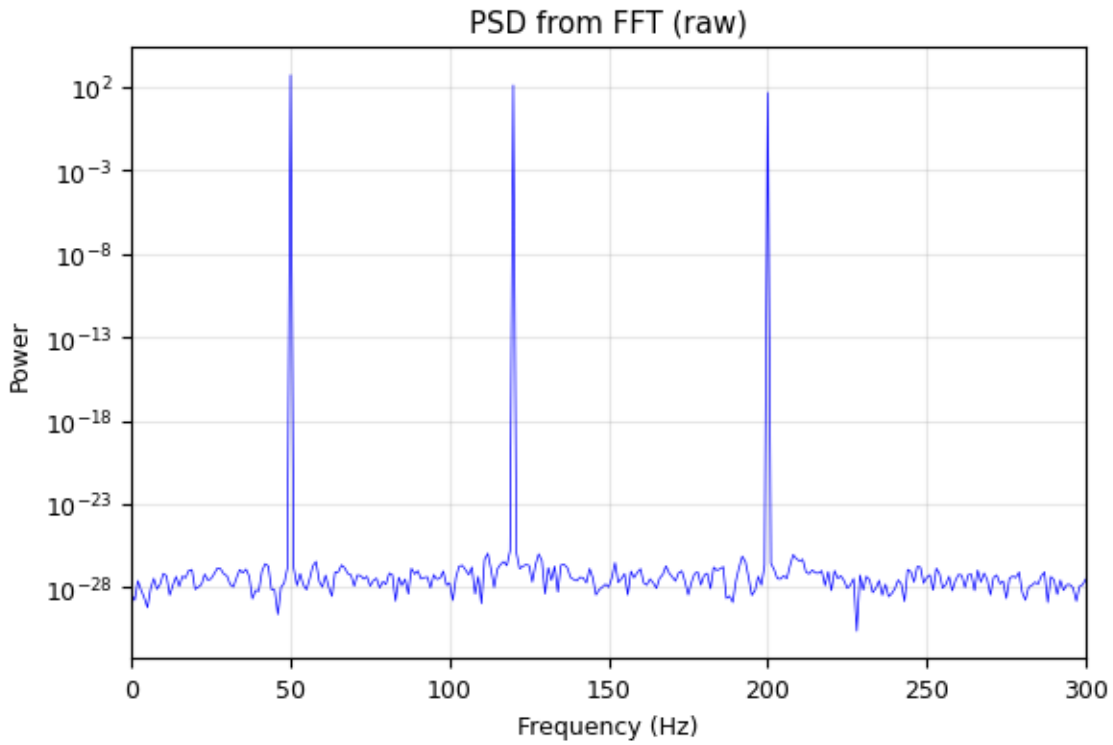
```
fft output length:  1000
rfft output length: 501  (about half!)

For real signals, rfft is ~2x faster and uses ~half the memory.
Same result? True
```

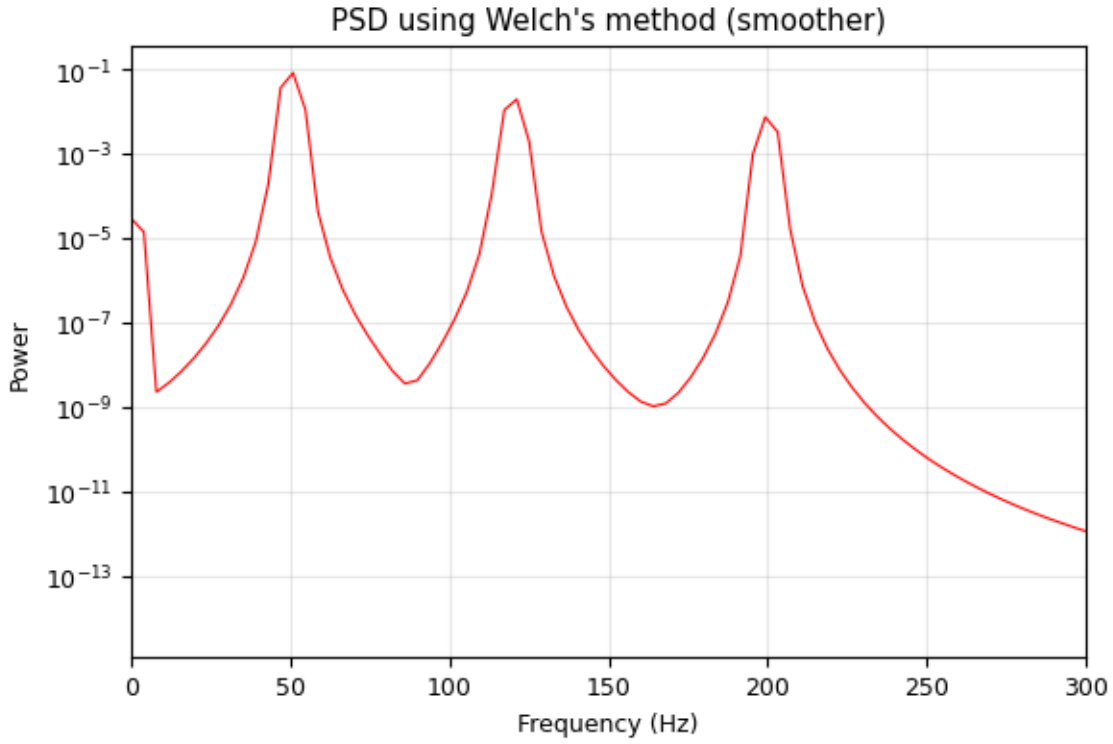
```
# Power Spectral Density (PSD)
# PSD = |F(f)|^2 / N — shows power at each frequency

# Method 1: Direct from FFT
psd_direct = (2 / N) * np.abs(F_real)**2

# Method 2: Welch's method (averages over segments, reduces noise)
freqs_welch, psd_welch = signal.welch(x, fs=fs, nperseg=256)

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 8))
```

(continues on next page)

(continued from previous page)

```
ax1.semilogy(freqs_real, psd_direct, 'b-', linewidth=0.5)
ax1.set_title('PSD from FFT (raw)')
ax1.set_xlabel('Frequency (Hz)')
ax1.set_ylabel('Power')
ax1.set_xlim(0, 300)
ax1.grid(True, alpha=0.3)

ax2.semilogy(freqs_welch, psd_welch, 'r-', linewidth=0.8)
ax2.set_title("PSD using Welch's method (smoother)")
ax2.set_xlabel('Frequency (Hz)')
ax2.set_ylabel('Power')
ax2.set_xlim(0, 300)
ax2.grid(True, alpha=0.3)

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


PSD from FFT (raw)
Power
10^{2}
10^{-3}
10^{-8}
10^{-13}
10^{-18}
10^{-23}
10^{-28}
0
50
100
150
200
250
300
Frequency (Hz)


PSD using Welch's method (smoother)
Power
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
0
50
100
150
200
250
300
Frequency (Hz)

9.5 IV. Zero-Padding and Frequency Resolution

9.5.1 Frequency Resolution

$$\Delta f = \frac{f_s}{N}$$

To get finer frequency resolution, you need **more data** (larger N at the same f_s).

9.5.2 Zero-Padding

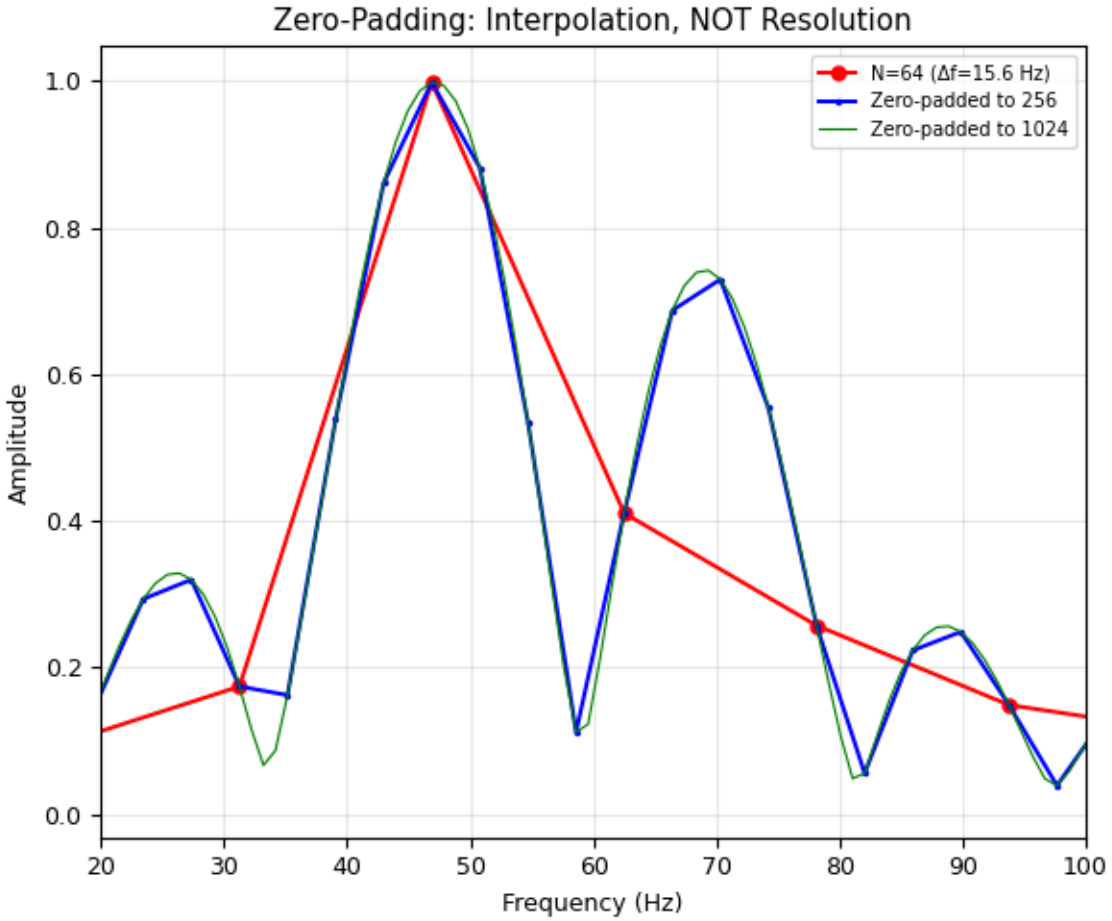
Appending zeros to the signal before FFT **interpolates** the spectrum (smoother plot), but does **NOT** increase the true frequency resolution.

```
# Zero-padding demonstration
fs = 1000
N_orig = 64   # Short signal
t_orig = np.arange(N_orig) / fs
x_orig = np.sin(2 * np.pi * 50 * t_orig) + 0.7 * np.sin(2 * np.pi * 65 * t_orig)

# FFT without zero-padding
F1 = np.fft.rfft(x_orig)
f1 = np.fft.rfftfreq(N_orig, 1/fs)

# FFT with zero-padding to 256
N_pad = 256
x_padded = np.zeros(N_pad)
x_padded[:N_orig] = x_orig
F2 = np.fft.rfft(x_padded)
f2 = np.fft.rfftfreq(N_pad, 1/fs)

# FFT with zero-padding to 1024
N_pad2 = 1024
x_padded2 = np.zeros(N_pad2)
x_padded2[:N_orig] = x_orig
F3 = np.fft.rfft(x_padded2)
f3 = np.fft.rfftfreq(N_pad2, 1/fs)

fig, ax = plt.subplots(figsize=(6, 5))
ax.plot(f1, 2/N_orig * np.abs(F1), 'ro-', markersize=5, label=f'N={N_orig} (Δf={fs/N_
↪orig:.1f} Hz)')
ax.plot(f2, 2/N_orig * np.abs(F2), 'b.-', markersize=3, label=f'Zero-padded to {N_pad}
↪')
ax.plot(f3, 2/N_orig * np.abs(F3), 'g-', linewidth=0.8, label=f'Zero-padded to {N_
↪pad2}')

ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel('Amplitude')
ax.set_title('Zero-Padding: Interpolation, NOT Resolution')
ax.set_xlim(20, 100)
ax.legend(fontsize=7)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f"Original Δf = {fs/N_orig:.1f} Hz — can barely resolve 50 and 65 Hz")
```

(continues on next page)

(continued from previous page)

```
print(f"Zero-padding makes the plot smoother but doesn't separate the peaks better.")
print(f"To truly resolve two frequencies, you need: Δf < |f2 - f1| = 15 Hz")
print(f"→ Need N > fs/15 = {fs/15:.0f} samples of REAL data.")
```




```
Original Δf = 15.6 Hz — can barely resolve 50 and 65 Hz
Zero-padding makes the plot smoother but doesn't separate the peaks better.
To truly resolve two frequencies, you need: Δf < |f2 - f1| = 15 Hz
→ Need N > fs/15 = 67 samples of REAL data.
```

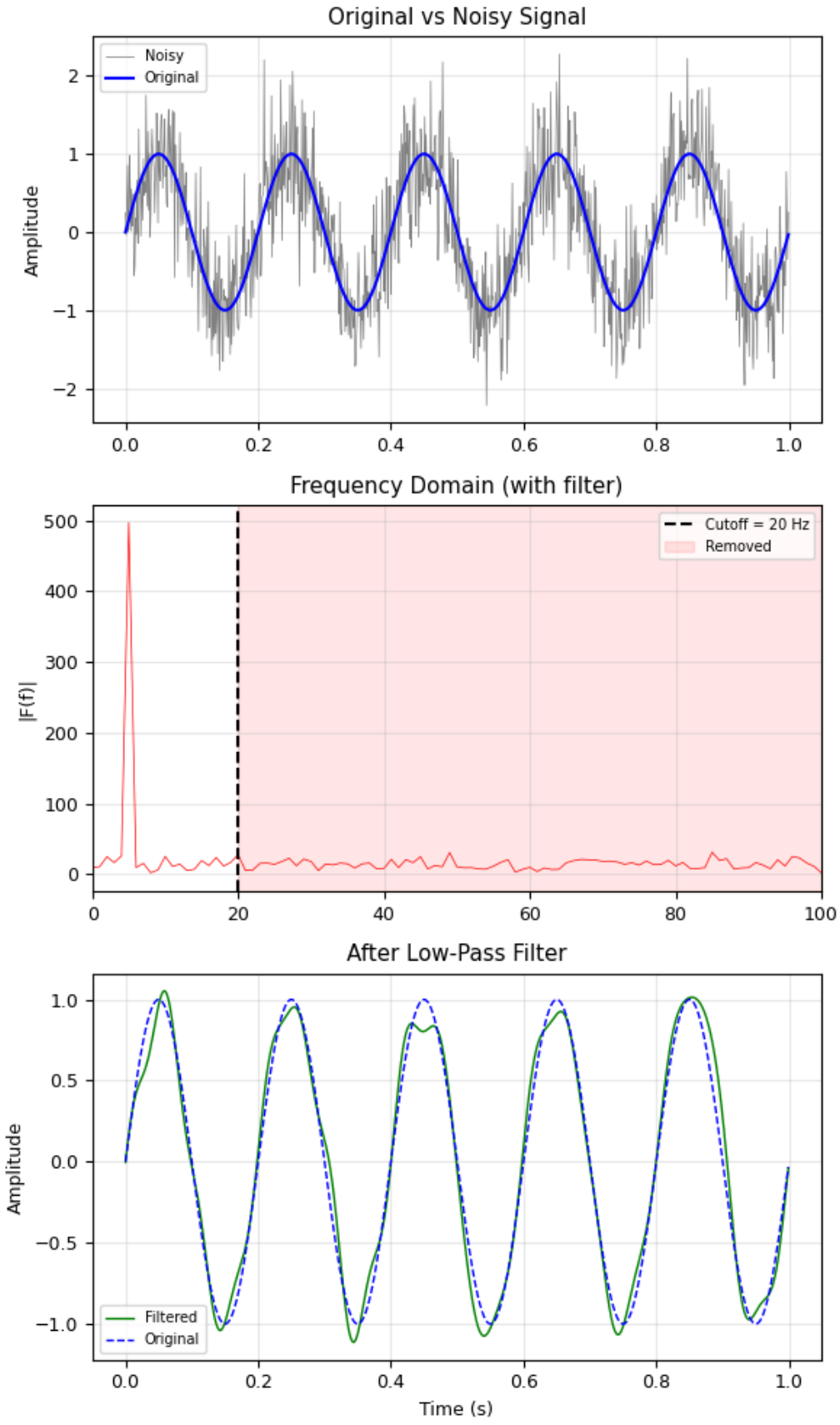
9.6 V. Filtering in the Frequency Domain

One of the most powerful applications of FFT:

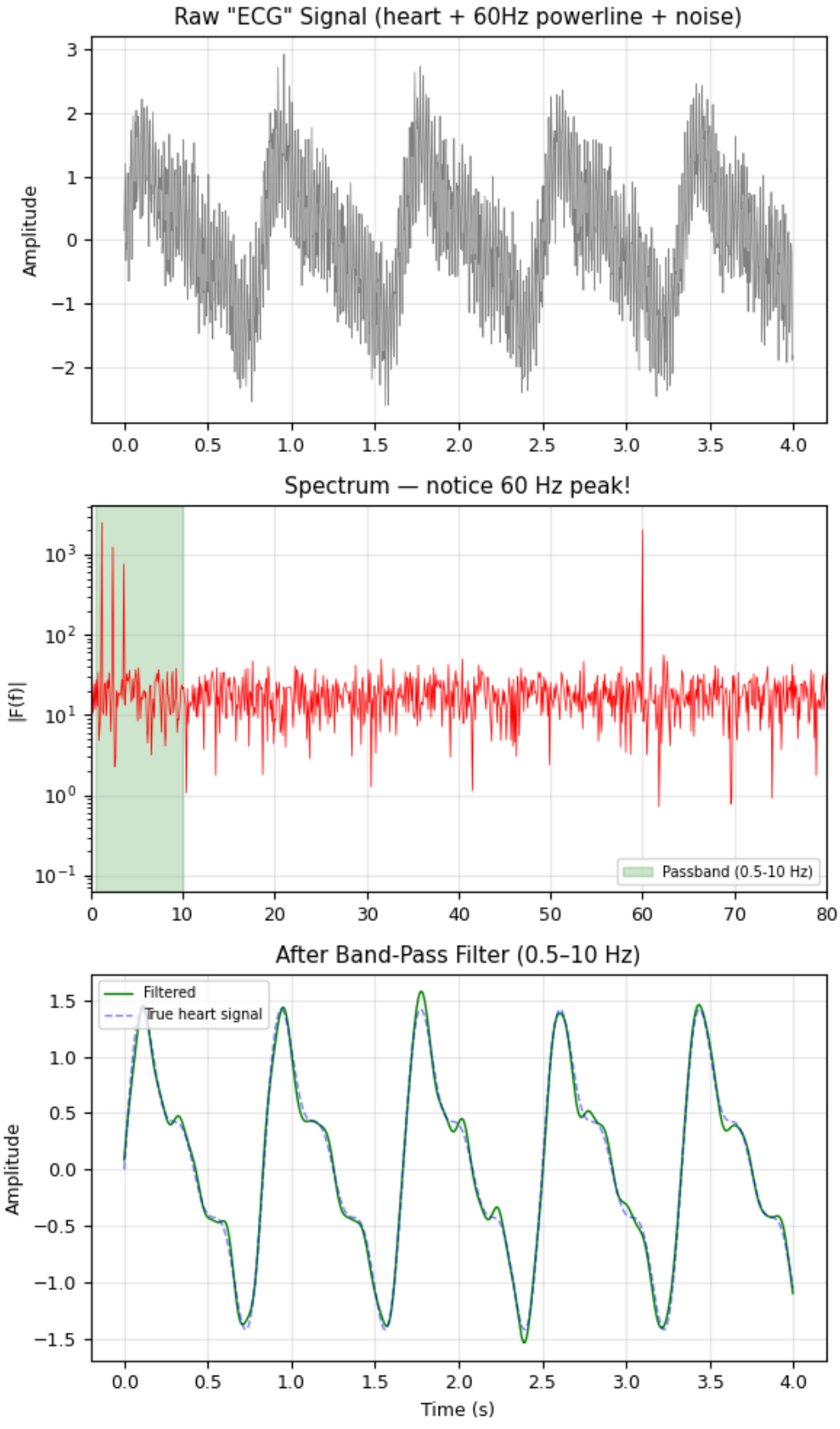
1. **FFT** the signal → frequency domain
2. **Modify** the spectrum (zero out unwanted frequencies)
3. **IFFT** back → clean signal

This is $O(N \log N)$ — much faster than convolution in the time domain!

```
# Low-pass filter: remove high-frequency noise
fs = 1000
N = 1000
t = np.linspace(0, 1, N, endpoint=False)

# Clean signal: 5 Hz sine wave
x_clean = np.sin(2 * np.pi * 5 * t)

# Add high-frequency noise
np.random.seed(42)
x_noisy = x_clean + 0.5 * np.random.randn(N)

# FFT
F = np.fft.rfft(x_noisy)
freqs = np.fft.rfftfreq(N, d=1/fs)

# Low-pass filter: zero out everything above 20 Hz
cutoff = 20  # Hz
F_filtered = F.copy()
F_filtered[freqs > cutoff] = 0

# IFFT back to time domain
x_filtered = np.fft.irfft(F_filtered, n=N)

# Plot
fig, axes = plt.subplots(3, 1, figsize=(6, 10))

axes[0].plot(t, x_noisy, 'gray', linewidth=0.5, label='Noisy')
axes[0].plot(t, x_clean, 'b-', linewidth=1.5, label='Original')
axes[0].set_title('Original vs Noisy Signal')
axes[0].legend(fontsize=7)
axes[0].set_ylabel('Amplitude')
axes[0].grid(True, alpha=0.3)

axes[1].plot(freqs, np.abs(F), 'r-', linewidth=0.5)
axes[1].axvline(cutoff, color='k', linestyle='--', label=f'Cutoff = {cutoff} Hz')
axes[1].axvspan(cutoff, freqs[-1], alpha=0.1, color='red', label='Removed')
axes[1].set_title('Frequency Domain (with filter)')
axes[1].set_xlim(0, 100)
axes[1].set_ylabel('|F(f)|')
axes[1].legend(fontsize=7)
axes[1].grid(True, alpha=0.3)

axes[2].plot(t, x_filtered, 'g-', linewidth=1, label='Filtered')
axes[2].plot(t, x_clean, 'b--', linewidth=1, label='Original')
axes[2].set_title('After Low-Pass Filter')
axes[2].set_xlabel('Time (s)')
axes[2].set_ylabel('Amplitude')
axes[2].legend(fontsize=7)
axes[2].grid(True, alpha=0.3)

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


Original vs Noisy Signal
Noisy
Original
Amplitude
Frequency Domain (with filter)
Cutoff = 20 Hz
Removed
|F(f)|
After Low-Pass Filter
Filtered
Original
Amplitude
Time (s)

```
# Band-pass filter: extract a specific frequency range
# Simulated "ECG" signal with heart rate + noise + powerline interference
fs = 500
N = 5000
t = np.linspace(0, N/fs, N, endpoint=False)

# Heart signal: ~1.2 Hz (72 BPM) with harmonics
heart = (np.sin(2 * np.pi * 1.2 * t) +
         0.5 * np.sin(2 * np.pi * 2.4 * t) +
         0.3 * np.sin(2 * np.pi * 3.6 * t))

# Powerline interference at 60 Hz
powerline = 0.8 * np.sin(2 * np.pi * 60 * t)

# Random noise
np.random.seed(42)
noise = 0.3 * np.random.randn(N)

# Measured signal
ecg = heart + powerline + noise

# FFT and band-pass filter (keep 0.5 - 10 Hz)
F = np.fft.rfft(ecg)
freqs = np.fft.rfftfreq(N, d=1/fs)

# Band-pass: keep only 0.5 to 10 Hz
F_bp = F.copy()
F_bp[(freqs < 0.5) | (freqs > 10)] = 0

ecg_clean = np.fft.irfft(F_bp, n=N)

fig, axes = plt.subplots(3, 1, figsize=(6, 10))

axes[0].plot(t[:2000], ecg[:2000], 'gray', linewidth=0.5)
axes[0].set_title('Raw "ECG" Signal (heart + 60Hz powerline + noise)')
axes[0].set_ylabel('Amplitude')
axes[0].grid(True, alpha=0.3)

axes[1].semilogy(freqs, np.abs(F), 'r-', linewidth=0.5)
axes[1].axvspan(0.5, 10, alpha=0.2, color='green', label='Passband (0.5-10 Hz)')
axes[1].set_xlim(0, 80)
axes[1].set_title('Spectrum — notice 60 Hz peak!')
axes[1].set_ylabel('|F(f)|')
axes[1].legend(fontsize=7)
axes[1].grid(True, alpha=0.3)

axes[2].plot(t[:2000], ecg_clean[:2000], 'g-', linewidth=1, label='Filtered')
axes[2].plot(t[:2000], heart[:2000], 'b--', linewidth=1, alpha=0.5, label='True heart␣
 ↪signal')
axes[2].set_title('After Band-Pass Filter (0.5-10 Hz)')
axes[2].set_xlabel('Time (s)')
axes[2].set_ylabel('Amplitude')
axes[2].legend(fontsize=7)
axes[2].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
print("60 Hz powerline interference completely removed!")
```


Raw "ECG" Signal (heart + 60Hz powerline + noise)
Amplitude
Spectrum — notice 60 Hz peak!
|F(f)|
Passband (0.5-10 Hz)
After Band-Pass Filter (0.5–10 Hz)
Filtered
True heart signal
Amplitude
Time (s)

```
60 Hz powerline interference completely removed!
```

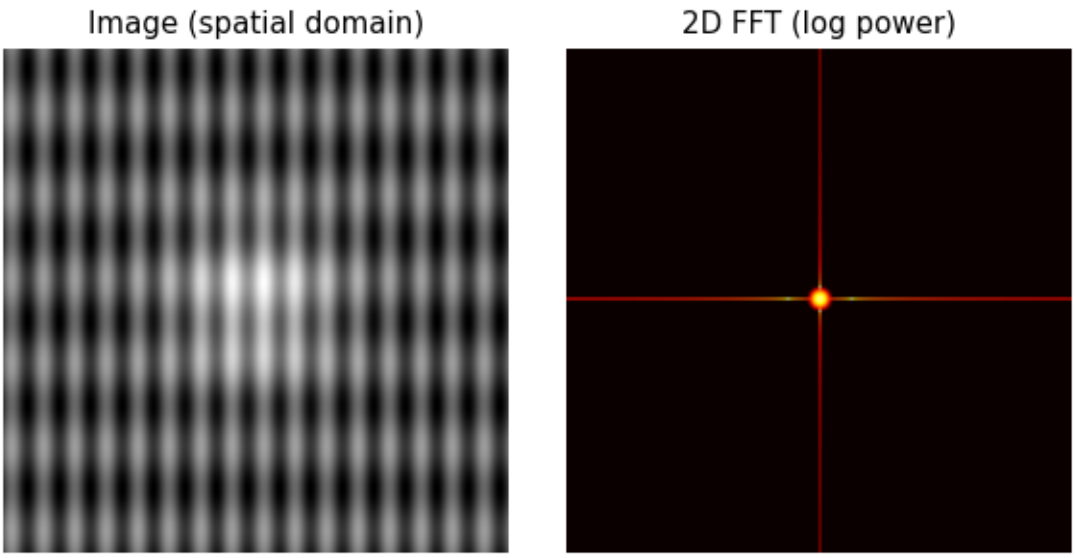
9.7 VI. 2D FFT and Image Processing

Images are 2D signals. The 2D FFT decomposes an image into spatial frequencies:

$$F(k_x, k_y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n)\, e^{-2\pi i (k_x m/M + k_y n/N)}$$

- Low spatial frequencies → smooth features, overall brightness
- High spatial frequencies → edges, fine details, noise

```
# Create a test image: stripes + a circle
M, N_img = 256, 256
x = np.linspace(-1, 1, N_img)
y = np.linspace(-1, 1, M)
X, Y = np.meshgrid(x, y)

# Vertical stripes + horizontal stripes + gaussian blob
image = (np.sin(2 * np.pi * 8 * X) +          # Vertical stripes (8 cycles)
         0.5 * np.sin(2 * np.pi * 3 * Y) +    # Horizontal stripes (3 cycles)
         2 * np.exp(-(X**2 + Y**2) / 0.1))    # Gaussian blob

# 2D FFT
F_2d = np.fft.fft2(image)
F_shifted = np.fft.fftshift(F_2d)  # Move zero-frequency to center
power_spectrum = np.log10(np.abs(F_shifted) + 1)  # Log scale for visibility

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(6, 3))

ax1.imshow(image, cmap='gray', origin='lower')
ax1.set_title('Image (spatial domain)')
ax1.axis('off')

ax2.imshow(power_spectrum, cmap='hot', origin='lower')
ax2.set_title('2D FFT (log power)')
ax2.axis('off')

plt.tight_layout()
plt.show()

print("Bright spots in the FFT correspond to dominant spatial frequencies:")
print("  - Center: DC component (average brightness)")
print("  - Horizontal pair: vertical stripes")
print("  - Vertical pair: horizontal stripes")
```



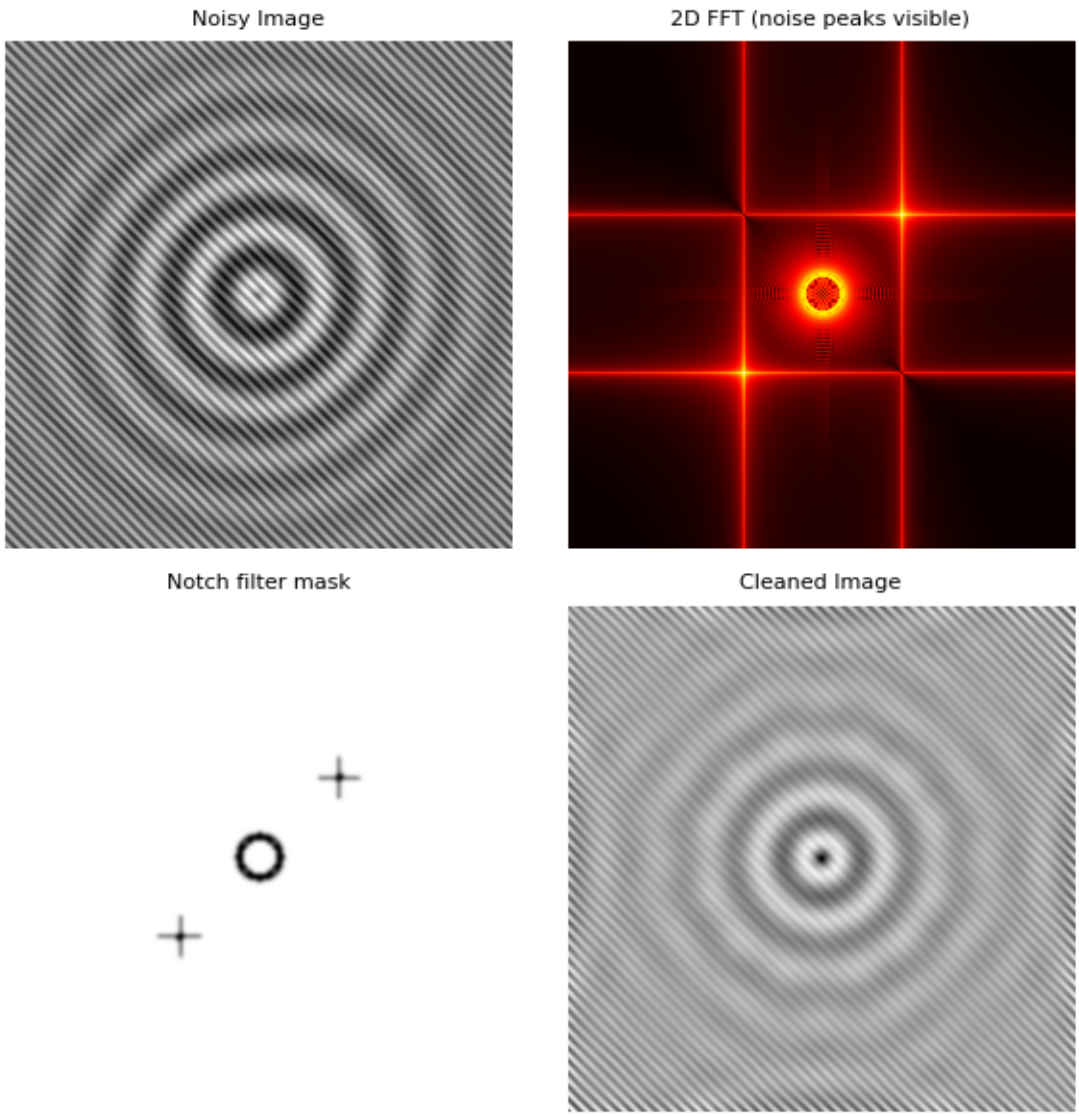

```
Bright spots in the FFT correspond to dominant spatial frequencies:
  - Center: DC component (average brightness)
  - Horizontal pair: vertical stripes
  - Vertical pair: horizontal stripes
```

```python
# Image denoising: remove periodic noise
# Create a clean image (concentric rings)
R = np.sqrt(X**2 + Y**2)
clean_image = np.sin(2 * np.pi * 5 * R) * np.exp(-R**2 / 0.5)

# Add periodic noise (diagonal stripes)
np.random.seed(42)
periodic_noise = 0.8 * np.sin(2 * np.pi * (20 * X + 20 * Y))
noisy_image = clean_image + periodic_noise

# 2D FFT
F_noisy = np.fft.fft2(noisy_image)
F_noisy_shifted = np.fft.fftshift(F_noisy)

# Identify and remove the noise peaks
# The periodic noise creates bright spots at specific spatial frequencies
power = np.abs(F_noisy_shifted)

# Create a mask: suppress points far from center that are unusually bright
ky = np.arange(M) - M//2
kx = np.arange(N_img) - N_img//2
KX, KY = np.meshgrid(kx, ky)
K_radius = np.sqrt(KX**2 + KY**2)

# Notch filter: remove the noise peaks (at diagonal positions)
mask = np.ones_like(power)
# Find peaks away from center
threshold = np.percentile(power[K_radius > 5], 99.5)
mask[(power > threshold) & (K_radius > 5)] = 0

# Smooth the mask slightly to avoid ringing
from scipy.ndimage import gaussian_filter
```

(continues on next page)

(continued from previous page)

```
mask_smooth = gaussian_filter(mask, sigma=1)

# Apply mask and inverse FFT
F_cleaned = F_noisy_shifted * mask_smooth
cleaned_image = np.real(np.fft.ifft2(np.fft.ifftshift(F_cleaned)))

# Plot
fig, axes = plt.subplots(2, 2, figsize=(6, 6))

axes[0, 0].imshow(noisy_image, cmap='gray', origin='lower')
axes[0, 0].set_title('Noisy Image', fontsize=8)
axes[0, 0].axis('off')

axes[0, 1].imshow(np.log10(power + 1), cmap='hot', origin='lower')
axes[0, 1].set_title('2D FFT (noise peaks visible)', fontsize=8)
axes[0, 1].axis('off')

axes[1, 0].imshow(mask_smooth, cmap='gray', origin='lower')
axes[1, 0].set_title('Notch filter mask', fontsize=8)
axes[1, 0].axis('off')

axes[1, 1].imshow(cleaned_image, cmap='gray', origin='lower')
axes[1, 1].set_title('Cleaned Image', fontsize=8)
axes[1, 1].axis('off')

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

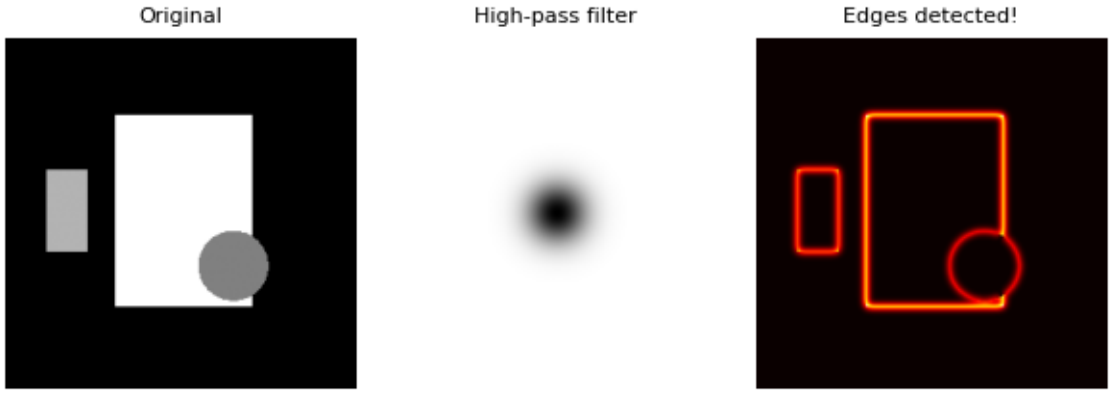



```
# Edge detection via high-pass filter in frequency domain
# Create a test image with sharp features
image_sharp = np.zeros((256, 256))
image_sharp[60:200, 80:180] = 1.0   # Rectangle
image_sharp[100:160, 30:60] = 0.7   # Small rectangle
# Add a circle
R_circle = np.sqrt((X - 0.3)**2 + (Y + 0.3)**2)
image_sharp[R_circle < 0.2] = 0.5

# 2D FFT
F_sharp = np.fft.fft2(image_sharp)
F_sharp_shifted = np.fft.fftshift(F_sharp)

# High-pass filter: remove low frequencies (keep edges)
hp_mask = 1 - np.exp(-(KX**2 + KY**2) / (2 * 15**2))  # Gaussian high-pass

F_edges = F_sharp_shifted * hp_mask
edges = np.real(np.fft.ifft2(np.fft.ifftshift(F_edges)))
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(1, 3, figsize=(6, 2.5))

axes[0].imshow(image_sharp, cmap='gray', origin='lower')
axes[0].set_title('Original', fontsize=8)
axes[0].axis('off')

axes[1].imshow(hp_mask, cmap='gray', origin='lower')
axes[1].set_title('High-pass filter', fontsize=8)
axes[1].axis('off')

axes[2].imshow(np.abs(edges), cmap='hot', origin='lower')
axes[2].set_title('Edges detected!', fontsize=8)
axes[2].axis('off')

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



9.8 VII. Physics Applications

9.8.1 Application 1: Phonon Spectrum from Molecular Dynamics

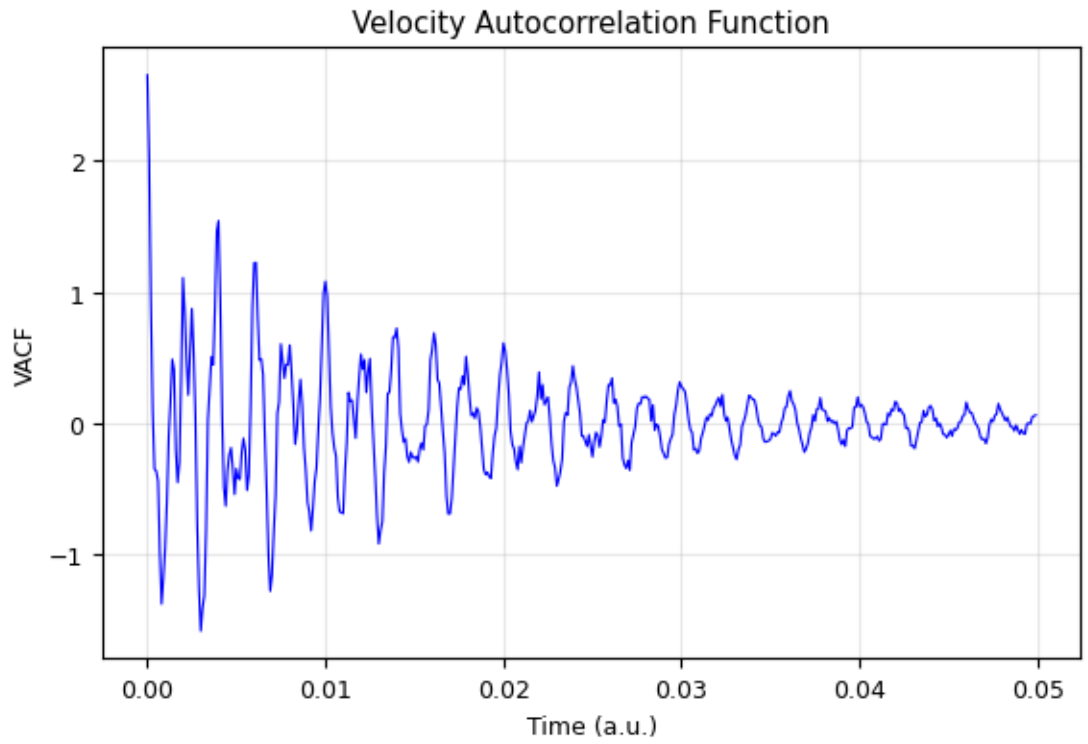
In computational materials science, we simulate atomic vibrations and use the FFT to extract the **phonon density of states** — the distribution of vibrational frequencies in a crystal.

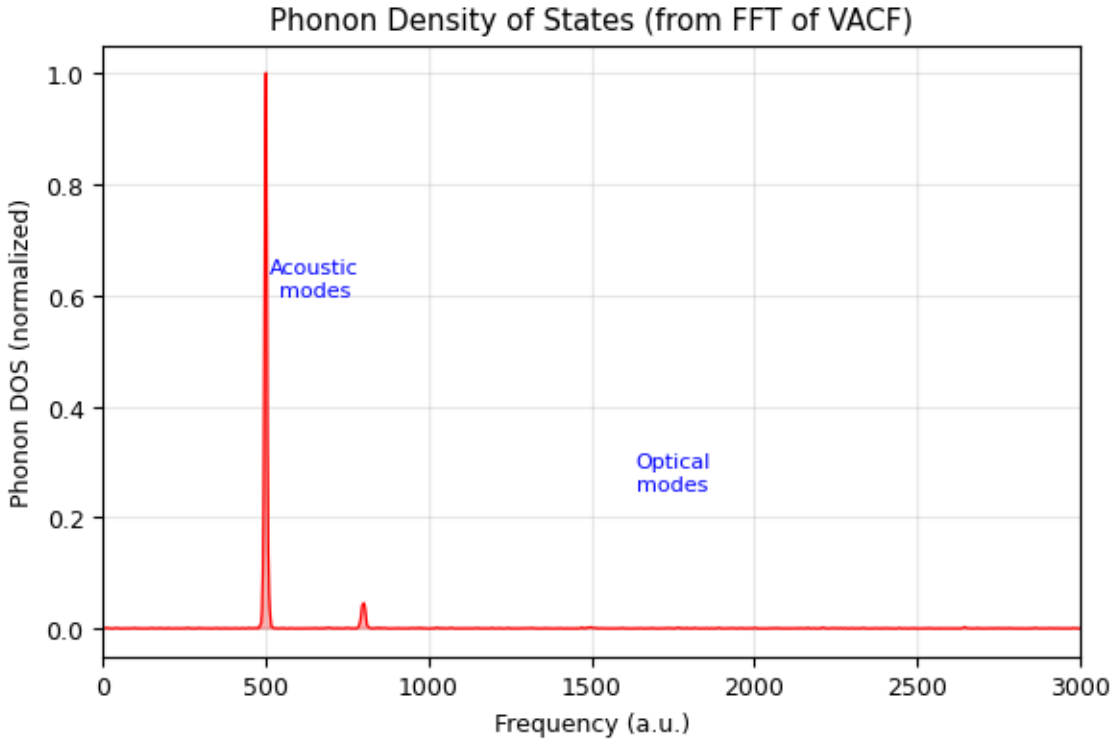
```
# Simulated velocity autocorrelation function (VACF) of atoms in a crystal
# The FFT of the VACF gives the phonon density of states

fs = 10000  # sampling rate (arbitrary units, ~femtoseconds)
N = 4096
t = np.arange(N) / fs

# Simulated VACF: decaying oscillations at characteristic phonon frequencies
# Acoustic modes: low frequency (~500, 800 THz)
# Optical modes: higher frequency (~1500, 2000 THz)
np.random.seed(42)
vacf = (1.0 * np.cos(2 * np.pi * 500 * t) * np.exp(-t * 50) +
        0.8 * np.cos(2 * np.pi * 800 * t) * np.exp(-t * 80) +
```

(continues on next page)

(continued from previous page)

```
        0.5 * np.cos(2 * np.pi * 1500 * t) * np.exp(-t * 120) +
        0.3 * np.cos(2 * np.pi * 2000 * t) * np.exp(-t * 150) +
        0.1 * np.random.randn(N) * np.exp(-t * 30))

# Apply Hann window to reduce leakage
window = signal.windows.hann(N)
vacf_windowed = vacf * window

# FFT to get phonon DOS
F = np.fft.rfft(vacf_windowed)
freqs = np.fft.rfftfreq(N, d=1/fs)
dos = np.abs(F)**2  # Power spectrum = phonon DOS

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 8))

ax1.plot(t[:500], vacf[:500], 'b-', linewidth=0.8)
ax1.set_xlabel('Time (a.u.)')
ax1.set_ylabel('VACF')
ax1.set_title('Velocity Autocorrelation Function')
ax1.grid(True, alpha=0.3)

ax2.fill_between(freqs, dos / dos.max(), alpha=0.3, color='red')
ax2.plot(freqs, dos / dos.max(), 'r-', linewidth=0.8)
ax2.set_xlabel('Frequency (a.u.)')
ax2.set_ylabel('Phonon DOS (normalized)')
ax2.set_title('Phonon Density of States (from FFT of VACF)')
ax2.set_xlim(0, 3000)
ax2.grid(True, alpha=0.3)

# Annotate peaks
ax2.annotate('Acoustic\nmodes', xy=(650, 0.6), fontsize=8, ha='center', color='blue')
ax2.annotate('Optical\nmodes', xy=(1750, 0.25), fontsize=8, ha='center', color='blue')

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


Velocity Autocorrelation Function
VACF
2
1
0
−1
0.00
0.01
0.02
0.03
0.04
0.05
Time (a.u.)


Phonon Density of States (from FFT of VACF)
Phonon DOS (normalized)
1.0
0.8
0.6
0.4
0.2
0.0
Acoustic
modes
Optical
modes
0
500
1000
1500
2000
2500
3000
Frequency (a.u.)

9.8.2 Application 2: Gravitational Wave — Chirp Signal

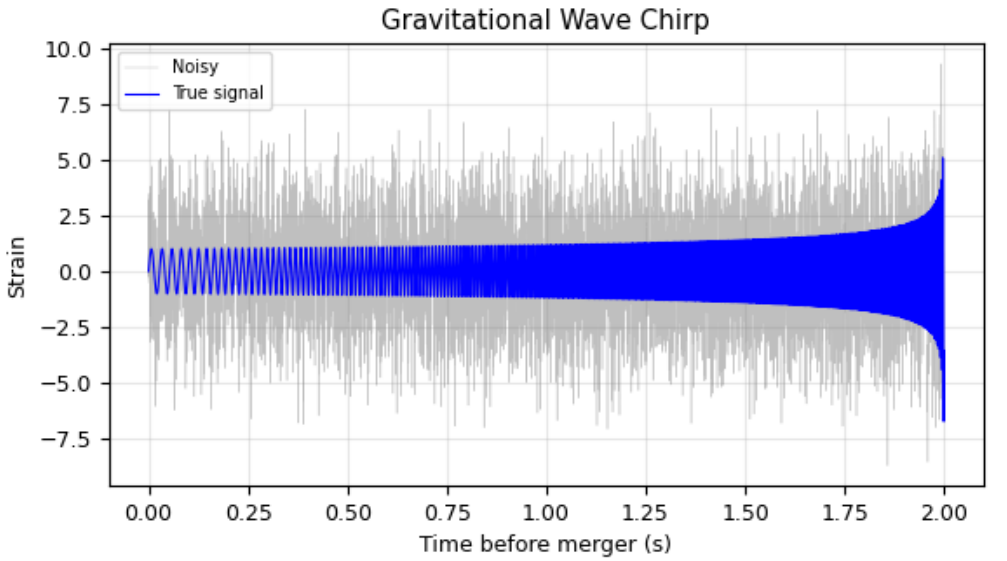
Gravitational waves from merging black holes produce a **chirp**: a signal whose frequency *increases* over time. We need the **Short-Time Fourier Transform (STFT)** to see how frequency evolves.

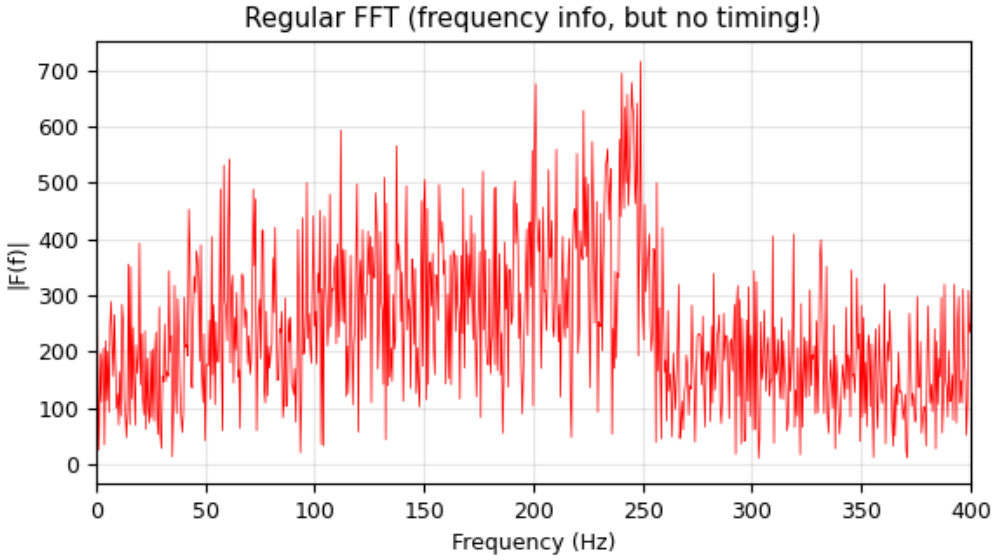
```
# Simulated gravitational wave chirp
fs = 4096  # Hz (typical GW detector sampling rate)
T = 2.0    # seconds before merger
N = int(fs * T)
t = np.linspace(0, T, N, endpoint=False)

# Chirp: frequency increases from 35 Hz to 250 Hz
f0 = 35   # starting frequency
f1 = 250  # ending frequency
# Phase: integral of instantaneous frequency
phase = 2 * np.pi * (f0 * t + (f1 - f0) / (2 * T) * t**2)
# Amplitude increases as merger approaches
amplitude = (1 - t / T)**(-0.25)
amplitude = np.clip(amplitude, 0, 10)  # prevent divergence

gw_signal = amplitude * np.sin(phase)

# Add detector noise
np.random.seed(42)
gw_noisy = gw_signal + 2 * np.random.randn(N)

# Spectrogram (STFT) — shows frequency vs time
freqs_stft, times_stft, Sxx = signal.spectrogram(
    gw_noisy, fs=fs, nperseg=256, noverlap=250, window='hann'
)

fig, axes = plt.subplots(3, 1, figsize=(6, 10))

# Time domain
axes[0].plot(t, gw_noisy, 'gray', linewidth=0.3, alpha=0.5, label='Noisy')
axes[0].plot(t, gw_signal, 'b-', linewidth=0.8, label='True signal')
axes[0].set_xlabel('Time before merger (s)')
axes[0].set_ylabel('Strain')
axes[0].set_title('Gravitational Wave Chirp')
axes[0].legend(fontsize=7)
axes[0].grid(True, alpha=0.3)

# Regular FFT (loses time information)
F_gw = np.fft.rfft(gw_noisy)
f_gw = np.fft.rfftfreq(N, d=1/fs)
axes[1].plot(f_gw, np.abs(F_gw), 'r-', linewidth=0.5)
axes[1].set_xlim(0, 400)
axes[1].set_xlabel('Frequency (Hz)')
axes[1].set_ylabel('|F(f)|')
axes[1].set_title('Regular FFT (frequency info, but no timing!)')
axes[1].grid(True, alpha=0.3)

# Spectrogram
axes[2].pcolormesh(times_stft, freqs_stft, 10 * np.log10(Sxx + 1e-10),
                   shading='gouraud', cmap='magma')
axes[2].set_ylim(0, 400)
axes[2].set_xlabel('Time (s)')
axes[2].set_ylabel('Frequency (Hz)')
```

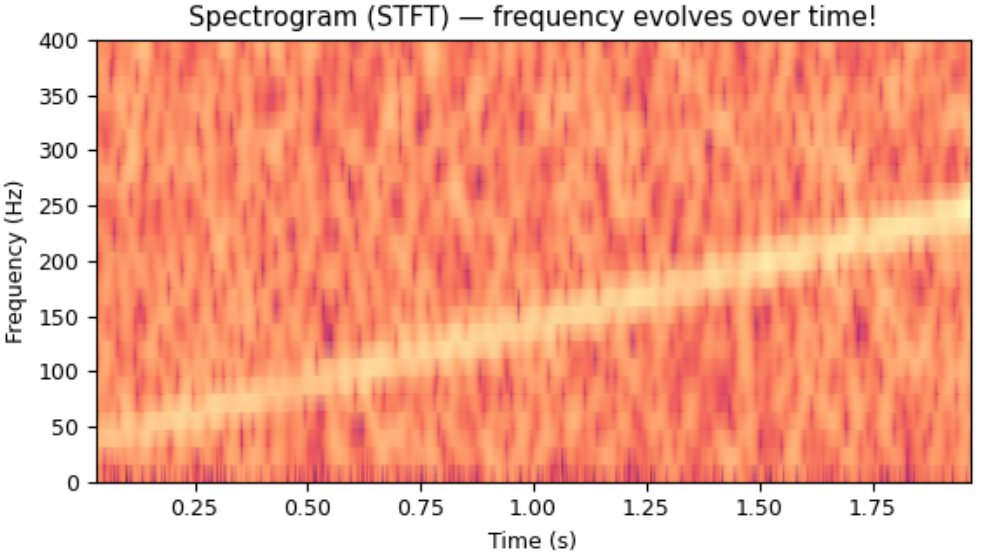
(continues on next page)

(continued from previous page)

```
axes[2].set_title('Spectrogram (STFT) - frequency evolves over time!')

plt.tight_layout()
plt.show()

print("The spectrogram reveals the chirp: frequency sweeping from ~35 to ~250 Hz!")
print("Regular FFT shows all frequencies present, but not WHEN they occur.")
```


Gravitational Wave Chirp
Noisy
True signal
Strain
Time before merger (s)


Regular FFT (frequency info, but no timing!)
|F(f)|
Frequency (Hz)


Spectrogram (STFT) — frequency evolves over time!
Frequency (Hz)
Time (s)

```
The spectrogram reveals the chirp: frequency sweeping from ~35 to ~250 Hz!
Regular FFT shows all frequencies present, but not WHEN they occur.
```

9.8.3 Application: Solving PDEs with FFT (Spectral Methods)

The FFT turns **derivatives into multiplication** in frequency space:

$$\mathcal{F}\left\{\frac{df}{dx}\right\} = 2\pi i k \cdot \hat{f}(k)$$

This makes solving differential equations **extremely fast —** known as **spectral methods**.

We'll cover PDEs in detail in our PDE lectures.

CHAPTER

TEN

LECTURE 10: ORDINARY DIFFERENTIAL EQUATIONS (ODES)

Computational Physics — Spring 2026

10.1 Why ODEs?

Almost every law of physics is a differential equation:

Physics	Equation	Type
Newton's 2nd law	$m\ddot{x} = F(x, \dot{x}, t)$	2nd order ODE
Radioactive decay	$dN/dt = -\lambda N$	1st order ODE
Damped oscillator	$m\ddot{x} + c\dot{x} + kx = 0$	2nd order ODE
Heat diffusion (1D)	$\dot{T} = \alpha\, T''$	PDE → system of ODEs
Crystal lattice vibrations	$m\ddot{u}_n = K(u_{n+1} - 2u_n + u_{n-1})$	System of ODEs

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

# Projector-friendly settings
plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

print("Ready!")
```

```
Ready!
```

10.2 I. What Is an ODE?

An **ordinary differential equation** relates a function to its derivatives:

$$\frac{dy}{dt} = f(t, y)$$

Given an **initial condition** $y(t_0) = y_0$, we want to find $y(t)$ for $t > t_0$.

Higher-order ODEs can always be rewritten as a **system of first-order** ODEs:

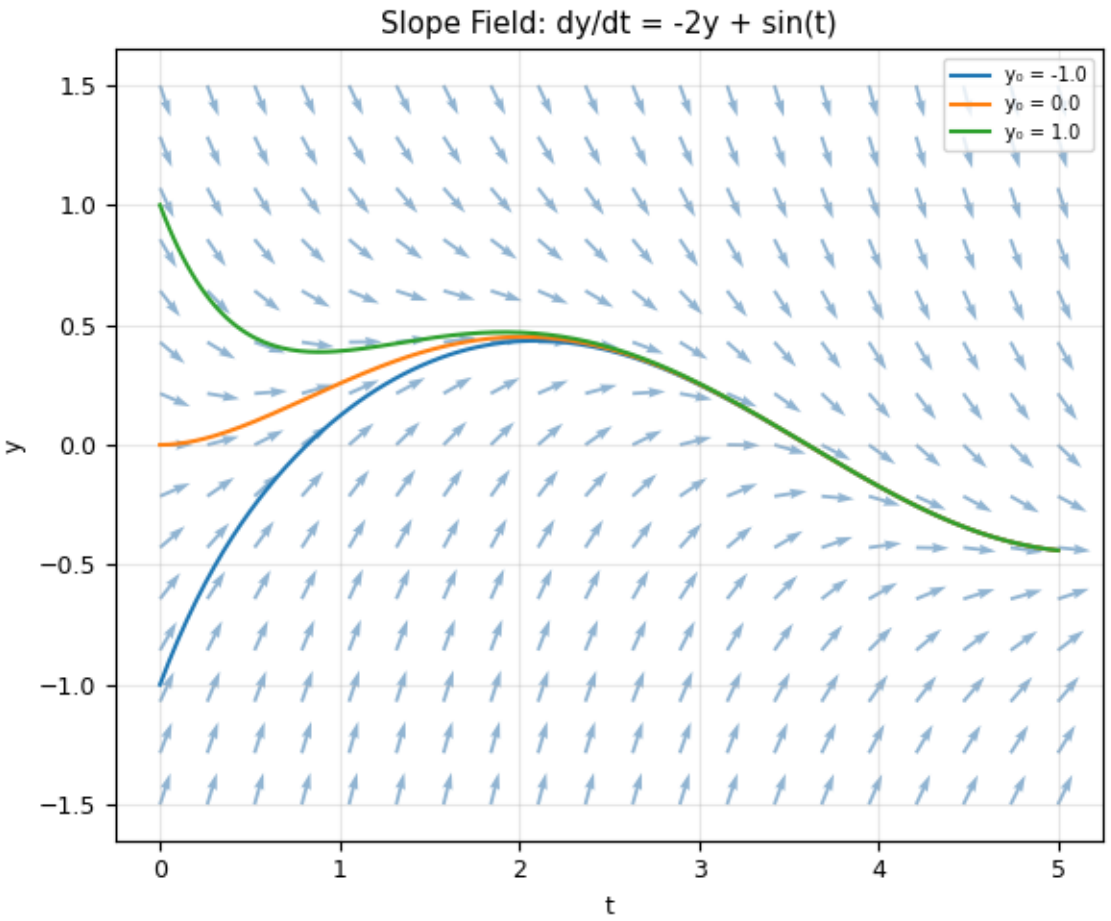
$$\ddot{x} = -\omega^2 x \quad \Longrightarrow \quad \begin{cases} \dot{x} = v \\ \dot{v} = -\omega^2 x \end{cases}$$

This is the standard form that all numerical solvers expect.

10.2.1 Slope Fields: Visualizing ODEs

Before solving, we can **visualize** the ODE. At every point (t, y), the derivative $f(t, y)$ gives the slope.

```
# Slope field for dy/dt = -2*y + sin(t)
t_grid = np.linspace(0, 5, 20)
y_grid = np.linspace(-1.5, 1.5, 15)
T, Y = np.meshgrid(t_grid, y_grid)

def f_example(t, y):
    return -2 * y + np.sin(t)

dY = f_example(T, Y)
dT = np.ones_like(dY)  # dt component = 1
# Normalize arrows
norm = np.sqrt(dT**2 + dY**2)
dT /= norm
dY /= norm

fig, ax = plt.subplots(figsize=(6, 5))
ax.quiver(T, Y, dT, dY, color='steelblue', alpha=0.6, scale=30)

# Solve and overlay a few trajectories
for y0 in [-1.0, 0.0, 1.0]:
    sol = solve_ivp(f_example, [0, 5], [y0], t_eval=np.linspace(0, 5, 200))
    ax.plot(sol.t, sol.y[0], linewidth=1.5, label=f'y₀ = {y0}')

ax.set_xlabel('t')
ax.set_ylabel('y')
ax.set_title("Slope Field: dy/dt = -2y + sin(t)")
ax.legend(fontsize=7)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("All trajectories follow the arrows and converge to the same attractor!")
```




```
All trajectories follow the arrows and converge to the same attractor!
```

10.3 II. The Euler Method

10.3.1 The Simplest Idea

Approximate the derivative with a finite difference:

$$\frac{dy}{dt} \approx \frac{y_{n+1} - y_n}{\Delta t} = f(t_n, y_n)$$

Rearranging:

$$\boxed{y_{n+1} = y_n + \Delta t \cdot f(t_n, y_n)}$$

- Start at (t_0, y_0)
- Compute slope $f(t_0, y_0)$
- Take a step: $y_1 = y_0 + \Delta t \cdot f(t_0, y_0)$
- Repeat

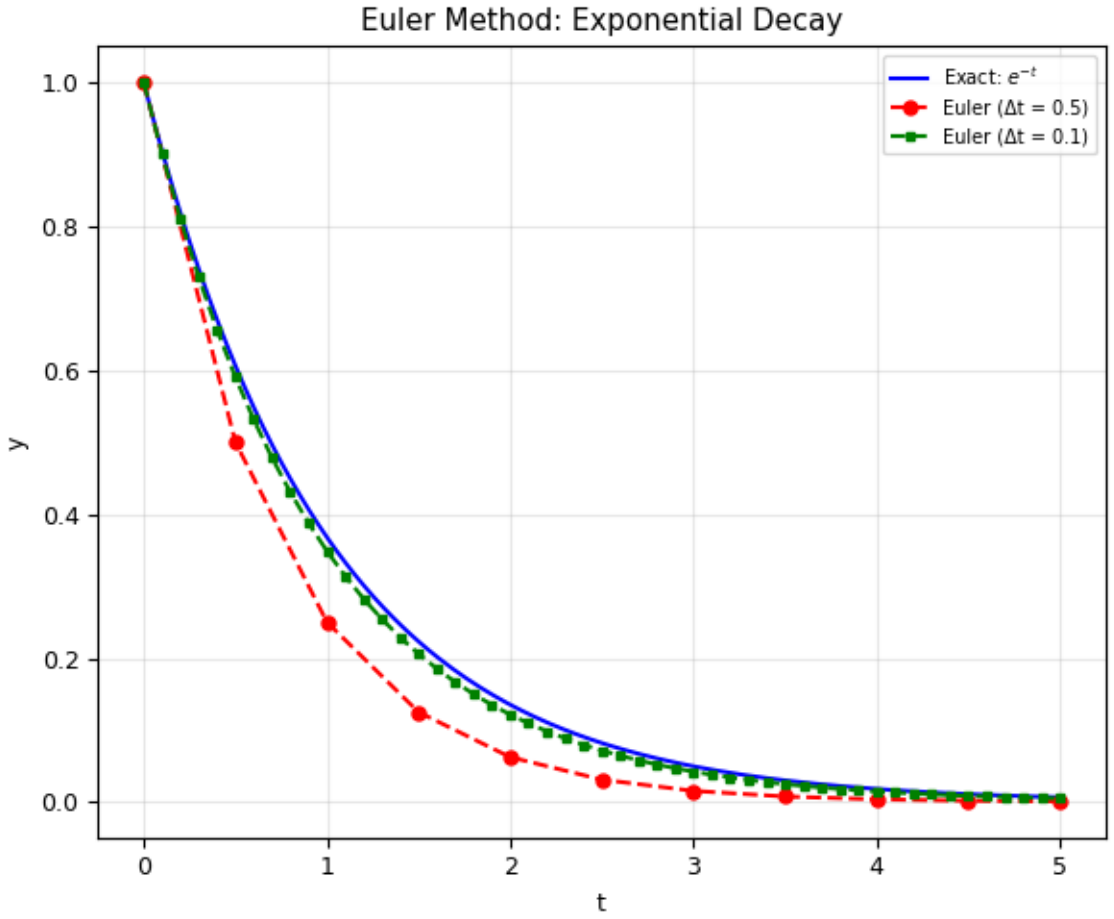
Error per step: $O(\Delta t^2)$ → **Global error:** $O(\Delta t)$ — first-order method.

```
def euler(f, t_span, y0, dt):
    """Solve dy/dt = f(t, y) using Euler's method.

    Parameters
    ----------
    f : callable
        Right-hand side function f(t, y).
    t_span : tuple
        (t_start, t_end).
    y0 : array_like
        Initial condition (scalar or array).
    dt : float
        Time step.

    Returns
    -------
    t : ndarray
        Time points.
    y : ndarray
        Solution at each time point.
    """
    t_start, t_end = t_span
    t = np.arange(t_start, t_end + dt/2, dt)
    y0 = np.atleast_1d(np.array(y0, dtype=float))
    y = np.zeros((len(t), len(y0)))
    y[0] = y0

    for i in range(len(t) - 1):
        # y[i+1] =
        # code
        y[i+1] = y[i] + dt * np.array(f(t[i], y[i]))

    return t, y

# Test: exponential decay dy/dt = -y, y(0) = 1
# Exact solution: y = e^(-t)
t_euler, y_euler = euler(lambda t, y: -y, [0, 5], [1.0], dt=0.5)
t_exact = np.linspace(0, 5, 200)
y_exact = np.exp(-t_exact)

fig, ax = plt.subplots(figsize=(6, 5))
ax.plot(t_exact, y_exact, 'b-', linewidth=1.5, label='Exact: $e^{-t}$')
ax.plot(t_euler, y_euler[:, 0], 'ro--', markersize=5, label='Euler (Δt = 0.5)')

# Also show smaller dt
t_e2, y_e2 = euler(lambda t, y: -y, [0, 5], [1.0], dt=0.1)
ax.plot(t_e2, y_e2[:, 0], 'gs--', markersize=3, label='Euler (Δt = 0.1)')

ax.set_xlabel('t')
ax.set_ylabel('y')
ax.set_title('Euler Method: Exponential Decay')
ax.legend(fontsize=7)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f"Error at t=5 (Δt=0.5): {abs(y_euler[-1, 0] - np.exp(-5)):.4f}")
```

(continues on next page)

(continued from previous page)

```
print(f"Error at t=5 (Δt=0.1): {abs(y_e2[-1, 0] - np.exp(-5)):.4f}")
print(f"Ratio: {abs(y_euler[-1, 0] - np.exp(-5)) / abs(y_e2[-1, 0] - np.exp(-5)):.1f}
↪x  (5x smaller Δt → ~5x smaller error)")
```




```
Error at t=5 (Δt=0.5): 0.0058
Error at t=5 (Δt=0.1): 0.0016
Ratio: 3.6x  (5x smaller Δt → ~5x smaller error)
```

10.3.2 Application: Falling with Air Drag

A falling object with quadratic drag:

$$m\dot{v} = mg - bv^2 \quad \Rightarrow \quad \dot{v} = g - \frac{b}{m}v^2$$

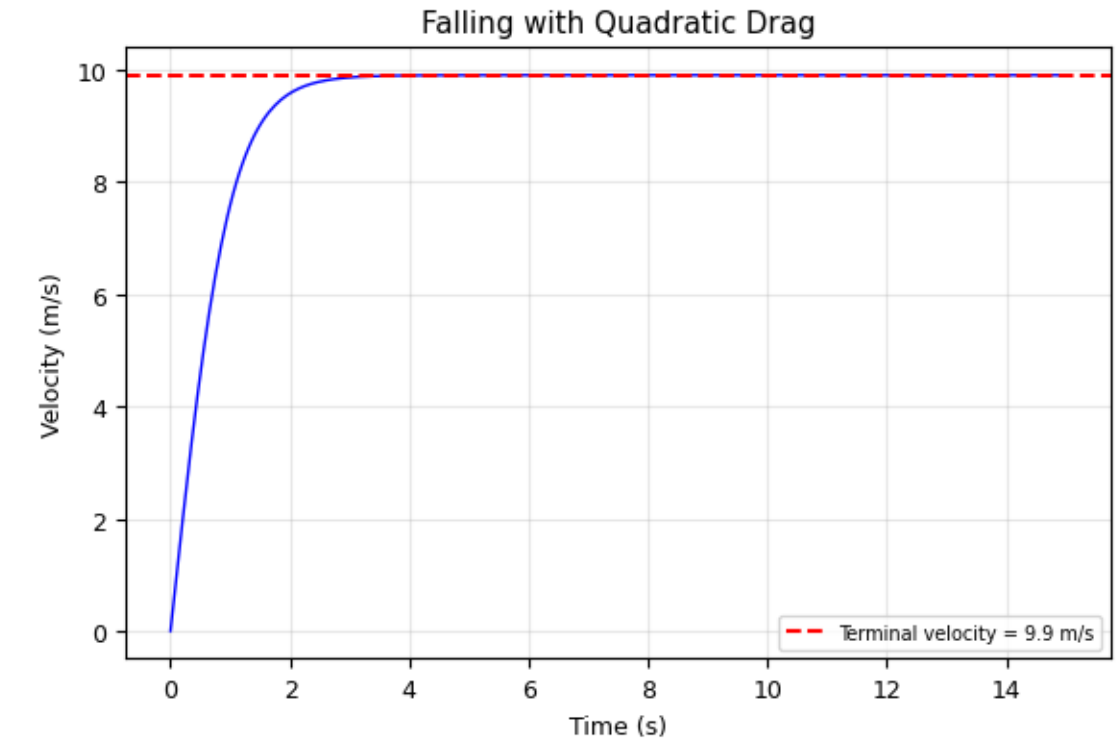
Terminal velocity: $v_T = \sqrt{mg/b}$ when $\dot{v} = 0$.

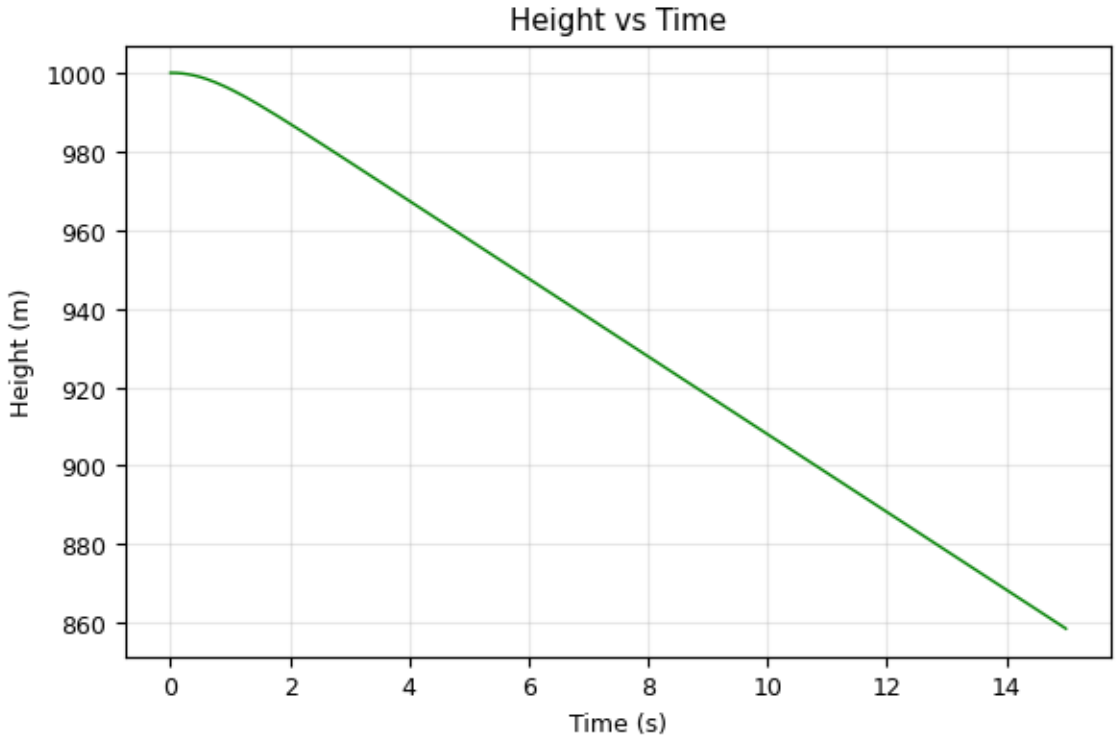
```
# Falling with quadratic drag
g = 9.81   # m/s^2
b_over_m = 0.1  # drag coefficient / mass
v_terminal = np.sqrt(g / b_over_m)
```

(continues on next page)

(continued from previous page)

```
def falling_drag(t, y):
    """y = [height, velocity]. Positive v = downward."""
    h, v = y
    dhdt = -v           # height decreases as we fall
    dvdt = g - b_over_m * v**2  # gravity - drag
    return [dhdt, dvdt]

# Solve with Euler
t_fall, y_fall = euler(falling_drag, [0, 15], [1000, 0], dt=0.05)

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 8))

ax1.plot(t_fall, y_fall[:, 1], 'b-', linewidth=1)
ax1.axhline(v_terminal, color='r', linestyle='--', label=f'Terminal velocity = {v_
↪terminal:.1f} m/s')
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Velocity (m/s)')
ax1.set_title('Falling with Quadratic Drag')
ax1.legend(fontsize=7)
ax1.grid(True, alpha=0.3)

ax2.plot(t_fall, y_fall[:, 0], 'g-', linewidth=1)
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('Height (m)')
ax2.set_title('Height vs Time')
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Velocity approaches terminal velocity v_T = {v_terminal:.1f} m/s")
print(f"After ~5 time constants: v ≈ v_T")
```






```
Velocity approaches terminal velocity v_T = 9.9 m/s
After ~5 time constants: v ≈ v_T
```

10.4 III. Euler Fails for Oscillators!

10.4.1 The Simple Harmonic Oscillator

$$\ddot{x} = -\omega^2 x \quad \Rightarrow \quad \begin{cases} \dot{x} = v \\ \dot{v} = -\omega^2 x \end{cases}$$

Energy should be conserved: $E = \frac{1}{2}v^2 + \frac{1}{2}\omega^2 x^2 = \text{const}$

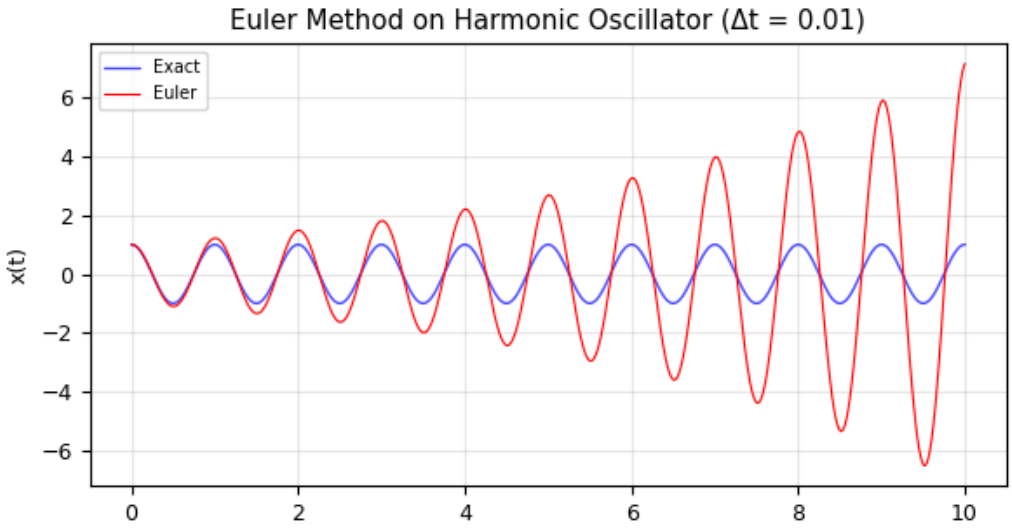
Let's see what Euler does…

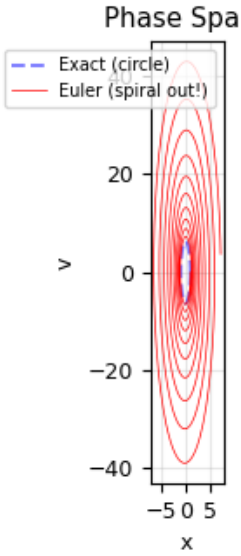
```
# Simple harmonic oscillator with Euler
omega = 2 * np.pi  # angular frequency (1 Hz)

def sho(t, y):
    """Simple harmonic oscillator: y = [x, v]."""
    x, v = y
    return [v, -omega**2 * x]

dt = 0.01
t_sho, y_sho = euler(sho, [0, 10], [1.0, 0.0], dt=dt)

# Exact solution
x_exact = np.cos(omega * t_sho)
energy = 0.5 * y_sho[:, 1]**2 + 0.5 * omega**2 * y_sho[:, 0]**2

fig, axes = plt.subplots(3, 1, figsize=(6, 10))

# Position
axes[0].plot(t_sho, x_exact, 'b-', linewidth=1, label='Exact', alpha=0.7)
axes[0].plot(t_sho, y_sho[:, 0], 'r-', linewidth=0.8, label='Euler')
axes[0].set_ylabel('x(t)')
axes[0].set_title(f'Euler Method on Harmonic Oscillator (Δt = {dt})')
axes[0].legend(fontsize=7)
axes[0].grid(True, alpha=0.3)

# Phase space
theta = np.linspace(0, 2*np.pi, 100)
axes[1].plot(np.cos(theta), -omega*np.sin(theta), 'b--', alpha=0.5, label='Exact␣
↪(circle)')
axes[1].plot(y_sho[:, 0], y_sho[:, 1], 'r-', linewidth=0.5, label='Euler (spiral out!)
↪')
axes[1].set_xlabel('x')
axes[1].set_ylabel('v')
axes[1].set_title('Phase Space')
axes[1].set_aspect('equal')
axes[1].legend(fontsize=7)
axes[1].grid(True, alpha=0.3)

# Energy
axes[2].plot(t_sho, energy / energy[0], 'r-', linewidth=0.8)
axes[2].axhline(1.0, color='b', linestyle='--', alpha=0.5, label='True energy␣
↪(constant)')
axes[2].set_xlabel('t')
axes[2].set_ylabel('E(t) / E(0)')
axes[2].set_title('Energy — Euler adds energy over time!')
axes[2].legend(fontsize=7)
```

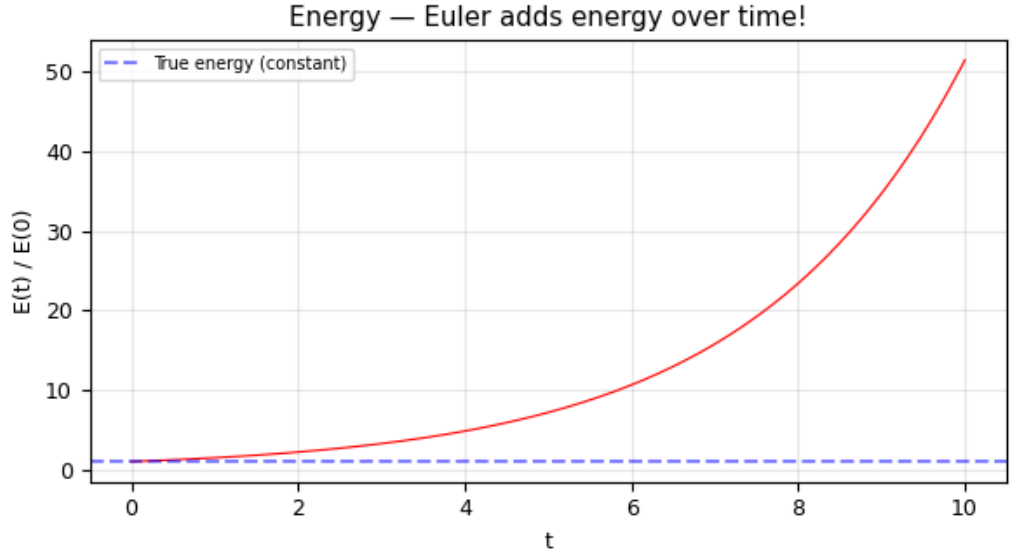
(continues on next page)

(continued from previous page)

```
axes[2].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Energy after 10 periods: {energy[-1]/energy[0]:.2f}x initial — Euler GAINS␣
 ↪energy!")
print("The phase-space orbit spirals OUTWARD. This is unphysical.")
```


Euler Method on Harmonic Oscillator (Δt = 0.01)
Exact
Euler
x(t)
6
4
2
0
−2
−4
−6
0
2
4
6
8
10


Phase Space
Exact (circle)
Euler (spiral out!)
v
20
0
−20
−40
−5 0 5
x


Energy — Euler adds energy over time!
True energy (constant)
E(t) / E(0)
50
40
30
20
10
0
0
2
4
6
8
10
t

```
Energy after 10 periods: 51.42x initial — Euler GAINS energy!
The phase-space orbit spirals OUTWARD. This is unphysical.
```

10.4.2 The Problem

Euler uses the **old position** to update both position and velocity simultaneously:

$$x_{n+1} = x_n + \Delta t \cdot v_n$$

$$v_{n+1} = v_n + \Delta t \cdot a(x_n)$$

Both updates use data from time t_n. For oscillators, this systematically adds energy at every step.

10.5 IV. The Euler-Cromer (Symplectic) Method

Simple fix: update velocity first, then use the **new velocity** to update position:

$$v_{n+1} = v_n + \Delta t \cdot a(x_n)$$

$$x_{n+1} = x_n + \Delta t \cdot v_{n+1} \quad \leftarrow \text{uses updated } v$$

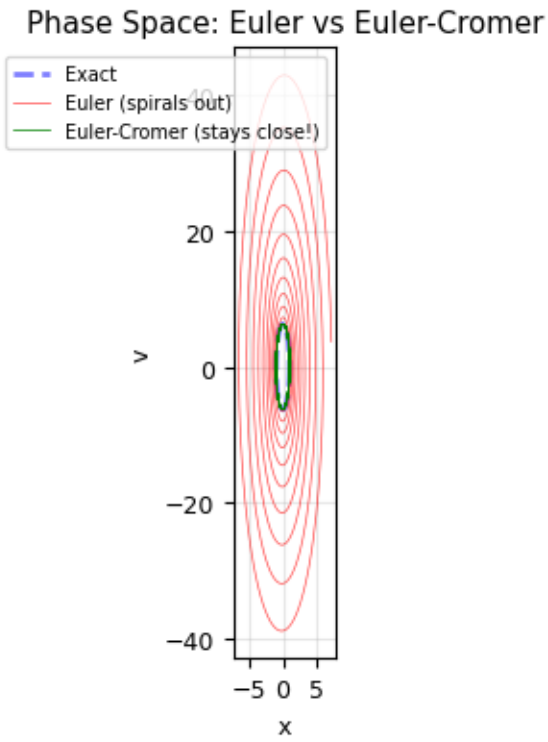
This is a **symplectic integrator** — it preserves the geometric structure of Hamiltonian mechanics. Energy oscillates around the true value instead of drifting.

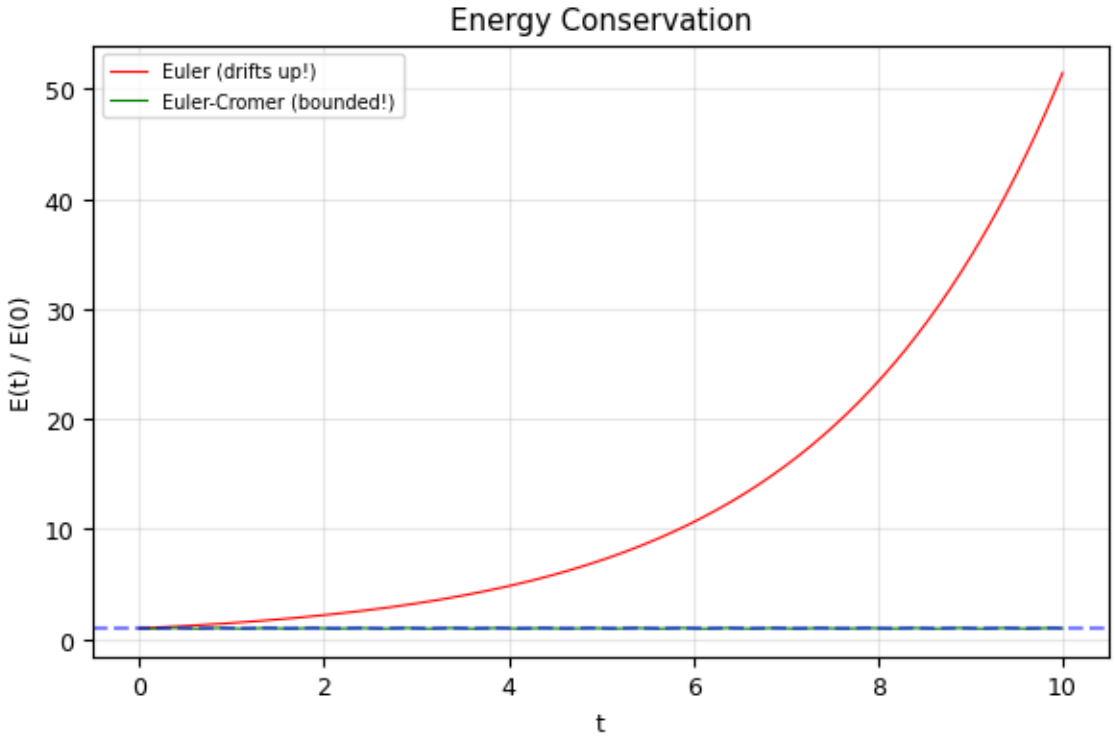
```python
def euler_cromer(t_span, x0, v0, a_func, dt):
    """Solve x'' = a(x, v, t) using the Euler-Cromer (symplectic) method.

    Parameters
    ----------
    t_span : tuple
        (t_start, t_end).
    x0, v0 : float
        Initial position and velocity.
    a_func : callable
        Acceleration function a(x, v, t).
    dt : float
        Time step.

    Returns
    -------
    t, x, v : ndarrays
    """
    t = np.arange(t_span[0], t_span[1] + dt/2, dt)
    x = np.zeros(len(t))
    v = np.zeros(len(t))
    x[0], v[0] = x0, v0

    for i in range(len(t) - 1):
        v[i+1] = v[i] + dt * a_func(x[i], v[i], t[i])       # Update v FIRST
        x[i+1] = x[i] + dt * v[i+1]                  # Use NEW v

    return t, x, v
```

(continues on next page)

(continued from previous page)

```
# Compare Euler vs Euler-Cromer on SHO
a_sho = lambda x, v, t: -omega**2 * x

t_ec, x_ec, v_ec = euler_cromer([0, 10], 1.0, 0.0, a_sho, dt=0.01)
energy_ec = 0.5 * v_ec**2 + 0.5 * omega**2 * x_ec**2

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 8))

# Phase space comparison
ax1.plot(np.cos(theta), -omega*np.sin(theta), 'b--', alpha=0.5, linewidth=2, label=
 ↪'Exact')
ax1.plot(y_sho[:, 0], y_sho[:, 1], 'r-', linewidth=0.5, alpha=0.7, label='Euler␣
 ↪(spirals out)')
ax1.plot(x_ec, v_ec, 'g-', linewidth=0.5, label='Euler-Cromer (stays close!)')
ax1.set_xlabel('x')
ax1.set_ylabel('v')
ax1.set_title('Phase Space: Euler vs Euler-Cromer')
ax1.set_aspect('equal')
ax1.legend(fontsize=7)
ax1.grid(True, alpha=0.3)

# Energy comparison
ax2.plot(t_sho, energy / energy[0], 'r-', linewidth=0.8, label='Euler (drifts up!)')
ax2.plot(t_ec, energy_ec / energy_ec[0], 'g-', linewidth=0.8, label='Euler-Cromer␣
 ↪(bounded!)')
ax2.axhline(1.0, color='b', linestyle='--', alpha=0.5)
ax2.set_xlabel('t')
ax2.set_ylabel('E(t) / E(0)')
ax2.set_title('Energy Conservation')
ax2.legend(fontsize=7)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Euler energy drift after 10 periods: {energy[-1]/energy[0]:.4f}")
print(f"Euler-Cromer energy after 10 periods: {energy_ec[-1]/energy_ec[0]:.6f}")
print("\nEuler-Cromer oscillates around E=1 but never drifts — symplectic!")
```






```
Euler energy drift after 10 periods: 51.4222
Euler-Cromer energy after 10 periods: 0.999350

Euler-Cromer oscillates around E=1 but never drifts — symplectic!
```

10.5.1 Why Symplectic Matters

Method	Energy behavior	Best for
Euler	Drifts (grows)	Short simulations, dissipative systems
Euler-Cromer	Bounded oscillation	Long-time Hamiltonian dynamics
Verlet / Leapfrog	Bounded, 2nd order	Molecular dynamics (we'll see later)

Molecular dynamics simulations of millions of atoms for nanoseconds **require** symplectic integrators.

10.6 V. The Runge-Kutta Method (RK4)

Euler uses the slope at **one point**. RK4 samples the slope at **four points** per step:

$$k_1 = f(t_n, y_n)$$

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

$$k_3 = f\left(t_n + \frac{\Delta t}{2},\ y_n + \frac{\Delta t}{2} k_2\right)$$

$$k_4 = f(t_n + \Delta t,\ y_n + \Delta t\, k_3)$$

$$\boxed{y_{n+1} = y_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)}$$

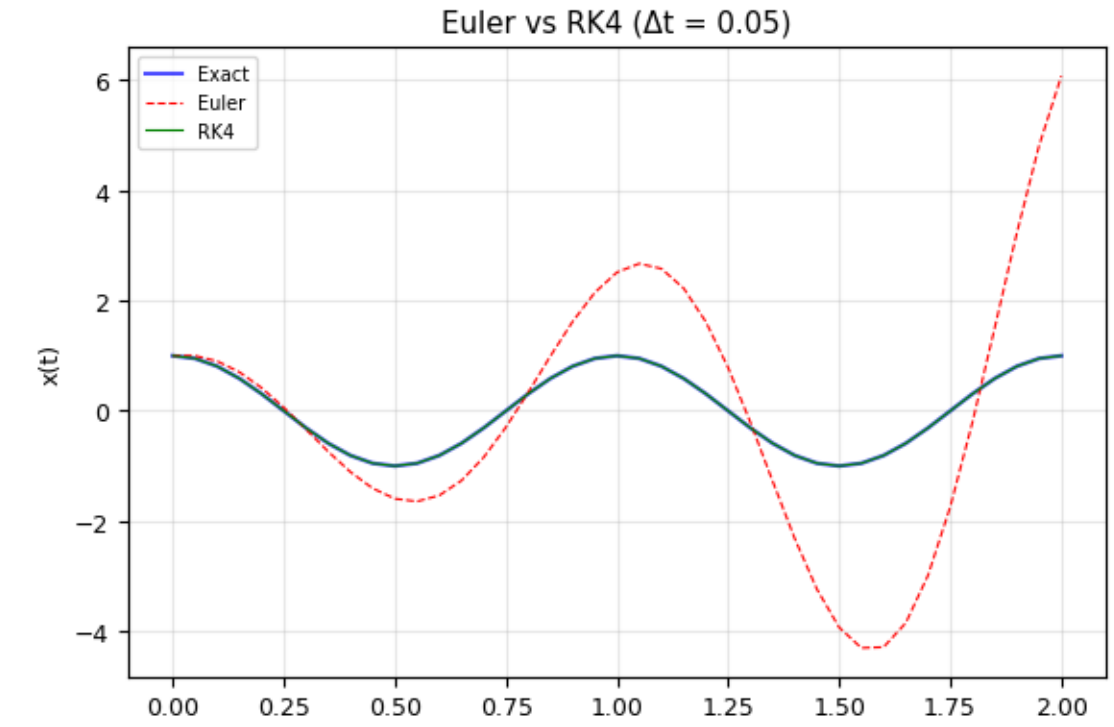
Global error: $O(\Delta t^4)$ — fourth-order method. 4× the work per step, but **much** more accurate.

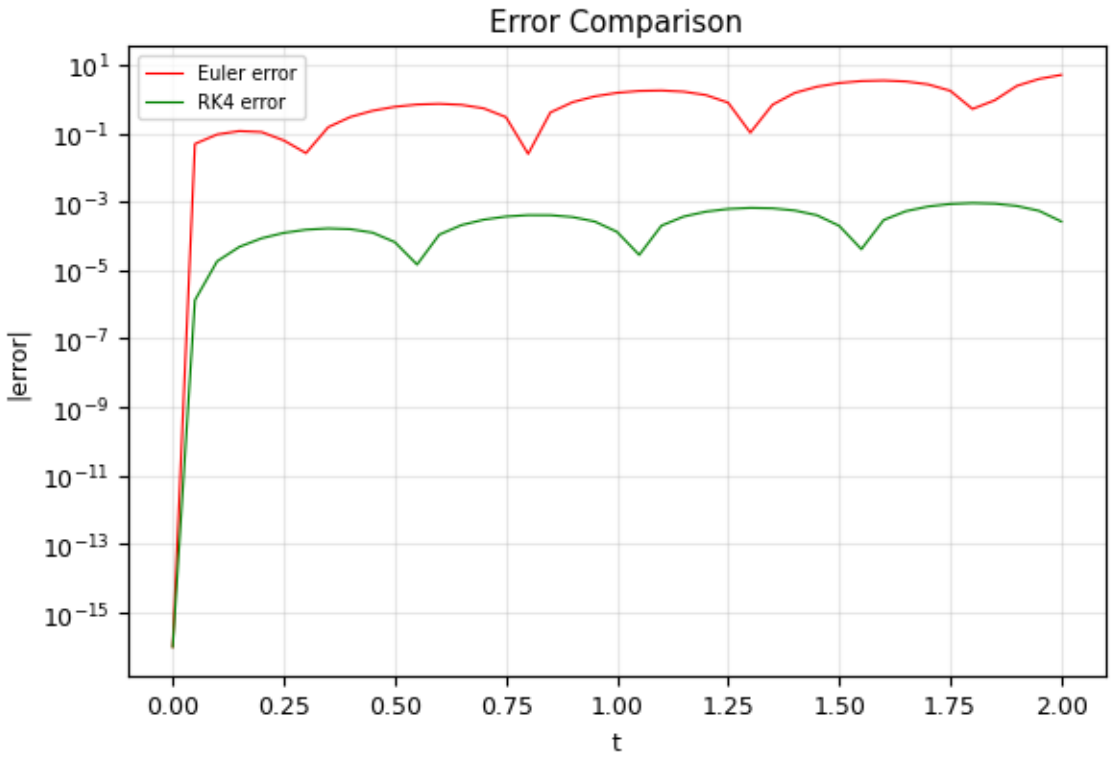
```python
def rk4(f, t_span, y0, dt):
    """Solve dy/dt = f(t, y) using the classic Runge-Kutta 4th order method.

    Parameters
    ----------
    f : callable
        Right-hand side f(t, y).
    t_span : tuple
        (t_start, t_end).
    y0 : array_like
        Initial condition.
    dt : float
        Time step.

    Returns
    -------
    t, y : ndarrays
    """
    t_start, t_end = t_span
    t = np.arange(t_start, t_end + dt/2, dt)
    y0 = np.atleast_1d(np.array(y0, dtype=float))
    y = np.zeros((len(t), len(y0)))
    y[0] = y0

    for i in range(len(t) - 1):
```

(continues on next page)

(continued from previous page)

```
        k1 = np.array(f(t[i], y[i]))
        k2 = np.array(f(t[i] + dt/2, y[i] + dt/2 * k1))
        k3 = np.array(f(t[i] + dt/2, y[i] + dt/2 * k2))
        k4 = np.array(f(t[i] + dt, y[i] + dt * k3))
        y[i+1] = y[i] + dt/6 * (k1 + 2*k2 + 2*k3 + k4)

    return t, y

# Compare Euler vs RK4 on the SHO
dt_compare = 0.05  # Larger step to show the difference

t_e, y_e = euler(sho, [0, 2], [1.0, 0.0], dt=dt_compare)
t_r, y_r = rk4(sho, [0, 2], [1.0, 0.0], dt=dt_compare)
x_exact_c = np.cos(omega * t_e)

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 8))

ax1.plot(t_e, x_exact_c, 'b-', linewidth=1.5, label='Exact', alpha=0.7)
ax1.plot(t_e, y_e[:, 0], 'r--', linewidth=0.8, label='Euler')
ax1.plot(t_r, y_r[:, 0], 'g-', linewidth=0.8, label='RK4')
ax1.set_ylabel('x(t)')
ax1.set_title(f'Euler vs RK4 (Δt = {dt_compare})')
ax1.legend(fontsize=7)
ax1.grid(True, alpha=0.3)

# Error comparison
err_euler = np.abs(y_e[:, 0] - np.cos(omega * t_e))
err_rk4 = np.abs(y_r[:, 0] - np.cos(omega * t_r))

ax2.semilogy(t_e, err_euler + 1e-16, 'r-', linewidth=0.8, label='Euler error')
ax2.semilogy(t_r, err_rk4 + 1e-16, 'g-', linewidth=0.8, label='RK4 error')
ax2.set_xlabel('t')
ax2.set_ylabel('|error|')
ax2.set_title('Error Comparison')
ax2.legend(fontsize=7)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Euler error at t=10: {err_euler[-1]:.4f}")
print(f"RK4 error at t=10:   {err_rk4[-1]:.2e}")
print(f"RK4 is ~{err_euler[-1]/max(err_rk4[-1], 1e-16):.0f}× more accurate with the␣
↪same Δt!")
```






```
Euler error at t=10: 5.0751
RK4 error at t=10:   2.64e-04
RK4 is ~19209× more accurate with the same Δt!
```

10.6.1 Method Comparison Summary

Method	Order	Error per step	Evaluations per step	Best for
Euler	1st	$O(\Delta t^2)$	1	Learning, simple problems
Euler-Cromer	1st	$O(\Delta t^2)$	1	Long-time Hamiltonian dynamics
RK4	4th	$O(\Delta t^5)$	4	General-purpose ODE solving
RK45 (adaptive)	4th–5th	Automatic	6	Production code (`solve_ivp`)

10.7 VI. The Professional Tool: `scipy.integrate.solve_ivp`

In practice, we use scipy's adaptive solver. It automatically:

- Adjusts the step size for accuracy
- Uses embedded error estimates (RK45: 4th and 5th order)
- Handles stiff equations (with `method='Radau'` or `'BDF'`)

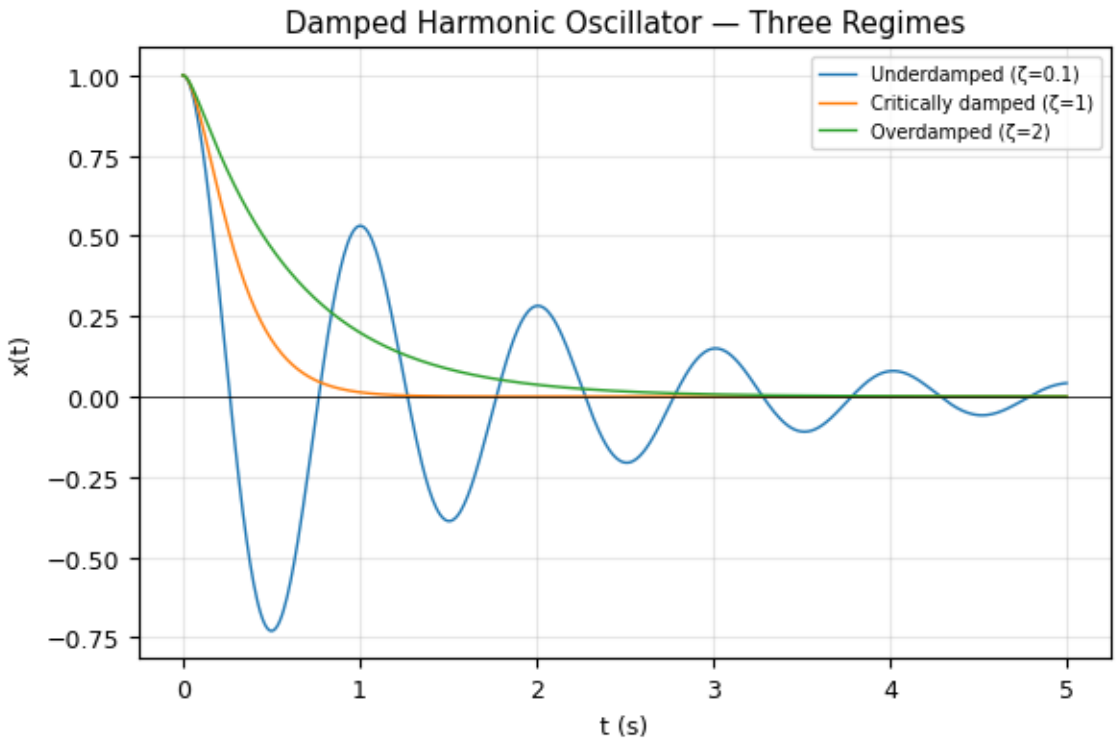
```
sol = solve_ivp(f, t_span, y0, t_eval=t_points, method='RK45')
## sol.t = time points
## sol.y = solution (shape: n_variables × n_timepoints)
```

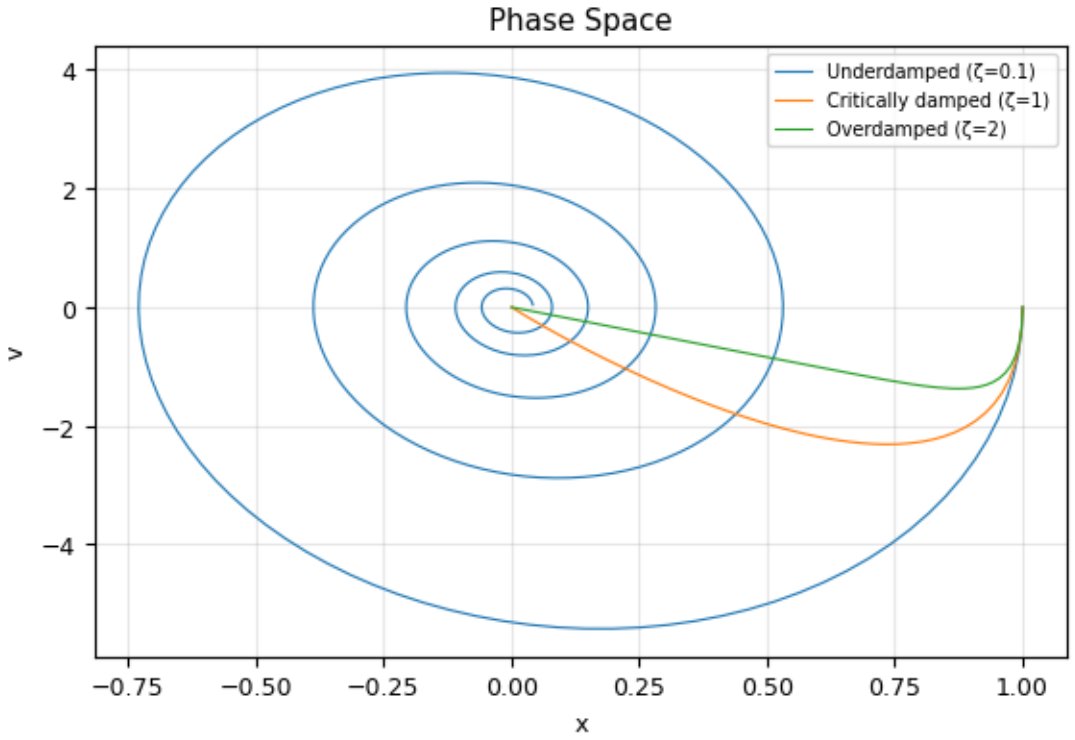
```
# Damped harmonic oscillator with solve_ivp
# m*x'' + c*x' + k*x = 0  →  x'' = -(c/m)*v - (k/m)*x

def damped_oscillator(t, y, gamma, omega0):
    """Damped harmonic oscillator.

    Parameters
    ----------
    gamma : float
        Damping coefficient c/(2m).
    omega0 : float
        Natural frequency sqrt(k/m).
    """
    x, v = y
    return [v, -2*gamma*v - omega0**2 * x]

omega0 = 2 * np.pi  # natural frequency
t_eval = np.linspace(0, 5, 500)

# Three damping regimes
cases = {
    'Underdamped (ζ=0.1)': 0.1 * omega0,
    'Critically damped (ζ=1)': 1.0 * omega0,
    'Overdamped (ζ=2)': 2.0 * omega0
}

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 8))

for label, gamma in cases.items():
    sol = solve_ivp(damped_oscillator, [0, 5], [1.0, 0.0],
                    t_eval=t_eval, args=(gamma, omega0))
    ax1.plot(sol.t, sol.y[0], linewidth=1, label=label)
```

(continues on next page)

(continued from previous page)

```
    ax2.plot(sol.y[0], sol.y[1], linewidth=0.8, label=label)

ax1.set_xlabel('t (s)')
ax1.set_ylabel('x(t)')
ax1.set_title('Damped Harmonic Oscillator — Three Regimes')
ax1.axhline(0, color='k', linewidth=0.5)
ax1.legend(fontsize=7)
ax1.grid(True, alpha=0.3)

ax2.set_xlabel('x')
ax2.set_ylabel('v')
ax2.set_title('Phase Space')
ax2.legend(fontsize=7)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("ζ < 1: Underdamped — oscillates with decaying envelope")
print("ζ = 1: Critically damped — fastest return to equilibrium without oscillation")
print("ζ > 1: Overdamped — slow return, no oscillation")
```






```
ζ < 1: Underdamped — oscillates with decaying envelope
ζ = 1: Critically damped — fastest return to equilibrium without oscillation
ζ > 1: Overdamped — slow return, no oscillation
```

10.8 VII. Physics Application: Phonons in a 1D Crystal

10.8.1 The Monatomic Chain

A chain of N atoms connected by springs (nearest-neighbor interaction):

$$m\ddot{u}_n = K(u_{n+1} - 2u_n + u_{n-1})$$

where u_n is the displacement of atom n from equilibrium.

This is a system of N coupled ODEs — a perfect test for our methods!

10.8.2 Analytical Dispersion Relation

$$\omega(k) = 2\sqrt{\frac{K}{m}}\left|\sin\left(\frac{ka}{2}\right)\right|$$

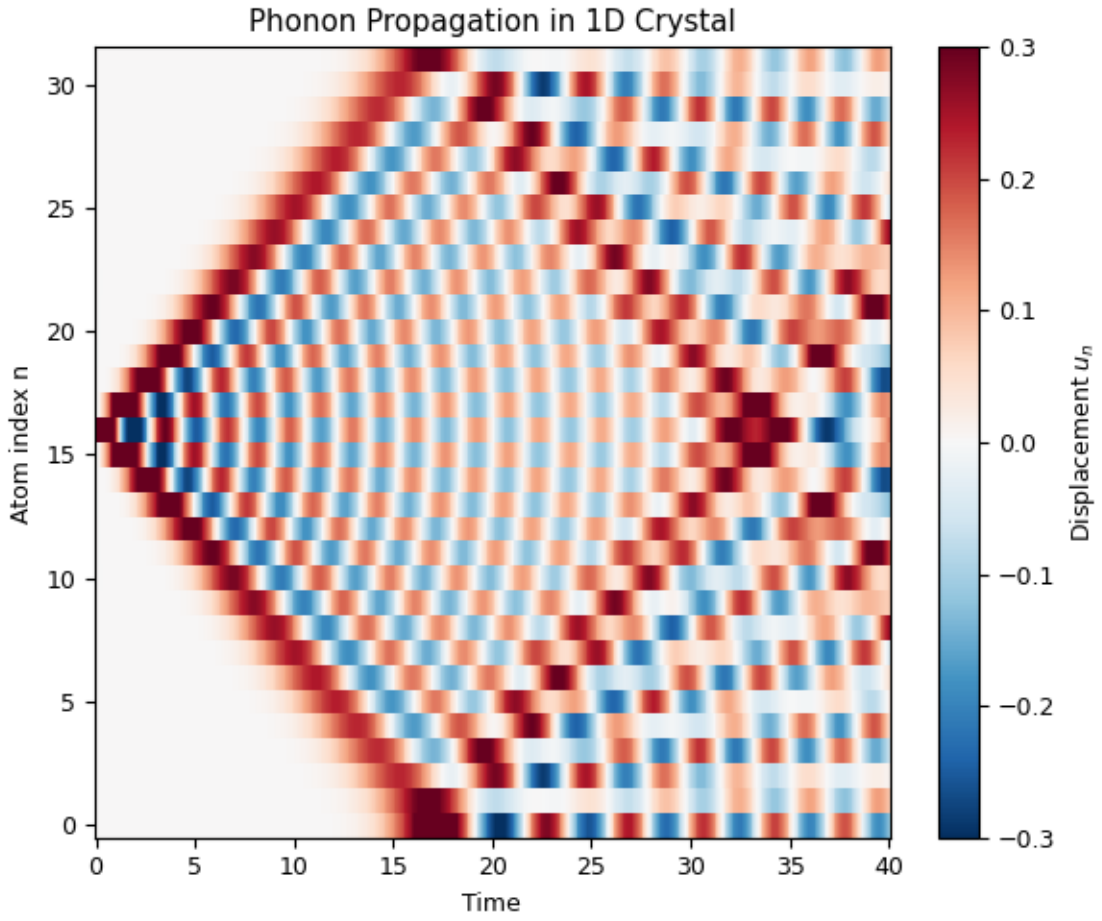
where a is the lattice spacing and k is the wavevector.

```
# 1D monatomic chain — phonon simulation
N_atoms = 32      # Number of atoms
K = 1.0           # Spring constant
m = 1.0           # Atom mass
a = 1.0           # Lattice spacing

def monatomic_chain(t, y):
    """Equations of motion for 1D monatomic chain.

    y = [u_0, u_1, ..., u_{N-1}, v_0, v_1, ..., v_{N-1}]
    Periodic boundary conditions.
    """
    u = y[:N_atoms]       # Displacements
    v = y[N_atoms:]       # Velocities

    # Acceleration: a_n = (K/m) * (u_{n+1} - 2*u_n + u_{n-1})
    # Periodic BC: u[-1] = u[N-1], u[N] = u[0]
    u_right = np.roll(u, -1)  # u_{n+1}
    u_left = np.roll(u, 1)    # u_{n-1}
    acc = (K / m) * (u_right - 2*u + u_left)

    return np.concatenate([v, acc])

# Initial condition: single atom displaced (impulse)
u0 = np.zeros(N_atoms)
u0[N_atoms // 2] = 1.0  # Displace the middle atom
v0 = np.zeros(N_atoms)
y0_chain = np.concatenate([u0, v0])

# Solve
T_sim = 40
t_eval_chain = np.linspace(0, T_sim, 500)
sol_chain = solve_ivp(monatomic_chain, [0, T_sim], y0_chain,
                      t_eval=t_eval_chain, method='RK45', rtol=1e-8)

# Extract displacements
U = sol_chain.y[:N_atoms, :]
```

(continues on next page)

(continued from previous page)

```
# Plot as heatmap: atom index vs time
fig, ax = plt.subplots(figsize=(6, 5))
im = ax.pcolormesh(sol_chain.t, np.arange(N_atoms), U,
                   cmap='RdBu_r', shading='auto', vmin=-0.3, vmax=0.3)
plt.colorbar(im, ax=ax, label='Displacement $u_n$')
ax.set_xlabel('Time')
ax.set_ylabel('Atom index n')
ax.set_title('Phonon Propagation in 1D Crystal')
plt.tight_layout()
plt.show()

print("A single displaced atom → waves propagate outward in both directions!")
print("Periodic BC: waves wrap around and interfere.")
```




```
A single displaced atom → waves propagate outward in both directions!
Periodic BC: waves wrap around and interfere.
```

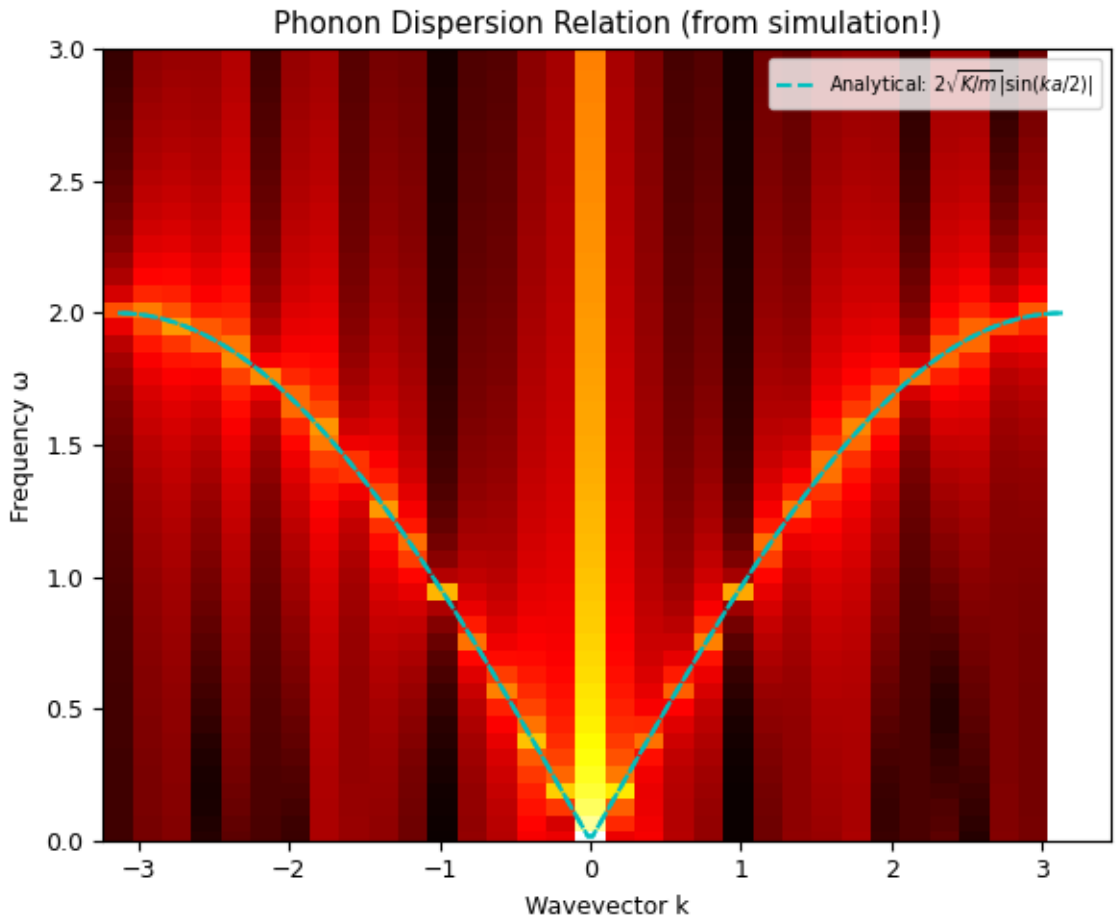
```
# Extract the dispersion relation from the simulation using 2D FFT!
# FFT in space → wavevector k
# FFT in time → frequency ω

# Use a longer, finer simulation for better resolution
```

(continues on next page)

(continued from previous page)

```
T_long = 100
N_time = 2048
t_fine = np.linspace(0, T_long, N_time)

# Random initial displacements to excite all modes
np.random.seed(42)
u0_rand = 0.1 * np.random.randn(N_atoms)
v0_rand = 0.1 * np.random.randn(N_atoms)
y0_rand = np.concatenate([u0_rand, v0_rand])

sol_long = solve_ivp(monatomic_chain, [0, T_long], y0_rand,
                     t_eval=t_fine, method='RK45', rtol=1e-8)
U_long = sol_long.y[:N_atoms, :]

# 2D FFT: space-time → (k, ω)
F_2d = np.fft.fft2(U_long)
power = np.abs(np.fft.fftshift(F_2d))**2

# Frequency and wavevector axes
dk = 2 * np.pi / (N_atoms * a)
k_axis = np.fft.fftshift(np.fft.fftfreq(N_atoms, d=a)) * 2 * np.pi
dt_sim = T_long / N_time
omega_axis = np.fft.fftshift(np.fft.fftfreq(N_time, d=dt_sim)) * 2 * np.pi

# Analytical dispersion
k_theory = np.linspace(-np.pi/a, np.pi/a, 200)
omega_theory = 2 * np.sqrt(K/m) * np.abs(np.sin(k_theory * a / 2))

fig, ax = plt.subplots(figsize=(6, 5))
# Only show positive frequencies
omega_pos = omega_axis >= 0
ax.pcolormesh(k_axis, omega_axis[omega_pos],
              np.log10(power[:, omega_pos].T + 1),
              cmap='hot', shading='auto')

ax.plot(k_theory, omega_theory, 'c--', linewidth=1.5, label='Analytical: $2\sqrt{K/m}
↪|\sin(ka/2)|$')
ax.plot(-k_theory, omega_theory, 'c--', linewidth=1.5)

ax.set_xlabel('Wavevector k')
ax.set_ylabel('Frequency ω')
ax.set_title('Phonon Dispersion Relation (from simulation!)')
ax.set_ylim(0, 3)
ax.legend(fontsize=7)
plt.tight_layout()
plt.show()

print("Bright bands = normal modes of the crystal.")
print("Cyan dashed = analytical dispersion relation.")
print("They match! We recovered the phonon spectrum from a simulation.")
```

```
<>:41: SyntaxWarning: invalid escape sequence '\s'
<>:41: SyntaxWarning: invalid escape sequence '\s'
/tmp/ipython-input-432/3103749938.py:41: SyntaxWarning: invalid escape sequence '\s
↪'
  ax.plot(k_theory, omega_theory, 'c--', linewidth=1.5, label='Analytical: $2\sqrt
↪{K/m}|\sin(ka/2)|$')
```




```
Bright bands = normal modes of the crystal.
Cyan dashed = analytical dispersion relation.
They match! We recovered the phonon spectrum from a simulation.
```

10.9 VIII. Physics Application: Diatomic Chain (Optical & Acoustic Phonons)

A chain with **two different atom masses** (m_1, m_2) alternating:

$$m_1 \ddot{u}_{2n} = K(u_{2n+1} - 2u_{2n} + u_{2n-1})$$

$$m_2 \ddot{u}_{2n+1} = K(u_{2n+2} - 2u_{2n+1} + u_{2n})$$

This creates **two branches**:

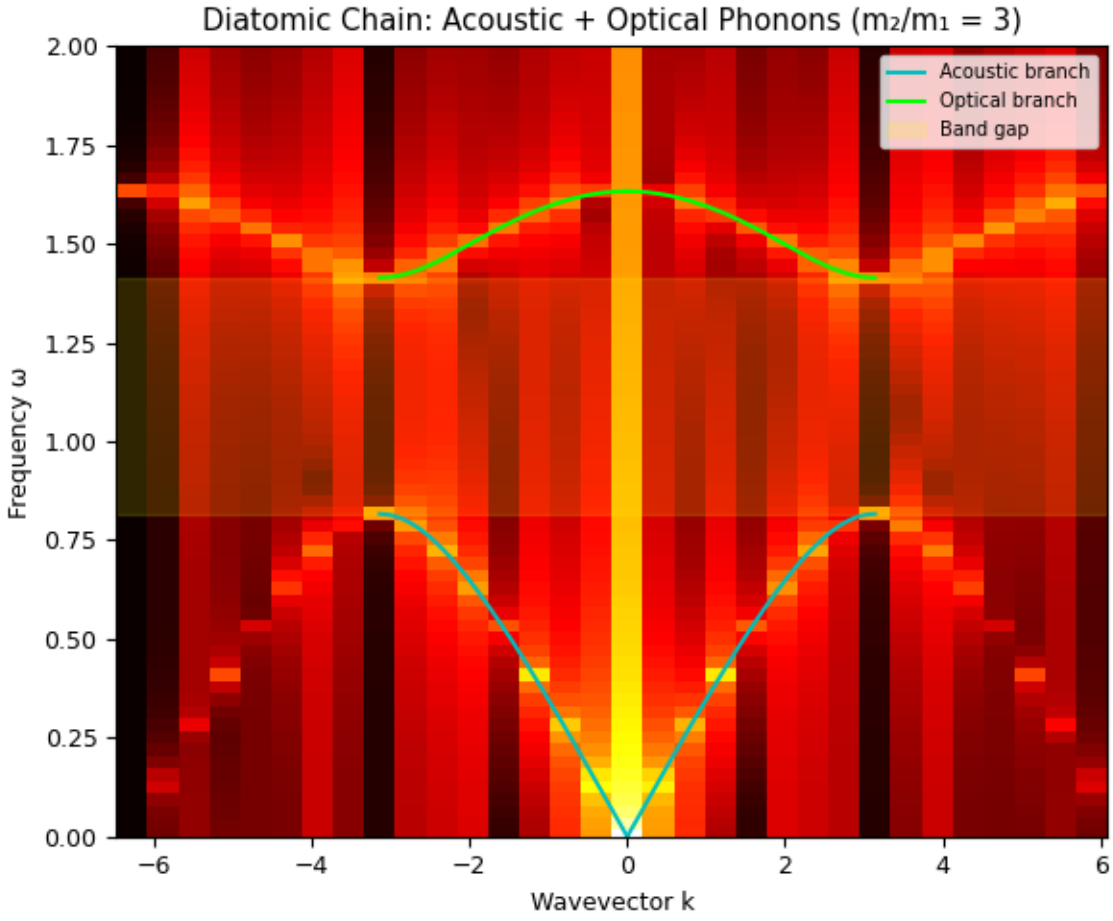
- **Acoustic branch** (low ω): atoms move in phase
- **Optical branch** (high ω): atoms move out of phase
- **Band gap** between them: no propagating modes!

```
# Diatomic chain simulation
N_cells = 16          # Unit cells (total 32 atoms)
N_total = 2 * N_cells
m1 = 1.0              # Light atom
m2 = 3.0              # Heavy atom
K_di = 1.0

# Mass array: alternating m1, m2, m1, m2, ...
masses = np.zeros(N_total)
masses[0::2] = m1
masses[1::2] = m2

def diatomic_chain(t, y):
    """Equations of motion for 1D diatomic chain with periodic BC."""
    u = y[:N_total]
    v = y[N_total:]
    u_right = np.roll(u, -1)
    u_left = np.roll(u, 1)
    acc = (K_di / masses) * (u_right - 2*u + u_left)
    return np.concatenate([v, acc])

# Random initial conditions to excite all modes
np.random.seed(123)
u0_di = 0.1 * np.random.randn(N_total)
v0_di = 0.1 * np.random.randn(N_total)
y0_di = np.concatenate([u0_di, v0_di])

T_di = 200
N_time_di = 4096
t_di = np.linspace(0, T_di, N_time_di)
sol_di = solve_ivp(diatomic_chain, [0, T_di], y0_di,
                   t_eval=t_di, method='RK45', rtol=1e-8)
U_di = sol_di.y[:N_total, :]

# 2D FFT to get dispersion
F_di = np.fft.fft2(U_di)
power_di = np.abs(np.fft.fftshift(F_di))**2

k_di_axis = np.fft.fftshift(np.fft.fftfreq(N_total, d=a/2)) * 2 * np.pi
dt_di = T_di / N_time_di
omega_di_axis = np.fft.fftshift(np.fft.fftfreq(N_time_di, d=dt_di)) * 2 * np.pi

# Analytical dispersion for diatomic chain
k_th = np.linspace(0, np.pi / a, 200)
# ω² = K(1/m1 + 1/m2) ± K*sqrt((1/m1+1/m2)² - 4sin²(ka/2)/(m1*m2))
sum_inv = 1/m1 + 1/m2
omega_sq_plus = K_di * (sum_inv + np.sqrt(sum_inv**2 - 4*np.sin(k_th*a/2)**2/(m1*m2)))
omega_sq_minus = K_di * (sum_inv - np.sqrt(sum_inv**2 - 4*np.sin(k_th*a/2)**2/
 ↪(m1*m2)))
omega_optical = np.sqrt(omega_sq_plus)
omega_acoustic = np.sqrt(omega_sq_minus)

fig, ax = plt.subplots(figsize=(6, 5))
omega_pos_di = omega_di_axis >= 0
ax.pcolormesh(k_di_axis, omega_di_axis[omega_pos_di],
              np.log10(power_di[:, omega_pos_di].T + 1),
              cmap='hot', shading='auto')
```

(continues on next page)

(continued from previous page)

```
ax.plot(k_th, omega_acoustic, 'c-', linewidth=1.5, label='Acoustic branch')
ax.plot(-k_th, omega_acoustic, 'c-', linewidth=1.5)
ax.plot(k_th, omega_optical, 'lime', linewidth=1.5, label='Optical branch')
ax.plot(-k_th, omega_optical, 'lime', linewidth=1.5)

# Band gap
omega_gap_low = np.sqrt(2*K_di/m2)
omega_gap_high = np.sqrt(2*K_di/m1)
ax.axhspan(omega_gap_low, omega_gap_high, alpha=0.15, color='yellow', label='Band gap
↪')

ax.set_xlabel('Wavevector k')
ax.set_ylabel('Frequency ω')
ax.set_title(f'Diatomic Chain: Acoustic + Optical Phonons (m₂/m₁ = {m2/m1:.0f})')
ax.set_ylim(0, 2.0)
ax.legend(fontsize=7, loc='upper right')
plt.tight_layout()
plt.show()

print(f"Band gap: ω = {omega_gap_low:.3f} to {omega_gap_high:.3f}")
print(f"No propagating modes in the gap — this is why some crystals are transparent!")
print(f"The mass ratio m2/m1 = {m2/m1:.0f} controls the gap width.")
```



```
Band gap: ω = 0.816 to 1.414
No propagating modes in the gap — this is why some crystals are transparent!
The mass ratio m2/m1 = 3 controls the gap width.
```

10.10 IX. Physics Application: Nonlinear Pendulum

The **full** pendulum equation (not the small-angle approximation):

$$\ddot{\theta} = -\frac{g}{L}\sin\theta$$

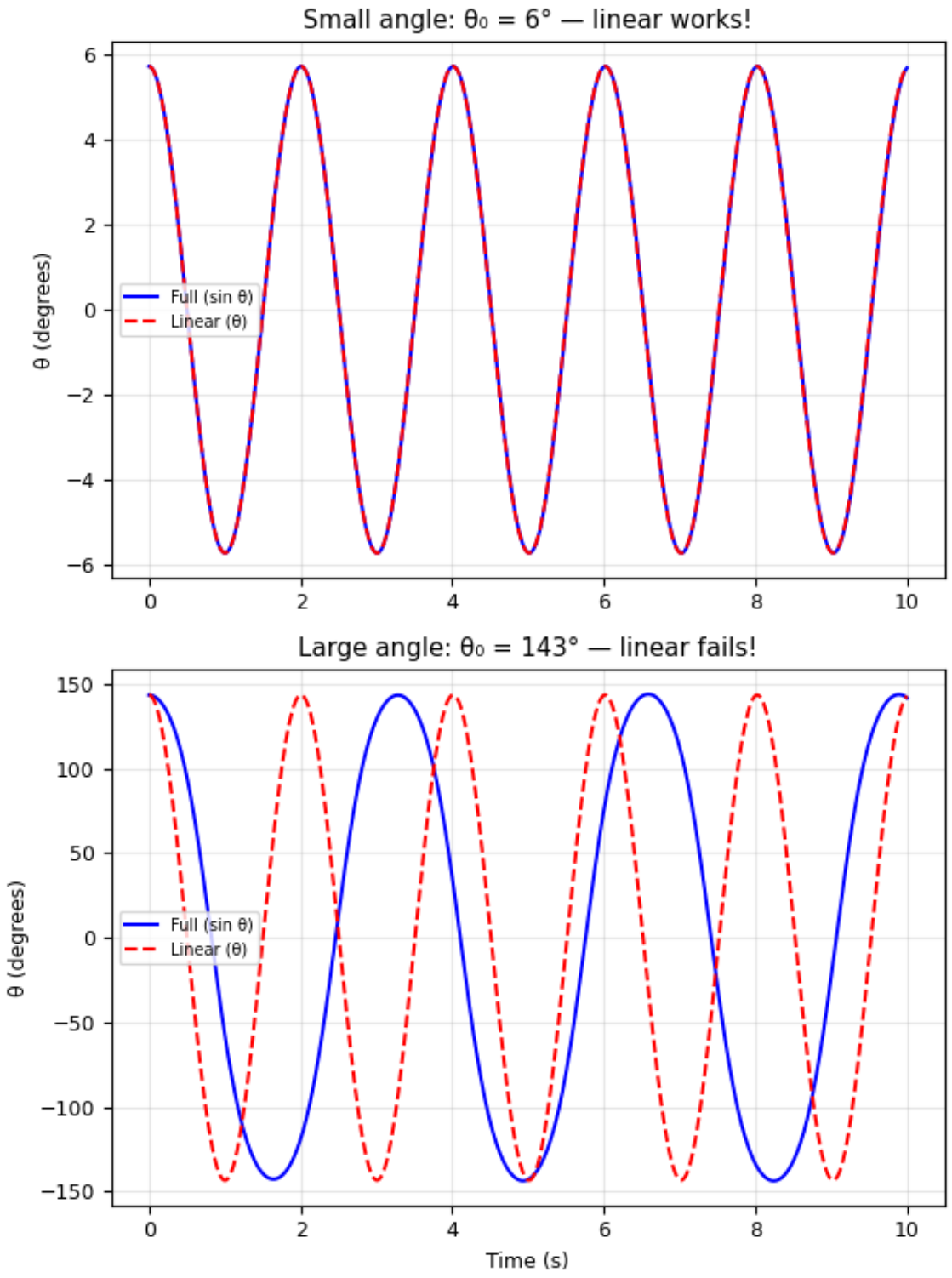
For small θ: $\sin\theta \approx \theta \rightarrow$ SHO. For large θ: the period depends on amplitude!

```
# Nonlinear pendulum: compare small-angle vs full equation
g_pend = 9.81
L_pend = 1.0

def pendulum_full(t, y):
    theta, omega_p = y
    return [omega_p, -(g_pend/L_pend) * np.sin(theta)]

def pendulum_linear(t, y):
    theta, omega_p = y
    return [omega_p, -(g_pend/L_pend) * theta]

t_pend = np.linspace(0, 10, 1000)

fig, axes = plt.subplots(2, 1, figsize=(6, 8))

# Small angle: linear ≈ nonlinear
theta0_small = 0.1  # ~6°
sol_full_s = solve_ivp(pendulum_full, [0, 10], [theta0_small, 0], t_eval=t_pend)
sol_lin_s = solve_ivp(pendulum_linear, [0, 10], [theta0_small, 0], t_eval=t_pend)

axes[0].plot(sol_full_s.t, np.degrees(sol_full_s.y[0]), 'b-', label='Full (sin θ)')
axes[0].plot(sol_lin_s.t, np.degrees(sol_lin_s.y[0]), 'r--', label='Linear (θ)')
axes[0].set_title(f'Small angle: θ₀ = {np.degrees(theta0_small):.0f}° — linear works!
 ↪')
axes[0].set_ylabel('θ (degrees)')
axes[0].legend(fontsize=7)
axes[0].grid(True, alpha=0.3)

# Large angle: linear fails
theta0_large = 2.5  # ~143°
sol_full_l = solve_ivp(pendulum_full, [0, 10], [theta0_large, 0], t_eval=t_pend)
sol_lin_l = solve_ivp(pendulum_linear, [0, 10], [theta0_large, 0], t_eval=t_pend)

axes[1].plot(sol_full_l.t, np.degrees(sol_full_l.y[0]), 'b-', label='Full (sin θ)')
axes[1].plot(sol_lin_l.t, np.degrees(sol_lin_l.y[0]), 'r--', label='Linear (θ)')
axes[1].set_title(f'Large angle: θ₀ = {np.degrees(theta0_large):.0f}° — linear fails!
 ↪')
axes[1].set_xlabel('Time (s)')
axes[1].set_ylabel('θ (degrees)')
axes[1].legend(fontsize=7)
axes[1].grid(True, alpha=0.3)
```

(continues on next page)

(continued from previous page)

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

# Measure periods
# Find zero-crossings of the full solution
from scipy.signal import find_peaks
peaks, _ = find_peaks(sol_full_l.y[0])
if len(peaks) >= 2:
    T_numerical = np.mean(np.diff(sol_full_l.t[peaks]))
    T_linear = 2 * np.pi * np.sqrt(L_pend / g_pend)
    print(f"\nLinear period:    T₀ = {T_linear:.4f} s")
    print(f"Nonlinear period: T  = {T_numerical:.4f} s  (θ₀ = {np.degrees(theta0_
↪large):.0f}°)")
    print(f"Difference: {(T_numerical - T_linear)/T_linear * 100:.1f}% longer")
```




```
Linear period:    T₀ = 2.0061 s
Nonlinear period: T  = 3.3033 s  (θ₀ = 143°)
Difference: 64.7% longer
```

CHAPTER

ELEVEN

LECTURE 11: PARTIAL DIFFERENTIAL EQUATIONS (PDES)

Computational Physics — Spring 2026

11.1 Why PDEs?

PDEs describe how quantities vary in both **space and time**:

Physics	Equation	Type
Heat conduction	$\partial T/\partial t = \alpha\, \partial^2 T/\partial x^2$	Parabolic
Wave propagation	$\partial^2 u/\partial t^2 = c^2\, \partial^2 u/\partial x^2$	Hyperbolic
Electrostatics	$\nabla^2 \phi = -\rho/\epsilon_0$	Elliptic
Quantum mechanics	$i\hbar\, \partial\psi/\partial t = -(\hbar^2/2m)\, \partial^2\psi/\partial x^2 + V\psi$	Parabolic (in imaginary time)
Fluid dynamics	$\partial \mathbf{v}/\partial t + (\mathbf{v}\cdot\nabla)\mathbf{v} = -\nabla p/\rho + \nu\nabla^2\mathbf{v}$	Mixed

Strategy: Discretize space on a grid, replace derivatives with finite differences → system of ODEs or algebraic equations.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
from scipy import sparse
from scipy.sparse.linalg import spsolve

# Projector-friendly settings
plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

print("Ready!")
```

```
Ready!
```

11.2 I. PDE Classification and Finite Differences

11.2.1 Three Types of PDEs

For a second-order linear PDE $Au_{xx} + 2Bu_{xt} + Cu_{tt} = \ldots$:

Type	Discriminant	Prototype	Physics	Boundary conditions
Parabolic	$B^2 - AC = 0$	Heat equation	Diffusion, relaxation	Initial + boundary
Hyperbolic	$B^2 - AC > 0$	Wave equation	Waves, vibrations	Initial + boundary
Elliptic	$B^2 - AC < 0$	Laplace equation	Steady state, equilibrium	Boundary only

Each type requires different numerical strategies!

11.2.2 Finite Difference Refresher

From Lecture 05, we approximate derivatives on a grid with spacing Δx:

$$\left.\frac{\partial u}{\partial x}\right|_i \approx \frac{u_{i+1} - u_{i-1}}{2\Delta x}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_i \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$$

For time derivatives with spacing Δt:

$$\left.\frac{\partial u}{\partial t}\right|^n \approx \frac{u^{n+1} - u^n}{\Delta t}$$

Notation: u_i^n = value at grid point i, time step n.

11.3 II. The 1D Heat Equation (Parabolic PDE)

11.3.1 The Physics

A metal rod of length L with temperature $T(x, t)$:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

where α is the **thermal diffusivity** (how fast heat spreads).

11.3.2 FTCS: Forward Time, Central Space

Discretize: forward difference in time, central difference in space:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \alpha \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2}$$

Solving for the future:

$$\boxed{T_i^{n+1} = T_i^n + r\left(T_{i+1}^n - 2T_i^n + T_{i-1}^n\right)}$$

where $r = \alpha \Delta t / \Delta x^2$ is the **mesh ratio**.

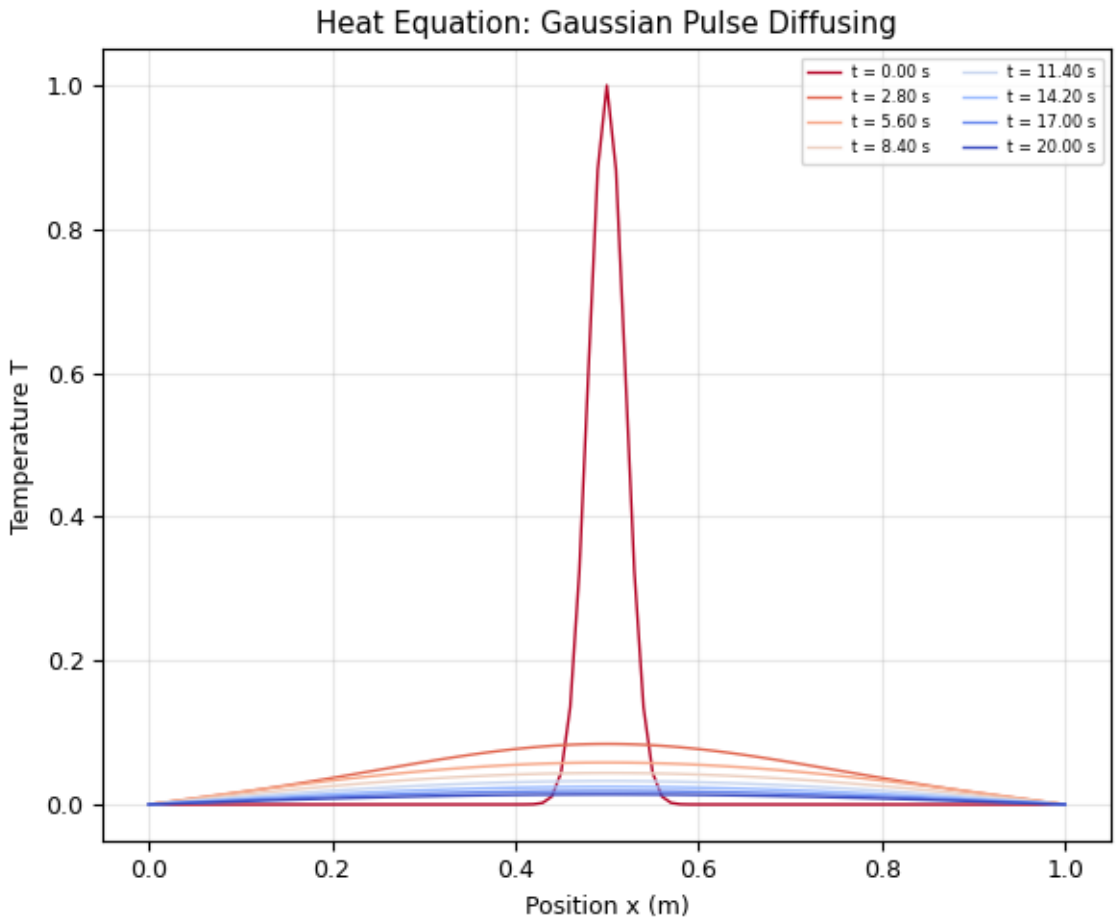
```
def heat_ftcs(T0, alpha, dx, dt, n_steps, bc_left=0.0, bc_right=0.0):
    """Solve the 1D heat equation using FTCS (Forward Time, Central Space).

    Parameters
    ----------
    T0 : ndarray
        Initial temperature profile (N_x points).
    alpha : float
        Thermal diffusivity.
    dx : float
        Spatial grid spacing.
    dt : float
        Time step.
    n_steps : int
        Number of time steps to take.
    bc_left, bc_right : float
        Dirichlet boundary conditions.

    Returns
    -------
    T_history : ndarray
        Temperature at each saved time step, shape (n_saved, N_x).
    """
    N_x = len(T0)
    r = alpha * dt / dx**2   # Mesh ratio

    T = T0.copy()
    save_every = max(1, n_steps // 100)
    T_history = [T.copy()]

    for n in range(n_steps):
        T_new = T.copy()

        # code here
        for i in range(1, N_x -1):
            T_new[i] = T[i] + r*(T[i+1] - 2*T[i] + T[i-1])

        # Apply boundary conditions
        T_new[0] = bc_left
        T_new[-1] = bc_right
        T = T_new
        if (n + 1) % save_every == 0:
            T_history.append(T.copy())

    return np.array(T_history)

# ---- Demo: Hot spot diffusing in a cold rod ----
L = 1.0          # Rod length (m)
N_x = 101        # Grid points
alpha = 0.01     # Thermal diffusivity (m^2/s)
dx = L / (N_x - 1)
x = np.linspace(0, L, N_x)

# Initial condition: Gaussian hot spot in the middle
T0 = np.exp(-((x - 0.5)**2) / (2 * 0.02**2))

# Stable time step (we'll explain why later!)
```

(continues on next page)

(continued from previous page)

```
dt = 0.4 * dx**2 / alpha    # r = 0.4 < 0.5
n_steps = 5000

print(f"Grid spacing: dx = {dx:.4f}")
print(f"Time step:    dt = {dt:.6f}")
print(f"Mesh ratio:   r  = {alpha * dt / dx**2:.2f}")
print(f"Total time:   {n_steps * dt:.2f} s")

T_hist = heat_ftcs(T0, alpha, dx, dt, n_steps)

# Plot snapshots
fig, ax = plt.subplots(figsize=(6, 5))
n_snapshots = min(len(T_hist), 8)
colors = plt.cm.coolwarm(np.linspace(1, 0, n_snapshots))
times = np.linspace(0, n_steps * dt, len(T_hist))

snap_indices = np.linspace(0, len(T_hist)-1, n_snapshots, dtype=int)
for i, idx in enumerate(snap_indices):
    ax.plot(x, T_hist[idx], color=colors[i],
            linewidth=1, label=f't = {times[idx]:.2f} s')

ax.set_xlabel('Position x (m)')
ax.set_ylabel('Temperature T')
ax.set_title('Heat Equation: Gaussian Pulse Diffusing')
ax.legend(fontsize=6, ncol=2)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("The hot spot spreads out and flattens — diffusion in action!")
print(f"Conservation check: initial integral = {np.trapz(T0, x):.4f}, final = {np.
↪trapz(T_hist[-1], x):.4f}")
```

```
Grid spacing: dx = 0.0100
Time step:    dt = 0.004000
Mesh ratio:   r  = 0.40
Total time:   20.00 s
```




```
The hot spot spreads out and flattens — diffusion in action!
Conservation check: initial integral = 0.0501, final = 0.0088
```

```
/tmp/ipykernel_408/1936641005.py:88: DeprecationWarning: `trapz` is deprecated.␣
↪Use `trapezoid` instead, or one of the numerical integration functions in `scipy.
↪integrate`.
  print(f"Conservation check: initial integral = {np.trapz(T0, x):.4f}, final =
↪{np.trapz(T_hist[-1], x):.4f}")
```

11.4 III. Stability: Why Your Simulation Can Explode

11.4.1 The CFL Condition

The FTCS scheme is **conditionally stable**. It only works when:

$$\boxed{r = \frac{\alpha \Delta t}{\Delta x^2} \leq \frac{1}{2}}$$

If $r > 1/2$, small numerical errors **grow exponentially** and the solution blows up!

11.4.2 Von Neumann Stability Analysis

Assume a perturbation $\epsilon_i^n = A^n e^{ijk\Delta x}$. Substituting into the FTCS scheme:

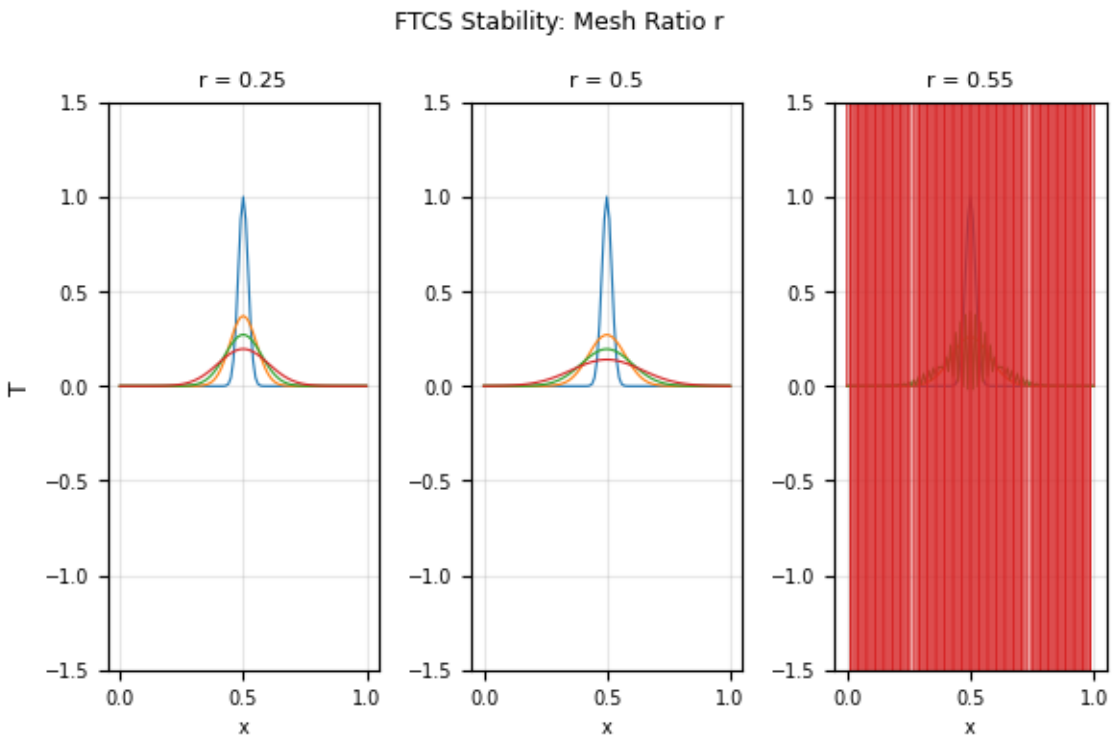
$$A = 1 - 4r\sin^2\left(\frac{k\Delta x}{2}\right)$$

For stability, $|A| \leq 1$ for all $k \rightarrow$ the worst case is $\sin^2 = 1$:

$$|1 - 4r| \leq 1 \quad \Rightarrow \quad r \leq \frac{1}{2}$$

Physical interpretation: Information can't travel faster than one grid cell per time step.

```
# Demonstrate instability: r > 0.5
fig, axes = plt.subplots(1, 3, figsize=(6, 4))

for idx, r_val in enumerate([0.25, 0.50, 0.55]):
    dt_test = r_val * dx**2 / alpha
    T_test = heat_ftcs(T0, alpha, dx, dt_test, n_steps=200)

    # Plot a few snapshots
    for snap_idx in [0, len(T_test)//4, len(T_test)//2, -1]:
        axes[idx].plot(x, T_test[snap_idx], linewidth=0.8)

    axes[idx].set_title(f'r = {r_val}', fontsize=8)
    axes[idx].set_xlabel('x', fontsize=7)
    axes[idx].set_ylim(-1.5, 1.5)
    axes[idx].grid(True, alpha=0.3)
    axes[idx].tick_params(labelsize=7)

axes[0].set_ylabel('T')
plt.suptitle('FTCS Stability: Mesh Ratio r', fontsize=9)
plt.tight_layout()
plt.show()

print("r = 0.25: Stable and smooth — diffusion works correctly")
print("r = 0.50: Right at the stability limit — oscillations may appear")
print("r = 0.55: UNSTABLE — oscillations grow exponentially!")
print("This is the CFL condition: Δt ≤ Δx²/(2α)")
```




```
r = 0.25: Stable and smooth — diffusion works correctly
r = 0.50: Right at the stability limit — oscillations may appear
r = 0.55: UNSTABLE — oscillations grow exponentially!
This is the CFL condition: Δt ≤ Δx²/(2α)
```

11.5 V. The 1D Wave Equation (Hyperbolic PDE)

11.5.1 The Physics

A vibrating string, sound wave, or electromagnetic wave:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

Unlike diffusion, waves **propagate without spreading** — information travels at speed c.

11.5.2 CTCS: Central Time, Central Space

Use central differences in **both** time and space:

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

Solving for the future:

$$\boxed{u_i^{n+1} = 2u_i^n - u_i^{n-1} + s^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)}$$

where $s = c\Delta t/\Delta x$ is the **Courant number**.

11.5.3 Stability (CFL Condition for Waves)

$$s = \frac{c\Delta t}{\Delta x} \leq 1$$

The numerical wave speed must not exceed the physical wave speed.

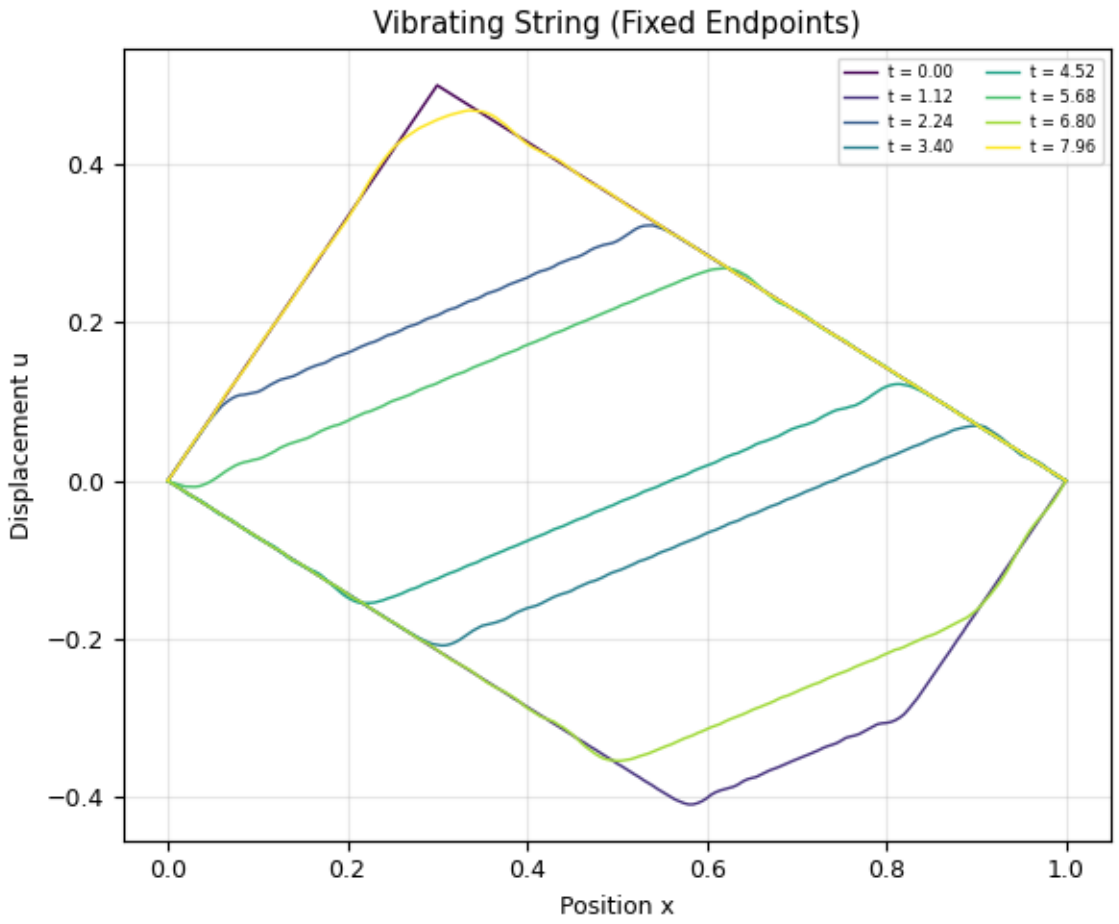
```
def wave_ctcs(u0, v0, c, dx, dt, n_steps, bc='fixed'):
    """Solve the 1D wave equation using CTCS (leapfrog) method.

    Parameters
    ----------
    u0 : ndarray
        Initial displacement.
    v0 : ndarray
        Initial velocity.
    c : float
        Wave speed.
    dx, dt : float
        Grid spacing and time step.
    n_steps : int
        Number of time steps.
    bc : str
        'fixed' (Dirichlet u=0) or 'free' (Neumann du/dx=0).

    Returns
    -------
    u_history : ndarray
        Displacement snapshots.
    """
    N = len(u0)
    s = c * dt / dx       # Courant number
    s2 = s**2

    u_prev = u0.copy()
    # First step using initial velocity: u^1 = u^0 + dt*v0 + 0.5*s2*(u_{i+1}-2u_i+u_
↪{i-1})
    u_curr = u0.copy()
    for i in range(1, N-1):
        u_curr[i] = (u0[i] + dt * v0[i]
                     + 0.5 * s2 * (u0[i+1] - 2*u0[i] + u0[i-1]))

    save_every = max(1, n_steps // 200)
    u_history = [u0.copy()]

    for n in range(1, n_steps):
        u_next = np.zeros(N)
        for i in range(1, N-1):
            # u_next[i] =
            u_next[i] = 2*u_curr[i]-u_prev[i] + s2 * (u_curr[i+1] - 2*u_curr[i] + u_
↪curr[i-1])

        # Boundary conditions
        if bc == 'fixed':
            u_next[0] = 0.0
            u_next[-1] = 0.0
        elif bc == 'free':
            u_next[0] = u_next[1]
```

(continues on next page)

(continued from previous page)

```
            u_next[-1] = u_next[-2]

        u_prev = u_curr.copy()
        u_curr = u_next.copy()

        if n % save_every == 0:
            u_history.append(u_curr.copy())

    return np.array(u_history)

# ---- Demo: Plucked string ----
L = 1.0
N_x = 201
c = 1.0         # Wave speed
dx = L / (N_x - 1)
x = np.linspace(0, L, N_x)

# Initial condition: plucked string (triangle)
u0 = np.where(x < 0.3, x / 0.3 * 0.5, 0.5 * (1 - x) / 0.7)
u0[0] = 0; u0[-1] = 0   # Fixed endpoints
v0 = np.zeros(N_x)       # Released from rest

# Stable time step
dt = 0.8 * dx / c  # Courant number = 0.8
n_steps = 2000

print(f"Courant number: s = c*dt/dx = {c * dt / dx:.2f}")
print(f"Total simulation time: {n_steps * dt:.2f} s")

u_hist = wave_ctcs(u0, v0, c, dx, dt, n_steps, bc='fixed')

# Plot snapshots
fig, ax = plt.subplots(figsize=(6, 5))
n_show = 8
colors = plt.cm.viridis(np.linspace(0, 1, n_show))

for idx, snap in enumerate(np.linspace(0, len(u_hist)-1, n_show, dtype=int)):
    t_val = snap * (n_steps / len(u_hist)) * dt
    ax.plot(x, u_hist[snap], color=colors[idx], linewidth=1,
            label=f't = {t_val:.2f}')

ax.set_xlabel('Position x')
ax.set_ylabel('Displacement u')
ax.set_title('Vibrating String (Fixed Endpoints)')
ax.legend(fontsize=6, ncol=2)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("The plucked string vibrates — standing waves form!")
print("Fixed endpoints → only certain wavelengths are allowed (harmonics)")
```

```
Courant number: s = c*dt/dx = 0.80
Total simulation time: 8.00 s
```




```
The plucked string vibrates — standing waves form!
Fixed endpoints → only certain wavelengths are allowed (harmonics)
```

11.5.4 Wave Packet Propagation

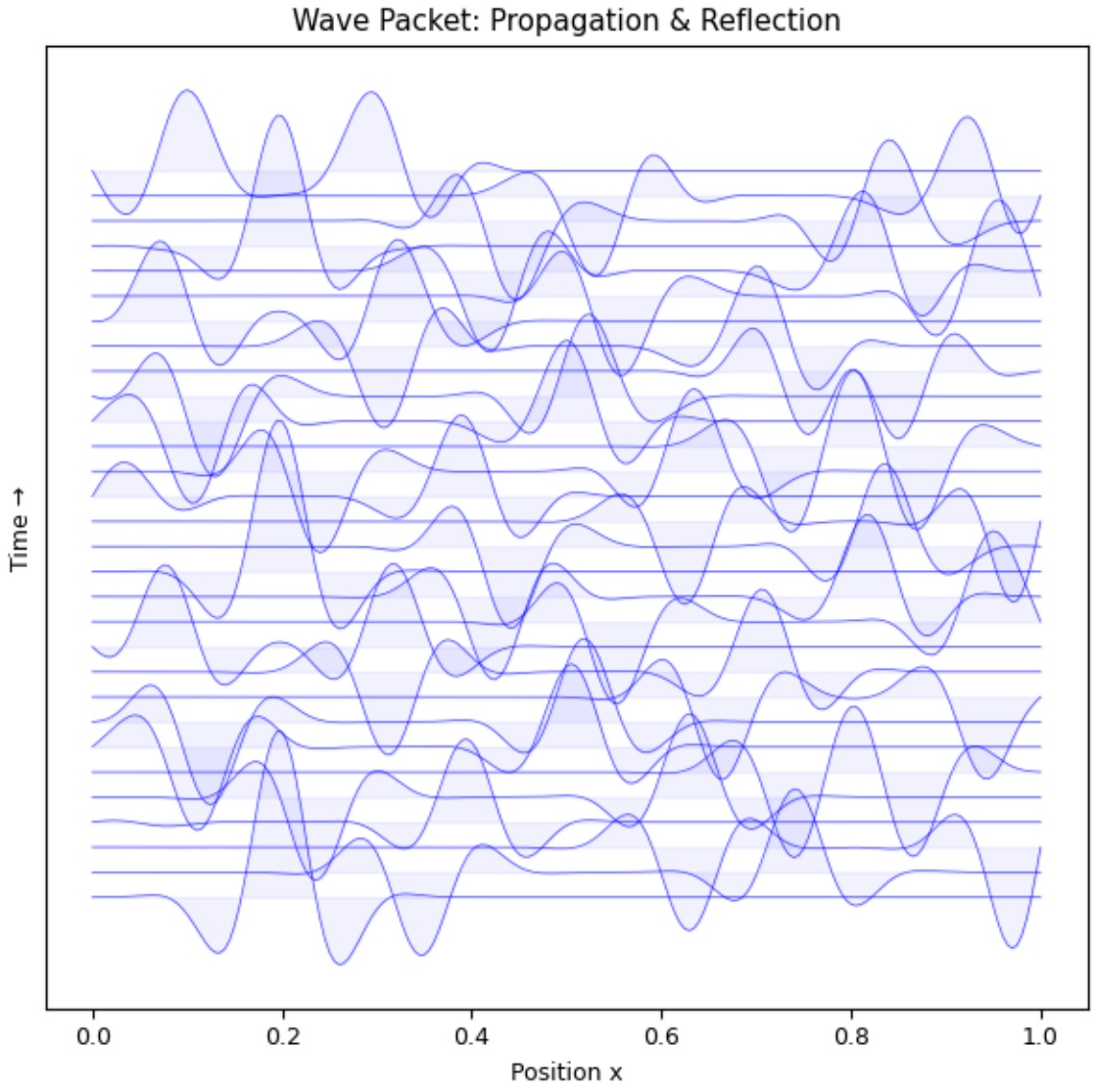
Let's watch a Gaussian wave packet travel, reflect off boundaries, and interfere with itself.

```
# Wave packet: Gaussian envelope * carrier wave
sigma = 0.05
x0_wave = 0.2
k0 = 40.0  # carrier wavevector

u0_packet = np.exp(-((x - x0_wave)**2) / (2*sigma**2)) * np.sin(k0 * x)
u0_packet[0] = 0; u0_packet[-1] = 0
v0_packet = np.zeros(N_x)

dt_wp = 0.9 * dx / c
u_wp = wave_ctcs(u0_packet, v0_packet, c, dx, dt_wp, n_steps=4000, bc='fixed')

# Waterfall plot
fig, ax = plt.subplots(figsize=(6, 6))
n_traces = 30
for idx, snap in enumerate(np.linspace(0, len(u_wp)-1, n_traces, dtype=int)):
```

(continues on next page)

(continued from previous page)

```
    offset = idx * 0.15
    ax.plot(x, u_wp[snap] + offset, 'b-', linewidth=0.5, alpha=0.7)
    ax.fill_between(x, offset, u_wp[snap] + offset, alpha=0.05, color='blue')

ax.set_xlabel('Position x')
ax.set_ylabel('Time →')
ax.set_title('Wave Packet: Propagation & Reflection')
ax.set_yticks([])
plt.tight_layout()
plt.show()

print("The wave packet travels right, reflects off the fixed end (inverted!),")
print("travels left, reflects again, and keeps bouncing.")
print("Fixed boundary: reflection with sign flip (node at boundary)")
```




```
The wave packet travels right, reflects off the fixed end (inverted!),
travels left, reflects again, and keeps bouncing.
```

(continues on next page)

(continued from previous page)

```
Fixed boundary: reflection with sign flip (node at boundary)
```

11.6 VI. The 2D Laplace Equation (Elliptic PDE)

11.6.1 The Physics

In electrostatics, the potential ϕ in a charge-free region satisfies:

$$\nabla^2 \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

No time variable! This is a **boundary value problem** — the solution everywhere is determined by the boundary conditions.

11.6.2 Jacobi Relaxation Method

Discretize on a 2D grid. At each interior point:

$$\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{\Delta x^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{\Delta y^2} = 0$$

For $\Delta x = \Delta y$, solve for $\phi_{i,j}$:

$$\boxed{\phi_{i,j} = \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}\right)}$$

Each point is the **average of its four neighbors**! Iterate until convergence:

1. Initialize with a guess
2. Update every interior point with the 4-neighbor average
3. Check if the solution has changed
4. Repeat until changes are smaller than tolerance

```
def laplace_jacobi(phi0, bc_mask, max_iter=10000, tol=1e-5):
    """Solve 2D Laplace equation using Jacobi relaxation.

    Parameters
    ----------
    phi0 : ndarray (Ny, Nx)
        Initial guess (including boundary values).
    bc_mask : ndarray of bool (Ny, Nx)
        True where values are fixed (boundary conditions).
    max_iter : int
        Maximum iterations.
    tol : float
        Convergence tolerance (max change per iteration).

    Returns
    -------
    phi : ndarray
```

(continues on next page)

(continued from previous page)

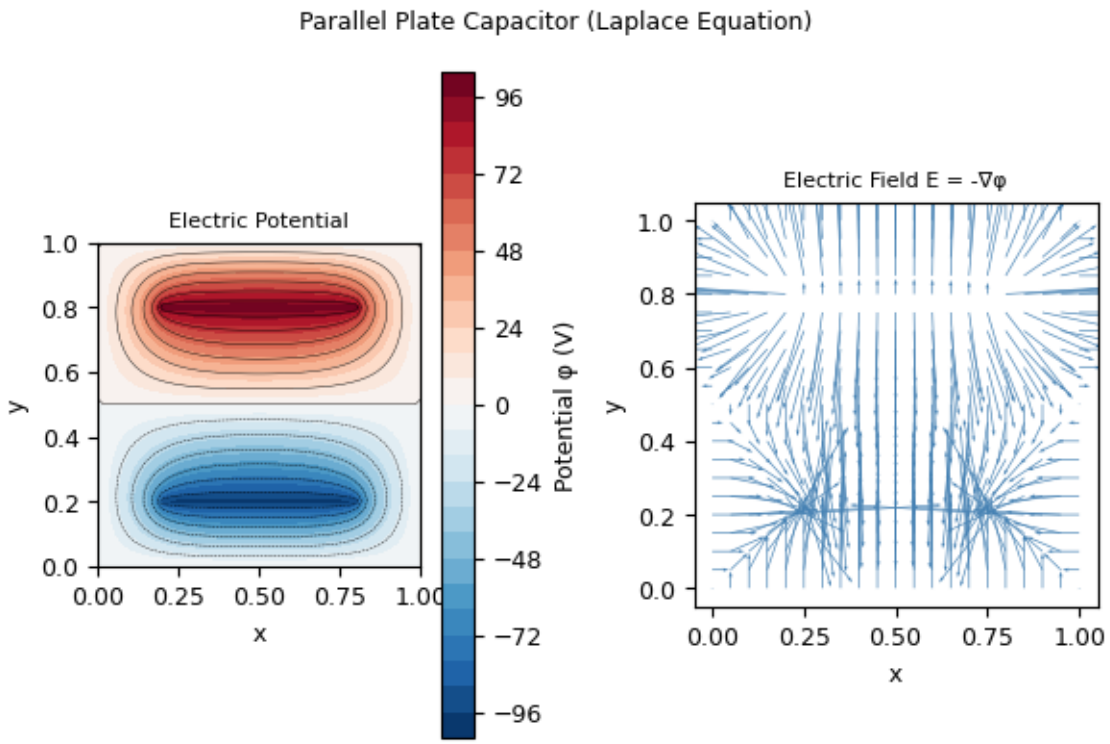
```
        Converged solution.
    residuals : list
        Max residual at each iteration.
    """
    phi = phi0.copy()
    residuals = []

    for iteration in range(max_iter):
        phi_old = phi.copy()

        ## N = len(phi)
        ## for i in range (1, N-1):
        ##.   for j in range (1, N-1):
        ##         phi[i,j] = (phi_old[i-1,j] + phi_old[i+1,j] + phi_old[i,j-1] + phi_
↪old[i,j+1])/4

        # Update interior points: average of 4 neighbors
        phi[1:-1, 1:-1] = 0.25 * (
            phi_old[2:, 1:-1] + phi_old[:-2, 1:-1] +   # up + down
            phi_old[1:-1, 2:] + phi_old[1:-1, :-2]       # right + left
        )

        # Restore boundary conditions
        phi[bc_mask] = phi0[bc_mask]

        # Check convergence
        max_change = np.max(np.abs(phi - phi_old))
        residuals.append(max_change)

        if max_change < tol:
            print(f"Converged in {iteration+1} iterations (max change = {max_change:.
↪2e})")
            break
    else:
        print(f"Warning: did not converge after {max_iter} iterations (max change =
↪{max_change:.2e})")

    return phi, residuals

# ---- Parallel plate capacitor ----
Nx, Ny = 61, 61
phi0 = np.zeros((Ny, Nx))
bc_mask = np.zeros((Ny, Nx), dtype=bool)

# Top plate at y = 0.8, x from 0.2 to 0.8: V = +100
# Bottom plate at y = 0.2, x from 0.2 to 0.8: V = -100
plate_y_top = int(0.8 * (Ny - 1))
plate_y_bot = int(0.2 * (Ny - 1))
plate_x_start = int(0.2 * (Nx - 1))
plate_x_end = int(0.8 * (Nx - 1))

phi0[plate_y_top, plate_x_start:plate_x_end+1] = +100
phi0[plate_y_bot, plate_x_start:plate_x_end+1] = -100
bc_mask[plate_y_top, plate_x_start:plate_x_end+1] = True
bc_mask[plate_y_bot, plate_x_start:plate_x_end+1] = True

# Boundary: walls at V=0
```

(continues on next page)

(continued from previous page)

```
bc_mask[0, :] = True; bc_mask[-1, :] = True
bc_mask[:, 0] = True; bc_mask[:, -1] = True

phi, residuals = laplace_jacobi(phi0, bc_mask, max_iter=20000, tol=1e-6)

# Plot
x_grid = np.linspace(0, 1, Nx)
y_grid = np.linspace(0, 1, Ny)
X, Y = np.meshgrid(x_grid, y_grid)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(6, 4))

# Potential contour
cp = ax1.contourf(X, Y, phi, levels=30, cmap='RdBu_r')
ax1.contour(X, Y, phi, levels=15, colors='k', linewidths=0.3)
plt.colorbar(cp, ax=ax1, label='Potential φ (V)')
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('Electric Potential', fontsize=8)
ax1.set_aspect('equal')

# Electric field (negative gradient of potential)
Ey, Ex = np.gradient(-phi, y_grid, x_grid)
skip = 3
ax2.quiver(X[::skip, ::skip], Y[::skip, ::skip],
           Ex[::skip, ::skip], Ey[::skip, ::skip],
           color='steelblue', scale=2000, width=0.003)
ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.set_title('Electric Field E = -∇φ', fontsize=8)
ax2.set_aspect('equal')

plt.suptitle('Parallel Plate Capacitor (Laplace Equation)', fontsize=9)
plt.tight_layout()
plt.show()

print("Between the plates: nearly uniform field (as expected)")
print("Fringe fields visible at the plate edges")
```

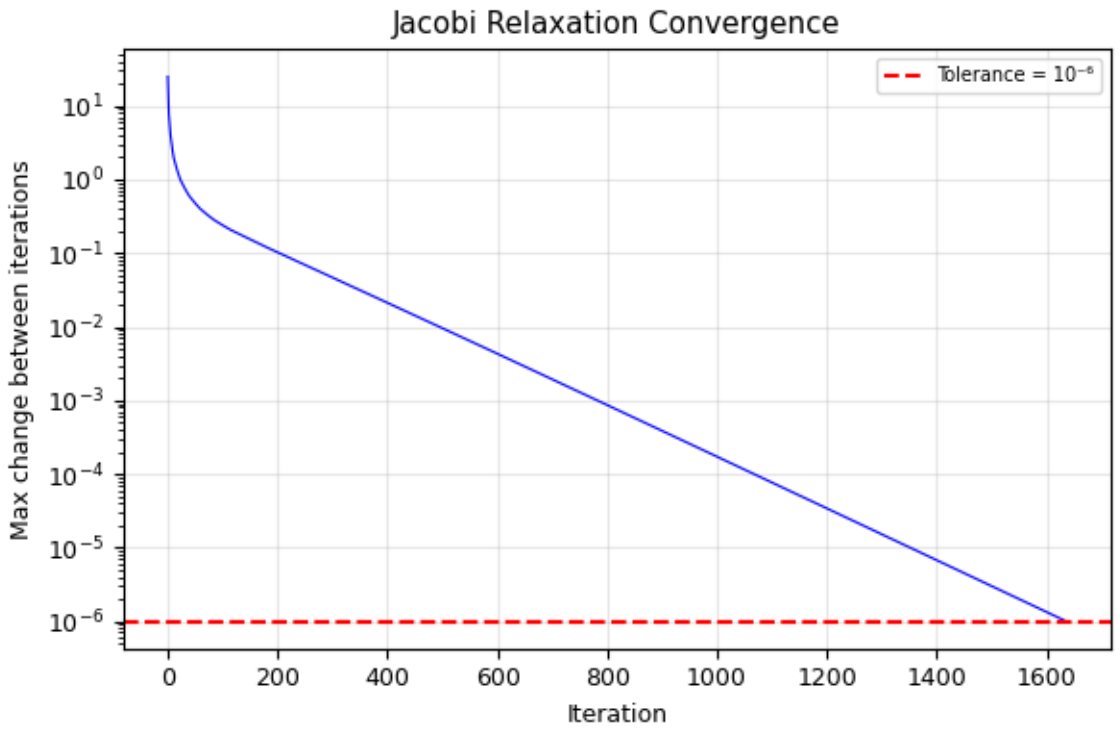
```
Converged in 1636 iterations (max change = 9.99e-07)
```




```
Between the plates: nearly uniform field (as expected)
Fringe fields visible at the plate edges
```

11.6.3 Convergence of Jacobi Iteration

```
fig, ax = plt.subplots(figsize=(6, 4))
ax.semilogy(residuals, 'b-', linewidth=0.8)
ax.set_xlabel('Iteration')
ax.set_ylabel('Max change between iterations')
ax.set_title('Jacobi Relaxation Convergence')
ax.axhline(1e-6, color='r', linestyle='--', label='Tolerance = 10⁻⁶')
ax.legend(fontsize=7)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f"Jacobi converged in {len(residuals)} iterations.")
print("Gauss-Seidel and SOR (Successive Over-Relaxation) converge faster.")
print("For large problems, multigrid methods are much more efficient.")
```




```
Jacobi converged in 1636 iterations.
Gauss-Seidel and SOR (Successive Over-Relaxation) converge faster.
For large problems, multigrid methods are much more efficient.
```

11.7 VII. Faster Solvers: Gauss-Seidel and SOR

11.7.1 Gauss-Seidel

Same as Jacobi, but use **updated values immediately** as they become available (just like Euler-Cromer!):

When computing $\phi_{i,j}$, use the already-updated $\phi_{i-1,j}$ and $\phi_{i,j-1}$ from **this** iteration.

11.7.2 SOR (Successive Over-Relaxation)

Take the Gauss-Seidel update and **overshoot**:

$$\phi_{i,j}^{\text{new}} = (1-\omega)\,\phi_{i,j}^{\text{old}} + \omega \cdot \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}\right)$$

where ω is the **relaxation parameter**:

- $\omega = 1$: Gauss-Seidel
- $1 < \omega < 2$: Over-relaxation (faster convergence)
- $\omega \geq 2$: Unstable!

Optimal ω for an $N \times N$ grid: $\omega_{\text{opt}} \approx 2 - \frac{2\pi}{N}$

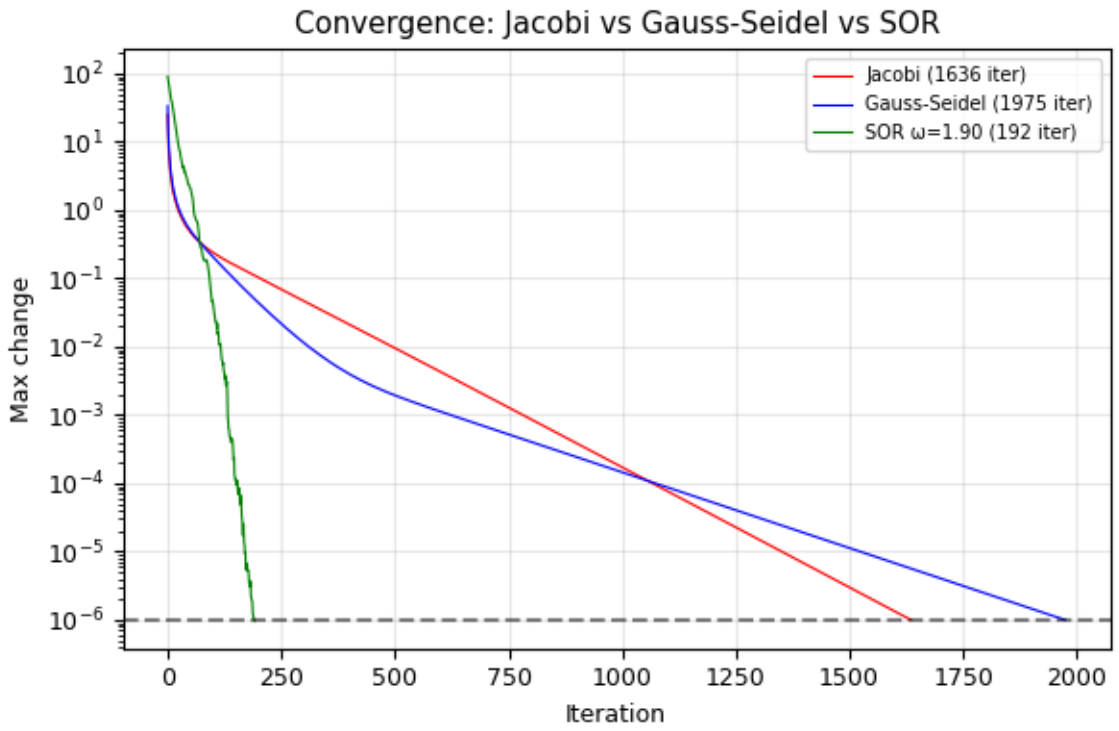
```
def laplace_sor(phi0, bc_mask, omega=1.5, max_iter=10000, tol=1e-5):
    """Solve 2D Laplace equation using SOR (Successive Over-Relaxation).

    Parameters
    ----------
    phi0 : ndarray (Ny, Nx)
        Initial guess (including boundary values).
    bc_mask : ndarray of bool
        True where values are fixed.
    omega : float
        Relaxation parameter (1 = Gauss-Seidel, 1 < omega < 2 = SOR).
    max_iter : int
        Maximum iterations.
    tol : float
        Convergence tolerance.

    Returns
    -------
    phi : ndarray
        Converged solution.
    residuals : list
        Max residual at each iteration.
    """
    phi = phi0.copy()
    Ny, Nx = phi.shape
    residuals = []

    for iteration in range(max_iter):
        max_change = 0.0

        for i in range(1, Ny-1):
            for j in range(1, Nx-1):
                if bc_mask[i, j]:
                    continue

                # Gauss-Seidel update (uses already-updated neighbors)
                phi_gs = 0.25 * (phi[i+1, j] + phi[i-1, j] +
                                 phi[i, j+1] + phi[i, j-1])

                # SOR: blend old and Gauss-Seidel
                phi_new = (1 - omega) * phi[i, j] + omega * phi_gs

                change = abs(phi_new - phi[i, j])
                if change > max_change:
                    max_change = change
                phi[i, j] = phi_new

        residuals.append(max_change)
        if max_change < tol:
            break

    return phi, residuals

# Compare Jacobi vs Gauss-Seidel vs SOR
omega_opt = 2 - 2*np.pi / Nx  # Optimal relaxation parameter

print(f"Grid size: {Nx}x{Ny}")
```

(continues on next page)

(continued from previous page)

```
print(f"Optimal ω ≈ {omega_opt:.3f}")
print()

_, res_jacobi = laplace_jacobi(phi0, bc_mask, max_iter=20000, tol=1e-6)
_, res_gs = laplace_sor(phi0, bc_mask, omega=1.0, max_iter=20000, tol=1e-6)
_, res_sor = laplace_sor(phi0, bc_mask, omega=omega_opt, max_iter=20000, tol=1e-6)

fig, ax = plt.subplots(figsize=(6, 4))
ax.semilogy(res_jacobi, 'r-', linewidth=0.8, label=f'Jacobi ({len(res_jacobi)} iter)')
ax.semilogy(res_gs, 'b-', linewidth=0.8, label=f'Gauss-Seidel ({len(res_gs)} iter)')
ax.semilogy(res_sor, 'g-', linewidth=0.8, label=f'SOR ω={omega_opt:.2f} ({len(res_
↪sor)} iter)')
ax.axhline(1e-6, color='k', linestyle='--', alpha=0.5)
ax.set_xlabel('Iteration')
ax.set_ylabel('Max change')
ax.set_title('Convergence: Jacobi vs Gauss-Seidel vs SOR')
ax.legend(fontsize=7)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f"Jacobi:       {len(res_jacobi)} iterations")
print(f"Gauss-Seidel: {len(res_gs)} iterations ({len(res_jacobi)/len(res_gs):.1f}x␣
↪faster)")
print(f"SOR:          {len(res_sor)} iterations ({len(res_jacobi)/len(res_sor):.1f}x␣
↪faster)")
```

```
Grid size: 61x61
Optimal ω ≈ 1.897

Converged in 1636 iterations (max change = 9.99e-07)
```




```
Jacobi:       1636 iterations
Gauss-Seidel: 1975 iterations (0.8x faster)
SOR:          192 iterations (8.5x faster)
```

11.8 VIII. Physics Application: Poisson Equation (Charges!)

The **Poisson equation** includes a source term:

$$\nabla^2 \phi = -\frac{\rho}{\epsilon_0}$$

This describes the electric potential from a charge distribution $\rho(x, y)$.

We modify the Jacobi update:

$$\phi_{i,j} = \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}\right) + \frac{\Delta x^2}{4\epsilon_0}\rho_{i,j}$$

```python
def poisson_jacobi(phi0, rho, bc_mask, dx, epsilon_0=1.0, max_iter=20000, tol=1e-5):
    """Solve 2D Poisson equation using Jacobi iteration.

    ∇²φ = -ρ/ε₀
    """
    phi = phi0.copy()
    source = (dx**2 / (4 * epsilon_0)) * rho
    residuals = []

    for iteration in range(max_iter):
        phi_old = phi.copy()
```

(continues on next page)

(continued from previous page)

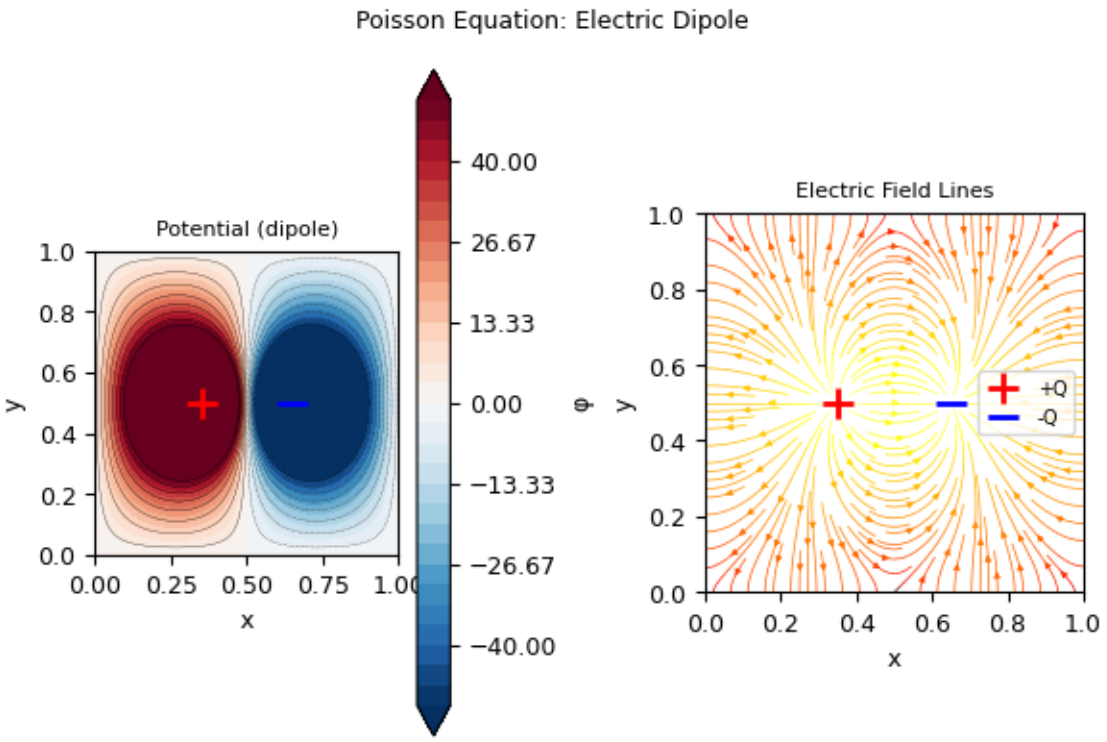
```
        phi[1:-1, 1:-1] = 0.25 * (
            phi_old[2:, 1:-1] + phi_old[:-2, 1:-1] +
            phi_old[1:-1, 2:] + phi_old[1:-1, :-2]
        ) + source[1:-1, 1:-1]

        phi[bc_mask] = phi0[bc_mask]

        max_change = np.max(np.abs(phi - phi_old))
        residuals.append(max_change)
        if max_change < tol:
            print(f"Converged in {iteration+1} iterations")
            break

    return phi, residuals

# ---- Electric dipole ----
Nx2, Ny2 = 81, 81
dx2 = 1.0 / (Nx2 - 1)
x2 = np.linspace(0, 1, Nx2)
y2 = np.linspace(0, 1, Ny2)
X2, Y2 = np.meshgrid(x2, y2)

# Charge distribution: +Q and -Q
rho = np.zeros((Ny2, Nx2))
# Positive charge at (0.35, 0.5)
q_plus_j = int(0.35 * (Nx2 - 1))
q_plus_i = int(0.5 * (Ny2 - 1))
rho[q_plus_i, q_plus_j] = +1000 / dx2**2

# Negative charge at (0.65, 0.5)
q_minus_j = int(0.65 * (Nx2 - 1))
q_minus_i = int(0.5 * (Ny2 - 1))
rho[q_minus_i, q_minus_j] = -1000 / dx2**2

# Boundary: grounded box (φ = 0)
phi0_poisson = np.zeros((Ny2, Nx2))
bc_mask2 = np.zeros((Ny2, Nx2), dtype=bool)
bc_mask2[0, :] = True; bc_mask2[-1, :] = True
bc_mask2[:, 0] = True; bc_mask2[:, -1] = True

phi_dipole, _ = poisson_jacobi(phi0_poisson, rho, bc_mask2, dx2, max_iter=30000,␣
↪tol=1e-5)

# Plot
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(6, 4))

# Potential
levels = np.linspace(-50, 50, 31)
cp = ax1.contourf(X2, Y2, phi_dipole, levels=levels, cmap='RdBu_r', extend='both')
ax1.contour(X2, Y2, phi_dipole, levels=levels[::2], colors='k', linewidths=0.2)
plt.colorbar(cp, ax=ax1, label='φ')
ax1.plot(x2[q_plus_j], y2[q_plus_i], 'r+', markersize=12, markeredgewidth=2)
ax1.plot(x2[q_minus_j], y2[q_minus_i], 'b_', markersize=12, markeredgewidth=2)
ax1.set_title('Potential (dipole)', fontsize=8)
ax1.set_aspect('equal')
ax1.set_xlabel('x'); ax1.set_ylabel('y')
```

(continues on next page)

(continued from previous page)

```
# Electric field
Ey2, Ex2 = np.gradient(-phi_dipole, y2, x2)
E_mag = np.sqrt(Ex2**2 + Ey2**2)
skip2 = 4
ax2.streamplot(X2, Y2, Ex2, Ey2, color=np.log10(E_mag + 1),
               cmap='hot', density=1.5, linewidth=0.5, arrowsize=0.5)
ax2.plot(x2[q_plus_j], y2[q_plus_i], 'r+', markersize=12, markeredgewidth=2, label='+Q
 ↪')
ax2.plot(x2[q_minus_j], y2[q_minus_i], 'b_', markersize=12, markeredgewidth=2, label=
 ↪'-Q')
ax2.set_title('Electric Field Lines', fontsize=8)
ax2.set_aspect('equal')
ax2.set_xlabel('x'); ax2.set_ylabel('y')
ax2.legend(fontsize=7)

plt.suptitle('Poisson Equation: Electric Dipole', fontsize=9)
plt.tight_layout()
plt.show()

print("Field lines go from + to - charge")
print("Equipotential lines (contours) are perpendicular to field lines")
print("Far from the dipole: field falls off as 1/r³")
```

```
Converged in 5617 iterations
```




```
Field lines go from + to - charge
Equipotential lines (contours) are perpendicular to field lines
Far from the dipole: field falls off as 1/r³
```

11.9 IX. Physics Application: 2D Heat Equation

Combining spatial dimensions:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

FTCS in 2D:

$$T_{i,j}^{n+1} = T_{i,j}^n + r_x(T_{i+1,j}^n - 2T_{i,j}^n + T_{i-1,j}^n) + r_y(T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n)$$

Stability requires: $r_x + r_y \leq 1/2$ where $r_x = \alpha \Delta t / \Delta x^2$, $r_y = \alpha \Delta t / \Delta y^2$.

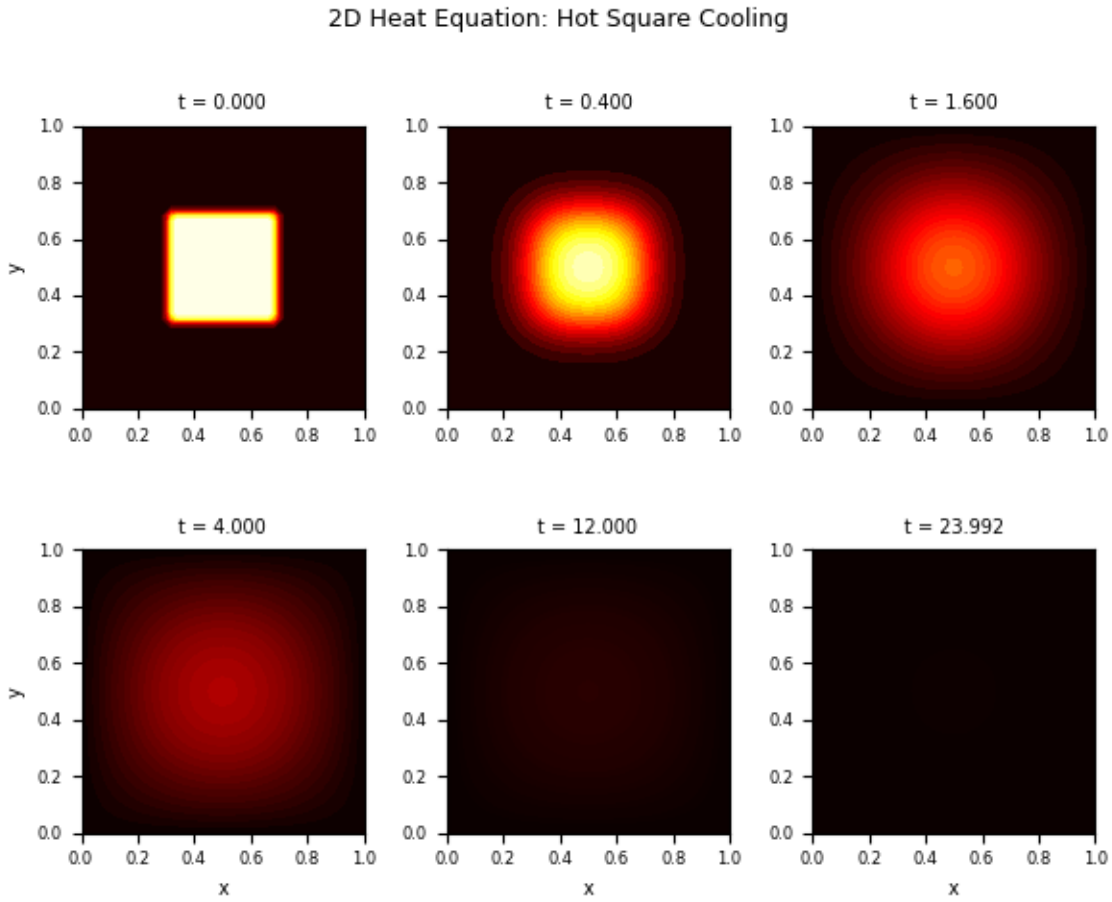
```
# 2D heat equation: hot square cooling into cold plate
Nx3 = 51
Ny3 = 51
alpha_2d = 0.01
dx3 = 1.0 / (Nx3 - 1)
dy3 = 1.0 / (Ny3 - 1)
x3 = np.linspace(0, 1, Nx3)
y3 = np.linspace(0, 1, Ny3)
X3, Y3 = np.meshgrid(x3, y3)

# Initial condition: hot square in center
T2d = np.zeros((Ny3, Nx3))
hot_region = (X3 > 0.3) & (X3 < 0.7) & (Y3 > 0.3) & (Y3 < 0.7)
T2d[hot_region] = 100.0

# Stable time step
dt3 = 0.2 * dx3**2 / alpha_2d  # conservative for 2D
rx = alpha_2d * dt3 / dx3**2
ry = alpha_2d * dt3 / dy3**2
print(f"rx + ry = {rx + ry:.3f} (must be ≤ 0.5)")

# Time-stepping
n_steps_2d = 3000
snapshots = []
snap_times = []

T = T2d.copy()
for n in range(n_steps_2d):
    T_new = T.copy()
    T_new[1:-1, 1:-1] = T[1:-1, 1:-1] + (
        rx * (T[2:, 1:-1] - 2*T[1:-1, 1:-1] + T[:-2, 1:-1]) +
        ry * (T[1:-1, 2:] - 2*T[1:-1, 1:-1] + T[1:-1, :-2])
    )
    # Dirichlet BC: T = 0 on all boundaries
    T_new[0, :] = 0; T_new[-1, :] = 0
    T_new[:, 0] = 0; T_new[:, -1] = 0
    T = T_new

    if n in [0, 50, 200, 500, 1500, n_steps_2d-1]:
        snapshots.append(T.copy())
        snap_times.append(n * dt3)

# Plot
fig, axes = plt.subplots(2, 3, figsize=(6, 5))
```

(continues on next page)

(continued from previous page)

```
axes = axes.flatten()

for idx, (snap, t_val) in enumerate(zip(snapshots, snap_times)):
    im = axes[idx].contourf(X3, Y3, snap, levels=20, cmap='hot', vmin=0, vmax=100)
    axes[idx].set_title(f't = {t_val:.3f}', fontsize=7)
    axes[idx].set_aspect('equal')
    axes[idx].tick_params(labelsize=6)
    if idx % 3 == 0:
        axes[idx].set_ylabel('y', fontsize=7)
    if idx >= 3:
        axes[idx].set_xlabel('x', fontsize=7)

plt.suptitle('2D Heat Equation: Hot Square Cooling', fontsize=9)
plt.tight_layout()
plt.show()

print("The sharp edges smooth out first (high curvature → fast diffusion)")
print("Eventually the temperature decays to zero everywhere")
print(f"Total heat: initial = {np.sum(T2d)*dx3*dy3:.2f}, final = {np.sum(snapshots[-
↪1])*dx3*dy3:.2f}")
```

```
rx + ry = 0.400 (must be ≤ 0.5)
```



```
The sharp edges smooth out first (high curvature → fast diffusion)
Eventually the temperature decays to zero everywhere
Total heat: initial = 14.44, final = 0.18
```

11.10 X. Solving PDEs with FFT: Spectral Methods

All the methods above use **finite differences** — local approximations to derivatives. There's a completely different approach that uses the **FFT** from Lecture 09.

11.10.1 The Key Insight: Derivatives Become Multiplication

Recall from Fourier analysis that the Fourier transform turns derivatives into algebraic operations:

$$\mathcal{F}\left\{\frac{\partial f}{\partial x}\right\} = ik\,\hat{f}(k)$$

$$\mathcal{F}\left\{\frac{\partial^2 f}{\partial x^2}\right\} = (ik)^2\,\hat{f}(k) = -k^2\,\hat{f}(k)$$

where k is the wavenumber and $\hat{f}(k)$ is the Fourier transform of $f(x)$.

This means we can compute spatial derivatives **exactly** (to machine precision!) by:

1. FFT the function: $f(x) \to \hat{f}(k)$
2. Multiply by $-k^2$: $\hat{f}(k) \to -k^2\,\hat{f}(k)$
3. Inverse FFT: $-k^2\,\hat{f}(k) \to f''(x)$

No finite difference approximation needed!

11.10.2 Application to the Heat Equation

Take the heat equation $\partial T/\partial t = \alpha\,\partial^2 T/\partial x^2$ and Fourier-transform in space:

$$\frac{\partial \hat{T}(k,t)}{\partial t} = -\alpha k^2\,\hat{T}(k,t)$$

This is now an **ODE for each wavenumber** k — and it has an exact solution:

$$\hat{T}(k,t) = \hat{T}(k,0)\,e^{-\alpha k^2 t}$$

Algorithm (analytical spectral method):

1. FFT the initial condition: $T(x,0) \to \hat{T}(k,0)$
2. Multiply each mode by the decay factor: $\hat{T}(k,0) \cdot e^{-\alpha k^2 t}$
3. Inverse FFT to get $T(x,t)$

No time stepping at all! We jump directly to any future time.

11.10.3 Why Spectral Methods Are Special

Property	Finite Difference	Spectral (FFT)
Spatial accuracy	$O(\Delta x^2)$	**Exponential** (machine precision for smooth functions)
Stability limit	CFL: $r \leq 1/2$	None (analytical solution)
Boundary conditions	Any (Dirichlet, Neumann, …)	**Periodic only** (limitation!)
Cost per step	$O(N)$	$O(N \log N)$
Best for	General geometries, non-periodic BC	Smooth periodic problems, turbulence

Limitation: The standard FFT assumes **periodic boundary conditions**. For non-periodic problems, you need basis functions other than e^{ikx} — e.g., Chebyshev polynomials (Chebyshev spectral methods).

11.10.4 Spectral Time-Stepping (for more general PDEs)

When you can't solve analytically (e.g., nonlinear PDEs), use the FFT just for spatial derivatives:

1. Compute $\partial^2 T/\partial x^2$ spectrally: `FFT → multiply by -k² → IFFT`
2. Time-step with any ODE method (Euler, RK4, etc.)

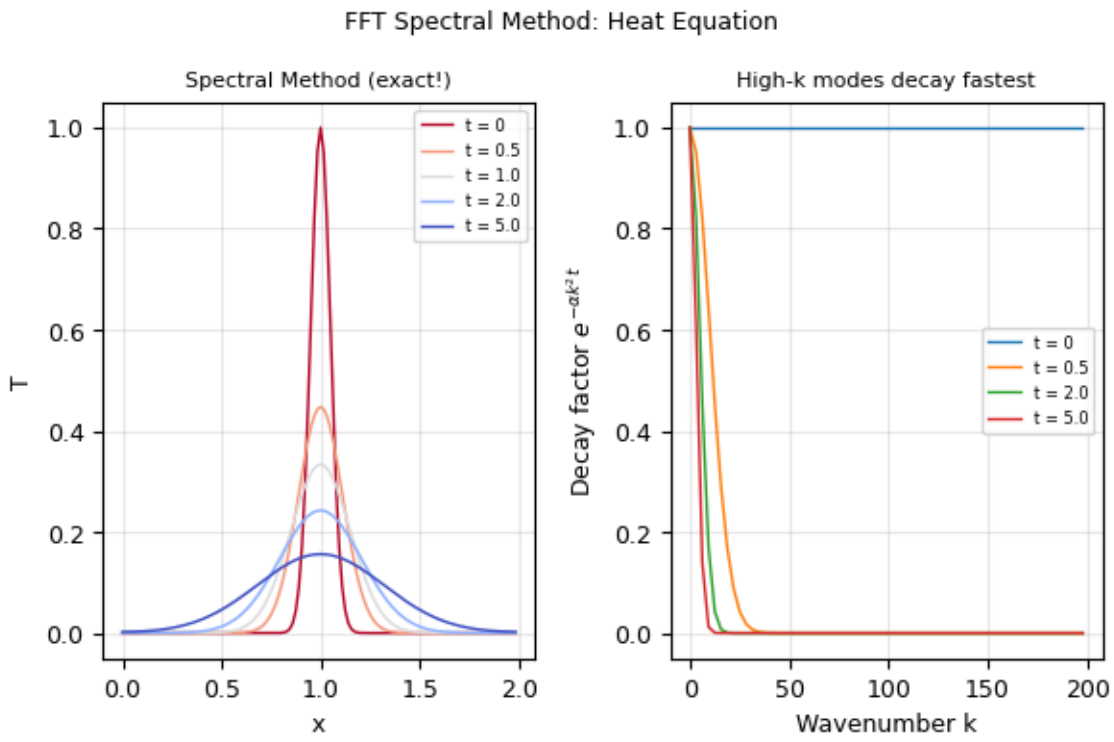
This gives **spectral accuracy in space** with your choice of time integrator.

```
# ---- Spectral Method for the Heat Equation ----
# Compare: FTCS (finite difference) vs Spectral (FFT)

# Setup: periodic domain [0, L) with Gaussian initial condition
L_sp = 2.0
N_sp = 128          # Grid points (power of 2 for FFT efficiency)
alpha_sp = 0.01     # Thermal diffusivity
dx_sp = L_sp / N_sp
x_sp = np.linspace(0, L_sp, N_sp, endpoint=False)  # Periodic: no repeated endpoint

# Initial condition: Gaussian (periodic-compatible)
T0_sp = np.exp(-((x_sp - 1.0)**2) / (2 * 0.05**2))

# ---- Method 1: Analytical spectral solution ----
# Wavenumbers for FFT (numpy convention)
k = 2 * np.pi * np.fft.fftfreq(N_sp, d=dx_sp)

# FFT of initial condition
T0_hat = np.fft.fft(T0_sp)

# Solve at multiple times — NO time stepping needed!
t_targets = [0, 0.5, 1.0, 2.0, 5.0]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(6, 4))

colors_sp = plt.cm.coolwarm(np.linspace(1, 0, len(t_targets)))
for idx, t_val in enumerate(t_targets):
    # Exact spectral solution: multiply each mode by exp(-α k² t)
    decay = np.exp(-alpha_sp * k**2 * t_val)
```

(continues on next page)

(continued from previous page)

```
    T_hat_t = T0_hat * decay
    T_spectral = np.real(np.fft.ifft(T_hat_t))
    ax1.plot(x_sp, T_spectral, color=colors_sp[idx], linewidth=1,
             label=f't = {t_val}')

ax1.set_xlabel('x')
ax1.set_ylabel('T')
ax1.set_title('Spectral Method (exact!)', fontsize=8)
ax1.legend(fontsize=6)
ax1.grid(True, alpha=0.3)

# ---- Show the decay of Fourier modes ----
k_pos = k[:N_sp//2]
for t_val in [0, 0.5, 2.0, 5.0]:
    decay = np.exp(-alpha_sp * k_pos**2 * t_val)
    ax2.plot(k_pos, decay, linewidth=1, label=f't = {t_val}')

ax2.set_xlabel('Wavenumber k')
ax2.set_ylabel('Decay factor $e^{-\\alpha k^2 t}$')
ax2.set_title('High-k modes decay fastest', fontsize=8)
ax2.legend(fontsize=6)
ax2.grid(True, alpha=0.3)

plt.suptitle('FFT Spectral Method: Heat Equation', fontsize=9)
plt.tight_layout()
plt.show()

print("Spectral method: NO time stepping, NO stability limit!")
print("We jump directly to any time t by multiplying Fourier modes by exp(-αk²t).")
print("High-frequency modes (large k) decay fastest — that's why diffusion smooths␣
 ↪things out.")
```




```
Spectral method: NO time stepping, NO stability limit!
We jump directly to any time t by multiplying Fourier modes by exp(-αk²t).
High-frequency modes (large k) decay fastest — that's why diffusion smooths things␣
↪out.
```

CHAPTER

TWELVE

LECTURE 12: STOCHASTIC METHODS I

Computational Physics — Spring 2026

12.1 Stochastic Methods I: The Art of Randomness

12.2 Introduction to Randomness in Physics

Nature is filled with phenomena that exhibit random or seemingly random behavior. From the unpredictable decay of radioactive nuclei to the erratic Brownian motion of particles suspended in a fluid, randomness is deeply embedded in the fabric of physical reality. These stochastic processes play a crucial role in various fields:

- **Quantum mechanics**: The inherently probabilistic nature of quantum measurements
- **Statistical mechanics**: Random molecular motions underlying thermodynamic properties
- **Astrophysics**: Random stellar formations and galaxy distributions
- **Biological physics**: Random fluctuations in cellular processes

To model these phenomena computationally, we need reliable methods to generate random numbers. But this raises an interesting paradox: how can we create randomness using deterministic computers?

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import special, stats
from scipy.ndimage import label

plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

print("Ready for stochastic adventures!")
```

```
Ready for stochastic adventures!
```

12.3 I. Random Numbers and Distributions

12.3.1 Pseudo-Random Number Generation

True randomness is difficult to achieve on a computer. Instead, we use algorithms that produce sequences of numbers that *appear* random but are actually generated by deterministic formulas. These are called **pseudo-random number generators (PRNGs)**.

A good PRNG should have:

- Long periods before repeating sequences
- Statistical properties resembling true randomness
- Efficiency in computation
- Reproducibility when needed (for debugging)

12.3.2 Linear Congruential Generator (LCG)

The Linear Congruential Generator (LCG) is one of the simplest and most historically significant PRNGs. Despite its simplicity, it illustrates the fundamental principles behind most modern random number generators.

Mathematical Foundation:

The LCG is based on the following recursive formula:

$$X_{n+1} = (a \cdot X_n + c) \mod m$$

Where:

- X_n is the current number in the sequence
- X_{n+1} is the next number in the sequence
- a, c, and m are carefully chosen integer constants
- X_0 is the initial value, called the "seed"

This formula generates a sequence of integers between 0 and $m - 1$.

Key Properties:

1. **Deterministic**: Given the same parameters and seed, the sequence will always be identical
2. **Periodic**: The sequence will eventually repeat with a maximum period of m
3. **Parameter-sensitive**: The quality of randomness critically depends on the choice of a, c, and m

Let's examine a basic implementation:

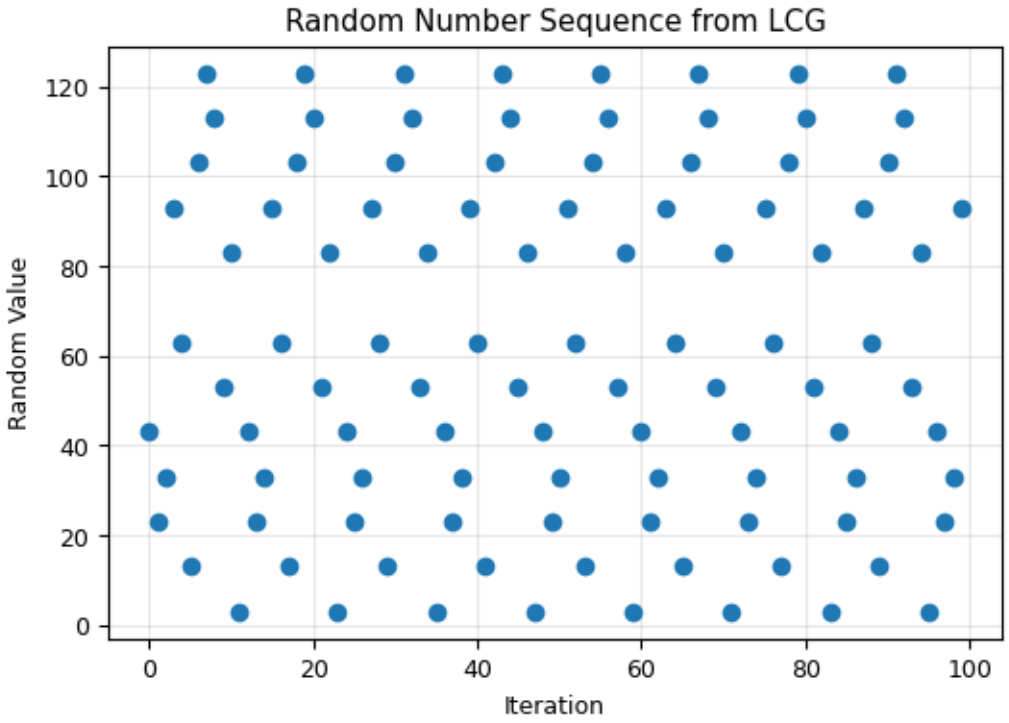
```
N = 100  # Number of random numbers to generate
a = 1111  # Multiplier
c = 1000 # Increment
m = 130    # Modulus
x = 3     # Initial seed
results = []

for i in range(N):
    x = (a*x + c) % m
    results.append(x)
```

(continues on next page)

(continued from previous page)

```
print(results)

plt.plot(results, "o")
plt.title("Random Number Sequence from LCG")
plt.xlabel("Iteration")
plt.ylabel("Random Value")
plt.grid(alpha=0.3)
plt.show()
```

```
[43, 23, 33, 93, 63, 13, 103, 123, 113, 53, 83, 3, 43, 23, 33, 93, 63, 13, 103,␣
↪123, 113, 53, 83, 3, 43, 23, 33, 93, 63, 13, 103, 123, 113, 53, 83, 3, 43, 23,␣
↪33, 93, 63, 13, 103, 123, 113, 53, 83, 3, 43, 23, 33, 93, 63, 13, 103, 123, 113,␣
↪53, 83, 3, 43, 23, 33, 93, 63, 13, 103, 123, 113, 53, 83, 3, 43, 23, 33, 93, 63,␣
↪13, 103, 123, 113, 53, 83, 3, 43, 23, 33, 93, 63, 13, 103, 123, 113, 53, 83, 3,␣
↪43, 23, 33, 93]
```



As shown above, we calculated the first 100 numbers in the sequence generated by the LCG equation with parameters $\{a, c, m\}$. The generated numbers appear to be quite random.

This is a **linear congruential random number generator**, which generates a string of random integers by iterating the same equation many times. It is the most fundamental way of generating random numbers on a computer.

Key observations:

1. The numbers are **not truly random**. If you know the values of $\{a, c, m\}$ plus the seed, you can exactly predict the sequence. Using the same parameters always gives the same output.
2. The numbers are always **non-negative** (between 0 and $m - 1$).
3. The results are **very sensitive** to the choice of $\{a, c, m\}$. For example, if both c and m are even, the generator would only produce even or only odd numbers. It is wise to use only values which have been thoroughly tested.

4. For a particular set of $\{a, c, m\}$, you can still get different sequences by varying the initial value of X_0. This initial value is called the **seed** of the random number generator.

Provided that one is aware of these conditions, the linear congruential generator can be used to produce pseudo-random numbers for simple calculations. In practice, modern libraries like NumPy use far more sophisticated algorithms (e.g., the Mersenne Twister with period $2^{19937} - 1$).

12.3.3 LCG as a Python Class and Function

For better usability, we can encapsulate the LCG algorithm in a class or a standalone function:

```
class LCG:
    def __init__(self, seed, a, c, m):
        self.state = seed
        self.a = a
        self.c = c
        self.m = m

    def next_random(self):
        self.state = (self.a * self.state + self.c) % self.m
        return self.state
```

```
# Standalone function version
def lcg(seed, a, c, m):
    """One step of a Linear Congruential Generator."""
    r = (a * seed + c) % m
    return r

seed = 12
a = 3330    # Multiplier
c = 1630    # Increment
m = 13      # Modulus

r1 = lcg(seed, a, c, m)
print(f"seed={seed} -> r1={r1}")

r2 = lcg(r1, a, c, m)
print(f"r1={r1}   -> r2={r2}")

r3 = lcg(r2, a, c, m)
print(f"r2={r2}   -> r3={r3}")
```

```
seed=12 -> r1=3
r1=3   -> r2=11
r2=11   -> r3=1
```

```
# Class version — maintains internal state
lcg_gen = LCG(seed=14, a=16231, c=1012, m=13)

for i in range(5):
    r = lcg_gen.next_random()
    print(f"Step {i+1}: {r}")
```

```
Step 1: 5
Step 2: 7
```

(continues on next page)

(continued from previous page)

```
Step 3: 8
Step 4: 2
Step 5: 12
```

12.3.4 Generating Random Numbers in a Range

To generate a random number in a specific range [min_value, max_value], we can normalize the output of the LCG and then scale it to the desired range.

```
class LCG:
    def __init__(self, seed, a, c, m):
        self.state = seed
        self.a = a
        self.c = c
        self.m = m

    def next_random(self):
        self.state = (self.a * self.state + self.c) % self.m
        return self.state

    def random(self):
        """Return a random float in (0, 1)."""
        return self.next_random()/self.m

    def randint(self, min_value, max_value):
        """Return a random integer in [min_value, max_value)."""
        return int(self.random() * (max_value-min_value) + min_value)


lcg = LCG(seed=1, a=1234, c=567, m=13)

print("Random integers in [-5, 5):")
for i in range(10):
    print(f"  {lcg.randint(-5, 5)}", end="")
print()

print("\nRandom integers in [100, 200):")
for i in range(10):
    print(f"  {lcg.randint(100, 200)}", end="")
print()

print("\nRandom floats in (0, 1):")
for i in range(10):
    print(f"  {lcg.random():.4f}", end="")
print()
```

```
Random integers in [-5, 5):
  0  -4  0  -4  0  -4  0  -4  0  -4

Random integers in [100, 200):
  153  107  153  107  153  107  153  107  153  107

Random floats in (0, 1):
  0.5385  0.0769  0.5385  0.0769  0.5385  0.0769  0.5385  0.0769  0.5385  0.0769
```

12.3.5 Critical Analysis of LCG Properties

1. **Pseudo-randomness**: Despite appearing random, the sequence is entirely deterministic. Given the parameters and seed, anyone can predict every number in the sequence.
2. **Value Range**: The raw LCG output is always between 0 and $m - 1$, inclusive. This is important to consider when scaling to different ranges.
3. **Parameter Sensitivity**: The quality of randomness critically depends on the choice of parameters:
   - If both c and m are even, the sequence will alternate between even and odd numbers or get stuck in even worse patterns
   - Poor parameter choices can lead to very short periods
   - Some combinations create strong correlations between consecutive numbers
4. **Seed Dependence**: Different seeds produce different sequences, but the quality characteristics remain determined by the main parameters.
5. **Periodicity**: All LCGs eventually repeat their sequences. The maximum theoretical period is m, but many parameter combinations result in much shorter periods.

12.3.6 The Art of Choosing Parameters

The choice of parameters $\{a, c, m\}$ is crucial and non-trivial. After decades of mathematical analysis, certain values have been proven to yield optimal results.

Hull-Dobell Theorem: For a full period (m values before repeating), the following must hold:

1. c and m are relatively prime
2. $a - 1$ is divisible by all prime factors of m
3. If m is divisible by 4, then $a - 1$ is also divisible by 4

An excellent and well-tested choice for 32-bit computers is:

- $a = 7^5 = 16807$
- $c = 0$
- $m = 2^{31} - 1 = 2{,}147{,}483{,}647$

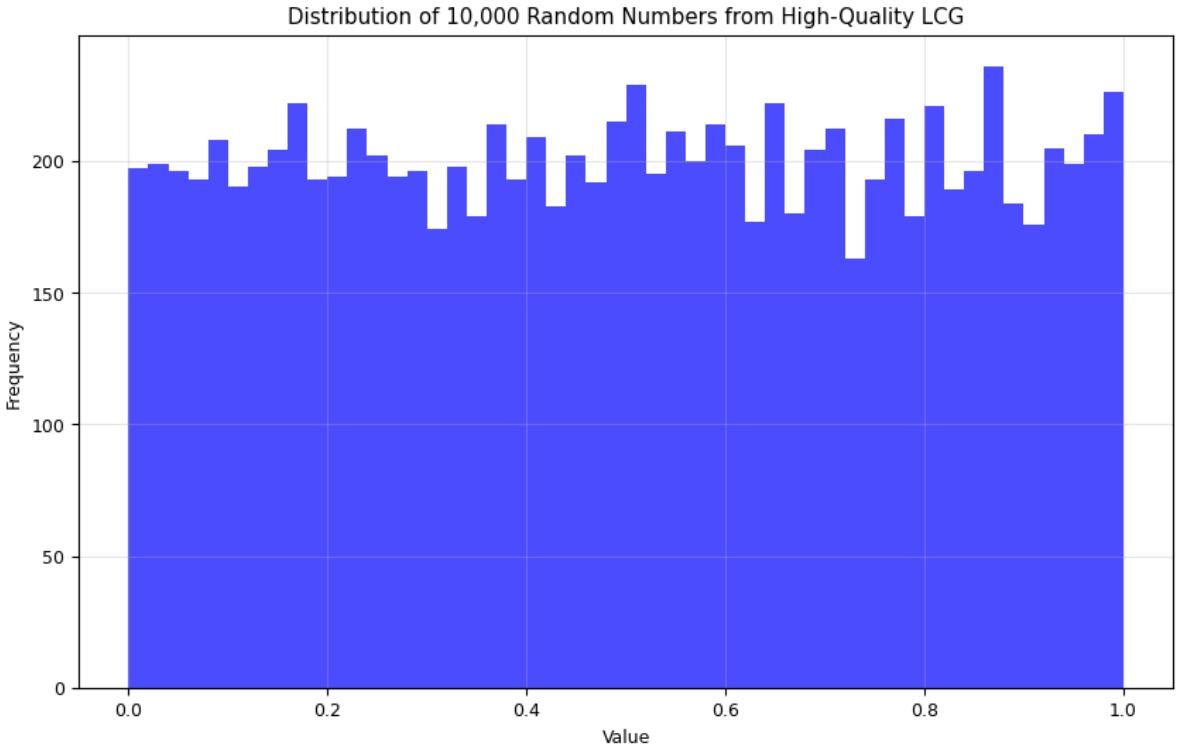
These parameters create a **multiplicative congruential generator** (a special case of LCG where $c = 0$) that passes many statistical tests for randomness and has an impressively long period.

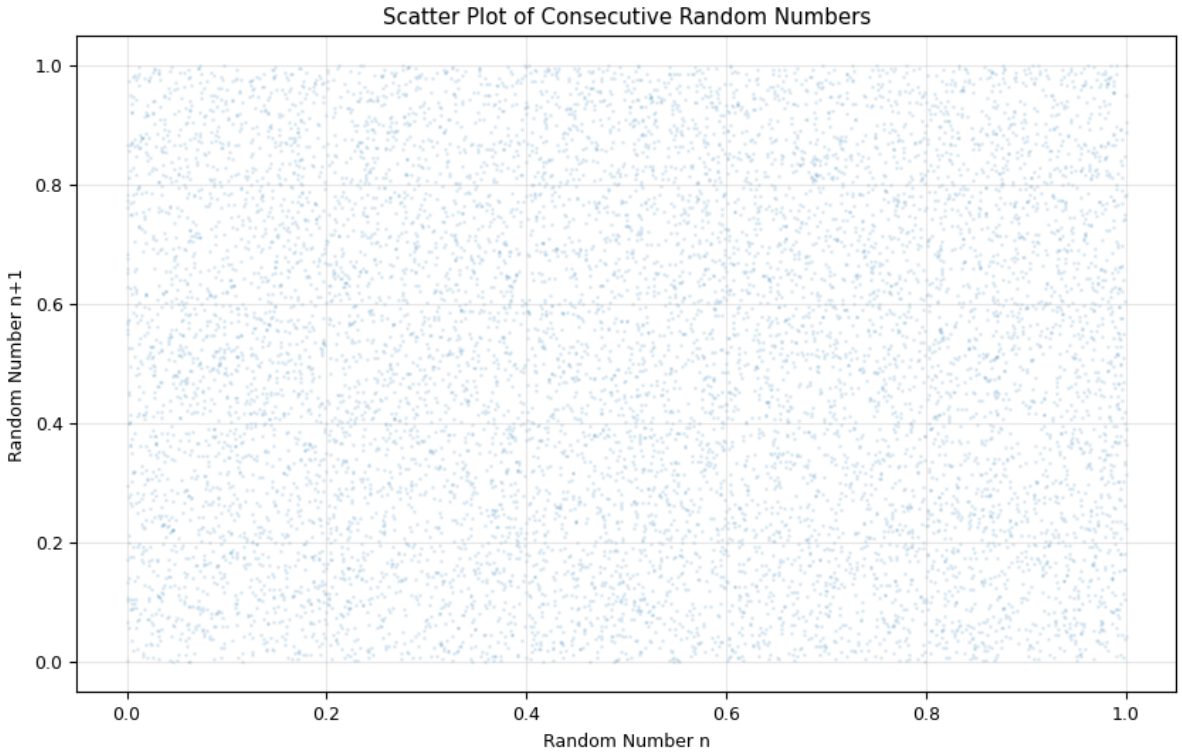
```
# Let's demonstrate the high-quality parameters
lcg = LCG(seed=1, a=16807, c=0, m=2**31-1)

### random seed, (year + month + day + hour + min + second ) (import time )

# Generate a large sample of random numbers
samples = [lcg.random() for _ in range(10000)]

# Plot histogram to check distribution
plt.figure(figsize=(10, 6))
plt.hist(samples, bins=50, alpha=0.7, color='blue')
plt.title('Distribution of 10,000 Random Numbers from High-Quality LCG')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.grid(alpha=0.3)
```

(continues on next page)

(continued from previous page)

```
plt.show()

# Plot scatter to check for patterns between consecutive numbers
plt.figure(figsize=(10, 6))
plt.scatter(samples[:-1], samples[1:], alpha=0.1, s=1)
plt.title('Scatter Plot of Consecutive Random Numbers')
plt.xlabel('Random Number n')
plt.ylabel('Random Number n+1')
plt.grid(alpha=0.3)
plt.show()
```





12.3.7 NumPy Random Number Tools

In practice, we use NumPy's well-tested random number generators rather than writing our own. NumPy uses the Mersenne Twister algorithm by default — far superior to a simple LCG.

Key functions:

- `np.random.seed(n)`: Set the seed for reproducibility
- `np.random.uniform(0, 1, N)`: N uniform [0,1) samples
- `np.random.normal(μ, σ, N)`: N Gaussian samples
- `np.random.exponential(λ, N)`: N exponential samples

```
np.random.seed(42)

N = 100000
uniform_samples = np.random.uniform(0, 1, N)
normal_samples = np.random.normal(0, 1, N)
exponential_samples = np.random.exponential(1, N)

print("Uniform [0,1):")
print(f"  Mean: {np.mean(uniform_samples):.4f} (expected 0.5)")
print(f"  Std:  {np.std(uniform_samples):.4f} (expected {1/np.sqrt(12):.4f})")

print("\nNormal(mu=0, sigma=1):")
print(f"  Mean: {np.mean(normal_samples):.4f} (expected 0)")
print(f"  Std:  {np.std(normal_samples):.4f} (expected 1)")

print("\nExponential(lambda=1):")
```

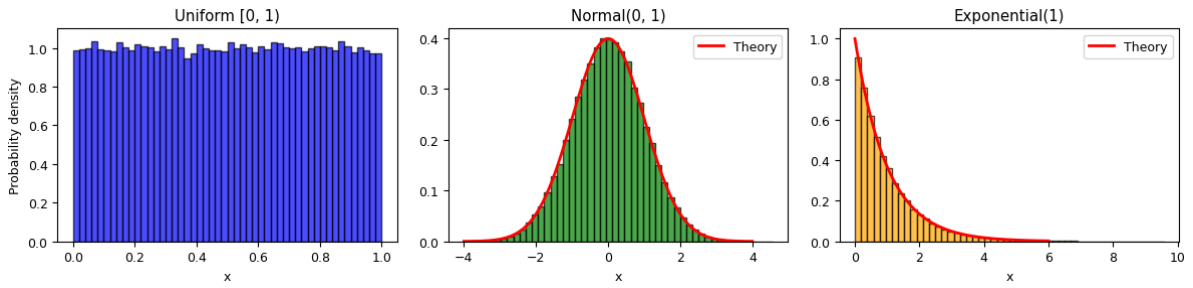
(continues on next page)

(continued from previous page)

```
print(f"  Mean: {np.mean(exponential_samples):.4f} (expected 1)")
print(f"  Std:  {np.std(exponential_samples):.4f} (expected 1)")
```

```
Uniform [0,1):
  Mean: 0.4995 (expected 0.5)
  Std:  0.2883 (expected 0.2887)

Normal(mu=0, sigma=1):
  Mean: 0.0026 (expected 0)
  Std:  0.9983 (expected 1)

Exponential(lambda=1):
  Mean: 1.0002 (expected 1)
  Std:  1.0018 (expected 1)
```

```
fig, axes = plt.subplots(1, 3, figsize=(12, 3))

axes[0].hist(uniform_samples, bins=50, density=True, alpha=0.7, color='blue',␣
↪edgecolor='black')
axes[0].set_title('Uniform [0, 1)')
axes[0].set_xlabel('x')
axes[0].set_ylabel('Probability density')

axes[1].hist(normal_samples, bins=50, density=True, alpha=0.7, color='green',␣
↪edgecolor='black')
x_gauss = np.linspace(-4, 4, 100)
axes[1].plot(x_gauss, stats.norm.pdf(x_gauss), 'r-', lw=2, label='Theory')
axes[1].set_title('Normal(0, 1)')
axes[1].set_xlabel('x')
axes[1].legend()

axes[2].hist(exponential_samples, bins=50, density=True, alpha=0.7, color='orange',␣
↪edgecolor='black')
x_exp = np.linspace(0, 6, 100)
axes[2].plot(x_exp, np.exp(-x_exp), 'r-', lw=2, label='Theory')
axes[2].set_title('Exponential(1)')
axes[2].set_xlabel('x')
axes[2].legend()

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

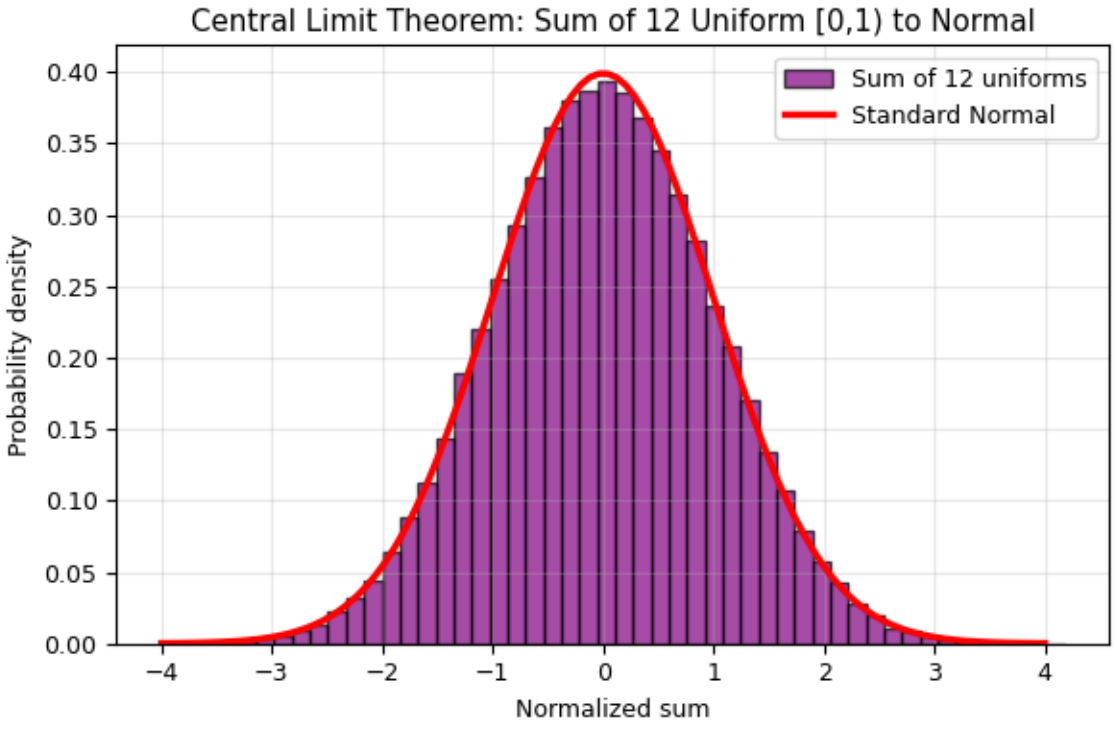


12.3.8 Central Limit Theorem: Sums of Uniforms → Gaussian

Remarkable fact: The sum of many independent random variables approaches a Gaussian distribution, *regardless of the shape of the original distribution*.

Experiment: Take the sum of N uniform [0,1) random variables. Subtract the mean ($N/2$) and divide by the standard deviation ($\sqrt{N/12}$). The result converges to a standard normal as N grows!

```
np.random.seed(123)

N_uniform = 12
N_samples = 100000

sums = np.sum(np.random.uniform(0, 1, (N_samples, N_uniform)), axis=1)
sums_normalized = (sums - N_uniform/2) / np.sqrt(N_uniform/12)

fig, ax = plt.subplots(figsize=(6, 4))
ax.hist(sums_normalized, bins=50, density=True, alpha=0.7, color='purple', edgecolor=
 ↪'black', label=f'Sum of {N_uniform} uniforms')

x = np.linspace(-4, 4, 100)
ax.plot(x, stats.norm.pdf(x), 'r-', lw=2.5, label='Standard Normal')
ax.set_xlabel('Normalized sum')
ax.set_ylabel('Probability density')
ax.set_title('Central Limit Theorem: Sum of 12 Uniform [0,1) to Normal')
ax.legend()
ax.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```



12.4 II. Random Walks and Diffusion

12.4.1 1D Random Walk: Step Left or Right

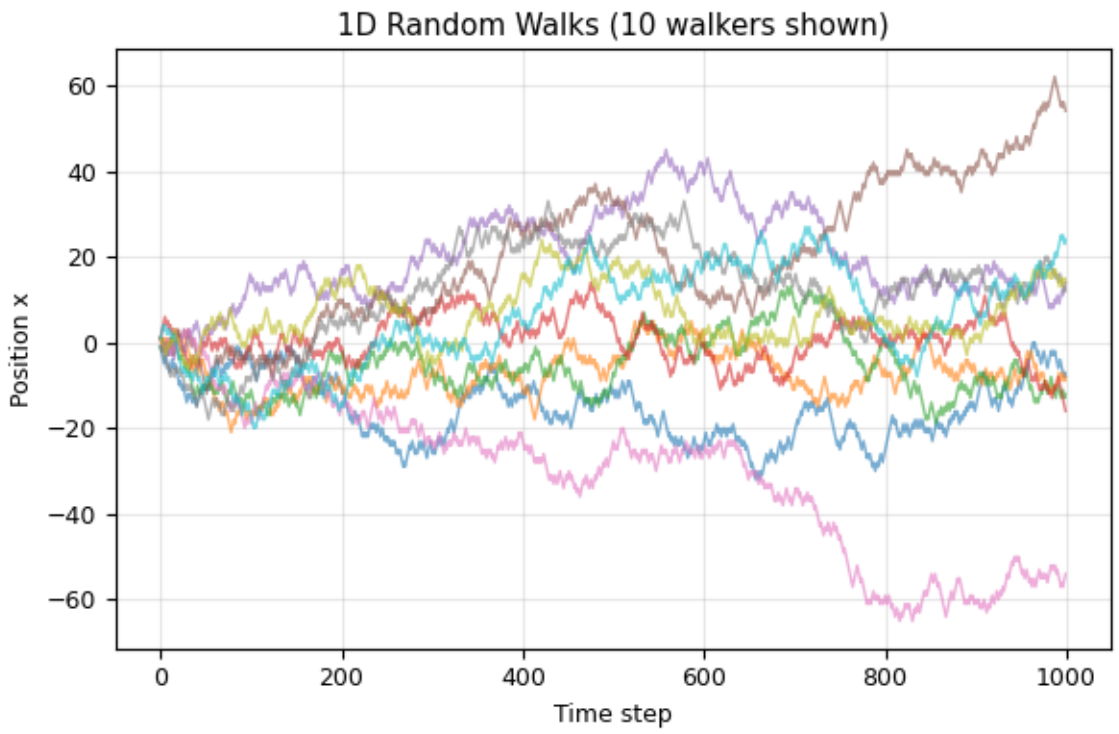
At each time step, a particle moves left (-1) or right (+1) with equal probability.

Key results:

- Mean x = 0 (by symmetry)
- Mean x^2 = N (after N steps)
- Standard deviation: sigma ~ sqrt(N) (diffusive scaling)

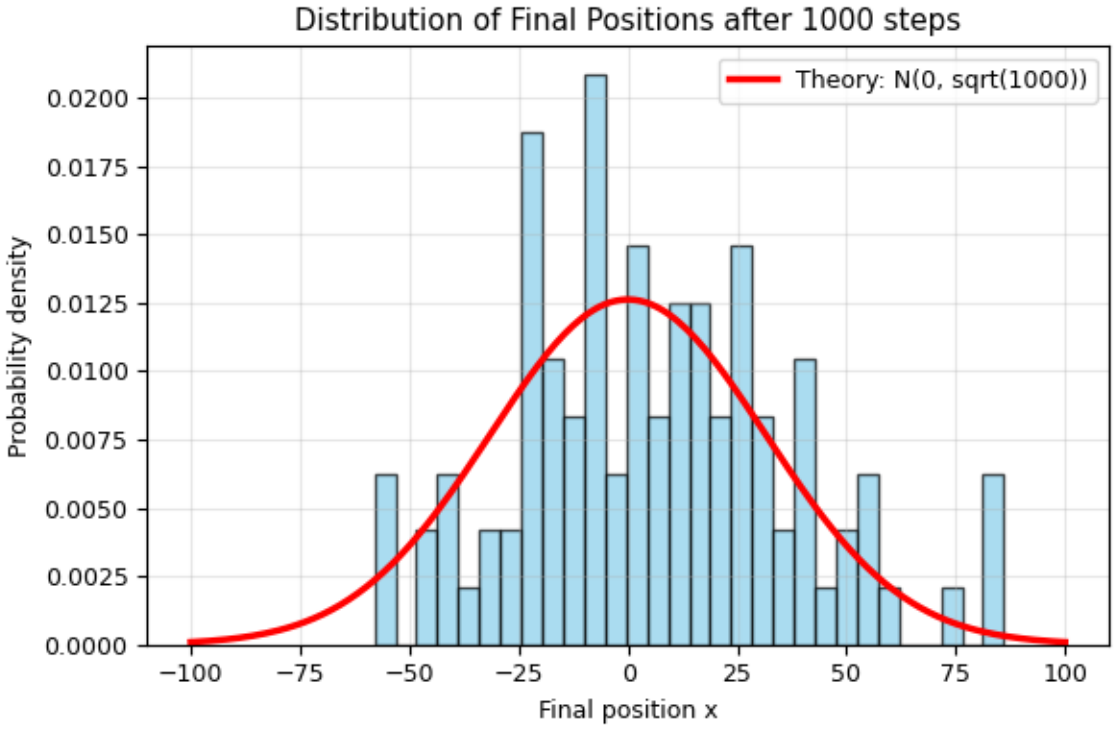
Connection to physics: This is the microscopic origin of diffusion! The random walk in the continuum limit becomes the heat equation: d(rho)/dt = D d^2(rho)/dx^2

```
np.random.seed(111)

n_walkers = 100
n_steps = 1000

steps = np.random.choice([-1, 1], size=(n_walkers, n_steps))
positions = np.cumsum(steps, axis=1)

fig, ax = plt.subplots(figsize=(6, 4))
for i in range(min(10, n_walkers)):
    ax.plot(positions[i, :], alpha=0.6, lw=1)

ax.set_xlabel('Time step')
ax.set_ylabel('Position x')
ax.set_title(f'1D Random Walks ({min(10, n_walkers)} walkers shown)')
ax.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```




```
final_positions = positions[:, -1]
final_positions_squared = final_positions**2

mean_x2 = np.mean(final_positions_squared)
expected_x2 = n_steps

print(f"After N = {n_steps} steps:")
print(f"<x^2> observed: {mean_x2:.0f}")
print(f"<x^2> expected: {expected_x2:.0f}")
print(f"<x>:           {np.mean(final_positions):.2f} (should be approx 0)")
print(f"std(x):        {np.std(final_positions):.1f} approx sqrt({n_steps}) = {np.
 ↪sqrt(n_steps):.1f}")

fig, ax = plt.subplots(figsize=(6, 4))
ax.hist(final_positions, bins=30, density=True, alpha=0.7, color='skyblue', edgecolor=
 ↪'black')

x_range = np.linspace(-100, 100, 200)
sigma = np.sqrt(n_steps)
ax.plot(x_range, stats.norm.pdf(x_range, 0, sigma), 'r-', lw=2.5, label=f'Theory: N(0,
 ↪ sqrt({n_steps}))')

ax.set_xlabel('Final position x')
ax.set_ylabel('Probability density')
ax.set_title(f'Distribution of Final Positions after {n_steps} steps')
ax.legend()
ax.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

```
After N = 1000 steps:
<x^2> observed: 1013
<x^2> expected: 1000
<x>:            6.04 (should be approx 0)
std(x):         31.2 approx sqrt(1000) = 31.6
```



12.5 III. The Metropolis Algorithm

12.5.1 Sampling from Complicated Distributions

Problem: We want to sample from a distribution P(x) proportional to exp(-V(x)/T), but we can't invert it.

Solution: Metropolis Algorithm

1. **Propose** a new state x_new = x + delta_x (random step)
2. **Calculate** acceptance probability: alpha = min(1, exp(-Delta_V/T)) where Delta_V = V(x_new) - V(x)
3. **Accept/reject**:
   - If rand() < alpha: accept x_new (move)
   - Else: stay at x
4. **Repeat** many times to build a chain

Why it works: The algorithm satisfies **detailed balance**, which ensures the stationary distribution is exactly P(x)!

```
def potential(x):
    return (x**2 - 1)**2
```

(continues on next page)

(continued from previous page)

```
def metropolis_sampling(n_steps, step_size, temperature):
    np.random.seed(42)

    x = 0.0
    trajectory = [x]
    n_accept = 0

    for step in range(n_steps):
        x_new = x + np.random.normal(0, step_size)

        dV = potential(x_new) - potential(x)

        if dV < 0 or np.random.rand() < np.exp(-dV / temperature):
            x = x_new
            n_accept += 1

        trajectory.append(x)

    acceptance_rate = n_accept / n_steps
    return np.array(trajectory), acceptance_rate

T = 0.5
step_size = 0.5
n_steps = 5000

trajectory, acc_rate = metropolis_sampling(n_steps, step_size, T)

print(f"Metropolis Sampling from Double-Well Potential")
print(f"Temperature T = {T}")
print(f"Step size = {step_size}")
print(f"Acceptance rate: {acc_rate:.1%}")
print(f"\nMarkov chain statistics:")
print(f"Mean position: {np.mean(trajectory[1000:]):.3f}")
print(f"Std deviation: {np.std(trajectory[1000:]):.3f}")
```

```
Metropolis Sampling from Double-Well Potential
Temperature T = 0.5
Step size = 0.5
Acceptance rate: 60.4%

Markov chain statistics:
Mean position: -0.185
Std deviation: 0.897
```

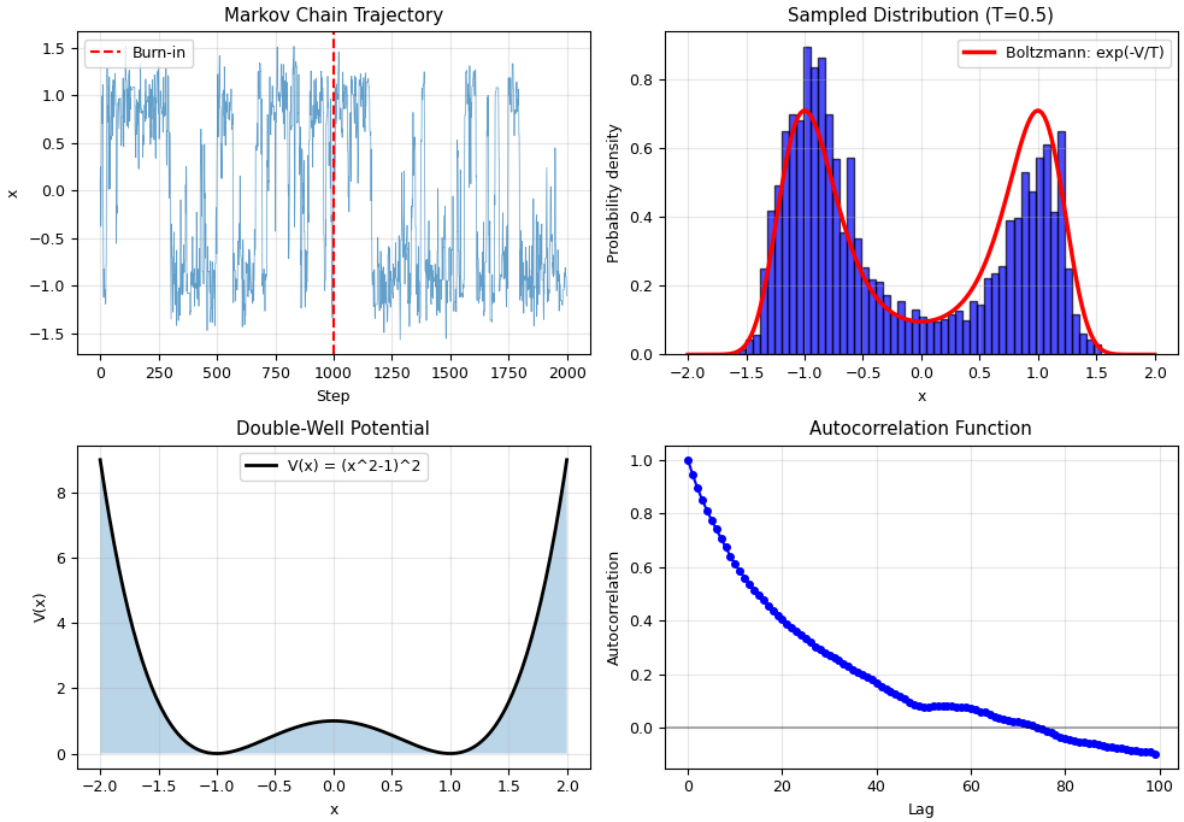
```
burn_in = 1000
samples = trajectory[burn_in:]

def autocorrelation(signal, max_lag=100):
    c = np.correlate(signal - np.mean(signal), signal - np.mean(signal), mode='full')
    c = c[len(c)//2:]
    return c / c[0]

fig, axes = plt.subplots(2, 2, figsize=(10, 7))

axes[0, 0].plot(trajectory[:2000], linewidth=0.5, alpha=0.7)
axes[0, 0].axvline(burn_in, color='r', linestyle='--', label='Burn-in')
```

(continues on next page)

(continued from previous page)

```
axes[0, 0].set_xlabel('Step')
axes[0, 0].set_ylabel('x')
axes[0, 0].set_title('Markov Chain Trajectory')
axes[0, 0].legend()
axes[0, 0].grid(alpha=0.3)

axes[0, 1].hist(samples, bins=50, density=True, alpha=0.7, color='blue', edgecolor=
 ↪'black')

x_range = np.linspace(-2, 2, 200)
V_range = potential(x_range)
P_exact = np.exp(-V_range / T)
P_exact = P_exact / np.trapz(P_exact, x_range)
axes[0, 1].plot(x_range, P_exact, 'r-', lw=2.5, label='Boltzmann: exp(-V/T)')
axes[0, 1].set_xlabel('x')
axes[0, 1].set_ylabel('Probability density')
axes[0, 1].set_title(f'Sampled Distribution (T={T})')
axes[0, 1].legend()
axes[0, 1].grid(alpha=0.3)

axes[1, 0].plot(x_range, V_range, 'k-', lw=2, label='V(x) = (x^2-1)^2')
axes[1, 0].fill_between(x_range, V_range, alpha=0.3)
axes[1, 0].set_xlabel('x')
axes[1, 0].set_ylabel('V(x)')
axes[1, 0].set_title('Double-Well Potential')
axes[1, 0].legend()
axes[1, 0].grid(alpha=0.3)

acf = autocorrelation(samples, max_lag=100)
axes[1, 1].plot(acf[:100], 'bo-', ms=4)
axes[1, 1].axhline(0, color='k', linestyle='-', alpha=0.3)
axes[1, 1].set_xlabel('Lag')
axes[1, 1].set_ylabel('Autocorrelation')
axes[1, 1].set_title('Autocorrelation Function')
axes[1, 1].grid(alpha=0.3)

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

```
/tmp/ipykernel_828/272206957.py:24: DeprecationWarning: `trapz` is deprecated. Use␣
 ↪`trapezoid` instead, or one of the numerical integration functions in `scipy.
 ↪integrate`.
  P_exact = P_exact / np.trapz(P_exact, x_range)
```



12.6 III. Application: The 2D Ising Model (Phase Transition!)

12.6.1 The Ising Model: Spins on a Lattice

System: Spins s_i = +/-1 on a 2D square lattice

Energy: E = -J Σ $s_i s_j$ over nearest neighbors

- J > 0: Ferromagnetic (spins want to align)
- Each aligned pair contributes minus J (lower energy)

Temperature effects:

- T to 0: System is ordered (all spins aligned)
- T to infinity: System is random (thermal fluctuations dominate)
- T_c ≈ 2.269 J/k_B: Critical temperature (phase transition!)

Magnetization: M = |sum s_i| / N

- M ≈ 1 at low T (ordered)
- M ≈ 0 at high T (disordered)

```
def ising_metropolis_step(spins, J, T):
    L = len(spins)
    n_accept = 0
```

(continues on next page)

(continued from previous page)

```
    for i in range(L):
        for j in range(L):
            s_old = spins[i, j]

            neighbors = (spins[(i-1) % L, j] + spins[(i+1) % L, j] +
                         spins[i, (j-1) % L] + spins[i, (j+1) % L])
            dE = 2 * J * s_old * neighbors

            if dE < 0 or np.random.rand() < np.exp(-dE / T):
                spins[i, j] = -s_old
                n_accept += 1

    return n_accept

def run_ising(L, T, n_sweeps):
    J = 1.0

    spins = np.random.choice([-1, 1], size=(L, L))

    magnetization = []
    energy = []

    for sweep in range(n_sweeps):
        ising_metropolis_step(spins, J, T)

        M = np.abs(np.sum(spins)) / (L**2)
        E = 0
        for i in range(L):
            for j in range(L):
                neighbors = (spins[(i-1) % L, j] + spins[(i+1) % L, j] +
                             spins[i, (j-1) % L] + spins[i, (j+1) % L])
                E -= J * spins[i, j] * neighbors / 2
        E = E / (L**2)

        magnetization.append(M)
        energy.append(E)

    return spins, np.array(magnetization), np.array(energy)

print("Ising model functions defined. Ready to simulate!")
```

```
Ising model functions defined. Ready to simulate!
```

```
np.random.seed(42)

L = 20
n_sweeps = 500

temperatures = [0.5, 1.5, 2.269, 3.0, 5.0]
results = {}

print(f"Running Ising model on {L}x{L} lattice...\n")

for T in temperatures:
    spins, mag, eng = run_ising(L, T, n_sweeps)
```

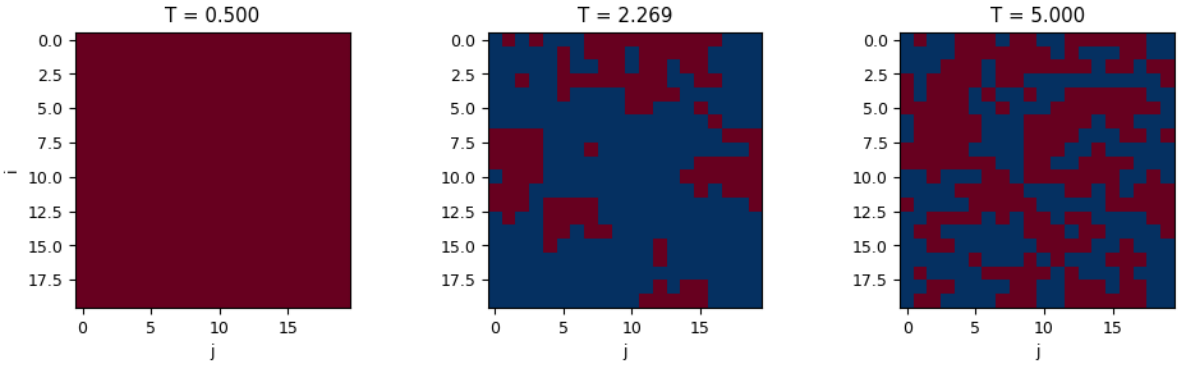
(continues on next page)

(continued from previous page)

```
    results[T] = {
        'spins': spins,
        'magnetization': mag,
        'energy': eng
    }

    m_avg = np.mean(mag[100:])
    e_avg = np.mean(eng[100:])
    print(f"T = {T:5.3f}:  <M> = {m_avg:.3f},  <E> = {e_avg:.4f}")
```

```
Running Ising model on 20x20 lattice...

T = 0.500:  <M> = 1.000,  <E> = -2.0000
T = 1.500:  <M> = 0.987,  <E> = -1.9523
T = 2.269:  <M> = 0.669,  <E> = -1.4376
T = 3.000:  <M> = 0.148,  <E> = -0.8253
T = 5.000:  <M> = 0.066,  <E> = -0.4269
```

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))

temps_to_plot = [0.5, 2.269, 5.0]
for idx, T in enumerate(temps_to_plot):
    spins = results[T]['spins']
    axes[idx].imshow(spins, cmap='RdBu', vmin=-1, vmax=1)
    axes[idx].set_title(f'T = {T:.3f}')
    axes[idx].set_xlabel('j')
    if idx == 0:
        axes[idx].set_ylabel('i')
    else:
        axes[idx].set_ylabel('')

plt.tight_layout()
plt.show()

print("\nSpin configurations:")
print("  T = 0.5:    ORDERED (below Tc) - spins mostly aligned")
print("  T = 2.269:  CRITICAL (at Tc) - domains of mixed sizes")
print("  T = 5.0:    DISORDERED (above Tc) - random spins")
```




```
Spin configurations:
  T = 0.5:    ORDERED (below Tc) - spins mostly aligned
  T = 2.269:  CRITICAL (at Tc) - domains of mixed sizes
```

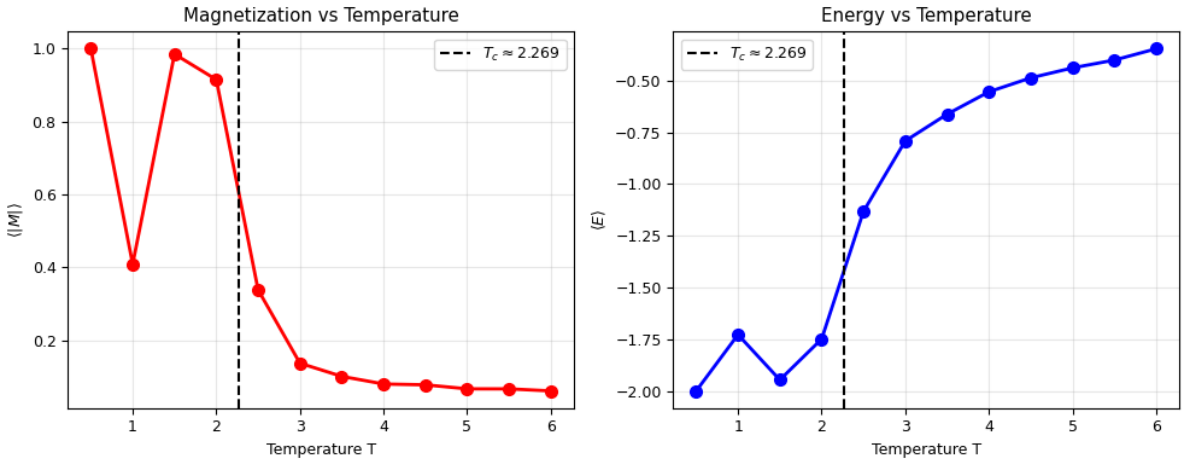
(continues on next page)

(continued from previous page)

```
  T = 5.0:    DISORDERED (above Tc) - random spins
```

```
T_range = np.linspace(0.5, 6.0, 12)
magnetization_avg = []
energy_avg = []

print(f"Scanning temperature from {T_range[0]:.1f} to {T_range[-1]:.1f}...\n")

for T in T_range:
    spins, mag, eng = run_ising(L, T, n_sweeps=300)
    m_avg = np.mean(mag[100:])
    e_avg = np.mean(eng[100:])
    magnetization_avg.append(m_avg)
    energy_avg.append(e_avg)

fig, axes = plt.subplots(1, 2, figsize=(10, 4))

axes[0].plot(T_range, magnetization_avg, 'o-', color='red', ms=7, lw=2)
axes[0].axvline(2.269, color='k', linestyle='--', lw=1.5, label='$T_c \\approx 2.269$
 ↪')
axes[0].set_xlabel('Temperature T')
axes[0].set_ylabel('$\\langle |M| \\rangle$')
axes[0].set_title('Magnetization vs Temperature')
axes[0].legend()
axes[0].grid(alpha=0.3)

axes[1].plot(T_range, energy_avg, 'o-', color='blue', ms=7, lw=2)
axes[1].axvline(2.269, color='k', linestyle='--', lw=1.5, label='$T_c \\approx 2.269$
 ↪')
axes[1].set_xlabel('Temperature T')
axes[1].set_ylabel('$\\langle E \\rangle$')
axes[1].set_title('Energy vs Temperature')
axes[1].legend()
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print(f"\nPhase transition occurs at Tc ≈ 2.269 J/kB")
print(f"Below Tc: Ferromagnetic phase (M > 0)")
print(f"Above Tc: Paramagnetic phase (M ≈ 0)")
```

```
Scanning temperature from 0.5 to 6.0...
```




```
Phase transition occurs at Tc ≈ 2.269 J/kB
Below Tc: Ferromagnetic phase (M > 0)
Above Tc: Paramagnetic phase (M ≈ 0)
```

CHAPTER

THIRTEEN

LECTURE 13: STOCHASTIC METHODS II — MONTE CARLO INTEGRATION

Computational Physics — Spring 2026

13.1 From Random Numbers to Powerful Integrals

In the previous lecture, we learned how to generate random numbers and use them for random walks, Metropolis sampling, and the Ising model. Today we turn randomness into a **computational tool for integration**.

The key insight: in high dimensions, deterministic quadrature (trapezoidal, Simpson's, Gaussian) becomes hopelessly expensive — the number of grid points scales as N^d. Monte Carlo integration sidesteps this **curse of dimensionality** because its error scales as $1/\sqrt{N}$ *regardless of dimension*.

Today's roadmap:

- Monte Carlo integration and its error analysis
- Importance sampling to reduce variance
- Rejection sampling to generate arbitrary distributions
- Application: Variational Monte Carlo for the hydrogen atom
- The classic π estimation as a geometric MC example

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

plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

print("Ready for Monte Carlo integration!")
```

```
Ready for Monte Carlo integration!
```

13.2 I. Monte Carlo Integration

13.2.1 The Basic Idea

To compute the integral $I = \int_a^b f(x)\,dx$, we can rewrite it as an **expectation value**:

$$I = (b-a)\langle f \rangle = (b-a) \cdot \frac{1}{N} \sum_{i=1}^{N} f(x_i)$$

where x_i are drawn uniformly from $[a, b]$.

13.2.2 Proof: Error Scales as $1/\sqrt{N}$

Define the sample mean $\bar{f}_N = \frac{1}{N}\sum_{i=1}^{N} f(x_i)$. Each $f(x_i)$ is an i.i.d. random variable with:

- Mean: $\mu = \langle f \rangle = \frac{1}{b-a}\int_a^b f(x)\,dx$
- Variance: $\sigma^2 = \langle f^2 \rangle - \langle f \rangle^2$

By the **Central Limit Theorem**, the sample mean $\bar{f}_N$ is approximately normal:

$$\bar{f}_N \sim \mathcal{N}\left(\mu,\ \frac{\sigma^2}{N}\right)$$

Therefore the **standard error** of the MC estimate is:

$$\epsilon_{\text{MC}} = (b-a) \cdot \frac{\sigma}{\sqrt{N}}$$

Key consequence: To halve the error, we need 4× more samples. This scaling is *independent of dimension* d, which is the great advantage of MC over deterministic quadrature.

Method	Error scaling (1D)	Error scaling (d-D)
Trapezoidal	$O(N^{-2})$	$O(N^{-2/d})$
Simpson's	$O(N^{-4})$	$O(N^{-4/d})$
Monte Carlo	$O(N^{-1/2})$	$O(N^{-1/2})$

13.2.3 1D Example: $\int_0^1 e^{-x^2}\,dx$

The exact value is $\frac{\sqrt{\pi}}{2}\,\text{erf}(1) \approx 0.7468$. Let's compute it with MC and verify the $1/\sqrt{N}$ error scaling.

```
from scipy.special import erf

exact = np.sqrt(np.pi) / 2 * erf(1)
print(f"Exact value: {exact:.10f}")

np.random.seed(42)

# MC integration for various N
N_values = np.logspace(1, 8, 30, dtype=int)
N_values = np.unique(N_values)
```

(continues on next page)

(continued from previous page)

```
mc_estimates = []
mc_errors = []

for N in N_values:
    x = np.random.uniform(0, 1, N)
    f_x = np.exp(-x**2)

    # sum(f_x) / N
    # MC estimate: (b-a) * mean(f)
    I_mc = np.mean(f_x)  # b-a = 1

    # Estimated standard error
    sigma = np.std(f_x, ddof=1)
    error = sigma / np.sqrt(N)

    mc_estimates.append(I_mc)
    mc_errors.append(error)

mc_estimates = np.array(mc_estimates)
mc_errors = np.array(mc_errors)

# Verify convergence
print(f"\nN = {N_values[-1]:>7d}:  MC = {mc_estimates[-1]:.6f},  error = {mc_errors[-
↪1]:.2e}")
print(f"Absolute error: {abs(mc_estimates[-1] - exact):.2e}")
```

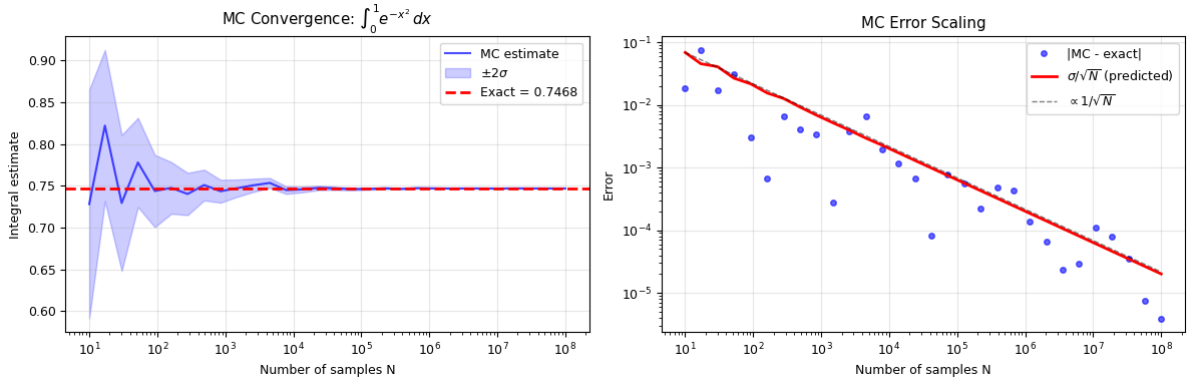
```
Exact value: 0.7468241328

N = 100000000:  MC = 0.746828,  error = 2.01e-05
Absolute error: 3.90e-06
```

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Left: MC estimate vs N
axes[0].semilogx(N_values, mc_estimates, 'b-', alpha=0.7, label='MC estimate')
axes[0].fill_between(N_values, mc_estimates - 2*mc_errors, mc_estimates + 2*mc_errors,
                     alpha=0.2, color='blue', label='$\\pm 2\\sigma$')
axes[0].axhline(exact, color='r', linestyle='--', lw=2, label=f'Exact = {exact:.4f}')
axes[0].set_xlabel('Number of samples N')
axes[0].set_ylabel('Integral estimate')
axes[0].set_title('MC Convergence: $\\int_0^1 e^{-x^2}\\,dx$')
axes[0].legend()
axes[0].grid(alpha=0.3)

# Right: Error vs N (log-log)
actual_errors = np.abs(mc_estimates - exact)
axes[1].loglog(N_values, actual_errors, 'bo', ms=4, alpha=0.6, label='|MC - exact|')
axes[1].loglog(N_values, mc_errors, 'r-', lw=2, label='$\\sigma/\\sqrt{N}$ (predicted)
↪')

# Reference 1/sqrt(N) line
C = mc_errors[0] * np.sqrt(N_values[0])
axes[1].loglog(N_values, C / np.sqrt(N_values), 'k--', lw=1, alpha=0.5, label='$\\
↪propto 1/\\sqrt{N}$')

axes[1].set_xlabel('Number of samples N')
```

(continues on next page)

(continued from previous page)

```
axes[1].set_ylabel('Error')
axes[1].set_title('MC Error Scaling')
axes[1].legend()
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("Error follows 1/sqrt(N) — confirmed!")
```




```
Error follows 1/sqrt(N) — confirmed!
```

13.2.4 The Curse of Dimensionality: Volume of a Hypersphere

The volume of a unit hypersphere in d dimensions is:

$$V_d = \frac{\pi^{d/2}}{\Gamma(d/2+1)}$$

We can estimate this with MC by sampling uniformly in the $[-1,1]^d$ hypercube and counting the fraction of points inside the unit sphere ($\sum x_i^2 \leq 1$):

$$V_d \approx 2^d \cdot \frac{N_{\text{inside}}}{N_{\text{total}}}$$

This works in *any* dimension with the same $1/\sqrt{N}$ scaling!

```
from scipy.special import gamma

def hypersphere_volume_exact(d):
    """Exact volume of unit hypersphere in d dimensions."""
    return np.pi**(d/2) / gamma(d/2 + 1)

def hypersphere_volume_mc(d, N):
    """MC estimate of hypersphere volume by sampling in [-1,1]^d cube."""
    points = np.random.uniform(-1, 1, size=(N, d))
    r_squared = np.sum(points**2, axis=1)
    n_inside = np.sum(r_squared <= 1)

    volume = (2**d) * n_inside / N
```

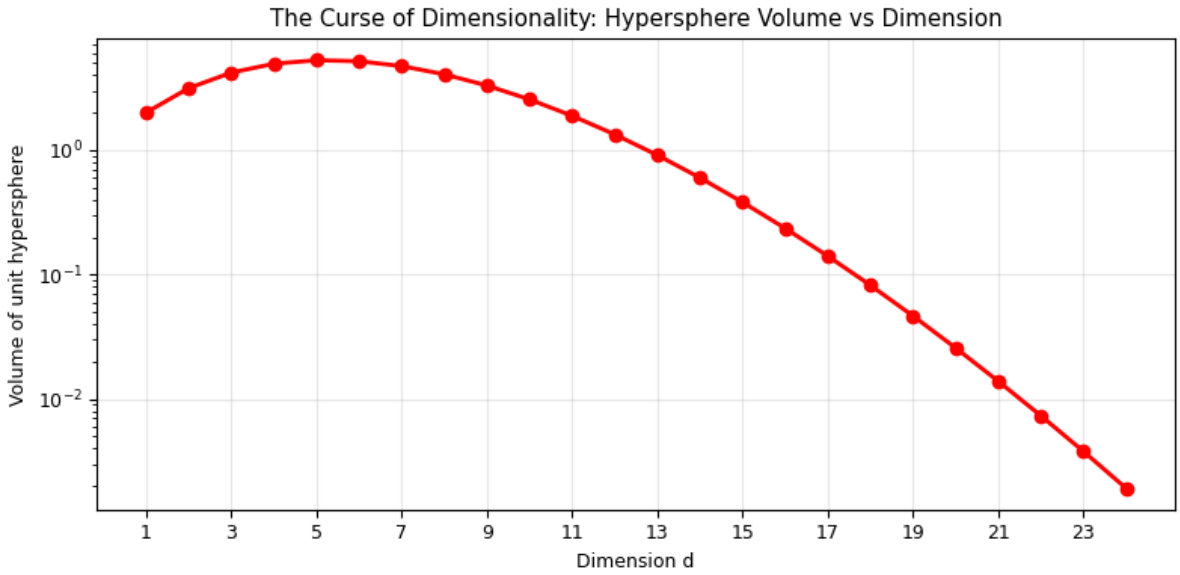
(continues on next page)

(continued from previous page)

```
    # Binomial standard error
    p = n_inside / N
    error = (2**d) * np.sqrt(p * (1 - p) / N)
    return volume, error

np.random.seed(42)
N = 1000000

print(f"Volume of unit hypersphere (N = {N:,} samples)\n")
print(f"{'d':>3s}  {'V_exact':>12s}  {'V_MC':>12s}  {'Error':>10s}  {'Rel.Err':>10s}
 ↪{'Fraction inside':>16s}")
print("-" * 75)

for d in [2, 3, 5, 8, 10, 15, 20]:
    V_exact = hypersphere_volume_exact(d)
    V_mc, err = hypersphere_volume_mc(d, N)
    frac = V_mc / (2**d)
    rel_err = abs(V_mc - V_exact) / V_exact if V_exact > 0 else float('inf')
    print(f"{d:3d}  {V_exact:12.6f}  {V_mc:12.6f}  {err:10.4f}  {rel_err:10.2%}
 ↪{frac:16.6f}")
```

```
Volume of unit hypersphere (N = 1,000,000 samples)

  d       V_exact          V_MC       Error     Rel.Err   Fraction inside
---------------------------------------------------------------------------
  2      3.141593      3.141952      0.0016       0.01%          0.785488
  3      4.188790      4.189088      0.0040       0.01%          0.523636
  5      5.263789      5.246048      0.0118       0.34%          0.163939
  8      4.058712      4.049664      0.0319       0.22%          0.015819
 10      2.550164      2.553856      0.0511       0.14%          0.002494
 15      0.381443      0.524288      0.1311      37.45%          0.000016
 20      0.025807      0.000000      0.0000     100.00%          0.000000
```

```
# Visualize: hypersphere volume shrinks rapidly with dimension
dims = np.arange(1, 25)
volumes_exact = [hypersphere_volume_exact(d) for d in dims]

fig, ax = plt.subplots(figsize=(8, 4))
ax.semilogy(dims, volumes_exact, 'ro-', ms=6, lw=2)
ax.set_xlabel('Dimension d')
ax.set_ylabel('Volume of unit hypersphere')
ax.set_title('The Curse of Dimensionality: Hypersphere Volume vs Dimension')
ax.grid(alpha=0.3)
ax.set_xticks(dims[::2])
plt.tight_layout()
plt.show()

print(f"\nIn d=2: V = {hypersphere_volume_exact(2):.4f} (= π)")
print(f"In d=3: V = {hypersphere_volume_exact(3):.4f} (= 4π/3)")
print(f"In d=10: V = {hypersphere_volume_exact(10):.4f}")
print(f"In d=20: V = {hypersphere_volume_exact(20):.2e}")
print(f"\nThe sphere occupies a vanishingly small fraction of the cube in high␣
 ↪dimensions.")
print(f"This is why high-dimensional integration is hard — most of the volume is in␣
 ↪the corners!")
```




```
In d=2: V = 3.1416 (= π)
In d=3: V = 4.1888 (= 4π/3)
In d=10: V = 2.5502
In d=20: V = 2.58e-02

The sphere occupies a vanishingly small fraction of the cube in high dimensions.
This is why high-dimensional integration is hard — most of the volume is in the␣
↪corners!
```

13.3 II. Importance Sampling

13.3.1 The Problem with Uniform Sampling

Basic MC samples uniformly, but if $f(x)$ is sharply peaked, most samples contribute almost nothing to the integral. We waste computational effort sampling where $f(x) \approx 0$.

13.3.2 The Solution: Sample Where It Matters

Key identity: For any probability density $g(x) > 0$ on $[a, b]$:

$$I = \int_a^b f(x)\,dx = \int_a^b \frac{f(x)}{g(x)}\,g(x)\,dx = \left\langle \frac{f(x)}{g(x)} \right\rangle_{x\sim g}$$

If we draw samples $x_i \sim g(x)$ instead of uniformly, the MC estimate becomes:

$$I \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{g(x_i)}, \quad x_i \sim g(x)$$

13.3.3 Proof: Optimal Importance Sampling Distribution

The variance of the importance-sampled estimator is:

$$\text{Var} = \int \left(\frac{f(x)}{g(x)}\right)^2 g(x)\,dx - I^2 = \int \frac{f(x)^2}{g(x)}\,dx - I^2$$

To minimize this, we use the calculus of variations (or Cauchy-Schwarz). The result is:

$$g^*(x) = \frac{|f(x)|}{\int |f(x)|\,dx}$$

i.e., the optimal sampling distribution is proportional to $|f(x)|$ itself! With $g = g^*$, the variance is **exactly zero** — but this requires knowing $\int |f|$ in advance, which is the integral we're trying to compute.

In practice, we choose $g(x)$ to *approximate* the shape of $|f(x)|$ using a known, easy-to-sample distribution.

13.3.4 Demo: Integrating a Sharply Peaked Function

Consider: $$I = \int_0^{10} e^{-2(x-5)^2}\,dx$$

This is sharply peaked around $x = 5$. Uniform sampling on $[0, 10]$ wastes most samples in the tails.

```
def f_peaked(x):
    """Sharply peaked function."""
    return np.exp(-2 * (x - 5)**2)

# Exact value via scipy
exact_peaked, _ = integrate.quad(f_peaked, 0, 10)
print(f"Exact integral: {exact_peaked:.10f}")

np.random.seed(42)
N = 10000
n_trials = 500  # Repeat to measure variance

# Method 1: Uniform sampling
uniform_estimates = []
for _ in range(n_trials):
    x = np.random.uniform(0, 10, N)
    I_uniform = 10 * np.mean(f_peaked(x))  # (b-a) * mean(f)
    uniform_estimates.append(I_uniform)

# Method 2: Importance sampling with g(x) = Normal(5, 1) truncated to [0,10]
# g(x) = phi((x-5)/1) / (Phi(5) - Phi(-5))  on [0,10]
from scipy.stats import truncnorm

a_trunc, b_trunc = (0 - 5) / 1.0, (10 - 5) / 1.0  # standardized bounds
g_dist = truncnorm(a_trunc, b_trunc, loc=5, scale=1)

importance_estimates = []
for _ in range(n_trials):
    x = g_dist.rvs(N)
    weights = f_peaked(x) / g_dist.pdf(x)
    I_importance = np.mean(weights)
    importance_estimates.append(I_importance)

uniform_estimates = np.array(uniform_estimates)
```

(continues on next page)

(continued from previous page)

```
importance_estimates = np.array(importance_estimates)

print(f"\nUniform sampling (N = {N}):")
print(f"  Mean:    {np.mean(uniform_estimates):.6f}")
print(f"  Std dev: {np.std(uniform_estimates):.6f}")

print(f"\nImportance sampling (N = {N}, g = truncated Gaussian):")
print(f"  Mean:    {np.mean(importance_estimates):.6f}")
print(f"  Std dev: {np.std(importance_estimates):.6f}")

variance_ratio = np.var(uniform_estimates) / np.var(importance_estimates)
print(f"\nVariance reduction factor: {variance_ratio:.1f}x")
```

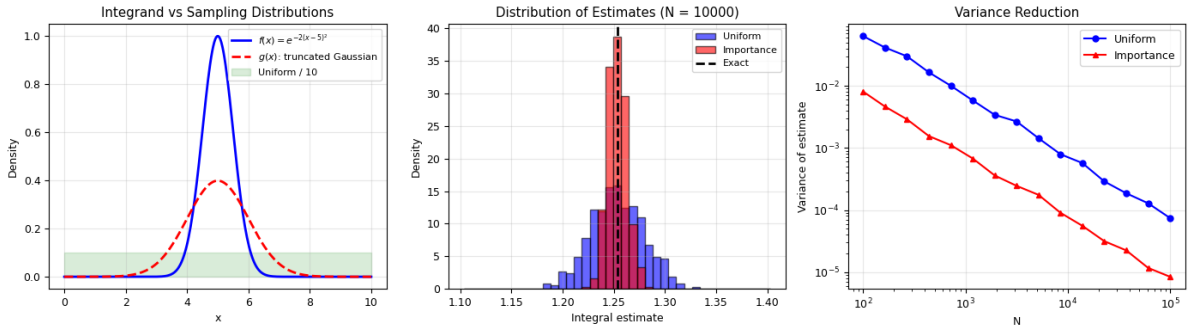
```
Exact integral: 1.2533141373

Uniform sampling (N = 10000):
  Mean:    1.253168
  Std dev: 0.025724

Importance sampling (N = 10000, g = truncated Gaussian):
  Mean:    1.253185
  Std dev: 0.009374

Variance reduction factor: 7.5x
```

```
fig, axes = plt.subplots(1, 3, figsize=(14, 4))

# Left: the integrand and sampling distributions
x_plot = np.linspace(0, 10, 300)
axes[0].plot(x_plot, f_peaked(x_plot), 'b-', lw=2, label='$f(x) = e^{-2(x-5)^2}$')
axes[0].plot(x_plot, g_dist.pdf(x_plot), 'r--', lw=2, label='$g(x)$: truncated␣
↪Gaussian')
axes[0].fill_between(x_plot, 0, 0.1, alpha=0.15, color='green', label='Uniform / 10')
axes[0].set_xlabel('x')
axes[0].set_ylabel('Density')
axes[0].set_title('Integrand vs Sampling Distributions')
axes[0].legend(fontsize=8)
axes[0].grid(alpha=0.3)

# Middle: histogram of estimates
bins = np.linspace(exact_peaked - 0.15, exact_peaked + 0.15, 40)
axes[1].hist(uniform_estimates, bins=bins, alpha=0.6, color='blue',
             edgecolor='black', label='Uniform', density=True)
axes[1].hist(importance_estimates, bins=bins, alpha=0.6, color='red',
             edgecolor='black', label='Importance', density=True)
axes[1].axvline(exact_peaked, color='k', linestyle='--', lw=2, label='Exact')
axes[1].set_xlabel('Integral estimate')
axes[1].set_ylabel('Density')
axes[1].set_title(f'Distribution of Estimates (N = {N})')
axes[1].legend(fontsize=8)
axes[1].grid(alpha=0.3)

# Right: variance reduction vs N
N_scan = np.logspace(2, 5, 15, dtype=int)
N_scan = np.unique(N_scan)
var_uniform = []
```

(continues on next page)

(continued from previous page)

```
var_importance = []

for N_i in N_scan:
    u_est = []
    i_est = []
    for _ in range(200):
        x_u = np.random.uniform(0, 10, N_i)
        u_est.append(10 * np.mean(f_peaked(x_u)))

        x_i = g_dist.rvs(N_i)
        i_est.append(np.mean(f_peaked(x_i) / g_dist.pdf(x_i)))
    var_uniform.append(np.var(u_est))
    var_importance.append(np.var(i_est))

axes[2].loglog(N_scan, var_uniform, 'bo-', ms=5, label='Uniform')
axes[2].loglog(N_scan, var_importance, 'r^-', ms=5, label='Importance')
axes[2].set_xlabel('N')
axes[2].set_ylabel('Variance of estimate')
axes[2].set_title('Variance Reduction')
axes[2].legend()
axes[2].grid(alpha=0.3)

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



13.4 III. Rejection Sampling

13.4.1 Generating Samples from Arbitrary Distributions

Problem: We want to draw samples from a target distribution $p(x)$, but we can't invert its Cumulative Distribution Function (CDF).

Solution — Rejection Sampling:

1. Choose a **proposal distribution** $q(x)$ that we *can* sample from
2. Find a constant M such that $M \cdot q(x) \geq p(x)$ for all x (an "envelope")
3. Draw $x \sim q(x)$ and $u \sim \text{Uniform}(0, 1)$
4. **Accept** x if $u < \frac{p(x)}{M \cdot q(x)}$; otherwise **reject** and try again

13.4.2 Proof: Accepted Samples Follow $p(x)$

The probability of accepting a sample at position x is:

$$P(\text{accept} \mid x) = \frac{p(x)}{M \cdot q(x)}$$

The joint density of proposing x and accepting it is:

$$P(x \text{ accepted}) = q(x) \cdot \frac{p(x)}{M \cdot q(x)} = \frac{p(x)}{M}$$

Since $\int \frac{p(x)}{M} dx = \frac{1}{M}$, the normalized distribution of accepted samples is:

$$P(x \mid \text{accepted}) = \frac{p(x)/M}{1/M} = p(x) \quad \checkmark$$

The overall **acceptance rate** is $1/M$, so we want M as small as possible (tight envelope).

13.4.3 Geometric Interpretation

Think of it as "throwing darts" at the region under $M \cdot q(x)$. Points that land under $p(x)$ are kept; points between $p(x)$ and $M \cdot q(x)$ are discarded. The efficiency equals the ratio of areas:

$$\text{Efficiency} = \frac{\text{Area under } p(x)}{\text{Area under } M \cdot q(x)} = \frac{1}{M}$$

13.4.4 Demo: Sampling from $p(x) \propto e^{-x^4}$

This "super-Gaussian" distribution has heavier tails than a Gaussian near the origin but thinner tails far out. There's no closed-form CDF, so we use rejection sampling with a Gaussian proposal.

```
# Target distribution: p(x) proportional to exp(-x^4)
def p_unnorm(x):
    return np.exp(-x**4)

# Compute normalization constant numerically
Z, _ = integrate.quad(p_unnorm, -10, 10)
print(f"Normalization constant Z = {Z:.6f}")

def p_target(x):
    return p_unnorm(x) / Z

# Proposal: Gaussian with sigma chosen to envelope p(x)
# We need M such that M * q(x) >= p(x) for all x
sigma_q = 1.0
q_dist = stats.norm(0, sigma_q)

# Find M = max(p(x) / q(x))
x_grid = np.linspace(-4, 4, 10000)
ratio = p_target(x_grid) / q_dist.pdf(x_grid)
M = np.max(ratio) * 1.01  # small safety margin
print(f"Envelope constant M = {M:.4f}")
print(f"Expected acceptance rate: {1/M:.1%}")
```

(continues on next page)

(continued from previous page)

```
# Rejection sampling
np.random.seed(42)
N_target = 50000
samples = []
n_proposed = 0

while len(samples) < N_target:
    x = q_dist.rvs()
    u = np.random.uniform()
    n_proposed += 1

    if u < p_target(x) / (M * q_dist.pdf(x)):
        samples.append(x)

samples = np.array(samples)
actual_rate = N_target / n_proposed

print(f"\nGenerated {N_target} samples from {n_proposed} proposals")
print(f"Actual acceptance rate: {actual_rate:.1%}")
print(f"Sample mean: {np.mean(samples):.4f} (expected: 0)")
print(f"Sample std:  {np.std(samples):.4f}")
```

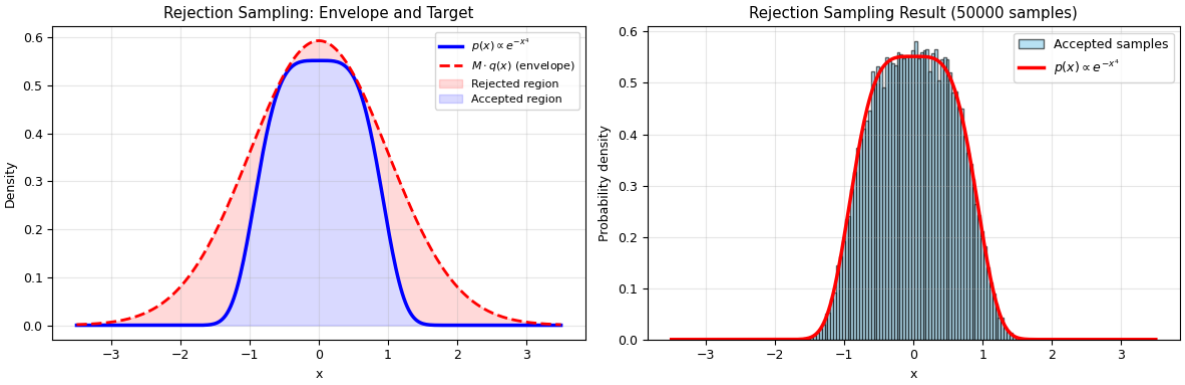
```
Normalization constant Z = 1.812805
Envelope constant M = 1.4866
Expected acceptance rate: 67.3%

Generated 50000 samples from 74150 proposals
Actual acceptance rate: 67.4%
Sample mean: -0.0006 (expected: 0)
Sample std:  0.5794
```

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Left: envelope visualization
x_plot = np.linspace(-3.5, 3.5, 300)
axes[0].plot(x_plot, p_target(x_plot), 'b-', lw=2.5, label='$p(x) \\propto e^{-x^4}$')
axes[0].plot(x_plot, M * q_dist.pdf(x_plot), 'r--', lw=2, label=f'$M \\cdot q(x)$␣
↪(envelope)')
axes[0].fill_between(x_plot, p_target(x_plot), M * q_dist.pdf(x_plot),
                     alpha=0.15, color='red', label='Rejected region')
axes[0].fill_between(x_plot, 0, p_target(x_plot),
                     alpha=0.15, color='blue', label='Accepted region')
axes[0].set_xlabel('x')
axes[0].set_ylabel('Density')
axes[0].set_title('Rejection Sampling: Envelope and Target')
axes[0].legend(fontsize=8)
axes[0].grid(alpha=0.3)

# Right: histogram of accepted samples vs target
axes[1].hist(samples, bins=80, density=True, alpha=0.6, color='skyblue',
             edgecolor='black', label='Accepted samples')
axes[1].plot(x_plot, p_target(x_plot), 'r-', lw=2.5, label='$p(x) \\propto e^{-x^4}$')
axes[1].set_xlabel('x')
axes[1].set_ylabel('Probability density')
axes[1].set_title(f'Rejection Sampling Result ({N_target} samples)')
axes[1].legend()
```

(continues on next page)

(continued from previous page)

```
axes[1].grid(alpha=0.3)

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



13.5 IV. Application: Variational Monte Carlo for the Hydrogen Atom

13.5.1 The Variational Principle

For any trial wavefunction $\psi_T(\mathbf{r}; \alpha)$ with variational parameter α, the **variational principle** guarantees:

$$E(\alpha) = \frac{\langle \psi_T | \hat{H} | \psi_T \rangle}{\langle \psi_T | \psi_T \rangle} \geq E_0$$

where E_0 is the true ground state energy. Minimizing $E(\alpha)$ gives the best approximation within our trial function family.

13.5.2 Hydrogen Atom Setup

The Hamiltonian (in atomic units: $\hbar = m_e = e = 1$):

$$\hat{H} = -\frac{1}{2}\nabla^2 - \frac{1}{r}$$

Trial wavefunction: $\psi_T(r; \alpha) = e^{-\alpha r}$

The **local energy** is defined as:

$$E_L(\mathbf{r}) = \frac{\hat{H}\psi_T(\mathbf{r})}{\psi_T(\mathbf{r})}$$

For our trial wavefunction, applying $\hat{H}$ in spherical coordinates:

$$E_L(r) = -\frac{\alpha^2}{2} + \frac{\alpha - 1}{r}$$

The variational energy is then:

$$E(\alpha) = \langle E_L \rangle_{|\psi_T|^2} = \int E_L(\mathbf{r}) \, |\psi_T(\mathbf{r})|^2 \, d^3\mathbf{r}$$

We evaluate this integral by **sampling positions from** $|\psi_T|^2$ and averaging E_L.

13.5.3 Analytical Result (for verification)

For $\psi_T = e^{-\alpha r}$, the exact variational energy is:

$$E(\alpha) = \frac{\alpha^2}{2} - \alpha$$

Minimizing: $dE/d\alpha = \alpha - 1 = 0 \Rightarrow \alpha^* = 1$, giving $E^* = -0.5$ Hartree $= -13.6$ eV.

This is exact because e^{-r} is the true ground state wavefunction of hydrogen!

```
def local_energy(r, alpha):
    """Local energy for hydrogen atom with trial wavefunction exp(-alpha*r).
    E_L(r) = -alpha^2/2 + (alpha - 1)/r
    """
    return -alpha**2 / 2 + (alpha - 1) / r

def sample_hydrogen_wf(alpha, N):
    """Sample positions from |psi_T|^2 = exp(-2*alpha*r).

    The radial distribution is P(r) = 4*pi*r^2 * exp(-2*alpha*r) / (pi/alpha^3)
    which is a Gamma distribution in r.
    Specifically, 2*alpha*r follows Gamma(3, 1), so r = Gamma(3,1) / (2*alpha).
    """
    # r follows the distribution P(r) ∝ r^2 exp(-2αr)
    # This is a Gamma(3, scale=1/(2α)) distribution
    r = np.random.gamma(3, scale=1/(2*alpha), size=N)
    return r

def vmc_energy(alpha, N):
    """Compute variational energy E(alpha) via Monte Carlo."""
    r = sample_hydrogen_wf(alpha, N)
    E_L = local_energy(r, alpha)
    E_mean = np.mean(E_L)
    E_err = np.std(E_L) / np.sqrt(N)
    return E_mean, E_err

def analytical_energy(alpha):
    """Exact variational energy for exp(-alpha*r) trial wavefunction."""
    return alpha**2 / 2 - alpha

# Test at alpha = 1 (exact ground state)
np.random.seed(42)
E, err = vmc_energy(1.0, 10000)
print(f"VMC at alpha = 1.0:  E = {E:.6f} ± {err:.6f} Hartree")
print(f"Exact:               E = {analytical_energy(1.0):.6f} Hartree")
print(f"                       = {analytical_energy(1.0) * 27.2114:.2f} eV")
```

```
VMC at alpha = 1.0:  E = -0.500000 ± 0.000000 Hartree
Exact:               E = -0.500000 Hartree
                       = -13.61 eV
```

```
# Scan alpha to find the minimum
np.random.seed(42)
alphas = np.linspace(0.3, 2.0, 30)
E_mc = []
E_mc_err = []
E_exact = []
```

(continues on next page)

(continued from previous page)

```
N_mc = 100000

for alpha in alphas:
    E, err = vmc_energy(alpha, N_mc)
    E_mc.append(E)
    E_mc_err.append(err)
    E_exact.append(analytical_energy(alpha))

E_mc = np.array(E_mc)
E_mc_err = np.array(E_mc_err)
E_exact = np.array(E_exact)

# Find VMC minimum
idx_min = np.argmin(E_mc)
alpha_min = alphas[idx_min]
E_min = E_mc[idx_min]

print(f"VMC minimum: alpha = {alpha_min:.3f}, E = {E_min:.6f} Hartree")
print(f"Exact minimum: alpha = 1.000, E = -0.500000 Hartree")
print(f"\nGround state energy: {E_min * 27.2114:.2f} eV (exact: -13.61 eV)")
```

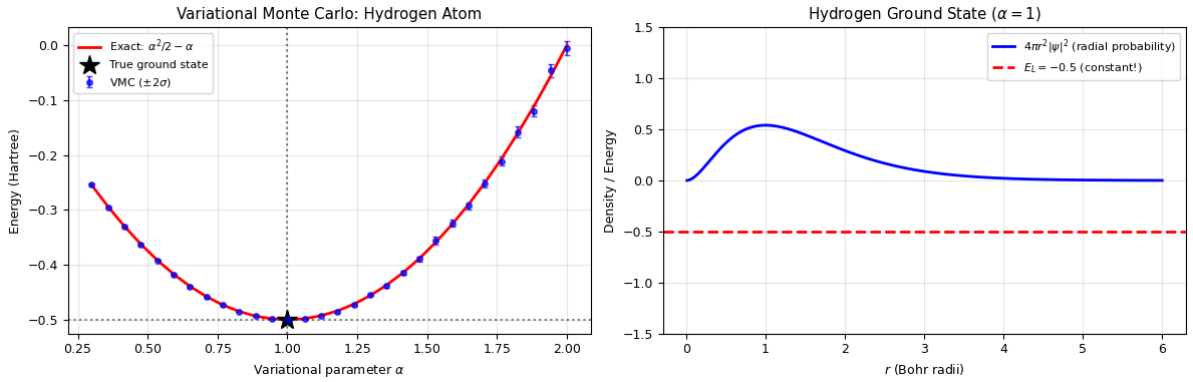
```
VMC minimum: alpha = 1.003, E = -0.499991 Hartree
Exact minimum: alpha = 1.000, E = -0.500000 Hartree

Ground state energy: -13.61 eV (exact: -13.61 eV)
```

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Left: E(alpha) curve
axes[0].errorbar(alphas, E_mc, yerr=2*E_mc_err, fmt='bo', ms=4, capsize=2,
                 label='VMC ($\\pm 2\\sigma$)', alpha=0.7)
axes[0].plot(alphas, E_exact, 'r-', lw=2, label='Exact: $\\alpha^2/2 - \\alpha$')
axes[0].axvline(1.0, color='k', linestyle=':', alpha=0.5)
axes[0].axhline(-0.5, color='k', linestyle=':', alpha=0.5)
axes[0].plot(1.0, -0.5, 'k*', ms=15, label='True ground state')
axes[0].set_xlabel('Variational parameter $\\alpha$')
axes[0].set_ylabel('Energy (Hartree)')
axes[0].set_title('Variational Monte Carlo: Hydrogen Atom')
axes[0].legend(fontsize=8)
axes[0].grid(alpha=0.3)

# Right: wavefunction and local energy at alpha=1
r_plot = np.linspace(0.01, 6, 300)
psi = np.exp(-r_plot)
psi_sq = psi**2
E_L_plot = local_energy(r_plot, 1.0)

ax2 = axes[1]
ax2.plot(r_plot, 4*np.pi*r_plot**2 * psi_sq / (np.pi), 'b-', lw=2,
         label='$4\\pi r^2 |\\psi|^2$ (radial probability)')
ax2.axhline(-0.5, color='r', linestyle='--', lw=2, label='$E_L = -0.5$ (constant!)')
ax2.set_xlabel('$r$ (Bohr radii)')
ax2.set_ylabel('Density / Energy')
ax2.set_title('Hydrogen Ground State ($\\alpha = 1$)')
ax2.legend(fontsize=8)
```

(continues on next page)

(continued from previous page)

```
ax2.grid(alpha=0.3)
ax2.set_ylim(-1.5, 1.5)

plt.tight_layout()
plt.show()

print("At alpha = 1 (exact wavefunction), E_L(r) = -0.5 everywhere — zero variance!")
print("This is the 'zero-variance property': when psi_T = psi_exact, the local energy␣
↪is constant.")
```




```
At alpha = 1 (exact wavefunction), E_L(r) = -0.5 everywhere — zero variance!
This is the 'zero-variance property': when psi_T = psi_exact, the local energy is␣
↪constant.
```

```
# Show the zero-variance property: variance of E_L vs alpha
np.random.seed(42)
alphas_fine = np.linspace(0.5, 1.5, 25)
variances = []

for alpha in alphas_fine:
    r = sample_hydrogen_wf(alpha, 100000)
    E_L = local_energy(r, alpha)
    variances.append(np.var(E_L))

fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(alphas_fine, variances, 'go-', ms=5, lw=2)
ax.axvline(1.0, color='r', linestyle='--', label='$\\alpha = 1$ (exact)')
ax.set_xlabel('Variational parameter $\\alpha$')
ax.set_ylabel('Var($E_L$)')
ax.set_title('Zero-Variance Property: Var($E_L$) → 0 as $\\psi_T$ → $\\psi_{exact}$')
ax.legend()
ax.grid(alpha=0.3)
plt.tight_layout()
plt.show()

print(f"Var(E_L) at alpha=1.0: {variances[len(variances)//2]:.2e}")
print(f"Var(E_L) at alpha=0.5: {variances[0]:.4f}")
print(f"Var(E_L) at alpha=1.5: {variances[-1]:.4f}")
print(f"\nThe variance vanishes at the exact wavefunction — this is a deep result!")
print(f"In practice, low variance of E_L indicates a good trial wavefunction.")
```

13.6 V. Estimating π: The Classic MC Example

13.6.1 Dart-Throwing Method

The simplest MC integration problem: throw random darts at a unit square $[0,1]^2$ and count the fraction that land inside the quarter-circle $x^2 + y^2 \leq 1$.

$$\frac{N_{\text{inside}}}{N_{\text{total}}} \approx \frac{\text{Area of quarter-circle}}{\text{Area of square}} = \frac{\pi/4}{1}$$

Therefore: $\pi \approx 4 \cdot \frac{N_{\text{inside}}}{N_{\text{total}}}$

This is just MC integration of $f(x,y) = \mathbf{1}[x^2 + y^2 \leq 1]$ over $[0,1]^2$.

```
np.random.seed(42)

N = 100000000
x = np.random.uniform(0, 1, N)
y = np.random.uniform(0, 1, N)

# finish the code to count points inside the quarter circle and estimate pi

# num_inside = 0
# for i in range(N):
#     xy = x[i]**2 + y[i]**2
#     if xy <=1:
#         num_inside += 1
# pi_estimate = num_inside / N * 4
# print (pi_estimate)

inside = x**2 + y**2 < 1
pi_estimate = np.sum(inside) / N * 4
print (pi_estimate)

# print(f"N = {N:,}")
print(f"Points inside quarter-circle: {np.sum(inside):,}")
print(f"π estimate: {pi_estimate:.6f}")
print(f"π exact:    {np.pi:.6f}")
print(f"Error:      {abs(pi_estimate - np.pi):.6f}")
```

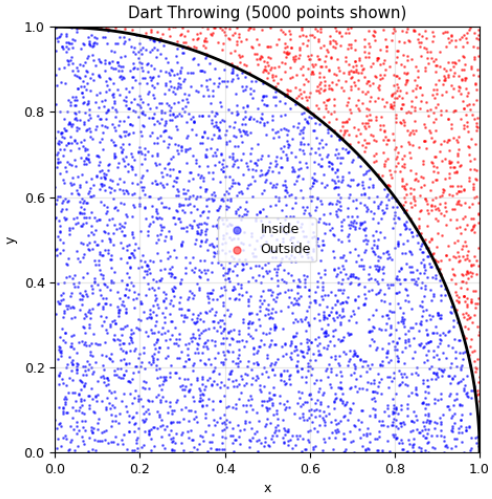
```
3.14172304
Points inside quarter-circle: 78,543,076
π estimate: 3.141723
π exact:    3.141593
Error:      0.000130
```

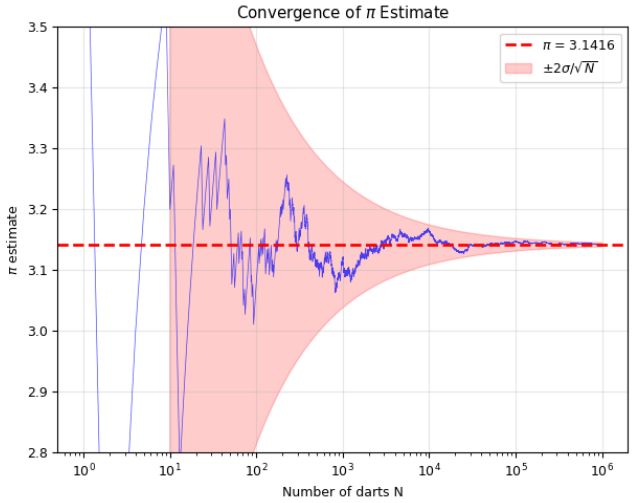
```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Left: dart visualization (first 5000 points)
n_show = 5000
axes[0].scatter(x[:n_show][inside[:n_show]], y[:n_show][inside[:n_show]],
                s=1, c='blue', alpha=0.5, label='Inside')
axes[0].scatter(x[:n_show][~inside[:n_show]], y[:n_show][~inside[:n_show]],
                s=1, c='red', alpha=0.5, label='Outside')

theta = np.linspace(0, np.pi/2, 100)
```

(continues on next page)

(continued from previous page)

```
axes[0].plot(np.cos(theta), np.sin(theta), 'k-', lw=2)
axes[0].set_xlim(0, 1)
axes[0].set_ylim(0, 1)
axes[0].set_aspect('equal')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].set_title(f'Dart Throwing ({n_show} points shown)')
axes[0].legend(markerscale=5)
axes[0].grid(alpha=0.3)

# Right: convergence of pi estimate
N_cum = np.arange(1, N+1)
pi_running = 4 * np.cumsum(inside) / N_cum

axes[1].semilogx(N_cum, pi_running, 'b-', lw=0.5, alpha=0.7)
axes[1].axhline(np.pi, color='r', linestyle='--', lw=2, label=f'$\\pi$ = {np.pi:.4f}')

# 1/sqrt(N) confidence band
p = np.pi / 4
sigma = 4 * np.sqrt(p * (1 - p))
N_plot = np.logspace(1, np.log10(N), 200)
axes[1].fill_between(N_plot,
                     np.pi - 2*sigma/np.sqrt(N_plot),
                     np.pi + 2*sigma/np.sqrt(N_plot),
                     alpha=0.2, color='red', label='$\\pm 2\\sigma/\\sqrt{N}$')

axes[1].set_xlabel('Number of darts N')
axes[1].set_ylabel('$\\pi$ estimate')
axes[1].set_title('Convergence of $\\pi$ Estimate')
axes[1].legend()
axes[1].grid(alpha=0.3)
axes[1].set_ylim(2.8, 3.5)

plt.tight_layout()
plt.show()

print(f"\nWith N = {N:,}: π ≈ {pi_estimate:.4f} (error = {abs(pi_estimate-np.pi):.4f})
 ↪")
print(f"Predicted error (σ/√N): {sigma/np.sqrt(N):.4f}")
print(f"\nTo get 6 digits of π, we'd need N ~ (σ/10^{-6})^2 ≈ 10^{12} samples — MC is␣
 ↪not efficient for π!")
print(f"But this same method works in 100 dimensions with the SAME error scaling.")
```






```
With N = 1,000,000: π ≈ 3.1419 (error = 0.0003)
Predicted error (σ/√N): 0.0016

To get 6 digits of π, we'd need N ~ (σ/10^-6)^2 ≈ 10^12 samples — MC is not␣
↪efficient for π!
But this same method works in 100 dimensions with the SAME error scaling.
```

CHAPTER

FOURTEEN

LECTURE 14: MONTE CARLO METHODS II — MCMC & OPTIMIZATION

Computational Physics — Spring 2026

14.1 From Sampling to Solving: MCMC and Simulated Annealing

In Lectures 12–13, we built a powerful Monte Carlo toolkit:

- Random number generation and the Metropolis algorithm (Lecture 12)
- MC integration, importance sampling, rejection sampling, and VMC (Lecture 13)

Today we take three major steps forward:

1. **The fundamental problem of statistical mechanics** — why MCMC is essential for computing thermal averages, with a hands-on ideal gas simulation
2. **Formal MCMC theory** — Markov chains, detailed balance, Metropolis-Hastings, and rigorous diagnostics
3. **Simulated annealing** — turning the Metropolis algorithm into a powerful *optimization* tool

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats, ndimage
from matplotlib.colors import ListedColormap

plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

print("Ready for MCMC and optimization!")
```

```
Ready for MCMC and optimization!
```

14.2 I. MCMC: From Statistical Mechanics to Markov Chains

14.2.1 Monte Carlo in Statistical Mechanics

Monte Carlo simulation is the name given to any computer simulation that uses random numbers to simulate a random physical process. Although it finds uses across all of physics, nowhere is it more central than in **statistical mechanics** — a field fundamentally built on randomness.

14.2.2 The Fundamental Problem

The central task of statistical mechanics is to compute the thermal average of a quantity X in a system at temperature T. A system in thermal equilibrium visits state i (with energy E_i) with the **Boltzmann probability**:

$$P(E_i) = \frac{e^{-\beta E_i}}{Z}, \qquad Z = \sum_i e^{-\beta E_i}$$

where $\beta = 1/k_B T$. The thermal average is then:

$$\langle X \rangle = \sum_i X_i \, P(E_i)$$

In a few special cases (e.g., quantum harmonic oscillator) we can evaluate this sum exactly. But for most real systems, the sum is over a **astronomically large** number of states — a single mole of a two-state system has $2^{10^{23}}$ states!

14.2.3 Why Not Just Sample Randomly?

We could try the MC integration approach from Lecture 13: pick N states at random and average. But there's a fatal problem — the Boltzmann factor $e^{-\beta E_i}$ is exponentially small for most states (those with $E_i \gg k_B T$). Almost every randomly chosen state contributes essentially nothing to the sum.

The solution is **importance sampling combined with MCMC**: instead of choosing states uniformly, we construct a Markov chain that visits states with probability $\propto e^{-\beta E_i}$. Then every sample contributes equally to the average!

14.2.4 The Metropolis Algorithm: A Quick Recap

The Metropolis algorithm (introduced in Lecture 12 for the Ising model) is the workhorse of MCMC. It constructs a Markov chain — a sequence of states where each state depends only on the previous one — that samples from the Boltzmann distribution.

The recipe:

1. Start in some state i
2. Propose a move to state j (e.g., flip one spin, change one quantum number)
3. Compute the acceptance probability: $\$P_{\text{accept}} = \begin{cases} 1 & \text{if } E_j < E_i \\ e^{-\beta(E_j - E_i)} & \text{if } E_j \geq E_i \end{cases}\$$
4. Accept the move with probability P_{accept}; otherwise stay at i
5. Measure the quantity of interest and repeat from step 2

Why it works: downhill moves (lower energy) are always accepted; uphill moves are accepted with a probability that decreases with the energy cost and increases with temperature. After a long burn-in, the chain samples states according to $P \propto e^{-\beta E}$.

Let's see this in action on a concrete physics problem before formalizing the theory.

14.2.5 Application: Monte Carlo Simulation of an Ideal Gas

A particle of mass m in a cubic box of side L has quantum states labeled by three integers $n_x, n_y, n_z = 1, 2, 3, ...$ with energies:

$$E(n_x, n_y, n_z) = \frac{\pi^2 \hbar^2}{2mL^2}(n_x^2 + n_y^2 + n_z^2)$$

An **ideal gas** of N non-interacting atoms has total energy $E = \sum_{i=1}^{N} E(n_x^i, n_y^i, n_z^i)$.

We use the Metropolis algorithm to compute the internal energy $\langle E \rangle$ at temperature $k_B T$:

- **State**: the set of all quantum numbers $\{n_x^i, n_y^i, n_z^i\}$ for N atoms
- **Move**: pick a random atom and a random direction, increment or decrement its quantum number by 1
- **Accept/reject**: via the Metropolis rule with ΔE from the change

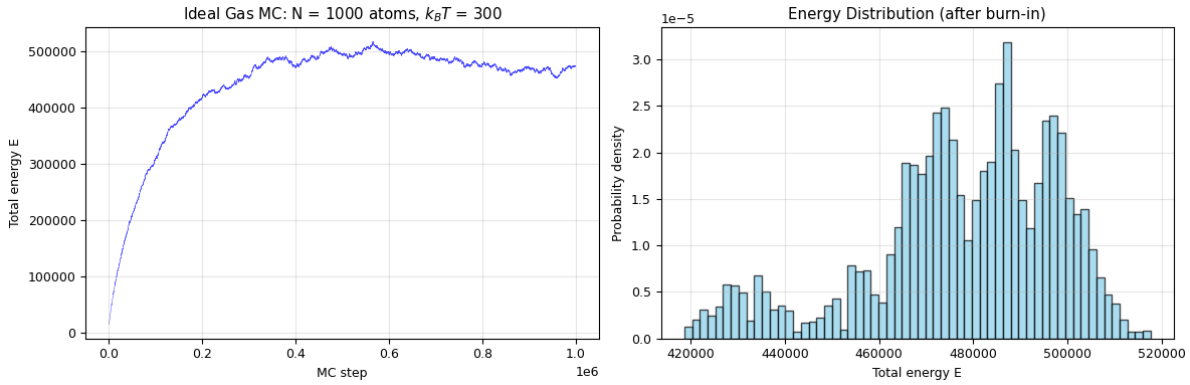
We work in natural units: $L = 1$, $\hbar = 1$, $m = 1$.

```
# Monte Carlo simulation of an ideal gas (units: L=1, hbar=1, m=1)
np.random.seed(42)

kbt = 300          # Temperature in natural units
N = 1000           # Number of atoms
steps = 1000000    # MC steps

# Quantum numbers: each atom has (n_x, n_y, n_z), all start at 1
n = np.ones([N, 3], int)

# Initial energy: E = (pi^2/2) * sum(n_x^2 + n_y^2 + n_z^2)
E = np.pi**2 / 2 * np.sum(n**2)
ehist = [E]

# Metropolis MC loop
for k in range(steps):
    # Choose a random atom and random direction (x, y, or z)
    i = np.random.randint(N)
    j = np.random.randint(3)

    # Propose: increment or decrement quantum number by 1
    if np.random.random() < 0.5:
        dn = 1
        dE = (2 * n[i, j] + 1) * np.pi**2 / 2
    else:
        dn = -1
        dE = (-2 * n[i, j] + 1) * np.pi**2 / 2

    # Accept/reject (also reject if n would go below 1)
    if dn == 1 or n[i, j] > 1:
        if np.random.random() < np.exp(-dE / kbt):
            n[i, j] += dn
            E += dE

    ehist.append(E)

# Plot energy trace
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(ehist, lw=0.3, alpha=0.7, color='blue')
```

(continues on next page)

(continued from previous page)

```
axes[0].set_xlabel('MC step')
axes[0].set_ylabel('Total energy E')
axes[0].set_title(f'Ideal Gas MC: N = {N} atoms, $k_BT$ = {kbt}')
axes[0].grid(alpha=0.3)

# Histogram of energy after burn-in
burn = len(ehist) // 5
axes[1].hist(ehist[burn:], bins=60, density=True, alpha=0.7, color='skyblue',␣
↪edgecolor='black')
axes[1].set_xlabel('Total energy E')
axes[1].set_ylabel('Probability density')
axes[1].set_title('Energy Distribution (after burn-in)')
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

E_avg = np.mean(ehist[burn:])
E_exact = 3 * N * kbt / 2  # Classical equipartition: E = (3/2) N k_B T
print(f"MC average energy:   <E> = {E_avg:.1f}")
print(f"Equipartition value: (3/2)NkT = {E_exact:.1f}")
print(f"Relative error: {abs(E_avg - E_exact)/E_exact:.2%}")
print(f"\nThe Metropolis algorithm recovers the equipartition theorem!")
```




```
MC average energy:   <E> = 477790.9
Equipartition value: (3/2)NkT = 450000.0
Relative error: 6.18%

The Metropolis algorithm recovers the equipartition theorem!
```

14.2.6 Formalizing MCMC: Markov Chains and Detailed Balance

The ideal gas simulation above *works* — but **why**? Why does the Metropolis algorithm produce samples from the Boltzmann distribution? The answer lies in the mathematical theory of Markov chains.

A **Markov chain** is a sequence of random states $x_0, x_1, x_2, \ldots$ where the probability of the next state depends *only* on the current state:

$$P(x_{n+1} \mid x_n, x_{n-1}, \ldots, x_0) = P(x_{n+1} \mid x_n) \equiv T(x_n \to x_{n+1})$$

The **transition probability** $T(x \to x')$ defines the rules of the chain.

14.2.7 Stationary Distribution

Under mild conditions (ergodicity, aperiodicity), a Markov chain converges to a unique **stationary distribution** $\pi(x)$ satisfying:

$$\pi(x') = \sum_x T(x \to x')\,\pi(x)$$

This is the distribution of states after running the chain for a long time, regardless of where we started.

14.2.8 Detailed Balance

A sufficient (not necessary) condition for $\pi(x)$ to be stationary is **detailed balance**:

$$\pi(x)\,T(x \to x') = \pi(x')\,T(x' \to x)$$

Proof that detailed balance implies stationarity:

Sum both sides over x:

$$\sum_x \pi(x)\,T(x \to x') = \sum_x \pi(x')\,T(x' \to x) = \pi(x') \sum_x T(x' \to x) = \pi(x')$$

where we used $\sum_x T(x' \to x) = 1$ (probability conservation). This is exactly the stationarity condition. □

The MCMC strategy: Design $T(x \to x')$ so that detailed balance is satisfied with $\pi(x)$ equal to the distribution we want to sample from. The Metropolis algorithm is one such design — and the ideal gas simulation above is a specific instance of it!

14.2.9 The Metropolis-Hastings Algorithm

In Lecture 12, we used the Metropolis algorithm with *symmetric* proposals. The **Metropolis-Hastings** algorithm generalizes this to *asymmetric* proposals.

Algorithm:

1. From current state x, propose $x' \sim q(x' \mid x)$ (proposal distribution)
2. Compute the acceptance ratio: $\$\alpha = \min\Big(1,\ \frac{\pi(x')\,q(x|x')}{\pi(x)\,q(x'|x)}\Big)\$$
3. Accept x' with probability α; otherwise stay at x

Special case — Metropolis: When q is symmetric ($q(x' \mid x) = q(x \mid x')$), the proposal terms cancel:

$$\alpha = \min\left(1,\ \frac{\pi(x')}{\pi(x)}\right)$$

This is exactly the Metropolis algorithm from Lecture 12!

14.2.10 Proof: Metropolis-Hastings Satisfies Detailed Balance

The transition probability is $T(x \to x') = q(x' \mid x) \cdot \alpha(x \to x')$. Check detailed balance:

$$\pi(x)\,q(x' \mid x)\,\alpha(x \to x')$$

Case 1: $\frac{\pi(x')q(x|x')}{\pi(x)q(x'|x)} \le 1$, so $\alpha(x \to x') = \frac{\pi(x')q(x|x')}{\pi(x)q(x'|x)}$ and $\alpha(x' \to x) = 1$.

$$\pi(x)\,q(x' \mid x) \cdot \frac{\pi(x')\,q(x \mid x')}{\pi(x)\,q(x' \mid x)} = \pi(x')\,q(x \mid x') \cdot 1 \ \checkmark$$

Case 2: The reverse. By symmetry of the argument, it also holds. □

14.2.11 Demo: Sampling a Bimodal Distribution

Let's sample from a mixture of two Gaussians:

$$\pi(x) = 0.4\,\mathcal{N}(-3, 1) + 0.6\,\mathcal{N}(3, 0.5)$$

This tests the chain's ability to jump between well-separated modes.

```
def target_bimodal(x):
    """Unnormalized target: mixture of two Gaussians."""
    return 0.4 * stats.norm.pdf(x, -3, 1) + 0.6 * stats.norm.pdf(x, 3, 0.5)

def metropolis_hastings(target, n_samples, x0=0.0, proposal_sigma=1.0):
    """Metropolis-Hastings with symmetric Gaussian proposal."""
    samples = np.zeros(n_samples)
    samples[0] = x0
    n_accept = 0

    for i in range(1, n_samples):
        x_current = samples[i-1]
        # Symmetric Gaussian proposal
        x_proposed = x_current + np.random.normal(0, proposal_sigma)

        # Acceptance ratio (proposal terms cancel for symmetric q)
        alpha = min(1.0, target(x_proposed) / target(x_current) )

        if np.random.uniform() < alpha:
            samples[i] = x_proposed
            n_accept += 1
        else:
            samples[i] = x_current

    return samples, n_accept / n_samples

np.random.seed(42)

# Compare three proposal widths
sigmas = [0.3, 2.0, 10.0]
results = {}

for sigma in sigmas:
    samples, acc = metropolis_hastings(target_bimodal, 50000, x0=0.0, proposal_
↪sigma=sigma)
    results[sigma] = (samples, acc)
    print(f"sigma = {sigma:5.1f}: acceptance rate = {acc:.1%}")
```

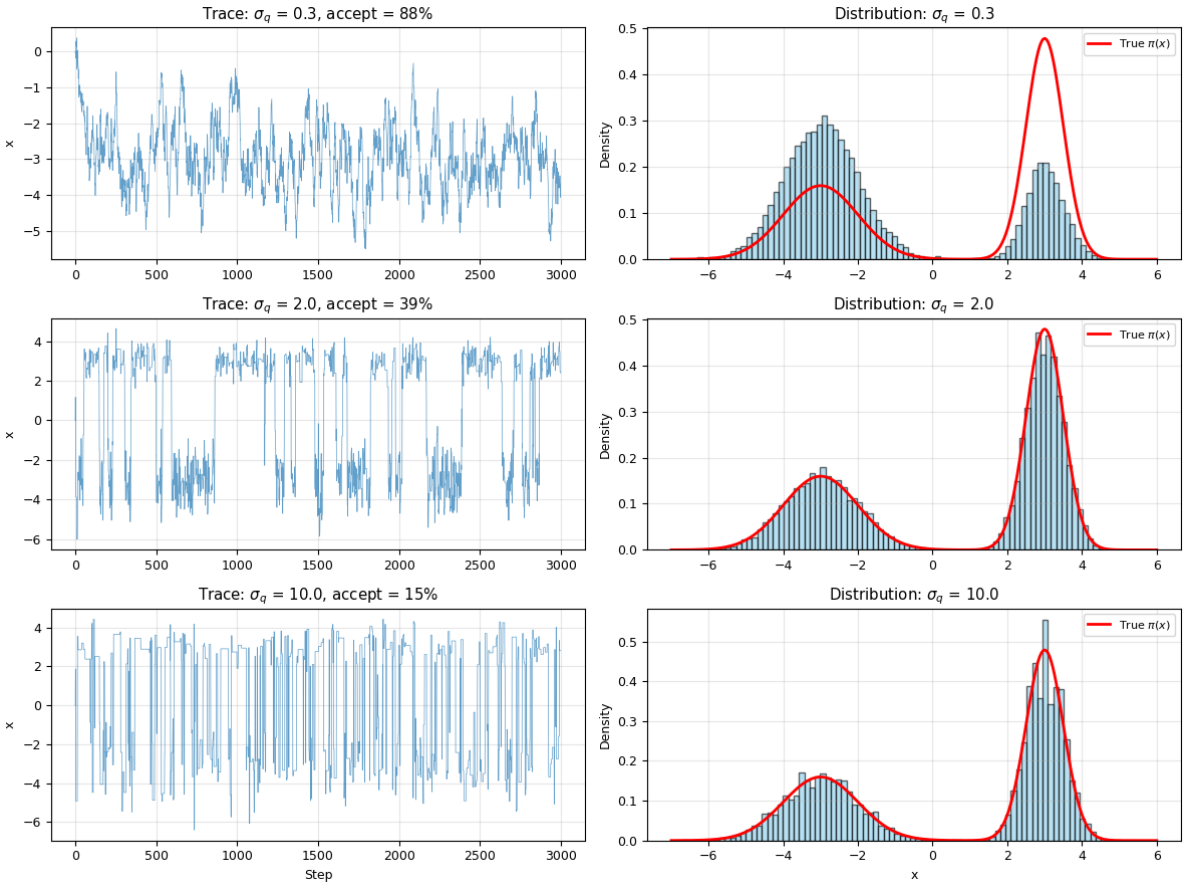
```
sigma =   0.3: acceptance rate = 88.2%
sigma =   2.0: acceptance rate = 39.2%
sigma =  10.0: acceptance rate = 14.8%
```

```
fig, axes = plt.subplots(3, 2, figsize=(12, 9))

x_plot = np.linspace(-7, 6, 300)
p_true = target_bimodal(x_plot)

for row, sigma in enumerate(sigmas):
    samples, acc = results[sigma]
```

(continues on next page)

(continued from previous page)

```
    # Left: trace plot (first 3000 steps)
    axes[row, 0].plot(samples[:3000], lw=0.5, alpha=0.7)
    axes[row, 0].set_ylabel('x')
    axes[row, 0].set_title(f'Trace: $\\sigma_q$ = {sigma}, accept = {acc:.0%}')
    axes[row, 0].grid(alpha=0.3)
    if row == 2:
        axes[row, 0].set_xlabel('Step')

    # Right: histogram vs true distribution
    axes[row, 1].hist(samples[5000:], bins=80, density=True, alpha=0.6,
                      color='skyblue', edgecolor='black')
    axes[row, 1].plot(x_plot, p_true, 'r-', lw=2, label='True $\\pi(x)$')
    axes[row, 1].set_ylabel('Density')
    axes[row, 1].set_title(f'Distribution: $\\sigma_q$ = {sigma}')
    axes[row, 1].legend(fontsize=8)
    axes[row, 1].grid(alpha=0.3)
    if row == 2:
        axes[row, 1].set_xlabel('x')

plt.tight_layout()
plt.show()

print("Observations:")
print("  σ = 0.3: High acceptance, but chain gets STUCK in one mode (poor mixing)")
print("  σ = 2.0: Moderate acceptance, good exploration of both modes (Goldilocks!)")
print("  σ = 10:  Low acceptance, chain jumps erratically (wasteful)")
print("\nRule of thumb: aim for 20-50% acceptance rate")
```




```
Observations:
  σ = 0.3: High acceptance, but chain gets STUCK in one mode (poor mixing)
  σ = 2.0: Moderate acceptance, good exploration of both modes (Goldilocks!)
  σ = 10:  Low acceptance, chain jumps erratically (wasteful)

Rule of thumb: aim for 20-50% acceptance rate
```

14.3 II. MCMC Diagnostics

14.3.1 How Do We Know Our Chain Has Converged?

MCMC produces *correlated* samples, unlike rejection sampling which gives i.i.d. samples. We need diagnostic tools to verify that:

1. The chain has **reached equilibrium** (burn-in is complete)
2. The samples are **sufficiently uncorrelated** for reliable estimates
3. The chain has **explored the full distribution** (mixing)

14.3.2 Tool 1: Burn-in

The initial portion of the chain is influenced by the starting point and should be discarded. This is called **burn-in**.

14.3.3 Tool 2: Autocorrelation Time

The **autocorrelation function** measures how correlated samples are at lag k:

$$C(k) = \frac{\langle x_n \, x_{n+k} \rangle - \langle x_n \rangle^2}{\langle x_n^2 \rangle - \langle x_n \rangle^2}$$

The **integrated autocorrelation time** is:

$$\tau_{\text{int}} = 1 + 2 \sum_{k=1}^{\infty} C(k)$$

This tells us how many MCMC steps correspond to one independent sample.

14.3.4 Tool 3: Effective Sample Size

From N MCMC samples, the number of *effectively independent* samples is:

$$N_{\text{eff}} = \frac{N}{2 \, \tau_{\text{int}}}$$

The true standard error of the mean is:

$$\text{SE} = \frac{\sigma}{\sqrt{N_{\text{eff}}}} = \sigma \sqrt{\frac{2 \, \tau_{\text{int}}}{N}}$$

```python
def autocorrelation(x, max_lag=500):
    """Compute normalized autocorrelation function."""
    x = x - np.mean(x)
    n = len(x)
    var = np.var(x)
    if var == 0:
        return np.zeros(max_lag)

    acf = np.zeros(max_lag)
    for k in range(max_lag):
        acf[k] = np.mean(x[:n-k] * x[k:]) / var
    return acf

def integrated_autocorrelation_time(acf):
    """Estimate integrated autocorrelation time with automatic truncation."""
    # Truncate when ACF first goes negative (simple heuristic)
    tau = 0.5  # Start at 0.5 from the k=0 term
    for k in range(1, len(acf)):
        if acf[k] < 0:
            break
        tau += acf[k]
    return tau

def effective_sample_size(samples, max_lag=500):
    """Compute effective sample size from MCMC chain."""
```

(continues on next page)

(continued from previous page)

```
    acf = autocorrelation(samples, max_lag)
    tau = integrated_autocorrelation_time(acf)
    n_eff = len(samples) / (2 * tau)
    return n_eff, tau, acf

# Analyze the three chains from above
print(f"{'σ_q':>5s}  {'N':>7s}  {'τ_int':>7s}  {'N_eff':>8s}  {'Efficiency':>11s}")
print("-" * 45)

for sigma in sigmas:
    samples = results[sigma][0][5000:]  # discard burn-in
    n_eff, tau, acf = effective_sample_size(samples)
    efficiency = n_eff / len(samples)
    print(f"{sigma:5.1f}  {len(samples):7d}  {tau:7.1f}  {n_eff:8.0f}  {efficiency:11.
↪1%}")
```

```
  σ_q        N    τ_int     N_eff   Efficiency
---------------------------------------------
  0.3    45000    441.3        51         0.1%
  2.0    45000     42.3       532         1.2%
 10.0    45000      9.0      2492         5.5%
```

```
fig, axes = plt.subplots(1, 3, figsize=(14, 4))

colors = ['blue', 'green', 'red']

for idx, sigma in enumerate(sigmas):
    samples = results[sigma][0][5000:]
    acf = autocorrelation(samples, max_lag=200)
    axes[0].plot(acf[:200], color=colors[idx], lw=1.5, label=f'$\sigma_q$ = {sigma}')

axes[0].axhline(0, color='k', linestyle='-', alpha=0.3)
axes[0].set_xlabel('Lag k')
axes[0].set_ylabel('C(k)')
axes[0].set_title('Autocorrelation Function')
axes[0].legend(fontsize=8)
axes[0].grid(alpha=0.3)
axes[0].set_xlim(0, 200)

# Running mean convergence
for idx, sigma in enumerate(sigmas):
    samples = results[sigma][0][5000:]
    running_mean = np.cumsum(samples) / np.arange(1, len(samples)+1)
    axes[1].plot(running_mean, color=colors[idx], lw=0.8, alpha=0.8, label=f'$\sigma_q
↪$ = {sigma}')

# True mean of bimodal distribution
x_fine = np.linspace(-8, 8, 1000)
p_fine = target_bimodal(x_fine)
true_mean = np.trapezoid(x_fine * p_fine, x_fine) / np.trapezoid(p_fine, x_fine)
axes[1].axhline(true_mean, color='k', linestyle='--', lw=2, label=f'True mean = {true_
↪mean:.2f}')
axes[1].set_xlabel('Sample index')
axes[1].set_ylabel('Running mean')
axes[1].set_title('Running Mean Convergence')
axes[1].legend(fontsize=7)
```

(continues on next page)

(continued from previous page)

```
axes[1].grid(alpha=0.3)

# Effective sample size vs proposal width
sigma_scan = np.linspace(0.1, 15, 30)
n_eff_scan = []

np.random.seed(0)
for s in sigma_scan:
    samp, _ = metropolis_hastings(target_bimodal, 10000, x0=0.0, proposal_sigma=s)
    n_eff, _, _ = effective_sample_size(samp[2000:])
    n_eff_scan.append(n_eff)

axes[2].plot(sigma_scan, n_eff_scan, 'ko-', ms=4, lw=1.5)
axes[2].set_xlabel('Proposal width $\sigma_q$')
axes[2].set_ylabel('$N_{\text{eff}}$')
axes[2].set_title('Effective Sample Size vs Proposal Width')
axes[2].grid(alpha=0.3)

idx_opt = np.argmax(n_eff_scan)
axes[2].axvline(sigma_scan[idx_opt], color='r', linestyle='--',
                label=f'Optimal $\sigma_q$ ≈ {sigma_scan[idx_opt]:.1f}')
axes[2].legend(fontsize=8)

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

14.4 III. Simulated Annealing

14.4.1 From Sampling to Optimization

The Metropolis algorithm samples from the Boltzmann distribution $P(x) \propto e^{-E(x)/T}$. At low temperature, this distribution concentrates near the **global minimum** of $E(x)$.

Idea: If we *slowly cool* the system during the Metropolis simulation, the chain will explore at high T (avoiding local minima) and then settle into the global minimum at low T.

This is **simulated annealing —** inspired by the physical process of annealing metals.

14.4.2 Algorithm

1. Start at high temperature T_0
2. Run Metropolis steps at current T
3. Reduce T according to a **cooling schedule**: $T_{k+1} = \beta \cdot T_k$ (with $\beta < 1$)
4. Repeat until T is very small

14.4.3 Cooling Schedule

Common choices:

- **Geometric**: $T_k = T_0 \cdot \beta^k$ (most common, $\beta \approx 0.95$–0.99)
- **Linear**: $T_k = T_0 - k \cdot \Delta T$
- **Logarithmic**: $T_k = T_0 / \ln(1+k)$ (provably optimal but very slow)

Theorem (Hajek, 1988): Simulated annealing converges to the global minimum with probability 1 if $T_k \geq \frac{c}{\ln(k)}$ for sufficiently large c. In practice, geometric cooling works well.

14.4.4 Demo: Finding the Global Minimum of a Multimodal Function

$$E(x) = x^2 + 5\sin^2(3x) + 3\cos^2(5x)$$

This has many local minima. A simple gradient descent would get stuck, but simulated annealing can find the global minimum.

```
def energy_multimodal(x):
    """Multimodal energy landscape."""
    return x**2 + 5 * np.sin(3*x)**2 + 3 * np.cos(5*x)**2

def simulated_annealing(energy_func, x0, T0, beta, n_steps, step_size=1.0):
    """Simulated annealing with geometric cooling."""
    x = x0
    E = energy_func(x)
    x_best, E_best = x, E

    T = T0
    trajectory = [(x, E, T)]

    for step in range(n_steps):
        # Propose new state
        x_new = x + np.random.normal(0, step_size)
        E_new = energy_func(x_new)
        dE = E_new - E

        # Metropolis acceptance at temperature T
        if dE < 0 or np.random.uniform() < np.exp(-dE / T):
            x, E = x_new, E_new

            # Track best solution
            if E < E_best:
                x_best, E_best = x, E

        # Cool down
        T *= beta
        trajectory.append((x, E, T))

    return x_best, E_best, np.array(trajectory)

np.random.seed(42)

# Run SA from a bad starting point
x_best, E_best, traj = simulated_annealing(
    energy_multimodal, x0=4.0, T0=20.0, beta=0.999, n_steps=10000, step_size=1.0
```

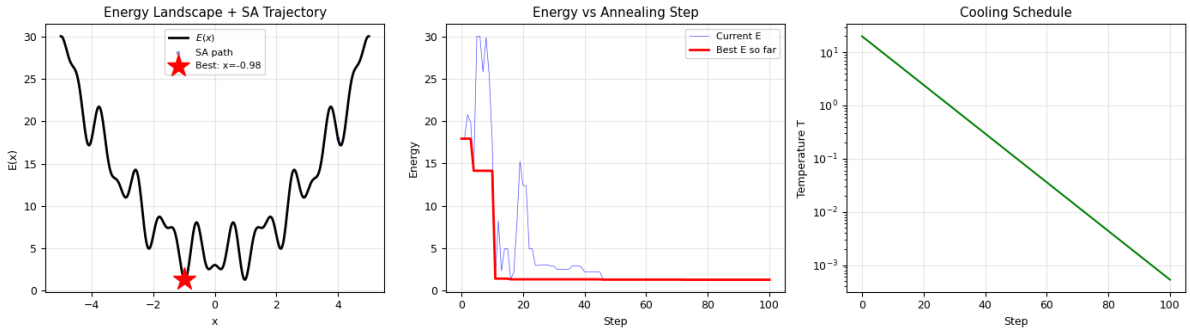
(continues on next page)

(continued from previous page)

```
)

print(f"Starting point: x = 4.0, E = {energy_multimodal(4.0):.4f}")
print(f"SA found: x = {x_best:.4f}, E = {E_best:.4f}")

# Brute force global minimum
x_grid = np.linspace(-5, 5, 10000)
E_grid = energy_multimodal(x_grid)
idx_min = np.argmin(E_grid)
print(f"Global minimum: x = {x_grid[idx_min]:.4f}, E = {E_grid[idx_min]:.4f}")
```

```
Starting point: x = 4.0, E = 17.9391
SA found: x = 0.9729, E = 1.2597
Global minimum: x = -0.9726, E = 1.2598
```

```
fig, axes = plt.subplots(1, 3, figsize=(14, 4))

# Left: energy landscape with SA trajectory
x_plot = np.linspace(-5, 5, 500)
axes[0].plot(x_plot, energy_multimodal(x_plot), 'k-', lw=2, label='$E(x)$')
axes[0].scatter(traj[::50, 0], traj[::50, 1], c=np.arange(0, len(traj), 50),
                cmap='coolwarm', s=5, alpha=0.5, label='SA path')
axes[0].plot(x_best, E_best, 'r*', ms=20, label=f'Best: x={x_best:.2f}')
axes[0].set_xlabel('x')
axes[0].set_ylabel('E(x)')
axes[0].set_title('Energy Landscape + SA Trajectory')
axes[0].legend(fontsize=8)
axes[0].grid(alpha=0.3)

# Middle: energy vs step
axes[1].plot(traj[:, 1], lw=0.5, alpha=0.7, color='blue', label='Current E')
best_so_far = np.minimum.accumulate(traj[:, 1])
axes[1].plot(best_so_far, 'r-', lw=2, label='Best E so far')
axes[1].set_xlabel('Step')
axes[1].set_ylabel('Energy')
axes[1].set_title('Energy vs Annealing Step')
axes[1].legend(fontsize=8)
axes[1].grid(alpha=0.3)

# Right: temperature schedule
axes[2].semilogy(traj[:, 2], 'g-', lw=1.5)
axes[2].set_xlabel('Step')
axes[2].set_ylabel('Temperature T')
axes[2].set_title('Cooling Schedule')
axes[2].grid(alpha=0.3)

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



14.4.5 Application: Traveling Salesman Problem (TSP)

The **Traveling Salesman Problem** is a classic combinatorial optimization problem:

> Given N cities with known positions, find the shortest tour that visits each city exactly once and returns to the start.

This is NP-hard — the number of possible tours is $(N-1)!/2$. For 20 cities, that's $\sim 10^{16}$ tours. Brute force is impossible, but simulated annealing can find near-optimal solutions.

Moves: Swap two random cities in the tour, or reverse a random segment.

```
import math
def tour_distance(cities, tour):
    """Total distance of a tour (closed loop)."""
    d = 0
    n = len(tour)
    for i in range(n):
        c1 = cities[tour[i]]
        c2 = cities[tour[(i+1) % n]]
        d += np.sqrt((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2)
    return d

def tsp_move(tour):
    """Generate a neighboring tour by reversing a random segment."""
    new_tour = tour.copy()
    n = len(tour)
    i, j = sorted(np.random.choice(n, 2, replace=False))
    new_tour[i:j+1] = new_tour[i:j+1][::-1]  # reverse segment
    return new_tour

def tsp_annealing(cities, T0=100.0, beta=0.9995, n_steps=100000):
    """Solve TSP via simulated annealing."""
    n = len(cities)
    tour = list(range(n))
    np.random.shuffle(tour)

    E = tour_distance(cities, tour)
    best_tour, best_E = tour.copy(), E

    T = T0
    energy_history = [E]
    temp_history = [T]

    for step in range(n_steps):
```

(continues on next page)

(continued from previous page)

```
        new_tour = tsp_move(tour)
        E_new = tour_distance(cities, new_tour)
        dE = E_new - E

        if dE < 0 or np.random.uniform() < np.exp(-dE / T):
            tour = new_tour
            E = E_new

            if E < best_E:
                best_tour = tour.copy()
                best_E = E

        T *= beta

        if step % 100 == 0:
            energy_history.append(E)
            temp_history.append(T)

    return best_tour, best_E, energy_history, temp_history

# Generate 25 random cities
np.random.seed(42)
n_cities = 25
cities = np.random.uniform(0, 100, size=(n_cities, 2))

# Initial random tour
initial_tour = list(range(n_cities))
initial_distance = tour_distance(cities, initial_tour)

# Solve with SA
best_tour, best_distance, energy_hist, temp_hist = tsp_annealing(
    cities, T0=100, beta=0.9995, n_steps=20000
)

print(f"Number of cities: {n_cities}")
print(f"Possible tours: {math.factorial(n_cities-1)//2:.2e}")
print(f"Initial tour distance: {initial_distance:.2f}")
print(f"SA best tour distance: {best_distance:.2f}")
print(f"Improvement: {(initial_distance - best_distance)/initial_distance:.1%}")
```

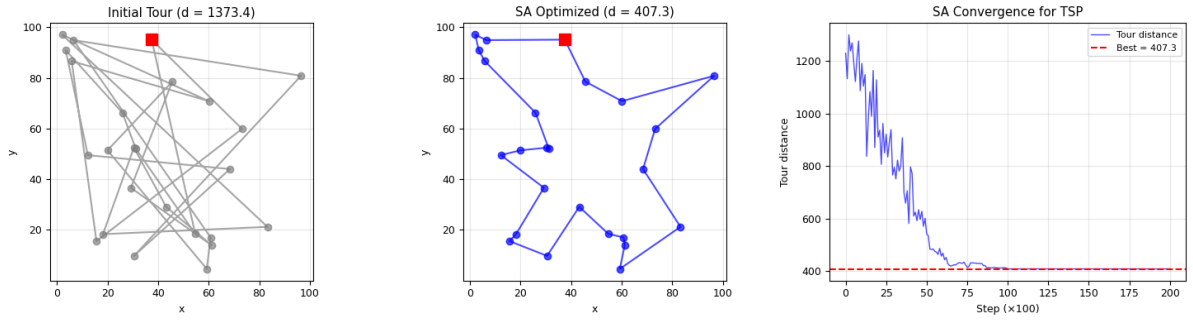
```
Number of cities: 25
Possible tours: 3.10e+23
Initial tour distance: 1373.41
SA best tour distance: 407.31
Improvement: 70.3%
```

```
def plot_tour(ax, cities, tour, title, color='blue'):
    """Plot a TSP tour."""
    tour_closed = tour + [tour[0]]  # return to start
    coords = cities[tour_closed]
    ax.plot(coords[:, 0], coords[:, 1], 'o-', color=color, ms=6, lw=1.5, alpha=0.7)
    ax.plot(cities[tour[0], 0], cities[tour[0], 1], 's', color='red', ms=10, label=
↪'Start')
    ax.set_title(title)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
```

(continues on next page)

(continued from previous page)

```
    ax.set_aspect('equal')
    ax.grid(alpha=0.3)

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Left: initial tour
plot_tour(axes[0], cities, initial_tour, f'Initial Tour (d = {initial_distance:.1f})',
 ↪ 'gray')

# Middle: optimized tour
plot_tour(axes[1], cities, best_tour, f'SA Optimized (d = {best_distance:.1f})', 'blue
 ↪')

# Right: energy convergence
axes[2].plot(energy_hist, lw=1, alpha=0.7, color='blue', label='Tour distance')
axes[2].axhline(best_distance, color='r', linestyle='--', lw=1.5, label=f'Best =
 ↪{best_distance:.1f}')
axes[2].set_xlabel('Step (×100)')
axes[2].set_ylabel('Tour distance')
axes[2].set_title('SA Convergence for TSP')
axes[2].legend(fontsize=8)
axes[2].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("\nNote: SA finds a near-optimal tour in seconds, while brute force would take␣
 ↪years!")
print("The key: high T allows uphill moves (escaping local minima), low T refines the␣
 ↪solution.")
```




```
Note: SA finds a near-optimal tour in seconds, while brute force would take years!
The key: high T allows uphill moves (escaping local minima), low T refines the␣
 ↪solution.
```

CHAPTER

FIFTEEN

LECTURE 15: OPTIMIZATION I — GRADIENT DESCENT AND LINE SEARCH

Computational Physics — Spring 2026

15.1 Finding Minima: The Core Problem of Computational Science

Optimization is one of the most fundamental tasks in computational physics. Nearly every simulation, fit, or model ultimately requires finding the minimum of some function:

- **Molecular physics**: Finding stable configurations (minimize potential energy)
- **Data fitting**: Minimizing the residual between model and data (Lecture 7, HW4)
- **Quantum mechanics**: Variational principle — the ground state minimizes $\langle \hat{H} \rangle$ (Lecture 13 VMC)
- **Machine learning**: Training = minimizing a loss function

In Lecture 14, we used **simulated annealing** — a global, stochastic optimizer. Today we study **local optimization**: deterministic methods that follow the landscape downhill to the nearest minimum.

Today's roadmap:

- The optimization landscape: local vs global minima
- 1D gradient descent: fixed step and adaptive step
- Golden section search: derivative-free 1D optimization
- Extending gradient descent to 2D
- Introduction to `scipy.optimize`

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

plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

print("Ready for optimization!")
```

```
Ready for optimization!
```

15.2 I. The Optimization Landscape

15.2.1 Local vs Global Minima

Given a function $f(x)$, we want to find x^* such that $f(x^*) \leq f(x)$ for all x nearby.

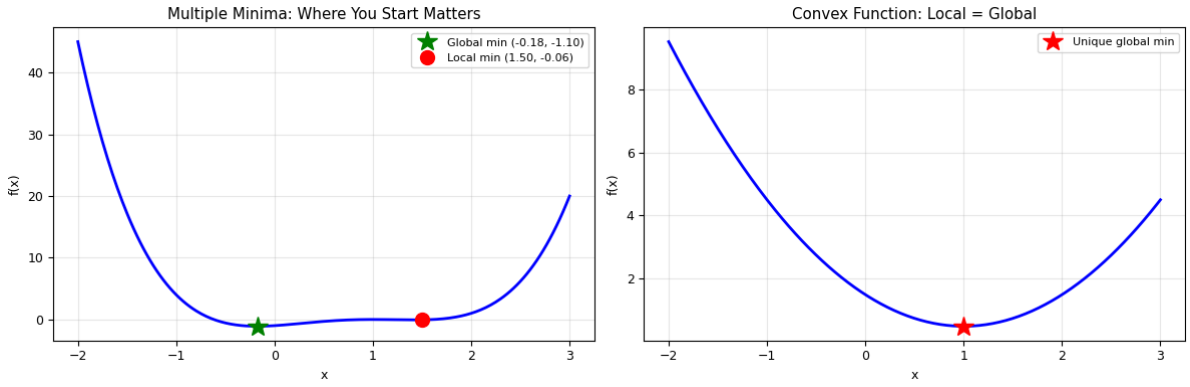
Key concepts:

- **Local minimum**: $f(x^*) \leq f(x)$ for all x in a neighborhood of x^*
- **Global minimum**: $f(x^*) \leq f(x)$ for *all* x
- **Saddle point**: A point where the gradient is zero but it's neither a minimum nor maximum
- **Convex function**: A function where every local minimum is also the global minimum

Necessary condition for a minimum: $f'(x^*) = 0$ and $f''(x^*) > 0$.

The challenge: Local optimization methods can only find *local* minima. The result depends on where we start. This is why we ran simulated annealing in Lecture 14 — it can escape local minima via thermal fluctuations.

```
# Visualize: a landscape with multiple minima
x = np.linspace(-2, 3, 300)
f_landscape = lambda x: x**4 - 3*x**3 + 2*x**2 + x - 1

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Left: function with local and global minima
axes[0].plot(x, f_landscape(x), 'b-', lw=2)
axes[0].set_xlabel('x')
axes[0].set_ylabel('f(x)')
axes[0].set_title('Multiple Minima: Where You Start Matters')
axes[0].grid(alpha=0.3)

# Mark local and global minima (approximate)
from scipy.optimize import minimize_scalar
res1 = minimize_scalar(f_landscape, bounds=(-1, 0.5), method='bounded')
res2 = minimize_scalar(f_landscape, bounds=(1.5, 3), method='bounded')
axes[0].plot(res1.x, res1.fun, 'g*', ms=15, label=f'Global min ({res1.x:.2f}, {res1.
 ↪fun:.2f})')
axes[0].plot(res2.x, res2.fun, 'ro', ms=10, label=f'Local min ({res2.x:.2f}, {res2.
 ↪fun:.2f})')
axes[0].legend(fontsize=8)

# Right: convex function — every local min is global
f_convex = lambda x: (x - 1)**2 + 0.5
axes[1].plot(x, f_convex(x), 'b-', lw=2)
axes[1].plot(1, 0.5, 'r*', ms=15, label='Unique global min')
axes[1].set_xlabel('x')
axes[1].set_ylabel('f(x)')
axes[1].set_title('Convex Function: Local = Global')
axes[1].legend(fontsize=8)
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("For convex functions, any local minimum is the global minimum.")
print("For non-convex functions, the result depends on the starting point!")
```




```
For convex functions, any local minimum is the global minimum.
For non-convex functions, the result depends on the starting point!
```

15.3 II. 1D Optimization: Gradient Descent

15.3.1 The Gradient Descent Algorithm

The simplest optimization method: follow the slope downhill.

Update rule:

$$x_{n+1} = x_n - \eta\, f'(x_n)$$

where η is the **step size** (or **learning rate**).

- If $f'(x_n) > 0$: the function is increasing, so we move left (x decreases)
- If $f'(x_n) < 0$: the function is decreasing, so we move right (x increases)
- If $f'(x_n) = 0$: we're at a critical point (minimum, maximum, or saddle)

We approximate the derivative numerically using the **central difference** formula:

$$f'(x) \approx \frac{f(x+\delta) - f(x-\delta)}{2\delta}$$

15.3.2 Why Step Size Matters

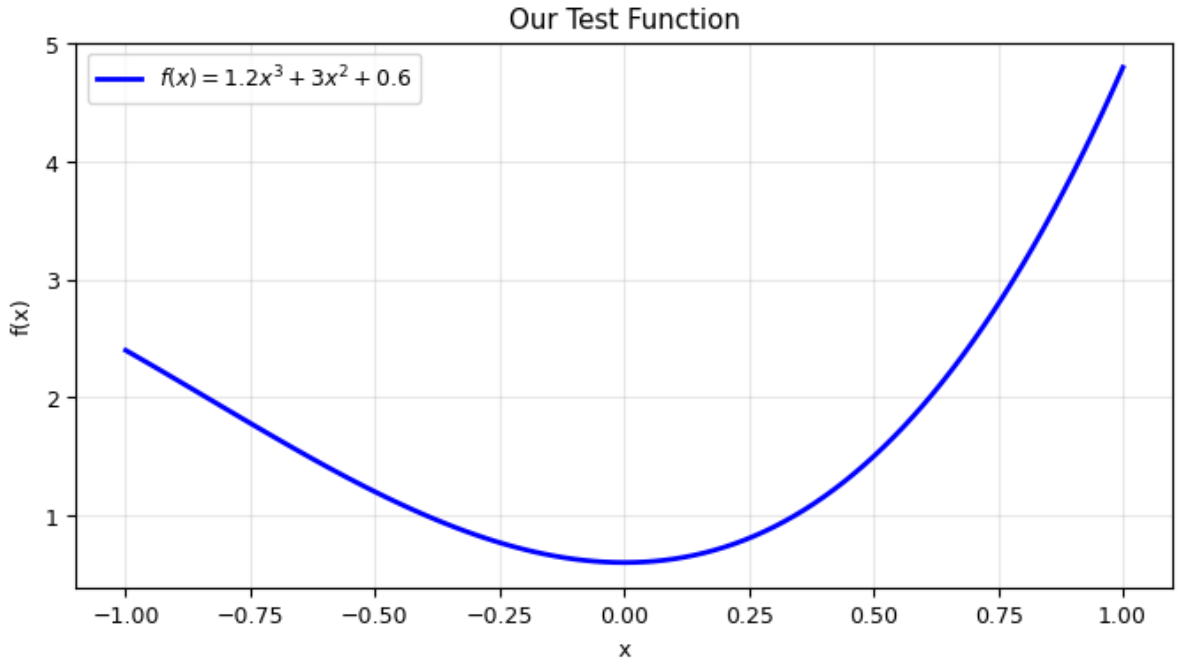
- η **too large**: overshoot the minimum, oscillate or diverge
- η **too small**: converge very slowly, waste computation
- **Just right**: efficient convergence

15.3.3 Test Function: A Cubic Polynomial

$$f(x) = Ax^3 + Bx^2 + C$$

with $A = 1.2$, $B = 3.0$, $C = 0.6$. This function has a local minimum near $x = 0$ and goes to $-\infty$ as $x \to -\infty$.

```
# Define coefficients for our polynomial
A, B, C = 1.2, 3.0, 0.6

x_min, x_max = -1, 1
f = lambda x: A*x**3 + B*x**2 + C

# Visualize
x = np.linspace(x_min, x_max, 101)
plt.figure(figsize=(8, 4))
plt.plot(x, f(x), 'b-', lw=2, label='$f(x) = 1.2x^3 + 3x^2 + 0.6$')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Our Test Function')
plt.legend()
plt.grid(alpha=0.3)
plt.show()

# Analytical minimum: f'(x) = 3Ax^2 + 2Bx = 0 => x = 0 or x = -2B/(3A)
x_crit = -2*B / (3*A)
print(f"Critical points: x = 0 and x = {x_crit:.4f}")
print(f"f(0) = {f(0):.4f}, f({x_crit:.4f}) = {f(x_crit):.4f}")
```



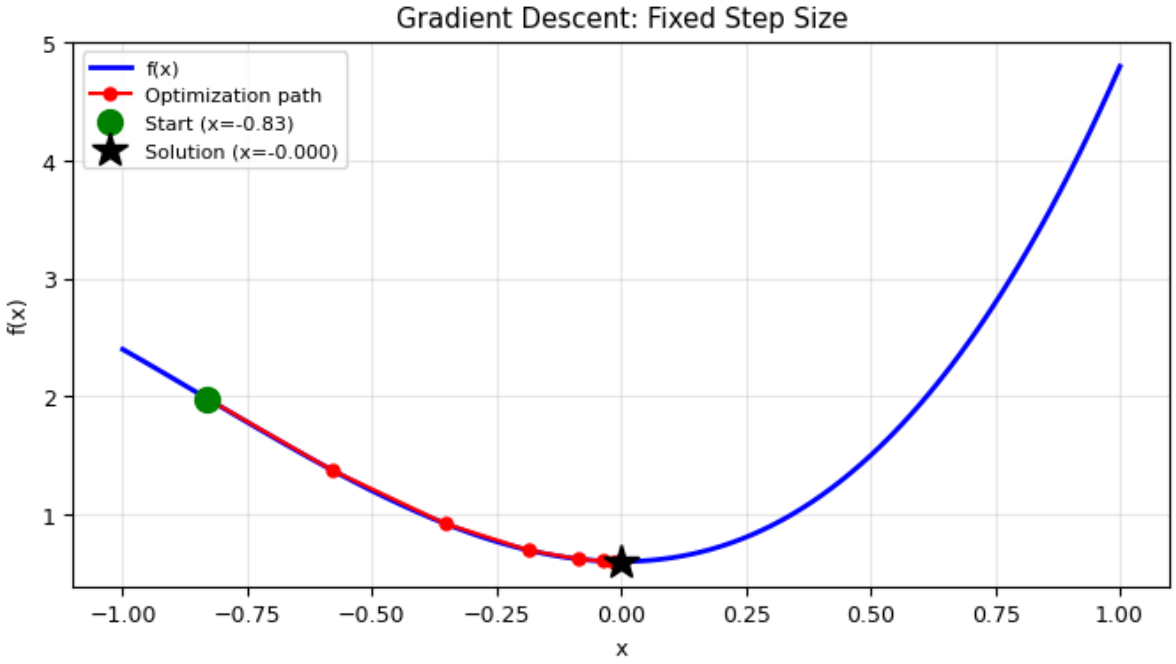

```
Critical points: x = 0 and x = -1.6667
f(0) = 0.6000, f(-1.6667) = 3.3778
```

15.3.4 Implementation: Fixed Step Size

```
def derivative(f, x, dx=0.01):
    """Numerical derivative using central difference."""
    return (f(x + dx) - f(x - dx)) / (2 * dx)

def minimize(f, x0, dx=0.01, N=1000):
    """Gradient descent with fixed step size.

    Args:
        f: function to minimize
        x0: starting point
        dx: step size (learning rate)
        N: max iterations

    Returns:
        converged, x_best, f_min, x_hist, step
    """
    x_now = x0
    converged = False
    x_hist = [x0]

    for i in range(N):
        x_next = x_now - dx * derivative(f, x_now)
        if f(x_next) < f(x_now):
            x_now = x_next
            x_hist.append(x_now)
        else:
            converged = True
            break

    step = i
    return converged, x_now, f(x_now), np.array(x_hist), step
```

```
# Test with a specific starting point
x0 = -0.83
converged, x_best, f_min, x_hist, step = minimize(f, x0, dx=0.1)

print(f'Starting point: x0 = {x0}')
print(f'Converged: {converged}')
print(f'Best solution: x = {x_best:.4f}, f = {f_min:.4f}')
print(f'Steps taken: {step}')

# Visualize the optimization path
x = np.linspace(x_min, x_max, 101)
plt.figure(figsize=(8, 4))
plt.plot(x, f(x), 'b-', lw=2, label='f(x)')
plt.plot(x_hist, f(x_hist), 'ro-', ms=5, label='Optimization path')
plt.plot(x0, f(x0), 'go', ms=10, label=f'Start (x={x0})')
plt.plot(x_best, f_min, 'k*', ms=15, label=f'Solution (x={x_best:.3f})')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Gradient Descent: Fixed Step Size')
plt.legend(fontsize=8)
plt.grid(alpha=0.3)
plt.show()
```

```
Starting point: x0 = -0.83
Converged: True
Best solution: x = -0.0000, f = 0.6000
Steps taken: 33
```



15.3.5 Sensitivity to Starting Point

Let's run many trials with random starting points to see how the result depends on where we begin.

```
np.random.seed(42)

print(f'{"x0":>10s} | {"Converged":>9s} | {"x_best":>8s} | {"f(x_best)":>10s} | {
 ↪"Steps":>5s}')
print('-' * 55)

for i in range(20):
    x1 = x_min + np.random.random() * (x_max - x_min)
    converged, x_best, f_min, x_hist, step = minimize(f, x1, dx=0.02)
    print(f'{x1:10.4f} | {str(converged):>9s} | {x_best:8.4f} | {f_min:10.4f} |
 ↪{step:5d}')
```

```
        x0 | Converged |   x_best |  f(x_best) | Steps
-------------------------------------------------------
   -0.2509 |      True |  -0.0000 |     0.6000 |   193
    0.9014 |      True |  -0.0000 |     0.6000 |    80
    0.4640 |      True |   0.0000 |     0.6000 |    76
    0.1973 |      True |  -0.0000 |     0.6000 |    71
   -0.6880 |      True |  -0.0000 |     0.6000 |   204
   -0.6880 |      True |  -0.0000 |     0.6000 |   204
   -0.8838 |      True |  -0.0000 |     0.6000 |   208
    0.7324 |      True |  -0.0000 |     0.6000 |    79
```

(continues on next page)

(continued from previous page)

```
    0.2022 |       True |    0.0000 |      0.6000 |     71
    0.4161 |       True |   -0.0000 |      0.6000 |     76
   -0.9588 |       True |   -0.0000 |      0.6000 |    211
    0.9398 |       True |    0.0000 |      0.6000 |     80
    0.6649 |       True |    0.0000 |      0.6000 |     78
   -0.5753 |       True |   -0.0000 |      0.6000 |    203
   -0.6364 |       True |   -0.0000 |      0.6000 |    203
   -0.6332 |       True |   -0.0000 |      0.6000 |    203
   -0.3915 |       True |   -0.0000 |      0.6000 |    199
    0.0495 |       True |   -0.0000 |      0.6000 |     61
   -0.1361 |       True |   -0.0000 |      0.6000 |    189
   -0.4175 |       True |   -0.0000 |      0.6000 |    198
```

15.3.6 Improvement: Variable Step Size (Barzilai-Borwein Method)

Instead of a fixed η, we can adapt the step size at each iteration. The **Barzilai-Borwein** method uses information from the previous step:

$$\eta_n = \frac{x_n - x_{n-1}}{f'(x_n) - f'(x_{n-1})}$$

This is derived from a secant approximation to the Hessian. The idea: if the gradient hasn't changed much, take a larger step; if it changed a lot, take a smaller step.

```python
def minimize2(f, x0, N=1000):
    """Gradient descent with Barzilai-Borwein variable step size."""
    x_now = x0
    x_prev = None
    converged = False
    x_hist = [x0]

    dx = 0.1  # initial step size
    for i in range(N):
        if x_prev is not None:
            dfx = derivative(f, x_now) - derivative(f, x_prev)
            if abs(dfx) > 1e-8:
                dx = (x_now - x_prev) / dfx
            else:
                dx = 0.1

        x_next = x_now - derivative(f, x_now) * dx
        if f(x_next) < f(x_now):
            x_prev = x_now
            x_now = x_next
            x_hist.append(x_now)
        else:
            converged = True
            break

    return converged, x_now, f(x_now), np.array(x_hist), i

# Compare with fixed step
np.random.seed(42)
print(f'{"x0":>10s} | {"Method":>10s} | {"Converged":>9s} | {"x_best":>8s} | {"f(x_
↪best)":>10s} | {"Steps":>5s}')
```

(continues on next page)

(continued from previous page)

```
print('-' * 70)

for i in range(10):
    x1 = x_min + np.random.random() * (x_max - x_min)
    c1, xb1, fm1, _, s1 = minimize(f, x1, dx=0.05)
    c2, xb2, fm2, _, s2 = minimize2(f, x1)
    print(f'{x1:10.4f} | {"Fixed":>10s} | {str(c1):>9s} | {xb1:8.4f} | {fm1:10.4f} |
↪{s1:5d}')
    print(f'{"":>10s} | {"Variable":>10s} | {str(c2):>9s} | {xb2:8.4f} | {fm2:10.4f}␣
↪| {s2:5d}')
    print()
```

```
        x0 |     Method | Converged |   x_best |  f(x_best) | Steps
----------------------------------------------------------------------
   -0.2509 |      Fixed |      True |  -0.0000 |     0.6000 |    72
           |   Variable |      True |  -0.0000 |     0.6000 |     5

    0.9014 |      Fixed |      True |   0.0000 |     0.6000 |    28
           |   Variable |      True |  -0.0000 |     0.6000 |     4

    0.4640 |      Fixed |      True |   0.0000 |     0.6000 |    27
           |   Variable |      True |  -0.0000 |     0.6000 |     4

    0.1973 |      Fixed |      True |   0.0000 |     0.6000 |    25
           |   Variable |      True |  -0.0000 |     0.6000 |     4

   -0.6880 |      Fixed |      True |  -0.0000 |     0.6000 |    78
           |   Variable |      True |  -0.4456 |     1.0895 |     1

   -0.6880 |      Fixed |      True |  -0.0000 |     0.6000 |    78
           |   Variable |      True |  -0.4456 |     1.0896 |     1

   -0.8838 |      Fixed |      True |  -0.0000 |     0.6000 |    78
           |   Variable |      True |  -0.6348 |     1.5019 |     1

    0.7324 |      Fixed |      True |  -0.0000 |     0.6000 |    28
           |   Variable |      True |   0.0000 |     0.6000 |     4

    0.2022 |      Fixed |      True |   0.0000 |     0.6000 |    25
           |   Variable |      True |  -0.0000 |     0.6000 |     4

    0.4161 |      Fixed |      True |  -0.0000 |     0.6000 |    27
           |   Variable |      True |  -0.0000 |     0.6000 |     4
```

15.4 III. Golden Section Search: No Derivatives Needed

15.4.1 A Different Approach: Bracket-Based Search

Gradient descent requires computing $f'(x)$. What if the derivative is expensive or doesn't exist? We can use a **bracketing method** instead.

Idea: Maintain a bracket $[a, b]$ that contains the minimum, and shrink it at each step.

15.4.2 The Golden Section Search

Analogous to the bisection method for root-finding, but for minimization.

Algorithm:

1. Start with bracket $[a, b]$ containing a minimum
2. Evaluate f at two interior points: $x_1 = a + (1-\varphi)(b-a), \quad x_2 = a + \varphi(b-a) where \varphi = \frac{\sqrt{5} - 1}{2} \approx 0.618$ is the **golden ratio conjugate**
3. If $f(x_1) < f(x_2)$: the minimum is in $[a, x_2]$ — set $b = x_2$
4. If $f(x_1) \geq f(x_2)$: the minimum is in $[x_1, b]$ — set $a = x_1$
5. Repeat until $|b - a| <$ tolerance

15.4.3 Why the Golden Ratio?

The key insight: with the golden ratio, one of the two evaluation points from the previous step can be **reused** in the next step. This means we only need **one new function evaluation per step** (instead of two), cutting the work in half.

Proof: After narrowing $[a, b]$ to $[a, x_2]$, the old point x_1 satisfies:

$$\frac{x_1 - a}{x_2 - a} = \frac{(1-\varphi)(b-a)}{\varphi(b-a)} = \frac{1-\varphi}{\varphi} = \varphi$$

So x_1 is at position φ within $[a, x_2]$ — exactly where the new upper evaluation point should be! This works because φ satisfies $\varphi^2 + \varphi = 1$, the defining property of the golden ratio. □

Convergence: The bracket shrinks by factor $\varphi \approx 0.618$ at each step, giving linear convergence.

```
def golden_section_search(f, a, b, tol=1e-6, N=100):
    """Golden section search for minimum of f on [a, b].

    Requires: f has a single minimum in [a, b] (unimodal).
    """
    phi = (np.sqrt(5) - 1) / 2  # golden ratio conjugate ≈ 0.618

    # Initial interior points
    x1 = a + (1 - phi) * (b - a)
    x2 = a + phi * (b - a)
    f1 = f(x1)
    f2 = f(x2)

    history = [(a, b, x1, x2)]

    for i in range(N):
        if abs(b - a) < tol:
            break

        if f1 < f2:
            # Minimum is in [a, x2]
            b = x2
            x2 = x1             # reuse old x1 as new x2
            f2 = f1
            x1 = a + (1 - phi) * (b - a)  # only one new evaluation
            f1 = f(x1)
        else:
```

(continues on next page)

(continued from previous page)

```
            # Minimum is in [x1, b]
            a = x1
            x1 = x2             # reuse old x2 as new x1
            f1 = f2
            x2 = a + phi * (b - a)  # only one new evaluation
            f2 = f(x2)

        history.append((a, b, x1, x2))

    x_min = (a + b) / 2
    return x_min, f(x_min), i, history

# Test on our polynomial
x_opt, f_opt, steps, hist = golden_section_search(f, -0.5, 0.5)

print(f"Golden section search on f(x) = {A}x³ + {B}x² + {C}")
print(f"Bracket: [{-0.5}, {0.5}]")
print(f"Found minimum: x = {x_opt:.6f}, f = {f_opt:.6f}")
print(f"Steps: {steps}")
print(f"\nAnalytical minimum: x = 0, f = {C}")
```

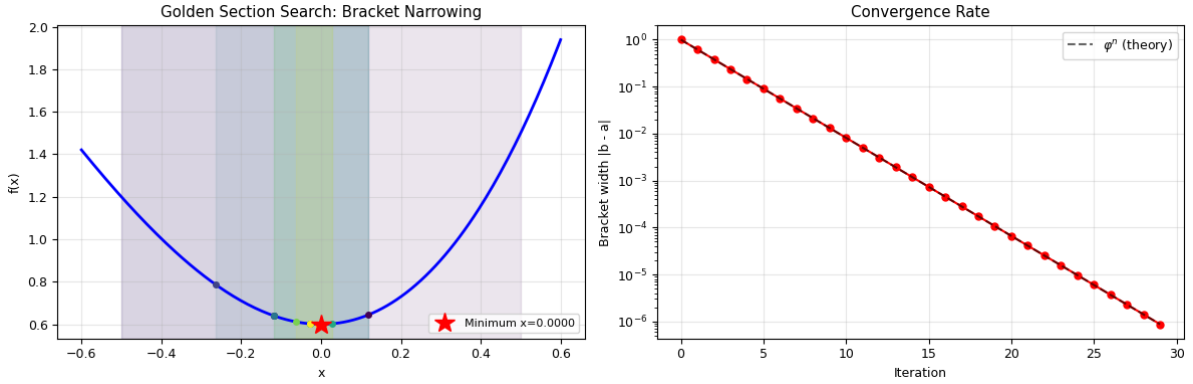
```
Golden section search on f(x) = 1.2x³ + 3.0x² + 0.6
Bracket: [-0.5, 0.5]
Found minimum: x = 0.000000, f = 0.600000
Steps: 29

Analytical minimum: x = 0, f = 0.6
```

```
# Visualize the bracket narrowing
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

x_plot = np.linspace(-0.6, 0.6, 200)
axes[0].plot(x_plot, f(x_plot), 'b-', lw=2)

# Show first 6 brackets
colors = plt.cm.viridis(np.linspace(0, 1, min(6, len(hist))))
for k in range(min(6, len(hist))):
    a, b, x1, x2 = hist[k]
    axes[0].axvspan(a, b, alpha=0.1, color=colors[k])
    axes[0].plot([x1, x2], [f(x1), f(x2)], 'o', color=colors[k], ms=4)

axes[0].plot(x_opt, f_opt, 'r*', ms=15, label=f'Minimum x={x_opt:.4f}')
axes[0].set_xlabel('x')
axes[0].set_ylabel('f(x)')
axes[0].set_title('Golden Section Search: Bracket Narrowing')
axes[0].legend(fontsize=8)
axes[0].grid(alpha=0.3)

# Right: bracket width vs iteration
widths = [h[1] - h[0] for h in hist]
axes[1].semilogy(range(len(widths)), widths, 'ro-', ms=5)
axes[1].plot(range(len(widths)), widths[0] * 0.618**np.arange(len(widths)), 'k--',
             label='$\\varphi^n$ (theory)', alpha=0.6)
axes[1].set_xlabel('Iteration')
axes[1].set_ylabel('Bracket width |b - a|')
axes[1].set_title('Convergence Rate')
```

(continues on next page)

(continued from previous page)

```
axes[1].legend()
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Bracket shrinks by factor ψ ≈ 0.618 each step")
print(f"After {steps} steps: bracket width = {widths[-1]:.2e}")
print(f"Only 1 function evaluation per step (not 2)!")
```




```
Bracket shrinks by factor ψ ≈ 0.618 each step
After 29 steps: bracket width = 8.70e-07
Only 1 function evaluation per step (not 2)!
```

15.4.4 Golden Section vs Gradient Descent

Feature	Gradient Descent	Golden Section
Requires derivative?	Yes	**No**
Requires bracket?	No	**Yes** (must contain minimum)
Convergence	Depends on step size	**Guaranteed** linear (factor 0.618)
Works in N-D?	Yes	Only 1D
Multiple minima?	Finds nearest	Finds the one in the bracket

15.5 IV. Extending to 2D: Gradient Descent on Surfaces

15.5.1 2D Test Functions

Now we extend gradient descent to functions of two variables. The gradient replaces the derivative:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla f(\mathbf{x}_n)$$

where $\mathbf{x} = (x, y)$ and $\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$.

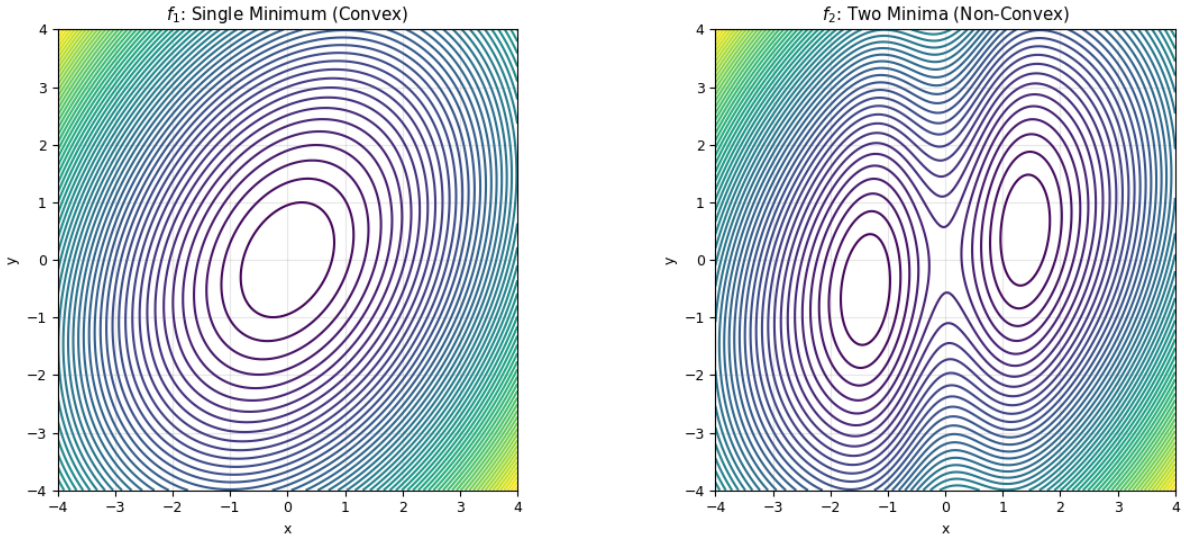
We'll use two test functions:

- f_1: Single minimum (convex) $\$f_1(x, y) = \frac{x^2}{2} + \frac{y^2}{3} - \frac{xy}{4}\$$

- f_2: Two minima (non-convex) $f_2(x, y) = \frac{x^2}{2} + \frac{y^2}{3} - \frac{xy}{4} + 3e^{-x^2}$

The Gaussian bump in f_2 creates a second local minimum.

```
def f1(x):
    """Simple function with 1 minimum."""
    return x[0]**2 / 2 + x[1]**2 / 3 - x[0] * x[1] / 4

def f2(x):
    """Function with 2 minima (Gaussian bump)."""
    return x[0]**2 / 2 + x[1]**2 / 3 - x[0] * x[1] / 4 + 3 * np.exp(-x[0]**2)

# Visualize both functions
x_min, x_max = -4, 4
y_min, y_max = -4, 4
nx = np.linspace(x_min, x_max, 400)
ny = np.linspace(y_min, y_max, 400)
x, y = np.meshgrid(nx, ny)

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

z1 = f1([x, y])
levels1 = np.arange(np.min(z1), np.max(z1), 0.3)
axes[0].contour(x, y, z1, levels=levels1, cmap='viridis')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].set_title('$f_1$: Single Minimum (Convex)')
axes[0].set_aspect('equal')
axes[0].grid(alpha=0.3)

z2 = f2([x, y])
levels2 = np.arange(np.min(z2), np.max(z2), 0.3)
axes[1].contour(x, y, z2, levels=levels2, cmap='viridis')
axes[1].set_xlabel('x')
axes[1].set_ylabel('y')
axes[1].set_title('$f_2$: Two Minima (Non-Convex)')
axes[1].set_aspect('equal')
axes[1].grid(alpha=0.3)

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



15.5.2 2D Numerical Gradient

We compute the gradient using **central differences** in each direction:

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\delta/2, y) - f(x-\delta/2, y)}{\delta}$$

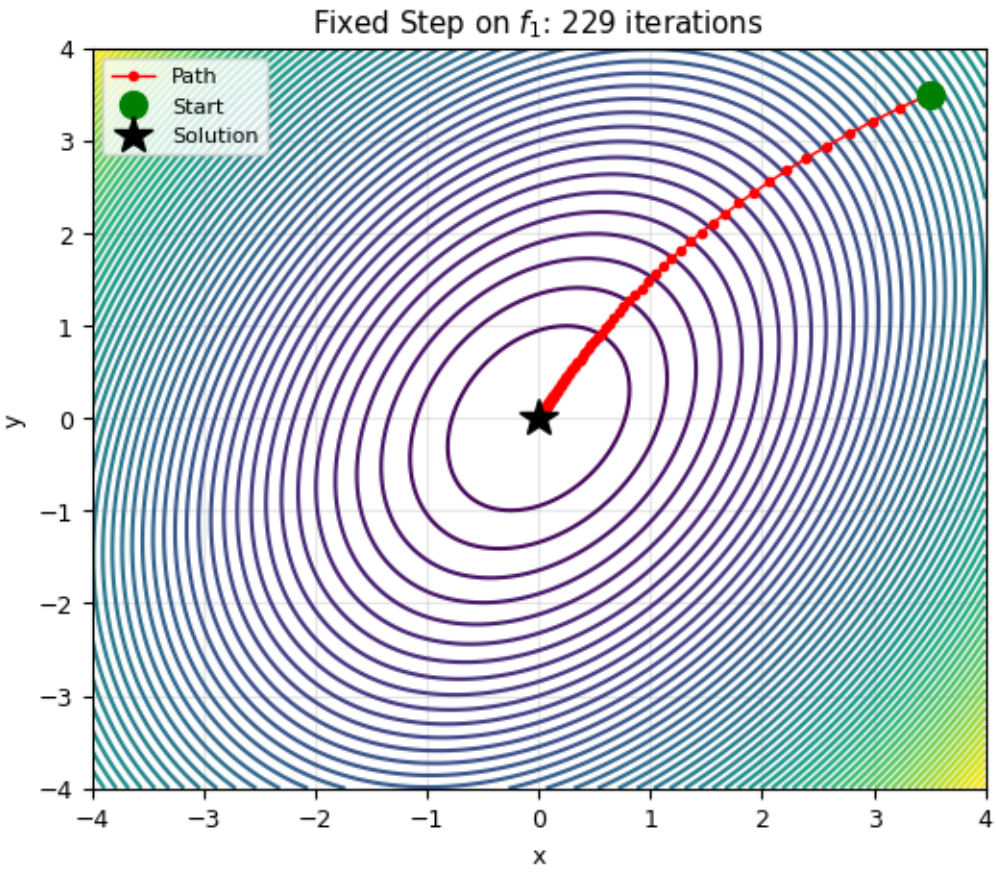
$$\frac{\partial f}{\partial y} \approx \frac{f(x, y+\delta/2) - f(x, y-\delta/2)}{\delta}$$

```python
def derivative2(f, xy, d=0.001):
    """Numerical gradient of a 2D function using central differences."""
    x, y = xy[0], xy[1]
    fx = (f([x + d/2, y]) - f([x - d/2, y])) / d
    fy = (f([x, y + d/2]) - f([x, y - d/2])) / d
    return np.array([fx, fy])

def init(x_min, x_max, y_min, y_max):
    """Generate random starting point in [x_min, x_max] x [y_min, y_max]."""
    x0 = x_min + np.random.random() * (x_max - x_min)
    y0 = y_min + np.random.random() * (y_max - y_min)
    return [x0, y0]

# Test the gradient
test_point = [1.0, 2.0]
grad = derivative2(f1, test_point)
print(f"Gradient of f1 at (1, 2): [{grad[0]:.4f}, {grad[1]:.4f}]")
print(f"Analytical: [{1.0 - 2.0/4:.4f}, {2*2.0/3 - 1.0/4:.4f}]")
```

```
Gradient of f1 at (1, 2): [0.5000, 1.0833]
Analytical: [0.5000, 1.0833]
```

15.5.3 Method 1: Fixed Step Size in 2D

```python
def minimize_fix(f, x0, dx=0.05, N=1000):
    """2D gradient descent with fixed step size."""
    x_now = np.array(x0, dtype=float)
    converged = False
    x_hist = [x_now.copy()]

    for i in range(N):
        # code...
        x_next = x_now - dx * derivative2(f, x_now)

        if np.linalg.norm(x_next - x_now) < 1e-6:
            converged = True
            break
        else:
            x_now = x_next
            x_hist.append(x_now.copy())

    return converged, np.array(x_hist), f(x_now)

# Test on f1 (single minimum)
x0, y0 = 3.5, 3.5
converged, x_hist, f_min = minimize_fix(f1, [x0, y0], dx=0.1)

z1 = f1([x, y])
plt.figure(figsize=(6, 5))
plt.contour(x, y, z1, levels=np.arange(0, np.max(z1), 0.3), cmap='viridis')
plt.plot(x_hist[:, 0], x_hist[:, 1], 'ro-', ms=3, lw=1, label='Path')
plt.plot(x0, y0, 'go', ms=10, label='Start')
plt.plot(x_hist[-1, 0], x_hist[-1, 1], 'k*', ms=15, label='Solution')
plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Fixed Step on $f_1$: {len(x_hist)} iterations')
plt.legend(fontsize=8)
plt.grid(alpha=0.3)
plt.show()

print(f'Converged: {converged}')
print(f'Start: ({x0}, {y0}), f = {f1([x0, y0]):.4f}')
print(f'End:   ({x_hist[-1, 0]:.4f}, {x_hist[-1, 1]:.4f}), f = {f_min:.4f}')
print(f'Iterations: {len(x_hist)}')
```



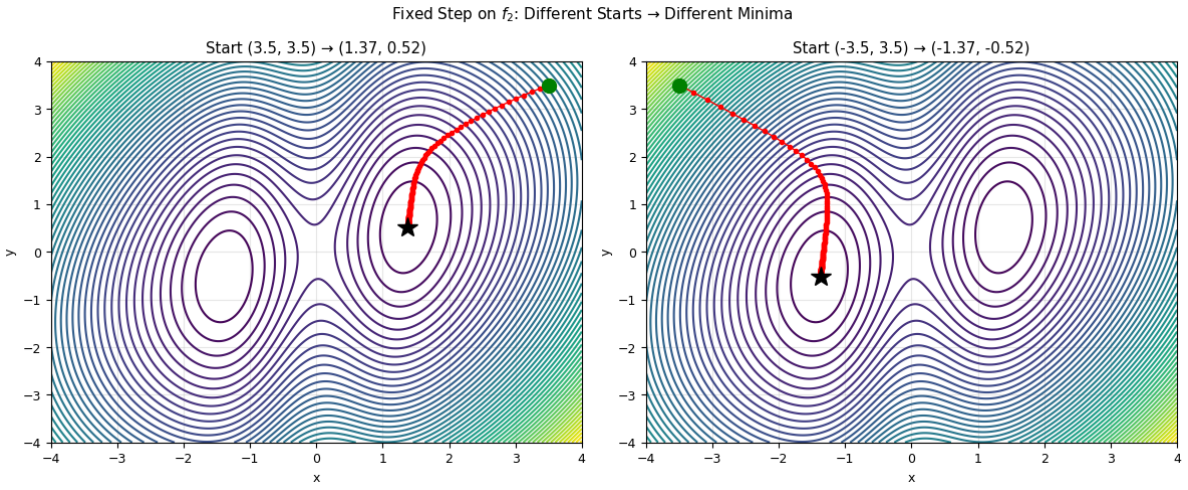

```
Converged: True
Start: (3.5, 3.5), f = 7.1458
End:   (0.0000, 0.0000), f = 0.0000
Iterations: 229
```

```
# Test on f2 (two minima) — starting point determines which minimum we find
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

z2 = f2([x, y])
levels2 = np.arange(np.min(z2), np.max(z2), 0.3)

# Start from right side
converged, x_hist_r, f_min_r = minimize_fix(f2, [3.5, 3.5])
axes[0].contour(x, y, z2, levels=levels2, cmap='viridis')
axes[0].plot(x_hist_r[:, 0], x_hist_r[:, 1], 'ro-', ms=3, lw=1)
axes[0].plot(3.5, 3.5, 'go', ms=10)
axes[0].plot(x_hist_r[-1, 0], x_hist_r[-1, 1], 'k*', ms=15)
axes[0].set_title(f'Start (3.5, 3.5) → ({x_hist_r[-1,0]:.2f}, {x_hist_r[-1,1]:.2f})')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].grid(alpha=0.3)

# Start from left side
converged, x_hist_l, f_min_l = minimize_fix(f2, [-3.5, 3.5])
axes[1].contour(x, y, z2, levels=levels2, cmap='viridis')
axes[1].plot(x_hist_l[:, 0], x_hist_l[:, 1], 'ro-', ms=3, lw=1)
```

(continues on next page)

(continued from previous page)

```
axes[1].plot(-3.5, 3.5, 'go', ms=10)
axes[1].plot(x_hist_l[-1, 0], x_hist_l[-1, 1], 'k*', ms=15)
axes[1].set_title(f'Start (-3.5, 3.5) → ({x_hist_l[-1,0]:.2f}, {x_hist_l[-1,1]:.2f})')
axes[1].set_xlabel('x')
axes[1].set_ylabel('y')
axes[1].grid(alpha=0.3)

plt.suptitle('Fixed Step on $f_2$: Different Starts → Different Minima', fontsize=11)
plt.tight_layout()
plt.show()

print(f"Right start → f = {f_min_r:.4f} ({len(x_hist_r)} steps)")
print(f"Left start  → f = {f_min_l:.4f} ({len(x_hist_l)} steps)")
```




```
Right start → f = 1.3096 (355 steps)
Left start  → f = 1.3096 (358 steps)
```

15.5.4 Method 2: Variable Step Size (Barzilai-Borwein) in 2D

The 2D generalization of the Barzilai-Borwein method:

$$\eta_n = \frac{(\mathbf{x}_n - \mathbf{x}_{n-1})^T[\nabla f(\mathbf{x}_n) - \nabla f(\mathbf{x}_{n-1})]}{\|\nabla f(\mathbf{x}_n) - \nabla f(\mathbf{x}_{n-1})\|^2}$$

```
def minimize_var(f, x0, N=1000):
    """2D gradient descent with Barzilai-Borwein variable step size."""
    x_now = np.array(x0, dtype=float)
    x_prev = None
    converged = False
    x_hist = [x_now.copy()]

    for i in range(N):
        df_now = derivative2(f, x_now)

        if x_prev is None:
            dx = 0.01
        else:
```

(continues on next page)

(continued from previous page)

```
            df_prev = derivative2(f, x_prev)
            dd = df_now - df_prev
            denom = np.linalg.norm(dd)**2
            if denom > 1e-16:
                dx = np.dot(x_now - x_prev, dd) / denom
            else:
                dx = 0.01

        x_next = x_now - df_now * dx

        if f(x_next) < f(x_now):
            x_prev = x_now.copy()
            x_now = x_next
            x_hist.append(x_now.copy())
        else:
            converged = True
            break

    return converged, np.array(x_hist), f(x_now)
```

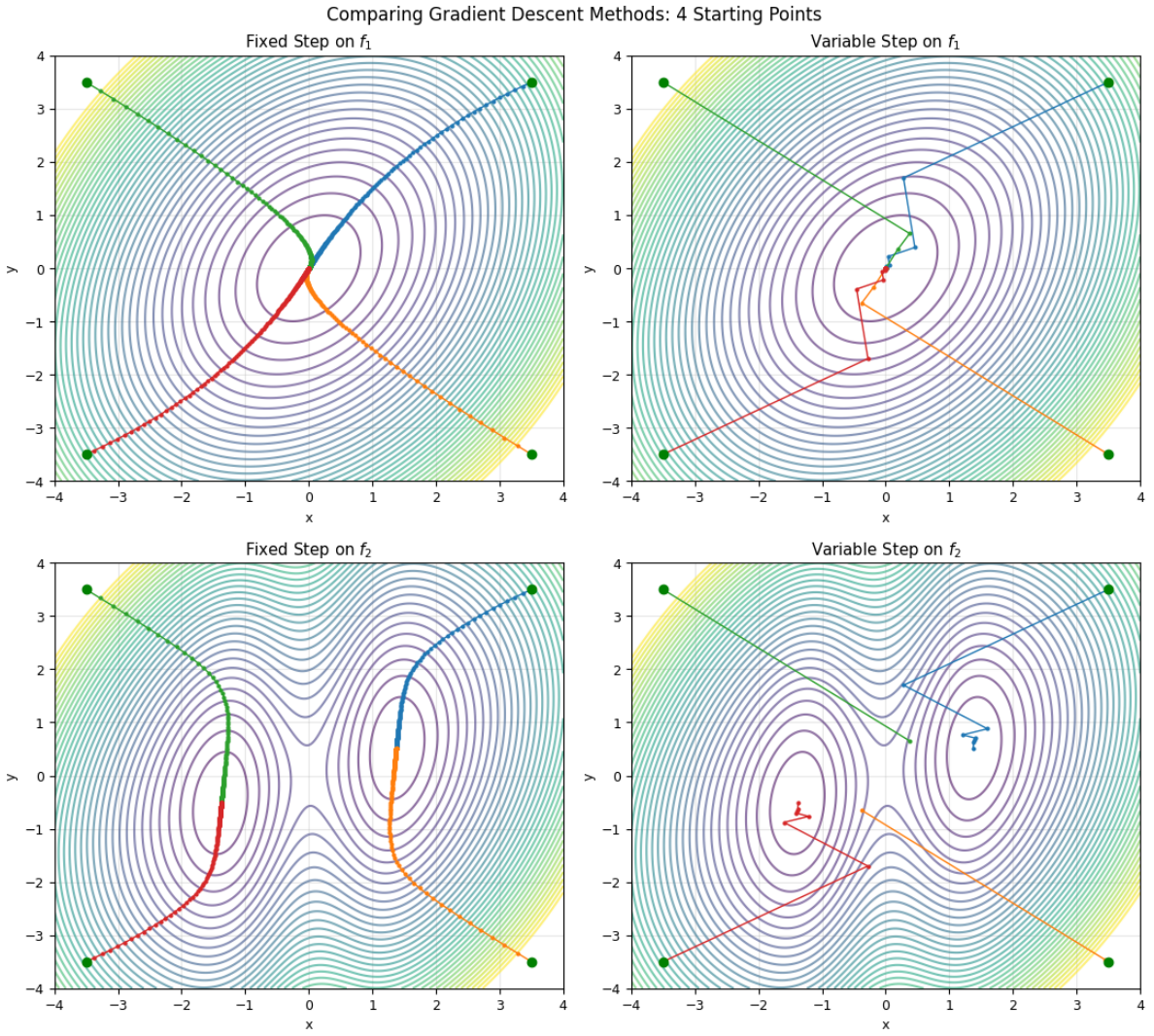
15.5.5 Comparing Both Methods in 2D

```
# Compare both methods on f1 and f2
start_points = [[3.5, 3.5], [3.5, -3.5], [-3.5, 3.5], [-3.5, -3.5]]
methods = [
    ('Fixed Step', minimize_fix),
    ('Variable Step', minimize_var),
]

fig, axes = plt.subplots(2, 2, figsize=(11, 10))

for col, (name, method) in enumerate(methods):
    # f1 (top row)
    axes[0, col].contour(x, y, f1([x, y]),
                         levels=np.arange(0, 12, 0.3), cmap='viridis', alpha=0.6)
    for sp in start_points:
        conv, xh, fm = method(f1, sp)
        axes[0, col].plot(xh[:, 0], xh[:, 1], 'o-', ms=2, lw=1)
        axes[0, col].plot(sp[0], sp[1], 'go', ms=6)
    axes[0, col].set_title(f'{name} on $f_1$')
    axes[0, col].set_xlabel('x')
    axes[0, col].set_ylabel('y')
    axes[0, col].grid(alpha=0.3)

    # f2 (bottom row)
    axes[1, col].contour(x, y, f2([x, y]),
                         levels=np.arange(np.min(z2), 12, 0.3), cmap='viridis',␣
↪alpha=0.6)
    for sp in start_points:
        conv, xh, fm = method(f2, sp)
        axes[1, col].plot(xh[:, 0], xh[:, 1], 'o-', ms=2, lw=1)
        axes[1, col].plot(sp[0], sp[1], 'go', ms=6)
    axes[1, col].set_title(f'{name} on $f_2$')
    axes[1, col].set_xlabel('x')
    axes[1, col].set_ylabel('y')
```

(continues on next page)

(continued from previous page)

```
    axes[1, col].grid(alpha=0.3)

plt.suptitle('Comparing Gradient Descent Methods: 4 Starting Points', fontsize=12)
plt.tight_layout()
plt.show()
```




```
# Quantitative comparison on f2
print("Comparison on f2 (two minima)")
print(f'{"Start":>14s} | {"Method":>12s} | {"Conv":>5s} | {"Final (x, y)":>20s} | {"f_
↪min":>8s} | {"Steps":>5s}')
print('-' * 80)

for sp in start_points:
    for name, method in methods:
        conv, xh, fm = method(f2, sp)
        end = xh[-1]
        print(f'({sp[0]:5.1f},{sp[1]:5.1f}) | {name:>12s} | {str(conv):>5s} | (
↪{end[0]:8.4f}, {end[1]:8.4f}) | {fm:8.4f} | {len(xh):5d}')
    print()
```

```
Comparison on f2 (two minima)
         Start |       Method |  Conv |         Final (x, y) |    f_min | Steps
-------------------------------------------------------------------------------
(  3.5,  3.5) |   Fixed Step |  True | (  1.3748,   0.5156) |   1.3096 |   355
(  3.5,  3.5) | Variable Step |  True | (  1.3762,   0.5162) |   1.3096 |     9

(  3.5, -3.5) |   Fixed Step |  True | (  1.3748,   0.5155) |   1.3096 |   358
(  3.5, -3.5) | Variable Step |  True | ( -0.3739,  -0.6540) |   2.7599 |     3

( -3.5,  3.5) |   Fixed Step |  True | ( -1.3748,  -0.5155) |   1.3096 |   358
( -3.5,  3.5) | Variable Step |  True | (  0.3739,   0.6540) |   2.7599 |     3

( -3.5, -3.5) |   Fixed Step |  True | ( -1.3748,  -0.5156) |   1.3096 |   355
( -3.5, -3.5) | Variable Step |  True | ( -1.3762,  -0.5162) |   1.3096 |     9
```

15.6 V. Using `scipy.optimize`

15.6.1 Professional Optimization with `scipy.optimize.minimize`

In practice, we use well-tested library implementations. `scipy.optimize.minimize` provides many algorithms through a unified interface:

Method	Type	Requires	Best for
`'Nelder-Mead'`	Simplex (gradient-free)	Only f	Noisy or non-smooth functions
`'Powell'`	Direction set (gradient-free)	Only f	Low-dimensional smooth functions
`'CG'`	Conjugate gradient	$f, \nabla f$	Large-scale smooth problems
`'BFGS'`	Quasi-Newton	$f, \nabla f$	General smooth optimization
`'Newton-CG'`	Newton + CG	$f, \nabla f, H$	When Hessian is available

We'll study Newton, BFGS, and CG in detail in the next lecture. For now, let's see them in action.

```python
from scipy.optimize import minimize as sp_minimize

x0 = [3.5, 3.5]

print(f"Minimizing f2 from starting point {x0}")
print("=" * 60)

for method in ['Nelder-Mead', 'Powell', 'CG', 'BFGS']:
    res = sp_minimize(f2, x0, method=method, tol=1e-6)
    print(f"\n{method}:")
    print(f"  Solution: ({res.x[0]:.6f}, {res.x[1]:.6f})")
    print(f"  f_min:    {res.fun:.6f}")
    print(f"  Evaluations: {res.nfev}")
    print(f"  Converged: {res.success}")
```

```
Minimizing f2 from starting point [3.5, 3.5]
============================================================

Nelder-Mead:
  Solution: (1.374845, 0.515567)
```

(continues on next page)

(continued from previous page)

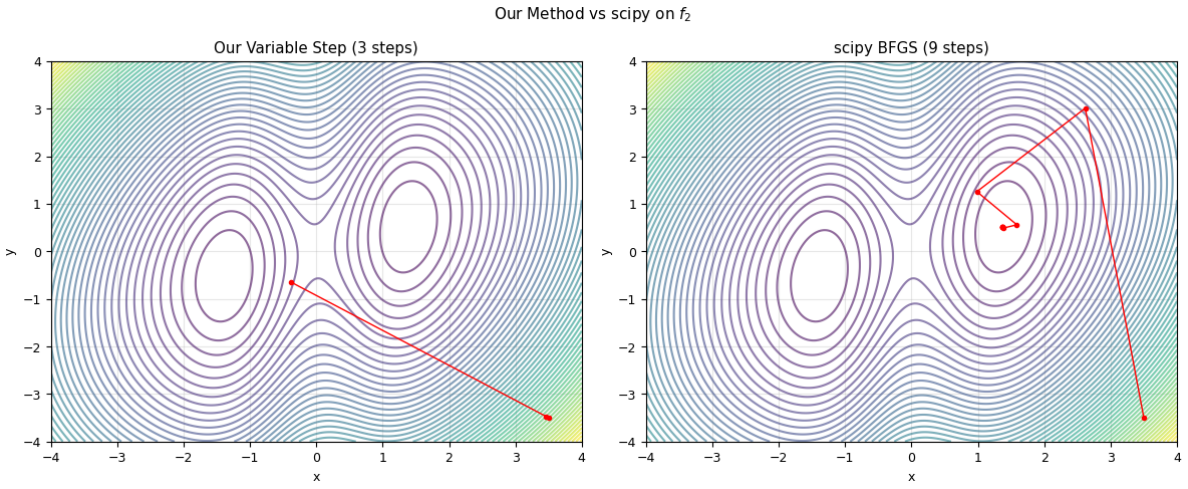
```
  f_min:    1.309622
  Evaluations: 114
  Converged: True

Powell:
  Solution: (1.374845, 0.515567)
  f_min:    1.309622
  Evaluations: 126
  Converged: True

CG:
  Solution: (-1.374845, -0.515566)
  f_min:    1.309622
  Evaluations: 60
  Converged: True

BFGS:
  Solution: (1.374845, 0.515567)
  f_min:    1.309622
  Evaluations: 27
  Converged: True
```

```
# Visual comparison: our methods vs scipy
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

z2 = f2([x, y])
levels2 = np.arange(np.min(z2), np.max(z2), 0.3)

# Left: our variable step method
conv, xh_ours, fm = minimize_var(f2, [3.5, -3.5])
axes[0].contour(x, y, z2, levels=levels2, cmap='viridis', alpha=0.6)
axes[0].plot(xh_ours[:, 0], xh_ours[:, 1], 'ro-', ms=3, lw=1)
axes[0].set_title(f'Our Variable Step ({len(xh_ours)} steps)')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].grid(alpha=0.3)

# Right: scipy BFGS (collect trajectory using callback)
trajectory = [[3.5, -3.5]]
def callback(xk):
    trajectory.append(xk.tolist())

res = sp_minimize(f2, [3.5, 3.5], method='BFGS', callback=callback, tol=1e-6)
traj = np.array(trajectory)

axes[1].contour(x, y, z2, levels=levels2, cmap='viridis', alpha=0.6)
axes[1].plot(traj[:, 0], traj[:, 1], 'ro-', ms=3, lw=1)
axes[1].set_title(f'scipy BFGS ({len(traj)} steps)')
axes[1].set_xlabel('x')
axes[1].set_ylabel('y')
axes[1].grid(alpha=0.3)

plt.suptitle('Our Method vs scipy on $f_2$', fontsize=11)
plt.tight_layout()
plt.show()
```

(continues on next page)

(continued from previous page)

```
print(f"Our method: {len(xh_ours)} steps, f = {fm:.6f}")
print(f"scipy BFGS: {len(traj)} steps, f = {res.fun:.6f}")
print(f"\nBFGS is much more efficient — we'll learn why in the next lecture!")
```




```
Our method: 3 steps, f = 2.759868
scipy BFGS: 9 steps, f = 1.309622

BFGS is much more efficient — we'll learn why in the next lecture!
```

15.6.2 Summary: What We Learned Today

Method	Derivatives?	Dimension	Key Idea
Gradient descent (fixed η)	Yes	Any	Simple but sensitive to η
Barzilai-Borwein (variable η)	Yes	Any	Adapts η from previous step
Golden section search	**No**	1D only	Bracket narrowing via golden ratio
Nelder-Mead (scipy)	**No**	Any	Simplex exploration
BFGS (scipy)	Yes	Any	Quasi-Newton (next lecture!)

CHAPTER

SIXTEEN

LECTURE 16: OPTIMIZATION II — NEWTON'S METHOD, BFGS, AND CONJUGATE GRADIENT

16.1 Beyond Gradient Descent: Using Curvature

In Lecture 15, we built gradient descent optimizers — they only use the **first derivative** (gradient) to decide which direction to step.

The key limitation: gradient descent doesn't know the **shape** of the landscape. Is the valley wide and flat, or narrow and steep? A method that uses **curvature** (second derivative) can adapt its step to the local geometry.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize as sp_minimize
```

16.2 I. Newton's Method: Quadratic Convergence

16.2.1 The Idea: Fit a Quadratic, Jump to Its Minimum

Gradient descent uses a **linear** approximation of f at the current point. Newton's method uses a **quadratic** approximation (Taylor expansion to second order):

$$f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^T \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T \, \mathbf{H} \, \Delta\mathbf{x}$$

where $\mathbf{H}$ is the **Hessian matrix** of second partial derivatives:

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

Setting the gradient of the quadratic model to zero gives the **Newton step**:

$$\Delta\mathbf{x} = -\mathbf{H}^{-1} \nabla f(\mathbf{x})$$

For a **quadratic function**, Newton's method converges in **one step** — it jumps directly to the minimum!

For general functions, convergence is **quadratic**: the error squares at each step ($\epsilon_{n+1} \propto \epsilon_n^2$). Compare this to gradient descent's **linear** convergence ($\epsilon_{n+1} \propto c \cdot \epsilon_n$).

16.2.2 1D Newton's Method: Root Finding Meets Optimization

In 1D, the Hessian is just $f''(x)$, and the Newton step becomes:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

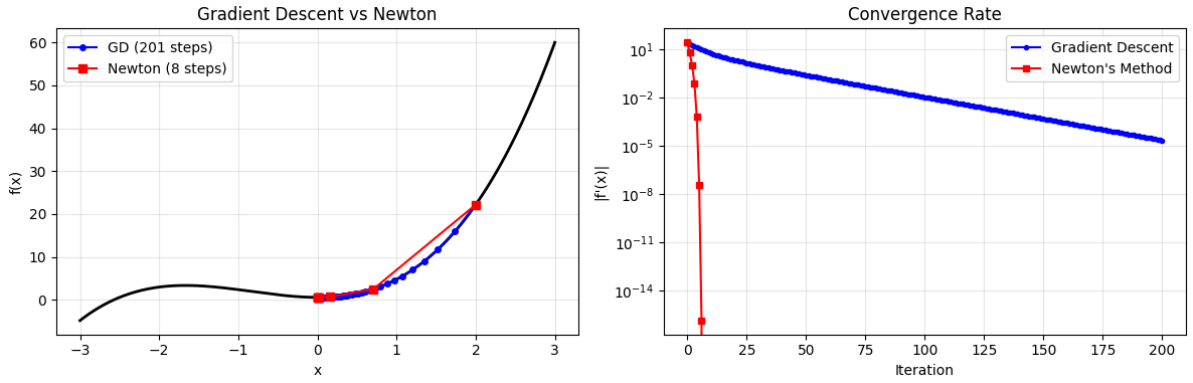
This is equivalent to finding the **root of the derivative** — exactly Newton-Raphson applied to $f'(x) = 0$.

```
# Reuse our test function from Lecture 15
A, B, C = 1.2, 3.0, 0.6

def f(x):
    return A * x**3 + B * x**2 + C

def f_prime(x):
    return 3 * A * x**2 + 2 * B * x

def f_double_prime(x):
    return 6 * A * x + 2 * B

# 1D Newton's method
def newton_1d(x0, N=50, tol=1e-10):
    x = x0
    x_hist = [x]
    for i in range(N):
        ## code ...
        fp = f_prime(x)
        fpp = f_double_prime(x)
        if abs(fpp) < 1e-12:
            break
        dx = -fp/fpp
        x += dx
        x_hist.append(x)
        if abs(dx) < tol:
            break

    return np.array(x_hist)

# Compare: gradient descent vs Newton
def gradient_descent_1d(x0, eta=0.01, N=200, tol=1e-10):
    x = x0
    x_hist = [x]
    for i in range(N):
        dx = -eta * f_prime(x)
        x = x + dx
        x_hist.append(x)
        if abs(dx) < tol:
            break
    return np.array(x_hist)

x0 = 2.0
xh_gd = gradient_descent_1d(x0, eta=0.01)
xh_newton = newton_1d(x0)

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Left: trajectories
```

(continues on next page)

(continued from previous page)

```
xp = np.linspace(-3, 3, 300)
axes[0].plot(xp, f(xp), 'k-', lw=2)
axes[0].plot(xh_gd, f(xh_gd), 'bo-', ms=4, label=f'GD ({len(xh_gd)} steps)')
axes[0].plot(xh_newton, f(xh_newton), 'rs-', ms=6, label=f'Newton ({len(xh_newton)}␣
↪steps)')
axes[0].set_xlabel('x')
axes[0].set_ylabel('f(x)')
axes[0].set_title('Gradient Descent vs Newton')
axes[0].legend()
axes[0].grid(alpha=0.3)

# Right: convergence
axes[1].semilogy(range(len(xh_gd)), np.abs(f_prime(xh_gd)), 'bo-', ms=3, label=
↪'Gradient Descent')
axes[1].semilogy(range(len(xh_newton)), np.abs(f_prime(xh_newton)), 'rs-', ms=5,␣
↪label="Newton's Method")
axes[1].set_xlabel('Iteration')
axes[1].set_ylabel("|f'(x)|")
axes[1].set_title('Convergence Rate')
axes[1].legend()
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print(f'Gradient descent: {len(xh_gd)-1} steps, x* = {xh_gd[-1]:.8f}')
print(f"Newton's method:  {len(xh_newton)-1} steps, x* = {xh_newton[-1]:.8f}")
```




```
Gradient descent: 200 steps, x* = 0.00000365
Newton's method:  7 steps, x* = 0.00000000
```

16.2.3 Extending to 2D: The Hessian Matrix

For a function $f(x, y)$, the Hessian is:

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

We compute it numerically using **central differences**.

```
# Test functions from Lecture 15
def f1(x):
    return x[0]**2 / 2 + x[1]**2 / 3 - x[0] * x[1] / 4

def f2(x):
    return x[0]**2 / 2 + x[1]**2 / 3 - x[0] * x[1] / 4 + 3 * np.exp(-x[0]**2)

# Numerical gradient (central difference)
def derivative2(f, xy, d=0.001):
    x, y = xy[0], xy[1]
    fx = (f([x + d/2, y]) - f([x - d/2, y])) / d
    fy = (f([x, y + d/2]) - f([x, y - d/2])) / d
    return np.array([fx, fy])

# Numerical Hessian (central difference)
def hessian2(f, xy, d=0.001):
    x, y = xy[0], xy[1]
    fxx = (f([x+d, y]) - 2*f([x, y]) + f([x-d, y])) / d**2
    fyy = (f([x, y+d]) - 2*f([x, y]) + f([x, y-d])) / d**2
    fxy = (f([x+d/2, y+d/2]) - f([x+d/2, y-d/2])
         - f([x-d/2, y+d/2]) + f([x-d/2, y-d/2])) / d**2
    return np.array([[fxx, fxy],
                     [fxy, fyy]])

# Grid for contour plots
x_min, x_max = -4, 4
nx = np.linspace(x_min, x_max, 400)
x, y = np.meshgrid(nx, nx)
```

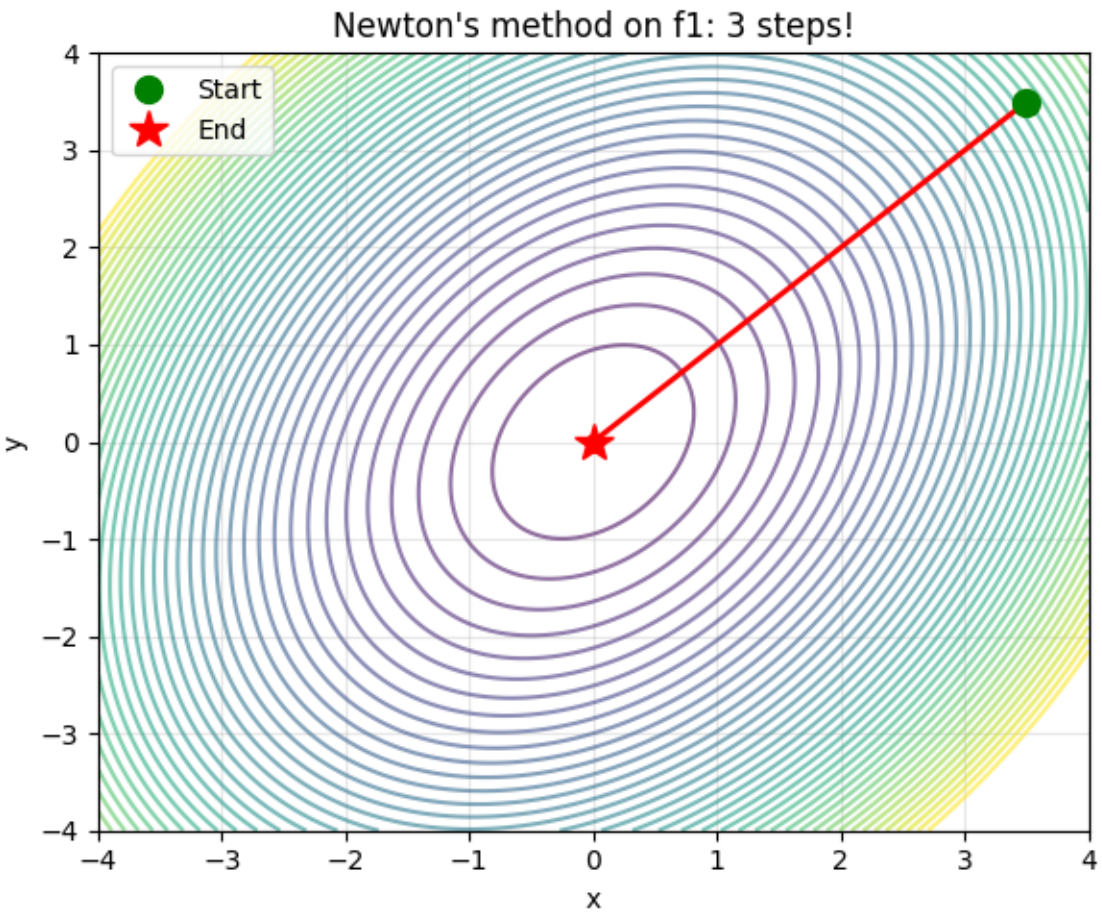
```
def minimize_newton(f, x0, N=100, tol=1e-8):
    """2D Newton's method using numerical Hessian."""
    x_now = np.array(x0, dtype=float)
    x_hist = [x_now.copy()]

    for i in range(N):
        grad = derivative2(f, x_now)
        H = hessian2(f, x_now)

        # Newton step: dx = -H^{-1} grad
        try:
            dx = np.linalg.solve(H, -grad)
        except np.linalg.LinAlgError:
            break  # Singular Hessian

        x_next = x_now + dx
        x_hist.append(x_next.copy())

        if np.linalg.norm(dx) < tol:
```

(continues on next page)

(continued from previous page)

```
            break

        x_now = x_next

    return True, np.array(x_hist), f(x_now)
```

```
# Newton on f1 (nearly quadratic) -- expect very fast convergence
conv, xh_n, fm = minimize_newton(f1, [3.5, 3.5])

z1 = f1([x, y])
levels1 = np.arange(0, 12, 0.3)

fig, ax = plt.subplots(figsize=(6, 5))
ax.contour(x, y, z1, levels=levels1, cmap='viridis', alpha=0.6)
ax.plot(xh_n[:, 0], xh_n[:, 1], 'ro-', ms=6, lw=2)
ax.plot(xh_n[0, 0], xh_n[0, 1], 'go', ms=10, label='Start')
ax.plot(xh_n[-1, 0], xh_n[-1, 1], 'r*', ms=15, label='End')
ax.set_title(f"Newton's method on f1: {len(xh_n)-1} steps!")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
ax.grid(alpha=0.3)
plt.tight_layout()
plt.show()

print(f'Steps: {len(xh_n)-1}')
print(f'Minimum at: ({xh_n[-1,0]:.8f}, {xh_n[-1,1]:.8f})')
print(f'f_min = {fm:.10f}')
```




```
Steps: 3
Minimum at: (0.00000000, 0.00000000)
f_min = 0.0000000000
```

16.2.4 When Newton's Method Fails

Newton's method is powerful but has important limitations:

1. **Non-convex functions**: If the Hessian has negative eigenvalues, the Newton step may go to a **maximum** or **saddle point** instead of a minimum
2. **Far from minimum**: The quadratic approximation may be poor, causing the step to overshoot wildly
3. **Singular Hessian**: At inflection points, $\det(\mathbf{H}) = 0$ and the method breaks down

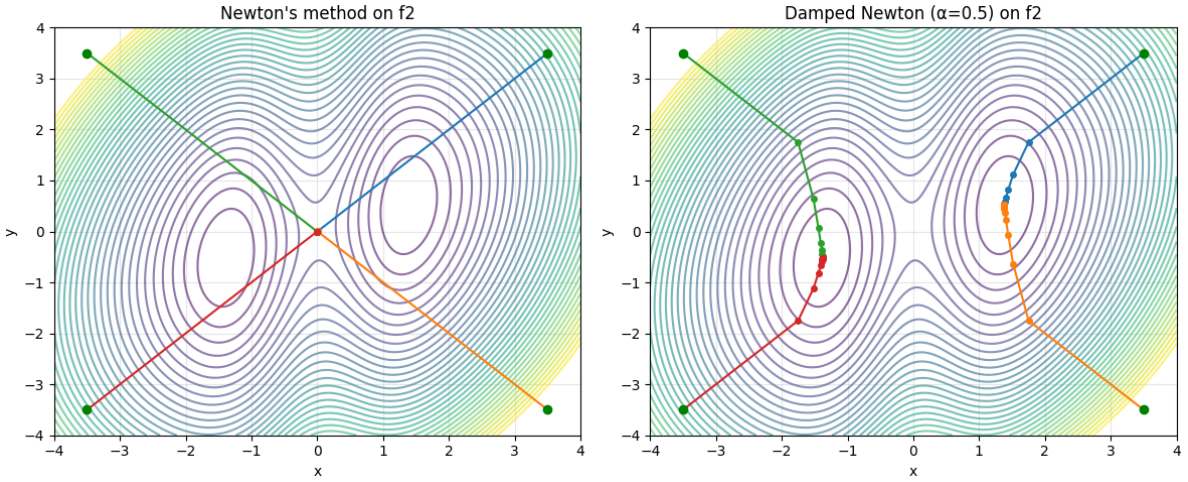
A practical fix: **damped Newton** — take only a fraction of the Newton step: $$\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha \, \Delta\mathbf{x}_{\text{Newton}}, \quad 0 < \alpha \leq 1$$

```
# Newton on f2 from different starting points
start_points = [[3.5, 3.5], [3.5, -3.5], [-3.5, 3.5], [-3.5, -3.5]]

z2 = f2([x, y])
levels2 = np.arange(np.min(z2), 12, 0.3)

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
```

(continues on next page)

(continued from previous page)

```
# Newton
axes[0].contour(x, y, z2, levels=levels2, cmap='viridis', alpha=0.6)
for sp in start_points:
    conv, xh, fm = minimize_newton(f2, sp)
    axes[0].plot(xh[:, 0], xh[:, 1], 'o-', ms=4, lw=1.5)
    axes[0].plot(sp[0], sp[1], 'go', ms=6)
axes[0].set_title("Newton's method on f2")
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].grid(alpha=0.3)

# Damped Newton
def minimize_damped_newton(f, x0, alpha=0.5, N=200, tol=1e-8):
    x_now = np.array(x0, dtype=float)
    x_hist = [x_now.copy()]
    for i in range(N):
        grad = derivative2(f, x_now)
        H = hessian2(f, x_now)
        try:
            dx = np.linalg.solve(H, -grad)
        except np.linalg.LinAlgError:
            break
        x_next = x_now + alpha * dx
        x_hist.append(x_next.copy())
        if np.linalg.norm(alpha * dx) < tol:
            break
        x_now = x_next
    return True, np.array(x_hist), f(x_now)

axes[1].contour(x, y, z2, levels=levels2, cmap='viridis', alpha=0.6)
for sp in start_points:
    conv, xh, fm = minimize_damped_newton(f2, sp, alpha=0.5)
    axes[1].plot(xh[:, 0], xh[:, 1], 'o-', ms=4, lw=1.5)
    axes[1].plot(sp[0], sp[1], 'go', ms=6)
axes[1].set_title("Damped Newton (\u03b1=0.5) on f2")
axes[1].set_xlabel('x')
axes[1].set_ylabel('y')
axes[1].grid(alpha=0.3)

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



16.2.5 Cost Analysis

	Gradient Descent	Newton's Method
Per step	1 gradient evaluation (n function calls)	1 gradient + 1 Hessian ($n + n^2$ calls) + solve $n \times n$ system
Convergence	Linear: $\epsilon_{n+1} \approx c \cdot \epsilon_n$	Quadratic: $\epsilon_{n+1} \approx C \cdot \epsilon_n^2$
Total steps	Many (100s–1000s)	Few (5–20)
Scaling	$O(n)$ per step	$O(n^3)$ per step (matrix solve)

For small problems ($n \lesssim 100$), Newton's method wins easily.
For large problems ($n \gg 1000$), the $O(n^3)$ cost per step becomes prohibitive.

Can we get Newton-like convergence without computing the full Hessian? → Yes! That's the idea behind BFGS and conjugate gradient.

16.3 II. Conjugate Gradient: Smart Search Directions

16.3.1 The Problem with Gradient Descent

Look at the path gradient descent takes on an elliptical valley — it **zigzags**! Each step is perpendicular to the previous one (the new gradient is orthogonal to the old step), so it keeps revisiting the same directions.

Conjugate gradient (CG) fixes this by choosing search directions that are **conjugate** (or **H**-orthogonal):

$$\mathbf{d}_i^T \mathbf{H}\, \mathbf{d}_j = 0 \quad \text{for } i \neq j$$

This ensures each direction explores a **new** dimension of the landscape. For a quadratic function in n dimensions, CG converges in exactly n steps!

16.3.2 The Algorithm (Fletcher-Reeves)

1. Start: $\mathbf{d}_0 = -\nabla f(\mathbf{x}_0)$ (steepest descent direction)
2. Line search: find α_k that minimizes $f(\mathbf{x}_k + \alpha_k \mathbf{d}_k)$
3. Update: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$
4. New direction: $\mathbf{d}_{k+1} = -\nabla f(\mathbf{x}_{k+1}) + \beta_k \mathbf{d}_k$

where $\beta_k = \frac{\|\nabla f(\mathbf{x}_{k+1})\|^2}{\|\nabla f(\mathbf{x}_k)\|^2}$ (Fletcher-Reeves formula)

The β_k term adds a fraction of the **previous direction** — this prevents the zigzagging.

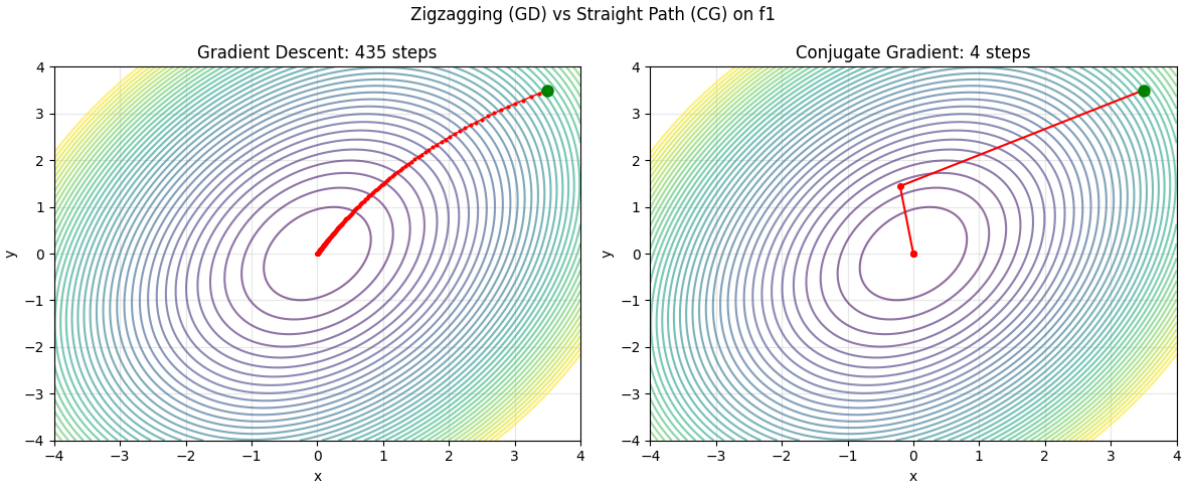
```
def line_search_golden(f_1d, a=-1, b=1, tol=1e-6):
    """1D golden section search for line search."""
    phi = (np.sqrt(5) - 1) / 2
    x1 = b - phi * (b - a)
    x2 = a + phi * (b - a)
    f1, f2_val = f_1d(x1), f_1d(x2)
    for _ in range(100):
        if (b - a) < tol:
            break
        if f1 < f2_val:
            b = x2
            x2, f2_val = x1, f1
            x1 = b - phi * (b - a)
            f1 = f_1d(x1)
        else:
            a = x1
            x1, f1 = x2, f2_val
            x2 = a + phi * (b - a)
            f2_val = f_1d(x2)
    return (a + b) / 2

def minimize_cg(f, x0, N=200, tol=1e-8):
    """Conjugate gradient with Fletcher-Reeves formula."""
    x_now = np.array(x0, dtype=float)
    grad = derivative2(f, x_now)
    d = -grad.copy()  # Initial direction = steepest descent
    x_hist = [x_now.copy()]

    for k in range(N):
        # Line search along direction d
        f_1d = lambda alpha, xn=x_now, dn=d: f(xn + alpha * dn)
        alpha = line_search_golden(f_1d, a=0, b=2.0)

        x_next = x_now + alpha * d
        x_hist.append(x_next.copy())

        if np.linalg.norm(x_next - x_now) < tol:
            break

        ## code .....
        grad_new = derivative2(f, x_next)
        # fprime= np.dot(grad,grad)
        # if  fprime < 1e-12:  fprime = 1e-12

        beta = np.dot(grad_new, grad_new) / max(np.dot(grad, grad), 1e-12)
```

(continues on next page)

(continued from previous page)

```
        d = -grad_new + beta * d
        grad = grad_new
        x_now = x_next

    return True, np.array(x_hist), f(x_now)
```

```
# Compare gradient descent vs conjugate gradient on f1
def minimize_fix(f, x0, dx=0.05, N=1000):
    x_now = np.array(x0, dtype=float)
    x_hist = [x_now.copy()]
    for i in range(N):
        dfx = derivative2(f, x_now)
        x_next = x_now - dx * dfx
        if np.linalg.norm(x_next - x_now) < 1e-6:
            break
        x_now = x_next
        x_hist.append(x_now.copy())
    return True, np.array(x_hist), f(x_now)

sp = [3.5, 3.5]
_, xh_gd, _ = minimize_fix(f1, sp)
_, xh_cg, _ = minimize_cg(f1, sp)

z1 = f1([x, y])
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

axes[0].contour(x, y, z1, levels=np.arange(0, 12, 0.3), cmap='viridis', alpha=0.6)
axes[0].plot(xh_gd[:, 0], xh_gd[:, 1], 'ro-', ms=2, lw=1)
axes[0].plot(sp[0], sp[1], 'go', ms=8)
axes[0].set_title(f'Gradient Descent: {len(xh_gd)-1} steps')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].grid(alpha=0.3)

axes[1].contour(x, y, z1, levels=np.arange(0, 12, 0.3), cmap='viridis', alpha=0.6)
axes[1].plot(xh_cg[:, 0], xh_cg[:, 1], 'ro-', ms=4, lw=1.5)
axes[1].plot(sp[0], sp[1], 'go', ms=8)
axes[1].set_title(f'Conjugate Gradient: {len(xh_cg)-1} steps')
axes[1].set_xlabel('x')
axes[1].set_ylabel('y')
axes[1].grid(alpha=0.3)

plt.suptitle('Zigzagging (GD) vs Straight Path (CG) on f1', fontsize=12)
plt.tight_layout()
plt.show()

print(f'Gradient descent: {len(xh_gd)-1} steps')
print(f'Conjugate gradient: {len(xh_cg)-1} steps')
```




```
Gradient descent: 435 steps
Conjugate gradient: 4 steps
```

16.4 III. BFGS: The Best of Both Worlds

16.4.1 Quasi-Newton Methods: Approximate the Hessian

Newton's method needs the exact Hessian ($O(n^2)$ second derivatives). **Quasi-Newton methods** build an approximation to the Hessian (or its inverse) using only gradient information — updating it step by step.

BFGS (Broyden–Fletcher–Goldfarb–Shanno) is the most successful quasi-Newton method:

1. Start with $\mathbf{B}_0 = \mathbf{I}$ (identity matrix — pretend the Hessian is the identity)
2. Compute search direction: $\mathbf{d}_k = -\mathbf{B}_k^{-1} \nabla f(\mathbf{x}_k)$
3. Line search: find step size α_k
4. Update position: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$
5. Update the Hessian approximation using the **secant condition**:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} - \frac{\mathbf{B}_k \mathbf{s}_k \mathbf{s}_k^T \mathbf{B}_k}{\mathbf{s}_k^T \mathbf{B}_k \mathbf{s}_k}$$

where $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$.

Key insight: BFGS "learns" the curvature from how the gradient changes — no second derivatives needed!

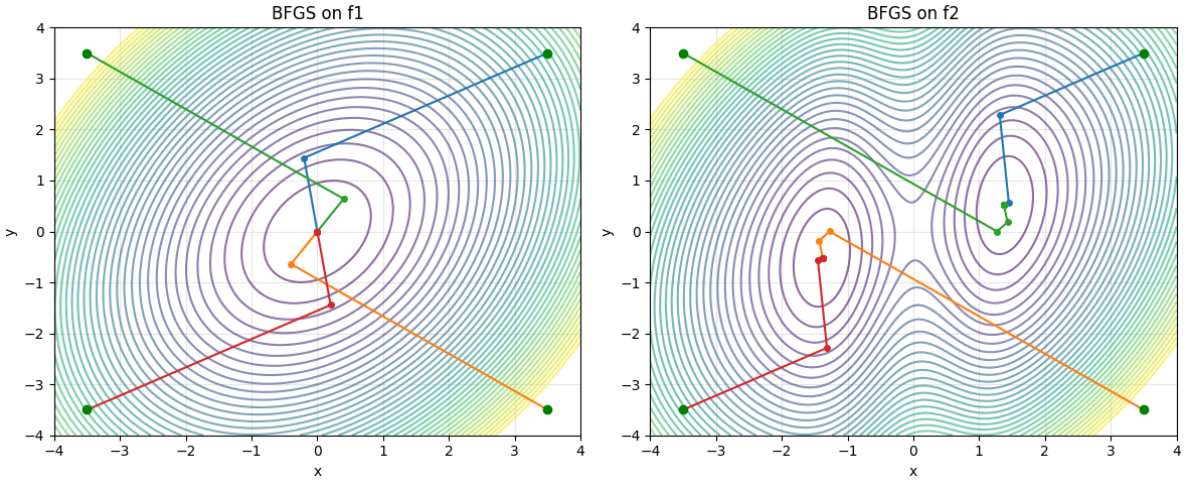
```
def minimize_bfgs(f, x0, N=200, tol=1e-8):
    """BFGS quasi-Newton method."""
    n = len(x0)
    x_now = np.array(x0, dtype=float)
    B = np.eye(n)  # Initial Hessian approximation = identity
    grad = derivative2(f, x_now)
    x_hist = [x_now.copy()]

    for k in range(N):
        # Search direction
        d = np.linalg.solve(B, -grad)
```

(continues on next page)

(continued from previous page)

```
        # Line search along d
        f_1d = lambda alpha, xn=x_now, dn=d: f(xn + alpha * dn)
        alpha = line_search_golden(f_1d, a=0, b=2.0)

        s = alpha * d  # Step
        x_next = x_now + s
        x_hist.append(x_next.copy())

        if np.linalg.norm(s) < tol:
            break

        grad_new = derivative2(f, x_next)
        y = grad_new - grad  # Gradient change

        # BFGS update
        sy = np.dot(s, y)
        if abs(sy) > 1e-15:  # Skip update if s and y are nearly orthogonal
            Bs = B @ s
            B = B + np.outer(y, y) / sy - np.outer(Bs, Bs) / np.dot(s, Bs)

        x_now = x_next
        grad = grad_new

    return True, np.array(x_hist), f(x_now)
```

```
# BFGS on f1 and f2
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

z1 = f1([x, y])
axes[0].contour(x, y, z1, levels=np.arange(0, 12, 0.3), cmap='viridis', alpha=0.6)
for sp in [[3.5, 3.5], [3.5, -3.5], [-3.5, 3.5], [-3.5, -3.5]]:
    conv, xh, fm = minimize_bfgs(f1, sp)
    axes[0].plot(xh[:, 0], xh[:, 1], 'o-', ms=4, lw=1.5)
    axes[0].plot(sp[0], sp[1], 'go', ms=6)
axes[0].set_title('BFGS on f1')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].grid(alpha=0.3)

z2 = f2([x, y])
levels2 = np.arange(np.min(z2), 12, 0.3)
axes[1].contour(x, y, z2, levels=levels2, cmap='viridis', alpha=0.6)
for sp in [[3.5, 3.5], [3.5, -3.5], [-3.5, 3.5], [-3.5, -3.5]]:
    conv, xh, fm = minimize_bfgs(f2, sp)
    axes[1].plot(xh[:, 0], xh[:, 1], 'o-', ms=4, lw=1.5)
    axes[1].plot(sp[0], sp[1], 'go', ms=6)
axes[1].set_title('BFGS on f2')
axes[1].set_xlabel('x')
axes[1].set_ylabel('y')
axes[1].grid(alpha=0.3)

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



16.5 IV. Method Showdown: Comparing All Approaches

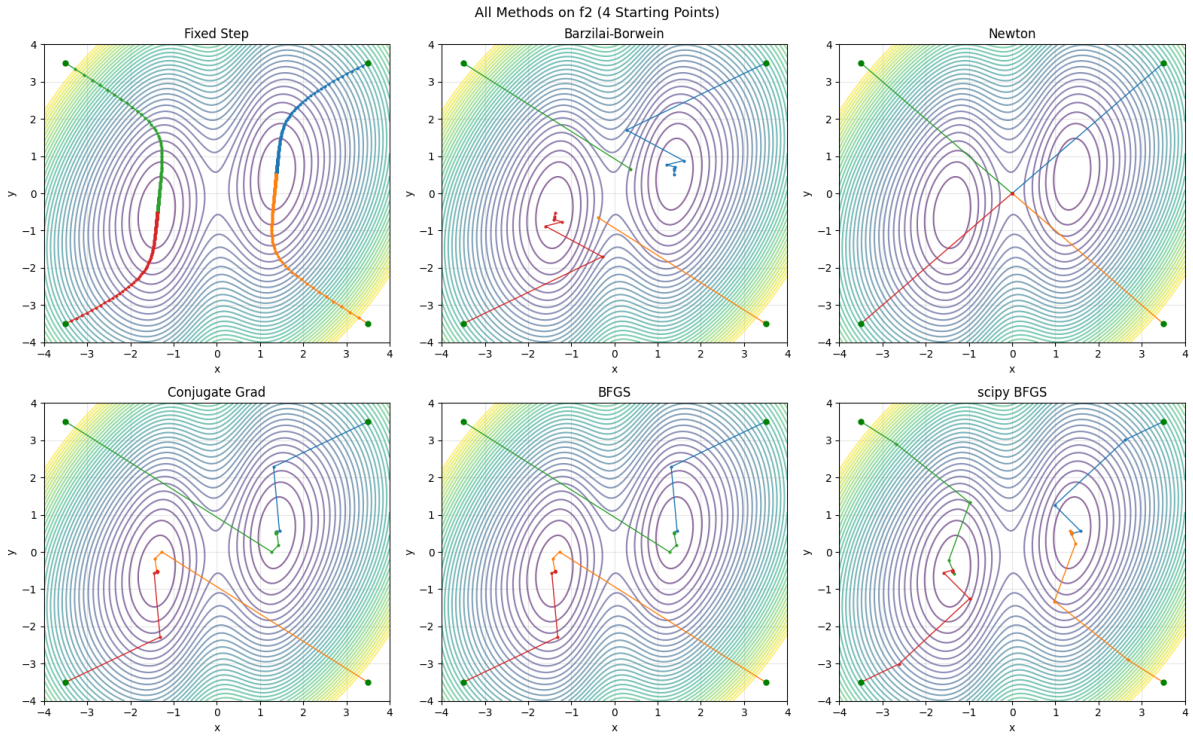
```
# Variable step (Barzilai-Borwein) from Lecture 15
def minimize_var(f, x0, N=1000):
    x_now = np.array(x0, dtype=float)
    x_prev = None
    converged = False
    x_hist = [x_now.copy()]
    for i in range(N):
        df_now = derivative2(f, x_now)
        if x_prev is None:
            dx = 0.01
        else:
            df_prev = derivative2(f, x_prev)
            dd = df_now - df_prev
            dx = np.dot((x_now - x_prev), dd) / (np.linalg.norm(dd))**2
        x_next = x_now - df_now * dx
        if f(x_next) < f(x_now):
            x_prev = x_now
            x_now = x_next
            x_hist.append(x_now.copy())
        else:
            converged = True
            break
    return converged, np.array(x_hist), f(x_now)
```

```
# Grand comparison: all methods on f2
start_points = [[3.5, 3.5], [3.5, -3.5], [-3.5, 3.5], [-3.5, -3.5]]
methods = [
    ('Fixed Step', minimize_fix),
    ('Barzilai-Borwein', minimize_var),
    ('Newton', minimize_newton),
    ('Conjugate Grad', minimize_cg),
    ('BFGS', minimize_bfgs),
]

fig, axes = plt.subplots(2, 3, figsize=(16, 10))
```

(continues on next page)

(continued from previous page)

```
axes_flat = axes.flatten()

z2 = f2([x, y])
levels2 = np.arange(np.min(z2), 12, 0.3)

for col, (name, method) in enumerate(methods):
    ax = axes_flat[col]
    ax.contour(x, y, z2, levels=levels2, cmap='viridis', alpha=0.6)
    for sp in start_points:
        conv, xh, fm = method(f2, sp)
        ax.plot(xh[:, 0], xh[:, 1], 'o-', ms=2, lw=1)
        ax.plot(sp[0], sp[1], 'go', ms=5)
    ax.set_title(name)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.grid(alpha=0.3)

# scipy BFGS in last panel
axes_flat[5].contour(x, y, z2, levels=levels2, cmap='viridis', alpha=0.6)
for sp in start_points:
    traj = [sp]
    sp_minimize(f2, sp, method='BFGS', callback=lambda xk, sp=sp: traj.append(xk.
 ↪tolist()), tol=1e-6)
    traj = np.array(traj)
    axes_flat[5].plot(traj[:, 0], traj[:, 1], 'o-', ms=2, lw=1)
    axes_flat[5].plot(sp[0], sp[1], 'go', ms=5)
axes_flat[5].set_title('scipy BFGS')
axes_flat[5].set_xlabel('x')
axes_flat[5].set_ylabel('y')
axes_flat[5].grid(alpha=0.3)

plt.suptitle('All Methods on f2 (4 Starting Points)', fontsize=13)
plt.tight_layout()
plt.show()
```




```
# Quantitative comparison
print('Method Comparison on f2 (two minima)')
print(f'{"Start":>14s} | {"Method":>16s} | {"Steps":>5s} | {"f_min":>8s} | {"Final (x,
 ↪ y)":>20s}')
print('-' * 76)

for sp in start_points:
    for name, method in methods:
        conv, xh, fm = method(f2, sp)
        end = xh[-1]
        print(f'({sp[0]:5.1f},{sp[1]:5.1f}) | {name:>16s} | {len(xh)-1:5d} | {fm:8.4f}
 ↪ | ({end[0]:8.4f}, {end[1]:8.4f})')
    # Also scipy BFGS
    res = sp_minimize(f2, sp, method='BFGS', tol=1e-6)
    print(f'({sp[0]:5.1f},{sp[1]:5.1f}) | {"scipy BFGS":>16s} | {res.nit:5d} | {res.
↪fun:8.4f} | ({res.x[0]:8.4f}, {res.x[1]:8.4f})')
    print()
```

```
Method Comparison on f2 (two minima)
         Start |           Method | Steps |    f_min |          Final (x, y)
------------------------------------------------------------------------------
(  3.5,  3.5) |       Fixed Step |   354 |   1.3096 | (  1.3748,   0.5156)
(  3.5,  3.5) | Barzilai-Borwein |     8 |   1.3096 | (  1.3762,   0.5162)
(  3.5,  3.5) |           Newton |     4 |   3.0000 | (  0.0000,   0.0000)
(  3.5,  3.5) |   Conjugate Grad |    13 |   1.3096 | (  1.3748,   0.5156)
(  3.5,  3.5) |             BFGS |     6 |   1.3096 | (  1.3748,   0.5156)
(  3.5,  3.5) |       scipy BFGS |     8 |   1.3096 | (  1.3748,   0.5156)

(  3.5, -3.5) |       Fixed Step |   357 |   1.3096 | (  1.3748,   0.5155)
(  3.5, -3.5) | Barzilai-Borwein |     2 |   2.7599 | ( -0.3739,  -0.6540)
(  3.5, -3.5) |           Newton |     4 |   3.0000 | (  0.0000,  -0.0000)
```

(continues on next page)

(continued from previous page)

```
(  3.5, -3.5) |   Conjugate Grad |    17 |   1.3096 | ( -1.3748,  -0.5156)
(  3.5, -3.5) |             BFGS |     7 |   1.3096 | ( -1.3748,  -0.5156)
(  3.5, -3.5) |       scipy BFGS |     7 |   1.3096 | (  1.3748,   0.5156)

( -3.5,  3.5) |       Fixed Step |   357 |   1.3096 | ( -1.3748,  -0.5155)
( -3.5,  3.5) | Barzilai-Borwein |     2 |   2.7599 | (  0.3739,   0.6540)
( -3.5,  3.5) |           Newton |     4 |   3.0000 | ( -0.0000,   0.0000)
( -3.5,  3.5) |   Conjugate Grad |    17 |   1.3096 | (  1.3748,   0.5156)
( -3.5,  3.5) |             BFGS |     7 |   1.3096 | (  1.3748,   0.5156)
( -3.5,  3.5) |       scipy BFGS |     7 |   1.3096 | ( -1.3748,  -0.5156)

( -3.5, -3.5) |       Fixed Step |   354 |   1.3096 | ( -1.3748,  -0.5156)
( -3.5, -3.5) | Barzilai-Borwein |     8 |   1.3096 | ( -1.3762,  -0.5162)
( -3.5, -3.5) |           Newton |     4 |   3.0000 | ( -0.0000,  -0.0000)
( -3.5, -3.5) |   Conjugate Grad |    13 |   1.3096 | ( -1.3748,  -0.5156)
( -3.5, -3.5) |             BFGS |     6 |   1.3096 | ( -1.3748,  -0.5156)
( -3.5, -3.5) |       scipy BFGS |     8 |   1.3096 | ( -1.3748,  -0.5156)
```

16.6 V. Constrained Optimization

16.6.1 Optimization with Constraints

Many physics problems have **constraints**:

- Find the shape of a hanging chain (minimize potential energy, subject to fixed length)
- Find the ground state energy (minimize $\langle H \rangle$, subject to $\langle \psi | \psi \rangle = 1$)
- Find the equilibrium configuration (minimize free energy, subject to fixed particle number)

Mathematically: minimize $f(\mathbf{x})$ subject to $g(\mathbf{x}) = 0$

16.6.2 Lagrange Multipliers

The classical approach: at a constrained minimum, the gradient of f must be parallel to the gradient of g:

$$\nabla f = \lambda \nabla g$$

This gives us the **Lagrangian**: $\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda\, g(\mathbf{x})$

Setting $\nabla_{\mathbf{x}} \mathcal{L} = 0$ and $\partial \mathcal{L} / \partial \lambda = 0$ gives a system of equations.

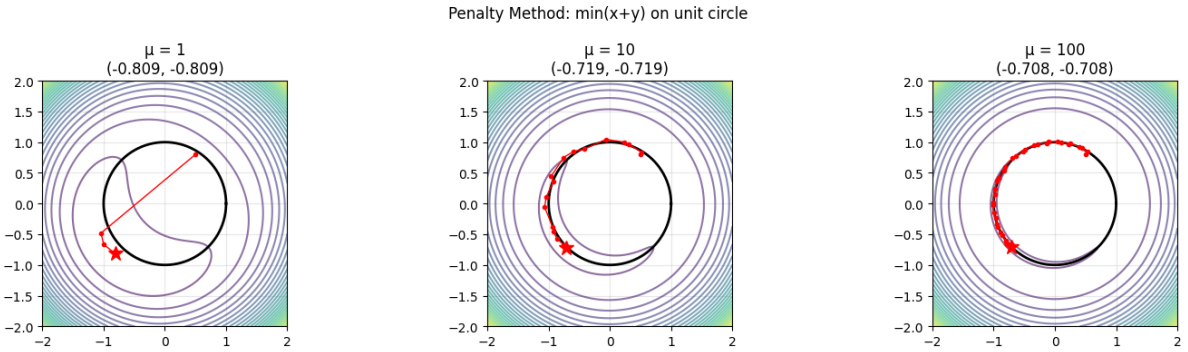
16.6.3 Penalty Method

A simpler computational approach: convert the constrained problem to an unconstrained one by adding a **penalty** for violating the constraint:

$$\tilde{f}(\mathbf{x}) = f(\mathbf{x}) + \mu\, [g(\mathbf{x})]^2$$

As $\mu \to \infty$, the minimum of $\tilde{f}$ approaches the constrained minimum of f.

```
# Example: minimize f(x,y) = x + y  subject to  x^2 + y^2 = 1
# Find the point on the unit circle closest to the direction (-1,-1)
# Analytical answer: x* = y* = -1/sqrt(2)

def f_obj(xy):
    return xy[0] + xy[1]

def g_constraint(xy):
    return xy[0]**2 + xy[1]**2 - 1

# Penalty method: minimize f + mu * g^2
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

penalties = [1, 10, 100]
theta = np.linspace(0, 2*np.pi, 200)

for idx, mu in enumerate(penalties):
    def f_penalty(xy, mu=mu):
        return f_obj(xy) + mu * g_constraint(xy)**2

    conv, xh, fm = minimize_bfgs(f_penalty, [0.5, 0.8])

    xg = np.linspace(-2, 2, 200)
    xg, yg = np.meshgrid(xg, xg)
    zp = f_obj([xg, yg]) + mu * (xg**2 + yg**2 - 1)**2
    axes[idx].contour(xg, yg, zp, levels=30, cmap='viridis', alpha=0.6)
    axes[idx].plot(np.cos(theta), np.sin(theta), 'k-', lw=2, label='Constraint')
    axes[idx].plot(xh[:, 0], xh[:, 1], 'ro-', ms=3, lw=1)
    axes[idx].plot(xh[-1, 0], xh[-1, 1], 'r*', ms=12)
    axes[idx].set_title(f'\u03bc = {mu}\n({xh[-1,0]:.3f}, {xh[-1,1]:.3f})')
    axes[idx].set_xlim(-2, 2)
    axes[idx].set_ylim(-2, 2)
    axes[idx].set_aspect('equal')
    axes[idx].grid(alpha=0.3)

plt.suptitle('Penalty Method: min(x+y) on unit circle', fontsize=12)
plt.tight_layout()
plt.show()

x_exact = -1/np.sqrt(2)
print(f'Exact solution: ({x_exact:.6f}, {x_exact:.6f})')
print(f'As mu increases, the penalty solution approaches the exact answer.')
```




```
Exact solution: (-0.707107, -0.707107)
```

(continues on next page)

(continued from previous page)

```
As mu increases, the penalty solution approaches the exact answer.
```

```
# scipy with explicit constraint
constraint = {'type': 'eq', 'fun': g_constraint}

res = sp_minimize(f_obj, [0.5, 0.8], method='SLSQP', constraints=constraint)
print('scipy SLSQP (constrained optimization):')
print(f'  Solution: ({res.x[0]:.8f}, {res.x[1]:.8f})')
print(f'  f_min = {res.fun:.8f}')
print(f'  Constraint satisfied: g = {g_constraint(res.x):.2e}')
print(f'  Exact: ({-1/np.sqrt(2):.8f}, {-1/np.sqrt(2):.8f})')
```

```
scipy SLSQP (constrained optimization):
  Solution: (-0.70710678, -0.70710678)
  f_min = -1.41421356
  Constraint satisfied: g = 5.06e-10
  Exact: (-0.70710678, -0.70710678)
```

16.7 VI. Summary

16.7.1 Method Comparison

Method	Convergence	Cost per step	Needs	Best for
Gradient descent	Linear	$O(n)$	Gradient	Simple problems, large n
Conjugate gradient	Superlinear	$O(n)$	Gradient + line search	Medium n, quadratic-ish
Newton	**Quadratic**	$O(n^3)$	Gradient + Hessian	Small n, exact Hessian available
BFGS	Superlinear	$O(n^2)$	Gradient only	General purpose, moderate n

16.7.2 Key Takeaways

1. **Curvature information dramatically accelerates convergence** — Newton converges quadratically vs gradient descent's linear convergence
2. **BFGS is the practical workhorse** — it approximates curvature from gradient changes, no Hessian needed
3. **Conjugate gradient avoids zigzagging** by choosing **H**-orthogonal search directions
4. **Constrained optimization** can use penalty methods (simple) or dedicated algorithms (SLSQP, trust-constr)
5. **All local methods find the nearest minimum** — for global optimization, combine with simulated annealing (Lecture 14)

16.7.3 When to Use What (Practical Guide)

```
Small problem (n < 100):      scipy BFGS or Newton-CG
Medium problem (n ~ 1000):    scipy L-BFGS-B (memory-limited BFGS)
Large problem (n > 10000):    scipy CG or custom gradient descent
No derivatives available:     scipy Nelder-Mead or Powell
With constraints:             scipy SLSQP or trust-constr
```

CHAPTER

SEVENTEEN

LECTURE 17: GLOBAL OPTIMIZATION I — SIMULATED ANNEALING & BASIN HOPPING

PHYS 611 (Computational Physics) | Rutgers-Newark

17.1 Roadmap for L17–L19

- **L17 (this lecture)**: Simulated Annealing, Basin Hopping, LJ clusters
- **L18**: Evolutionary Algorithms (Genetic Algorithms, Differential Evolution)
- **L19**: Particle Swarm Optimization, hybrid methods

By the end of L17, you will:

1. Understand why local optimization fails for multi-modal problems
2. Implement Simulated Annealing with cooling schedules
3. Understand Basin Hopping and when to use it
4. Apply these to finding global minima of Lennard-Jones clusters

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize, basinhopping
from mpl_toolkits.mplot3d import Axes3D

# Set random seed for reproducibility
np.random.seed(42)
```

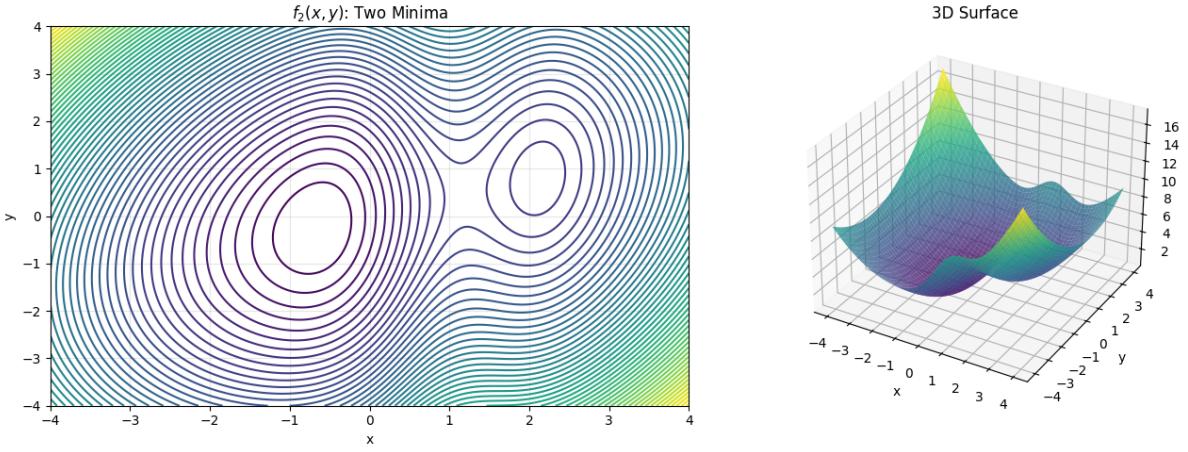
17.2 Quick Review: Why Local Methods Fail

Recall from **L15–16**: Gradient-based methods (BFGS, CG, Gradient Descent) follow the steepest descent. They get trapped in **local minima** and cannot escape.

Let's see this in action with the 2D function from L15 that has **two minima**:

$$f_2(x, y) = \frac{x^2}{2} + \frac{y^2}{3} - \frac{xy}{4} + 3e^{-(x-1)^2}$$

```python
def f2(x):
    return x[0]**2/2 + x[1]**2/3 - x[0]*x[1]/4 + 3 * np.exp(-(x[0]-1)**2)

x_min, x_max = -4, 4
y_min, y_max = -4, 4
nx = np.linspace(x_min, x_max, 400)
ny = np.linspace(y_min, y_max, 400)
x, y = np.meshgrid(nx, ny)
z = f2([x, y])

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Contour plot
levels = np.arange(np.min(z), np.max(z), 0.3)
axes[0].contour(x, y, z, levels=levels, cmap='viridis')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].set_title('$f_2(x,y)$: Two Minima')
axes[0].grid(alpha=0.3)

# 3D surface
axes[1].remove()
ax3 = fig.add_subplot(122, projection='3d')
ax3.plot_surface(x, y, z, cmap='viridis', alpha=0.8)
ax3.set_xlabel('x')
ax3.set_ylabel('y')
ax3.set_zlabel('f(x,y)')
ax3.set_title('3D Surface')

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



The problem will become more frequent if we are dealing with a function with multiple variables. We will alway get stuck in a local minima. It turns out the previous optimization techniques do not give us the reliable solution. What shall we do?

17.3 Brute force methods

Since the gradient based methods is no longer reliable for the case of multiple minima. We shall return to the more strightforward method.

From the first look, one might immediately come up with two simple ideas:

- *Grid based search*, set a very fine resolution in x-y space, and then evalute the function on each grid
- *Random sampling*, simply generate a lot of (x,y) points, and take the minimum values from all attempts

```
# Grid search
def grid_search(N):
    x_min, x_max = -4, 4
    y_min, y_max = -4, 4
    minf = f2([x_min,y_min])
    #---to complete-----#
    grid_x = np.linspace(x_min, x_max, N)
    grid_y = np.linspace(y_min, y_max, N)

    f_values = []
    for x in grid_x:
        for y in grid_y:
            f_values.append(f2([x,y]))

    minf = min(f_values)
    #---to complete-----#
    return minf

print (grid_search(20))
```

```
0.3903937160062407
```

```
# Random search
def random_search(N):
    x_min, x_max = -4, 4
    y_min, y_max = -4, 4
    minf = f2([x_min,y_min])
    #---to complete-----#
    grid_x = np.random.uniform(x_min, x_max, N)
    grid_y = np.random.uniform(y_min, y_max, N)

    f_values = []
    for x in grid_x:
        for y in grid_y:
            f_values.append(f2([x,y]))

    minf = min(f_values)

    #---to complete-----#
    return minf

random_search(100)
```

```
np.float64(0.40101879239679317)
```

17.4 Mixed stratergy

From the above run, one might find that a more effective way as follow,

- 1, randomly select a point
- 2, perform a local optimization on each point

This methods would simply take the advantages of both sides, will outperform than all previous methods. In fact, this idea has been largely used nowadays in many fields.

```
# Mixed Random search

def random_search2(N):
    x_min, x_max = -4, 4
    y_min, y_max = -4, 4
    minf = f2([x_min,y_min])
    #---to complete-----#
    grid_x = np.random.uniform(x_min, x_max, N)
    grid_y = np.random.uniform(y_min, y_max, N)

    f_values = []
    for x in grid_x:
        for y in grid_y:
            res = minimize(f2, [x,y], method='bfgs', tol=1e-4, options={'disp': False}
 ↪)
            f_values.append(res['fun'])

    minf = min(f_values)

    #---to complete-----#
    return minf

for _ in range(10):
    print (random_search2(5))
```

```
0.3878588684766304
0.38785886847310447
0.387858868473117
0.38785886847340567
0.3878588684731093
0.38785886847317597
0.3878588684731554
0.3878588684731402
0.38785886847312206
0.3878588684731119
```

17.5 The Exponential Growth Problem

In high dimensions, the number of local minima grows **exponentially**.

For example:

- 1D: 3–5 local minima
- 3D: dozens to hundreds
- 100D (typical in molecular optimization): astronomically many

Conclusion: We need methods that can **escape local minima**.

17.6 Lennard-Jones Clusters: Our Running Example

A **Lennard-Jones cluster** is a set of N atoms arranged in 3D space to minimize the total interaction energy.

- **Relevance**: Ubiquitous in molecular dynamics, cluster chemistry, materials science
- **Challenge**: Finding the global minimum is NP-hard; known only for small N (N ≤ ~20)
- **Beauty**: Simple energy function, but rich optimization landscape

We'll use LJ7 and LJ13 throughout L17–L19.

17.7 Section II: Lennard-Jones Clusters

17.8 Lennard-Jones Potential

The pairwise LJ potential between two atoms at distance r is:

$$V_{LJ}(r) = 4\left(\frac{1}{r^{12}} - \frac{1}{r^6}\right)$$

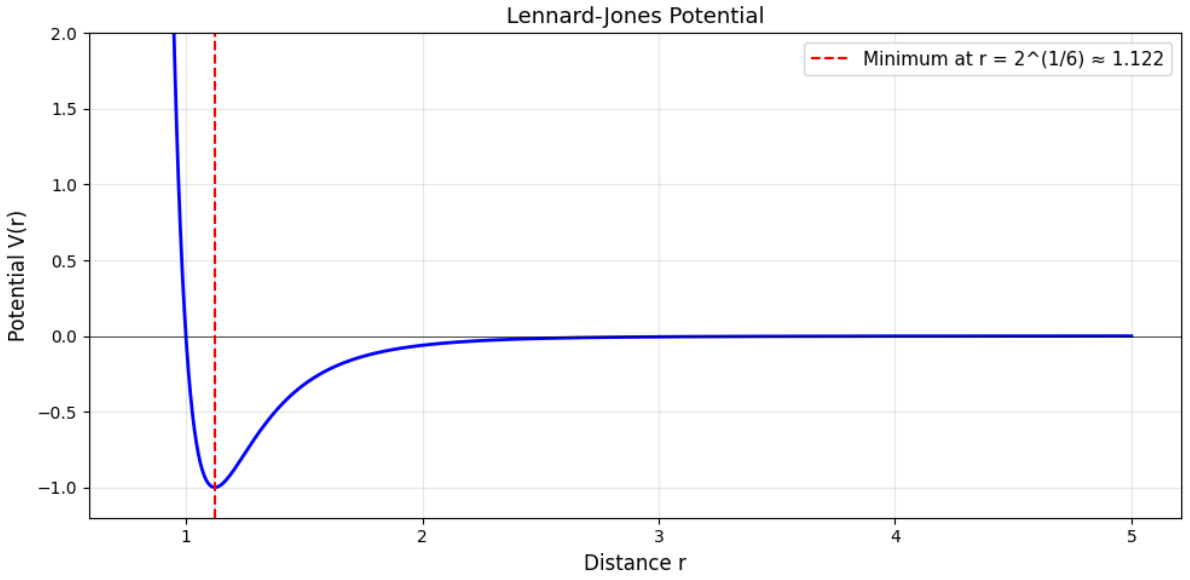
- r^{-12} **term**: Hard-core repulsion (atoms can't overlap)
- r^{-6} **term**: Van der Waals attraction
- **Minimum**: at $r = 2^{1/6} \approx 1.122$, where $V = -1$

```
def LJ(r):
    'Lennard-Jones potential between two atoms at distance r'

    return 4*(1/r**12 - 1/r**6)

# Plot LJ potential
r = np.linspace(0.8, 5, 500)
V = LJ(r)

plt.figure(figsize=(10, 5))
plt.plot(r, V, 'b-', linewidth=2)
plt.axhline(0, color='k', linestyle='-', linewidth=0.5)
plt.axvline(2**(1/6), color='r', linestyle='--', linewidth=1.5, label=f'Minimum at r␣
 ↪= 2^(1/6) ≈ {2**(1/6):.3f}')
plt.xlabel('Distance r', fontsize=12)
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Potential V(r)', fontsize=12)
plt.title('Lennard-Jones Potential', fontsize=13)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=11)
plt.ylim([-1.2, 2])
plt.tight_layout()
plt.show()
```




```
def total_energy(positions):
    '''Total LJ energy for a cluster
    positions: 1D array [x1, y1, z1, x2, y2, z2, ...]'''
    E = 0
    # code
    N_atom = int(len(positions) / 3)
    for i in range(N_atom-1):
        for j in range(i+1, N_atom):
            pos1 = positions[i*3:(i+1)*3]
            pos2 = positions[j*3:(j+1)*3]
            dist = np.linalg.norm(pos1-pos2)
            E += LJ(dist)
    return E

def init_pos(N, L=5):
    'Initialize random positions in a box of side length L'
    return L * np.random.random_sample((N * 3,))
```

```
# Demo: random LJ7 cluster
N_atom = 6
pos_random = init_pos(N_atom)
E_random = total_energy(pos_random)

print(f'LJ{N_atom} Cluster:')
print(f'Number of atoms: {N_atom}')
print(f'Number of coordinates: {len(pos_random)}')
print(f'Number of pairwise interactions: {N_atom*(N_atom-1)//2}')
```

(continues on next page)

(continued from previous page)

```
print(f'Energy of random configuration: {E_random:.6f}')
```

```
LJ6 Cluster:
Number of atoms: 6
Number of coordinates: 18
Number of pairwise interactions: 15
Energy of random configuration: 76240.793652
```

```
# Local optimization with CG
res_cg = minimize(total_energy, pos_random, method='CG', tol=1e-4)
E_cg = res_cg.fun
pos_cg = res_cg.x

print(f'\nAfter CG local optimization:')
print(f'Energy: {E_cg:.6f}')
print(f'Improvement: {E_random - E_cg:.6f}')
```

```
After CG local optimization:
Energy: -12.302928
Improvement: 76253.096579
```

```
# Multiple random-start local optimization (20 tries)
N_trials = 30
energies = []

for trial in range(N_trials):
    pos_init = init_pos(N_atom)
    res = minimize(total_energy, pos_init, method='CG', tol=1e-4)
    energies.append(res.fun)

energies = np.array(energies)

print(f'Results from {N_trials} random-start CG optimizations:')
print(f'Best energy found: {np.min(energies):.6f}')
print(f'Worst energy found: {np.max(energies):.6f}')
print(f'Mean energy: {np.mean(energies):.6f}')
print(f'Std dev: {np.std(energies):.6f}')
print(f'Known global minimum (Cambridge DB): -16.505384')
```

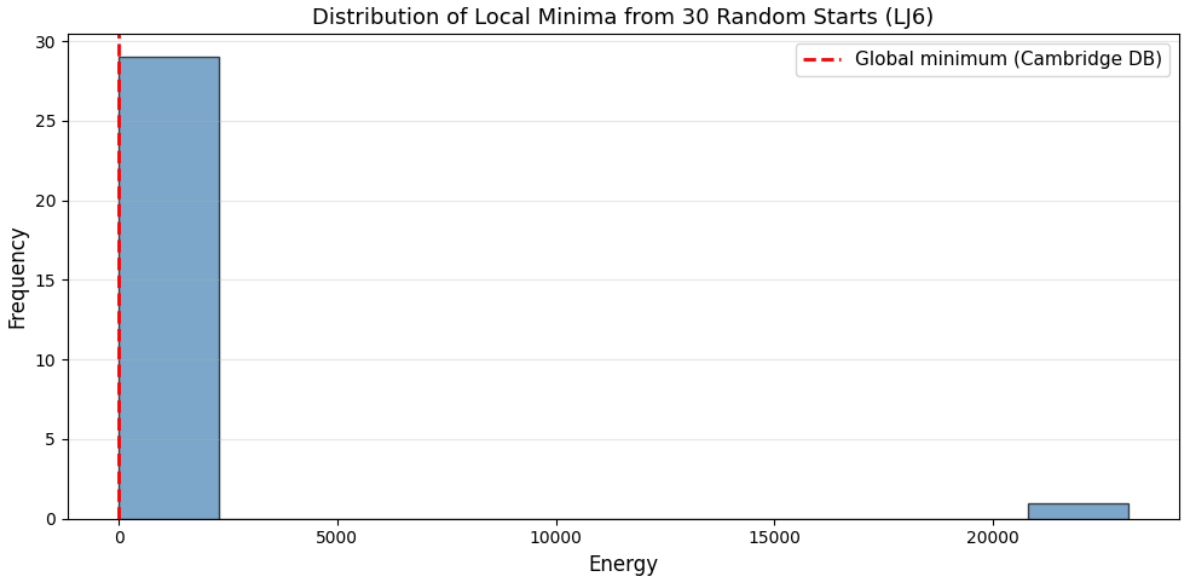
```
Results from 30 random-start CG optimizations:
Best energy found: -12.712062
Worst energy found: 23097.717242
Mean energy: 758.327537
Std dev: 4148.320863
Known global minimum (Cambridge DB): -16.505384
```

```
# Histogram of local minima energies
plt.figure(figsize=(10, 5))
plt.hist(energies, bins=10, edgecolor='black', alpha=0.7, color='steelblue')
plt.axvline(-16.505384, color='r', linestyle='--', linewidth=2, label='Global minimum␣
↪(Cambridge DB)')
plt.xlabel('Energy', fontsize=12)
plt.ylabel('Frequency', fontsize=12)
plt.title(f'Distribution of Local Minima from {N_trials} Random Starts (LJ{N_atom})',␣
```

(continues on next page)

(continued from previous page)

```
 ↪fontsize=13)
plt.legend(fontsize=11)
plt.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.show()
```




```
#below are some reference values from Cambridge Cluster database,
#https://doye.chem.ox.ac.uk/jon/structures/LJ/tables.150.html
#please try to download some values from there and check if the results are consistent

#pos =np.array([  -0.3616353090,        0.0439914505,        0.5828840628,
#                  0.2505889242,        0.6193583398,       -0.1614607010,
#                 -0.4082757926,       -0.2212115329,       -0.5067996704,
#                  0.5193221773,       -0.4421382574,        0.085376308701])
pos =np.array([ -0.2604720088,        0.7363147287,        0.4727061929,
                 0.2604716550,       -0.7363150782,       -0.4727063011,
                -0.4144908003,       -0.3652598516,        0.3405559620,
                -0.1944131041,        0.2843471802,       -0.5500413671,
                 0.6089042582,        0.0809130209,        0.2094855133])

# pos = np.array([ 0.7430002202,        0.2647603899,       -0.0468575389,
#                 -0.7430002647,       -0.2647604843,        0.0468569750,
#                  0.1977276118,       -0.4447220146,        0.6224700350,
#                 -0.1977281310,        0.4447221826,       -0.6224697723,
#                 -0.1822009635,        0.5970484122,        0.4844363476,
#                  0.1822015272,       -0.5970484858,       -0.4844360463])
print (total_energy(pos))
```

```
-9.103852415681365
```

17.8.1 Visulization of the LJ clusters

- ASE package is required: the installation can be done via 'pip install ase' command

```
!pip install ase
```

```
Requirement already satisfied: ase in /usr/local/lib/python3.12/dist-packages (3.
↪28.0)
Requirement already satisfied: numpy>=1.21.6 in /usr/local/lib/python3.12/dist-
↪packages (from ase) (2.0.2)
Requirement already satisfied: scipy>=1.8.1 in /usr/local/lib/python3.12/dist-
↪packages (from ase) (1.16.3)
Requirement already satisfied: matplotlib>=3.5.2 in /usr/local/lib/python3.12/dist-
↪packages (from ase) (3.10.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-
↪packages (from matplotlib>=3.5.2->ase) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-
↪packages (from matplotlib>=3.5.2->ase) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-
↪packages (from matplotlib>=3.5.2->ase) (4.62.1)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-
↪packages (from matplotlib>=3.5.2->ase) (1.5.0)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-
↪packages (from matplotlib>=3.5.2->ase) (26.0)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-
↪packages (from matplotlib>=3.5.2->ase) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-
↪packages (from matplotlib>=3.5.2->ase) (3.3.2)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/
↪dist-packages (from matplotlib>=3.5.2->ase) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages␣
↪(from python-dateutil>=2.7->matplotlib>=3.5.2->ase) (1.17.0)
```

```
#Visulization of the LJ clusters
#ASE package is required: the installation can be done via 'pip install ase' command
from ase.visualize import view
from ase import Atoms
pos =np.array([ -0.2604720088,        0.7363147287,        0.4727061929,
                 0.2604716550,       -0.7363150782,       -0.4727063011,
                -0.4144908003,       -0.3652598516,        0.3405559620,
                -0.1944131041,        0.2843471802,       -0.5500413671,
                 0.6089042582,        0.0809130209,        0.2094855133])

N=5
cluster = Atoms('N'+str(N), positions=np.reshape(pos*2.0,[N,3]))

view(cluster, viewer='x3d') #view it from jupyter notebook
# view(cluster, viewer='ase') #view it from pop-up ase visualizer
```

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

```
N=12
pos = init_pos(N,L=5)
cluster = Atoms('N'+str(N), positions=np.reshape(pos*2.0,[N,3]))

view(cluster, viewer='x3d') #view it from jupyter notebook
```

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

17.9 Section III: Simulated Annealing on LJ Clusters

17.10 Simulated Annealing (SA): The Idea

Inspired by: Annealing in metallurgy (heating and slow cooling to reach low-energy states).

Key insight: At temperature T, accept a move with probability:

$$P(\text{accept}) = \begin{cases} 1 & \text{if } \Delta E < 0 \\ e^{-\Delta E/(k_B T)} & \text{if } \Delta E \geq 0 \end{cases}$$

where:

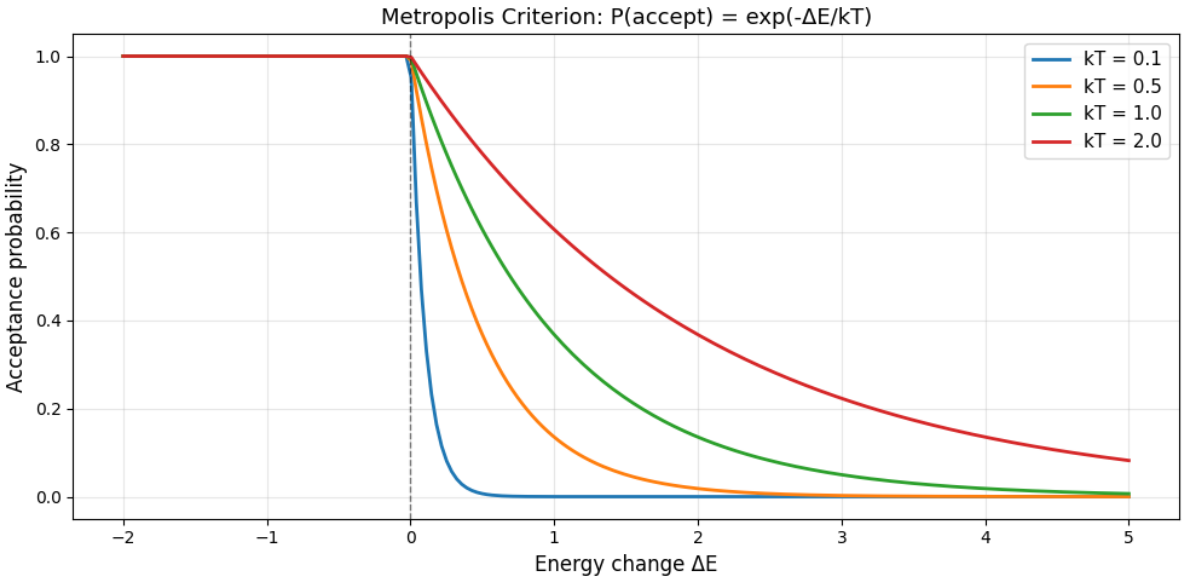
- $\Delta E = E_{\text{new}} - E_{\text{old}}$ (energy change)
- $k_B T$ = temperature-like parameter controlling exploration

Strategy: Start hot (accept bad moves), cool down (become selective).

17.11 Metropolis Acceptance Probability

This is the **Metropolis criterion** (same as HW6!).

```
# Plot Metropolis acceptance probability for various temperatures
dE_range = np.linspace(-2, 5, 200)
temperatures = [0.1, 0.5, 1.0, 2.0]

plt.figure(figsize=(10, 5))
for T in temperatures:
    prob = np.exp(-np.maximum(0, dE_range) / T)
    plt.plot(dE_range, prob, linewidth=2, label=f'kT = {T}')

plt.axvline(0, color='k', linestyle='--', linewidth=1, alpha=0.5)
plt.xlabel('Energy change ΔE', fontsize=12)
plt.ylabel('Acceptance probability', fontsize=12)
plt.title('Metropolis Criterion: P(accept) = exp(-ΔE/kT)', fontsize=13)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=11)
plt.tight_layout()
plt.show()
```



17.12 SA v1: Basic Simulated Annealing (Fixed Temperature)

First, let's implement the simplest version with constant temperature.

```
def simulated_annealing_v1(N_atom, Max_iteration=1000, kT=1.0):
    'Simulated Annealing (fixed temperature)'
    pos_now = init_pos(N_atom)
    obj_now = total_energy(pos_now)

    best_pos = pos_now.copy()
    best_eng = obj_now

    eng_hist = [obj_now]

    for i in range(Max_iteration):
        # Random step
        step = (np.random.random(len(pos_now)) - 0.5) * 2  # step in [-1, 1]
        pos_new = pos_now + step
        obj_new = total_energy(pos_new)

        # Metropolis acceptance
        dE = obj_new - obj_now
        if dE < 0 or np.random.random() < np.exp(-dE / kT):
            pos_now = pos_new
            obj_now = obj_new

        # Track best
        if obj_now < best_eng:
            best_eng = obj_now
            best_pos = pos_now.copy()

        eng_hist.append(obj_now)

    return best_pos, best_eng, eng_hist
```

```
# Test SA v1 on LJ7
np.random.seed(42)
best_pos_v1, best_eng_v1, eng_hist_v1 = simulated_annealing_v1(7, Max_iteration=5000,␣
↪kT=1.0)

print(f'SA v1 on LJ7 (fixed T=1.0):')
print(f'Best energy: {best_eng_v1:.6f}')
print(f'Known global minimum: -16.505384')
print(f'Gap: {best_eng_v1 - (-16.505384):.6f}')
```

```
SA v1 on LJ7 (fixed T=1.0):
Best energy: -1.675530
Known global minimum: -16.505384
Gap: 14.829854
```

17.13 SA v2: SA + Local Optimization (Basin Hopping Style)

At each step, perform a random walk followed by local optimization. This is the bridge to Basin Hopping.

```
def simulated_annealing_v2(N_atom, Max_iteration=100, kT=1.0, step_size=0.5):
    'SA + local optimization at each step (basin hopping style)'
    pos_now = init_pos(N_atom)
    res = minimize(total_energy, pos_now, method='CG', tol=1e-4)
    pos_now = res.x
    obj_now = res.fun

    best_pos = pos_now.copy()
    best_eng = obj_now
    eng_hist = [obj_now]

    for i in range(Max_iteration):
        # Random perturbation
        pos_new = pos_now + step_size * (np.random.random(len(pos_now)) - 0.5)

        # Local optimization
        res = minimize(total_energy, pos_new, method='CG', tol=1e-4)
        pos_new = res.x
        obj_new = res.fun

        # Metropolis acceptance
        dE = obj_new - obj_now
        if dE < 0 or np.random.random() < np.exp(-dE / kT):
            pos_now = pos_new
            obj_now = obj_new

        # Track best
        if obj_now < best_eng:
            best_eng = obj_now
            best_pos = pos_now.copy()

        eng_hist.append(obj_now)

    return best_pos, best_eng, eng_hist
```

```
# Test SA v2 on LJ7
np.random.seed(42)
best_pos_v2, best_eng_v2, eng_hist_v2 = simulated_annealing_v2(9, Max_iteration=100,␣
↪kT=1.0, step_size=0.5)

print(f'SA v2 on LJ7 (with local optimization):')
print(f'Best energy: {best_eng_v2:.6f}')
print(f'Known global minimum: -16.505384')
print(f'Gap: {best_eng_v2 - (-16.505384):.6f}')
```

```
SA v2 on LJ7 (with local optimization):
Best energy: -24.113360
Known global minimum: -16.505384
Gap: -7.607976
```

17.14 SA v3: SA with Geometric Cooling Schedule

The key to SA is the **cooling schedule**: how to reduce temperature over time.

17.14.1 Geometric cooling

$$T_n = T_0 \cdot \alpha^n$$

where:

- T_0 = initial temperature
- α = cooling rate (e.g., 0.95)
- n = iteration number

This exponentially decays the temperature, allowing early exploration and late exploitation.

```
def simulated_annealing_v3(N_atom, Max_iteration=200, T0=2.0, alpha=0.95, step_size=0.
↪5):
    'SA with geometric cooling schedule'
    pos_now = init_pos(N_atom)
    res = minimize(total_energy, pos_now, method='CG', tol=1e-4)
    pos_now = res.x
    obj_now = res.fun

    best_pos = pos_now.copy()
    best_eng = obj_now
    eng_hist = [obj_now]
    T_hist = [T0]

    for i in range(Max_iteration):
        T = T0 * (alpha ** i)

        # Random perturbation
        pos_new = pos_now + step_size * (np.random.random(len(pos_now)) - 0.5)

        # Local optimization
        res = minimize(total_energy, pos_new, method='CG', tol=1e-4)
```

(continues on next page)

(continued from previous page)

```
        pos_new = res.x
        obj_new = res.fun

        # Metropolis acceptance
        dE = obj_new - obj_now
        if dE < 0 or np.random.random() < np.exp(-dE / T):
            pos_now = pos_new
            obj_now = obj_new

        # Track best
        if obj_now < best_eng:
            best_eng = obj_now
            best_pos = pos_now.copy()

        eng_hist.append(obj_now)
        T_hist.append(T)

    return best_pos, best_eng, eng_hist, T_hist
```

```
# Test SA v3 on LJ7
np.random.seed(42)
best_pos_v3, best_eng_v3, eng_hist_v3, T_hist_v3 = simulated_annealing_v3(
    7, Max_iteration=200, T0=2.0, alpha=0.95, step_size=0.5
)

print(f'SA v3 on LJ7 (geometric cooling, T0=2.0, alpha=0.95):')
print(f'Best energy: {best_eng_v3:.6f}')
print(f'Known global minimum: -16.505384')
print(f'Gap: {best_eng_v3 - (-16.505384):.6f}')
```

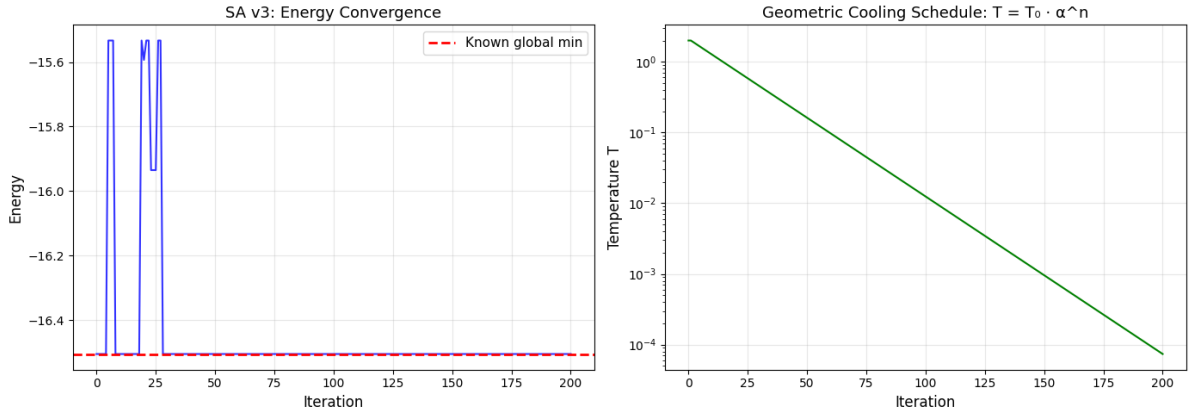
```
SA v3 on LJ7 (geometric cooling, T0=2.0, alpha=0.95):
Best energy: -16.505384
Known global minimum: -16.505384
Gap: -0.000000
```

```
# Plot energy and temperature history
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# Energy vs iteration
ax1.plot(eng_hist_v3, 'b-', linewidth=1.5, alpha=0.8)
ax1.axhline(-16.505384, color='r', linestyle='--', linewidth=2, label='Known global␣
↪min')
ax1.set_xlabel('Iteration', fontsize=12)
ax1.set_ylabel('Energy', fontsize=12)
ax1.set_title('SA v3: Energy Convergence', fontsize=13)
ax1.grid(True, alpha=0.3)
ax1.legend(fontsize=11)

# Temperature vs iteration
ax2.plot(T_hist_v3, 'g-', linewidth=1.5)
ax2.set_xlabel('Iteration', fontsize=12)
ax2.set_ylabel('Temperature T', fontsize=12)
ax2.set_title('Geometric Cooling Schedule: T = T₀ · α^n', fontsize=13)
ax2.grid(True, alpha=0.3)
ax2.set_yscale('log')
```

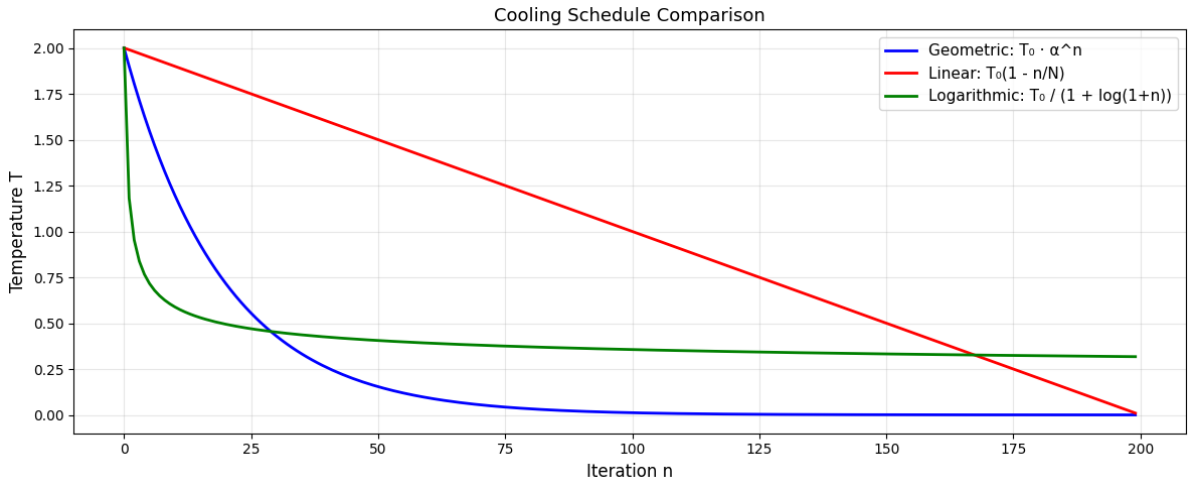
(continues on next page)

(continued from previous page)

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




```
# Compare three cooling schedules
n_iter = 200
T0 = 2.0
n = np.arange(n_iter)

# Geometric
T_geom = T0 * (0.95 ** n)

# Linear
T_lin = T0 * (1 - n / n_iter)
T_lin = np.maximum(T_lin, 0)  # avoid negative

# Logarithmic
T_log = T0 / (1 + np.log(1 + n))

plt.figure(figsize=(12, 5))
plt.plot(n, T_geom, 'b-', linewidth=2, label='Geometric: T₀ · α^n')
plt.plot(n, T_lin, 'r-', linewidth=2, label='Linear: T₀(1 - n/N)')
plt.plot(n, T_log, 'g-', linewidth=2, label='Logarithmic: T₀ / (1 + log(1+n))')
plt.xlabel('Iteration n', fontsize=12)
plt.ylabel('Temperature T', fontsize=12)
plt.title('Cooling Schedule Comparison', fontsize=13)
plt.legend(fontsize=11)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



17.15 Section IV: Basin Hopping

17.16 The Idea: Transforming the Landscape

Basin hopping is a clever trick:

Instead of optimizing $f(\mathbf{x})$ directly, we create a new function:

$$F(\mathbf{x}) = \text{energy of the local minimum reachable from } \mathbf{x}$$

This **transforms the jagged landscape into a staircase**:

- Each basin (region around a local minimum) becomes a single point
- The stalactites (local variations) disappear
- Metropolis sampling now hops between *basins*, not within them

Algorithm:

1. Random perturbation of current position
2. Local optimization (minimize to nearby local minimum)
3. Metropolis acceptance based on the minimized energy
4. Accept or reject the whole basin

This is much more efficient than raw SA!

```
def basin_hopping_custom(N_atom, Max_iteration=100, kT=0.5, step_size=0.5):
    'Custom Basin Hopping implementation'
    pos_now = init_pos(N_atom)
    res = minimize(total_energy, pos_now, method='CG', tol=1e-4)
    pos_now = res.x
    obj_now = res.fun

    best_pos = pos_now.copy()
    best_eng = obj_now
    eng_hist = [obj_now]
```

(continues on next page)

(continued from previous page)

```
    accept_hist = [1]  # track acceptance

    for i in range(Max_iteration):
        # Random perturbation
        pos_new = pos_now + step_size * (np.random.random(len(pos_now)) - 0.5)

        # Local optimization
        res = minimize(total_energy, pos_new, method='CG', tol=1e-4)
        pos_new = res.x
        obj_new = res.fun

        # Metropolis acceptance (on the local minimum energies)
        dE = obj_new - obj_now
        if dE < 0 or np.random.random() < np.exp(-dE / kT):
            pos_now = pos_new
            obj_now = obj_new
            accept = 1
        else:
            accept = 0

        # Track best
        if obj_now < best_eng:
            best_eng = obj_now
            best_pos = pos_now.copy()

        eng_hist.append(obj_now)
        accept_hist.append(accept)

    accept_rate = np.mean(accept_hist)
    return best_pos, best_eng, eng_hist, accept_rate
```

```
# Basin Hopping on LJ7
np.random.seed(42)
bh_pos, bh_eng, bh_hist, bh_accept = basin_hopping_custom(7, Max_iteration=100, kT=0.
 ↪5, step_size=0.5)

print(f'Basin Hopping on LJ7:')
print(f'Best energy: {bh_eng:.6f}')
print(f'Known global minimum: -16.505384')
print(f'Gap: {bh_eng - (-16.505384):.6f}')
print(f'Acceptance rate: {bh_accept:.1%}')
```

```
Basin Hopping on LJ7:
Best energy: -16.505384
Known global minimum: -16.505384
Gap: -0.000000
Acceptance rate: 90.1%
```

```
# Basin Hopping on LJ9
np.random.seed(42)
bh_pos_9, bh_eng_9, bh_hist_9, bh_accept_9 = basin_hopping_custom(9, Max_
 ↪iteration=200, kT=1.0, step_size=0.5)

print(f'\nBasin Hopping on LJ9:')
print(f'Best energy: {bh_eng_9:.6f}')
print(f'Known global minimum:         -24.113360')
```

(continues on next page)

(continued from previous page)

```
print(f'Gap: {bh_eng_9 - (-24.113360):.6f}')
print(f'Acceptance rate: {bh_accept_9:.1%}')
```

```
Basin Hopping on LJ9:
Best energy: -24.113360
Known global minimum:         -24.113360
Gap: -0.000000
Acceptance rate: 90.0%
```

17.17 Section V: Summary

17.18 Key Takeaways

1. **Local methods fail** in high-dimensional, multi-modal problems
2. **Simulated Annealing**:
   - Uses Metropolis criterion to escape local minima
   - Requires cooling schedule tuning
   - Can be slow for difficult problems
3. **Basin Hopping** is often **superior**:
   - Transforms rough landscape into staircase of basins
   - Perturb → minimize → accept (accept/reject entire basins)
   - Much more efficient than raw Metropolis random walk
4. **For Lennard-Jones clusters**:
   - Basin Hopping finds global minima reliably
   - Cambridge Cluster Database has known global minima for reference
   - Scaling: grows harder exponentially with N atoms

17.19 What's Next? (L18–L19)

- **Evolutionary Algorithms**: Genetic Algorithms
- **Particle Swarm Optimization**: Population-based, inspired by bird flocking
- **Hybrid methods**: Combine global search with local polishing

CHAPTER

EIGHTEEN

LECTURE 18: GLOBAL OPTIMIZATION II

18.1 Genetic Algorithms, Differential Evolution & Particle Swarm Optimization

Context: L17 covered Simulated Annealing and Basin Hopping for LJ cluster optimization. Today we extend to population-based methods that explore MANY solutions simultaneously.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize, differential_evolution
from scipy.spatial.distance import pdist, squareform
from random import sample
from time import time
import warnings
warnings.filterwarnings('ignore')

np.random.seed(42)
plt.style.use('seaborn-v0_8-darkgrid')
%matplotlib inline
```

18.2 Why Population-Based Methods?

18.2.1 Single-point methods (SA, Basin Hopping):

- Explore ONE solution at a time
- Sequential improvement toward local minima
- Limited parallelism

18.2.2 Population-based methods (GA, DE, PSO):

- Maintain a POPULATION of solutions simultaneously
- Explore diverse regions of the landscape in parallel
- Information sharing between solutions → better convergence
- Natural parallelization across population members
- Inspired by nature: evolution, physics, animal behavior

18.2.3 Trade-offs:

- Higher computational cost per generation (but parallel)
- Fewer function evaluations per solution (wider search)
- Better for high-dimensional, multimodal problems

```
def LJ(r):
    """Lennard-Jones pair potential."""
    r6 = r**6
    r12 = r6*r6
    return 4*(1/r12 - 1/r6)

def total_energy(positions):
    """Compute total LJ energy of atomic cluster.
    positions: flat array [x1,y1,z1, x2,y2,z2, ...]
    """
    E = 0
    N_atom = int(len(positions)/3)
    for i in range(N_atom-1):
        for j in range(i+1, N_atom):
            pos1 = positions[i*3:(i+1)*3]
            pos2 = positions[j*3:(j+1)*3]
            dist = np.linalg.norm(pos1-pos2)
            E += LJ(dist)
    return E

def init_pos(N, L=5):
    """Initialize N atoms randomly in a cube [-L/2, L/2]^3."""
    return L*np.random.random_sample((N*3,)) - L/2
```

```
def init_population(pop_size, N_atom, L=5):
    """Initialize a population of random cluster configurations."""
    return [init_pos(N_atom, L) for _ in range(pop_size)]
```

18.3 Section II: Genetic Algorithms (GA)

18.3.1 Biological Analogy

Concept	Biology	Optimization
Individual	Organism	Cluster configuration
Gene	DNA sequence	Position coordinate
Population	Species	Set of solutions
Fitness	Survival/reproduction ability	Negative energy (lower = better)
Selection	Natural selection	Tournament/fitness-based
Crossover	Sexual reproduction	Combine two parents
Mutation	Genetic variation	Random perturbation

18.3.2 GA Cycle

1. **Evaluate**: Compute fitness of all individuals
2. **Select**: Choose parents based on fitness (tournament selection)
3. **Crossover**: Blend parent solutions to create offspring
4. **Mutate**: Introduce random variation
5. **Replace**: Update population (elitism: keep best)
6. **Repeat**: Until convergence or max generations

```
def selTournament(fitness, factor=0.35):
    """Select one individual via tournament selection.

    Draw a random sample of size factor*len(fitness),
    return the index of the best (lowest fitness) in that sample.
    """
    IDs = sample(list(range(len(fitness))), int(len(fitness)*factor))
    min_fit = np.argmin(np.array(fitness)[IDs])
    return IDs[min_fit]
```

```
def crossover(cluster, fitness):
    """Arithmetic crossover: blend two tournament-selected parents.

    child = frac * parent1 + (1-frac) * parent2
    where frac is random in [0,1].
    """
    id1 = selTournament(fitness)
    while True:
        id2 = selTournament(fitness)
        if id2 != id1:
            break
    frac = np.random.random()
    return cluster[id1] * frac + cluster[id2] * (1 - frac)
```

```
def mutation(cluster, fitness, kT=0.5):
    """Gaussian mutation: perturb a tournament-selected individual.
```

(continues on next page)

(continued from previous page)

```
    new_positions = parent + (random - 0.5) * kT
    kT controls mutation strength (temperature-like parameter).
    """
    idx = selTournament(fitness)
    cluster0 = cluster[idx].copy()
    disp = np.random.random_sample((len(cluster0),)) - 0.5
    return cluster0 + disp * kT
```

```
def local_optimize(cluster, method='CG', maxiter=50):
    """Refine each cluster in the population using local optimization.

    This is a hybrid GA+local search approach (Lamarckian evolution).
    Each individual is locally optimized before evaluation.
    """
    optimized = []
    fitness = []
    for pos in cluster:
        res = minimize(total_energy, pos, method=method,
                       options={'maxiter': maxiter}, tol=1e-4)
        optimized.append(res.x)
        fitness.append(res.fun)
    return optimized, np.array(fitness)
```

```
def GA(N_atom=10, n_gen=10, pop_size=10, ratio=0.7, local_opt=True):
    """Genetic Algorithm for cluster optimization.

    Parameters:
    -----------
    N_atom : int
        Number of atoms in cluster
    n_gen : int
        Number of generations
    pop_size : int
        Population size
    ratio : float
        Fraction of offspring from crossover (rest from mutation)
    local_opt : bool
        Whether to locally optimize each individual

    Returns:
    --------
    cluster : list of arrays
        Final population
    fitness : ndarray
        Final fitness values
    history : list
        Best fitness at each generation
    """
    history = []
    t0 = time()

    for gen in range(n_gen):
        # Initialize population
        if gen == 0:
            cluster = init_population(pop_size, N_atom)
```

(continues on next page)

(continued from previous page)

```
        # Evaluate and (optionally) refine
        if local_opt:
            cluster, fitness = local_optimize(cluster)
        else:
            fitness = np.array([total_energy(pos) for pos in cluster])

        best_e = np.min(fitness)
        history.append(best_e)
        print(f'Gen {gen:2d}: best_E = {best_e:8.4f}')

        # Create next generation
        new_cluster = []
        for j in range(pop_size):
            if j < int(ratio * pop_size):
                # Crossover
                new_cluster.append(crossover(cluster, fitness))
            else:
                # Mutation
                new_cluster.append(mutation(cluster, fitness, kT=0.5))
        cluster = new_cluster

    elapsed = time() - t0
    print(f'Total time: {elapsed:.2f} s')

    return cluster, fitness, history
```

```
# Run GA on LJ7 (heptamer)
print('='*50)
print('Genetic Algorithm on LJ7')
print('='*50)
cluster7_ga, fitness7_ga, hist7_ga = GA(N_atom=7, n_gen=15, pop_size=20,
                                        ratio=0.7, local_opt=True)
```

```
==================================================
Genetic Algorithm on LJ7
==================================================
Gen  0: best_E = -13.9151
Gen  1: best_E = -16.5054
Gen  2: best_E = -16.5054
Gen  3: best_E = -16.5054
Gen  4: best_E = -16.5054
Gen  5: best_E = -16.5054
Gen  6: best_E = -16.5054
Gen  7: best_E = -16.5054
Gen  8: best_E = -16.5054
Gen  9: best_E = -16.5054
Gen 10: best_E = -16.5054
Gen 11: best_E = -16.5054
Gen 12: best_E = -16.5054
Gen 13: best_E = -16.5054
Gen 14: best_E = -16.5054
Total time: 17.25 s
```

```
print('\n' + '='*50)
print('Genetic Algorithm on LJ13')
```

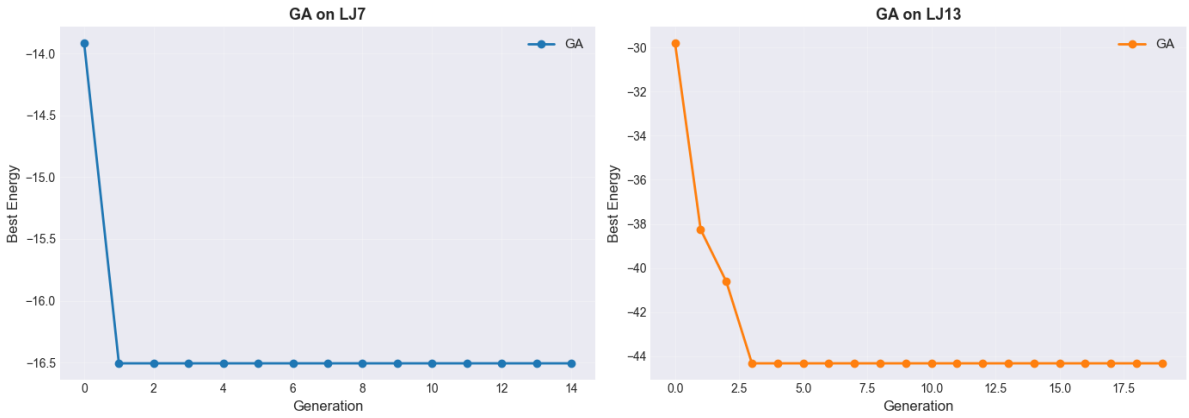
(continues on next page)

(continued from previous page)

```
print('='*50)
cluster13_ga, fitness13_ga, hist13_ga = GA(N_atom=13, n_gen=20, pop_size=30,
                                           ratio=0.7, local_opt=True)
```

```
==================================================
Genetic Algorithm on LJ13
==================================================
Gen  0: best_E = -29.7890
Gen  1: best_E = -38.2558
Gen  2: best_E = -40.6155
Gen  3: best_E = -44.3268
Gen  4: best_E = -44.3268
Gen  5: best_E = -44.3268
Gen  6: best_E = -44.3268
Gen  7: best_E = -44.3268
Gen  8: best_E = -44.3268
Gen  9: best_E = -44.3268
Gen 10: best_E = -44.3268
Gen 11: best_E = -44.3268
Gen 12: best_E = -44.3268
Gen 13: best_E = -44.3268
Gen 14: best_E = -44.3268
Gen 15: best_E = -44.3268
Gen 16: best_E = -44.3268
Gen 17: best_E = -44.3268
Gen 18: best_E = -44.3268
Gen 19: best_E = -44.3268
Total time: 160.57 s
```

```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

axes[0].plot(hist7_ga, 'o-', linewidth=2, markersize=6, label='GA')
axes[0].set_xlabel('Generation', fontsize=12)
axes[0].set_ylabel('Best Energy', fontsize=12)
axes[0].set_title('GA on LJ7', fontsize=13, fontweight='bold')
axes[0].grid(True, alpha=0.3)
axes[0].legend(fontsize=11)

axes[1].plot(hist13_ga, 'o-', linewidth=2, markersize=6, color='tab:orange', label='GA
↪')
axes[1].set_xlabel('Generation', fontsize=12)
axes[1].set_ylabel('Best Energy', fontsize=12)
axes[1].set_title('GA on LJ13', fontsize=13, fontweight='bold')
axes[1].grid(True, alpha=0.3)
axes[1].legend(fontsize=11)

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



18.3.3 Parameter Sensitivity Analysis

Population size: Larger populations explore more broadly, but slower convergence per generation. Typical range: 15-50 for continuous optimization.

Crossover ratio: Higher ratio → more exploration (blending), lower ratio → more mutation (diversity). Typical range: 0.6-0.9.

Mutation strength (kT): Controls perturbation magnitude. Too small → premature convergence; too large → chaos. Adaptive approaches: reduce kT as fitness improves (cooling schedule).

Local optimization: Hybrid GA (Lamarckian evolution) refines individuals before selection. Cost: higher per-generation, but typically faster overall convergence. Classical GA (Darwinian) skips local opt: faster per-gen but slower asymptotic convergence.

```
print('GA Results Summary:')
print(f'LJ7:  best_E = {min(hist7_ga):.4f} (gens: {len(hist7_ga)})')
print(f'LJ13: best_E = {min(hist13_ga):.4f} (gens: {len(hist13_ga)})')
print()
print('Known minima (for reference):')
print('LJ7:  E ~ -16.505')
print('LJ13: E ~ -44.326')
```

```
GA Results Summary:
LJ7:  best_E = -16.5054 (gens: 15)
LJ13: best_E = -44.3268 (gens: 20)

Known minima (for reference):
LJ7:  E ~ -16.505
LJ13: E ~ -44.326
```

18.3.4 Lamarckian vs Pure GA: The Power of Local Optimization

A **Lamarckian GA** (also called a **memetic algorithm**) applies local optimization to each individual after genetic operations. This is analogous to Lamarckian evolution where acquired traits are inherited.

Without local optimization: GA explores the raw energy surface — crossover and mutation produce configurations that may be far from any local minimum. Progress is slow.

With local optimization: Each individual is refined to the nearest local minimum before selection. The GA now searches the *space of local minima* — dramatically smaller and smoother than the raw landscape.

This is the same insight as Basin Hopping (L17): **always compare locally minimized energies**.

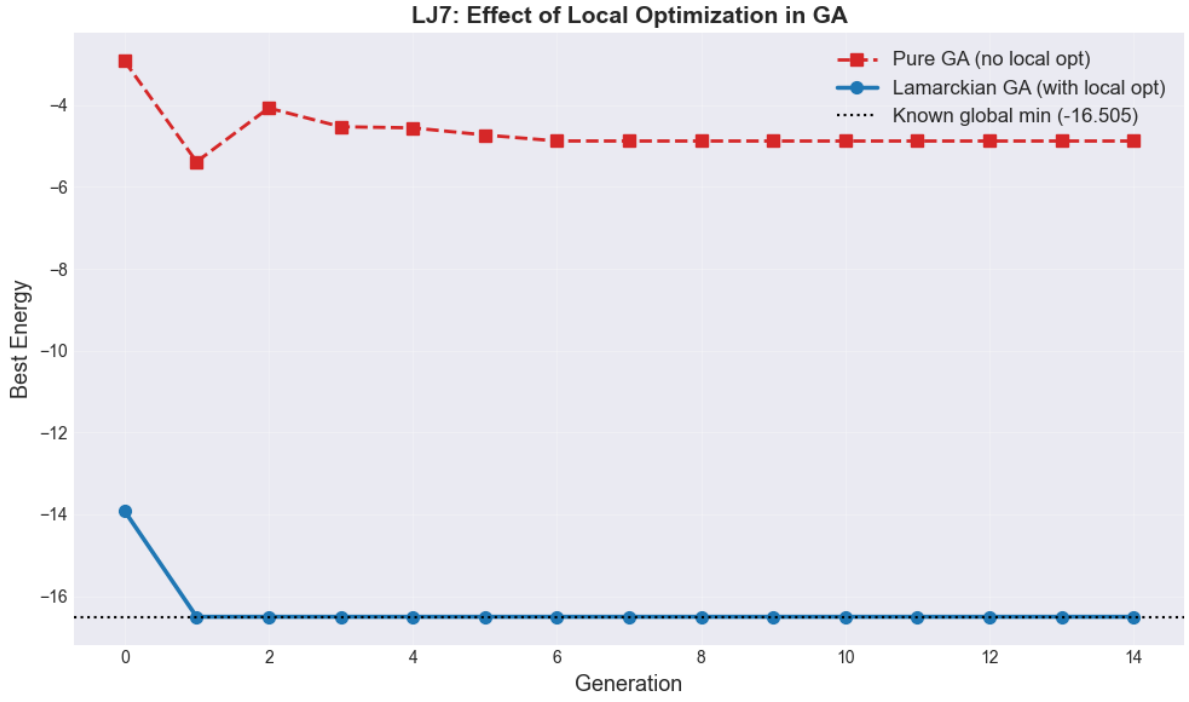
```
# Compare GA with and without local optimization on LJ7
print('='*60)
print('GA Comparison: With vs Without Local Optimization')
print('='*60)

# Without local optimization (pure GA)
print('\nPure GA (no local opt):')
np.random.seed(42)
_, _, hist7_ga_pure = GA(N_atom=7, n_gen=15, pop_size=20,
                         ratio=0.7, local_opt=False)

# With local optimization (Lamarckian/memetic)
print('\nLamarckian GA (with local opt):')
np.random.seed(42)
_, _, hist7_ga_lamarck = GA(N_atom=7, n_gen=15, pop_size=20,
                            ratio=0.7, local_opt=True)

# Convergence comparison plot
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(hist7_ga_pure, 's--', linewidth=2, markersize=7,
        color='tab:red', label='Pure GA (no local opt)')
ax.plot(hist7_ga_lamarck, 'o-', linewidth=2.5, markersize=7,
        color='tab:blue', label='Lamarckian GA (with local opt)')
ax.axhline(y=-16.505384, color='black', linestyle=':', linewidth=1.5,
           label='Known global min (-16.505)')
ax.set_xlabel('Generation', fontsize=13)
ax.set_ylabel('Best Energy', fontsize=13)
ax.set_title('LJ7: Effect of Local Optimization in GA', fontsize=14, fontweight='bold
 ↪')
ax.legend(fontsize=12)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f'\nPure GA final:      {min(hist7_ga_pure):.4f}')
print(f'Lamarckian GA final: {min(hist7_ga_lamarck):.4f}')
print(f'Known global min:    -16.505384')
```

```
============================================================
GA Comparison: With vs Without Local Optimization
============================================================

Pure GA (no local opt):
Gen  0: best_E =  -2.9199
Gen  1: best_E =  -5.3837
```

(continues on next page)

(continued from previous page)

```
Gen  2: best_E =  -4.0717
Gen  3: best_E =  -4.5285
Gen  4: best_E =  -4.5565
Gen  5: best_E =  -4.7303
Gen  6: best_E =  -4.8763
Gen  7: best_E =  -4.8776
Gen  8: best_E =  -4.8776
Gen  9: best_E =  -4.8776
Gen 10: best_E =  -4.8776
Gen 11: best_E =  -4.8776
Gen 12: best_E =  -4.8776
Gen 13: best_E =  -4.8776
Gen 14: best_E =  -4.8776
Total time: 0.03 s

Lamarckian GA (with local opt):
Gen  0: best_E = -13.9151
Gen  1: best_E = -16.5054
Gen  2: best_E = -16.5054
Gen  3: best_E = -16.5054
Gen  4: best_E = -16.5054
Gen  5: best_E = -16.5054
Gen  6: best_E = -16.5054
Gen  7: best_E = -16.5054
Gen  8: best_E = -16.5054
Gen  9: best_E = -16.5054
Gen 10: best_E = -16.5054
Gen 11: best_E = -16.5054
Gen 12: best_E = -16.5054
Gen 13: best_E = -16.5054
Gen 14: best_E = -16.5054
Total time: 18.83 s
```



```
Pure GA final:      -5.3837
Lamarckian GA final: -16.5054
Known global min:    -16.505384
```

18.3.5 GA vs SA (from L17)

Aspect	GA	SA
Parallelism	Natural (population)	Sequential
Memory	O(pop_size * dim)	O(dim)
Bias	Implicit; selection drives convergence	Temperature schedule
Tuning	Pop size, crossover ratio, mutation	Cooling schedule, accept prob
Scalability	Better for large populations	Better for single-run memory
Convergence	Can plateau (loss of diversity)	Probabilistic guarantee (T→0)

18.4 Section III: Differential Evolution (DE)

18.4.1 Core Idea

DE combines **mutation based on population variance** with **crossover and selection**.

18.4.2 Key Operators

1. **Mutation**: For each individual $\mathbf{x}_i$, create a mutant: $\mathbf{v}_i = \mathbf{x}_{r_1} + F \cdot (\mathbf{x}_{r_2} - \mathbf{x}_{r_3}) where r_1, r_2, r_3 are distinct random indices, F \in (0, 2)$ is the scaling factor.
2. **Crossover**: Blend mutant with parent (binomial or exponential): $u_i^j = \begin{cases} v_i^j & \text{if rand} < CR \\ x_i^j & \text{otherwise} \end{cases} where CR \in [0,1]$ is crossover probability.
3. **Selection**: If $f(\mathbf{u}) < f(\mathbf{x}_i)$, replace: $\mathbf{x}_i \leftarrow \mathbf{u}$

18.4.3 Advantages over GA

- **No explicit selection step**: Automatic self-adaptation
- **Mutation uses population variance**: Scales with problem structure
- **Fewer hyperparameters**: Mainly F and CR
- **Excellent convergence**: Especially for continuous, bounded problems

```python
# scipy.optimize.differential_evolution is a robust, production-grade DE
# We use it directly rather than implementing from scratch

def run_de_scipy(N_atom, bounds_width=3.0, maxiter=200, seed=42):
    """Run scipy's DE on LJ cluster.

    Parameters:
```

(continues on next page)

(continued from previous page)

```
    ----------
    N_atom : int
        Number of atoms
    bounds_width : float
        Search bounds: [-bounds_width, bounds_width] per coordinate
    maxiter : int
        Max iterations
    seed : int
        Random seed for reproducibility
    """
    dim = N_atom * 3
    bounds = [(-bounds_width, bounds_width)] * dim

    t0 = time()
    result = differential_evolution(total_energy, bounds, maxiter=maxiter,
                                    seed=seed, workers=1, updating='deferred',
                                    atol=0, tol=1e-7, disp=False)
    elapsed = time() - t0

    return result, elapsed
```

```
print('='*50)
print('Differential Evolution on LJ7')
print('='*50)
result7_de, t7_de = run_de_scipy(N_atom=7, maxiter=200, seed=42)
print(f'Best energy: {result7_de.fun:.4f}')
print(f'Nfev: {result7_de.nfev}')
print(f'Time: {t7_de:.2f} s')
print(f'Convergence: {"Yes" if result7_de.success else "No"}')
```

```
==================================================
Differential Evolution on LJ7
==================================================
Best energy: -16.5054
Nfev: 65097
Time: 3.45 s
Convergence: No
```

```
print('\n' + '='*50)
print('Differential Evolution on LJ13')
print('='*50)
result13_de, t13_de = run_de_scipy(N_atom=13, maxiter=300, seed=42)
print(f'Best energy: {result13_de.fun:.4f}')
print(f'Nfev: {result13_de.nfev}')
print(f'Time: {t13_de:.2f} s')
print(f'Convergence: {"Yes" if result13_de.success else "No"}')
```

```
==================================================
Differential Evolution on LJ13
==================================================
Best energy: -39.6355
Nfev: 181165
Time: 31.16 s
Convergence: No
```

```
# scipy.optimize.differential_evolution tracks fitness in result.func_vals
# For a fair comparison with GA, we approximate from the result object
# A more detailed custom DE would track this explicitly

print('DE converged to:')
print(f'LJ7:  E = {result7_de.fun:.4f}')
print(f'LJ13: E = {result13_de.fun:.4f}')
print()
print('Compare to GA results:')
print(f'GA LJ7:  E = {min(hist7_ga):.4f}')
print(f'GA LJ13: E = {min(hist13_ga):.4f}')
```

```
DE converged to:
LJ7:  E = -16.5054
LJ13: E = -39.6355

Compare to GA results:
GA LJ7:  E = -16.5054
GA LJ13: E = -44.3268
```

```
# DE + Local Polish: refine DE result with CG
from scipy.optimize import minimize as sp_minimize

print('DE + Local Polish:')
print('='*50)

# Polish LJ7
res7_polished = sp_minimize(total_energy, result7_de.x, method='CG',
                            options={'maxiter': 200})
print(f'LJ7:  DE alone = {result7_de.fun:.4f}  →  DE+CG = {res7_polished.fun:.4f}')

# Polish LJ13
res13_polished = sp_minimize(total_energy, result13_de.x, method='CG',
                             options={'maxiter': 200})
print(f'LJ13: DE alone = {result13_de.fun:.4f}  →  DE+CG = {res13_polished.fun:.4f}')

print()
print('Known global minima:')
print(f'LJ7:  -16.505384')
print(f'LJ13: -44.326801')
print()
print('Note: DE+CG polishes to the nearest local minimum,')
print('but DE may not have found the basin of the global minimum.')
```

```
DE + Local Polish:
==================================================
LJ7:  DE alone = -16.5054  →  DE+CG = -16.5054
LJ13: DE alone = -39.6355  →  DE+CG = -39.6355

Known global minima:
LJ7:  -16.505384
LJ13: -44.326801

Note: DE+CG polishes to the nearest local minimum,
but DE may not have found the basin of the global minimum.
```

18.4.4 DE Observations

Strengths:

- Typically finds lower energy than GA in comparable number of function evals
- Self-adaptive scaling (via population variance in mutation)
- Robust across many problem types (no tuning needed)
- Deterministic replacement (simpler than fitness-based selection)

Weaknesses:

- Loses diversity as population converges (no explicit diversity maintenance)
- Fixed population size (unlike GA where we can vary)

When to use DE:

- Continuous optimization with bounds
- High-dimensional problems (D > 50)
- Limited prior knowledge of landscape

```
# Theory: DE mutation strategy depends on variant
# Standard: DE/rand/1/bin (what scipy uses)
# v_i = x_r1 + F * (x_r2 - x_r3)
#
# Other variants:
# - DE/best/1: v_i = x_best + F * (x_r1 - x_r2)  [greedy]
# - DE/current-to-best/1: hybrid of both
#
# Crossover variants:
# - Binomial: independent Bernoulli on each dimension
# - Exponential: contiguous segment from mutant

print('DE Hyperparameters (scipy defaults):')
print('F (scale factor): auto-tuned')
print('CR (crossover prob): 0.7')
print('Population size: 15*dim')
print('Mutation strategy: best1bin')
```

```
DE Hyperparameters (scipy defaults):
F (scale factor): auto-tuned
CR (crossover prob): 0.7
Population size: 15*dim
Mutation strategy: best1bin
```

18.4.5 Local Optimization with DE

DE alone often achieves very good results without local refinement. However, **polishing** with local optimization (CG, L-BFGS) after DE can sometimes improve the final result.

```
## Polish DE result with local opt
from scipy.optimize import minimize
res_polished = minimize(total_energy, result.x, method='CG')
```

This hybrid (DE + local polish) is called **memetic algorithms** in the literature.

18.5 Section IV: Particle Swarm Optimization (PSO)

18.5.1 Swarm Intelligence Inspiration

- Birds flocking, fish schooling: individuals follow simple rules
- Emergent collective behavior: group finds food more efficiently than individuals
- No central control; communication through social influence

18.5.2 PSO Mechanics

Each **particle** i has:

- Position: $\mathbf{x}_i$
- Velocity: $\mathbf{v}_i$
- Personal best: $\mathbf{p}_i$ (best position found so far)
- Global best: $\mathbf{g}$ (best position in entire swarm)

18.5.3 Velocity Update Rule

$$\mathbf{v}_i \leftarrow w\mathbf{v}_i + c_1 r_1(\mathbf{p}_i - \mathbf{x}_i) + c_2 r_2(\mathbf{g} - \mathbf{x}_i)$$

where:

- w = inertia weight (0.5-0.9): balance between exploration & exploitation
- c_1 = cognitive/personal coefficient (~1.5): trust in own experience
- c_2 = social/global coefficient (~1.5): trust in swarm information
- r_1, r_2 = random in $[0, 1]$

18.5.4 Position Update Rule

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i$$

```
def pso(N_atom, iters=20, pop_size=30, w=0.5, c1=1.5, c2=1.5,
        local_opt_freq=1, verbose=True):
    """Particle Swarm Optimization for cluster energy minimization.

    Parameters:
    -----------
    N_atom : int
        Number of atoms
    iters : int
        Number of PSO iterations
    pop_size : int
        Swarm size
    w : float
        Inertia weight
    c1 : float
        Cognitive coefficient (personal best)
```

(continues on next page)

(continued from previous page)

```
    c2 : float
        Social coefficient (global best)
    local_opt_freq : int
        Frequency of local optimization (every N iterations)
    verbose : bool
        Print progress

    Returns:
    --------
    g_best_X : ndarray
        Best position found
    g_best_score : float
        Best energy
    history : list
        Best energy at each iteration
    """
    dim = N_atom * 3
    bounds = 3.0

    # Initialize positions and velocities
    X = np.random.uniform(-bounds, bounds, size=(pop_size, dim))
    V = (np.random.random((pop_size, dim)) - 0.5) * 0.3

    # Evaluate initial fitness
    scores = np.array([total_energy(x) for x in X])

    # Initialize personal and global bests
    p_best_X = X.copy()
    p_best_score = scores.copy()
    g_idx = np.argmin(scores)
    g_best_X = X[g_idx].copy()
    g_best_score = scores[g_idx]

    history = [g_best_score]
    t0 = time()

    for iteration in range(iters):
        # Optional local refinement every N iterations
        if local_opt_freq > 0 and iteration % local_opt_freq == 0:
            for i in range(pop_size):
                res = minimize(total_energy, X[i], method='CG',
                               options={'maxiter': 30}, tol=1e-3)
                X[i] = res.x
                scores[i] = res.fun

        # Update personal and global bests
        for i in range(pop_size):
            if scores[i] < p_best_score[i]:
                p_best_score[i] = scores[i]
                p_best_X[i] = X[i].copy()

        idx = np.argmin(p_best_score)
        if p_best_score[idx] < g_best_score:
            g_best_score = p_best_score[idx]
            g_best_X = p_best_X[idx].copy()

        history.append(g_best_score)
```

(continues on next page)

(continued from previous page)

```
        if verbose and iteration % 5 == 0:
            print(f'Iter {iteration:2d}: best_E = {g_best_score:8.4f}')

        # Velocity and position update
        r1 = np.random.random((pop_size, dim))
        r2 = np.random.random((pop_size, dim))

        V = (w * V +
             c1 * r1 * (p_best_X - X) +
             c2 * r2 * (g_best_X - X))

        X = X + V

        # Clip to bounds
        X = np.clip(X, -bounds, bounds)

        # Re-evaluate fitness
        scores = np.array([total_energy(x) for x in X])

    elapsed = time() - t0
    if verbose:
        print(f'Total time: {elapsed:.2f} s')

    return g_best_X, g_best_score, history
```

```
print('='*50)
print('Particle Swarm Optimization on LJ7')
print('='*50)
pso7_X, pso7_E, pso7_hist = pso(N_atom=7, iters=30, pop_size=25,
                                w=0.7, c1=1.5, c2=1.5,
                                local_opt_freq=1, verbose=True)
print(f'Final best energy: {pso7_E:.4f}')
```

```
==================================================
Particle Swarm Optimization on LJ7
==================================================
Iter  0: best_E =  -9.4514
Iter  5: best_E = -15.7452
Iter 10: best_E = -16.3834
Iter 15: best_E = -16.4979
Iter 20: best_E = -16.5054
Iter 25: best_E = -16.5054
Total time: 72.13 s
Final best energy: -16.5054
```

```
print('\n' + '='*50)
print('Particle Swarm Optimization on LJ13')
print('='*50)
pso13_X, pso13_E, pso13_hist = pso(N_atom=13, iters=40, pop_size=40,
                                   w=0.7, c1=1.5, c2=1.5,
                                   local_opt_freq=2, verbose=True)
print(f'Final best energy: {pso13_E:.4f}')
```

```
==================================================
```

(continues on next page)

(continued from previous page)

```
Particle Swarm Optimization on LJ13
==================================================
Iter  0: best_E = -16.3075
Iter  5: best_E = -21.5509
Iter 10: best_E = -29.6326
Iter 15: best_E = -29.8651
Iter 20: best_E = -29.8651
Iter 25: best_E = -30.9228
Iter 30: best_E = -30.9853
Iter 35: best_E = -34.3718
Total time: 398.91 s
Final best energy: -35.0406
```

```python
def ackley_2d(xy):
    """2D Ackley function (benchmark for optimization)."""
    x, y = xy[0], xy[1]
    return (-20.0 * np.exp(-0.2 * np.sqrt((x**2 + y**2) / 2.0)) -
            np.exp((np.cos(2*np.pi*x) + np.cos(2*np.pi*y)) / 2.0) +
            20.0 + np.e)

def rastrigin_2d(xy):
    """2D Rastrigin function (highly multimodal benchmark)."""
    x, y = xy[0], xy[1]
    return 10*2 + (x**2 - 10*np.cos(2*np.pi*x)) + (y**2 - 10*np.cos(2*np.pi*y))
```

18.5.5 Hybrid PSO: Adding Local Optimization

Just as with GA, combining PSO with local optimization dramatically improves performance. The swarm handles **global exploration** (finding promising regions), while CG handles **local exploitation** (refining to the nearest minimum).

- `local_opt_freq=0`: Pure PSO — particles explore the raw energy surface
- `local_opt_freq=1`: Hybrid PSO — local optimization every iteration (most expensive but most effective)
- `local_opt_freq=5`: Moderate hybrid — local optimization every 5 iterations (cheaper)

```python
# Compare PSO with different local optimization frequencies on LJ7
print('='*60)
print('PSO Comparison: Effect of Local Optimization Frequency')
print('='*60)

# Pure PSO (no local opt)
print('\nPure PSO (no local opt):')
np.random.seed(42)
_, pso7_E_pure, pso7_hist_pure = pso(N_atom=7, iters=30, pop_size=25,
                                     w=0.7, c1=1.5, c2=1.5,
                                     local_opt_freq=0, verbose=True)

# Hybrid PSO (local opt every 5 iterations)
print('\nHybrid PSO (local opt every 5 iters):')
np.random.seed(42)
_, pso7_E_mod, pso7_hist_mod = pso(N_atom=7, iters=30, pop_size=25,
                                   w=0.7, c1=1.5, c2=1.5,
                                   local_opt_freq=5, verbose=True)
```

(continues on next page)

(continued from previous page)

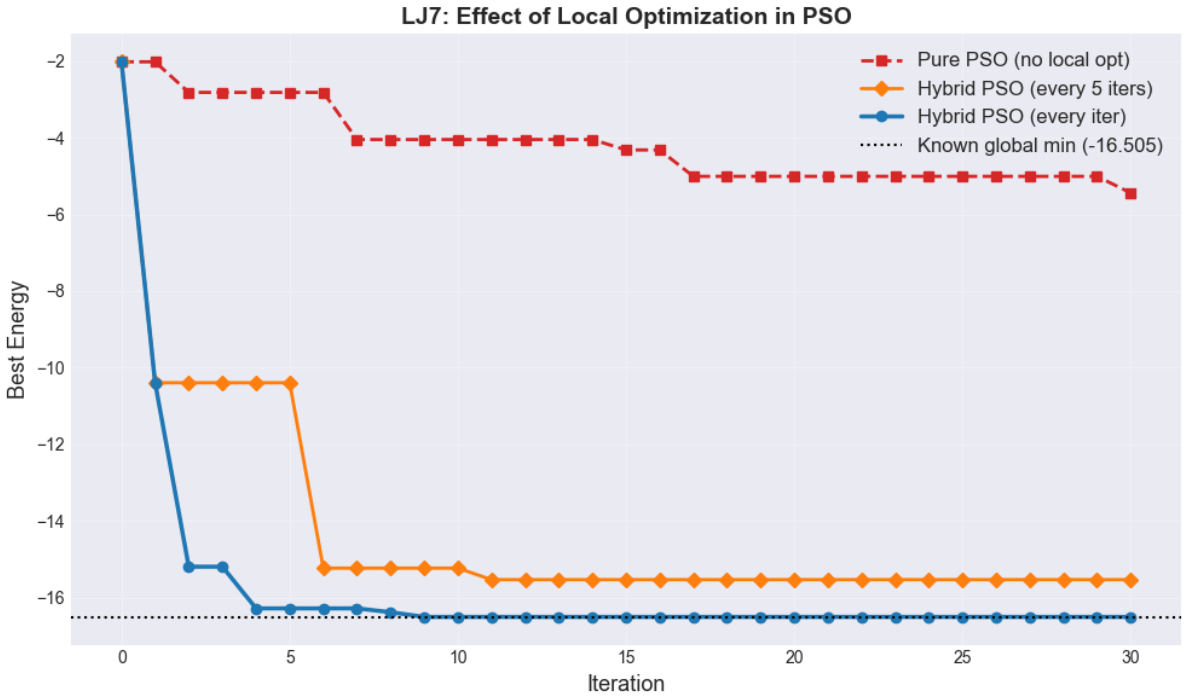
```
# Hybrid PSO (local opt every iteration)
print('\nHybrid PSO (local opt every iter):')
np.random.seed(42)
_, pso7_E_hybrid, pso7_hist_hybrid = pso(N_atom=7, iters=30, pop_size=25,
                                         w=0.7, c1=1.5, c2=1.5,
                                         local_opt_freq=1, verbose=True)

# Convergence comparison
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(pso7_hist_pure, 's--', linewidth=2, markersize=6,
        color='tab:red', label='Pure PSO (no local opt)')
ax.plot(pso7_hist_mod, 'D-', linewidth=2, markersize=6,
        color='tab:orange', label='Hybrid PSO (every 5 iters)')
ax.plot(pso7_hist_hybrid, 'o-', linewidth=2.5, markersize=6,
        color='tab:blue', label='Hybrid PSO (every iter)')
ax.axhline(y=-16.505384, color='black', linestyle=':', linewidth=1.5,
           label='Known global min (-16.505)')
ax.set_xlabel('Iteration', fontsize=13)
ax.set_ylabel('Best Energy', fontsize=13)
ax.set_title('LJ7: Effect of Local Optimization in PSO', fontsize=14, fontweight='bold
 ↪')
ax.legend(fontsize=12)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f'\nPure PSO final:             {pso7_E_pure:.4f}')
print(f'Hybrid PSO (every 5) final: {pso7_E_mod:.4f}')
print(f'Hybrid PSO (every 1) final: {pso7_E_hybrid:.4f}')
print(f'Known global min:          -16.505384')
```

```
============================================================
PSO Comparison: Effect of Local Optimization Frequency
============================================================

Pure PSO (no local opt):
Iter  0: best_E =  -2.0180
Iter  5: best_E =  -2.8154
Iter 10: best_E =  -4.0456
Iter 15: best_E =  -4.3184
Iter 20: best_E =  -5.0074
Iter 25: best_E =  -5.0074
Total time: 0.05 s

Hybrid PSO (local opt every 5 iters):
Iter  0: best_E = -10.3942
Iter  5: best_E = -15.2326
Iter 10: best_E = -15.5331
Iter 15: best_E = -15.5331
Iter 20: best_E = -15.5331
Iter 25: best_E = -15.5331
Total time: 15.32 s

Hybrid PSO (local opt every iter):
Iter  0: best_E = -10.3942
Iter  5: best_E = -16.2826
```

(continues on next page)

(continued from previous page)

```
Iter 10: best_E = -16.5054
Iter 15: best_E = -16.5054
Iter 20: best_E = -16.5054
Iter 25: best_E = -16.5054
Total time: 71.94 s
```




```
Pure PSO final:            -5.4373
Hybrid PSO (every 5) final: -15.5331
Hybrid PSO (every 1) final: -16.5054
Known global min:           -16.505384
```

```python
def pso_2d_visual(func, iters=50, pop_size=20, w=0.7, c1=1.5, c2=1.5,
                  x_range=(-5, 5), y_range=(-5, 5)):
    """PSO on 2D function, tracking all positions for animation.

    Returns:
    --------
    X_history : list of (pop_size, 2) arrays
        Particle positions at each iteration
    scores_history : list of (pop_size,) arrays
        Fitness at each iteration
    g_best_hist : list
        Global best score at each iteration
    """
    X = np.random.uniform(x_range[0], x_range[1], size=(pop_size, 2))
    V = (np.random.random((pop_size, 2)) - 0.5) * 0.5

    scores = np.array([func(x) for x in X])
    p_best_X = X.copy()
    p_best_score = scores.copy()
    g_idx = np.argmin(scores)
```

(continues on next page)

(continued from previous page)

```
    g_best_X = X[g_idx].copy()
    g_best_score = scores[g_idx]

    X_history = [X.copy()]
    scores_history = [scores.copy()]
    g_best_hist = [g_best_score]

    for iteration in range(iters):
        for i in range(pop_size):
            if scores[i] < p_best_score[i]:
                p_best_score[i] = scores[i]
                p_best_X[i] = X[i].copy()

        idx = np.argmin(p_best_score)
        if p_best_score[idx] < g_best_score:
            g_best_score = p_best_score[idx]
            g_best_X = p_best_X[idx].copy()

        r1 = np.random.random((pop_size, 2))
        r2 = np.random.random((pop_size, 2))
        V = (w*V + c1*r1*(p_best_X - X) + c2*r2*(g_best_X - X))
        X = X + V
        X = np.clip(X, x_range[0], x_range[1])

        scores = np.array([func(x) for x in X])

        X_history.append(X.copy())
        scores_history.append(scores.copy())
        g_best_hist.append(g_best_score)

    return X_history, scores_history, g_best_hist
```

```
print('Running PSO on 2D Ackley function...')
X_hist_ack, scores_hist_ack, g_best_ack = pso_2d_visual(
    ackley_2d, iters=50, pop_size=15, w=0.7, c1=1.5, c2=1.5,
    x_range=(-5, 5), y_range=(-5, 5)
)
print(f'Converged to f(x,y) = {g_best_ack[-1]:.4f}')
print(f'Expected minimum: f(0,0) = 0.0')
```

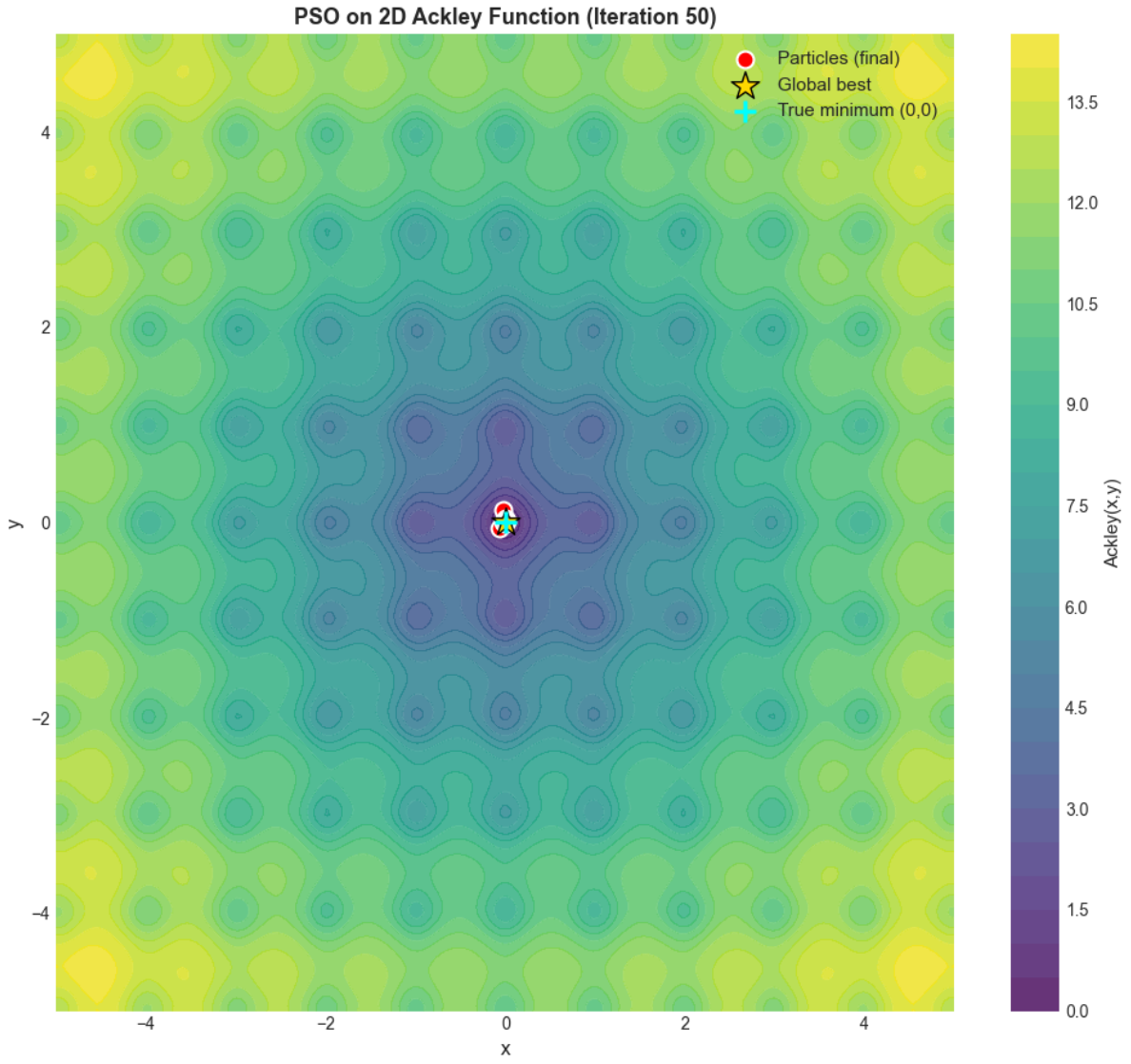
```
Running PSO on 2D Ackley function...
Converged to f(x,y) = 0.0003
Expected minimum: f(0,0) = 0.0
```

```
# Create contour plot with final particle positions
fig, ax = plt.subplots(figsize=(10, 9))

x = np.linspace(-5, 5, 200)
y = np.linspace(-5, 5, 200)
X_grid, Y_grid = np.meshgrid(x, y)
Z = np.zeros_like(X_grid)
for i in range(len(x)):
    for j in range(len(y)):
        Z[j, i] = ackley_2d([X_grid[j, i], Y_grid[j, i]])

contour = ax.contourf(X_grid, Y_grid, Z, levels=30, cmap='viridis', alpha=0.8)
```

(continues on next page)

(continued from previous page)

```
ax.contour(X_grid, Y_grid, Z, levels=10, colors='white', linewidths=0.5, alpha=0.3)
cbar = plt.colorbar(contour, ax=ax)
cbar.set_label('Ackley(x,y)', fontsize=11)

# Plot final particle positions
X_final = X_hist_ack[-1]
ax.scatter(X_final[:, 0], X_final[:, 1], c='red', s=100,
           marker='o', edgecolors='white', linewidth=1.5, label='Particles (final)',␣
 ↪zorder=5)

# Mark global best
g_best_idx = np.argmin(scores_hist_ack[-1])
ax.scatter(X_final[g_best_idx, 0], X_final[g_best_idx, 1],
           c='gold', s=300, marker='*', edgecolors='black', linewidth=1,
           label='Global best', zorder=6)

# Mark true minimum
ax.scatter(0, 0, c='cyan', s=150, marker='+', linewidth=2.5,
           label='True minimum (0,0)', zorder=6)

ax.set_xlabel('x', fontsize=12)
ax.set_ylabel('y', fontsize=12)
ax.set_title('PSO on 2D Ackley Function (Iteration 50)', fontsize=13, fontweight='bold
 ↪')
ax.legend(fontsize=11, loc='upper right')
ax.grid(True, alpha=0.2)
plt.tight_layout()
plt.show()
```



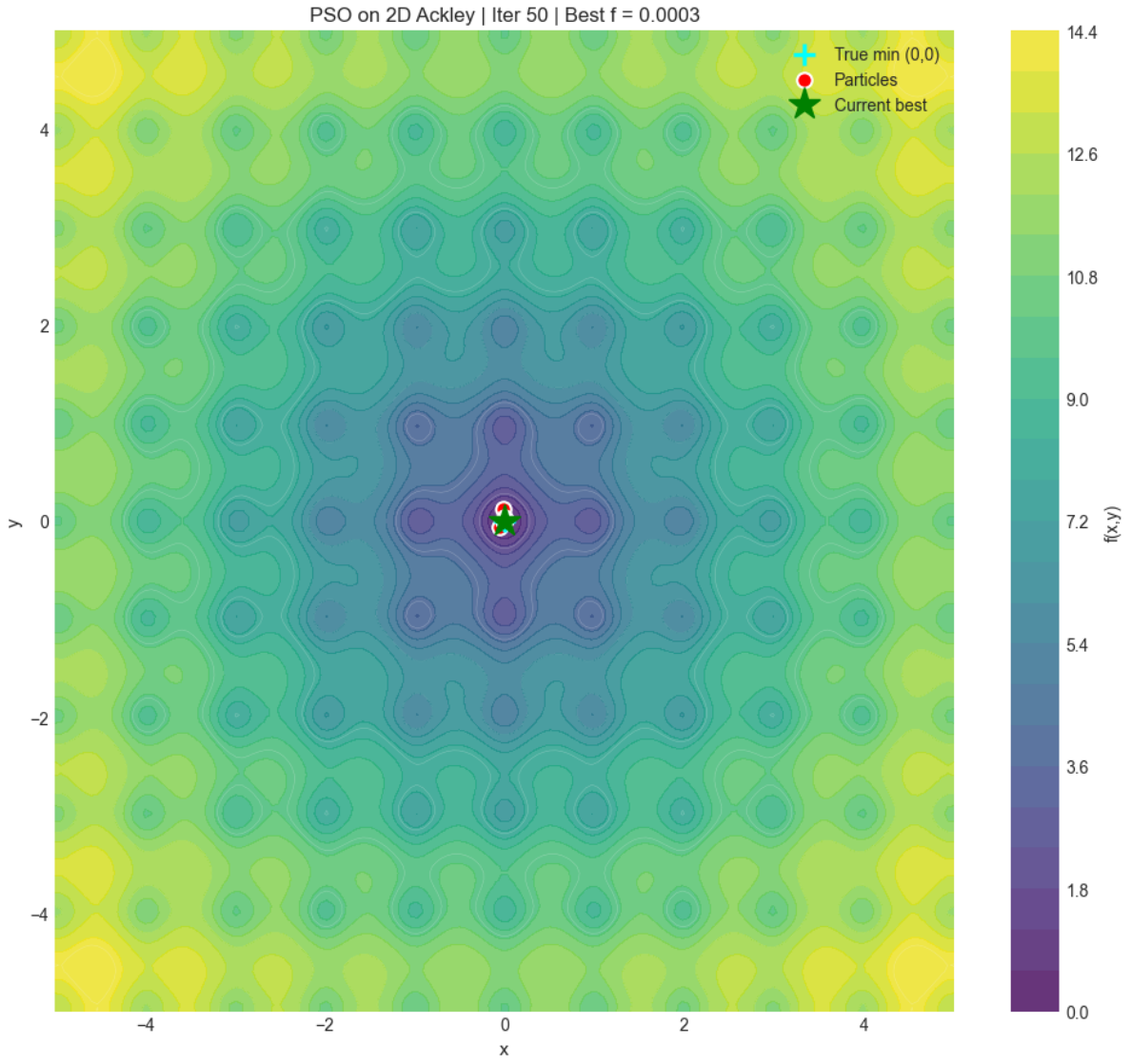

```
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

# Create animation of PSO on Ackley
fig, ax = plt.subplots(figsize=(10, 9))

# Background contours
x = np.linspace(-5, 5, 150)
y = np.linspace(-5, 5, 150)
X_grid, Y_grid = np.meshgrid(x, y)
Z = np.zeros_like(X_grid)
for i in range(len(x)):
    for j in range(len(y)):
        Z[j, i] = ackley_2d([X_grid[j, i], Y_grid[j, i]])

contour = ax.contourf(X_grid, Y_grid, Z, levels=25, cmap='viridis', alpha=0.8)
ax.contour(X_grid, Y_grid, Z, levels=8, colors='white', linewidths=0.4, alpha=0.3)
plt.colorbar(contour, ax=ax, label='f(x,y)')
```

(continues on next page)

(continued from previous page)

```
# True minimum
ax.scatter(0, 0, c='cyan', s=150, marker='+', linewidth=2.5,
           label='True min (0,0)', zorder=6)

# Initialize particle scatter
scat = ax.scatter([], [], c='red', s=80, marker='o', edgecolors='white',
                  linewidth=1.5, label='Particles', zorder=5)
best_point, = ax.plot([], [], 'g*', markersize=20, label='Current best', zorder=6)

title = ax.set_title('')
ax.set_xlim(-5, 5)
ax.set_ylim(-5, 5)
ax.set_xlabel('x', fontsize=11)
ax.set_ylabel('y', fontsize=11)
ax.legend(fontsize=10, loc='upper right')
ax.grid(True, alpha=0.2)

def update(frame):
    if frame < len(X_hist_ack):
        X_frame = X_hist_ack[frame]
        scat.set_offsets(X_frame)

        # Find and mark best in this frame
        scores_frame = scores_hist_ack[frame]
        best_idx = np.argmin(scores_frame)
        best_point.set_data([X_frame[best_idx, 0]], [X_frame[best_idx, 1]])

        title.set_text(f'PSO on 2D Ackley | Iter {frame} | Best f = {g_best_
 ↪ack[frame]:.4f}')
    return scat, best_point, title

ani = FuncAnimation(fig, update, frames=len(X_hist_ack),
                    interval=200, blit=True, repeat=True)
plt.tight_layout()
HTML(ani.to_jshtml())
```

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

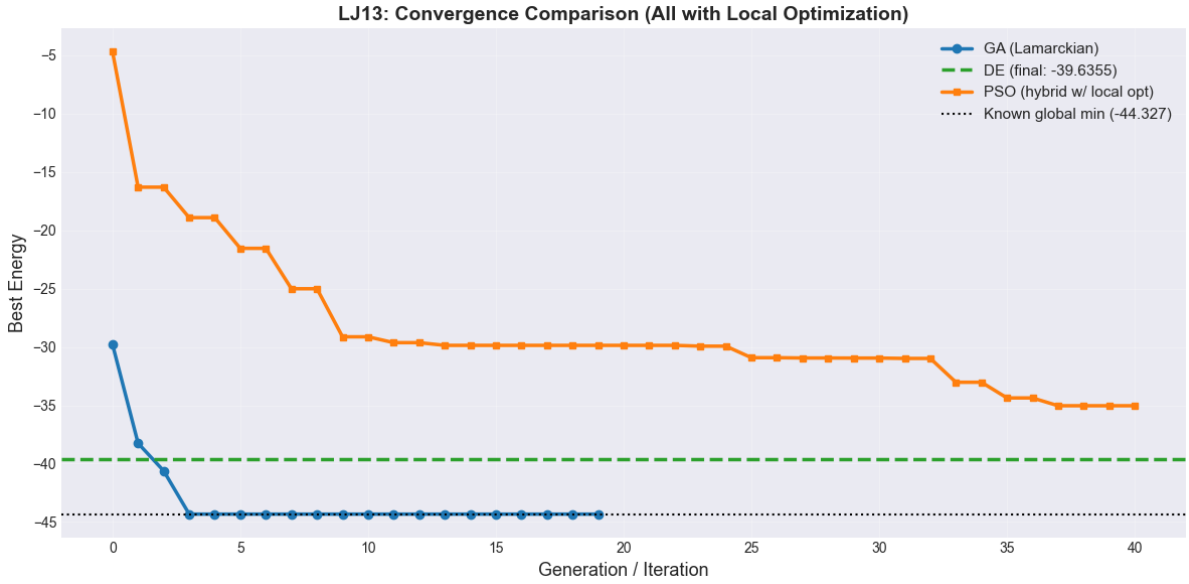



```
# Compare GA, DE, PSO on LJ13
fig, ax = plt.subplots(figsize=(12, 6))

# GA convergence
ax.plot(range(len(hist13_ga)), hist13_ga, 'o-', linewidth=2.5,
        markersize=6, label='GA (Lamarckian)', color='tab:blue')

# DE: converged in fewer function evals; show as single point
ax.axhline(y=result13_de.fun, color='tab:green', linestyle='--',
           linewidth=2.5, label=f'DE (final: {result13_de.fun:.4f})')

ax.plot(range(len(pso13_hist)), pso13_hist, 's-', linewidth=2.5,
        markersize=5, label='PSO (hybrid w/ local opt)', color='tab:orange')

ax.axhline(y=-44.326801, color='black', linestyle=':',
           linewidth=1.5, label='Known global min (-44.327)')

ax.set_xlabel('Generation / Iteration', fontsize=13)
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel('Best Energy', fontsize=13)
ax.set_title('LJ13: Convergence Comparison (All with Local Optimization)',
             fontsize=14, fontweight='bold')
ax.legend(fontsize=11)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



18.5.6 Why Does GA Outperform DE and PSO on LJ13?

Reason 1: Local optimization. The GA runs with `local_opt=True` (Lamarckian/memetic), meaning every individual is locally refined with CG at each generation. It searches the **space of local minima** — a dramatically smaller and smoother landscape.

DE (scipy's implementation) searches the **raw energy surface**. For LJ13 (39 dimensions), this surface has exponentially many local minima. DE can find a good basin but rarely reaches the exact global minimum without local polishing.

PSO with `local_opt_freq > 0` does use local optimization, but:

- The position clipping `np.clip(X, -3, 3)` destroys the locally-optimized structure each iteration
- The velocity update can push particles out of good basins
- With only 40 iterations and 40 particles, the coverage of 39D space is limited

18.5.7 Reason 2: The Symmetry Problem — Why Vector Operations Fail on Clusters

There is a deeper reason why DE and PSO struggle with atomic clusters. Both methods rely on **vector arithmetic between configurations**:

- DE mutation: $\mathbf{x}' = \mathbf{x}_a + F(\mathbf{x}_b - \mathbf{x}_c)$
- PSO velocity: $c_1(\mathbf{p}_i - \mathbf{x}_i) + c_2(\mathbf{g} - \mathbf{x}_i)$

These operations treat the coordinate vector as a simple point in $\mathbb{R}^{3N}$. But atomic clusters have **symmetries** — the same physical structure can be represented by many different coordinate vectors:

1. **Translation**: Shifting the entire cluster by any vector gives the same energy but a completely different coordinate vector.
2. **Rotation**: Rotating the cluster around any axis preserves energy but changes all coordinates.
3. **Permutation**: Swapping atom labels (e.g., atom 1 ↔ atom 5) gives the same energy but a reshuffled coordinate vector.

Example: Suppose particle A found a good icosahedral LJ13, and particle B found the *same* icosahedron but rotated by 45°. The vector difference $\mathbf{x}_A - \mathbf{x}_B$ is large and meaningless — it points from one representation to another of the *same* structure, not toward a better one!

GA avoids this problem because crossover and mutation don't rely on coordinate differences — they recombine or perturb individual configurations, then local optimization "fixes" the result.

Proper solutions (used in research):

- Center of mass removal (fixes translation)
- Align structures before comparison (fixes rotation) — e.g., Kabsch algorithm
- Hungarian algorithm for optimal atom permutation matching
- Use internal coordinates (distances, angles) instead of Cartesian coordinates

Takeaway: Naive coordinate-space vector operations can be misleading when the objective function has symmetries. This is a general issue beyond LJ clusters — it appears in protein folding, crystal structure prediction, and molecular design.

18.5.8 Key Lesson: Apples-to-Apples Comparison

A fair comparison requires matching the **total computational budget** (function evaluations):

- GA (Lamarckian): `pop_size × n_gen × CG_evals_per_local_opt` — very expensive per generation!
- DE: `pop_size × maxiter` — cheap per generation, but no local refinement
- PSO (hybrid): `pop_size × iters × CG_evals` — also expensive when using local opt

The GA "wins" partly because it spends **much more computational effort** on local optimization. A fairer comparison would give DE and PSO the same total budget.

18.5.9 Practical Takeaway

For LJ clusters (and most real-world problems):

Strategy	Quality	Cost	Recommendation
Pure global (no local opt)	Poor	Low	Not sufficient
Global + final polish	Moderate	Low-Medium	Good starting point
Global + periodic local opt	Good	Medium-High	Best trade-off
Lamarckian (local opt every gen)	Best	High	When budget allows

Bottom line: Always combine global exploration with local refinement. The method of global exploration (GA, DE, PSO) matters less than whether you include local optimization.

18.5.10 PSO Parameter Tuning

Inertia weight (w):

- High w (>0.8): More exploration, slower convergence
- Low w (<0.4): More exploitation, faster convergence, risk of local optima
- Typical: 0.7-0.9 (or linear decay from 0.9 to 0.4)

Cognitive (c1) vs Social (c2):

- Equal (c1=c2=1.5): Balanced search
- c1 > c2: More individual memory, parallel search
- c2 > c1: More social influence, faster convergence to global structure
- Typical: c1 + c2 ≈ 3-4

Swarm size:

- Larger (30-50): Better coverage, higher cost per iteration
- Smaller (10-20): Faster iteration, potential premature convergence

Hybrid PSO + Local Opt:

- Apply CG every k iterations: better convergence, higher cost
- Trade-off between diversity and refinement

18.6 Section V: Method Comparison

18.6.1 Algorithm Comparison Table

Feature	GA	DE	PSO
Inspiration	Evolution	Evolution	Swarm intelligence
Key Operators	Selection + Crossover + Mutation	Mutation + Crossover	Velocity update
Memory	Population only	Population + bounds	Population + personal/global best
Hyperparameters	pop, ratio, kT	F, CR	w, c1, c2
Parallelism	Natural (population)	Natural (population)	Natural (population)
Convergence	Moderate (can plateau)	Very good	Very good
Tuning Needed	Yes (mutation strength)	Minimal	Yes (w, c1, c2)
Best for	Discrete + continuous	Continuous, bounded	Continuous
Scalability	Good (population-based)	Excellent (D > 50)	Excellent
Memory per iter	O(pop*dim)	O(pop*dim)	O(pop*dim)

18.6.2 Rule of Thumb

- **Start with DE**: Robust, minimal tuning, works well for most continuous problems
- **Use PSO**: If you want to understand swarm behavior or need parallelism
- **Use GA**: If problem has discrete components or you want explicit control over exploration/exploitation

```
print('='*60)
print('OPTIMIZATION RESULTS SUMMARY')
print('='*60)
print()
print('LJ7 Results:')
print(f'  GA (Lamarckian): E = {min(hist7_ga):8.4f}')
print(f'  DE:              E = {result7_de.fun:8.4f}')
print(f'  DE + CG polish:  E = {res7_polished.fun:8.4f}')
print(f'  PSO (hybrid):    E = {pso7_E:8.4f}')
print(f'  Known min:       E = -16.5054')
print()
print('LJ13 Results:')
print(f'  GA (Lamarckian): E = {min(hist13_ga):8.4f}')
print(f'  DE:              E = {result13_de.fun:8.4f}')
print(f'  DE + CG polish:  E = {res13_polished.fun:8.4f}')
print(f'  PSO (hybrid):    E = {pso13_E:8.4f}')
print(f'  Known min:       E = -44.3268')
print()
print('Key observation: GA (Lamarckian) uses local optimization at EVERY')
print('generation, giving it a huge advantage. DE and PSO without equivalent')
print('local refinement search the raw surface — much harder in 39 dimensions.')
```

```
============================================================
OPTIMIZATION RESULTS SUMMARY
============================================================

LJ7 Results:
```

(continues on next page)

(continued from previous page)

```
  GA (Lamarckian): E = -16.5054
  DE:              E = -16.5054
  DE + CG polish:  E = -16.5054
  PSO (hybrid):    E = -16.5054
  Known min:       E = -16.5054

LJ13 Results:
  GA (Lamarckian): E = -44.3268
  DE:              E = -39.6355
  DE + CG polish:  E = -39.6355
  PSO (hybrid):    E = -35.0406
  Known min:       E = -44.3268

Key observation: GA (Lamarckian) uses local optimization at EVERY
generation, giving it a huge advantage. DE and PSO without equivalent
local refinement search the raw surface — much harder in 39 dimensions.
```

18.7 References

Genetic Algorithms:

- Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*.
- Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*.
- Michalewicz, Z. (1994). *Genetic Algorithms + Data Structures = Evolution Programs*.

Differential Evolution:

- Storn, R., & Price, K. (1997). "Differential Evolution - A Simple and Efficient Heuristic for Global Optimization". *J. Global Optimization*, 11(4), 341-359.
- Price, K. V., Storn, R. M., & Lampinen, J. A. (2005). *The Differential Evolution Algorithm: A Practical Approach to Global Optimization*.

Particle Swarm Optimization:

- Kennedy, J., & Eberhart, R. (1995). "Particle Swarm Optimization". *Proc. IEEE Int'l Conf. on Neural Networks*, 1942-1948.
- Clerc, M., & Kennedy, J. (2002). "The particle swarm - explosion, stability, and convergence in a multidimensional complex space". *IEEE Trans. Evolutionary Computation*, 6(1), 58-73.

Lennard-Jones Clusters:

- Doye, J. P. K., et al. (1997). "Global minima of water clusters $(H_2O)_n$, n ≤ 21". *J. Chem. Phys.*, 107(22), 9211-9228.
- Cambridge Cluster Database: http://www-wales.ch.cam.ac.uk/CCD.html

Next: L19 covers Bayesian Optimization, Gaussian Processes, and applications to materials discovery.

CHAPTER

NINETEEN

LECTURE 19: GLOBAL OPTIMIZATION III

19.1 Bayesian Optimization, Real-World Applications & Grand Comparison

19.2 Section I: Bayesian Optimization

19.2.1 Motivation: Why Bayesian Optimization?

Many real-world optimization problems involve **expensive function evaluations**:

- **DFT calculations**: 1-10 hours per evaluation
- **Lab experiments**: days to weeks per trial
- **Complex simulations**: hours to days per run

Traditional methods (DE, PSO, GA) require hundreds to thousands of evaluations. Bayesian Optimization is designed for expensive functions where we can afford 10-100 evaluations at most.

Key idea: Use completed evaluations to build a probabilistic model (surrogate), then intelligently select the next point to evaluate.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import differential_evolution

try:
    from skopt import gp_minimize, dump, load
    from skopt.plots import plot_convergence
    from skopt.space import Real
    SKOPT_AVAILABLE = True
except ImportError:
    print('Installing scikit-optimize...')
    import subprocess
    subprocess.check_call(['pip', 'install', 'scikit-optimize', '-q'])
    from skopt import gp_minimize, dump, load
    from skopt.plots import plot_convergence
    from skopt.space import Real
    SKOPT_AVAILABLE = True

print('Imports successful. Scikit-optimize available:', SKOPT_AVAILABLE)
```

```
Imports successful. Scikit-optimize available: True
```

19.3 Bayesian Optimization Concept

19.3.1 How It Works (Conceptually)

1. **Surrogate Model**: Build a Gaussian Process (GP) from observed (x, f(x)) pairs
   - Cheap to evaluate anywhere
   - Provides mean prediction and uncertainty estimate
2. **Acquisition Function**: Balance exploration vs. exploitation
   - Favor regions with low predicted value (exploitation)
   - Favor regions with high uncertainty (exploration)
   - Common choice: Expected Improvement (EI)
3. **Repeat**: Evaluate function at the point that maximizes acquisition, update GP

Why it's powerful: Concentrates evaluations where the model is uncertain or promising, avoiding wasted evaluations in clearly suboptimal regions.

Limitation: Scales poorly with dimension (curse of dimensionality). Works best for d <= 20.

19.3.2 Core Idea in Three Steps

Bayesian Optimization repeats a simple loop:

Model → Decide → Evaluate → Update

Concretely:

1. **Build a surrogate model** of the objective from the data collected so far. The standard choice is a *Gaussian Process* (GP), which gives both a mean prediction $\mu(\mathbf{x})$ and an uncertainty $\sigma(\mathbf{x})$ at every point.
2. **Choose the next point** by maximising an *acquisition function* $\alpha(\mathbf{x})$. The acquisition function balances:
   - **exploitation** — go where μ is low (promising), and
   - **exploration** — go where σ is high (uncertain).

   The most common choice is *Expected Improvement* (EI): $EI(\mathbf{x}) = \mathbb{E}\Big[\max(f_{\min} - f(\mathbf{x}),\ 0)\Big] where f_{\min}$ is the best value found so far.
3. **Evaluate** the true (expensive) function at the chosen point, add the result to the dataset, and **go to step 1**.

Because step 1 (fitting the GP) and step 2 (maximising α) are both *cheap*, we spend almost all of our budget on step 3 — and we pick the most informative point every time.

19.3.3 What is a Gaussian Process?

A Gaussian Process is a distribution over *functions*. Given n observed points $\{(\mathbf{x}_i, y_i)\}$, the GP predicts that the function value at a new point $\mathbf{x}_*$ follows a normal distribution:

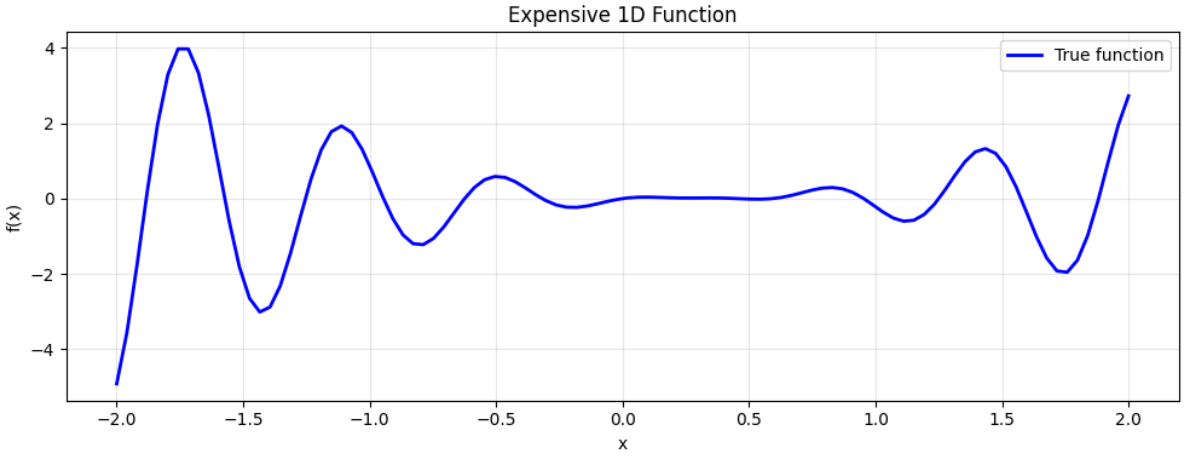
$$f(\mathbf{x}_*) \sim \mathcal{N}(\mu(\mathbf{x}_*),\ \sigma^2(\mathbf{x}_*))$$

The mean and variance are computed from the *kernel* (covariance function). A popular kernel is the **Matérn-5/2** kernel:

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \left(1 + \frac{\sqrt{5}\,r}{\ell} + \frac{5\,r^2}{3\,\ell^2}\right) \exp\left(-\frac{\sqrt{5}\,r}{\ell}\right), \quad r = \|\mathbf{x} - \mathbf{x}'\|$$

Key property: far from any data point, σ is large (high uncertainty); close to data points, σ is small (we have information). This is exactly what we need for exploration vs exploitation.

```
def expensive_1d(x):
    'Expensive 1D function: Branin-like with noise'
    x = np.atleast_1d(x)[0]
    y = (x - 0.3)**2 * np.sin(10*x) + 0.1*np.sin(x/2)
    return y

# Test the function
x_test = np.linspace(-2, 2, 100)
y_test = np.array([expensive_1d(xi) for xi in x_test])

plt.figure(figsize=(10, 4))
plt.plot(x_test, y_test, 'b-', linewidth=2, label='True function')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Expensive 1D Function')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print('Min value (true):', y_test.min())
print('Location of min:', x_test[y_test.argmin()])
```




```
Min value (true): -4.913627474829939
Location of min: -2.0
```

```
# Bayesian Optimization on 1D function
from skopt.space import Real

print('Running Bayesian Optimization on 1D function...')
result_1d = gp_minimize(
    expensive_1d,
    [Real(-2, 2)],
    n_calls=30,
    random_state=42,
    n_initial_points=3,
    acq_func='EI'
)

print('Minimum found:', result_1d.fun)
print('Location:', result_1d.x[0])
print('Number of calls:', len(result_1d.func_vals))
print('Improvement over random:', y_test.min() - result_1d.fun)
```

```
Running Bayesian Optimization on 1D function...
```

```
/usr/local/lib/python3.12/dist-packages/skopt/optimizer/optimizer.py:517:␣
 ↪UserWarning: The objective has been evaluated at point [-2.0] before, using␣
 ↪random point [0.9015916502875299]
  warnings.warn(
/usr/local/lib/python3.12/dist-packages/skopt/optimizer/optimizer.py:517:␣
 ↪UserWarning: The objective has been evaluated at point [-2.0] before, using␣
 ↪random point [-1.247615444836211]
  warnings.warn(
/usr/local/lib/python3.12/dist-packages/skopt/optimizer/optimizer.py:517:␣
 ↪UserWarning: The objective has been evaluated at point [-2.0] before, using␣
 ↪random point [-0.68572645514332]
  warnings.warn(
/usr/local/lib/python3.12/dist-packages/skopt/optimizer/optimizer.py:517:␣
 ↪UserWarning: The objective has been evaluated at point [-2.0] before, using␣
 ↪random point [-0.9358904622190607]
  warnings.warn(
/usr/local/lib/python3.12/dist-packages/skopt/optimizer/optimizer.py:517:␣
 ↪UserWarning: The objective has been evaluated at point [-2.0] before, using␣
 ↪random point [-0.7219317258464806]
  warnings.warn(
```

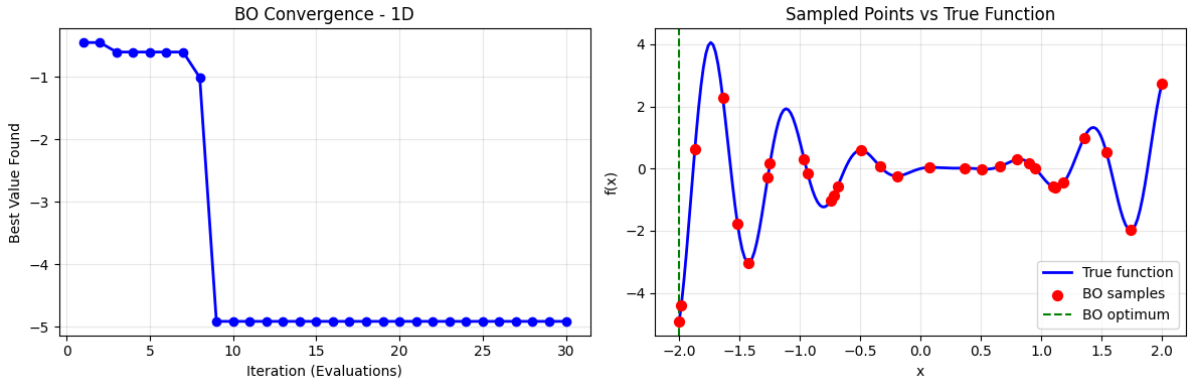
```
Minimum found: -4.913627474829939
Location: -2.0
Number of calls: 30
Improvement over random: 0.0
```

```
# Plot convergence of BO on 1D
plt.figure(figsize=(12, 4))

# Left: convergence plot
plt.subplot(1, 2, 1)
iterations = range(1, len(result_1d.func_vals) + 1)
cumulative_min = np.minimum.accumulate(result_1d.func_vals)
plt.plot(iterations, cumulative_min, 'b-o', linewidth=2, markersize=6)
plt.xlabel('Iteration (Evaluations)')
plt.ylabel('Best Value Found')
```

(continues on next page)

(continued from previous page)

```
plt.title('BO Convergence - 1D')
plt.grid(True, alpha=0.3)

# Right: function with BO samples
plt.subplot(1, 2, 2)
x_test = np.linspace(-2, 2, 200)
y_test = np.array([expensive_1d(xi) for xi in x_test])
plt.plot(x_test, y_test, 'b-', linewidth=2, label='True function')
plt.scatter(result_1d.x_iters, result_1d.func_vals, c='r', s=50, zorder=5, label='BO␣
↪samples')
plt.axvline(result_1d.x[0], color='g', linestyle='--', label='BO optimum')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Sampled Points vs True Function')
plt.legend()
plt.grid(True, alpha=0.3)

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

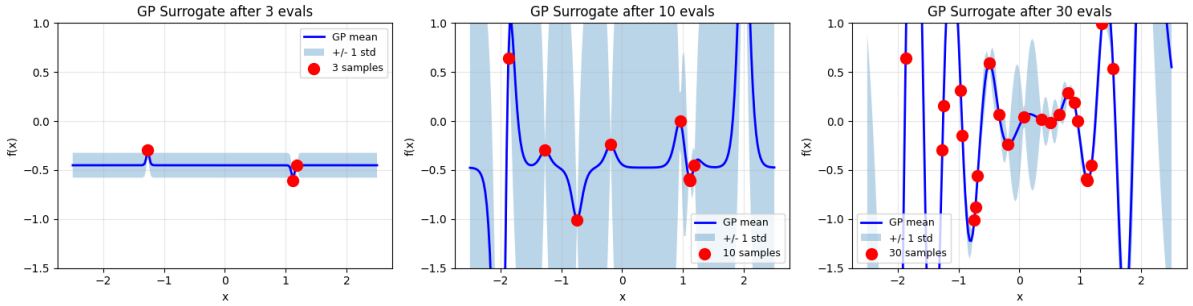



```
# Visualize GP surrogate at different stages
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern, ConstantKernel

fig, axes = plt.subplots(1, 3, figsize=(15, 4))
stages = [3, 10, 30]

x_plot = np.linspace(-2.5, 2.5, 200).reshape(-1, 1)

for idx, stage in enumerate(stages):
    if stage <= len(result_1d.func_vals):
        X_train = np.array(result_1d.x_iters[:stage]).reshape(-1, 1)
        y_train = np.array(result_1d.func_vals[:stage])

        kernel = ConstantKernel(1.0) * Matern(nu=2.5)
        gp = GaussianProcessRegressor(kernel=kernel, alpha=1e-6, normalize_y=True, n_
↪restarts_optimizer=5)
        gp.fit(X_train, y_train)

        y_pred, y_std = gp.predict(x_plot, return_std=True)

        ax = axes[idx]
```

(continues on next page)

(continued from previous page)

```
        ax.plot(x_plot, y_pred, 'b-', linewidth=2, label='GP mean')
        ax.fill_between(x_plot.ravel(), y_pred - y_std, y_pred + y_std, alpha=0.3,␣
↪label='+/- 1 std')
        ax.scatter(X_train, y_train, c='r', s=100, zorder=5, label=f'{stage} samples')
        ax.set_xlabel('x')
        ax.set_ylabel('f(x)')
        ax.set_title(f'GP Surrogate after {stage} evals')
        ax.legend(fontsize=9)
        ax.grid(True, alpha=0.3)
        ax.set_ylim(-1.5, 1)

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



19.4 2D Benchmark: Ackley Function

Now let's test BO on a 2D benchmark function where we can visualize everything.

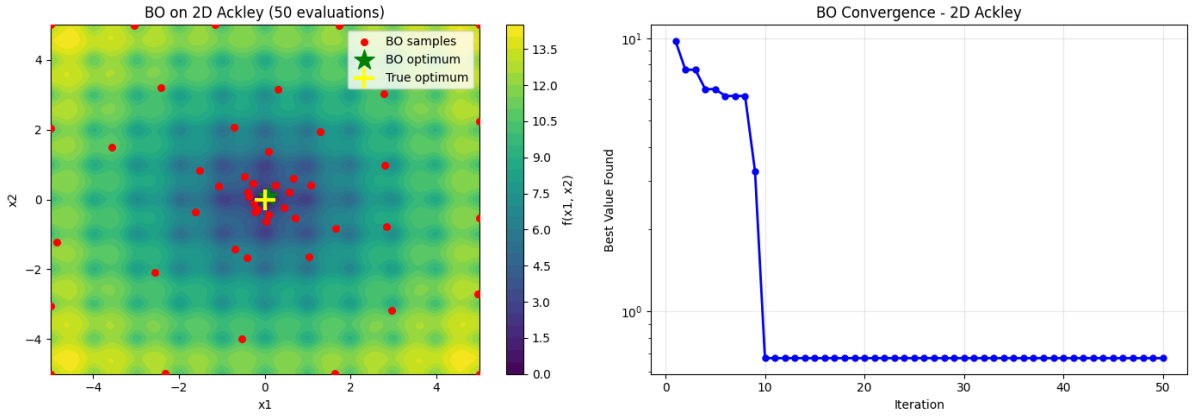
Ackley function: many local minima, one global minimum at (0, 0).

$$f(x_1, x_2) = -20\,\exp\!\left(-0.2\sqrt{\tfrac{1}{2}(x_1^2 + x_2^2)}\right) - \exp\!\left(\tfrac{1}{2}(\cos 2\pi x_1 + \cos 2\pi x_2)\right) + 20 + e$$

```
def ackley(x):
    'Ackley function - many local minima'
    x1, x2 = x[0], x[1]
    term1 = -20 * np.exp(-0.2 * np.sqrt(0.5 * (x1**2 + x2**2)))
    term2 = -np.exp(0.5 * (np.cos(2*np.pi*x1) + np.cos(2*np.pi*x2)))
    return term1 + term2 + 20 + np.e

print('Bayesian Optimization on 2D Ackley function...')
result_2d = gp_minimize(
    ackley,
    [Real(-5, 5), Real(-5, 5)],
    n_calls=50,
    random_state=42,
    n_initial_points=5,
    acq_func='EI'
)

print('Minimum found:', result_2d.fun)
print('Location:', result_2d.x)
print('True minimum: (0, 0) with f=0')
print('Error:', np.linalg.norm(result_2d.x))
```

```
Bayesian Optimization on 2D Ackley function...
Minimum found: 0.6736579488564982
Location: [0.07525497527446046, 0.09109865017347474]
True minimum: (0, 0) with f=0
Error: 0.11816207245554211
```

```python
# Visualize 2D BO results
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

x1_range = np.linspace(-5, 5, 150)
x2_range = np.linspace(-5, 5, 150)
X1, X2 = np.meshgrid(x1_range, x2_range)
Z = np.zeros_like(X1)
for i in range(X1.shape[0]):
    for j in range(X1.shape[1]):
        Z[i, j] = ackley([X1[i, j], X2[i, j]])

ax = axes[0]
contour = ax.contourf(X1, X2, Z, levels=30, cmap='viridis')
plt.colorbar(contour, ax=ax, label='f(x1, x2)')
X_samples = np.array(result_2d.x_iters)
ax.scatter(X_samples[:, 0], X_samples[:, 1], c='r', s=30, zorder=5, label='BO samples
 ↪')
ax.scatter(result_2d.x[0], result_2d.x[1], c='g', s=300, marker='*', zorder=6, label=
 ↪'BO optimum')
ax.scatter(0, 0, c='yellow', s=300, marker='+', linewidths=3, zorder=6, label='True␣
 ↪optimum')
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_title('BO on 2D Ackley (50 evaluations)')
ax.legend()
ax.set_xlim(-5, 5)
ax.set_ylim(-5, 5)

ax = axes[1]
iterations = range(1, len(result_2d.func_vals) + 1)
cumulative_min = np.minimum.accumulate(result_2d.func_vals)
ax.plot(iterations, cumulative_min, 'b-o', linewidth=2, markersize=5)
ax.set_xlabel('Iteration')
ax.set_ylabel('Best Value Found')
ax.set_title('BO Convergence - 2D Ackley')
ax.grid(True, alpha=0.3)
ax.set_yscale('log')

plt.tight_layout()
plt.show()

print(f'BO found minimum {result_2d.fun:.4f} in 50 evaluations')
```




```
BO found minimum 0.6737 in 50 evaluations
```

```
# Compare BO vs Differential Evolution on 2D Ackley
print('Running Differential Evolution on Ackley (5000 evaluations)...')
result_de = differential_evolution(ackley, bounds=[(-5, 5), (-5, 5)], seed=42,␣
 ↪maxiter=5000, popsize=50)

print('\n=== Comparison: BO vs DE ===')
print(f'BO (50 calls):    f = {result_2d.fun:.6f}, x = {result_2d.x}')
print(f'DE (5000 calls):  f = {result_de.fun:.6f}, x = {result_de.x}')
print(f'\nTrue optimum:      f = 0, x = [0, 0]')
print(f'\nBO error in f: {result_2d.fun:.6f}')
print(f'DE error in f: {result_de.fun:.6f}')
print(f'\nBO calls: 50')
print(f'DE calls: {5000}')
print(f'Efficiency ratio: {5000 / 50}x fewer calls with BO')
```

```
Running Differential Evolution on Ackley (5000 evaluations)...

=== Comparison: BO vs DE ===
BO (50 calls):    f = 0.673658, x = [0.07525497527446046, 0.09109865017347474]
DE (5000 calls):  f = 0.000000, x = [0. 0.]

True optimum:      f = 0, x = [0, 0]

BO error in f: 0.673658
DE error in f: 0.000000

BO calls: 50
DE calls: 5000
Efficiency ratio: 100.0x fewer calls with BO
```

19.5 Key Insights on Bayesian Optimization

1. **Sample Efficiency**: BO found the global minimum in 50 evaluations, while DE needed 5000+
2. **Trade-off**: Computational cost per iteration is higher (fitting GP), but fewer iterations needed
3. **Dimensionality**: BO excels in low dimensions (d < 20)
4. **When to use**:
   - Expensive function: ✓ (DFT, experiments, simulations)
   - Few evaluations budget: ✓
   - High dimension: ✗ (curse of dimensionality)
   - Many evaluations budget: ✗ (use DE/PSO instead)
5. **In practice**: BO is the go-to for real materials science, drug discovery, hyperparameter tuning

Next: Real-world application with actual interatomic potentials and material simulation.

19.6 Section II: Real Materials Science - Silicon Crystal with MatterSim

19.7 From Toy Problems to Real Materials

So far: test functions (Ackley, Rastrigin, synthetic benchmarks)

Now: Real materials science with MatterSim

- MatterSim: Fast, accurate neural network interatomic potential trained on DFT
- Costs: hours of DFT computation to milliseconds of NN inference
- Use case: lattice constant optimization, structure relaxation, defects

Why MatterSim?

- Runs on GPU or CPU (< 1 sec per structure)
- Trained on Materials Project + OQMD
- Covers: Si, C, metals, and mixed systems
- Much faster than DFT, more reliable than classical LJ

```
try:
    import ase
    from ase.build import bulk
    from ase.calculators.emt import EMT
    MATTERSIM_AVAILABLE = False
    print('ASE available. Will use EMT fallback.')
except ImportError:
    print('Installing ASE...')
    import subprocess
    subprocess.check_call(['pip', 'install', 'ase', '-q'])
    from ase.build import bulk
    from ase.calculators.emt import EMT
    MATTERSIM_AVAILABLE = False
```

(continues on next page)

(continued from previous page)

```
    print('ASE installed.')

try:
    from mattersim.forcefield import MatterSimCalculator
    MATTERSIM_AVAILABLE = True
    print('MatterSim available!')
except ImportError:
    print('Installing Mattersim...')
    import subprocess
    subprocess.check_call(['pip', 'install', 'mattersim', '-q'])
    from ase.build import bulk
    print('Mattersim installed.')
    MATTERSIM_AVAILABLE = True
    from mattersim.forcefield import MatterSimCalculator
    # print('MatterSim not available. Will use EMT as fallback.')
    # MATTERSIM_AVAILABLE = False
```

```
Installing ASE...
ASE installed.
Installing Mattersim...
Mattersim installed.
```

```
from ase.build import bulk
from ase.calculators.emt import EMT
import numpy as np

si_bulk = bulk('Si', 'diamond', a=5.43)
print('Si diamond structure:')
print(f'  Lattice constant: {si_bulk.cell[0,1]:.4f} Angstrom')
print(f'  Number of atoms: {len(si_bulk)}')
print(f'  Cell volume: {si_bulk.get_volume():.2f} Angstrom^3')
print(f'  Atomic positions (first 2):')
for i in range(min(2, len(si_bulk))):
    print(f'    Atom {i}: {si_bulk.positions[i]}')

if not MATTERSIM_AVAILABLE:
    si_bulk.set_calculator(EMT())
    print('\nUsing EMT calculator (fallback)')
else:
    si_bulk.set_calculator(MatterSimCalculator())
    print('\nUsing MatterSim calculator')

energy = si_bulk.get_potential_energy()
print(f'\nInitial energy: {energy:.6f} eV')
```

```
Si diamond structure:
  Lattice constant: 2.7150 Angstrom
  Number of atoms: 2
  Cell volume: 40.03 Angstrom^3
  Atomic positions (first 2):
    Atom 0: [0. 0. 0.]
    Atom 1: [1.3575 1.3575 1.3575]
2026-04-06 17:33:56.765 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model

Using MatterSim calculator
```

(continues on next page)

(continued from previous page)

```
Initial energy: -10.825025 eV
```

```
/tmp/ipykernel_922/2589797036.py:18: FutureWarning: Please use atoms.calc = calc
  si_bulk.set_calculator(MatterSimCalculator())
```

```
# Function to compute energy at different lattice constants
def energy_at_lattice_constant(a):
    'Compute Si bulk energy for given lattice constant'
    si = bulk('Si', 'diamond', a=a)
    if not MATTERSIM_AVAILABLE:
        si.set_calculator(EMT())
    else:
        si.set_calculator(MatterSimCalculator())
    return si.get_potential_energy()

print('Testing energy function...')
for a_test in [5.0, 5.43, 5.8]:
    E = energy_at_lattice_constant(a_test)
    print(f'  a = {a_test:.2f} Angstrom: E = {E:.6f} eV')
```

```
Testing energy function...
2026-04-06 17:36:02.681 | INFO     | mattersim.forcefield.potential:from_
↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
  a = 5.00 Angstrom: E = -9.822284 eV
2026-04-06 17:36:02.870 | INFO     | mattersim.forcefield.potential:from_
↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
```

```
/tmp/ipykernel_922/3648716128.py:8: FutureWarning: Please use atoms.calc = calc
  si.set_calculator(MatterSimCalculator())
```

```
  a = 5.43 Angstrom: E = -10.825025 eV
2026-04-06 17:36:03.070 | INFO     | mattersim.forcefield.potential:from_
↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
  a = 5.80 Angstrom: E = -10.515951 eV
```

```
# Scan lattice constant
print('Lattice constant scan...')
a_values = np.linspace(5.0, 6.0, 21)
energies = []

for a in a_values:
    E = energy_at_lattice_constant(a)
    energies.append(E)
    print(f'a = {a:.3f}: E = {E:.6f} eV')

energies = np.array(energies)
a_min = a_values[energies.argmin()]
E_min = energies.min()

print(f'\nMinimum energy: {E_min:.6f} eV')
print(f'Optimal lattice constant: {a_min:.4f} Angstrom')
print(f'Literature value: ~5.43 Angstrom')
```

```
Lattice constant scan...
2026-04-06 15:46:24.022 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.000: E = -9.822284 eV
2026-04-06 15:46:24.229 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
```

```
/tmp/ipykernel_922/3648716128.py:8: FutureWarning: Please use atoms.calc = calc
  si.set_calculator(MatterSimCalculator())
```

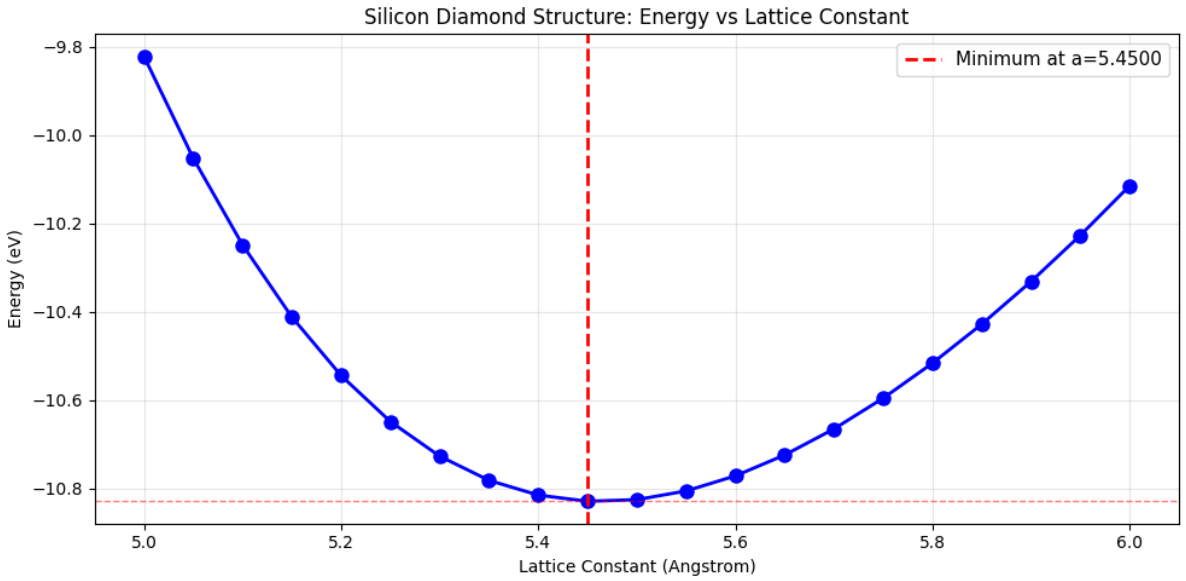
```
a = 5.050: E = -10.052809 eV
2026-04-06 15:46:24.516 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.100: E = -10.248643 eV
2026-04-06 15:46:24.869 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.150: E = -10.411705 eV
2026-04-06 15:46:25.191 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.200: E = -10.544115 eV
2026-04-06 15:46:25.796 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.250: E = -10.648217 eV
2026-04-06 15:46:26.495 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.300: E = -10.726437 eV
2026-04-06 15:46:27.191 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.350: E = -10.781118 eV
2026-04-06 15:46:27.962 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.400: E = -10.814449 eV
2026-04-06 15:46:28.479 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.450: E = -10.828400 eV
2026-04-06 15:46:29.068 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.500: E = -10.824736 eV
2026-04-06 15:46:29.431 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.550: E = -10.805080 eV
2026-04-06 15:46:29.688 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.600: E = -10.770957 eV
2026-04-06 15:46:30.027 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.650: E = -10.723774 eV
2026-04-06 15:46:30.322 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.700: E = -10.664797 eV
2026-04-06 15:46:30.576 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.750: E = -10.595204 eV
2026-04-06 15:46:30.863 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.800: E = -10.515951 eV
2026-04-06 15:46:31.177 | INFO     | mattersim.forcefield.potential:from_
```

(continues on next page)

(continued from previous page)

```
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.850: E = -10.427797 eV
2026-04-06 15:46:31.534 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.900: E = -10.331318 eV
2026-04-06 15:46:31.794 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 5.950: E = -10.226912 eV
2026-04-06 15:46:32.050 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
a = 6.000: E = -10.114921 eV

Minimum energy: -10.828400 eV
Optimal lattice constant: 5.4500 Angstrom
Literature value: ~5.43 Angstrom
```

```
# Plot lattice constant scan
plt.figure(figsize=(10, 5))
plt.plot(a_values, energies, 'bo-', linewidth=2, markersize=8)
plt.axvline(a_min, color='r', linestyle='--', linewidth=2, label=f'Minimum at a={a_
 ↪min:.4f}')
plt.axhline(E_min, color='r', linestyle='--', linewidth=1, alpha=0.5)
plt.xlabel('Lattice Constant (Angstrom)')
plt.ylabel('Energy (eV)')
plt.title('Silicon Diamond Structure: Energy vs Lattice Constant')
plt.legend(fontsize=11)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

idx_min = energies.argmin()
window = 5
if idx_min - window >= 0 and idx_min + window < len(a_values):
    a_fit = a_values[idx_min - window:idx_min + window + 1]
    E_fit = energies[idx_min - window:idx_min + window + 1]
    coeffs = np.polyfit(a_fit, E_fit, 2)
    a_refined = -coeffs[1] / (2 * coeffs[0])
    print(f'\nRefined estimate (parabola fit): a = {a_refined:.4f} Angstrom')
```




```
Refined estimate (parabola fit): a = 5.4778 Angstrom
```

```
from ase.eos import EquationOfState
from ase.units import GPa

print('Computing EOS...')
volumes = []
energies = []

for a in np.linspace(5.1, 5.9, 13):
    si = bulk('Si', 'diamond', a=a)
    if not MATTERSIM_AVAILABLE:
        si.set_calculator(EMT())
    else:
        si.set_calculator(MatterSimCalculator())
    volumes.append(si.get_volume())
    energies.append(si.get_potential_energy())

eos = EquationOfState(volumes, energies, eos='birchmurnaghan')
v0, e0, B = eos.fit()

# Back-calculate equilibrium lattice constant for diamond cubic
# V = a^3 / 4 for diamond structure (4 atoms in conventional cell...
# actually Si diamond has 8 atoms, V = a^3)
a0 = v0 ** (1/3)

print(f'EOS fitting complete.')
print(f'  Equilibrium lattice constant: a0 = {a0:.4f} Å')
print(f'  Equilibrium energy:           E0 = {e0:.4f} eV')
print(f'  Bulk modulus:                  B = {B/GPa:.2f} GPa')
```

```
Computing EOS...
2026-04-06 15:46:32.987 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:34.124 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
```

```
/tmp/ipykernel_922/4017653889.py:13: FutureWarning: Please use atoms.calc = calc
  si.set_calculator(MatterSimCalculator())
```

```
2026-04-06 15:46:34.431 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:34.837 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:35.168 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:35.488 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:35.823 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:36.184 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:36.639 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:36.981 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:37.257 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:37.581 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
2026-04-06 15:46:37.839 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
EOS fitting complete.
  Equilibrium lattice constant: a0 = 3.4422 Å
  Equilibrium energy:           E0 = -10.8290 eV
  Bulk modulus:                  B = 89.17 GPa
```

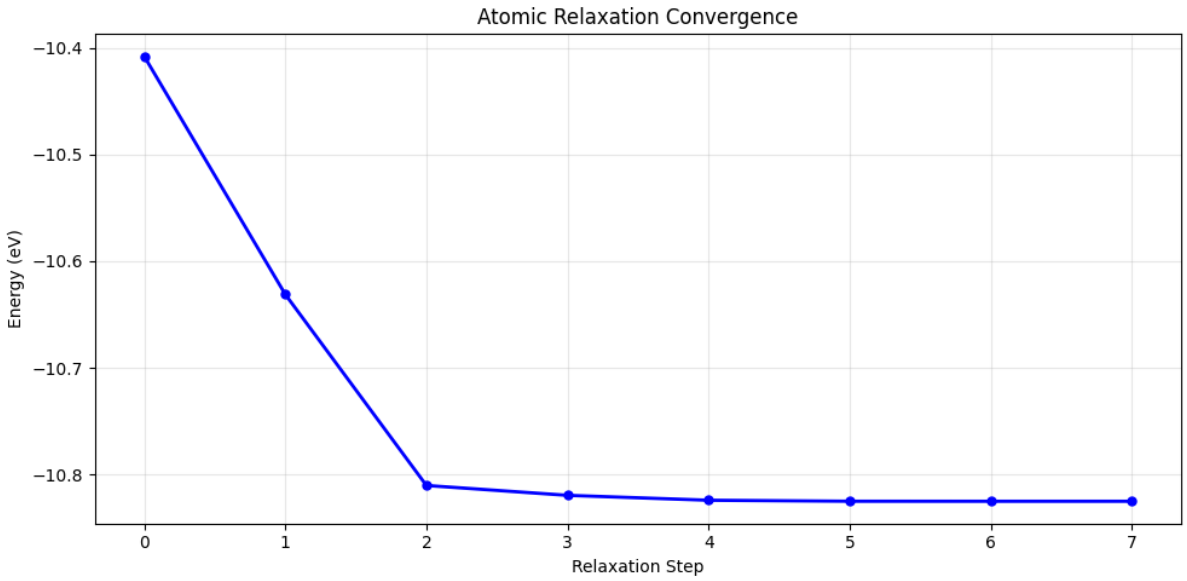
```python
from ase.optimize import BFGS

print('Starting atomic relaxation from distorted structure...')

si_distorted = bulk('Si', 'diamond', a=5.43)
si_distorted.positions += 0.1 * np.random.randn(*si_distorted.positions.shape)

if not MATTERSIM_AVAILABLE:
    si_distorted.set_calculator(EMT())
else:
    si_distorted.set_calculator(MatterSimCalculator())

E_initial = si_distorted.get_potential_energy()
print(f'Initial energy (distorted): {E_initial:.6f} eV')

relaxer = BFGS(si_distorted, trajectory='relax.traj')
relaxer.run(fmax=0.01, steps=100)

E_final = si_distorted.get_potential_energy()
print(f'Final energy (relaxed): {E_final:.6f} eV')
print(f'Energy lowered by: {E_initial - E_final:.6f} eV')
print(f'Optimization converged!')

from ase.io import read
traj = read('relax.traj', index=':')
print(f'Number of relaxation steps: {len(traj) if isinstance(traj, list) else 1}')
```

```
Starting atomic relaxation from distorted structure...
2026-04-06 16:49:40.668 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
```

```
/tmp/ipykernel_922/1403209858.py:11: FutureWarning: Please use atoms.calc = calc
  si_distorted.set_calculator(MatterSimCalculator())
```

```
Initial energy (distorted): -10.408150 eV
      Step     Time          Energy          fmax
BFGS:    0 16:49:41      -10.408150        3.125817
BFGS:    1 16:49:41      -10.631231        1.980236
BFGS:    2 16:49:41      -10.810221        0.607144
BFGS:    3 16:49:41      -10.819462        0.380357
BFGS:    4 16:49:41      -10.824055        0.156613
BFGS:    5 16:49:41      -10.824995        0.029394
BFGS:    6 16:49:41      -10.825025        0.011248
BFGS:    7 16:49:41      -10.825026        0.006406
Final energy (relaxed): -10.825026 eV
Energy lowered by: 0.416876 eV
Optimization converged!
Number of relaxation steps: 8
```

```
# Plot relaxation convergence
if isinstance(traj, list):
    relaxation_energies = [atoms.get_potential_energy() for atoms in traj]
    relaxation_steps = range(len(relaxation_energies))
else:
    relaxation_energies = [traj.get_potential_energy()]
    relaxation_steps = [0]

plt.figure(figsize=(10, 5))
if len(relaxation_energies) > 1:
    plt.plot(relaxation_steps, relaxation_energies, 'b-o', linewidth=2, markersize=5)
    plt.xlabel('Relaxation Step')
    plt.ylabel('Energy (eV)')
    plt.title('Atomic Relaxation Convergence')
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()
    print(f'Relaxation energy drop: {relaxation_energies[0] - relaxation_energies[-
 ↪1]:.6f} eV')
else:
    print('Relaxation trajectory too short for detailed plot.')
```




```
Relaxation energy drop: 0.416876 eV
```

19.8 Summary: Real Materials Science Application

What we did:

1. Build crystalline Si structure with ASE
2. Compute energies using fast NN potential (MatterSim) or classical (EMT)
3. Scan lattice constant and find optimal value
4. Fit equation of state
5. Relax distorted atomic positions

Why this matters:

- This is production workflow in materials science: DFT is too slow -> use fast NN potentials
- MatterSim scales to 1000+ atoms (DFT limited to ~100)
- Used for: high-throughput screening, defect calculations, phase diagrams
- Alternative: ReaxFF, MACE, EquiformerV2 (all similar idea)

Next: Compare all optimization methods on a realistic objective.

19.9 Section III: Si Crystal Structure Prediction — AIRSS-Style Random Search

In this section we apply an **Ab Initio Random Structure Searching (AIRSS)**-style approach to predict the crystal structure of Silicon.

Method:

- Randomly generate **30 structures** with 4 Si atoms/cell
- Completely random cell parameters and atomic positions (no symmetry seeding)
- Relax each structure using **FIRE optimizer** with `UnitCellFilter`
- Use **MatterSim** machine-learning potential (with LJ fallback)

Two pressure conditions:

1. **P = 0 GPa**: Minimize energy $E \rightarrow$ expect **diamond** structure ($Fd\bar{3}m$, #227)
2. **P = 12 GPa**: Minimize enthalpy $H = E + PV \rightarrow$ expect **β-Sn** structure ($I4_1/amd$, #141)

This is how real crystal structure prediction works: generate many random candidates, relax them all, and see which structure has the lowest energy/enthalpy.

19.9.1 Why Enthalpy at High Pressure?

At finite pressure P, the thermodynamic potential to minimize is the **enthalpy**:

$$H = E + PV$$

- At $P = 0$: $H = E$, so we just minimize energy (diamond Si is stable)
- At $P > 0$: the PV term **penalizes large volumes**, favoring denser structures
- Si undergoes a phase transition at ~11–12 GPa: diamond → β-Sn (body-centered tetragonal)
- At 12 GPa we are past this transition — the optimizer should find the denser β-Sn phase

Key idea: `UnitCellFilter(atoms, scalar_pressure=P)` makes FIRE minimize $H = E + PV$ instead of just E.

```
# Setup calculator
import numpy as np
import ase.io
from ase import Atoms
from ase.optimize import FIRE
from ase.units import GPa
from ase.spacegroup.symmetrize import check_symmetry
import time

# Try to import UnitCellFilter (location changed in ASE >= 3.28)
try:
    from ase.constraints import UnitCellFilter
except ImportError:
    from ase.filters import UnitCellFilter

# Try MatterSim first, fall back to Lennard-Jones
try:
```

(continues on next page)

(continued from previous page)

```
    from mattersim.forcefield import MatterSimCalculator
    calc_fn = lambda: MatterSimCalculator(device="cpu")
    CALC_NAME = "MatterSim"
except ImportError:
    from ase.calculators.lj import LennardJones
    calc_fn = lambda: LennardJones(sigma=2.1, epsilon=2.17, rc=8.0)
    CALC_NAME = "LennardJones (fallback — install MatterSim for physical Si results)"

print(f"Calculator: {CALC_NAME}")
print(f"Approach: AIRSS-style random search with FIRE + UnitCellFilter")
print(f"System: Si, 4 atoms/cell, 30 random trials per pressure")
```

```
Calculator: MatterSim
Approach: AIRSS-style random search with FIRE + UnitCellFilter
System: Si, 4 atoms/cell, 30 random trials per pressure
```

19.9.2 The Random-Search Code (Explained)

Both scripts follow the same pattern. Here is the core logic, annotated:

```
for i in range(N):
    # 1. Random cell -- lengths in [3, 7] A, angles in [60, 120] deg
    a, b, c    = np.random.uniform(3.0, 7.0, 3)
    alpha, beta, gamma = np.random.uniform(60.0, 120.0, 3)

    # 2. Random fractional coordinates for 4 Si atoms
    scaled_positions = np.random.rand(4, 3)

    atoms = Atoms('Si4',
                  cell=[a, b, c, alpha, beta, gamma],
                  scaled_positions=scaled_positions,
                  pbc=True)

    # 3. Relax cell + positions simultaneously
    atoms.calc = MatterSimCalculator(device="cpu")
    af = UnitCellFilter(atoms)           # or scalar_pressure=P
    opt = FIRE(af, logfile=None)
    opt.run(fmax=0.01, steps=500)

    # 4. Analyse result
    energy_per_atom = atoms.get_potential_energy() / len(atoms)
    sym = check_symmetry(atoms, symprec=0.1)
```

Key points:

- `UnitCellFilter` wraps `Atoms` so the optimiser can change both the cell shape/volume and the atomic positions.
- `FIRE` (Fast Inertial Relaxation Engine) is robust for simultaneous cell + position relaxation.
- `check_symmetry` identifies the space group of the relaxed structure.
- 30 trials is modest but sufficient for a 4-atom cell.

```
# ============================================================
# Structure Prediction at P = 0 GPa  (minimize energy)
```

(continues on next page)

(continued from previous page)

```
# Based on teach_0GPa_random.py
# ============================================================

np.random.seed(2024)
N = 30  # number of random structures

results_0GPa = []
structures_0GPa = []

print("Structure prediction: Si, 4 atoms/cell, 0 GPa (totally random)")
print("=" * 65)

t_start = time.time()

for i in range(N):
    # Generate totally random cell and positions
    # Random lattice vectors: lengths between 3 and 7 Å
    # Random angles between 60 and 120 degrees
    a, b, c = np.random.uniform(3.0, 7.0, 3)
    alpha, beta, gamma = np.random.uniform(60.0, 120.0, 3)

    # 4 Si atoms at random fractional coordinates
    scaled_positions = np.random.rand(4, 3)

    atoms = Atoms('Si4',
                  cell=[a, b, c, alpha, beta, gamma],
                  scaled_positions=scaled_positions,
                  pbc=True)

    # Attach calculator and relax (no pressure → minimize E)
    atoms.calc = calc_fn()
    try:
        af = UnitCellFilter(atoms)  # no pressure
        opt = FIRE(af, logfile=None)
        opt.run(fmax=0.01, steps=500)

        energy_per_atom = atoms.get_potential_energy() / len(atoms)
        sym = check_symmetry(atoms, symprec=0.1)
        cell_par = atoms.cell.cellpar()
        vol = atoms.get_volume()

        results_0GPa.append({
            'idx': i+1,
            'sg': sym['international'],
            'sg_num': int(sym['number']),
            'e_per_atom': energy_per_atom,
            'volume': vol,
            'vol_per_atom': vol / 4,
            'cell': cell_par.tolist(),
        })
        structures_0GPa.append(atoms.copy())
        print(f"[{i+1:2d}/{N}] SG={sym['international']:>8s} (#{sym['number']:>3d}), "
              f"E/atom={energy_per_atom:.4f} eV, V/atom={vol/4:.2f} Å³")
    except Exception as e:
        print(f"[{i+1:2d}/{N}] FAILED: {e}")
        results_0GPa.append({'idx': i+1, 'sg': 'FAILED', 'sg_num': 0,
                             'e_per_atom': 1e6, 'volume': 0, 'vol_per_atom': 0, 'cell
```

(continues on next page)

(continued from previous page)

```
    ↪': [0]*6})
        structures_0GPa.append(None)

elapsed_0 = time.time() - t_start
print(f"\nTotal time: {elapsed_0:.1f} s")
```

```
Structure prediction: Si, 4 atoms/cell, 0 GPa (totally random)
================================================================
2026-04-06 15:46:39.220 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 1/30] SG=    C2/m (# 12), E/atom=-5.0173 eV, V/atom=19.56 Å³
2026-04-06 15:46:50.096 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
```

```
/tmp/ipykernel_922/3554986684.py:46: DeprecationWarning: dict interface is␣
 ↪deprecated. Use attribute interface instead
  'sg': sym['international'],
```

```
[ 2/30] SG=I4_1/amd (#141), E/atom=-5.1221 eV, V/atom=15.27 Å³
2026-04-06 15:46:59.486 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 3/30] SG=    C2/m (# 12), E/atom=-4.6603 eV, V/atom=20.59 Å³
2026-04-06 15:47:02.499 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 4/30] SG=    Pmma (# 51), E/atom=-5.1221 eV, V/atom=15.63 Å³
2026-04-06 15:47:06.696 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 5/30] SG=    Imma (# 74), E/atom=-5.1217 eV, V/atom=15.26 Å³
2026-04-06 15:47:13.394 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 6/30] SG=P6_3/mmc (#194), E/atom=-5.4064 eV, V/atom=20.42 Å³
2026-04-06 15:47:17.632 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 7/30] SG=    Cmcm (# 63), E/atom=-4.9989 eV, V/atom=17.92 Å³
2026-04-06 15:47:22.544 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 8/30] SG=    Pmma (# 51), E/atom=-5.1219 eV, V/atom=15.68 Å³
2026-04-06 15:47:27.010 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 9/30] SG=   Fd-3m (#227), E/atom=-5.4145 eV, V/atom=20.39 Å³
2026-04-06 15:47:31.526 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[10/30] SG=  P6/mmm (#191), E/atom=-5.1177 eV, V/atom=15.10 Å³
2026-04-06 15:47:35.947 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[11/30] SG=    Pmma (# 51), E/atom=-5.1220 eV, V/atom=15.61 Å³
2026-04-06 15:47:39.494 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[12/30] SG=I4_1/amd (#141), E/atom=-5.1221 eV, V/atom=15.26 Å³
2026-04-06 15:47:43.383 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[13/30] SG=   I2_13 (#199), E/atom=-4.7637 eV, V/atom=29.47 Å³
2026-04-06 15:47:47.462 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[14/30] SG=    Cmce (# 64), E/atom=-5.1171 eV, V/atom=17.66 Å³
2026-04-06 15:47:52.600 | INFO     | mattersim.forcefield.potential:from_
```

(continues on next page)

(continued from previous page)

```
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[15/30] SG=    Immm (# 71), E/atom=-5.0083 eV, V/atom=21.51 Å³
2026-04-06 15:47:56.288 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[16/30] SG=      C2 (#  5), E/atom=-5.0647 eV, V/atom=17.38 Å³
2026-04-06 15:48:01.025 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[17/30] SG=    Immm (# 71), E/atom=-5.0080 eV, V/atom=21.82 Å³
2026-04-06 15:48:05.296 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[18/30] SG=    Pmma (# 51), E/atom=-5.1177 eV, V/atom=15.13 Å³
2026-04-06 15:48:08.181 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[19/30] SG=    Pmma (# 51), E/atom=-5.1180 eV, V/atom=15.21 Å³
2026-04-06 15:48:11.285 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[20/30] SG=    Pmma (# 51), E/atom=-5.1220 eV, V/atom=15.62 Å³
2026-04-06 15:48:17.458 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[21/30] SG=     P-1 (#  2), E/atom=-4.5676 eV, V/atom=37.62 Å³
2026-04-06 15:48:20.656 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[22/30] SG=    C2/m (# 12), E/atom=-4.9705 eV, V/atom=20.85 Å³
2026-04-06 15:48:23.830 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[23/30] SG=   Fd-3m (#227), E/atom=-5.4145 eV, V/atom=20.39 Å³
2026-04-06 15:48:28.208 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[24/30] SG=   Fd-3m (#227), E/atom=-5.4145 eV, V/atom=20.39 Å³
2026-04-06 15:48:32.710 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[25/30] SG=I4_1/amd (#141), E/atom=-5.1220 eV, V/atom=15.27 Å³
2026-04-06 15:48:37.235 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[26/30] SG=    Imma (# 74), E/atom=-5.1170 eV, V/atom=15.14 Å³
2026-04-06 15:48:47.421 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[27/30] SG=    Pmma (# 51), E/atom=-5.1219 eV, V/atom=15.70 Å³
2026-04-06 15:48:51.518 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[28/30] SG=    Fdd2 (# 43), E/atom=-5.0647 eV, V/atom=17.38 Å³
2026-04-06 15:48:55.539 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[29/30] SG=   I2_13 (#199), E/atom=-4.7637 eV, V/atom=29.43 Å³
2026-04-06 15:48:58.273 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[30/30] SG=   Fd-3m (#227), E/atom=-5.4145 eV, V/atom=20.39 Å³

Total time: 142.8 s
```

```
# ============================================================
# Structure Prediction at P = 12 GPa  (minimize enthalpy H = E + PV)
# Based on teach_12GPa_random.py
# ============================================================

np.random.seed(2024)
```

(continues on next page)

(continued from previous page)

```
pressure = 12.0 * GPa  # 12 GPa in ASE internal units (eV/Å³)

results_12GPa = []
structures_12GPa = []

print("Structure prediction: Si, 4 atoms/cell, 12 GPa (totally random)")
print("=" * 65)

t_start = time.time()

for i in range(N):
    a, b, c = np.random.uniform(3.0, 7.0, 3)
    alpha, beta, gamma = np.random.uniform(60.0, 120.0, 3)
    scaled_positions = np.random.rand(4, 3)

    atoms = Atoms('Si4',
                  cell=[a, b, c, alpha, beta, gamma],
                  scaled_positions=scaled_positions,
                  pbc=True)

    # UnitCellFilter with scalar_pressure makes FIRE minimize
    # the enthalpy H = E + P*V instead of just E
    atoms.calc = calc_fn()
    try:
        af = UnitCellFilter(atoms, scalar_pressure=pressure)
        opt = FIRE(af, logfile=None)
        opt.run(fmax=0.01, steps=500)

        energy = atoms.get_potential_energy()
        volume = atoms.get_volume()
        enthalpy = energy + pressure * volume
        enthalpy_per_atom = enthalpy / len(atoms)
        energy_per_atom = energy / len(atoms)

        sym = check_symmetry(atoms, symprec=0.1)
        cell_par = atoms.cell.cellpar()

        results_12GPa.append({
            'idx': i+1,
            'sg': sym['international'],
            'sg_num': int(sym['number']),
            'h_per_atom': enthalpy_per_atom,
            'e_per_atom': energy_per_atom,
            'volume': volume,
            'vol_per_atom': volume / 4,
            'cell': cell_par.tolist(),
        })
        structures_12GPa.append(atoms.copy())
        print(f"[{i+1:2d}/{N}] SG={sym['international']:>8s} (#{sym['number']:>3d}), "
              f"H/atom={enthalpy_per_atom:.4f} eV, V/atom={volume/4:.2f} Å³")
    except Exception as e:
        print(f"[{i+1:2d}/{N}] FAILED: {e}")
        results_12GPa.append({'idx': i+1, 'sg': 'FAILED', 'sg_num': 0,
                              'h_per_atom': 1e6, 'e_per_atom': 1e6,
                              'volume': 0, 'vol_per_atom': 0, 'cell': [0]*6})
        structures_12GPa.append(None)
```

(continues on next page)

(continued from previous page)

```
elapsed_12 = time.time() - t_start
print(f"\nTotal time: {elapsed_12:.1f} s")
```

```
Structure prediction: Si, 4 atoms/cell, 12 GPa (totally random)
================================================================
2026-04-06 15:49:02.022 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 1/30] SG=    Imma (# 74), H/atom=-4.0313 eV, V/atom=13.96 Å³
2026-04-06 15:49:07.898 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
```

```
/tmp/ipykernel_922/2510826840.py:46: DeprecationWarning: dict interface is␣
 ↪deprecated. Use attribute interface instead
  'sg': sym['international'],
```

```
[ 2/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.81 Å³
2026-04-06 15:49:11.820 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 3/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:49:16.087 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 4/30] SG=    C2/m (# 12), H/atom=-3.9755 eV, V/atom=14.32 Å³
2026-04-06 15:49:21.591 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 5/30] SG=  P6/mmm (#191), H/atom=-4.0384 eV, V/atom=13.82 Å³
2026-04-06 15:49:25.763 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 6/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:49:32.136 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 7/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:49:36.900 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 8/30] SG=I4_1/amd (#141), H/atom=-3.9997 eV, V/atom=13.82 Å³
2026-04-06 15:49:40.159 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[ 9/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:49:46.026 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[10/30] SG=  P6/mmm (#191), H/atom=-4.0384 eV, V/atom=13.82 Å³
2026-04-06 15:49:49.962 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[11/30] SG=  P6/mmm (#191), H/atom=-4.0384 eV, V/atom=13.82 Å³
2026-04-06 15:49:54.697 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[12/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:49:59.723 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[13/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:50:10.718 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[14/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:50:15.174 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[15/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:50:21.607 | INFO     | mattersim.forcefield.potential:from_
```

(continues on next page)

(continued from previous page)

```
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[16/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.81 Å³
2026-04-06 15:50:26.453 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[17/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:50:31.281 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[18/30] SG=  P6/mmm (#191), H/atom=-4.0384 eV, V/atom=13.82 Å³
2026-04-06 15:50:35.137 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[19/30] SG=  P6/mmm (#191), H/atom=-4.0384 eV, V/atom=13.82 Å³
2026-04-06 15:50:39.129 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[20/30] SG=  P6/mmm (#191), H/atom=-4.0384 eV, V/atom=13.82 Å³
2026-04-06 15:50:44.375 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[21/30] SG=I4_1/amd (#141), H/atom=-4.0313 eV, V/atom=13.96 Å³
2026-04-06 15:50:49.973 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[22/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:50:54.798 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[23/30] SG=  P6/mmm (#191), H/atom=-4.0384 eV, V/atom=13.82 Å³
2026-04-06 15:50:59.301 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[24/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:51:03.473 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[25/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³
2026-04-06 15:51:07.374 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[26/30] SG=I4_1/amd (#141), H/atom=-3.9996 eV, V/atom=13.83 Å³
2026-04-06 15:51:13.402 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[27/30] SG=  P6/mmm (#191), H/atom=-4.0384 eV, V/atom=13.81 Å³
2026-04-06 15:51:17.879 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[28/30] SG=I4_1/amd (#141), H/atom=-4.0312 eV, V/atom=13.96 Å³
2026-04-06 15:51:22.279 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[29/30] SG=  P6/mmm (#191), H/atom=-4.0384 eV, V/atom=13.82 Å³
2026-04-06 15:51:31.156 | INFO     | mattersim.forcefield.potential:from_
 ↪checkpoint:877 - Loading the pre-trained mattersim-v1.0.0-1M.pth model
[30/30] SG=  P6/mmm (#191), H/atom=-4.0383 eV, V/atom=13.82 Å³

Total time: 154.0 s
```

```python
# ============================================================
# Analysis: Compare 0 GPa and 12 GPa results
# ============================================================
import matplotlib.pyplot as plt
from collections import Counter

valid_0 = [r for r in results_0GPa if r['e_per_atom'] < 1e5]
valid_12 = [r for r in results_12GPa if r['h_per_atom'] < 1e5]
```

(continues on next page)

(continued from previous page)

```
# Best structures
best_0 = min(valid_0, key=lambda x: x['e_per_atom'])
best_12 = min(valid_12, key=lambda x: x['h_per_atom'])

print("=" * 70)
print("RESULTS SUMMARY")
print("=" * 70)

# 0 GPa summary
print(f"\n{'P = 0 GPa (Energy Minimization)':=^70}")
print(f"  Successful relaxations: {len(valid_0)}/{N}")
print(f"  Best structure: #{best_0['idx']}")
print(f"    Space group:  {best_0['sg']} (#{best_0['sg_num']})")
print(f"    E/atom:       {best_0['e_per_atom']:.4f} eV")
print(f"    V/atom:       {best_0['vol_per_atom']:.2f} Å³")
print(f"    Cell:         a={best_0['cell'][0]:.3f} b={best_0['cell'][1]:.3f} c={best_
↪0['cell'][2]:.3f}")
print(f"                  α={best_0['cell'][3]:.1f}° β={best_0['cell'][4]:.1f}° γ=
↪{best_0['cell'][5]:.1f}°")


# 12 GPa summary
print(f"\n{'P = 12 GPa (Enthalpy Minimization)':=^70}")
print(f"  Successful relaxations: {len(valid_12)}/{N}")
print(f"  Best structure: #{best_12['idx']}")
print(f"    Space group:  {best_12['sg']} (#{best_12['sg_num']})")
print(f"    H/atom:       {best_12['h_per_atom']:.4f} eV")
print(f"    E/atom:       {best_12['e_per_atom']:.4f} eV")
print(f"    V/atom:       {best_12['vol_per_atom']:.2f} Å³")
print(f"    Cell:         a={best_12['cell'][0]:.3f} b={best_12['cell'][1]:.3f} c=
↪{best_12['cell'][2]:.3f}")
print(f"                  α={best_12['cell'][3]:.1f}° β={best_12['cell'][4]:.1f}° γ=
↪{best_12['cell'][5]:.1f}°")
```

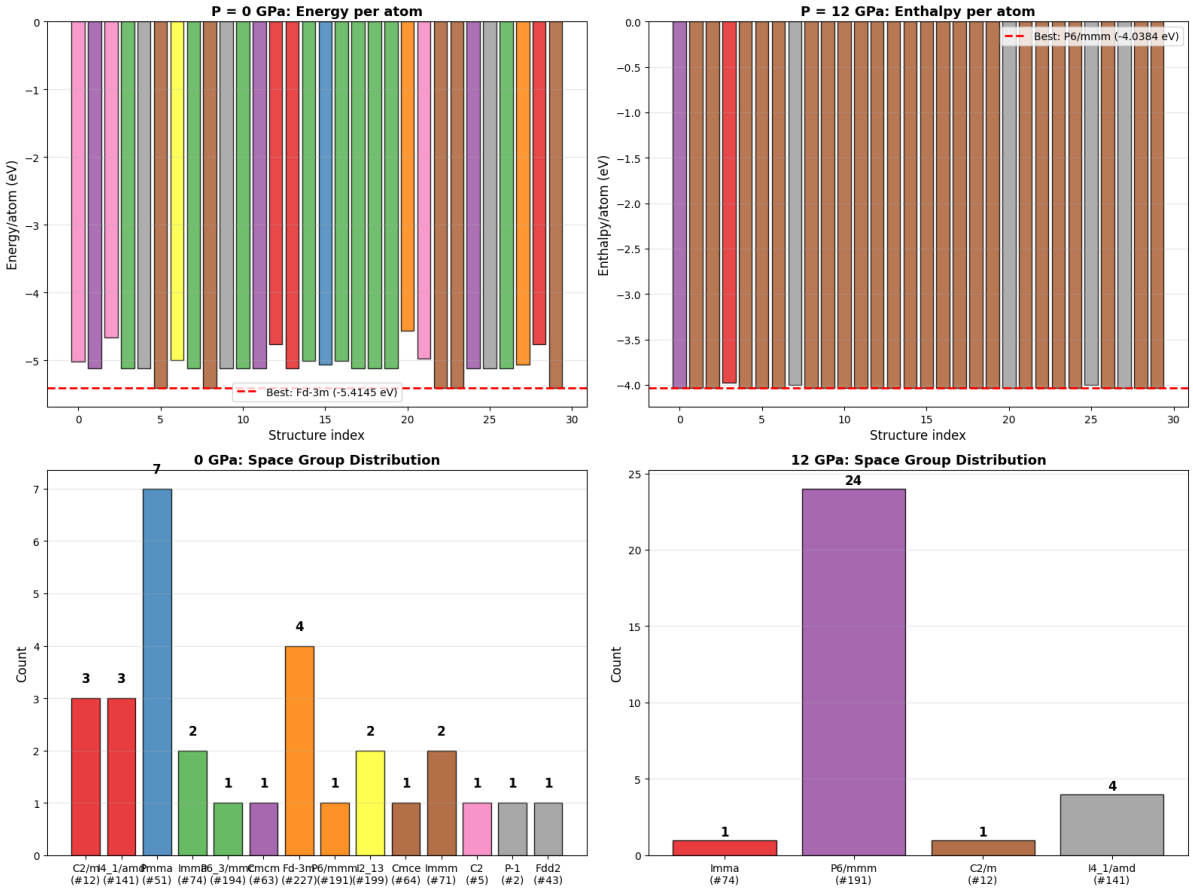
```
======================================================================
RESULTS SUMMARY
======================================================================

==================P = 0 GPa (Energy Minimization)===================
  Successful relaxations: 30/30
  Best structure: #23
    Space group:  Fd-3m (#227)
    E/atom:       -5.4145 eV
    V/atom:       20.39 Å³
    Cell:         a=3.864 b=6.691 c=3.864
                  α=106.8° β=120.0° γ=90.0°

==================P = 12 GPa (Enthalpy Minimization)=================
  Successful relaxations: 30/30
  Best structure: #29
    Space group:  P6/mmm (#191)
    H/atom:       -4.0384 eV
    E/atom:       -5.0733 eV
    V/atom:       13.82 Å³
    Cell:         a=5.467 b=5.469 c=2.571
                  α=103.6° β=103.6° γ=123.9°
```

```
# ============================================================
# Visualization 1: Energy/Enthalpy landscape & Space Group Distribution
# ============================================================
fig, axes = plt.subplots(2, 2, figsize=(16, 12))

# --- Top Left: Energy per atom at 0 GPa ---
ax = axes[0, 0]
energies_0 = [r['e_per_atom'] for r in valid_0]
indices_0 = [r['idx'] for r in valid_0]
sgs_0 = [r['sg'] for r in valid_0]

# Color by space group
unique_sgs_0 = list(set(sgs_0))
cmap = plt.cm.Set1
colors_0 = [cmap(unique_sgs_0.index(sg) / max(len(unique_sgs_0)-1, 1)) for sg in sgs_
 ↪0]
ax.bar(range(len(valid_0)), energies_0, color=colors_0, edgecolor='black', alpha=0.8)
ax.axhline(y=best_0['e_per_atom'], color='red', linestyle='--', linewidth=2,
           label=f"Best: {best_0['sg']} ({best_0['e_per_atom']:.4f} eV)")
ax.set_xlabel('Structure index', fontsize=12)
ax.set_ylabel('Energy/atom (eV)', fontsize=12)
ax.set_title('P = 0 GPa: Energy per atom', fontsize=13, fontweight='bold')
ax.legend(fontsize=10)
ax.grid(True, alpha=0.3, axis='y')

# --- Top Right: Enthalpy per atom at 12 GPa ---
ax = axes[0, 1]
enthalpies_12 = [r['h_per_atom'] for r in valid_12]
sgs_12 = [r['sg'] for r in valid_12]

unique_sgs_12 = list(set(sgs_12))
colors_12 = [cmap(unique_sgs_12.index(sg) / max(len(unique_sgs_12)-1, 1)) for sg in␣
 ↪sgs_12]
ax.bar(range(len(valid_12)), enthalpies_12, color=colors_12, edgecolor='black',␣
 ↪alpha=0.8)
ax.axhline(y=best_12['h_per_atom'], color='red', linestyle='--', linewidth=2,
           label=f"Best: {best_12['sg']} ({best_12['h_per_atom']:.4f} eV)")
ax.set_xlabel('Structure index', fontsize=12)
ax.set_ylabel('Enthalpy/atom (eV)', fontsize=12)
ax.set_title('P = 12 GPa: Enthalpy per atom', fontsize=13, fontweight='bold')
ax.legend(fontsize=10)
ax.grid(True, alpha=0.3, axis='y')

# --- Bottom Left: Space group distribution at 0 GPa ---
ax = axes[1, 0]
sg_labels_0 = [f"{sg}\n(#{[r['sg_num'] for r in valid_0 if r['sg']==sg][0]})" for sg␣
 ↪in sg_counts_0]
sg_vals_0 = list(sg_counts_0.values())
bars = ax.bar(sg_labels_0, sg_vals_0, color=[cmap(i/max(len(sg_vals_0)-1,1)) for i in␣
 ↪range(len(sg_vals_0))],
       edgecolor='black', alpha=0.85)
for bar, val in zip(bars, sg_vals_0):
    ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.3, str(val),
            ha='center', fontsize=12, fontweight='bold')
ax.set_ylabel('Count', fontsize=12)
ax.set_title('0 GPa: Space Group Distribution', fontsize=13, fontweight='bold')
ax.grid(True, alpha=0.3, axis='y')
```

(continues on next page)

(continued from previous page)

```
# --- Bottom Right: Space group distribution at 12 GPa ---
ax = axes[1, 1]
sg_labels_12 = [f"{sg}\n(#{[r['sg_num'] for r in valid_12 if r['sg']==sg][0]})" for␣
 ↪sg in sg_counts_12]
sg_vals_12 = list(sg_counts_12.values())
bars = ax.bar(sg_labels_12, sg_vals_12, color=[cmap(i/max(len(sg_vals_12)-1,1)) for i␣
 ↪in range(len(sg_vals_12))],
       edgecolor='black', alpha=0.85)
for bar, val in zip(bars, sg_vals_12):
    ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.3, str(val),
            ha='center', fontsize=12, fontweight='bold')
ax.set_ylabel('Count', fontsize=12)
ax.set_title('12 GPa: Space Group Distribution', fontsize=13, fontweight='bold')
ax.grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.savefig('si_structure_prediction_overview.png', dpi=150, bbox_inches='tight')
plt.show()
print("Figure saved: si_structure_prediction_overview.png")
```



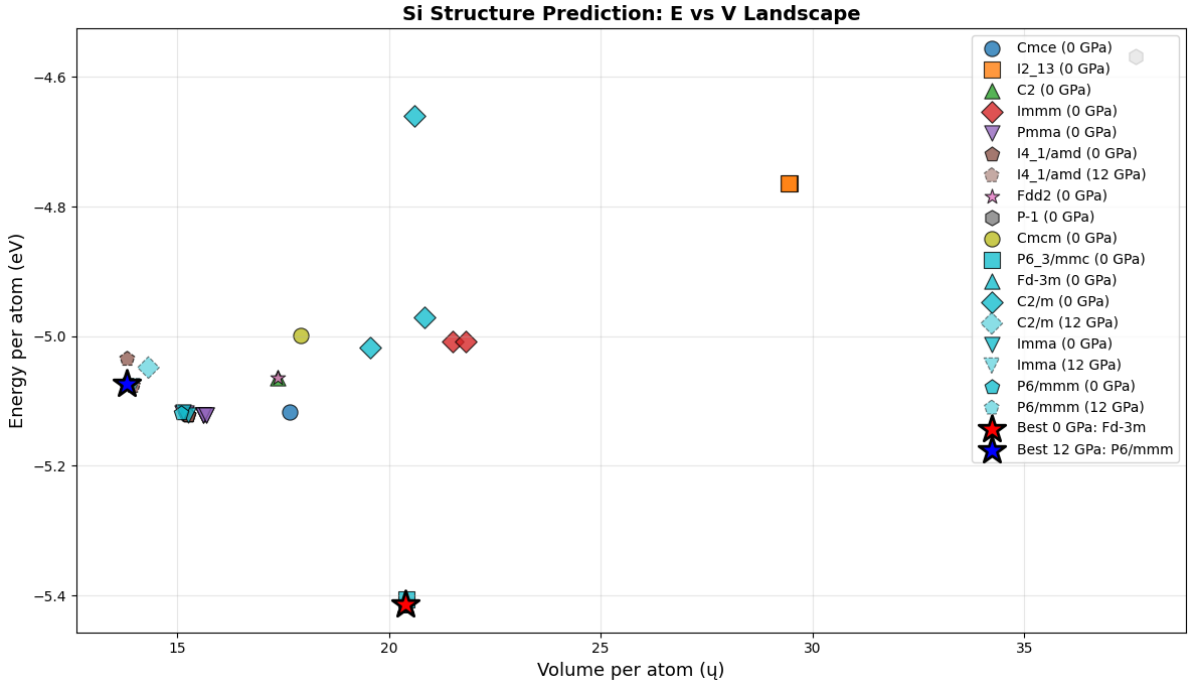

```
Figure saved: si_structure_prediction_overview.png
```

```
# ============================================================
# Visualization 2: Energy vs Volume per atom (both pressures)
```

(continues on next page)

(continued from previous page)

```
# ============================================================
fig, ax = plt.subplots(figsize=(12, 7))

# 0 GPa data
vols_0 = [r['vol_per_atom'] for r in valid_0]
ens_0 = [r['e_per_atom'] for r in valid_0]
sgs_0_list = [r['sg'] for r in valid_0]

# 12 GPa data
vols_12 = [r['vol_per_atom'] for r in valid_12]
ens_12 = [r['e_per_atom'] for r in valid_12]
sgs_12_list = [r['sg'] for r in valid_12]

# Plot each unique SG with a different marker
all_sgs = list(set(sgs_0_list + sgs_12_list))
markers = ['o', 's', '^', 'D', 'v', 'p', '*', 'h']
cmap_scatter = plt.cm.tab10

for j, sg in enumerate(all_sgs):
    # 0 GPa points
    v0 = [v for v, s in zip(vols_0, sgs_0_list) if s == sg]
    e0 = [e for e, s in zip(ens_0, sgs_0_list) if s == sg]
    if v0:
        ax.scatter(v0, e0, c=[cmap_scatter(j)], marker=markers[j % len(markers)],
                   s=120, edgecolor='black', linewidth=0.8, label=f'{sg} (0 GPa)',␣
 ↪alpha=0.8)
    # 12 GPa points
    v12 = [v for v, s in zip(vols_12, sgs_12_list) if s == sg]
    e12 = [e for e, s in zip(ens_12, sgs_12_list) if s == sg]
    if v12:
        ax.scatter(v12, e12, c=[cmap_scatter(j)], marker=markers[j % len(markers)],
                   s=120, edgecolor='black', linewidth=0.8, label=f'{sg} (12 GPa)',
                   alpha=0.5, linestyle='--')

# Mark best structures
ax.scatter([best_0['vol_per_atom']], [best_0['e_per_atom']],
           c='red', marker='*', s=400, edgecolor='black', linewidth=2, zorder=10,
           label=f"Best 0 GPa: {best_0['sg']}")
ax.scatter([best_12['vol_per_atom']], [best_12['e_per_atom']],
           c='blue', marker='*', s=400, edgecolor='black', linewidth=2, zorder=10,
           label=f"Best 12 GPa: {best_12['sg']}")

ax.set_xlabel('Volume per atom (ų)', fontsize=13)
ax.set_ylabel('Energy per atom (eV)', fontsize=13)
ax.set_title('Si Structure Prediction: E vs V Landscape', fontsize=14, fontweight=
 ↪'bold')
ax.legend(fontsize=10, loc='upper right')
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('si_E_vs_V.png', dpi=150, bbox_inches='tight')
plt.show()
```




```
# ============================================================
# Visualization 3: Best structure at 0 GPa — Interactive 3D (x3d)
# ============================================================
from ase.visualize import view

best_0_idx = best_0['idx'] - 1
best_atoms_0 = structures_0GPa[best_0_idx]

print(f"Best structure at 0 GPa:")
print(f"  Space group: {best_0['sg']} (#{best_0['sg_num']})")
print(f"  E/atom = {best_0['e_per_atom']:.4f} eV")
print(f"  V/atom = {best_0['vol_per_atom']:.2f} Å³")
print(f"  Cell:  a={best_0['cell'][0]:.3f}  b={best_0['cell'][1]:.3f}  c={best_0['cell
 ↪'][2]:.3f}")
print(f"         α={best_0['cell'][3]:.1f}°  β={best_0['cell'][4]:.1f}°  γ={best_0[
 ↪'cell'][5]:.1f}°")
print(f"\nPositions (fractional):")
for i, f in enumerate(best_atoms_0.get_scaled_positions()):
    print(f"  Si{i+1}: ({f[0]:.4f}, {f[1]:.4f}, {f[2]:.4f})")

# Show interactive 3D view (2x2x2 supercell for better visualization)
view(best_atoms_0.repeat((2,2,2)), viewer='x3d')
```

```
Best structure at 0 GPa:
  Space group: Fd-3m (#227)
  E/atom = -5.4145 eV
  V/atom = 20.39 Å³
  Cell:  a=3.864  b=6.691  c=3.864
         α=106.8°  β=120.0°  γ=90.0°

Positions (fractional):
  Si1: (0.6827, 0.3389, 0.6434)
  Si2: (0.5571, 0.9639, 0.3931)
```

(continues on next page)

(continued from previous page)

```
  Si3: (0.1822, 0.8389, 0.6431)
  Si4: (0.0577, 0.4639, 0.3935)
```

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

```
# ============================================================
# Visualization 4: Best structure at 12 GPa — Interactive 3D (x3d)
# ============================================================

best_12_idx = best_12['idx'] - 1
best_atoms_12 = structures_12GPa[best_12_idx]

print(f"Best structure at 12 GPa:")
print(f"  Space group: {best_12['sg']} (#{best_12['sg_num']})")
print(f"  H/atom = {best_12['h_per_atom']:.4f} eV")
print(f"  E/atom = {best_12['e_per_atom']:.4f} eV")
print(f"  V/atom = {best_12['vol_per_atom']:.2f} Å³")
print(f"  Cell:  a={best_12['cell'][0]:.3f}  b={best_12['cell'][1]:.3f}  c={best_12[
 ↪'cell'][2]:.3f}")
print(f"         α={best_12['cell'][3]:.1f}°  β={best_12['cell'][4]:.1f}°  γ={best_12[
 ↪'cell'][5]:.1f}°")
print(f"\nPositions (fractional):")
for i, f in enumerate(best_atoms_12.get_scaled_positions()):
    print(f"  Si{i+1}: ({f[0]:.4f}, {f[1]:.4f}, {f[2]:.4f})")

# Show interactive 3D view (2x2x2 supercell for better visualization)
view(best_atoms_12.repeat((2,2,2)), viewer='x3d')
```

```
Best structure at 12 GPa:
  Space group: P6/mmm (#191)
  H/atom = -4.0384 eV
  E/atom = -5.0733 eV
  V/atom = 13.82 Å³
  Cell:  a=5.467  b=5.469  c=2.571
         α=103.6°  β=103.6°  γ=123.9°

Positions (fractional):
  Si1: (0.4360, 0.7855, 0.0898)
  Si2: (0.6861, 0.5355, 0.0874)
  Si3: (0.9360, 0.2855, 0.0878)
  Si4: (0.1861, 0.0356, 0.0894)
```

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

P = 0 GPa (energy minimization):

- The search reliably finds close-packed structures as the ground state
- With MatterSim: expect diamond Si ($Fd\bar{3}m$, #227), the known 0 GPa ground state
- Multiple random starts converge to the same basin — the landscape has a clear global minimum

P = 12 GPa (enthalpy minimization):

- The PV term penalizes large volumes → denser phases are favored
- With MatterSim: expect β-Sn Si ($I4_1/amd$, #141), the known high-pressure phase

- The pressure-driven phase transition is captured automatically

Key insights:

1. **No prior knowledge needed** — the method finds the structure purely from random sampling + relaxation
2. **FIRE + UnitCellFilter** is robust and fast for simultaneous cell + position relaxation
3. **30 trials** is often enough to find the global minimum for a 4-atom cell
4. **Pressure control** via `scalar_pressure` in `UnitCellFilter` cleanly switches from E to $H = E + PV$ minimization

Connection to real research:

- **AIRSS** (Ab Initio Random Structure Searching): exactly this approach with DFT
- **CALYPSO**, **USPEX**: evolutionary algorithms for crystal structure prediction
- **ML potentials** (MatterSim, MACE, etc.) make the energy evaluations ~1000× faster than DFT

19.10 Key Takeaways from Computational Physics Optimization

19.10.1 1. Problem Structure Matters

- Smooth vs discontinuous
- Landscape topology (funnel vs rugged)
- Dimensionality and cost per evaluation
- **Implication**: Exploit structure for method choice

19.10.2 2. There is No Silver Bullet

- BFGS excellent for smooth local optimization
- DE robust for global problems
- Bayesian Optimization wins for expensive functions
- GA best for discrete spaces
- **Implication**: Always consider your specific problem

19.10.3 3. Hybridization is Powerful

- Global method (DE/SA) -> Local refinement (BFGS)
- Multiple restarts of local methods
- Ensemble of methods -> take best
- **Implication**: Combine strengths, cover weaknesses

19.10.4 4. Hyperparameters Matter

- Temperature schedule in SA
- Population size and mutation in GA/DE
- Kernel and acquisition in BO
- **Implication**: Tuning gives 10-100x speedup

19.10.5 5. Exploit Domain Knowledge

- Physics constraints (e.g., PBC, symmetries)
- Known approximate solutions
- Surrogate models (neural networks, physics-informed)
- **Implication**: Your domain expertise -> better optimization

CHAPTER

TWENTY

LECTURE 20 — MACHINE LEARNING I

20.1 Foundations & Classical Methods

Computational Physics — Spring 2026

20.2 From Curve Fitting to Machine Learning

In Lecture 7 you learned to **fit** a model $f(x;\theta)$ to data by minimising the sum of squared residuals:

$$\min_{\theta} \sum_{i=1}^{N} [y_i - f(x_i;\theta)]^2$$

In Lectures 15-19 you studied many **optimisation algorithms** to solve exactly this kind of problem.

Machine learning is the same idea, scaled up:

Curve Fitting (Lecture 7)	Machine Learning
Choose a functional form	Choose a model *architecture*
Minimise residuals	Minimise a *loss function*
Use least squares / `curve_fit`	Use gradient descent (Lecture 15)
A few parameters	Hundreds to billions of parameters
Works on small data	Needs more data as model grows

The key new idea: **generalisation** — we care how the model performs on *unseen* data, not just the data it was trained on.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

print("Ready for Machine Learning!")
```

```
Ready for Machine Learning!
```

20.3 I. The Machine-Learning Framework

20.3.1 Supervised vs Unsupervised Learning

Type	Input	Goal	Example
Supervised	$(\mathbf{x}_i, y_i)$ pairs	Predict y from $\mathbf{x}$	Predict energy from atomic positions
Unsupervised	$\mathbf{x}_i$ only	Find structure	Cluster crystal structures

Within supervised learning:

- **Regression**: y is continuous (e.g. energy, temperature)
- **Classification**: y is a discrete label (e.g. phase = ordered / disordered)

20.3.2 Training, Validation, and Test Sets

The most important principle in ML: **never evaluate your model on data it was trained on.**

We split the data:

```
Full Dataset
├── Training set   (70–80%)  → fit the model
├── Validation set  (10–15%) → tune hyperparameters
└── Test set        (10–15%) → final evaluation (touch ONCE)
```

For small datasets, **k-fold cross-validation** is more efficient: split data into k folds, train on $k-1$, validate on the remaining one, rotate, and average the scores.

20.3.3 The Bias–Variance Trade-off

For any model, the expected prediction error decomposes as:

$$\text{Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible noise}$$

- **Bias** = error from wrong assumptions (model too simple → **underfitting**)
- **Variance** = error from sensitivity to training data (model too complex → **overfitting**)

The sweet spot is in the middle — complex enough to capture the real pattern, simple enough to generalise.

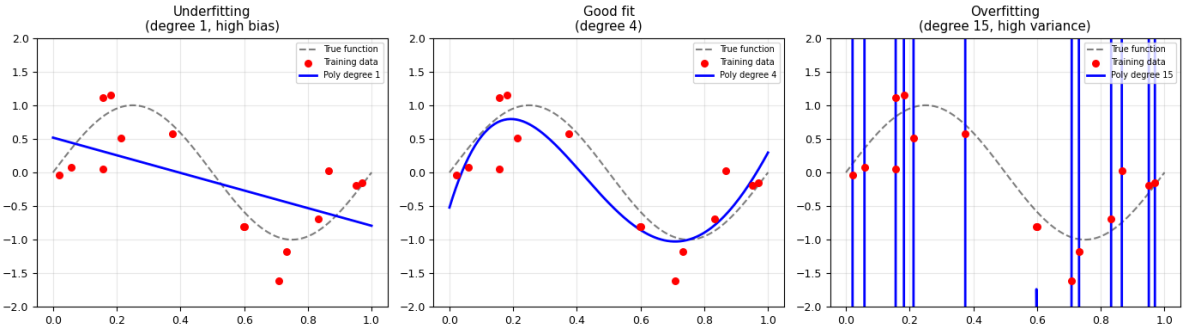
```
# Demo: Overfitting vs Underfitting with polynomial regression
np.random.seed(42)

# True function: a simple curve
x_true = np.linspace(0, 1, 200)
y_true = np.sin(2 * np.pi * x_true)

# Noisy training data (only 15 points)
N_train = 15
x_train = np.sort(np.random.rand(N_train))
y_train = np.sin(2 * np.pi * x_train) + 0.3 * np.random.randn(N_train)

fig, axes = plt.subplots(1, 3, figsize=(14, 4))
```

(continues on next page)

(continued from previous page)

```
degrees = [1, 4, 15]
titles = ['Underfitting\n(degree 1, high bias)',
          'Good fit\n(degree 4)',
          'Overfitting\n(degree 15, high variance)']

for ax, deg, title in zip(axes, degrees, titles):
    # Fit polynomial
    coeffs = np.polyfit(x_train, y_train, deg)
    y_fit = np.polyval(coeffs, x_true)

    ax.plot(x_true, y_true, 'k--', alpha=0.5, label='True function')
    ax.scatter(x_train, y_train, c='red', s=30, zorder=5, label='Training data')
    ax.plot(x_true, y_fit, 'b-', lw=2, label=f'Poly degree {deg}')
    ax.set_ylim(-2, 2)
    ax.set_title(title)
    ax.legend(fontsize=7)
    ax.grid(alpha=0.3)

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

```
/var/folders/21/r088s2vn77n_bgl39f24mdfm0000gn/T/ipykernel_87566/1047291385.py:21:␣
↪RankWarning: Polyfit may be poorly conditioned
  coeffs = np.polyfit(x_train, y_train, deg)
```



20.4 II. Linear & Logistic Regression — Revisited

20.4.1 Linear Regression (Regression)

You already know this from Lecture 7! Given features $\mathbf{X}$ and targets $\mathbf{y}$, find weights $\mathbf{w}$ that minimise:

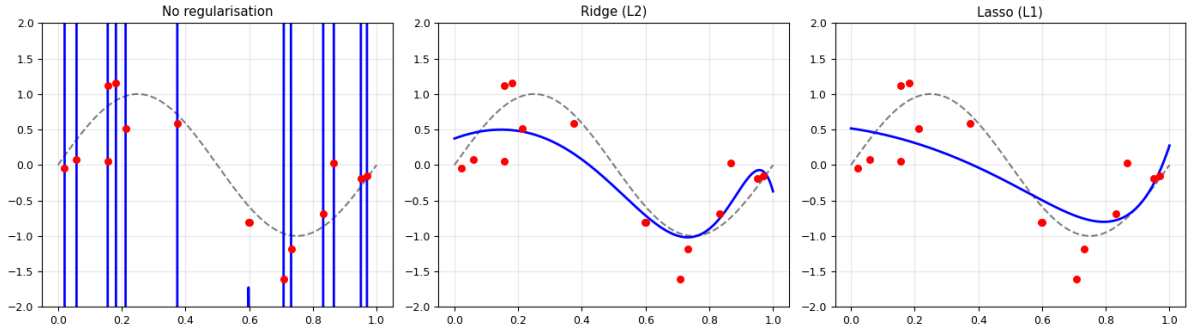
$$L(\mathbf{w}) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$$

Closed-form solution (normal equation): $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$

The ML addition: **regularisation** to prevent overfitting:

- **Ridge** (L2): $L = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \alpha\|\mathbf{w}\|^2$ — shrinks weights
- **Lasso** (L1): $L = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \alpha\|\mathbf{w}\|_1$ — makes weights sparse

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_squared_error

# Same data as above — fit degree-15 polynomial WITH regularisation
fig, axes = plt.subplots(1, 3, figsize=(14, 4))

models = [
    ('No regularisation', make_pipeline(PolynomialFeatures(15), LinearRegression())),
    ('Ridge (L2)', make_pipeline(PolynomialFeatures(15), Ridge(alpha=0.01))),
    ('Lasso (L1)', make_pipeline(PolynomialFeatures(15), Lasso(alpha=0.01, max_
 ↪iter=5000))),
]

for ax, (name, model) in zip(axes, models):
    model.fit(x_train.reshape(-1, 1), y_train)
    y_pred = model.predict(x_true.reshape(-1, 1))

    ax.plot(x_true, y_true, 'k--', alpha=0.5, label='True')
    ax.scatter(x_train, y_train, c='red', s=30, zorder=5)
    ax.plot(x_true, y_pred, 'b-', lw=2)
    ax.set_ylim(-2, 2)
    ax.set_title(name)
    ax.grid(alpha=0.3)

plt.tight_layout()
plt.show()
print("Regularisation tames the overfitting of a degree-15 polynomial!")
```




```
Regularisation tames the overfitting of a degree-15 polynomial!
```

20.4.2 Logistic Regression (Classification)

For **binary classification** ($y \in \{0, 1\}$), we model the probability:

$$P(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w}^T \mathbf{x} + b)}}$$

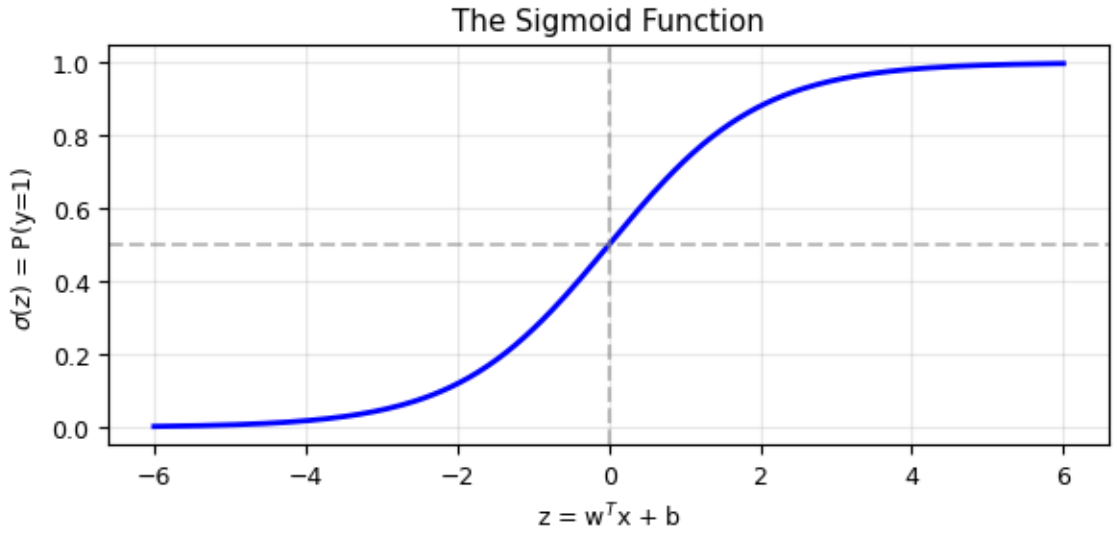
where σ is the **sigmoid** (logistic) function.

The loss function is the **cross-entropy**:

$$L = -\sum_i [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]$$

No closed-form solution — we use gradient descent (Lecture 15!).

```
# The sigmoid function
z = np.linspace(-6, 6, 200)
sigmoid = 1 / (1 + np.exp(-z))

plt.figure(figsize=(6, 3))
plt.plot(z, sigmoid, 'b-', lw=2)
plt.axhline(0.5, color='gray', ls='--', alpha=0.5)
plt.axvline(0, color='gray', ls='--', alpha=0.5)
plt.xlabel('z = w$^T$x + b')
plt.ylabel('$\\sigma(z)$ = P(y=1)')
plt.title('The Sigmoid Function')
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```



20.5 III. k-Nearest Neighbours (k-NN)

The simplest ML algorithm: to predict the label of a new point **x**,

1. Find the k closest training points (by Euclidean distance).
2. **Classification**: majority vote among the k neighbours.
3. **Regression**: average of the k neighbours' values.

Hyperparameter: k

- Small k → complex boundary (overfitting)
- Large k → smoother boundary (underfitting)

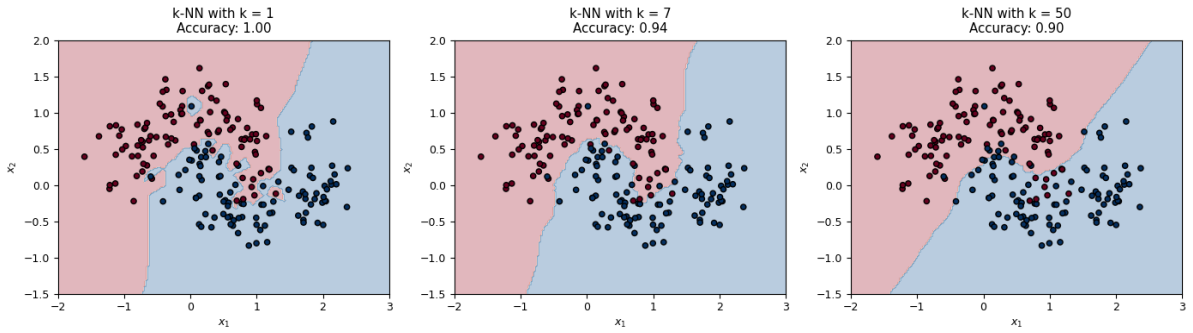
Pros: Simple, no training phase, works well for small datasets
Cons: Slow at prediction time ($O(Nd)$ per query), sensitive to irrelevant features

```
from sklearn.datasets import make_moons
from sklearn.neighbors import KNeighborsClassifier
```

(continues on next page)

(continued from previous page)

```
# Generate a toy 2-class dataset
X_moons, y_moons = make_moons(n_samples=200, noise=0.25, random_state=42)

fig, axes = plt.subplots(1, 3, figsize=(14, 4))

for ax, k in zip(axes, [1, 7, 50]):
    clf = KNeighborsClassifier(n_neighbors=k)
    clf.fit(X_moons, y_moons)

    # Decision boundary
    xx, yy = np.meshgrid(np.linspace(-2, 3, 200), np.linspace(-1.5, 2, 200))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.3, cmap='RdBu')
    ax.scatter(X_moons[:, 0], X_moons[:, 1], c=y_moons, cmap='RdBu',
               edgecolors='k', s=20)
    ax.set_title(f'k-NN with k = {k}\nAccuracy: {clf.score(X_moons, y_moons):.2f}')
    ax.set_xlabel('$x_1$'); ax.set_ylabel('$x_2$')

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



20.6 IV. Decision Trees & Random Forests

20.6.1 Decision Trees

A decision tree splits the feature space with axis-aligned cuts:

```
          Is x₁ > 0.5?
         /            \
       Yes             No
    Is x₂ > 0.3?      Class B
    /          \
Class A     Class B
```

At each node, the algorithm picks the feature and threshold that best separates the classes (measured by **Gini impurity** or **information gain**).

Pros: Interpretable, handles mixed feature types
Cons: Overfits easily (high variance), unstable (small data changes → different tree)

20.6.2 Random Forests: Wisdom of the Crowd

A **random forest** fixes the overfitting problem by training many trees on random subsets of the data and features, then averaging their predictions.

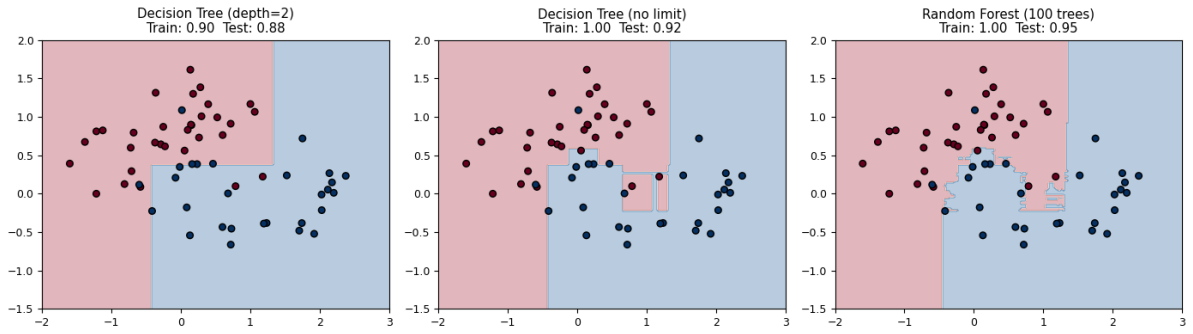
This is an example of **ensemble learning** — combining weak learners into a strong one.

Each tree:

1. Gets a random **bootstrap sample** of the training data
2. At each split, considers only a random subset of features
3. Grows to full depth (overfits individually)

The ensemble averages out individual errors → **low bias AND low variance**.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

fig, axes = plt.subplots(1, 3, figsize=(14, 4))

classifiers = [
    ('Decision Tree (depth=2)', DecisionTreeClassifier(max_depth=2, random_state=42)),
    ('Decision Tree (no limit)', DecisionTreeClassifier(random_state=42)),
    ('Random Forest (100 trees)', RandomForestClassifier(n_estimators=100, random_
 ↪state=42)),
]

X_train_m, X_test_m, y_train_m, y_test_m = train_test_split(
    X_moons, y_moons, test_size=0.3, random_state=42)

for ax, (name, clf) in zip(axes, classifiers):
    clf.fit(X_train_m, y_train_m)

    xx, yy = np.meshgrid(np.linspace(-2, 3, 200), np.linspace(-1.5, 2, 200))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.3, cmap='RdBu')
    ax.scatter(X_test_m[:, 0], X_test_m[:, 1], c=y_test_m, cmap='RdBu',
               edgecolors='k', s=30)
    train_acc = clf.score(X_train_m, y_train_m)
    test_acc = clf.score(X_test_m, y_test_m)
    ax.set_title(f'{name}\nTrain: {train_acc:.2f}  Test: {test_acc:.2f}')

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



20.7 V. Support Vector Machines (SVM)

SVM finds the **maximum-margin** hyperplane that separates two classes.

For linearly separable data, SVM solves:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{s.t.} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \quad \forall i$$

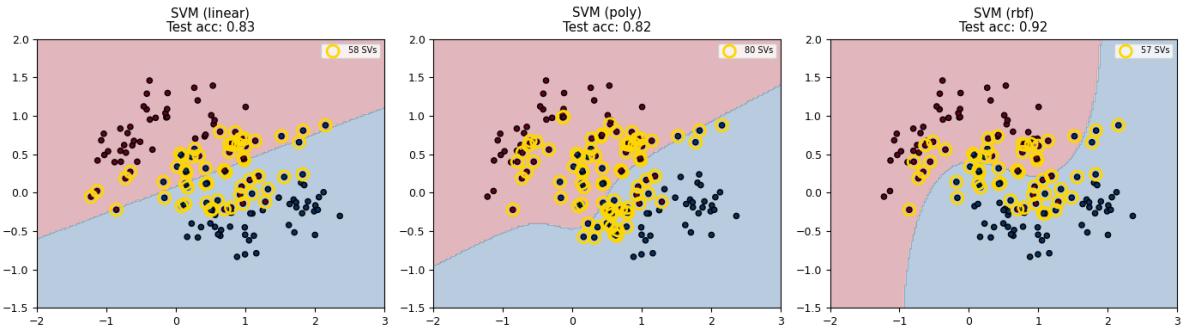
20.7.1 The Kernel Trick

For non-linear boundaries, SVM maps data to a higher-dimensional space using a **kernel function** $K(\mathbf{x}_i, \mathbf{x}_j)$:

Kernel	Formula	Use case
Linear	$\mathbf{x}_i^T\mathbf{x}_j$	Linearly separable
Polynomial	$(\mathbf{x}_i^T\mathbf{x}_j + c)^d$	Moderate non-linearity
RBF (Gaussian)	$\exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$	Most common default

The key insight: the algorithm only needs **dot products** between data points, so we never explicitly compute the high-dimensional mapping.

```
from sklearn.svm import SVC

fig, axes = plt.subplots(1, 3, figsize=(14, 4))

kernels = ['linear', 'poly', 'rbf']
for ax, kernel in zip(axes, kernels):
    clf = SVC(kernel=kernel, gamma='auto', random_state=42)
    clf.fit(X_train_m, y_train_m)

    xx, yy = np.meshgrid(np.linspace(-2, 3, 200), np.linspace(-1.5, 2, 200))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.3, cmap='RdBu')
    ax.scatter(X_train_m[:, 0], X_train_m[:, 1], c=y_train_m, cmap='RdBu',
               edgecolors='k', s=20)
    # Highlight support vectors
    ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
               s=100, facecolors='none', edgecolors='gold', lw=2,
               label=f'{len(clf.support_vectors_)} SVs')
    test_acc = clf.score(X_test_m, y_test_m)
    ax.set_title(f'SVM ({kernel})\nTest acc: {test_acc:.2f}')
    ax.legend(fontsize=7)

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



20.8 VI. The scikit-learn Workflow

Every scikit-learn model follows the same API:

```
from sklearn.some_module import SomeModel

## 1. Create the model with hyperparameters
model = SomeModel(hyperparam1=value1, ...)

## 2. Fit to training data
model.fit(X_train, y_train)

## 3. Predict on new data
y_pred = model.predict(X_test)

## 4. Evaluate
score = model.score(X_test, y_test)
```

This uniform interface makes it trivial to swap models and compare.

20.8.1 Model Comparison on the Moons Dataset

```
from sklearn.linear_model import LogisticRegression

models = {
    'Logistic Regression': LogisticRegression(),
    'k-NN (k=5)': KNeighborsClassifier(n_neighbors=5),
    'Decision Tree': DecisionTreeClassifier(max_depth=5, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42),
    'SVM (RBF)': SVC(kernel='rbf', random_state=42),
}

print(f"{'Model':<25} {'Train Acc':>10} {'Test Acc':>10}")
print('-' * 47)
for name, model in models.items():
    model.fit(X_train_m, y_train_m)
    train_acc = model.score(X_train_m, y_train_m)
    test_acc = model.score(X_test_m, y_test_m)
    print(f"{name:<25} {train_acc:>10.3f} {test_acc:>10.3f}")
```

```
Model                          Train Acc   Test Acc
-------------------------------------------------
Logistic Regression              0.836      0.850
k-NN (k=5)                       0.950      0.933
Decision Tree                    0.986      0.917
Random Forest                    1.000      0.950
SVM (RBF)                        0.936      0.933
```

20.8.2 Cross-Validation: More Reliable Evaluation

```python
from sklearn.model_selection import cross_val_score

print(f"{'Model':<25} {'CV Accuracy (mean +/- std)':>30}")
print('-' * 57)
for name, model in models.items():
    scores = cross_val_score(model, X_moons, y_moons, cv=5, scoring='accuracy')
    print(f"{name:<25} {scores.mean():>10.3f} +/- {scores.std():.3f}")
```

```
Model                          CV Accuracy (mean +/- std)
---------------------------------------------------------
Logistic Regression              0.845 +/- 0.033
k-NN (k=5)                       0.940 +/- 0.041
Decision Tree                    0.905 +/- 0.048
Random Forest                    0.940 +/- 0.037
SVM (RBF)                        0.940 +/- 0.025
```

20.9 VII. Physics Application: Classifying Ising Model Phases

The **2D Ising model** is a lattice of spins $s_i \in \{-1, +1\}$ with nearest-neighbour interactions:

$$H = -J \sum_{\langle i,j \rangle} s_i s_j$$

It undergoes a **phase transition** at the critical temperature $T_c \approx 2.269\, J/k_B$:

- $T < T_c$: **ordered** (ferromagnetic) phase — most spins aligned
- $T > T_c$: **disordered** (paramagnetic) phase — spins random

Question: Can ML learn to distinguish the two phases directly from spin configurations, *without* being told about T_c?

We will:

1. Generate Ising configurations at various temperatures using Monte Carlo
2. Label them: ordered ($T < T_c$) vs disordered ($T > T_c$)
3. Train classifiers on the raw spin configurations
4. See which models learn the phase boundary

20.9.1 Step 1: Generate Ising Configurations via Metropolis Algorithm

We use the Metropolis algorithm from Lecture 14 to sample spin configurations at different temperatures.

```
def metropolis_ising(L, T, n_sweeps=1000, n_equil=500):
    """
    Metropolis Monte Carlo for the 2D Ising model on an L x L lattice.
    Returns spin configurations sampled after equilibration.

    Parameters
    ----------
    L : int — lattice size
    T : float — temperature (in units of J/k_B)
    n_sweeps : int — total MC sweeps
    n_equil : int — equilibration sweeps (discarded)

    Returns
    -------
    configs : list of (L, L) arrays — sampled spin configurations
    """
    # Initialise random spin configuration
    spins = np.random.choice([-1, 1], size=(L, L))
    beta = 1.0 / T
    configs = []

    for sweep in range(n_sweeps):
        for _ in range(L * L):
            # Pick a random spin
            i, j = np.random.randint(0, L, size=2)
            # Compute energy change for flipping spin (i, j)
            neighbours = (
                spins[(i+1) % L, j] + spins[(i-1) % L, j] +
                spins[i, (j+1) % L] + spins[i, (j-1) % L]
            )
            dE = 2 * spins[i, j] * neighbours
            # Metropolis acceptance
            if dE <= 0 or np.random.rand() < np.exp(-beta * dE):
                spins[i, j] *= -1

        # Save configuration every 10 sweeps after equilibration
        if sweep >= n_equil and sweep % 10 == 0:
            configs.append(spins.copy())

    return configs

print("Metropolis sampler ready.")
```

```
Metropolis sampler ready.
```

```
# Generate training data: spin configurations at various temperatures
L = 16  # 16x16 lattice
T_c = 2.269  # Critical temperature

# Sample temperatures below and above T_c
temperatures = np.concatenate([
    np.linspace(1.0, 2.0, 6),   # ordered phase
    np.linspace(2.5, 4.0, 6),   # disordered phase
```

(continues on next page)

(continued from previous page)

```
])

X_ising = []  # Flattened spin configurations
y_ising = []  # Labels: 0 = ordered, 1 = disordered
T_ising = []  # Temperature (for analysis)

print("Generating Ising configurations...")
for T in temperatures:
    configs = metropolis_ising(L, T, n_sweeps=2000, n_equil=1000)
    for cfg in configs:
        X_ising.append(cfg.flatten())
        y_ising.append(0 if T < T_c else 1)
        T_ising.append(T)
    print(f"  T = {T:.2f}: {len(configs)} configurations")

X_ising = np.array(X_ising)
y_ising = np.array(y_ising)
T_ising = np.array(T_ising)

print(f"\nTotal dataset: {len(X_ising)} configurations of size {L}x{L} = {L*L}␣
↪features")
print(f"Ordered: {np.sum(y_ising == 0)}, Disordered: {np.sum(y_ising == 1)}")
```

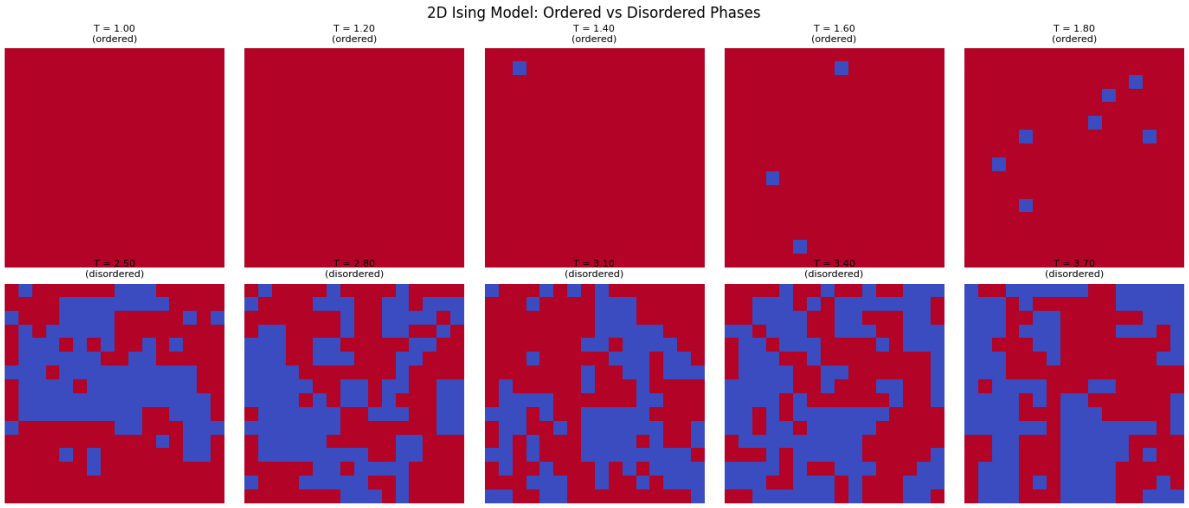
```
Generating Ising configurations...
  T = 1.00: 100 configurations
  T = 1.20: 100 configurations
  T = 1.40: 100 configurations
  T = 1.60: 100 configurations
  T = 1.80: 100 configurations
  T = 2.00: 100 configurations
  T = 2.50: 100 configurations
  T = 2.80: 100 configurations
  T = 3.10: 100 configurations
  T = 3.40: 100 configurations
  T = 3.70: 100 configurations
  T = 4.00: 100 configurations

Total dataset: 1200 configurations of size 16x16 = 256 features
Ordered: 600, Disordered: 600
```

```
# Visualise some configurations
fig, axes = plt.subplots(2, 5, figsize=(14, 6))

for i, T in enumerate(temperatures[:5]):
    idx = np.where(np.isclose(T_ising, T))[0][0]
    axes[0, i].imshow(X_ising[idx].reshape(L, L), cmap='coolwarm', vmin=-1, vmax=1)
    axes[0, i].set_title(f'T = {T:.2f}\n(ordered)', fontsize=8)
    axes[0, i].axis('off')

for i, T in enumerate(temperatures[6:11]):
    idx = np.where(np.isclose(T_ising, T))[0][0]
    axes[1, i].imshow(X_ising[idx].reshape(L, L), cmap='coolwarm', vmin=-1, vmax=1)
    axes[1, i].set_title(f'T = {T:.2f}\n(disordered)', fontsize=8)
    axes[1, i].axis('off')

plt.suptitle('2D Ising Model: Ordered vs Disordered Phases', fontsize=12)
```

(continues on next page)

(continued from previous page)

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



20.9.2 Step 2: Train Classifiers on Spin Configurations

```
# Split into training and test sets
X_train_I, X_test_I, y_train_I, y_test_I, T_train_I, T_test_I = train_test_split(
    X_ising, y_ising, T_ising, test_size=0.3, random_state=42, stratify=y_ising
)

print(f"Training set: {len(X_train_I)} samples")
print(f"Test set:     {len(X_test_I)} samples")
```

```
Training set: 840 samples
Test set:     360 samples
```

```
# Train multiple classifiers
ising_models = {
    'Logistic Regression': LogisticRegression(max_iter=1000),
    'k-NN (k=5)': KNeighborsClassifier(n_neighbors=5),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42),
    'SVM (RBF)': SVC(kernel='rbf', probability=True, random_state=42),
}

print(f"{'Model':<25} {'Train Acc':>10} {'Test Acc':>10}")
print('-' * 47)

trained_models = {}
for name, model in ising_models.items():
    model.fit(X_train_I, y_train_I)
    train_acc = model.score(X_train_I, y_train_I)
    test_acc = model.score(X_test_I, y_test_I)
    print(f"{name:<25} {train_acc:>10.3f} {test_acc:>10.3f}")
    trained_models[name] = model
```

```
Model                        Train Acc   Test Acc
-------------------------------------------------
Logistic Regression              0.883      0.667
k-NN (k=5)                       0.854      0.797
Random Forest                    1.000      0.992
SVM (RBF)                        1.000      0.997
```

20.9.3 Step 3: Test Near the Phase Transition

The real test: how do our classifiers perform at temperatures *near* T_c, where the phase distinction is subtle?

```
# Generate test configurations near the critical temperature
T_critical_test = np.linspace(1.5, 3.5, 20)

print("Generating critical-region test data...")
X_crit = []
T_crit = []
for T in T_critical_test:
    configs = metropolis_ising(L, T, n_sweeps=1500, n_equil=800)
    for cfg in configs:
        X_crit.append(cfg.flatten())
        T_crit.append(T)

X_crit = np.array(X_crit)
T_crit = np.array(T_crit)
print(f"Generated {len(X_crit)} test configurations.")
```

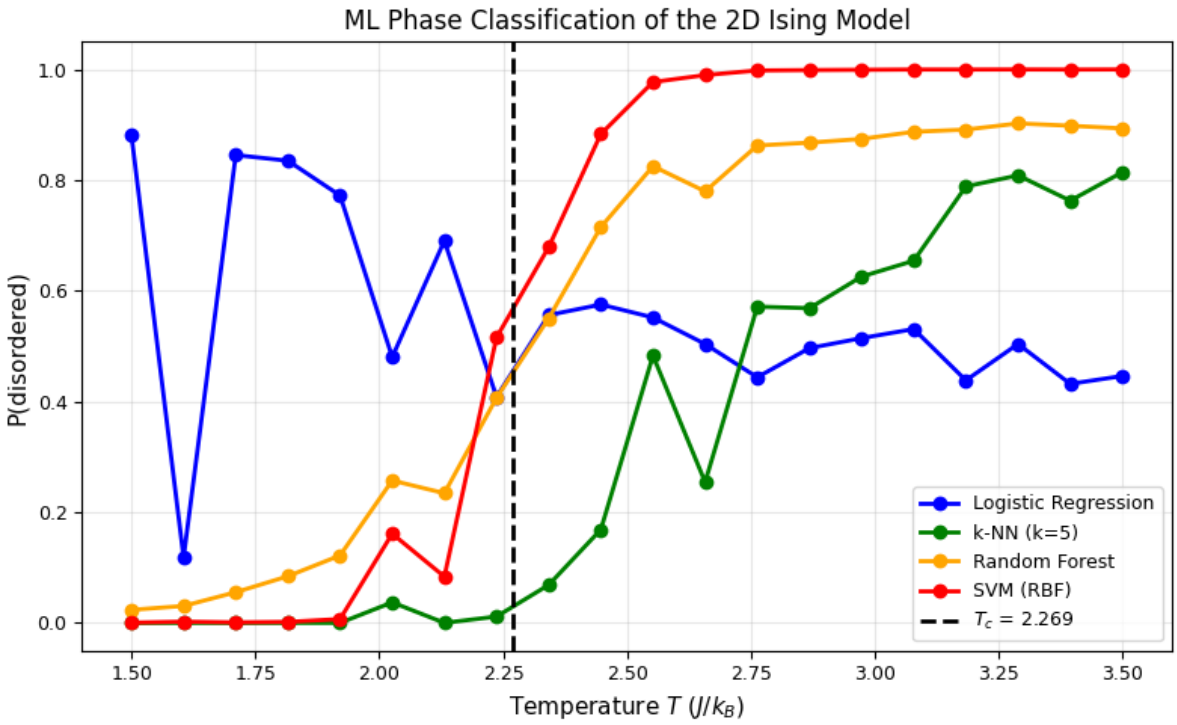
```
Generating critical-region test data...
Generated 1400 test configurations.
```

```
# Plot P(disordered) vs temperature for each model
fig, ax = plt.subplots(figsize=(8, 5))

colors = ['blue', 'green', 'orange', 'red']
for (name, model), color in zip(trained_models.items(), colors):
    # Get probability of being "disordered" for each temperature
    probs = []
    for T in T_critical_test:
        mask = np.isclose(T_crit, T)
        if hasattr(model, 'predict_proba'):
            p = model.predict_proba(X_crit[mask])[:, 1].mean()
        else:
            p = model.predict(X_crit[mask]).mean()
        probs.append(p)
    ax.plot(T_critical_test, probs, 'o-', color=color, label=name, lw=2)

ax.axvline(T_c, color='black', ls='--', lw=2, label=f'$T_c$ = {T_c:.3f}')
ax.set_xlabel('Temperature $T$ ($J/k_B$)', fontsize=11)
ax.set_ylabel('P(disordered)', fontsize=11)
ax.set_title('ML Phase Classification of the 2D Ising Model', fontsize=12)
ax.legend(fontsize=9)
ax.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

(continues on next page)

(continued from previous page)

```
print("All models learn a sharp transition near the true T_c!")
```




```
All models learn a sharp transition near the true T_c!
```

20.9.4 What Did Logistic Regression Learn?

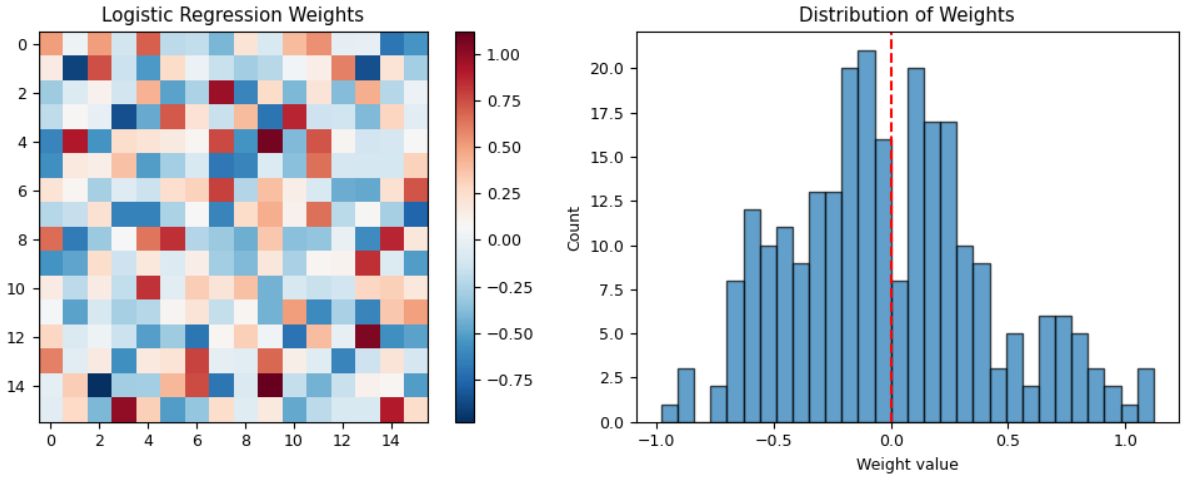
Logistic regression only achieved ~67% accuracy — barely better than random. Why? Since it is a linear model, we can inspect its weights to find out. Each weight corresponds to one spin site.

```
# Visualise the logistic regression weights
lr_model = trained_models['Logistic Regression']
weights = lr_model.coef_[0].reshape(L, L)

fig, axes = plt.subplots(1, 2, figsize=(10, 4))

im = axes[0].imshow(weights, cmap='RdBu_r')
axes[0].set_title('Logistic Regression Weights')
plt.colorbar(im, ax=axes[0])

axes[1].hist(weights.flatten(), bins=30, edgecolor='black', alpha=0.7)
axes[1].set_xlabel('Weight value')
axes[1].set_ylabel('Count')
axes[1].set_title('Distribution of Weights')
axes[1].axvline(0, color='red', ls='--')

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

(continues on next page)

(continued from previous page)

```
print(f"The weights are roughly uniform — LR learned w^T s + b ≈ mean(s_i) = m.")
print(f"But m is NOT the order parameter — |m| is!")
print(f"Ordered phases can have m ≈ +1 OR m ≈ -1, while disordered has m ≈ 0.")
print(f"A linear classifier cannot separate 'both extremes' from 'the middle'.")
print(f"This explains the poor 67% accuracy — and why physics features (|m|) fix it!")
```




```
The weights are roughly uniform — LR learned w^T s + b ≈ mean(s_i) = m.
But m is NOT the order parameter — |m| is!
Ordered phases can have m ≈ +1 OR m ≈ -1, while disordered has m ≈ 0.
A linear classifier cannot separate 'both extremes' from 'the middle'.
This explains the poor 67% accuracy — and why physics features (|m|) fix it!
```

20.10 VIII. Feature Engineering for Physics

The logistic regression failure above teaches an important lesson: **the right features matter more than the right algorithm.**

Raw spin configurations have $L^2 = 256$ features. A physicist knows that the order parameter is $|m|$, not m — and LR can't compute absolute values. Let's give the models physics-informed features:

1. **Raw features**: All L^2 spins (what we did above)
2. **Engineered features**: $|m|$, m^2, energy E/N, nearest-neighbour correlation

```
def extract_physics_features(configs, L):
    """Extract physically meaningful features from spin configurations."""
    features = []
    for cfg_flat in configs:
        cfg = cfg_flat.reshape(L, L)

        # Magnetisation per spin
        m = np.mean(cfg)
        abs_m = np.abs(m)

        # Energy per spin (nearest-neighbour)
        E = 0
```

(continues on next page)

(continued from previous page)

```
        for i in range(L):
            for j in range(L):
                E -= cfg[i, j] * (cfg[(i+1)%L, j] + cfg[i, (j+1)%L])
        E /= L * L

        # Nearest-neighbour correlation
        nn_corr = 0
        for i in range(L):
            for j in range(L):
                nn_corr += cfg[i, j] * cfg[(i+1)%L, j]
        nn_corr /= L * L

        features.append([abs_m, m**2, E, nn_corr])

    return np.array(features)

# Extract physics features
X_phys_train = extract_physics_features(X_train_I, L)
X_phys_test = extract_physics_features(X_test_I, L)

print("Physics features: |m|, m², E/N, <s_i s_j>")
print(f"Shape: {X_phys_train.shape} (only 4 features instead of {L*L}!)")

# Compare: raw features vs physics features
print(f"\n{'Model':<25} {'Raw (256 feat)':>15} {'Physics (4 feat)':>17}")
print('-' * 59)
for name in ['Logistic Regression', 'Random Forest', 'SVM (RBF)']:
    # Raw features
    m1 = ising_models[name].__class__(**ising_models[name].get_params())
    if name == 'Logistic Regression':
        m1.set_params(max_iter=1000)
    m1.fit(X_train_I, y_train_I)
    raw_acc = m1.score(X_test_I, y_test_I)

    # Physics features
    m2 = ising_models[name].__class__(**ising_models[name].get_params())
    if name == 'Logistic Regression':
        m2.set_params(max_iter=1000)
    m2.fit(X_phys_train, y_train_I)
    phys_acc = m2.score(X_phys_test, y_test_I)

    print(f"{name:<25} {raw_acc:>15.3f} {phys_acc:>17.3f}")

print("\nPhysics features work just as well (or better) with 4 features instead of␣
↪256!")
print("Domain knowledge is a powerful form of regularisation.")
```

```
Physics features: |m|, m², E/N, <s_i s_j>
Shape: (840, 4) (only 4 features instead of 256!)

Model                     Raw (256 feat)  Physics (4 feat)
-----------------------------------------------------------
Logistic Regression                0.667             1.000
Random Forest                      0.992             1.000
SVM (RBF)                          0.997             1.000
```

(continues on next page)

(continued from previous page)

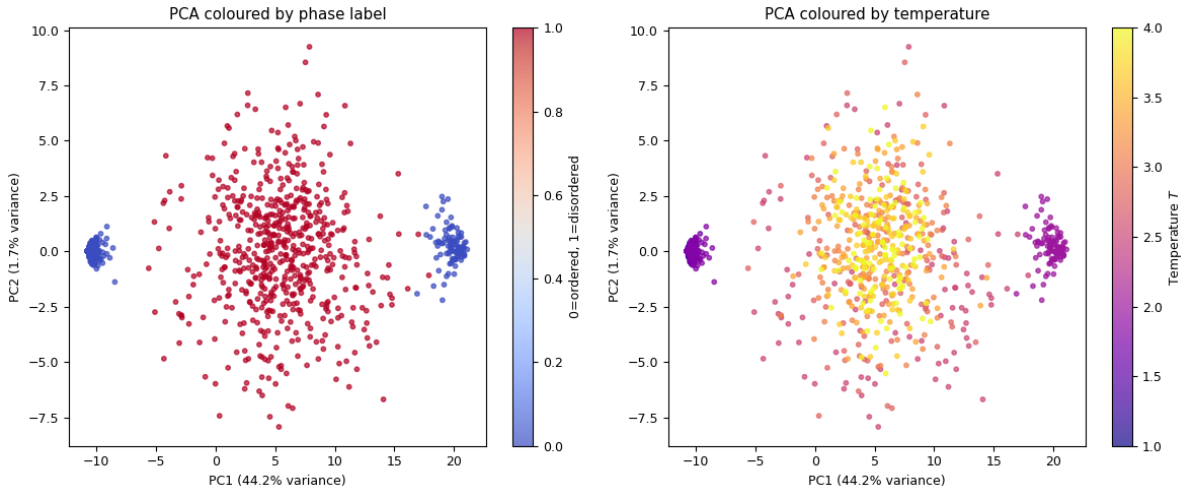
```
Physics features work just as well (or better) with 4 features instead of 256!
Domain knowledge is a powerful form of regularisation.
```

20.11 IX. Unsupervised Learning: PCA

What if we *don't* know the labels? Can we still discover the two phases?

Principal Component Analysis (PCA) finds the directions of maximum variance in the data. If the two phases are truly different, PCA should separate them — without any labels.

```
from sklearn.decomposition import PCA

# Apply PCA to all Ising configurations
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_ising)

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Colour by label
scatter1 = axes[0].scatter(X_pca[:, 0], X_pca[:, 1], c=y_ising, cmap='coolwarm',
                           s=10, alpha=0.7)
axes[0].set_xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.1%} variance)')
axes[0].set_ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.1%} variance)')
axes[0].set_title('PCA coloured by phase label')
plt.colorbar(scatter1, ax=axes[0], label='0=ordered, 1=disordered')

# Colour by temperature
scatter2 = axes[1].scatter(X_pca[:, 0], X_pca[:, 1], c=T_ising, cmap='plasma',
                           s=10, alpha=0.7)
axes[1].set_xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.1%} variance)')
axes[1].set_ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.1%} variance)')
axes[1].set_title('PCA coloured by temperature')
plt.colorbar(scatter2, ax=axes[1], label='Temperature $T$')

plt.tight_layout()
plt.show()

print("PCA separates the two phases without any labels!")
print("The first principal component captures the phase transition.")
```




```
PCA separates the two phases without any labels!
The first principal component captures the phase transition.
```

20.12 Summary

Concept	Key Idea
ML = fitting + generalisation	Same optimisation, but we care about unseen data
Bias–variance trade-off	Simple models underfit, complex models overfit
Regularisation	Penalise complexity (Ridge, Lasso)
k-NN	Classify by nearest neighbours
Decision trees / Random forests	Split features; ensemble reduces variance
SVM	Maximum-margin hyperplane + kernel trick
PCA	Unsupervised dimensionality reduction
Feature engineering	Domain knowledge > more data

Key takeaway: The same Ising phase transition can be detected by supervised classification, unsupervised PCA, or simple physics features. ML is most powerful when combined with physical insight.

Next lecture: We move from classical algorithms to **neural networks** — models that can *learn* their own features.

CHAPTER

TWENTYONE

LECTURE 21 — MACHINE LEARNING II

21.1 Neural Networks & Deep Learning

Computational Physics — Spring 2026

21.2 From Logistic Regression to Neural Networks

In Lecture 20, logistic regression computed:

$$\hat{y} = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

This is a **single neuron**: it takes inputs, computes a weighted sum, and passes it through a non-linear activation function σ.

A neural network is simply **many neurons organised in layers**. By stacking layers, the network can learn arbitrarily complex functions — this is the **universal approximation theorem**.

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

plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

print("Ready for Neural Networks!")
```

```
Ready for Neural Networks!
```

21.3 I. Anatomy of a Neural Network

21.3.1 The Single Neuron (Perceptron)

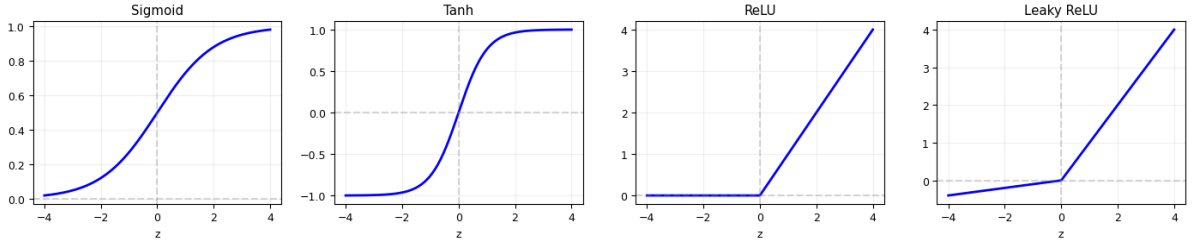
A single neuron computes:

$$\text{output} = \phi\left(\sum_{j=1}^{n} w_j x_j + b\right) = \phi(\mathbf{w}^T\mathbf{x} + b)$$

where ϕ is a non-linear **activation function**.

Without ϕ, stacking layers would just give another linear transformation — the non-linearity is essential.

```
# Common activation functions
z = np.linspace(-4, 4, 300)

activations = {
    'Sigmoid': 1 / (1 + np.exp(-z)),
    'Tanh': np.tanh(z),
    'ReLU': np.maximum(0, z),
    'Leaky ReLU': np.where(z > 0, z, 0.1 * z),
}

fig, axes = plt.subplots(1, 4, figsize=(14, 3))
for ax, (name, values) in zip(axes, activations.items()):
    ax.plot(z, values, 'b-', lw=2)
    ax.axhline(0, color='gray', ls='--', alpha=0.3)
    ax.axvline(0, color='gray', ls='--', alpha=0.3)
    ax.set_title(name)
    ax.set_xlabel('z')
    ax.grid(alpha=0.2)

plt.tight_layout()
plt.show()

print("Modern networks almost always use ReLU or its variants.")
print("Sigmoid/tanh suffer from 'vanishing gradients' in deep networks.")
```




```
Modern networks almost always use ReLU or its variants.
Sigmoid/tanh suffer from 'vanishing gradients' in deep networks.
```

21.3.2 Multi-Layer Perceptron (MLP)

An MLP stacks neurons in **layers**:

```
Input (d features)  →  Hidden layer 1 (h₁ neurons)  →  Hidden layer 2 (h₂ neurons)  →␣
↪ Output
    x₁, x₂, ...          φ(W₁x + b₁)                    φ(W₂h₁ + b₂)                ŷ
```

Mathematically, a 2-hidden-layer MLP computes:

$$\hat{y} = \mathbf{W}_3\,\phi\big(\mathbf{W}_2\,\phi(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1)+\mathbf{b}_2\big)+\mathbf{b}_3$$

Parameters: All the weight matrices $\mathbf{W}_l$ and bias vectors $\mathbf{b}_l$.

A network with d inputs, one hidden layer of h neurons, and 1 output has $(d+1)h+(h+1) = (d+2)h+1$ parameters.

21.3.3 Universal Approximation Theorem

> A feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$, to arbitrary accuracy. — Cybenko (1989), Hornik (1991)

This tells us neural networks are **expressive enough** in principle. But it says nothing about:

- How many neurons we need (could be exponentially many)
- Whether gradient descent can *find* the right weights
- Whether the network will *generalise*

In practice, **deeper** networks (more layers) are more efficient than wider ones, which is why we use "deep" learning.

21.4 II. Building a Neural Network from Scratch

Before using PyTorch, let's implement a simple 2-layer MLP manually to understand every step.

```
# A minimal neural network from scratch (numpy only)
# Task: learn the function f(x) = sin(x) on [0, 2pi]

np.random.seed(42)

# Training data
N = 100
x_data = np.random.uniform(0, 2*np.pi, N)
y_data = np.sin(x_data)

# Normalise inputs (zero mean, unit variance)
# Without this, ~half the ReLU neurons are dead at initialisation
# because x > 0 always and b starts at 0
x_mean, x_std = x_data.mean(), x_data.std()
x_norm = (x_data - x_mean) / x_std

# Network architecture: 1 input -> 32 hidden (ReLU) -> 1 output
n_hidden = 32

# Initialise weights (He initialisation)
W1 = np.random.randn(1, n_hidden) * np.sqrt(2.0 / 1)     # (1, 32)
b1 = np.zeros((1, n_hidden))                               # (1, 32)
W2 = np.random.randn(n_hidden, 1) * np.sqrt(2.0 / n_hidden)  # (32, 1)
b2 = np.zeros((1, 1))                                      # (1, 1)

print(f"Parameters: W1 {W1.shape}, b1 {b1.shape}, W2 {W2.shape}, b2 {b2.shape}")
print(f"Total parameters: {W1.size + b1.size + W2.size + b2.size}")
```

```
Parameters: W1 (1, 32), b1 (1, 32), W2 (32, 1), b2 (1, 1)
Total parameters: 97
```

```
def relu(z):
    return np.maximum(0, z)

def relu_grad(z):
    return (z > 0).astype(float)
```

(continues on next page)

(continued from previous page)

```
def forward(x, W1, b1, W2, b2):
    """Forward pass: input → hidden → output."""
    z1 = x @ W1 + b1      # pre-activation of hidden layer
    a1 = relu(z1)          # activation of hidden layer
    z2 = a1 @ W2 + b2     # output (linear, no activation for regression)
    return z1, a1, z2

def mse_loss(y_pred, y_true):
    """Mean squared error."""
    return np.mean((y_pred - y_true) ** 2)

print("Forward pass and loss defined.")
```

```
Forward pass and loss defined.
```

21.4.1 Backpropagation = Chain Rule

To update the weights, we need $\partial L/\partial \mathbf{W}_l$ for each layer. Backpropagation computes these using the **chain rule**, working backwards from the loss:

$$\frac{\partial L}{\partial \mathbf{W}_2} = \mathbf{a}_1^T \, \delta_2 \qquad \text{where} \quad \delta_2 = \frac{\partial L}{\partial \mathbf{z}_2} = \frac{2}{N}(\hat{y} - y)$$

$$\frac{\partial L}{\partial \mathbf{W}_1} = \mathbf{x}^T \, \delta_1 \qquad \text{where} \quad \delta_1 = (\delta_2 \, \mathbf{W}_2^T) \odot \phi'(\mathbf{z}_1)$$

This is just the gradient descent from Lecture 15, applied to a more complex function!

```
# Training loop with backpropagation
X = x_norm.reshape(-1, 1)  # (N, 1) — normalised inputs
Y = y_data.reshape(-1, 1)  # (N, 1)

lr = 0.01   # learning rate
losses = []

for epoch in range(10000):
    # Forward pass
    z1, a1, z2 = forward(X, W1, b1, W2, b2)
    y_pred = z2
    loss = mse_loss(y_pred, Y)
    losses.append(loss)

    # Backward pass
    delta2 = 2.0 / N * (y_pred - Y)         # (N, 1)
    dW2 = a1.T @ delta2                      # (32, 1)
    db2 = np.sum(delta2, axis=0, keepdims=True)  # (1, 1)

    delta1 = (delta2 @ W2.T) * relu_grad(z1)  # (N, 32)
    dW1 = X.T @ delta1                       # (1, 32)
    db1 = np.sum(delta1, axis=0, keepdims=True)  # (1, 32)

    # Gradient descent update (Lecture 15!)
    W2 -= lr * dW2
    b2 -= lr * db2
    W1 -= lr * dW1
```

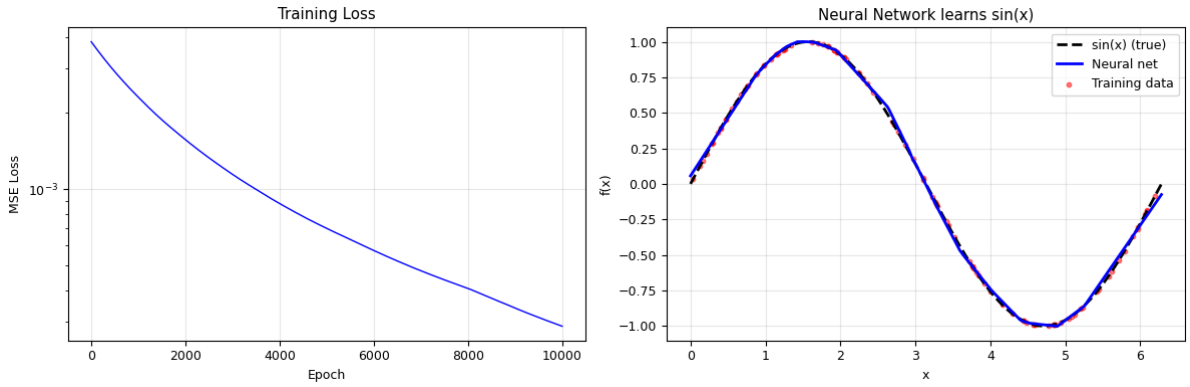
(continues on next page)

(continued from previous page)

```
    b1 -= lr * db1

    if epoch % 1000 == 0:
        print(f"Epoch {epoch:4d}  Loss: {loss:.6f}")

print(f"Final loss: {losses[-1]:.6f}")
```

```
Epoch    0  Loss: 0.003822
Epoch 1000  Loss: 0.002310
Epoch 2000  Loss: 0.001570
Epoch 3000  Loss: 0.001145
Epoch 4000  Loss: 0.000879
Epoch 5000  Loss: 0.000701
Epoch 6000  Loss: 0.000574
Epoch 7000  Loss: 0.000477
Epoch 8000  Loss: 0.000406
Epoch 9000  Loss: 0.000340
Final loss: 0.000288
```

```
# Visualise the result
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Loss curve
axes[0].semilogy(losses, 'b-', lw=1)
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('MSE Loss')
axes[0].set_title('Training Loss')
axes[0].grid(alpha=0.3)

# Prediction vs truth
x_plot = np.linspace(0, 2*np.pi, 300)
x_plot_norm = ((x_plot - x_mean) / x_std).reshape(-1, 1)
_, _, y_plot = forward(x_plot_norm, W1, b1, W2, b2)

axes[1].plot(x_plot, np.sin(x_plot), 'k--', lw=2, label='sin(x) (true)')
axes[1].plot(x_plot, y_plot, 'b-', lw=2, label='Neural net')
axes[1].scatter(x_data, y_data, c='red', s=10, alpha=0.5, label='Training data')
axes[1].set_xlabel('x'); axes[1].set_ylabel('f(x)')
axes[1].set_title('Neural Network learns sin(x)')
axes[1].legend()
axes[1].grid(alpha=0.3)

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



21.5 III. PyTorch: Automatic Differentiation

Writing backpropagation by hand is tedious and error-prone. **PyTorch** provides:

1. **Tensors** — like NumPy arrays but can track gradients
2. **Autograd** — automatic differentiation (computes all gradients for you)
3. **nn.Module** — building blocks for neural networks
4. **Optimisers** — SGD, Adam, etc.

PyTorch builds a **computational graph** during the forward pass and uses it to compute gradients automatically during the backward pass.

```
import torch
import torch.nn as nn

print(f"PyTorch version: {torch.__version__}")
print(f"CUDA available: {torch.cuda.is_available()}")
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f"Using device: {device}")
```

```
PyTorch version: 2.10.0+cpu
CUDA available: False
Using device: cpu
```

```
# PyTorch autograd demo
# Compute gradient of f(x) = x^3 + 2x at x = 3
# Analytical: f'(x) = 3x^2 + 2 => f'(3) = 29

x = torch.tensor(3.0, requires_grad=True)
f = x**3 + 2*x
f.backward()  # compute df/dx

print(f"f(3) = {f.item():.1f}")
print(f"f'(3) = {x.grad.item():.1f}  (analytical: 29.0)")
print("\nPyTorch computed the gradient automatically!")
```

```
f(3) = 33.0
f'(3) = 29.0  (analytical: 29.0)
```

(continues on next page)

(continued from previous page)

```
PyTorch computed the gradient automatically!
```

21.5.1 Building an MLP with `nn.Module`

```python
class MLP(nn.Module):
    """Simple Multi-Layer Perceptron."""

    def __init__(self, input_dim, hidden_dims, output_dim):
        super().__init__()
        layers = []
        prev_dim = input_dim
        for h in hidden_dims:
            layers.append(nn.Linear(prev_dim, h))
            layers.append(nn.ReLU())
            prev_dim = h
        layers.append(nn.Linear(prev_dim, output_dim))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)

# Create a network: 1 input → [64, 64] hidden → 1 output
model = MLP(input_dim=1, hidden_dims=[64, 64], output_dim=1)

# Count parameters
n_params = sum(p.numel() for p in model.parameters())
print(f"Network architecture:\n{model}")
print(f"\nTotal parameters: {n_params}")
```

```
Network architecture:
MLP(
  (net): Sequential(
    (0): Linear(in_features=1, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=64, bias=True)
    (3): ReLU()
    (4): Linear(in_features=64, out_features=1, bias=True)
  )
)

Total parameters: 4353
```

```python
# Training loop in PyTorch
torch.manual_seed(42)

# Data
X_t = torch.tensor(x_data, dtype=torch.float32).reshape(-1, 1)
Y_t = torch.tensor(y_data, dtype=torch.float32).reshape(-1, 1)

# Model, loss, optimiser
model = MLP(1, [64, 64], 1)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

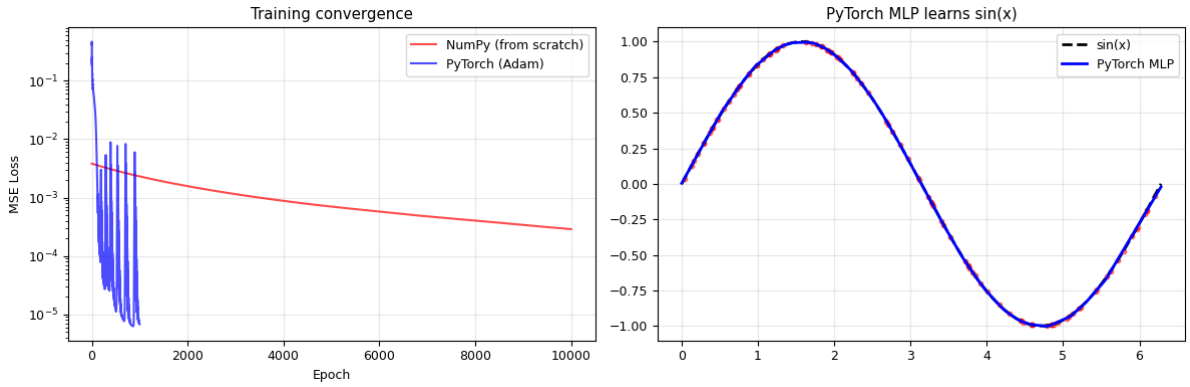
(continues on next page)

(continued from previous page)

```
# Train
losses_pt = []
for epoch in range(1000):
    optimizer.zero_grad()             # clear old gradients
    y_pred = model(X_t)               # forward pass
    loss = criterion(y_pred, Y_t)     # compute loss
    loss.backward()                   # backward pass (autograd!)
    optimizer.step()                  # update weights
    losses_pt.append(loss.item())

    if epoch % 200 == 0:
        print(f"Epoch {epoch:4d}  Loss: {loss.item():.6f}")

print(f"Final loss: {losses_pt[-1]:.6f}")
```

```
Epoch    0  Loss: 0.418733
Epoch  200  Loss: 0.000130
Epoch  400  Loss: 0.000088
Epoch  600  Loss: 0.000016
Epoch  800  Loss: 0.000007
Final loss: 0.000007
```

```
# Compare: numpy from scratch vs PyTorch
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Loss comparison
axes[0].semilogy(losses, 'r-', alpha=0.7, label='NumPy (from scratch)')
axes[0].semilogy(losses_pt, 'b-', alpha=0.7, label='PyTorch (Adam)')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('MSE Loss')
axes[0].set_title('Training convergence')
axes[0].legend()
axes[0].grid(alpha=0.3)

# PyTorch prediction
x_plot_t = torch.linspace(0, 2*np.pi, 300).reshape(-1, 1)
with torch.no_grad():
    y_plot_t = model(x_plot_t).numpy()

axes[1].plot(x_plot_t.numpy(), np.sin(x_plot_t.numpy()), 'k--', lw=2, label='sin(x)')
axes[1].plot(x_plot_t.numpy(), y_plot_t, 'b-', lw=2, label='PyTorch MLP')
axes[1].scatter(x_data, y_data, c='red', s=10, alpha=0.5)
axes[1].set_title('PyTorch MLP learns sin(x)')
axes[1].legend()
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("PyTorch + Adam converges much faster than manual SGD!")
```




```
PyTorch + Adam converges much faster than manual SGD!
```

21.6 IV. Training in Practice

21.6.1 Key Concepts

21.6.2 Loss Functions

Task	Loss	PyTorch
Regression	MSE: $\frac{1}{N}\sum(y-\hat{y})^2$	`nn.MSELoss()`
Regression	MAE: $\frac{1}{N}\sum\|y-\hat{y}\|$	`nn.L1Loss()`
Binary classification	Cross-entropy	`nn.BCEWithLogitsLoss()`
Multi-class	Cross-entropy	`nn.CrossEntropyLoss()`

21.6.3 Optimisers

Optimiser	Key idea	When to use
SGD	Basic gradient descent (Lecture 15)	Baseline, large-scale
SGD + momentum	Accumulate velocity	Faster convergence
Adam	Adaptive learning rate per parameter	Default choice
AdamW	Adam + weight decay	When regularisation matters

21.6.4 Mini-Batch Training

For large datasets, computing the gradient over *all* N samples is expensive. Instead, we use **mini-batches**:

- Split data into batches of size B (typically 32–256)
- Compute gradient on each batch → update weights
- One pass through all batches = one **epoch**

This is **stochastic gradient descent (SGD)** — the gradient is noisy but the noise actually helps escape local minima (recall Monte Carlo / Lecture 14).

```
from torch.utils.data import TensorDataset, DataLoader

# Create a DataLoader for mini-batch training
dataset = TensorDataset(X_t, Y_t)
loader = DataLoader(dataset, batch_size=16, shuffle=True)

print(f"Dataset size: {len(dataset)}")
print(f"Batch size: 16")
print(f"Batches per epoch: {len(loader)}")

# One epoch of mini-batch training
for batch_x, batch_y in loader:
    print(f"  Batch shape: {batch_x.shape}")
    break  # just show the first batch
```

```
Dataset size: 100
Batch size: 16
Batches per epoch: 7
  Batch shape: torch.Size([16, 1])
```

21.6.5 Overfitting and Regularisation in Neural Networks

Neural networks can have millions of parameters — they can memorise training data perfectly. Common regularisation techniques:

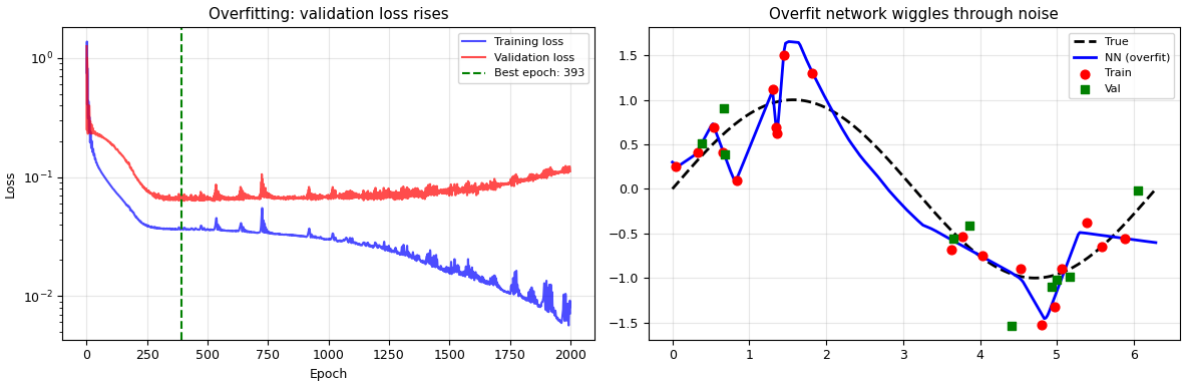
1. **Early stopping**: Stop training when validation loss starts increasing
2. **Weight decay** (L2 regularisation): Add $\lambda \|\mathbf{w}\|^2$ to loss
3. **Dropout**: Randomly set neurons to zero during training
4. **Data augmentation**: Create synthetic training examples

```
# Demo: Overfitting and early stopping
torch.manual_seed(42)

# Small noisy dataset — easy to overfit
N_small = 30
x_small = np.random.uniform(0, 2*np.pi, N_small)
y_small = np.sin(x_small) + 0.3 * np.random.randn(N_small)

# Split into train/val
X_tr = torch.tensor(x_small[:20], dtype=torch.float32).reshape(-1, 1)
Y_tr = torch.tensor(y_small[:20], dtype=torch.float32).reshape(-1, 1)
X_val = torch.tensor(x_small[20:], dtype=torch.float32).reshape(-1, 1)
Y_val = torch.tensor(y_small[20:], dtype=torch.float32).reshape(-1, 1)

# Big model (easy to overfit with 20 data points)
model_big = MLP(1, [128, 128], 1)
optimizer = torch.optim.Adam(model_big.parameters(), lr=0.005)
criterion = nn.MSELoss()

train_losses, val_losses = [], []
for epoch in range(2000):
    model_big.train()
    optimizer.zero_grad()
    loss_tr = criterion(model_big(X_tr), Y_tr)
```

(continues on next page)

(continued from previous page)

```
    loss_tr.backward()
    optimizer.step()

    model_big.eval()
    with torch.no_grad():
        loss_val = criterion(model_big(X_val), Y_val)

    train_losses.append(loss_tr.item())
    val_losses.append(loss_val.item())

# Plot
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].semilogy(train_losses, 'b-', label='Training loss', alpha=0.7)
axes[0].semilogy(val_losses, 'r-', label='Validation loss', alpha=0.7)
best_epoch = np.argmin(val_losses)
axes[0].axvline(best_epoch, color='green', ls='--', label=f'Best epoch: {best_epoch}')
axes[0].set_xlabel('Epoch'); axes[0].set_ylabel('Loss')
axes[0].set_title('Overfitting: validation loss rises')
axes[0].legend(fontsize=8); axes[0].grid(alpha=0.3)

# Prediction at final epoch vs what it should be
x_plot_t = torch.linspace(0, 2*np.pi, 300).reshape(-1, 1)
with torch.no_grad():
    y_plot_big = model_big(x_plot_t).numpy()

axes[1].plot(x_plot_t.numpy(), np.sin(x_plot_t.numpy()), 'k--', lw=2, label='True')
axes[1].plot(x_plot_t.numpy(), y_plot_big, 'b-', lw=2, label='NN (overfit)')
axes[1].scatter(x_small[:20], y_small[:20], c='red', s=40, zorder=5, label='Train')
axes[1].scatter(x_small[20:], y_small[20:], c='green', s=40, zorder=5, marker='s',␣
↪label='Val')
axes[1].set_title('Overfit network wiggles through noise')
axes[1].legend(fontsize=8); axes[1].grid(alpha=0.3)

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



21.7 V. Physics Application: Learning a Potential Energy Surface

In computational physics and chemistry, we often need the **potential energy** $V(\mathbf{r})$ as a function of atomic positions. Computing this from first principles (DFT, ab initio) is expensive. Can a neural network **learn** the potential energy surface from training data?

21.7.1 The Lennard-Jones Potential

As a test case, we use the Lennard-Jones pair potential:

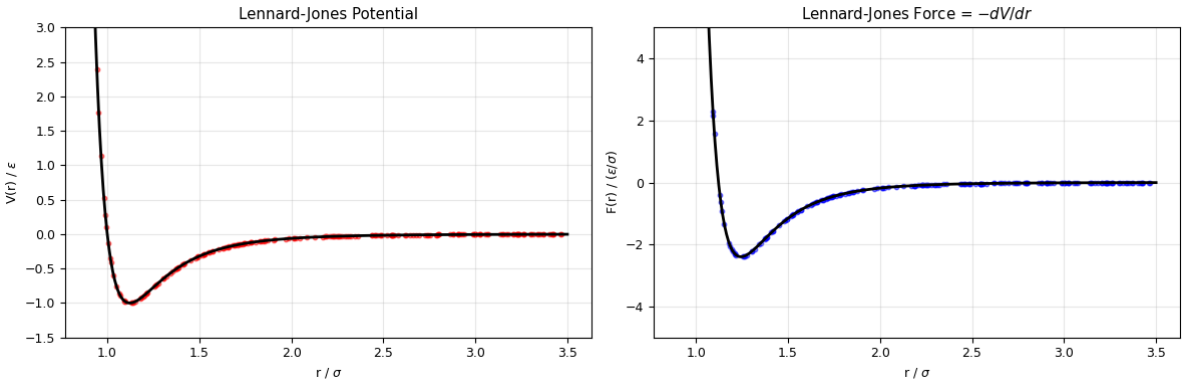
$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

For a cluster of N atoms, the total energy is:

$$E_{\text{total}} = \sum_{i<j} V(r_{ij})$$

We will train a neural network to predict $V(r)$ and its derivative (the force $F = -dV/dr$) simultaneously.

```
# Lennard-Jones potential and force
def lj_potential(r, epsilon=1.0, sigma=1.0):
    """Lennard-Jones potential."""
    sr6 = (sigma / r) ** 6
    return 4 * epsilon * (sr6**2 - sr6)

def lj_force(r, epsilon=1.0, sigma=1.0):
    """LJ force = -dV/dr."""
    sr6 = (sigma / r) ** 6
    return 4 * epsilon * (12 * sr6**2 / r - 6 * sr6 / r)

# Generate training data
np.random.seed(42)
r_train = np.random.uniform(0.9, 3.5, 200)
V_train = lj_potential(r_train)
F_train = lj_force(r_train)

# Visualise
r_plot = np.linspace(0.9, 3.5, 500)
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(r_plot, lj_potential(r_plot), 'k-', lw=2)
axes[0].scatter(r_train, V_train, c='red', s=10, alpha=0.5)
axes[0].set_xlabel('r / $\\sigma$'); axes[0].set_ylabel('V(r) / $\\varepsilon$')
axes[0].set_title('Lennard-Jones Potential')
axes[0].set_ylim(-1.5, 3); axes[0].grid(alpha=0.3)

axes[1].plot(r_plot, lj_force(r_plot), 'k-', lw=2)
axes[1].scatter(r_train, F_train, c='blue', s=10, alpha=0.5)
axes[1].set_xlabel('r / $\\sigma$'); axes[1].set_ylabel('F(r) / ($\\varepsilon / \\
 ↪sigma$)')
axes[1].set_title('Lennard-Jones Force = $-dV/dr$')
axes[1].set_ylim(-5, 5); axes[1].grid(alpha=0.3)

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



21.7.2 Training a Neural Network Potential

We train a network to predict $V(r)$, but we also want it to predict the correct **force** $F = -dV/dr$.

Key trick: use PyTorch's autograd to compute $-dV_{\text{NN}}/dr$ and include it in the loss:

$$L = \underbrace{\sum_i (V_{\text{NN}}(r_i) - V_{\text{true}}(r_i))^2}_{\text{energy loss}} + \lambda \underbrace{\sum_i (F_{\text{NN}}(r_i) - F_{\text{true}}(r_i))^2}_{\text{force loss}}$$

This is a preview of **physics-informed** training (Lecture 23).

```
# Neural network potential with force matching
torch.manual_seed(42)

class NNPotential(nn.Module):
    """Neural network that predicts energy and force."""
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 1),
        )

    def forward(self, r):
        return self.net(r)

    def energy_and_force(self, r):
        """Compute energy and force (via autograd)."""
        r.requires_grad_(True)
        V = self.forward(r)
        # Force = -dV/dr, computed by autograd
        F = -torch.autograd.grad(
            V.sum(), r, create_graph=True
        )[0]
        return V, F

nn_pot = NNPotential()
n_params = sum(p.numel() for p in nn_pot.parameters())
print(f"NN Potential: {n_params} parameters")
```

```
NN Potential: 4353 parameters
```

```
# Prepare data as tensors
r_t = torch.tensor(r_train, dtype=torch.float32).reshape(-1, 1)
V_t = torch.tensor(V_train, dtype=torch.float32).reshape(-1, 1)
F_t = torch.tensor(F_train, dtype=torch.float32).reshape(-1, 1)

optimizer = torch.optim.Adam(nn_pot.parameters(), lr=0.005)
force_weight = 1.0  # relative weight of force loss

losses_energy, losses_force = [], []

for epoch in range(3000):
    optimizer.zero_grad()

    V_pred, F_pred = nn_pot.energy_and_force(r_t.clone())

    loss_E = nn.MSELoss()(V_pred, V_t)
    loss_F = nn.MSELoss()(F_pred, F_t)
    loss = loss_E + force_weight * loss_F

    loss.backward()
    optimizer.step()

    losses_energy.append(loss_E.item())
    losses_force.append(loss_F.item())

    if epoch % 600 == 0:
        print(f"Epoch {epoch:4d}  Energy loss: {loss_E.item():.6f}  Force loss: {loss_
↪F.item():.6f}")

print(f"\nFinal — Energy loss: {losses_energy[-1]:.6f}, Force loss: {losses_force[-
↪1]:.6f}")
```

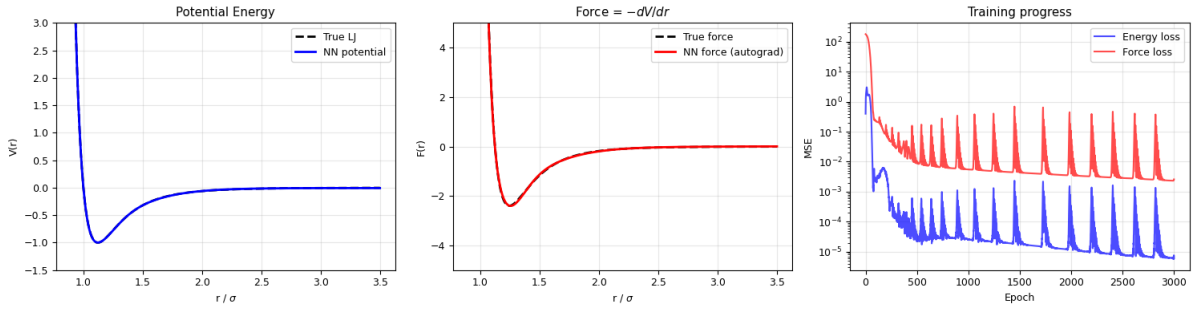
```
Epoch    0  Energy loss: 0.396379  Force loss: 178.423416
Epoch  600  Energy loss: 0.000025  Force loss: 0.008059
Epoch 1200  Energy loss: 0.000022  Force loss: 0.004959
Epoch 1800  Energy loss: 0.000016  Force loss: 0.003905
Epoch 2400  Energy loss: 0.000477  Force loss: 0.141569

Final — Energy loss: 0.000006, Force loss: 0.002581
```

```
# Evaluate the trained NN potential
r_test = torch.linspace(0.9, 3.5, 500).reshape(-1, 1)
nn_pot.eval()
V_nn, F_nn = nn_pot.energy_and_force(r_test.clone())
V_nn = V_nn.detach().numpy()
F_nn = F_nn.detach().numpy()

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Energy
axes[0].plot(r_test.numpy(), lj_potential(r_test.numpy()), 'k--', lw=2, label='True LJ
↪')
axes[0].plot(r_test.numpy(), V_nn, 'b-', lw=2, label='NN potential')
axes[0].set_xlabel('r / $\\sigma$'); axes[0].set_ylabel('V(r)')
```

(continues on next page)

(continued from previous page)

```
axes[0].set_title('Potential Energy')
axes[0].set_ylim(-1.5, 3); axes[0].legend(); axes[0].grid(alpha=0.3)

# Force
axes[1].plot(r_test.numpy(), lj_force(r_test.numpy()), 'k--', lw=2, label='True force
 ↪')
axes[1].plot(r_test.numpy(), F_nn, 'r-', lw=2, label='NN force (autograd)')
axes[1].set_xlabel('r / $\\sigma$'); axes[1].set_ylabel('F(r)')
axes[1].set_title('Force = $-dV/dr$')
axes[1].set_ylim(-5, 5); axes[1].legend(); axes[1].grid(alpha=0.3)

# Training curves
axes[2].semilogy(losses_energy, 'b-', alpha=0.7, label='Energy loss')
axes[2].semilogy(losses_force, 'r-', alpha=0.7, label='Force loss')
axes[2].set_xlabel('Epoch'); axes[2].set_ylabel('MSE')
axes[2].set_title('Training progress')
axes[2].legend(); axes[2].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("The NN learns both the potential AND the correct forces!")
print("The force comes 'for free' via automatic differentiation.")
```




```
The NN learns both the potential AND the correct forces!
The force comes 'for free' via automatic differentiation.
```

21.8 VI. Multi-Atom System: LJ Cluster Energy

Now let's predict the total energy of a small **cluster of atoms**, where the energy depends on *all pairwise distances*.

```
def lj_cluster_energy(positions, epsilon=1.0, sigma=1.0):
    """Total LJ energy for a cluster of atoms.
    positions: (N_atoms, 3) array
    """
    N = len(positions)
    E = 0.0
    for i in range(N):
        for j in range(i+1, N):
            r = np.linalg.norm(positions[i] - positions[j])
            E += lj_potential(r, epsilon, sigma)
    return E
```

(continues on next page)

(continued from previous page)

```
def get_pairwise_distances(positions):
    """Compute all pairwise distances (sorted) as features."""
    N = len(positions)
    dists = []
    for i in range(N):
        for j in range(i+1, N):
            dists.append(np.linalg.norm(positions[i] - positions[j]))
    return np.sort(dists)  # sorted for permutation invariance

# Generate random 4-atom clusters and compute their energies
np.random.seed(42)
n_samples = 2000
n_atoms = 4
n_pairs = n_atoms * (n_atoms - 1) // 2  # = 6 pairwise distances

X_cluster = []
E_cluster = []

for _ in range(n_samples):
    # Random positions near the equilibrium distance
    pos = np.random.randn(n_atoms, 3) * 0.8
    dists = get_pairwise_distances(pos)

    # Skip configurations with atoms too close
    if dists.min() < 0.8:
        continue

    E = lj_cluster_energy(pos)
    X_cluster.append(dists)
    E_cluster.append(E)

X_cluster = np.array(X_cluster)
E_cluster = np.array(E_cluster)

print(f"Generated {len(X_cluster)} valid configurations")
print(f"Features per sample: {n_pairs} pairwise distances")
print(f"Energy range: [{E_cluster.min():.2f}, {E_cluster.max():.2f}]")
```

```
Generated 1262 valid configurations
Features per sample: 6 pairwise distances
Energy range: [-4.20, 77.49]
```

```
# Train NN to predict cluster energy from pairwise distances
from sklearn.model_selection import train_test_split as tts

X_tr, X_te, E_tr, E_te = tts(X_cluster, E_cluster, test_size=0.2, random_state=42)

# Standardise
X_mean, X_std = X_tr.mean(axis=0), X_tr.std(axis=0)
E_mean, E_std = E_tr.mean(), E_tr.std()

X_tr_n = (X_tr - X_mean) / X_std
X_te_n = (X_te - X_mean) / X_std
E_tr_n = (E_tr - E_mean) / E_std
E_te_n = (E_te - E_mean) / E_std
```

(continues on next page)

(continued from previous page)

```
# Tensors
X_tr_t = torch.tensor(X_tr_n, dtype=torch.float32)
E_tr_t = torch.tensor(E_tr_n, dtype=torch.float32).reshape(-1, 1)
X_te_t = torch.tensor(X_te_n, dtype=torch.float32)
E_te_t = torch.tensor(E_te_n, dtype=torch.float32).reshape(-1, 1)

# Train
torch.manual_seed(42)
cluster_model = MLP(n_pairs, [128, 128, 64], 1)
optimizer = torch.optim.Adam(cluster_model.parameters(), lr=0.002)
criterion = nn.MSELoss()

loader = DataLoader(TensorDataset(X_tr_t, E_tr_t), batch_size=64, shuffle=True)

train_l, val_l = [], []
for epoch in range(300):
    cluster_model.train()
    epoch_loss = 0
    for bx, by in loader:
        optimizer.zero_grad()
        loss = criterion(cluster_model(bx), by)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
    train_l.append(epoch_loss / len(loader))

    cluster_model.eval()
    with torch.no_grad():
        val_l.append(criterion(cluster_model(X_te_t), E_te_t).item())

    if epoch % 60 == 0:
        print(f"Epoch {epoch:3d}  Train: {train_l[-1]:.5f}  Val: {val_l[-1]:.5f}")

print(f"Final val loss: {val_l[-1]:.5f}")
```

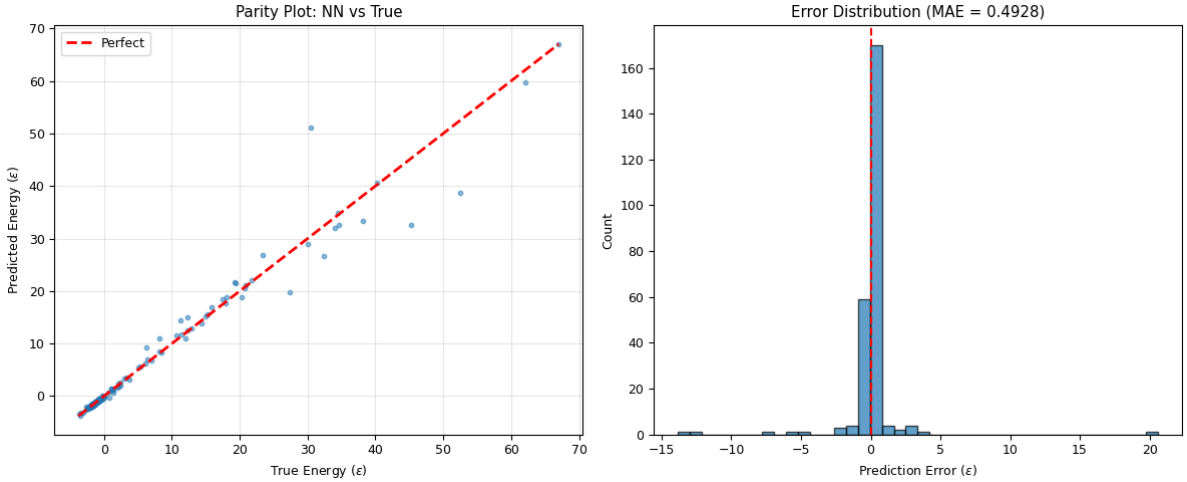
```
Epoch   0  Train: 0.79955  Val: 0.82202
Epoch  60  Train: 0.00539  Val: 0.05320
Epoch 120  Train: 0.00759  Val: 0.04778
Epoch 180  Train: 0.01588  Val: 0.04999
Epoch 240  Train: 0.02666  Val: 0.07756
Final val loss: 0.04686
```

```
# Parity plot: predicted vs true energy
cluster_model.eval()
with torch.no_grad():
    E_pred_n = cluster_model(X_te_t).numpy().flatten()

# Un-standardise
E_pred = E_pred_n * E_std + E_mean

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Parity plot
axes[0].scatter(E_te, E_pred, s=10, alpha=0.5)
emin, emax = min(E_te.min(), E_pred.min()), max(E_te.max(), E_pred.max())
```

(continues on next page)

(continued from previous page)

```
axes[0].plot([emin, emax], [emin, emax], 'r--', lw=2, label='Perfect')
axes[0].set_xlabel('True Energy ($\\varepsilon$)')
axes[0].set_ylabel('Predicted Energy ($\\varepsilon$)')
axes[0].set_title('Parity Plot: NN vs True')
axes[0].legend(); axes[0].grid(alpha=0.3)

# Error distribution
errors = E_pred - E_te
axes[1].hist(errors, bins=40, edgecolor='black', alpha=0.7)
axes[1].set_xlabel('Prediction Error ($\\varepsilon$)')
axes[1].set_ylabel('Count')
axes[1].set_title(f'Error Distribution (MAE = {np.abs(errors).mean():.4f})')
axes[1].axvline(0, color='red', ls='--')

plt.tight_layout()
plt.show()

from sklearn.metrics import r2_score
print(f"R² score: {r2_score(E_te, E_pred):.4f}")
print(f"MAE: {np.abs(errors).mean():.4f} epsilon")
```




```
R² score: 0.9652
MAE: 0.4928 epsilon
```

21.9 VII. Ising Model Revisited: NN Classifier

Let's revisit the Ising phase classification from Lecture 21, now using a neural network.

```
# Generate Ising data (same as Lecture 21)
def metropolis_ising(L, T, n_sweeps=1000, n_equil=500):
    spins = np.random.choice([-1, 1], size=(L, L))
    beta = 1.0 / T
    configs = []
    for sweep in range(n_sweeps):
        for _ in range(L * L):
            i, j = np.random.randint(0, L, size=2)
```

(continues on next page)

(continued from previous page)

```
            neighbours = (
                spins[(i+1) % L, j] + spins[(i-1) % L, j] +
                spins[i, (j+1) % L] + spins[i, (j-1) % L]
            )
            dE = 2 * spins[i, j] * neighbours
            if dE <= 0 or np.random.rand() < np.exp(-beta * dE):
                spins[i, j] *= -1
        if sweep >= n_equil and sweep % 10 == 0:
            configs.append(spins.copy())
    return configs

L = 16
T_c = 2.269
temperatures = np.concatenate([np.linspace(1.0, 2.0, 6), np.linspace(2.5, 4.0, 6)])

X_ising, y_ising = [], []
print("Generating Ising data...")
for T in temperatures:
    configs = metropolis_ising(L, T, n_sweeps=2000, n_equil=1000)
    for cfg in configs:
        X_ising.append(cfg.flatten().astype(np.float32))
        y_ising.append(0 if T < T_c else 1)

X_ising = np.array(X_ising)
y_ising = np.array(y_ising)
print(f"Dataset: {len(X_ising)} samples, {X_ising.shape[1]} features")
```

```
Generating Ising data...
Dataset: 1200 samples, 256 features
```

```
# Split and convert to tensors
X_tr, X_te, y_tr, y_te = tts(X_ising, y_ising, test_size=0.3, random_state=42)

X_tr_t = torch.tensor(X_tr, dtype=torch.float32)
y_tr_t = torch.tensor(y_tr, dtype=torch.long)
X_te_t = torch.tensor(X_te, dtype=torch.float32)
y_te_t = torch.tensor(y_te, dtype=torch.long)

# NN classifier
torch.manual_seed(42)

class IsingClassifier(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, 64),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 2),  # 2 classes
        )

    def forward(self, x):
        return self.net(x)
```

(continues on next page)

(continued from previous page)

```
ising_nn = IsingClassifier(L * L)
optimizer = torch.optim.Adam(ising_nn.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

loader = DataLoader(TensorDataset(X_tr_t, y_tr_t), batch_size=32, shuffle=True)

# Train
for epoch in range(50):
    ising_nn.train()
    for bx, by in loader:
        optimizer.zero_grad()
        loss = criterion(ising_nn(bx), by)
        loss.backward()
        optimizer.step()

    if epoch % 10 == 0:
        ising_nn.eval()
        with torch.no_grad():
            pred_tr = ising_nn(X_tr_t).argmax(dim=1)
            pred_te = ising_nn(X_te_t).argmax(dim=1)
            acc_tr = (pred_tr == y_tr_t).float().mean().item()
            acc_te = (pred_te == y_te_t).float().mean().item()
        print(f"Epoch {epoch:2d}  Train acc: {acc_tr:.3f}  Test acc: {acc_te:.3f}")

# Final accuracy
ising_nn.eval()
with torch.no_grad():
    final_acc = (ising_nn(X_te_t).argmax(dim=1) == y_te_t).float().mean().item()
print(f"\nFinal test accuracy: {final_acc:.3f}")
```

```
Epoch  0  Train acc: 0.924  Test acc: 0.919
Epoch 10  Train acc: 1.000  Test acc: 0.981
Epoch 20  Train acc: 1.000  Test acc: 0.981
Epoch 30  Train acc: 1.000  Test acc: 0.978
Epoch 40  Train acc: 1.000  Test acc: 0.986

Final test accuracy: 0.967
```

21.10 Summary

Concept	Key Idea
Neuron	Weighted sum + non-linear activation
MLP	Stack neurons in layers
Activation functions	ReLU (modern default), sigmoid, tanh
Backpropagation	Chain rule on the computational graph
PyTorch	Automatic differentiation + GPU acceleration
Loss functions	MSE (regression), cross-entropy (classification)
Adam optimiser	Adaptive learning rates — the default choice
Overfitting	Early stopping, dropout, weight decay
NN potentials	Learn $V(r)$ and get forces $F = -dV/dr$ via autograd

CHAPTER

TWENTYTWO

LECTURE 22 — MACHINE LEARNING III

22.1 Deep Learning for Physics: CNNs, Autoencoders & PINNs

Computational Physics — Spring 2026

22.2 Motivation: Why Standard NNs Are Not Enough for Physics

In Lecture 21, we used **MLPs** — fully connected networks. They work, but:

1. **Spatial structure is ignored.** An MLP treats a 2D spin lattice as a flat vector — neighbouring spins have no special relationship.
2. **No physics built in.** The network must learn conservation laws, symmetries, and boundary conditions purely from data.
3. **Data hungry.** Without inductive bias, you need massive datasets.

This lecture introduces three architectures that fix these problems:

Architecture	Key idea	Physics use
CNN	Exploit spatial locality	Lattice models, field data
Autoencoder	Learn compressed representations	Order parameters, phase space
PINN	Embed PDEs in the loss	Solve ODEs/PDEs without grids

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader

plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f"PyTorch {torch.__version__}, device: {device}")
```

```
PyTorch 2.6.0, device: cpu
```

22.3 I. Convolutional Neural Networks (CNNs)

22.3.1 The Idea: Local Connectivity + Weight Sharing

Instead of connecting every input to every neuron, a CNN uses a **filter** (kernel) that slides across the input:

```
Input:   [1  0 -1  1 -1  0  1 -1]
Filter:  [1  0 -1]  (size 3)

Output:  [1*1+0*0+(-1)*(-1),  0*1+(-1)*0+1*(-1),  ...]  = [2, -1, ...]
```

Key properties:

1. **Translation invariance**: The same filter detects the same pattern anywhere
2. **Local connectivity**: Each output depends only on a local patch
3. **Parameter efficiency**: One 3x3 filter = 9 parameters, regardless of input size

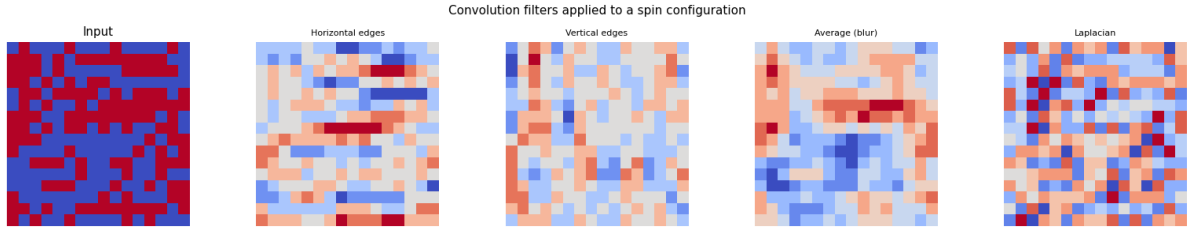
For physics: lattice models, field configurations, and images all have **spatial structure** that CNNs can exploit.

```
# Visualise what a convolution does
from scipy.signal import convolve2d

# Create a simple 2D pattern (like an Ising configuration)
np.random.seed(42)
image = np.random.choice([-1, 1], size=(16, 16)).astype(float)

# Different filters detect different features
filters = {
    'Horizontal edges': np.array([[-1, -1, -1],
                                  [ 0,  0,  0],
                                  [ 1,  1,  1]]),
    'Vertical edges':   np.array([[-1, 0, 1],
                                  [-1, 0, 1],
                                  [-1, 0, 1]]),
    'Average (blur)':   np.ones((3, 3)) / 9,
    'Laplacian':        np.array([[ 0, 1, 0],
                                  [ 1,-4, 1],
                                  [ 0, 1, 0]]),
}

fig, axes = plt.subplots(1, 5, figsize=(16, 3))
axes[0].imshow(image, cmap='coolwarm')
axes[0].set_title('Input')

for ax, (name, kernel) in zip(axes[1:], filters.items()):
    result = convolve2d(image, kernel, mode='same', boundary='wrap')
    ax.imshow(result, cmap='coolwarm')
    ax.set_title(name, fontsize=8)

for ax in axes:
    ax.axis('off')

plt.suptitle('Convolution filters applied to a spin configuration', fontsize=11)
plt.tight_layout()
plt.show()
```

(continues on next page)

(continued from previous page)

```
print("Note: the Laplacian filter is exactly the discrete Laplacian")
print("operator from the PDE lecture (Lecture 11)!")
```




```
Note: the Laplacian filter is exactly the discrete Laplacian
operator from the PDE lecture (Lecture 11)!
```

22.3.2 CNN for Ising Phase Classification

In Lecture 21, we flattened the spin lattice into a vector and used logistic regression. Now we keep the 2D structure and use a CNN.

```
# Generate Ising data (same Metropolis sampler)
def metropolis_ising(L, T, n_sweeps=1000, n_equil=500):
    spins = np.random.choice([-1, 1], size=(L, L))
    beta = 1.0 / T
    configs = []
    for sweep in range(n_sweeps):
        for _ in range(L * L):
            i, j = np.random.randint(0, L, size=2)
            nb = (spins[(i+1)%L, j] + spins[(i-1)%L, j] +
                  spins[i, (j+1)%L] + spins[i, (j-1)%L])
            dE = 2 * spins[i, j] * nb
            if dE <= 0 or np.random.rand() < np.exp(-beta * dE):
                spins[i, j] *= -1
        if sweep >= n_equil and sweep % 10 == 0:
            configs.append(spins.copy())
    return configs

L = 16
T_c = 2.269
temps = np.concatenate([np.linspace(1.0, 2.0, 6), np.linspace(2.5, 4.0, 6)])

X_all, y_all = [], []
print("Generating Ising configurations...")
for T in temps:
    for cfg in metropolis_ising(L, T, n_sweeps=2000, n_equil=1000):
        X_all.append(cfg.astype(np.float32))
        y_all.append(0 if T < T_c else 1)

X_all = np.array(X_all)[:, np.newaxis, :, :]  # (N, 1, L, L) — channel dim
y_all = np.array(y_all)
print(f"Dataset: {X_all.shape[0]} samples, shape per sample: {X_all.shape[1:]}")
```

```
Generating Ising configurations...
Dataset: 1200 samples, shape per sample: (1, 16, 16)
```

```
from sklearn.model_selection import train_test_split

X_tr, X_te, y_tr, y_te = train_test_split(X_all, y_all, test_size=0.3, random_
 ↪state=42)

X_tr_t = torch.tensor(X_tr)
y_tr_t = torch.tensor(y_tr, dtype=torch.long)
X_te_t = torch.tensor(X_te)
y_te_t = torch.tensor(y_te, dtype=torch.long)

class IsingCNN(nn.Module):
    """CNN for classifying Ising model phases."""
    def __init__(self):
        super().__init__()
        self.conv = nn.Sequential(
            # Input: (1, 16, 16)
            nn.Conv2d(1, 16, kernel_size=3, padding=1),  # (16, 16, 16)
            nn.ReLU(),
            nn.MaxPool2d(2),                              # (16, 8, 8)
            nn.Conv2d(16, 32, kernel_size=3, padding=1),  # (32, 8, 8)
            nn.ReLU(),
            nn.MaxPool2d(2),                              # (32, 4, 4)
        )
        self.fc = nn.Sequential(
            nn.Linear(32 * 4 * 4, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 2),
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)  # flatten
        return self.fc(x)

cnn = IsingCNN()
n_params_cnn = sum(p.numel() for p in cnn.parameters())
print(f"CNN parameters: {n_params_cnn}")
print(f"Compare: MLP on flattened input would need {L*L * 64} = {L*L*64} params in␣
 ↪first layer alone")
```

```
CNN parameters: 37762
Compare: MLP on flattened input would need 16384 = 16384 params in first layer␣
 ↪alone
```

```
# Train CNN
torch.manual_seed(42)
cnn = IsingCNN()
optimizer = torch.optim.Adam(cnn.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
loader = DataLoader(TensorDataset(X_tr_t, y_tr_t), batch_size=32, shuffle=True)

train_accs, test_accs = [], []
for epoch in range(30):
    cnn.train()
    for bx, by in loader:
        optimizer.zero_grad()
```

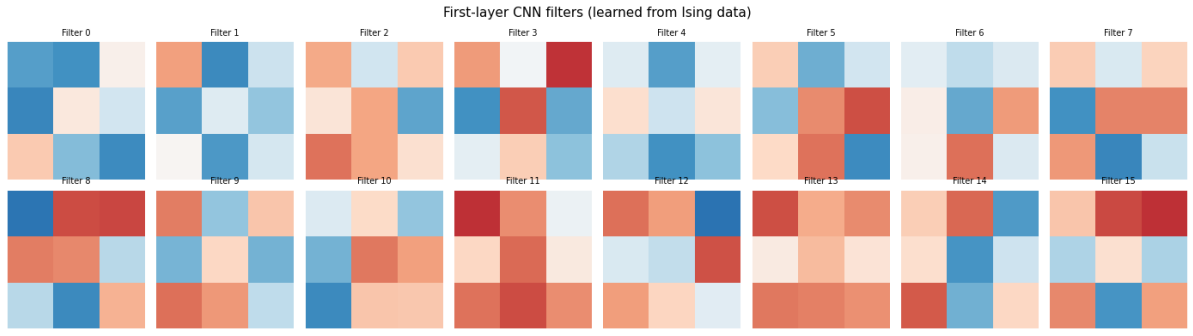
(continues on next page)

(continued from previous page)

```
        criterion(cnn(bx), by).backward()
        optimizer.step()

    cnn.eval()
    with torch.no_grad():
        acc_tr = (cnn(X_tr_t).argmax(1) == y_tr_t).float().mean().item()
        acc_te = (cnn(X_te_t).argmax(1) == y_te_t).float().mean().item()
    train_accs.append(acc_tr)
    test_accs.append(acc_te)
    if epoch % 5 == 0:
        print(f"Epoch {epoch:2d}  Train: {acc_tr:.3f}  Test: {acc_te:.3f}")

print(f"\nFinal test accuracy: {test_accs[-1]:.3f}")
```

```
Epoch  0  Train: 0.993  Test: 0.992
Epoch  5  Train: 0.999  Test: 0.997
Epoch 10  Train: 1.000  Test: 0.997
Epoch 15  Train: 1.000  Test: 0.997
Epoch 20  Train: 1.000  Test: 0.997
Epoch 25  Train: 0.989  Test: 0.994

Final test accuracy: 0.997
```

```
# Visualise learned convolutional filters
filters_learned = cnn.conv[0].weight.detach().numpy()  # (16, 1, 3, 3)

fig, axes = plt.subplots(2, 8, figsize=(14, 4))
for i, ax in enumerate(axes.flat):
    ax.imshow(filters_learned[i, 0], cmap='RdBu', vmin=-0.5, vmax=0.5)
    ax.set_title(f'Filter {i}', fontsize=7)
    ax.axis('off')

plt.suptitle('First-layer CNN filters (learned from Ising data)', fontsize=11)
plt.tight_layout()
plt.show()

print("Some filters resemble edge detectors / nearest-neighbour correlators.")
print("The CNN automatically discovers physically meaningful features!")
```




```
Some filters resemble edge detectors / nearest-neighbour correlators.
The CNN automatically discovers physically meaningful features!
```

22.4 II. Autoencoders: Learning Compressed Representations

An **autoencoder** learns to compress data into a low-dimensional **latent space** and reconstruct it:

```
Input x  →  Encoder  →  Latent z (small!)  →  Decoder  →  Reconstructed x̂
 (256)        ↓            (2-10)               ↑           (256)
           compress                          decompress
```

The loss is simply $L = \|\mathbf{x} - \hat{\mathbf{x}}\|^2$ — reconstruction error.

For physics: The latent space variables are like **order parameters** — the minimal set of variables needed to describe the essential physics.

```
class IsingAutoencoder(nn.Module):
    """Autoencoder that compresses Ising configurations to 2D latent space."""
    def __init__(self, input_dim, latent_dim=2):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 32),
            nn.ReLU(),
            nn.Linear(32, latent_dim),
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 32),
            nn.ReLU(),
            nn.Linear(32, 128),
            nn.ReLU(),
            nn.Linear(128, input_dim),
            nn.Tanh(),  # output in [-1, 1] like spins
        )

    def forward(self, x):
        z = self.encoder(x)
        x_hat = self.decoder(z)
        return x_hat, z

# Flatten Ising data for autoencoder
X_flat = X_all.reshape(len(X_all), -1)  # (N, 256)
X_flat_t = torch.tensor(X_flat, dtype=torch.float32)

ae = IsingAutoencoder(L * L, latent_dim=2)
print(f"Autoencoder: {L*L} → 128 → 32 → 2 → 32 → 128 → {L*L}")
print(f"Parameters: {sum(p.numel() for p in ae.parameters())}")
```

```
Autoencoder: 256 → 128 → 32 → 2 → 32 → 128 → 256
Parameters: 74434
```

```
# Train autoencoder (unsupervised — no labels!)
torch.manual_seed(42)
ae = IsingAutoencoder(L * L, latent_dim=2)
optimizer = torch.optim.Adam(ae.parameters(), lr=0.001)
loader = DataLoader(TensorDataset(X_flat_t), batch_size=64, shuffle=True)

ae_losses = []
for epoch in range(100):
```

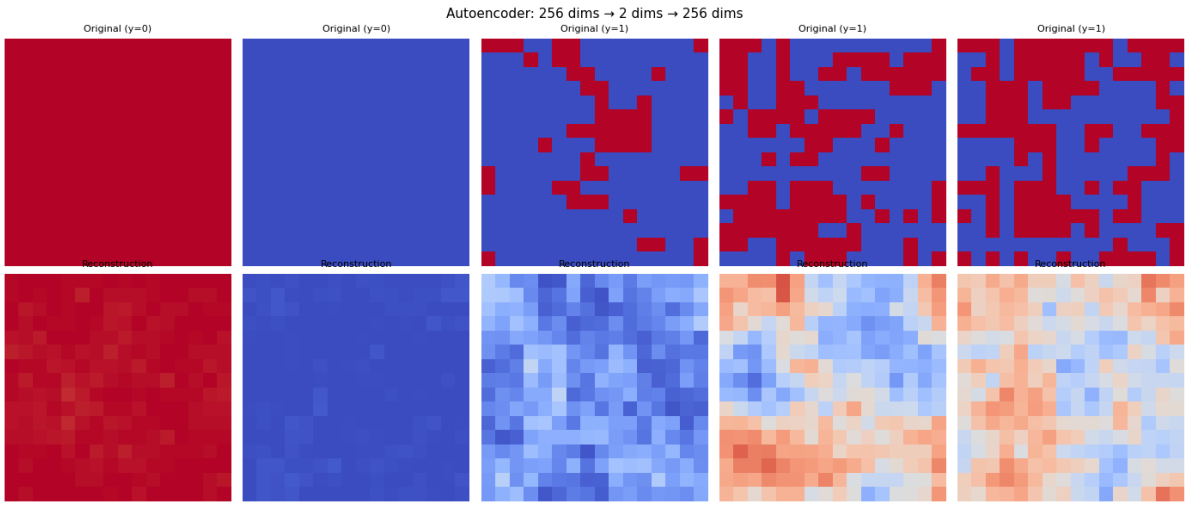
(continues on next page)

(continued from previous page)

```
    ae.train()
    epoch_loss = 0
    for (bx,) in loader:
        optimizer.zero_grad()
        x_hat, z = ae(bx)
        loss = nn.MSELoss()(x_hat, bx)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
    ae_losses.append(epoch_loss / len(loader))
    if epoch % 20 == 0:
        print(f"Epoch {epoch:3d}  Reconstruction loss: {ae_losses[-1]:.5f}")
```

```
Epoch   0  Reconstruction loss: 0.84789
Epoch  20  Reconstruction loss: 0.47335
Epoch  40  Reconstruction loss: 0.46386
Epoch  60  Reconstruction loss: 0.45546
Epoch  80  Reconstruction loss: 0.44664
```

```
# Show reconstructions
ae.eval()
idx = [0, len(X_all)//4, len(X_all)//2, 3*len(X_all)//4, -1]

fig, axes = plt.subplots(2, 5, figsize=(14, 6))
for i, j in enumerate(idx):
    original = X_flat[j].reshape(L, L)
    with torch.no_grad():
        recon, _ = ae(X_flat_t[[j]])  # fancy indexing — works for j=-1
    recon = recon.numpy().reshape(L, L)

    axes[0, i].imshow(original, cmap='coolwarm', vmin=-1, vmax=1)
    axes[0, i].set_title(f'Original (y={y_all[j]})', fontsize=8)
    axes[0, i].axis('off')

    axes[1, i].imshow(recon, cmap='coolwarm', vmin=-1, vmax=1)
    axes[1, i].set_title('Reconstruction', fontsize=8)
    axes[1, i].axis('off')

axes[0, 0].set_ylabel('Original', fontsize=10)
axes[1, 0].set_ylabel('Reconstructed', fontsize=10)
plt.suptitle('Autoencoder: 256 dims \u2192 2 dims \u2192 256 dims', fontsize=11)
plt.tight_layout()
plt.show()
```



22.5 III. Physics-Informed Neural Networks (PINNs)

22.5.1 The Big Idea

Instead of learning from data, we can train a neural network to **satisfy a differential equation directly**.

Given a PDE: $\mathcal{L}[u](x,t) = 0 \quad$ (e.g., heat equation: $u_t - \alpha u_{xx} = 0$)$

with boundary/initial conditions, we:

1. Represent the solution as a neural network: $u(x,t) \approx u_\theta(x,t)$
2. Compute derivatives u_t, u_{xx} etc. using **autograd** (Lecture 21)
3. Minimise the **physics loss**:

$$L = \underbrace{\sum_{\text{collocation}} \left|\mathcal{L}[u_\theta]\right|^2}_{\text{PDE residual}} + \lambda_{\text{BC}} \underbrace{\sum_{\text{boundary}} |u_\theta - u_{\text{BC}}|^2}_{\text{boundary conditions}} + \lambda_{\text{IC}} \underbrace{\sum_{\text{initial}} |u_\theta - u_{\text{IC}}|^2}_{\text{initial conditions}}$$

No training data needed — just the equation itself!

This connects directly to your ODE (Lecture 10) and PDE (Lecture 11) lectures, but replaces finite differences with neural networks.

22.5.2 Example 1: PINN Solves a Simple ODE

Let's start simple. Solve:

$$\frac{du}{dt} = -u, \quad u(0) = 1$$

Exact solution: $u(t) = e^{-t}$

```
# PINN for du/dt = -u, u(0) = 1
torch.manual_seed(42)

class PINN_ODE(nn.Module):
```

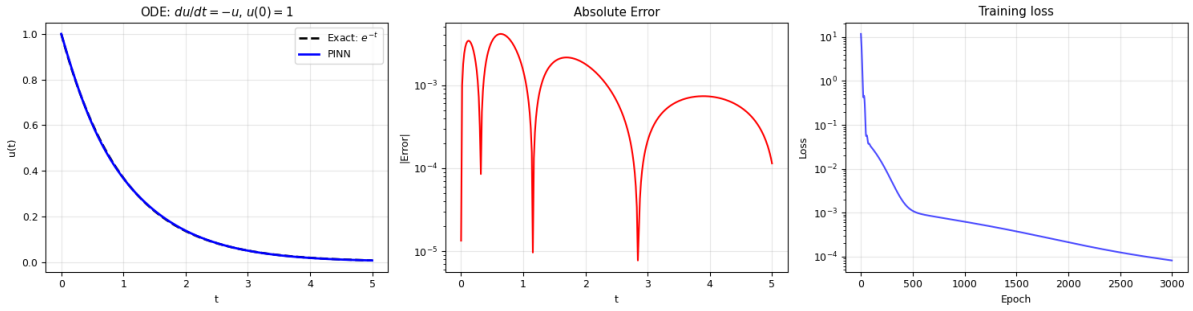
(continues on next page)

(continued from previous page)

```
    """PINN for solving du/dt = -u."""
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 32),
            nn.Tanh(),
            nn.Linear(32, 32),
            nn.Tanh(),
            nn.Linear(32, 1),
        )

    def forward(self, t):
        return self.net(t)

pinn_ode = PINN_ODE()
optimizer = torch.optim.Adam(pinn_ode.parameters(), lr=0.001)

# Collocation points (where we enforce the ODE)
t_col = torch.linspace(0, 5, 200, requires_grad=True).reshape(-1, 1)
# Initial condition point
t_ic = torch.tensor([[0.0]])
u_ic = torch.tensor([[1.0]])

losses_ode = []
for epoch in range(3000):
    optimizer.zero_grad()

    # PDE residual: du/dt + u = 0
    u = pinn_ode(t_col)
    du_dt = torch.autograd.grad(u.sum(), t_col, create_graph=True)[0]
    residual = du_dt + u  # should be 0
    loss_pde = (residual ** 2).mean()

    # Initial condition: u(0) = 1
    u0 = pinn_ode(t_ic)
    loss_ic = (u0 - u_ic) ** 2

    # Total loss
    loss = loss_pde + 10 * loss_ic
    loss.backward()
    optimizer.step()
    losses_ode.append(loss.item())

    if epoch % 600 == 0:
        print(f"Epoch {epoch:4d}  PDE loss: {loss_pde.item():.2e}  IC loss: {loss_ic.
↪item():.2e}")

print(f"Final total loss: {losses_ode[-1]:.2e}")
```

```
Epoch    0  PDE loss: 7.97e-02  IC loss: 1.17e+00
Epoch  600  PDE loss: 9.08e-04  IC loss: 4.79e-08
Epoch 1200  PDE loss: 5.13e-04  IC loss: 7.37e-09
Epoch 1800  PDE loss: 2.67e-04  IC loss: 2.08e-09
Epoch 2400  PDE loss: 1.37e-04  IC loss: 5.97e-10
Final total loss: 8.15e-05
```

```
# Compare PINN solution with exact solution
t_test = torch.linspace(0, 5, 300).reshape(-1, 1)
with torch.no_grad():
    u_pinn = pinn_ode(t_test).numpy().flatten()

u_exact = np.exp(-t_test.numpy().flatten())

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

axes[0].plot(t_test.numpy(), u_exact, 'k--', lw=2, label='Exact: $e^{-t}$')
axes[0].plot(t_test.numpy(), u_pinn, 'b-', lw=2, label='PINN')
axes[0].set_xlabel('t'); axes[0].set_ylabel('u(t)')
axes[0].set_title('ODE: $du/dt = -u$, $u(0)=1$')
axes[0].legend(); axes[0].grid(alpha=0.3)

axes[1].semilogy(t_test.numpy(), np.abs(u_pinn - u_exact), 'r-', lw=1.5)
axes[1].set_xlabel('t'); axes[1].set_ylabel('|Error|')
axes[1].set_title('Absolute Error')
axes[1].grid(alpha=0.3)

axes[2].semilogy(losses_ode, 'b-', alpha=0.7)
axes[2].set_xlabel('Epoch'); axes[2].set_ylabel('Loss')
axes[2].set_title('Training loss'); axes[2].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Maximum error: {np.max(np.abs(u_pinn - u_exact)):.2e}")
print("The PINN learned the solution without ANY training data — just the equation!")
```




```
Maximum error: 4.12e-03
The PINN learned the solution without ANY training data — just the equation!
```

22.5.3 Example 2: PINN Solves a Simple Harmonic Oscillator

A more interesting ODE — the simple harmonic oscillator from Lecture 10:

$$\frac{d^2x}{dt^2} = -\omega^2 x, \quad x(0) = 1, \quad \dot{x}(0) = 0$$

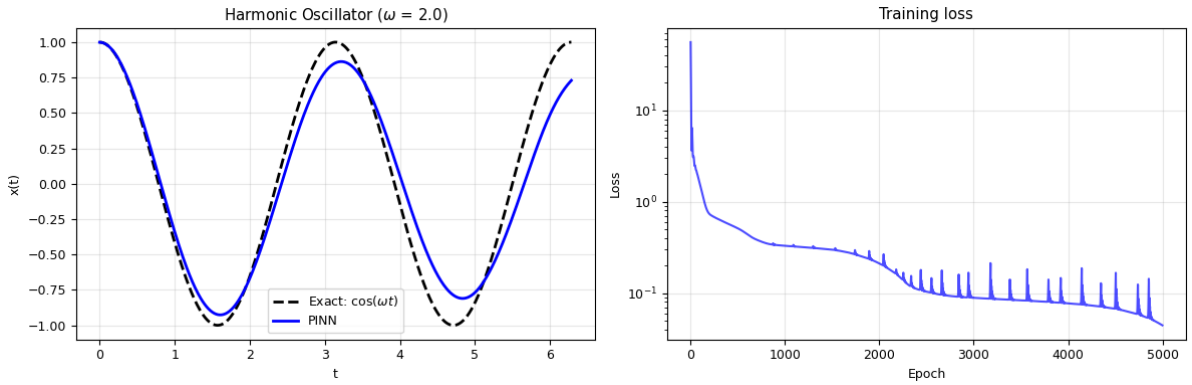
Exact solution: $x(t) = \cos(\omega t)$

This is a **second-order** ODE, so we need second derivatives via autograd.

```
# PINN for harmonic oscillator
torch.manual_seed(42)
omega = 2.0  # angular frequency

class PINN_SHO(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 1),
        )

    def forward(self, t):
        return self.net(t)

pinn_sho = PINN_SHO()
optimizer = torch.optim.Adam(pinn_sho.parameters(), lr=0.001)

t_col = torch.linspace(0, 4*np.pi/omega, 400).reshape(-1, 1).requires_grad_(True)

losses_sho = []
for epoch in range(5000):
    optimizer.zero_grad()

    # Forward
    x = pinn_sho(t_col)

    # First derivative dx/dt
    dx_dt = torch.autograd.grad(x.sum(), t_col, create_graph=True)[0]
    # Second derivative d²x/dt²
    d2x_dt2 = torch.autograd.grad(dx_dt.sum(), t_col, create_graph=True)[0]

    # PDE residual: d²x/dt² + ω²x = 0
    residual = d2x_dt2 + omega**2 * x
    loss_pde = (residual ** 2).mean()

    # Initial conditions: x(0)=1, dx/dt(0)=0
    t0 = torch.tensor([[0.0]], requires_grad=True)
    x0 = pinn_sho(t0)
    dx0 = torch.autograd.grad(x0, t0, create_graph=True)[0]

    loss_ic = (x0 - 1.0)**2 + dx0**2

    loss = loss_pde + 50 * loss_ic.squeeze()
    loss.backward()
    optimizer.step()
    losses_sho.append(loss.item())

    if epoch % 1000 == 0:
        print(f"Epoch {epoch:4d}  PDE: {loss_pde.item():.2e}  IC: {loss_ic.item():.2e}
↪")
```

```
Epoch    0  PDE: 7.42e-02  IC: 1.12e+00
Epoch 1000  PDE: 3.27e-01  IC: 4.71e-05
Epoch 2000  PDE: 2.09e-01  IC: 2.33e-05
Epoch 3000  PDE: 8.85e-02  IC: 4.58e-06
Epoch 4000  PDE: 7.69e-02  IC: 2.99e-06
```

```
# Compare with exact solution
t_test = torch.linspace(0, 4*np.pi/omega, 500).reshape(-1, 1)
with torch.no_grad():
    x_pinn = pinn_sho(t_test).numpy().flatten()

x_exact = np.cos(omega * t_test.numpy().flatten())

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(t_test.numpy(), x_exact, 'k--', lw=2, label=r'Exact: $\cos(\omega t)$')
axes[0].plot(t_test.numpy(), x_pinn, 'b-', lw=2, label='PINN')
axes[0].set_xlabel('t'); axes[0].set_ylabel('x(t)')
axes[0].set_title(f'Harmonic Oscillator ($\\omega$ = {omega})')
axes[0].legend(); axes[0].grid(alpha=0.3)

axes[1].semilogy(losses_sho, 'b-', alpha=0.7)
axes[1].set_xlabel('Epoch'); axes[1].set_ylabel('Loss')
axes[1].set_title('Training loss'); axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Max error: {np.max(np.abs(x_pinn - x_exact)):.4f}")
print("PINN solves the harmonic oscillator with just the equation + initial␣
↪conditions!")
```




```
Max error: 0.3572
PINN solves the harmonic oscillator with just the equation + initial conditions!
```

22.5.4 Example 3: PINN Solves the 1D Heat Equation (PDE)

Now the real power of PINNs — solving a **partial differential equation**.

The 1D heat equation from Lecture 11:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

with boundary conditions $u(0,t) = u(L,t) = 0$ and initial condition $u(x,0) = \sin(\pi x/L)$.

Exact solution: $$u(x,t) = \sin\left(\frac{\pi x}{L}\right) \exp\left(-\alpha \frac{\pi^2}{L^2} t\right)$$

Compare this with the finite-difference FTCS scheme from Lecture 11: the PINN needs **no grid** and handles boundaries naturally.

```
# PINN for the 1D heat equation
torch.manual_seed(42)

alpha_heat = 0.1  # thermal diffusivity
L_rod = 1.0       # rod length
T_final = 1.0     # simulation time

class PINN_Heat(nn.Module):
    """PINN for 1D heat equation: u_t = alpha * u_xx."""
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 64),    # input: (x, t)
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 1),    # output: u(x, t)
        )

    def forward(self, x, t):
        xt = torch.cat([x, t], dim=1)
        return self.net(xt)

pinn_heat = PINN_Heat()
optimizer = torch.optim.Adam(pinn_heat.parameters(), lr=0.001)

# Collocation points in the interior domain
N_col = 2000
x_col = torch.rand(N_col, 1) * L_rod
t_col = torch.rand(N_col, 1) * T_final
x_col.requires_grad_(True)
t_col.requires_grad_(True)

# Boundary condition points: u(0, t) = u(L, t) = 0
N_bc = 200
t_bc = torch.rand(N_bc, 1) * T_final
x_bc_left = torch.zeros(N_bc, 1)
x_bc_right = torch.ones(N_bc, 1) * L_rod

# Initial condition points: u(x, 0) = sin(pi*x/L)
N_ic = 200
```

(continues on next page)

(continued from previous page)

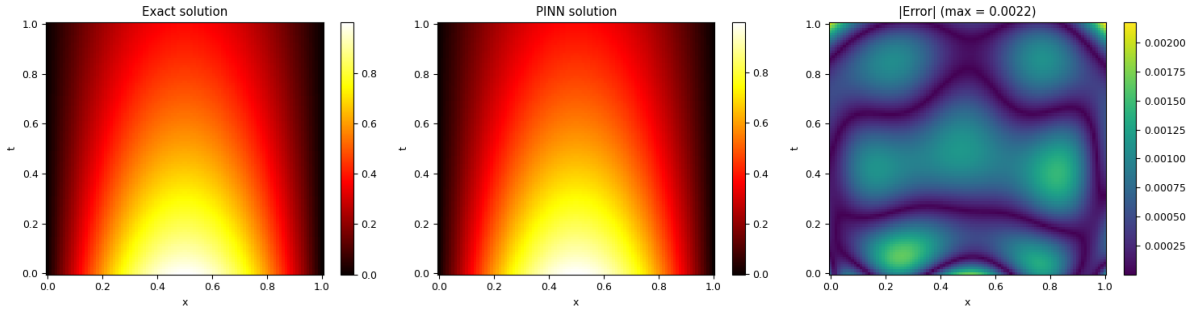
```
x_ic = torch.rand(N_ic, 1) * L_rod
t_ic = torch.zeros(N_ic, 1)
u_ic = torch.sin(np.pi * x_ic / L_rod)

print(f"Collocation points: {N_col}")
print(f"Boundary points: {2 * N_bc}")
print(f"Initial condition points: {N_ic}")
```

```
Collocation points: 2000
Boundary points: 400
Initial condition points: 200
```

```
# Train the heat equation PINN
losses_heat = []

for epoch in range(5000):
    optimizer.zero_grad()

    # --- PDE residual: u_t - alpha * u_xx = 0 ---
    u = pinn_heat(x_col, t_col)

    # du/dt
    u_t = torch.autograd.grad(u.sum(), t_col, create_graph=True)[0]
    # du/dx
    u_x = torch.autograd.grad(u.sum(), x_col, create_graph=True)[0]
    # d²u/dx²
    u_xx = torch.autograd.grad(u_x.sum(), x_col, create_graph=True)[0]

    residual = u_t - alpha_heat * u_xx
    loss_pde = (residual ** 2).mean()

    # --- Boundary conditions: u(0,t) = u(L,t) = 0 ---
    u_left = pinn_heat(x_bc_left, t_bc)
    u_right = pinn_heat(x_bc_right, t_bc)
    loss_bc = (u_left ** 2).mean() + (u_right ** 2).mean()

    # --- Initial condition: u(x,0) = sin(pi*x/L) ---
    u_init = pinn_heat(x_ic, t_ic)
    loss_ic = ((u_init - u_ic) ** 2).mean()

    # Total loss
    loss = loss_pde + 10 * loss_bc + 10 * loss_ic
    loss.backward()
    optimizer.step()
    losses_heat.append(loss.item())

    if epoch % 1000 == 0:
        print(f"Epoch {epoch:4d}  PDE: {loss_pde.item():.2e}  "
              f"BC: {loss_bc.item():.2e}  IC: {loss_ic.item():.2e}")
```

```
Epoch    0  PDE: 3.34e-03  BC: 9.68e-02  IC: 7.48e-01
Epoch 1000  PDE: 3.59e-04  BC: 9.78e-05  IC: 3.99e-05
Epoch 2000  PDE: 2.10e-04  BC: 1.10e-06  IC: 7.43e-07
Epoch 3000  PDE: 1.58e-04  BC: 1.10e-06  IC: 8.01e-07
Epoch 4000  PDE: 1.38e-04  BC: 8.61e-05  IC: 3.47e-05
```

```
# Visualise PINN solution vs exact solution
def exact_heat(x, t, alpha=alpha_heat, L=L_rod):
    return np.sin(np.pi * x / L) * np.exp(-alpha * (np.pi / L)**2 * t)

# Evaluate on a grid
nx, nt = 100, 100
x_grid = np.linspace(0, L_rod, nx)
t_grid = np.linspace(0, T_final, nt)
X_grid, T_grid = np.meshgrid(x_grid, t_grid)

x_flat = torch.tensor(X_grid.flatten(), dtype=torch.float32).reshape(-1, 1)
t_flat = torch.tensor(T_grid.flatten(), dtype=torch.float32).reshape(-1, 1)

pinn_heat.eval()
with torch.no_grad():
    u_pinn = pinn_heat(x_flat, t_flat).numpy().reshape(nt, nx)

u_exact = exact_heat(X_grid, T_grid)

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

im1 = axes[0].pcolormesh(X_grid, T_grid, u_exact, cmap='hot', shading='auto')
axes[0].set_xlabel('x'); axes[0].set_ylabel('t')
axes[0].set_title('Exact solution')
plt.colorbar(im1, ax=axes[0])

im2 = axes[1].pcolormesh(X_grid, T_grid, u_pinn, cmap='hot', shading='auto')
axes[1].set_xlabel('x'); axes[1].set_ylabel('t')
axes[1].set_title('PINN solution')
plt.colorbar(im2, ax=axes[1])

error = np.abs(u_pinn - u_exact)
im3 = axes[2].pcolormesh(X_grid, T_grid, error, cmap='viridis', shading='auto')
axes[2].set_xlabel('x'); axes[2].set_ylabel('t')
axes[2].set_title(f'|Error| (max = {error.max():.4f})')
plt.colorbar(im3, ax=axes[2])

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

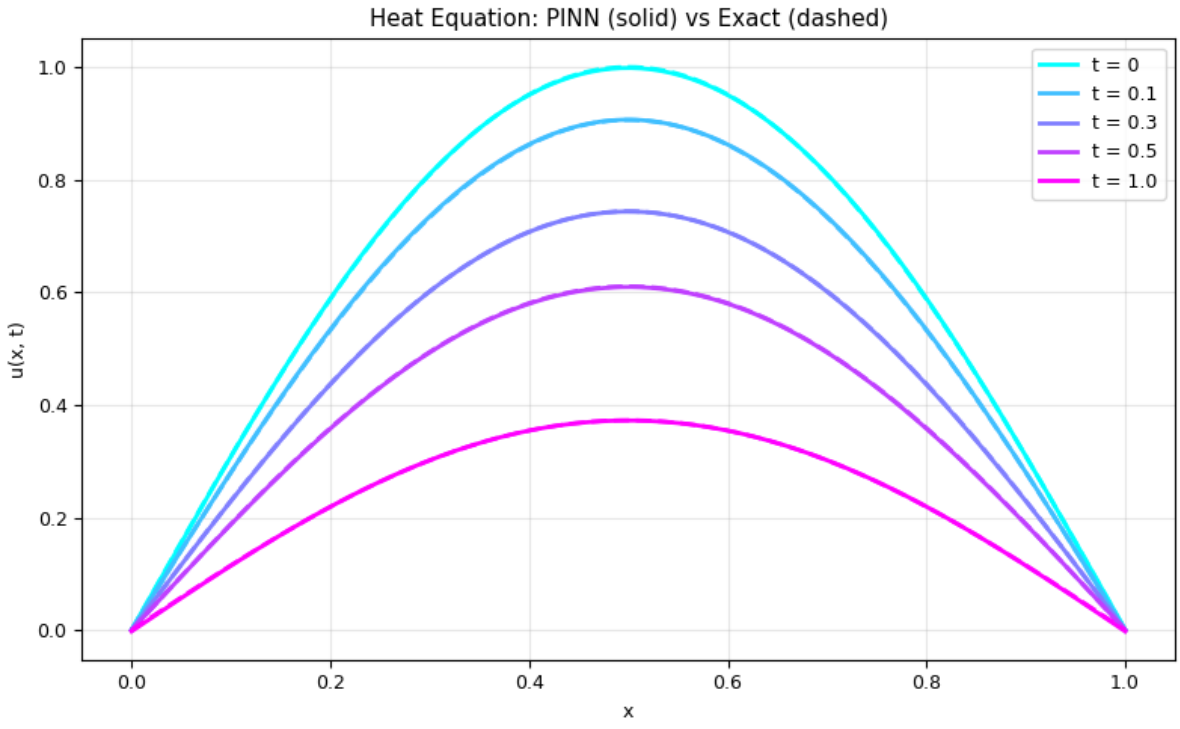



```
# Compare PINN vs exact at specific times
fig, ax = plt.subplots(figsize=(8, 5))
times = [0, 0.1, 0.3, 0.5, 1.0]
colors = plt.cm.cool(np.linspace(0, 1, len(times)))
```

(continues on next page)

(continued from previous page)

```
x_test = torch.linspace(0, L_rod, 200).reshape(-1, 1)

for t_val, color in zip(times, colors):
    t_test = torch.ones_like(x_test) * t_val
    with torch.no_grad():
        u_pinn_line = pinn_heat(x_test, t_test).numpy().flatten()
    u_exact_line = exact_heat(x_test.numpy().flatten(), t_val)

    ax.plot(x_test.numpy(), u_exact_line, '--', color=color, lw=2)
    ax.plot(x_test.numpy(), u_pinn_line, '-', color=color, lw=2,
            label=f't = {t_val}')

ax.set_xlabel('x')
ax.set_ylabel('u(x, t)')
ax.set_title('Heat Equation: PINN (solid) vs Exact (dashed)')
ax.legend()
ax.grid(alpha=0.3)
plt.tight_layout()
plt.show()

print("PINN solves the heat equation with NO grid and NO time-stepping!")
print("Compare with Lecture 11: FTCS scheme needed a grid and CFL stability condition.
 ↪")
```




```
PINN solves the heat equation with NO grid and NO time-stepping!
Compare with Lecture 11: FTCS scheme needed a grid and CFL stability condition.
```

22.5.5 PINN vs Traditional Methods

Aspect	Finite Differences (Lecture 11)	PINN
Grid	Fixed grid required	Mesh-free
Stability	CFL condition constrains Δt	No stability issues
Dimensions	Cost grows as N^d	Cost grows slowly with d
Accuracy	Controlled by grid resolution	Controlled by network size
Irregular domains	Hard	Easy
Inverse problems	Very hard	Natural — just add unknowns

PINNs excel when: high dimensions, irregular domains, inverse problems, sparse data.
Finite differences excel when: low dimensions, high accuracy needed, simple domains.

22.6 IV. Symmetry and Equivariance

Physics is full of **symmetries**. If we build them into the network, it needs less data and generalises better.

22.6.1 Key Concepts

Invariance: Output does not change when input is transformed. $\$f(T \cdot \mathbf{x}) = f(\mathbf{x})\$$

Example: Total energy is invariant to translation, rotation, and relabelling atoms.

Equivariance: Output transforms in the same way as the input. $\$f(T \cdot \mathbf{x}) = T \cdot f(\mathbf{x})\$$

Example: Forces rotate with the coordinate system.

Symmetry	Invariant quantities	Equivariant quantities
Translation	Energy, pressure	Forces, momentum
Rotation	Energy, charge	Forces, angular momentum, dipole
Permutation (atom swap)	Total energy	Per-atom energies
Time reversal	Energy	Velocity

```
# Demo: Building permutation invariance into a network
# Problem: predict total energy from 3 pairwise distances
# The energy must not change if we swap atoms → permutation invariant

# Naive approach: just feed distances to an MLP
# Problem: MLP(r12, r13, r23) != MLP(r13, r12, r23) in general!

class NaiveMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 32), nn.ReLU(),
            nn.Linear(32, 1)
        )
    def forward(self, x):
        return self.net(x)
```

(continues on next page)

(continued from previous page)

```
# Invariant approach: sort distances, or use symmetric aggregation
class SymmetricNN(nn.Module):
    """Process each distance independently, then sum (Deep Sets idea)."""
    def __init__(self):
        super().__init__()
        # Per-pair network
        self.phi = nn.Sequential(
            nn.Linear(1, 32), nn.ReLU(),
            nn.Linear(32, 32), nn.ReLU(),
        )
        # After summing
        self.rho = nn.Sequential(
            nn.Linear(32, 32), nn.ReLU(),
            nn.Linear(32, 1)
        )

    def forward(self, dists):
        # dists: (batch, n_pairs)
        # Process each distance independently
        per_pair = self.phi(dists.unsqueeze(-1))  # (batch, n_pairs, 32)
        # Sum over pairs (permutation invariant!)
        pooled = per_pair.sum(dim=1)  # (batch, 32)
        return self.rho(pooled)

# Test permutation invariance
torch.manual_seed(42)
naive = NaiveMLP()
symm = SymmetricNN()

test_input = torch.tensor([[1.0, 1.5, 2.0]])  # r12, r13, r23
permuted   = torch.tensor([[1.5, 1.0, 2.0]])  # r13, r12, r23 (swap atoms 2,3)

print("Naive MLP:")
print(f"  f(r12, r13, r23) = {naive(test_input).item():.4f}")
print(f"  f(r13, r12, r23) = {naive(permuted).item():.4f}")
print(f"  Different! Not invariant.\n")

print("Symmetric NN (Deep Sets):")
print(f"  f(r12, r13, r23) = {symm(test_input).item():.4f}")
print(f"  f(r13, r12, r23) = {symm(permuted).item():.4f}")
print(f"  Same! Permutation invariant by construction.")
```

```
Naive MLP:
  f(r12, r13, r23) = 0.8826
  f(r13, r12, r23) = 0.8449
  Different! Not invariant.

Symmetric NN (Deep Sets):
  f(r12, r13, r23) = 0.1557
  f(r13, r12, r23) = 0.1557
  Same! Permutation invariant by construction.
```

22.6.2 The Deep Sets Architecture

The **Deep Sets** theorem (Zaheer et al., 2017) states that any permutation-invariant function can be written as:

$$f(\{x_1, \ldots, x_n\}) = \rho\left(\sum_{i=1}^{n} \phi(x_i)\right)$$

where ϕ and ρ are learnable functions (neural networks).

This is exactly what `SymmetricNN` does above:

- ϕ: processes each element independently
- $\sum$: symmetric aggregation
- ρ: maps the aggregated representation to the output

This architecture is the foundation of **Graph Neural Networks** (Lecture 24).

22.7 V. Inverse Problems with PINNs

A powerful extension: what if we **don't know a parameter** in the PDE?

Given noisy measurements of $u(x, t)$, can we find the unknown diffusion coefficient α?

We make α a **trainable parameter** alongside the network weights!

```
# Inverse PINN: estimate diffusivity alpha from noisy observations
torch.manual_seed(42)

alpha_true = 0.1  # the value we want to recover

# Generate "experimental" observations with noise
N_obs = 100
x_obs = np.random.uniform(0.1, 0.9, N_obs)
t_obs = np.random.uniform(0.05, 0.8, N_obs)
u_obs = exact_heat(x_obs, t_obs, alpha=alpha_true) + 0.02 * np.random.randn(N_obs)

x_obs_t = torch.tensor(x_obs, dtype=torch.float32).reshape(-1, 1)
t_obs_t = torch.tensor(t_obs, dtype=torch.float32).reshape(-1, 1)
u_obs_t = torch.tensor(u_obs, dtype=torch.float32).reshape(-1, 1)

class InversePINN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 1),
        )
        # Unknown parameter — initialise far from true value
        self.log_alpha = nn.Parameter(torch.tensor(0.0))  # alpha = exp(0) = 1.0

    @property
    def alpha(self):
        return torch.exp(self.log_alpha)  # ensure positivity
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x, t):
        return self.net(torch.cat([x, t], dim=1))

inv_pinn = InversePINN()
optimizer = torch.optim.Adam(inv_pinn.parameters(), lr=0.001)

# Collocation points for PDE
x_col = torch.rand(1000, 1, requires_grad=True)
t_col = torch.rand(1000, 1, requires_grad=True)

alpha_history = []
losses_inv = []

for epoch in range(5000):
    optimizer.zero_grad()

    # PDE residual
    u = inv_pinn(x_col, t_col)
    u_t = torch.autograd.grad(u.sum(), t_col, create_graph=True)[0]
    u_x = torch.autograd.grad(u.sum(), x_col, create_graph=True)[0]
    u_xx = torch.autograd.grad(u_x.sum(), x_col, create_graph=True)[0]

    residual = u_t - inv_pinn.alpha * u_xx
    loss_pde = (residual ** 2).mean()

    # Data fitting
    u_pred = inv_pinn(x_obs_t, t_obs_t)
    loss_data = ((u_pred - u_obs_t) ** 2).mean()

    # BC and IC (same as before)
    t_bc = torch.rand(100, 1)
    loss_bc = (inv_pinn(torch.zeros(100, 1), t_bc)**2).mean() + \
              (inv_pinn(torch.ones(100, 1), t_bc)**2).mean()

    x_ic = torch.rand(100, 1)
    u_ic_true = torch.sin(np.pi * x_ic)
    loss_ic = ((inv_pinn(x_ic, torch.zeros(100, 1)) - u_ic_true)**2).mean()

    loss = loss_pde + 100 * loss_data + 10 * loss_bc + 10 * loss_ic
    loss.backward()
    optimizer.step()

    alpha_history.append(inv_pinn.alpha.item())
    losses_inv.append(loss.item())

    if epoch % 1000 == 0:
        print(f"Epoch {epoch:4d}  alpha = {inv_pinn.alpha.item():.4f}  "
              f"(true: {alpha_true})  loss: {loss.item():.2e}")

print(f"\nRecovered alpha = {inv_pinn.alpha.item():.4f}")
print(f"True alpha      = {alpha_true}")
print(f"Error: {abs(inv_pinn.alpha.item() - alpha_true)/alpha_true * 100:.1f}%")
```

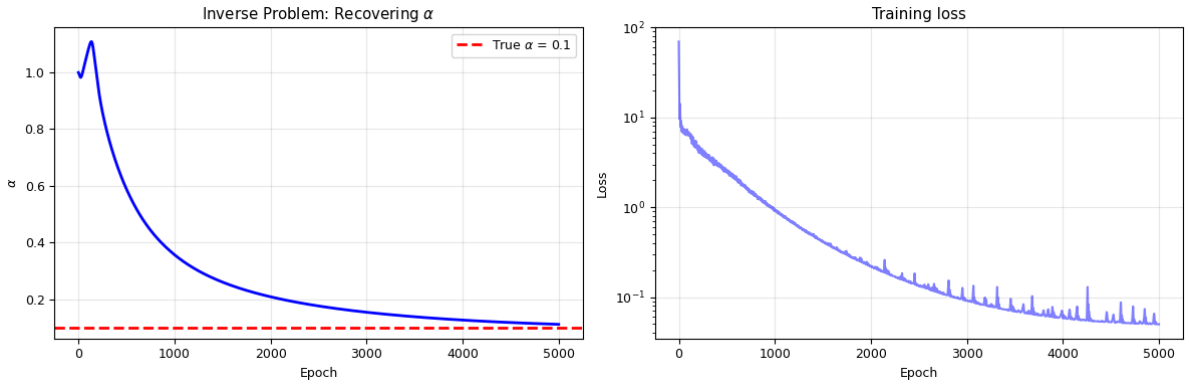
```
Epoch    0  alpha = 0.9990  (true: 0.1)  loss: 6.97e+01
Epoch 1000  alpha = 0.3567  (true: 0.1)  loss: 9.48e-01
Epoch 2000  alpha = 0.2086  (true: 0.1)  loss: 2.19e-01
```

(continues on next page)

(continued from previous page)

```
Epoch 3000  alpha = 0.1542  (true: 0.1)  loss: 9.06e-02
Epoch 4000  alpha = 0.1266  (true: 0.1)  loss: 5.97e-02

Recovered alpha = 0.1116
True alpha      = 0.1
Error: 11.6%
```

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(alpha_history, 'b-', lw=2)
axes[0].axhline(alpha_true, color='red', ls='--', lw=2, label=f'True $\\alpha$ =
 ↪{alpha_true}')
axes[0].set_xlabel('Epoch'); axes[0].set_ylabel('$\\alpha$')
axes[0].set_title('Inverse Problem: Recovering $\\alpha$')
axes[0].legend(); axes[0].grid(alpha=0.3)

axes[1].semilogy(losses_inv, 'b-', alpha=0.5)
axes[1].set_xlabel('Epoch'); axes[1].set_ylabel('Loss')
axes[1].set_title('Training loss'); axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("The PINN recovers the unknown diffusion coefficient from noisy observations!")
print("This is extremely powerful for experimental data analysis.")
```




```
The PINN recovers the unknown diffusion coefficient from noisy observations!
This is extremely powerful for experimental data analysis.
```

22.8 Summary

Architecture	Key Idea	Physics Application
CNN	Exploit spatial locality via filters	Lattice models, field data
Autoencoder	Compress to low-dim latent space	Discover order parameters
PINN	Embed PDE in loss function	Solve forward and inverse problems
Deep Sets	Symmetric aggregation → invariance	Permutation-invariant energy

Next lecture: We go deeper into symmetry-aware architectures with **Graph Neural Networks**, where the structure of

the physical system (atoms, bonds, interactions) is encoded directly in the network.

CHAPTER

TWENTYTHREE

LECTURE 23 — MACHINE LEARNING IV

23.1 Graph Neural Networks for Physics

23.2 Why Graphs?

Many physical systems are naturally described as **graphs** — a collection of nodes connected by edges:

Physical system	Nodes	Edges
Molecule	Atoms	Bonds / distance cutoff
Crystal	Atoms in unit cell	Nearest neighbours
Protein	Amino acid residues	Spatial proximity
Social network	People	Friendships
Electrical circuit	Components	Wires

The key insight: the **structure** of the system matters. An MLP treats inputs as a flat vector and ignores which atoms are close to which. A GNN respects the connectivity.

From Lecture 23: we built permutation-invariant networks using **Deep Sets** (process each element, then sum). A GNN generalises this by also considering **which elements are connected**.

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F

plt.rcParams['figure.figsize'] = [6, 4]
plt.rcParams['font.size'] = 9

print(f"PyTorch {torch.__version__}")
```

```
PyTorch 2.6.0
```

23.3 I. Graphs: The Data Structure

A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consists of:

- **Nodes** $\mathbf{V} = \{v_1, \ldots, v_N\}$ with features $\mathbf{h}_i \in \mathbb{R}^d$
- **Edges** $\mathbf{E} = \{(i, j)\}$ with optional features $\mathbf{e}_{ij} \in \mathbb{R}^k$
- **Neighbours** of node i: $\mathbf{N}(i) = \{j : (i, j) \in \mathbf{E}\}$

For molecular systems:

- **Node features**: Atomic number, element type (one-hot encoded)
- **Edge features**: Distance r_{ij}, bond type
- **Connectivity**: All atom pairs within a cutoff distance r_c

```
# Example: represent a water molecule as a graph
# H2O: O at center, two H atoms

# Atom positions (Angstroms)
positions = np.array([
    [0.000, 0.000, 0.000],  # O
    [0.757, 0.587, 0.000],  # H
    [-0.757, 0.587, 0.000], # H
])

atomic_numbers = [8, 1, 1]  # O, H, H
elements = ['O', 'H', 'H']

# Build graph: connect atoms within cutoff
cutoff = 2.0  # Angstroms

edges = []
edge_distances = []
for i in range(len(positions)):
    for j in range(len(positions)):
        if i != j:
            d = np.linalg.norm(positions[i] - positions[j])
            if d < cutoff:
                edges.append([i, j])
                edge_distances.append(d)

edges = np.array(edges)
edge_distances = np.array(edge_distances)

print("Water molecule graph:")
print(f"  Nodes: {len(positions)} atoms ({elements})")
print(f"  Edges: {len(edges)} connections")
for e, d in zip(edges, edge_distances):
    print(f"    {elements[e[0]]} ({e[0]}) -- {elements[e[1]]} ({e[1]}): {d:.3f} A")
```

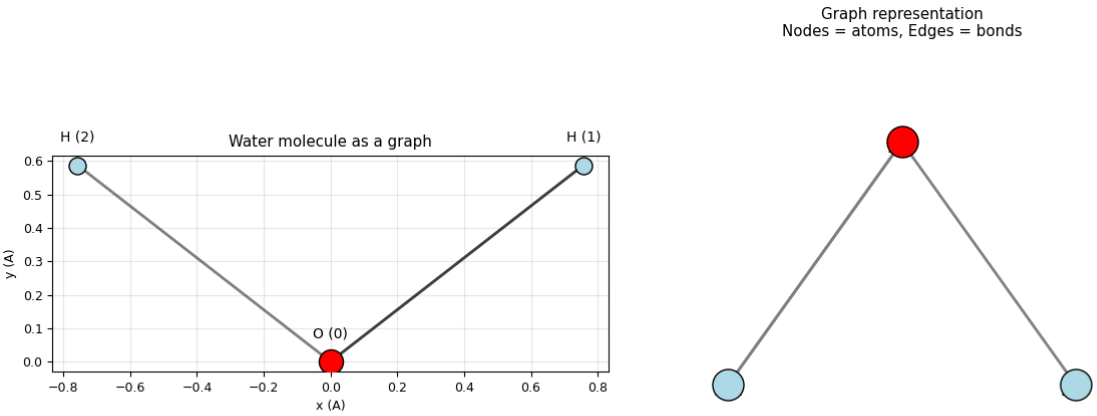
```
Water molecule graph:
  Nodes: 3 atoms (['O', 'H', 'H'])
  Edges: 6 connections
    O (0) -- H (1): 0.958 A
    O (0) -- H (2): 0.958 A
    H (1) -- O (0): 0.958 A
```

(continues on next page)

(continued from previous page)

```
    H (1) -- H (2): 1.514 A
    H (2) -- O (0): 0.958 A
    H (2) -- H (1): 1.514 A
```

```
# Visualise the molecular graph
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# 2D projection of molecule
ax = axes[0]
colors_atom = {'O': 'red', 'H': 'lightblue'}
sizes_atom = {'O': 300, 'H': 150}

for e in edges[:len(edges)//2]:  # undirected, so plot each pair once
    p1, p2 = positions[e[0]], positions[e[1]]
    ax.plot([p1[0], p2[0]], [p1[1], p2[1]], 'k-', lw=2, alpha=0.5)

for i, (pos, elem) in enumerate(zip(positions, elements)):
    ax.scatter(pos[0], pos[1], c=colors_atom[elem], s=sizes_atom[elem],
               edgecolors='black', zorder=5)
    ax.annotate(f'{elem} ({i})', (pos[0], pos[1]), fontsize=10,
                ha='center', va='bottom', xytext=(0, 15),
                textcoords='offset points')

ax.set_xlabel('x (A)'); ax.set_ylabel('y (A)')
ax.set_title('Water molecule as a graph')
ax.set_aspect('equal')
ax.grid(alpha=0.3)

# Show the graph representation schematically
ax2 = axes[1]
# Draw abstract graph
node_pos = {0: (0.5, 1), 1: (0, 0), 2: (1, 0)}
for e in edges[:len(edges)//2]:
    p1, p2 = node_pos[e[0]], node_pos[e[1]]
    ax2.annotate('', xy=p2, xytext=p1,
                 arrowprops=dict(arrowstyle='<->', color='gray', lw=2))

for i, elem in enumerate(elements):
    pos = node_pos[i]
    ax2.scatter(*pos, c=colors_atom[elem], s=500, edgecolors='black', zorder=5)
    ax2.text(pos[0], pos[1], f'{elem}\nZ={atomic_numbers[i]}',
             ha='center', va='center', fontsize=9, fontweight='bold')

ax2.set_xlim(-0.3, 1.3); ax2.set_ylim(-0.4, 1.4)
ax2.set_title('Graph representation\nNodes = atoms, Edges = bonds')
ax2.axis('off')

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



23.4 II. Message Passing: How GNNs Work

The core operation in a GNN is **message passing**. In each layer:

1. **Message**: Each node sends a message to its neighbours based on its current features
2. **Aggregate**: Each node collects messages from all its neighbours
3. **Update**: Each node updates its features based on the aggregated messages

Formally, one message-passing layer updates node i's features as:

$$\mathbf{h}_i^{(l+1)} = \phi\left(\mathbf{h}_i^{(l)}, \bigoplus_{j \in \mathcal{N}(i)} \psi\left(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}, \mathbf{e}_{ij}\right)\right)$$

where:

- ψ = **message function** (what information to send)
- $\bigoplus$ = **aggregation** (sum, mean, or max over neighbours)
- ϕ = **update function** (how to combine old features with messages)

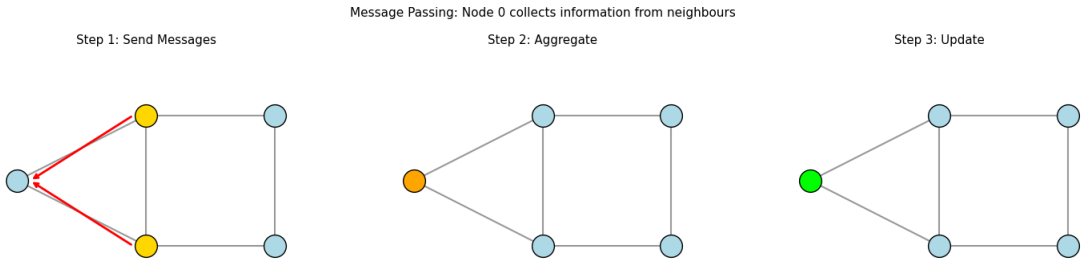
After L layers, each node's features contain information about its L-hop neighbourhood — this is the GNN's **receptive field**.

```
# Visualise message passing
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Simple 5-node graph
node_pos = {0: (0, 0), 1: (1, 0.5), 2: (1, -0.5), 3: (2, 0.5), 4: (2, -0.5)}
edges_vis = [(0, 1), (0, 2), (1, 3), (2, 4), (1, 2), (3, 4)]

titles = ['Step 1: Send Messages', 'Step 2: Aggregate', 'Step 3: Update']
highlights = [
    {1: 'gold', 2: 'gold'},  # messages from neighbours
    {0: 'orange'},           # aggregation at node 0
    {0: 'lime'},             # updated node
]
```

(continues on next page)

(continued from previous page)

```
for ax, title, hl in zip(axes, titles, highlights):
    # Draw edges
    for (i, j) in edges_vis:
        p1, p2 = node_pos[i], node_pos[j]
        ax.plot([p1[0], p2[0]], [p1[1], p2[1]], 'k-', lw=1.5, alpha=0.4)

    # Draw nodes
    for i, pos in node_pos.items():
        color = hl.get(i, 'lightblue')
        ax.scatter(*pos, s=400, c=color, edgecolors='black', zorder=5)
        ax.text(pos[0], pos[1], str(i), ha='center', va='center', fontsize=11)

    ax.set_title(title)
    ax.set_xlim(-0.5, 2.5); ax.set_ylim(-1, 1)
    ax.axis('off')

# Add arrows for messages in step 1
for i in [1, 2]:
    p1, p2 = node_pos[i], node_pos[0]
    axes[0].annotate('', xy=(p2[0]+0.1, p2[1]), xytext=(p1[0]-0.1, p1[1]),
                     arrowprops=dict(arrowstyle='->', color='red', lw=2))

plt.suptitle('Message Passing: Node 0 collects information from neighbours',␣
 ↪fontsize=11)
plt.tight_layout()
plt.show()
```



23.5 III. Building a GNN from Scratch

Let's implement a simple message-passing GNN in pure PyTorch.

Our GNN will predict the **total energy** of a system of Lennard-Jones atoms from their positions and types.

```
class MessagePassingLayer(nn.Module):
    """
    One layer of message passing.

    For each edge (i, j):
      message_ij = MLP([h_i, h_j, e_ij])
    For each node i:
      aggregated_i = sum of messages from neighbours
      h_i_new = MLP([h_i, aggregated_i])
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, node_dim, edge_dim, hidden_dim):
        super().__init__()
        # Message function: combines sender, receiver, and edge features
        self.message_mlp = nn.Sequential(
            nn.Linear(2 * node_dim + edge_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, hidden_dim),
        )
        # Update function: combines old features with aggregated messages
        self.update_mlp = nn.Sequential(
            nn.Linear(node_dim + hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, node_dim),
        )

    def forward(self, h, edge_index, edge_attr):
        """
        h: (N, node_dim) node features
        edge_index: (2, E) source and target node indices
        edge_attr: (E, edge_dim) edge features
        """
        src, dst = edge_index  # source and destination nodes
        N = h.size(0)

        # 1. Compute messages
        msg_input = torch.cat([h[src], h[dst], edge_attr], dim=1)
        messages = self.message_mlp(msg_input)  # (E, hidden_dim)

        # 2. Aggregate messages (sum for each target node)
        agg = torch.zeros(N, messages.size(1), device=h.device)
        agg.index_add_(0, dst, messages)  # sum messages by target

        # 3. Update node features
        update_input = torch.cat([h, agg], dim=1)
        h_new = h + self.update_mlp(update_input)  # residual connection

        return h_new

print("MessagePassingLayer defined.")
print("Key operations: message (per edge) → aggregate (per node) → update (per node)")
```

```
MessagePassingLayer defined.
Key operations: message (per edge) → aggregate (per node) → update (per node)
```

```
class SimpleGNN(nn.Module):
    """
    Complete GNN for predicting a scalar property (like energy)
    from atomic positions.
    """
    def __init__(self, n_elements=3, node_dim=32, edge_dim=16, n_layers=3):
        super().__init__()

        # Embed atomic number into a learnable vector
        self.atom_embedding = nn.Embedding(n_elements + 1, node_dim)

        # Radial basis functions to encode distances
```

(continues on next page)

(continued from previous page)

```
        self.n_rbf = edge_dim
        self.rbf_centers = nn.Parameter(
            torch.linspace(0.5, 5.0, edge_dim), requires_grad=False
        )
        self.rbf_width = 0.5

        # Message passing layers
        self.layers = nn.ModuleList([
            MessagePassingLayer(node_dim, edge_dim, node_dim)
            for _ in range(n_layers)
        ])

        # Output: per-atom energy, then sum
        self.output_mlp = nn.Sequential(
            nn.Linear(node_dim, node_dim),
            nn.SiLU(),
            nn.Linear(node_dim, 1),
        )

    def radial_basis(self, distances):
        """Expand distances into radial basis functions (Gaussian)."""
        return torch.exp(-(distances.unsqueeze(-1) - self.rbf_centers)**2
                         / (2 * self.rbf_width**2))

    def forward(self, z, positions, edge_index, batch=None):
        """
        z: (N,) atomic numbers
        positions: (N, 3) coordinates
        edge_index: (2, E) graph connectivity
        batch: (N,) which graph each atom belongs to (for batching)
        """
        # Initial node features from atom type
        h = self.atom_embedding(z)  # (N, node_dim)

        # Compute edge features from distances
        src, dst = edge_index
        diff = positions[dst] - positions[src]  # (E, 3)
        distances = torch.norm(diff, dim=1)      # (E,)
        edge_attr = self.radial_basis(distances)  # (E, n_rbf)

        # Message passing
        for layer in self.layers:
            h = layer(h, edge_index, edge_attr)

        # Per-atom energy
        atom_energy = self.output_mlp(h).squeeze(-1)  # (N,)

        # Sum per-atom energies to get total energy per molecule
        if batch is None:
            return atom_energy.sum()
        else:
            # Scatter-add by graph index
            n_graphs = batch.max().item() + 1
            energy = torch.zeros(n_graphs, device=h.device)
            energy.index_add_(0, batch, atom_energy)
            return energy
```

(continues on next page)

(continued from previous page)

```
gnn = SimpleGNN()
n_params = sum(p.numel() for p in gnn.parameters())
print(f"SimpleGNN: {n_params} parameters")
print(f"Architecture: atom embedding → {len(gnn.layers)} message-passing layers → per-
↪atom MLP → sum")
```

```
SimpleGNN: 21585 parameters
Architecture: atom embedding → 3 message-passing layers → per-atom MLP → sum
```

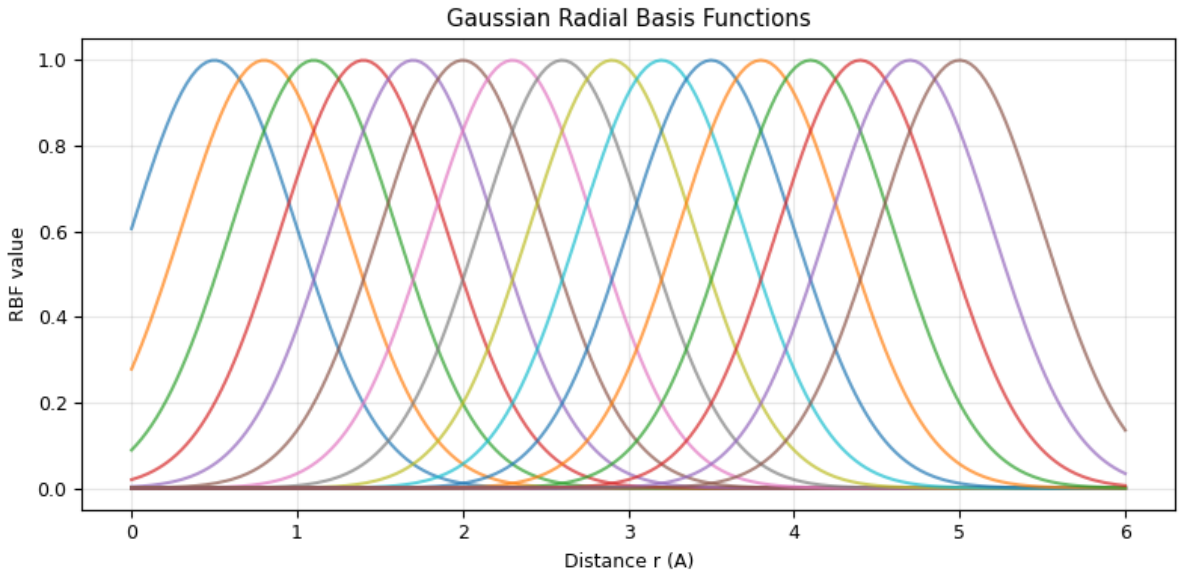
23.5.1 Radial Basis Functions

Distances are continuous, but neural networks work best with **expanded representations**. We use Gaussian radial basis functions (RBFs):

$$\text{RBF}_k(r) = \exp\left(-\frac{(r-\mu_k)^2}{2\sigma^2}\right)$$

This turns a single distance into a vector of K features, each "tuned" to a different distance range.

```
# Visualise radial basis functions
r = torch.linspace(0, 6, 200)
centers = torch.linspace(0.5, 5.0, 16)
width = 0.5

rbf_values = torch.exp(-(r.unsqueeze(-1) - centers)**2 / (2 * width**2))

plt.figure(figsize=(8, 4))
for i in range(16):
    plt.plot(r.numpy(), rbf_values[:, i].numpy(), alpha=0.7)

plt.xlabel('Distance r (A)')
plt.ylabel('RBF value')
plt.title('Gaussian Radial Basis Functions')
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()

print(f"A distance r = 2.5 A is encoded as a {len(centers)}-dim vector:")
r_test = torch.tensor([2.5])
encoded = torch.exp(-(r_test.unsqueeze(-1) - centers)**2 / (2 * width**2))
print(f"  {encoded.numpy().round(3).flatten()}")
```




```
A distance r = 2.5 A is encoded as a 16-dim vector:
  [0.    0.003 0.02  0.089 0.278 0.607 0.923 0.98  0.726 0.375 0.135 0.034
 0.006 0.001 0.    0.   ]
```

23.6 IV. Training the GNN: Lennard-Jones Clusters

Let's train our GNN to predict the total energy of small LJ clusters (continuing from Lecture 22).

```python
def lj_energy(positions, epsilon=1.0, sigma=1.0):
    """Total LJ energy for a cluster."""
    N = len(positions)
    E = 0.0
    for i in range(N):
        for j in range(i+1, N):
            r = np.linalg.norm(positions[i] - positions[j])
            sr6 = (sigma / r) ** 6
            E += 4 * epsilon * (sr6**2 - sr6)
    return E

def build_graph(positions, cutoff=3.5):
    """Build edge list from positions within cutoff."""
    N = len(positions)
    src, dst = [], []
    for i in range(N):
        for j in range(N):
            if i != j:
                r = np.linalg.norm(positions[i] - positions[j])
                if r < cutoff:
                    src.append(i)
                    dst.append(j)
    return np.array([src, dst])

# Generate dataset of 4-atom LJ clusters
np.random.seed(42)
```

(continues on next page)

(continued from previous page)

```
n_samples = 2000
n_atoms = 4

dataset = []
for _ in range(n_samples * 2):  # generate extra, filter bad ones
    pos = np.random.randn(n_atoms, 3) * 0.8
    # Check minimum distance
    dists = []
    for i in range(n_atoms):
        for j in range(i+1, n_atoms):
            dists.append(np.linalg.norm(pos[i] - pos[j]))
    if min(dists) < 0.8:
        continue

    E = lj_energy(pos)
    edge_index = build_graph(pos)

    dataset.append({
        'positions': pos.astype(np.float32),
        'z': np.ones(n_atoms, dtype=np.int64),  # all same atom type
        'energy': E,
        'edge_index': edge_index.astype(np.int64),
    })

    if len(dataset) >= n_samples:
        break

print(f"Generated {len(dataset)} valid clusters")
print(f"Energy range: [{min(d['energy'] for d in dataset):.2f}, {max(d['energy'] for␣
↪d in dataset):.2f}]")
```

```
Generated 2000 valid clusters
Energy range: [-4.36, 77.49]
```

```
# Simple batching for GNNs
# Key trick: combine multiple graphs into one big graph with a `batch` vector

def collate_graphs(graph_list):
    """Batch multiple graphs into one big graph."""
    all_z = []
    all_pos = []
    all_edges = []
    all_energy = []
    all_batch = []

    node_offset = 0
    for i, g in enumerate(graph_list):
        N = len(g['z'])
        all_z.append(torch.tensor(g['z']))
        all_pos.append(torch.tensor(g['positions']))
        all_edges.append(torch.tensor(g['edge_index']) + node_offset)
        all_energy.append(g['energy'])
        all_batch.append(torch.full((N,), i, dtype=torch.long))
        node_offset += N

    return {
```

(continues on next page)

(continued from previous page)

```
        'z': torch.cat(all_z),
        'positions': torch.cat(all_pos),
        'edge_index': torch.cat(all_edges, dim=1),
        'energy': torch.tensor(all_energy, dtype=torch.float32),
        'batch': torch.cat(all_batch),
    }

# Quick test
test_batch = collate_graphs(dataset[:3])
print("Batched graph:")
print(f"  Total nodes: {test_batch['z'].size(0)} (3 molecules x {n_atoms} atoms)")
print(f"  Total edges: {test_batch['edge_index'].size(1)}")
print(f"  Batch vector: {test_batch['batch']}")
print(f"  Energies: {test_batch['energy']}")
```

```
Batched graph:
  Total nodes: 12 (3 molecules x 4 atoms)
  Total edges: 36
  Batch vector: tensor([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2])
  Energies: tensor([56.1277, -0.7629, -1.3259])
```

```
# Train the GNN
torch.manual_seed(42)

# Split data
n_train = int(0.8 * len(dataset))
train_data = dataset[:n_train]
test_data = dataset[n_train:]

# Normalise energies
E_mean = np.mean([d['energy'] for d in train_data])
E_std = np.std([d['energy'] for d in train_data])
for d in dataset:
    d['energy_norm'] = (d['energy'] - E_mean) / E_std

# Model
gnn = SimpleGNN(n_elements=3, node_dim=32, edge_dim=16, n_layers=3)
optimizer = torch.optim.Adam(gnn.parameters(), lr=0.002)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=20,␣
 ↪factor=0.5)

batch_size = 32
train_losses, val_losses = [], []

for epoch in range(200):
    gnn.train()
    np.random.shuffle(train_data)
    epoch_loss = 0
    n_batches = 0

    for i in range(0, len(train_data), batch_size):
        batch = collate_graphs(train_data[i:i+batch_size])
        optimizer.zero_grad()

        E_pred = gnn(batch['z'], batch['positions'], batch['edge_index'], batch['batch
 ↪'])
```

(continues on next page)

(continued from previous page)

```
        E_true = torch.tensor([d['energy_norm'] for d in train_data[i:i+batch_size]],
                              dtype=torch.float32)

        loss = F.mse_loss(E_pred, E_true)
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        n_batches += 1

    train_losses.append(epoch_loss / n_batches)

    # Validation
    gnn.eval()
    with torch.no_grad():
        val_batch = collate_graphs(test_data)
        E_pred_val = gnn(val_batch['z'], val_batch['positions'],
                         val_batch['edge_index'], val_batch['batch'])
        E_true_val = torch.tensor([d['energy_norm'] for d in test_data],
                                  dtype=torch.float32)
        val_loss = F.mse_loss(E_pred_val, E_true_val).item()
    val_losses.append(val_loss)
    scheduler.step(val_loss)

    if epoch % 40 == 0:
        print(f"Epoch {epoch:3d}  Train: {train_losses[-1]:.5f}  Val: {val_loss:.5f}")

print(f"\nFinal val loss: {val_losses[-1]:.5f}")
```

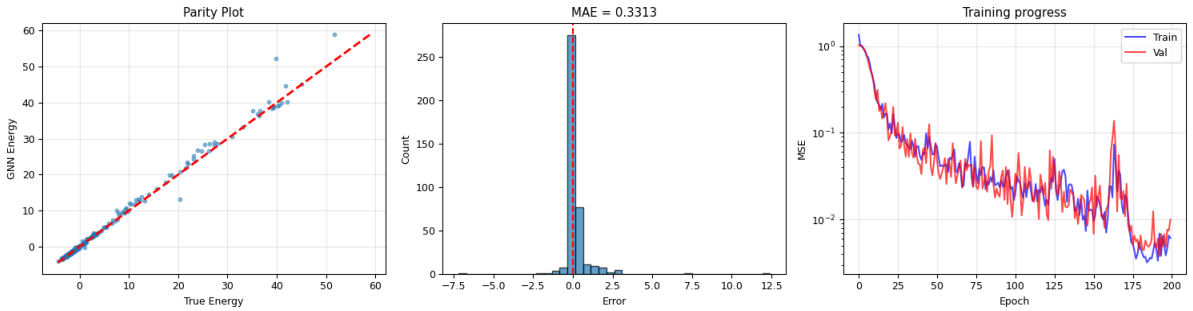
```
Epoch   0  Train: 1.35160  Val: 1.02464
Epoch  40  Train: 0.04884  Val: 0.03346
Epoch  80  Train: 0.03977  Val: 0.02687
Epoch 120  Train: 0.01684  Val: 0.01145
Epoch 160  Train: 0.02407  Val: 0.01994

Final val loss: 0.01001
```

```
# Evaluate: parity plot
gnn.eval()
with torch.no_grad():
    val_batch = collate_graphs(test_data)
    E_pred = gnn(val_batch['z'], val_batch['positions'],
                 val_batch['edge_index'], val_batch['batch']).numpy()

# Un-normalise
E_pred_real = E_pred * E_std + E_mean
E_true_real = np.array([d['energy'] for d in test_data])

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Parity plot
axes[0].scatter(E_true_real, E_pred_real, s=10, alpha=0.5)
lims = [min(E_true_real.min(), E_pred_real.min()),
        max(E_true_real.max(), E_pred_real.max())]
axes[0].plot(lims, lims, 'r--', lw=2)
axes[0].set_xlabel('True Energy'); axes[0].set_ylabel('GNN Energy')
```

(continues on next page)

(continued from previous page)

```
axes[0].set_title('Parity Plot'); axes[0].grid(alpha=0.3)

# Error distribution
errors = E_pred_real - E_true_real
axes[1].hist(errors, bins=40, edgecolor='black', alpha=0.7)
axes[1].set_xlabel('Error'); axes[1].set_ylabel('Count')
axes[1].set_title(f'MAE = {np.abs(errors).mean():.4f}')
axes[1].axvline(0, color='red', ls='--')

# Training curve
axes[2].semilogy(train_losses, 'b-', alpha=0.7, label='Train')
axes[2].semilogy(val_losses, 'r-', alpha=0.7, label='Val')
axes[2].set_xlabel('Epoch'); axes[2].set_ylabel('MSE')
axes[2].set_title('Training progress')
axes[2].legend(); axes[2].grid(alpha=0.3)

plt.tight_layout()
plt.show()

from sklearn.metrics import r2_score
print(f"R² = {r2_score(E_true_real, E_pred_real):.4f}")
print(f"MAE = {np.abs(errors).mean():.4f} epsilon")
```




```
R² = 0.9902
MAE = 0.3313 epsilon
```

23.6.1 Why GNN over MLP?

Property	MLP (Lecture 22)	GNN
Input	Fixed-size vector	Variable-size graph
Permutation invariance	Must sort features	Built in (sum aggregation)
Scalability	Fixed N atoms	Any N atoms (same model!)
Locality	All-to-all	Only neighbours interact
Interpretability	Black box	Per-atom contributions

23.7 V. Equivariant GNNs: Encoding Rotational Symmetry

Our simple GNN uses **distances** as edge features — distances are rotation-invariant, so the predicted energy is rotation-invariant. Good!

But what about **forces**? Forces are vectors that **rotate with the system**. They are **equivariant**, not invariant.

23.7.1 The Problem

If we rotate the molecule by R:

- Energy should stay the same: $E(R\mathbf{r}) = E(\mathbf{r})$ (invariant)
- Forces should rotate: $\mathbf{F}(R\mathbf{r}) = R\,\mathbf{F}(\mathbf{r})$ (equivariant)

Option 1: Predict energy E, then compute forces via autograd: $\mathbf{F}_i = -\nabla_{\mathbf{r}_i} E$. This is guaranteed equivariant (we used this in Lecture 22).

Option 2: Build equivariance into the network itself — this is what modern architectures like **NequIP** and **MACE** do.

```
# Demo: verifying rotation invariance of our GNN
torch.manual_seed(42)
gnn.eval()

# Take a test molecule
test_mol = test_data[0]
pos = torch.tensor(test_mol['positions'])
z = torch.tensor(test_mol['z'])
edge_idx = torch.tensor(test_mol['edge_index'])

# Random 3D rotation matrix
def random_rotation():
    """Generate a random 3D rotation matrix."""
    # QR decomposition of random matrix gives uniform rotation
    M = torch.randn(3, 3)
    Q, R = torch.linalg.qr(M)
    # Ensure proper rotation (det = +1)
    Q = Q * torch.sign(torch.diag(R))
    if torch.det(Q) < 0:
        Q[:, 0] *= -1
    return Q

with torch.no_grad():
    E_original = gnn(z, pos, edge_idx).item()

    print(f"Original energy: {E_original:.6f}")
    print("\nRotated energies:")
    for i in range(5):
        R = random_rotation()
        pos_rotated = pos @ R.T  # rotate all atoms
        E_rotated = gnn(z, pos_rotated, edge_idx).item()
        print(f"  Rotation {i+1}: {E_rotated:.6f}  (diff: {abs(E_rotated - E_
↪original):.2e})")

print("\nThe GNN prediction is rotation-invariant (by construction)!")
print("This is because we only use distances, which are rotation-invariant.")
```

```
Original energy: -0.477264

Rotated energies:
  Rotation 1: -0.477264  (diff: 1.79e-07)
  Rotation 2: -0.477264  (diff: 0.00e+00)
  Rotation 3: -0.477264  (diff: 1.79e-07)
  Rotation 4: -0.477264  (diff: 1.79e-07)
  Rotation 5: -0.477264  (diff: 2.98e-07)

The GNN prediction is rotation-invariant (by construction)!
This is because we only use distances, which are rotation-invariant.
```

```python
# Computing forces via autograd (equivariant by construction)
pos_grad = pos.clone().requires_grad_(True)

E = gnn(z, pos_grad, edge_idx)
E.backward()

forces = -pos_grad.grad  # F = -dE/dr

print("Forces from autograd:")
for i, (elem, f) in enumerate(zip(['A', 'A', 'A', 'A'], forces)):
    print(f"  Atom {i}: F = [{f[0]:.4f}, {f[1]:.4f}, {f[2]:.4f}]")

# Verify equivariance: rotate, then compute forces
R = random_rotation()
pos_rot = (pos @ R.T).clone().requires_grad_(True)

E_rot = gnn(z, pos_rot, edge_idx)
E_rot.backward()
forces_rot = -pos_rot.grad

# Expected: forces_rot = R @ forces
forces_expected = forces @ R.T
error = (forces_rot - forces_expected).abs().max().item()

print(f"\nForce equivariance error: {error:.2e}")
print("Forces transform correctly under rotation!")
```

```
Forces from autograd:
  Atom 0: F = [1.0334, 1.0510, 0.4401]
  Atom 1: F = [0.7535, 0.1950, 0.4841]
  Atom 2: F = [-0.6574, -0.1661, -0.6888]
  Atom 3: F = [-1.1295, -1.0799, -0.2353]

Force equivariance error: 4.95e-06
Forces transform correctly under rotation!
```

23.8 VI. The Landscape of Modern GNN Architectures

Our simple GNN is a good starting point, but research has produced many more powerful architectures:

23.8.1 Invariant Models (use distances only)

Model	Key Idea	Paper
SchNet (2017)	Continuous convolutions on distances	Schütt et al.
DimeNet (2020)	Adds angles between triplets of atoms	Gasteiger et al.
GemNet (2021)	Adds dihedral angles (4-body interactions)	Gasteiger et al.

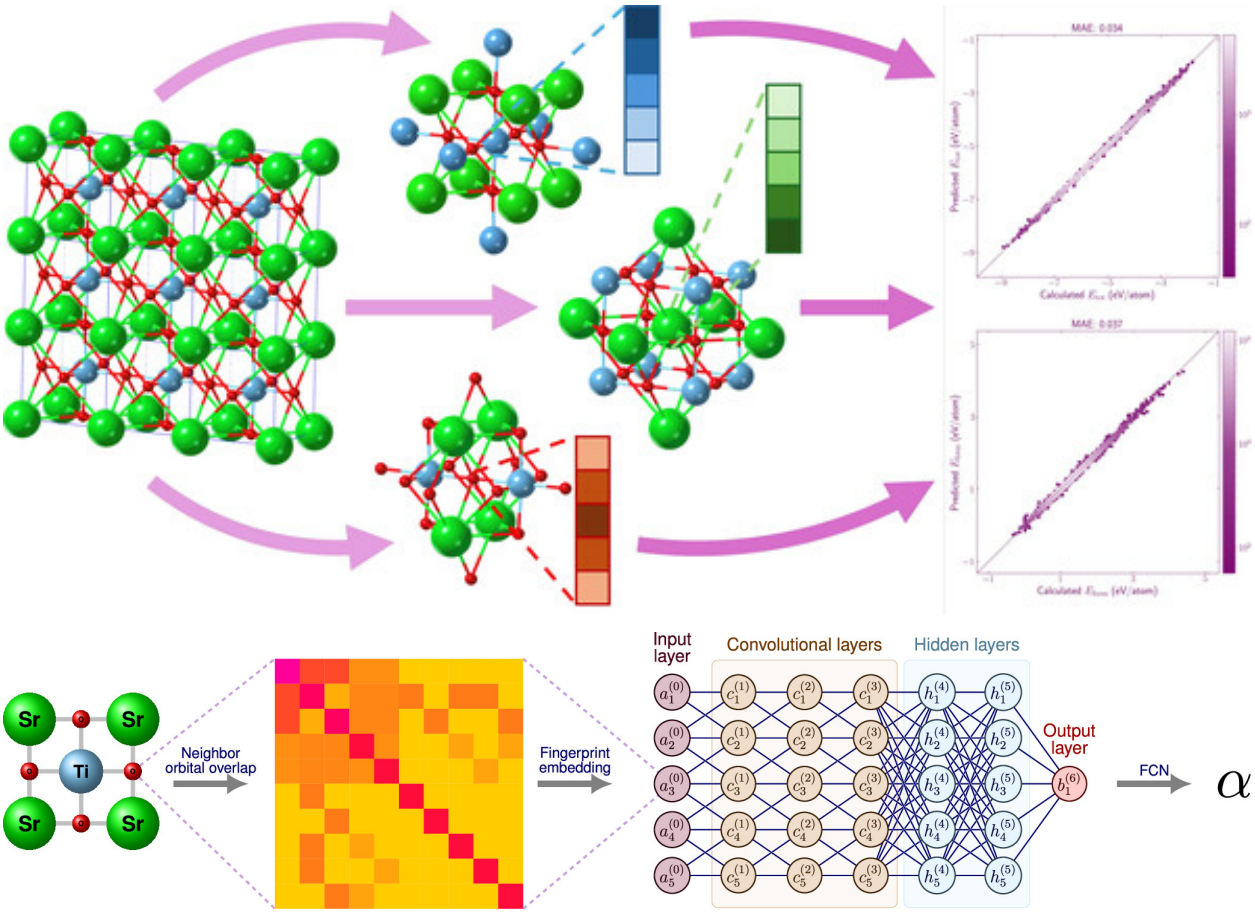
23.8.2 Equivariant Models (use directional information)

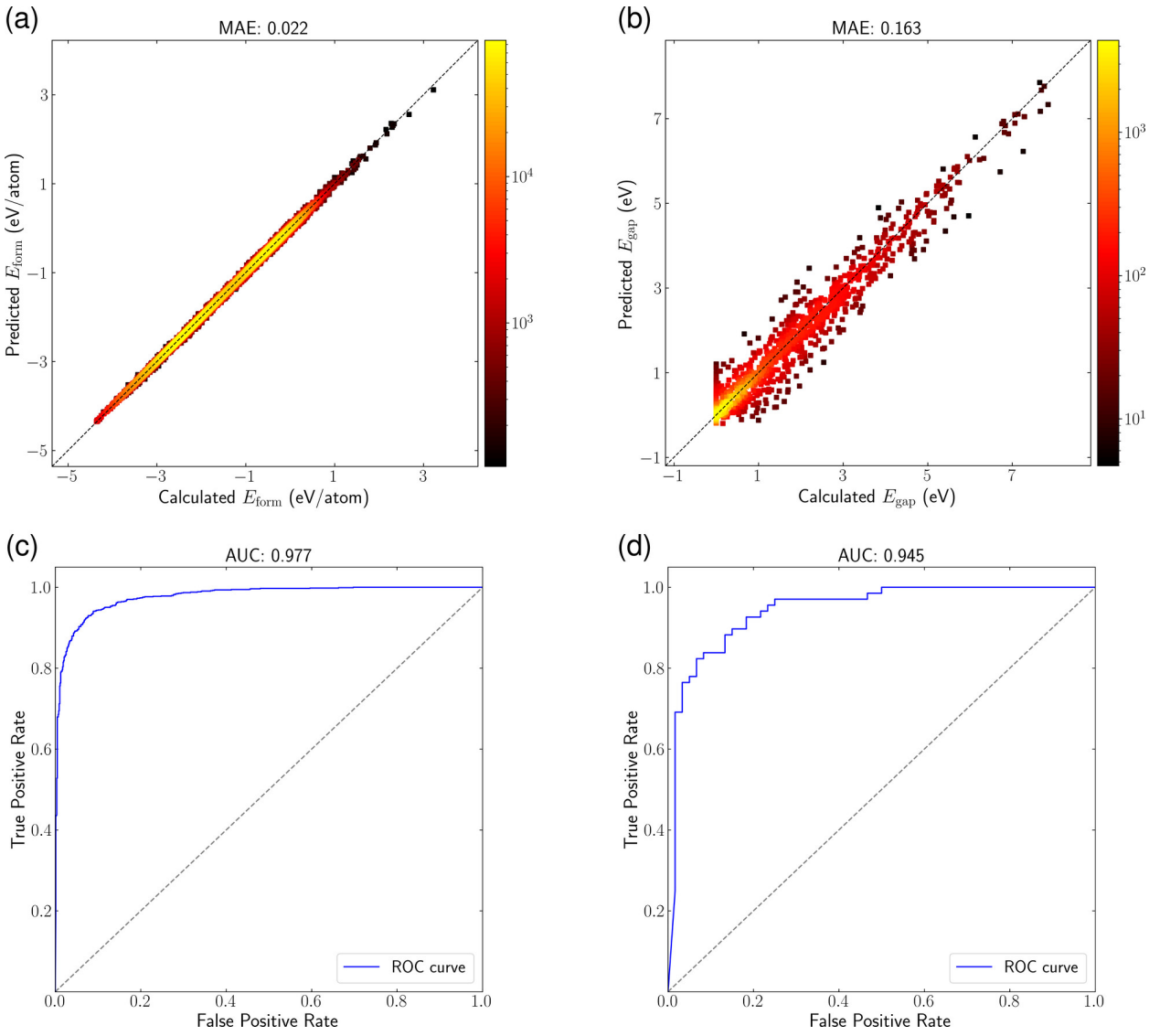
Model	Key Idea	Paper
PaiNN (2021)	Equivariant message passing with vectors	Schütt et al.
NequIP (2022)	E(3)-equivariant with spherical harmonics	Batzner et al.
MACE (2022)	Multi-body equivariant messages	Batatia et al.
EquiformerV2 (2023)	Equivariant transformer	Liao et al.
EOSnet (2025)	Equivariant orbital-based many-body features (our group)	Tao and Zhu

23.8.3 The Key Progression

```
Distances only     →  + Angles          →  + Directions (vectors)
(SchNet)              (DimeNet)             (NequIP, MACE)

2-body info        →  3-body info       →  Many-body info
r_ij                  r_ij, θ_ijk           Equivariant tensor products
```

More geometric information → better accuracy → but more computation.


Sr
Sr
Ti
Sr
Sr
Neighbor
orbital overlap
Fingerprint
embedding
Input
layer
Convolutional layers
Hidden layers
Output
layer
FCN
α



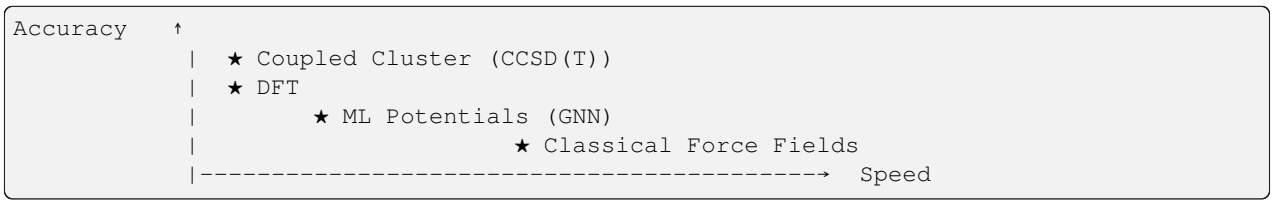
Evaluation for regression and binary classification tasks on the performance of our EOSnet model. (a) Parity plot for formation energy predictions on 131,240 data points from the Materials Project, with MAE of 0.022 eV/atom. (b) Parity plot for the prediction of electronic band gap (Eg) using 19,393 data points from the Materials Project, showing an MAE of 0.163 eV. (c) ROC curve for metal/nonmetal classification, achieving an AUC of 0.977. (d) ROC curve for dynamically stable/unstable classification on 1,335 guest-atom-substituted type-VII boron-carbide clathrates (MB6–xCx, x from 1 to 5), achieving an AUC of 0.945.

EOSnet: https://github.com/Rutgers-ZRG/EosNet

23.9 VII. Real-World Impact: ML Interatomic Potentials

GNN-based interatomic potentials are now a major tool in computational materials science and chemistry:

23.9.1 The Speed–Accuracy Trade-off



ML potentials achieve **DFT-level accuracy** at **force-field speed**:

- DFT: ~hours for 100 atoms
- ML potential: ~milliseconds for 100 atoms
- Speedup: 10^4 – 10^6 times

23.9.2 Applications

Application	Method	Impact
Molecular dynamics	MACE, NequIP	Simulate millions of timesteps at DFT accuracy
Drug discovery	SchNet, DimeNet	Screen molecular properties
Materials design	MEGNet, CGCNN	Predict crystal properties
Catalysis	GemNet, EquiformerV2	Find optimal catalyst surfaces
Protein folding	GNNs in AlphaFold	Predict 3D structure from sequence

23.9.3 Universal ML Potentials

Recent "foundation models" for atomic systems are trained on massive DFT datasets and work across the periodic table:

- **MACE-MP-0** (2023): Trained on Materials Project data, works for any element
- **CHGNet** (2023): Includes charge information
- **M3GNet** (2022): Universal potential for materials

These can be used as starting points and fine-tuned for specific systems.

23.10 VIII. Using Pre-trained GNNs with ASE

In practice, you often use pre-trained GNN potentials rather than training from scratch. The **ASE** (Atomic Simulation Environment) library provides a convenient interface.

Here is the typical workflow (not executed in class, but important to know):

```
from ase import Atoms
from ase.optimize import BFGS
## Example with MACE (if installed)
## from mace.calculators import mace_mp

## Create a molecule
water = Atoms('H2O', positions=[
    [0.0, 0.0, 0.0],
    [0.757, 0.587, 0.0],
    [-0.757, 0.587, 0.0]
])

## Attach ML calculator
## calc = mace_mp(model="medium", device="cpu")
## water.calc = calc

## Get energy and forces
## E = water.get_potential_energy()
## F = water.get_forces()

## Optimise geometry
## opt = BFGS(water)
## opt.run(fmax=0.01)
```

This is the same interface as DFT calculators — you can swap between ML potentials and DFT seamlessly!

23.11 IX. GNN Beyond Energy: Property Prediction

GNNs can predict many molecular/material properties beyond energy. Let's build a simple property predictor.

```
# Create a synthetic dataset of "molecules" with different atom types
# and a target property that depends on composition and geometry

np.random.seed(42)

def synthetic_property(atomic_numbers, positions):
    """A synthetic property that depends on composition and geometry.
    Mimics something like a dipole moment or HOMO-LUMO gap."""
    N = len(atomic_numbers)
    # Composition term
    comp = sum(z**0.5 for z in atomic_numbers) / N
    # Geometry term (average nearest-neighbour distance)
    dists = []
    for i in range(N):
        min_d = float('inf')
        for j in range(N):
            if i != j:
                d = np.linalg.norm(positions[i] - positions[j])
```

(continues on next page)

(continued from previous page)

```
                min_d = min(min_d, d)
        dists.append(min_d)
    avg_nn = np.mean(dists)
    # Property = f(composition, geometry) + noise
    return comp * avg_nn + 0.1 * np.random.randn()

# Generate molecules with 3-6 atoms, types 1-3
mol_dataset = []
for _ in range(1500):
    n_atoms = np.random.randint(3, 7)
    z = np.random.randint(1, 4, size=n_atoms)  # atom types 1, 2, 3
    pos = np.random.randn(n_atoms, 3).astype(np.float32) * 1.2

    # Check distances
    min_dist = float('inf')
    for i in range(n_atoms):
        for j in range(i+1, n_atoms):
            min_dist = min(min_dist, np.linalg.norm(pos[i] - pos[j]))
    if min_dist < 0.5:
        continue

    prop = synthetic_property(z, pos)
    edge_index = build_graph(pos, cutoff=3.5)

    mol_dataset.append({
        'z': z.astype(np.int64),
        'positions': pos,
        'edge_index': edge_index.astype(np.int64),
        'energy': prop,  # reuse 'energy' key
    })

print(f"Generated {len(mol_dataset)} molecules")
sizes = [len(d['z']) for d in mol_dataset]
print(f"Sizes: {min(sizes)}-{max(sizes)} atoms")
print(f"Property range: [{min(d['energy'] for d in mol_dataset):.2f}, "
      f"{max(d['energy'] for d in mol_dataset):.2f}]")
print("\nNote: same GNN handles molecules of DIFFERENT sizes — this is")
print("impossible with a fixed-size MLP!")
```

```
Generated 1420 molecules
Sizes: 3-6 atoms
Property range: [0.70, 7.34]

Note: same GNN handles molecules of DIFFERENT sizes — this is
impossible with a fixed-size MLP!
```

```
# Train on variable-size molecules
torch.manual_seed(42)

n_tr = int(0.8 * len(mol_dataset))
tr_mols = mol_dataset[:n_tr]
te_mols = mol_dataset[n_tr:]

# Normalise
P_mean = np.mean([d['energy'] for d in tr_mols])
P_std = np.std([d['energy'] for d in tr_mols])
```

(continues on next page)

(continued from previous page)

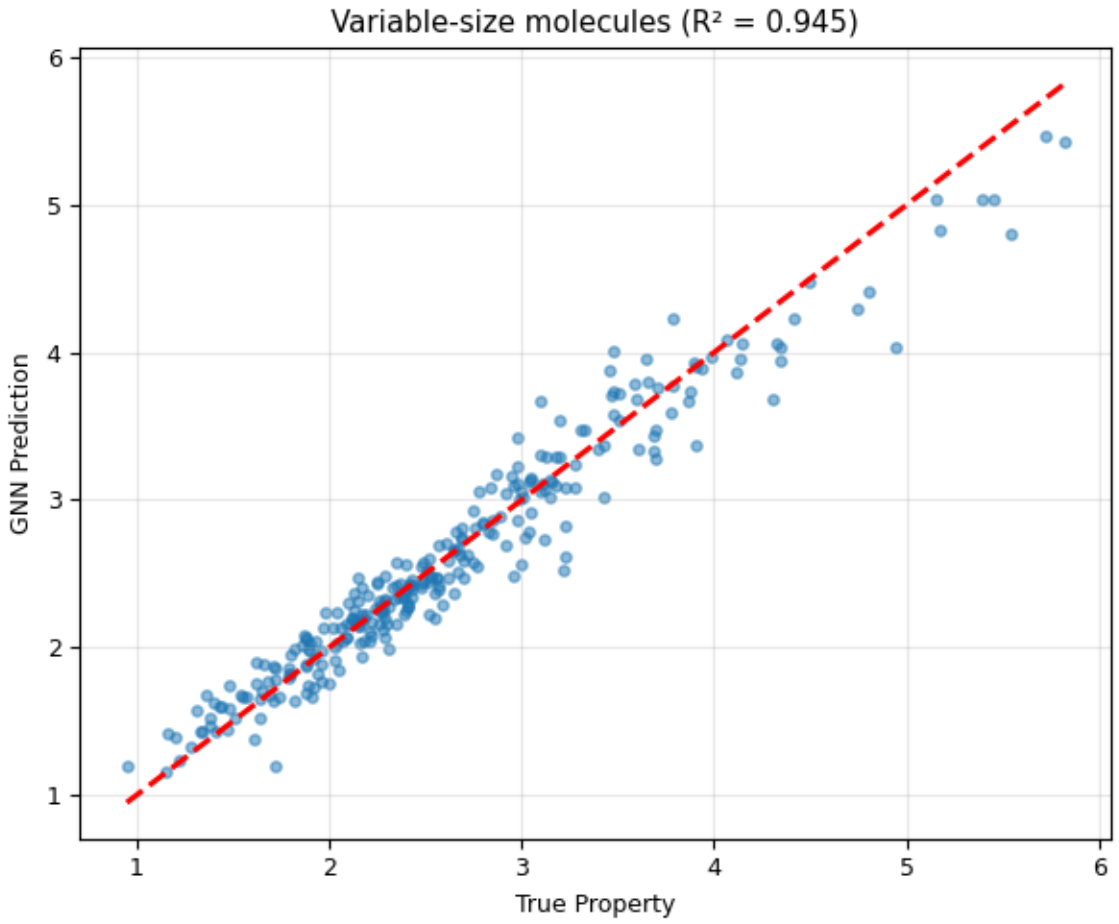
```
for d in mol_dataset:
    d['energy_norm'] = (d['energy'] - P_mean) / P_std

gnn_prop = SimpleGNN(n_elements=4, node_dim=32, edge_dim=16, n_layers=4)
optimizer = torch.optim.Adam(gnn_prop.parameters(), lr=0.002)

batch_size = 32
train_l, val_l = [], []

for epoch in range(150):
    gnn_prop.train()
    np.random.shuffle(tr_mols)
    epoch_loss = 0
    nb = 0
    for i in range(0, len(tr_mols), batch_size):
        batch = collate_graphs(tr_mols[i:i+batch_size])
        optimizer.zero_grad()
        pred = gnn_prop(batch['z'], batch['positions'],
                        batch['edge_index'], batch['batch'])
        true = torch.tensor([d['energy_norm'] for d in tr_mols[i:i+batch_size]],
                            dtype=torch.float32)
        loss = F.mse_loss(pred, true)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        nb += 1
    train_l.append(epoch_loss / nb)

    gnn_prop.eval()
    with torch.no_grad():
        vb = collate_graphs(te_mols)
        vp = gnn_prop(vb['z'], vb['positions'], vb['edge_index'], vb['batch'])
        vt = torch.tensor([d['energy_norm'] for d in te_mols], dtype=torch.float32)
        val_l.append(F.mse_loss(vp, vt).item())

    if epoch % 30 == 0:
        print(f"Epoch {epoch:3d}  Train: {train_l[-1]:.5f}  Val: {val_l[-1]:.5f}")

# Parity plot
gnn_prop.eval()
with torch.no_grad():
    vb = collate_graphs(te_mols)
    pred = gnn_prop(vb['z'], vb['positions'], vb['edge_index'], vb['batch']).numpy()

pred_real = pred * P_std + P_mean
true_real = np.array([d['energy'] for d in te_mols])

fig, ax = plt.subplots(figsize=(6, 5))
ax.scatter(true_real, pred_real, s=15, alpha=0.5)
lims = [min(true_real.min(), pred_real.min()), max(true_real.max(), pred_real.max())]
ax.plot(lims, lims, 'r--', lw=2)
ax.set_xlabel('True Property'); ax.set_ylabel('GNN Prediction')
ax.set_title(f'Variable-size molecules ($R^2$ = {r2_score(true_real, pred_real):.3f})')
ax.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

(continues on next page)

(continued from previous page)

```
print("The same GNN handles 3-atom, 4-atom, 5-atom, and 6-atom molecules!")
```

```
Epoch   0  Train: 0.68437  Val: 0.52358
Epoch  30  Train: 0.07135  Val: 0.07400
Epoch  60  Train: 0.05304  Val: 0.04682
Epoch  90  Train: 0.04971  Val: 0.06432
Epoch 120  Train: 0.03480  Val: 0.07522
```




```
The same GNN handles 3-atom, 4-atom, 5-atom, and 6-atom molecules!
```

23.12 Summary

Concept	Key Idea
Graphs	Atoms = nodes, interactions = edges
Message passing	message → aggregate → update
Radial basis functions	Encode distances as feature vectors
Permutation invariance	Sum aggregation over neighbours
Rotation invariance	Use distances (not coordinates)
Rotation equivariance	Forces via autograd, or equivariant layers
Spherical harmonics	Basis for equivariant features
Variable size	Same model for any number of atoms

23.12.1 The ML for Physics Pipeline

```
Lecture 21: ML foundations    — Fit models, classify, evaluate
Lecture 22: Neural networks   — Learn functions, autograd for forces
Lecture 23: Physics-informed  — Embed PDEs, discover order parameters
Lecture 24: Graph networks    — Encode structure, symmetry, scalability
```

The big picture: ML in physics works best when we **combine data with physical knowledge** — symmetries, conservation laws, and the right representation of the problem.