# Computational Physics I (PHYS 3500K)

**Andreas Papaefstathiou**

**Jul 21, 2025**

# CONTENTS

# Introduction

This book is constructed from the repository dedicated to Computational Physics I (PHYS 3500K) at Kennesaw State University.

Here, you will find examples and notes pertaining to the course, as well as additional material.

Note that the repository is meant to evolve during the semester. Solutions to the homework problems are given to the students taking the course, and they can be provided from the author upon request.

- *Making Computers Obey*
- *Computer Number Representations*
- *Randomness and Random Walks*
- *Numerical Differentiation and Integration*
- *Monte Carlo Methods*
- *Matrix Computing, Trial-and-Error Searching and Data Fitting*
- *Ordinary Differential Equations*
- *An Introduction to Nonlinear Dynamics and Chaos*
- *Boundary Value and Eigenvalue Problems*
- *Partial Differential Equations*
- *More Monte Carlo: The Metropolis Algorithm*

## About the Author

Andreas Papaefstathiou is Assistant Professor of Physics at Kennesaw State University. This website was originally created in January 2024 and is updated on a best-effort basis.

## References


- Computational Physics, Problem Solving with Python - Rubin H. Landau, Manuel J. Páez, Christian C. Bordeianu.
- Computational Physics (Fortran Version) - Steve E. Koonin, Dawn C. Meredith.
- Nonlinear Dynamics and Chaos - Steven H. Strogatz.

CHAPTER

ONE

# MAKING COMPUTERS OBEY

## 1.1 Machine Language

Computers always do exactly as they are told! Instructions they understand are in a machine language.

But when writing and running programs, we communicate to the computer through "shells", in high-level languages (e.g. Python, Java, Fortran, C, …), or through problem-solving environments (e.g. Maple, Mathematica, Matlab, …).

Eventually these commands/programs are translated into the basic machine language that the hardware understands.

## 1.2 Shells, Operating Systems and Compilers

A shell: is a command-line interpreter: a small set of programs run by a computer that respond to the commands that you key in. The job of th eshell is to run programs, compilers and utilities. A demonstration of this will be given during the lectures, but we won't be using shells extensively!

Operating systems, e.g. Unix, DOS, Linux, MacOS, Windows, are a group of programs used by the computer to communicate with users and devices, to store and read data, and to execute programs.

When you submit a program to your computer in a high-level language, the computer uses a compiler to process it. The compiler translates your program into machine language.

Fortran and C (e.g.) read the entire program and then translate it into basic machine instructions. These are known as "compiled languages".

BASIC/Maple translate each line of program as it is entered. These are "interpreted languages".

Python and Java are a mix of both.

## 1.3 Programming Warmup

Here's some "pseudocode" for a program that intends to calculate the area of a circle:

```
read radius # the input
calculate area of circle # the numerics
print area # the output
```

To actually get the computer to do the numerics, we have to specify an **algorithm**. The pseudocode would then look like:

```
read radius # the input
PI = 3.14 # a constant
area = PI * r * r # the algorithm
print area # output
```

## 1.4 Structure and Reproducible Program Design

Programming is a written art that blends the elements of science, math, and computer science into a set of instructions that permit a computer to accomplish a desired task.

It is important that the source code of your program itself is available to others so that they can reproduce and extend your results!

Reproducibility is an essential ingredient in science.

In addition to the grammar of the computer language, a scientific program should include a number of essential elements to ensure the program's validity and useability.

As with other arts, it is recommended that until you know better, you should follow some simple rules:

A good program should:

- give correct answers.
- be clear and easy to read, with the action of each part easy to analyze.
- document itself for the readers/programmer.
- be easy to use.
- be built out of small programs that can be independently verified.
- be easy to modify and robust enough to keep giving correct answers arfter modification and simple debugging.
- document the data fomats used.
- use trusted libraries.
- be published or passed onto others to use and to develop further.

An elementary way to make any program clear is to structure it with indentation, skipped lines, parentheses, all placed strategically.

Python uses indentation as a structure element, as well as for clarity.

## 1.5 Introduction to Python

From the Python tutorial: https://docs.python.org/3.12/tutorial/index.html

> "Python is an easy to learn, powerful programming language.
>
> It has efficient high-level data structures and a simple but effective approach to object-oriented programming.
>
> Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms."

Today we will learn how to use Python in a jupyter notebook (such as this one) and use it to solve some basic problems, including visualization.

### 1.5.1 What is Python?

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes.

### 1.5.2 Aside: Why is it called that?!

When he began implementing Python, Guido van Rossum was also reading the published scripts from "Monty Python's Flying Circus", a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.

### 1.5.3 Jupyter Notebooks, the Gitlab repository and Binder

There are many ways to use Python. We will explore some of them (see Options 1-3), but I will mainly be using jupyter notebooks, such as the one you are currently looking at. The easiest option is to use "binder" (https://mybinder.org/) in conjunction with the course code/notes repository: https://github.com/apapaefs/phys3500k_sp25.

You will find the environment you see here (if you are in class right now!), along with the course files.

### 1.5.4 Let's write some code!

We can already write code in this notebook:

```
print("Hello Whimsical World of Pythonic Physics!")
```

```
Hello Whimsical World of Pythonic Physics!
```

Try it now! Congratulations! You've written your first program in Python!

Let's now define some variables:

```
eggs = 3
text1 = "Break the"
text2 = "eggs" # comments are written this way in Python
```

You can use Python as a calculator:

```
2 + 3
```

```
5
```

```
2 * 42
```

```
84
```

```
8/5
```

```
1.6
```

Note that the above is a float of double precision! (see later)

```
3**2
```

```
9
```

And you can use variables during these operations:

```
eggs * 2
```

```
6
```

It's very easy to manipulate text in Python (represented by "strings"). You can define strings as above, either in "" or ''.

```
text3 = text1 + " " + str(eggs) +  " " + text2
print(text3)
```

```
Break the 3 eggs
```

Strings can be indexed:

```
text3[0]
```

```
'B'
```

```
text3[1]
```

```
'r'
```

Negative integers start counting from the right:

```
text3[-1]
```

```
's'
```

```
text3[-2]
```

```
'g'
```

You can also "slice" strings:

```
text3[0:2]  # characters from position 0 (included) to 2 (excluded)
```

```
'Br'
```

```
text3[2:5]  # characters from position 2 (included) to 5 (excluded)
```

```
'eak'
```

```
text3[4:]   # characters from position 4 (included) to the end
```

```
'k the 3 eggs'
```

len() tells you how long a string is:

```
len(text3)
```

```
16
```

Lists are used to group together values. They are written as a list of comma-separated values (items) between square brackets. They may contain items of different types but usually the items all have the same type.

```
squares = [1, 4, 9, 16, 25]
```

Like strings, lists can be indexed and sliced:

```
squares[3]  # indexing returns the item
```

```
16
```

```
squares[-1] # indexing returns the last item
```

```
25
```

```
squares[1:] # returns the second element up to the end
```

```
[4, 9, 16, 25]
```

Lists also support operations like concatenation:

```
squares + [36, 49, 64, 81, 100]
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
cubes = [1, 8, 27, 65, 125]  # something's wrong here
4 ** 3  # the cube of 4 is 64, not 65!
```

```
64
```

```
cubes[3] = 64  # replace the wrong value
cubes
```

```
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the list.append() method:

```
cubes.append(216)  # add the cube of 6
cubes.append(7 ** 3)  # and the cube of 7
cubes
```

```
[1, 8, 27, 64, 125, 216, 343]
```

len() also applies to lists:

```
len(cubes)
```

```
7
```

```
len(cubes)
```

```
7
```

## 1.6 Control Flow Tools

### 1.6.1 `if` Statements

One of the most well-known statement type is the `if` statement. An example:

```
x = int(input("Please enter an integer: "))
```

```
---------------------------------------------------------------------------
StdinNotImplementedError                  Traceback (most recent call last)
Cell In[27], line 1
----> 1 x = int(input("Please enter an integer: "))

File /usr/local/lib/python3.11/site-packages/ipykernel/kernelbase.py:1281, in␣
↪Kernel.raw_input(self, prompt)
   1279 if not self._allow_stdin:
   1280     msg = "raw_input was called, but this frontend does not support input␣
↪requests."
-> 1281     raise StdinNotImplementedError(msg)
   1282 return self._input_request(
   1283     str(prompt),
   1284     self._parent_ident["shell"],
   1285     self.get_parent("shell"),
   1286     password=False,
   1287 )

StdinNotImplementedError: raw_input was called, but this frontend does not support␣
↪input requests.
```

```
if x < 0:
    x = 0
    print('Negative integer detected, changed to zero!')
elif x == 0:
    print("Zero")
elif x == 1:
    print("One")
else:
    print("Positive integer, not one")
```

```
Positive integer, not one
```

'elif' is short for 'else if', there can be any number of such statements (including none). 'else' is optional and is 'triggered' if all other statements are not satisfied.

### 1.6.2 `for` and `while` Statements

Python's `for` statement iterates over the items of any sequence (e.g. a list or a string), in the order they appear in the sequence. E.g.:

```
# measure the length of all the strings in a list:
words = ['Einstein', 'Galileo', 'Copernicus']
for w in words:
    print(w, len(w))
```

```
Einstein 8
Galileo 7
Copernicus 10
```

If instead you want to iterate over the sequence of numbers, the range() function comes in handy:

```
for i in range(3):
    print(words[i], len(words[i]))
```

```
Einstein 8
Galileo 7
Copernicus 10
```

or, equivalently:

```
for i in range(len(words)):
    print(i, words[i], len(words[i]))
```

```
0 Einstein 8
1 Galileo 7
2 Copernicus 10
```

A `while` statement continues until the given condition stops being true:

```
j = 0
while j < 3:
    print(j, words[j], len(words[j]))
    j = j + 1
```

```
0 Einstein 8
1 Galileo 7
2 Copernicus 10
```

Let's use a `while` loop to write down the first few terms in the Fibonacci series:

```
# Fibonacci series:
# the sum of two elements defines the next
a = 0
b = 1 # define the first two
while a < 10: # do this while the number a is less than 10, stop when it exceeds 10.
    print(a) # print the next number in the series
    c = b
    b = a+c # a becomes the next number to be printed, and calculate the one after␣
 ↪that.
    a = c
```

```
0
1
1
2
3
5
8
```

### 1.6.3 `break` and `continue` Statements, and `else` Clauses on Loops

The break statement breaks out the innermost enclosing for or while loop. E.g.:

```
for w in words:
    if w == 'Galileo':
        continue
    print(w)
```

```
Einstein
Copernicus
```

A `for` or `while` loop can include an `else` clause. In a `for` loop the `else` clause is executed after the loop reaches its final iteration. In a `while` loop, it's executed after the loop's condition becomes false. In either kind of loop, the `else` clause is *not* executed if the loop was terminated by a `break`.

In the following example, an `else` clause is used at the end of a `for` loop which contains an `if` statement that checks whether a number is divisible by all numbers smaller than itself. (The `n % x` part calculates the remainder of the division n/x).

```
for n in range(2,10):
    for x in range(2, n):
        if n%x==0:
            print(n, 'equals', x, '*', n/x)
            break # we have found a number smaller than n is a factor of n, so n is␣
↪not a prime number! -> break the for loop here -> else is not executed!
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2.0
5 is a prime number
6 equals 2 * 3.0
7 is a prime number
8 equals 2 * 4.0
9 equals 3 * 3.0
```

The `continue` statement continues with the next iteration of the loop:

```
for num in range(2, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        continue
    print("Found an odd number", num)
```

```
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

## 1.7 Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
```

The keyword `def` introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented. The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string.

We can execute the function by calling it with a parameter, e.g.:

```
fib(2000)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Functions without `return` statements return `None`:

```
print(fib(2000))
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 None
```

We can instead create a function that *returns* the values we are after, e.g. in a list:

```
def fib2(n):  # return Fibonacci series up to n
    """Return a list containing the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)    # see below
        a, b = b, a+b
    return result
```

And we can call it to create a list, e.g. `fib100`:

```
f100 = fib2(100)
```

```
print(f100)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Note that `append` is a `method` that acts on the list object `result` to add a new element at its end.

For more details on defining functions, see https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions. We will discuss some of those aspects where necessary during the course.

## 1.8 Data Structures

### 1.8.1 List Methods

We already mentionewd the `list.append()` method for a list. The list data type has some more methods, some of which are:

- `list.append(x)`: Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.
- `list.insert(i, x)`: Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `list.remove(x)`: Remove the first item from the list whose value is equal to x. It raises a ValueError if there is no such item.
- `list.pop([i])`: Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)
- `list.clear()`: Remove all items from the list.
- `list.count(x)`: Return the number of times x appears in the list.
- `list.reverse()`: Reverse the elements of the list in place.
- `list.sort(*, key=None, reverse=False)`: Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).
- `list.reverse()`: Reverse the elements of the list in place.
- `list.copy()`: Return a shallow copy of the list. Equivalent to `a[:]`.

The following examples use several of the list methods:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
```

```
fruits.count('apple')
```

```
2
```

```
fruits.count('tangerine')
```

```
0
```

```
fruits.index('banana')
```

```
3
```

```
fruits.index('banana', 4)  # Find next banana starting at position 4
```

```
6
```

```
fruits.reverse()
fruits
```

```
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
```

```
fruits.append('grape')
fruits
```

```
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
```

```
fruits.sort()
fruits
```

```
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
```

```
fruits.pop()
```

```
'pear'
```

### 1.8.2 List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

E.g. let's assume we want to create a list of squares, like:

```
squares = []
for x in range(10):
    squares.append(x**2)
squares
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can instead use a *list comprehension*:

```
squares = [x**2 for x in range(10)]
squares
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more for or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.

E.g. Let's say we want the even numbers in the squares list:

```
even_squares = [y for y in squares if y%2 == 0]
even_squares
```

```
[0, 4, 16, 36, 64]
```

### 1.8.3 Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of sequence data types (see Sequence Types — list, tuple, range). There is also another standard sequence data type: the tuple.

A tuple consists of a number of values separated by commas, for instance:

```
t = 12345, 54321, 'hello!'
t[0]
```

```
12345
```

```
# Tuples may be nested:
u = t, (1, 2, 3, 4, 5)
u
```

```
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are *immutable*, and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later) or indexing. Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

```
# Tuples are immutable:
t[0] = 88888
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[61], line 2
      1 # Tuples are immutable:
----> 2 t[0] = 88888

TypeError: 'tuple' object does not support item assignment
```

### 1.8.4 Dictionaries

Another useful data type built into Python is the dictionary. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

It is best to think of a dictionary as a set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the in keyword.

Here is a small example using a dictionary:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
tel
```

```
{'jack': 4098, 'sape': 4139, 'guido': 4127}
```

```
tel['jack']
```

```
4098
```

```
del tel['sape']
tel['irv'] = 4127
tel
```

```
{'jack': 4098, 'guido': 4127, 'irv': 4127}
```

```
list(tel)
```

```
['jack', 'guido', 'irv']
```

```
sorted(tel)
```

```
['guido', 'irv', 'jack']
```

```
'guido' in tel
```

```
True
```

```
'jack' not in tel
```

```
False
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
```

```
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
{x: x**2 for x in (2, 4, 6)}
```

```
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
dict(sape=4139, guido=4127, jack=4098)
```

```
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

### 1.8.5 Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `items()` method:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
    print(k, v)
```

```
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

```
for i, v in enumerate(['tic', 'tac', 'toe']):
    print(i, v)
```

```
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
    print('What is your {0}?  It is {1}.'.format(q, a))
```

```
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered.

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
    print(i)
```

```
apple
apple
banana
orange
orange
pear
```

Using `set()` on a sequence eliminates duplicate elements. The use of `sorted()` in combination with `set()` over a sequence is an idiomatic way to loop over unique elements of the sequence in sorted order.

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for f in sorted(set(basket)):
    print(f)
```

```
apple
banana
orange
pear
```

## 1.9 Modules

### 1.9.1 User-Defined Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. The same happens when you open a new jupyter notebook. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, we have created a file called `fibo.py` in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
```

(continues on next page)

(continued from previous page)

```
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Let's import this module:

```
import fibo
```

This does not add the names of the functions defined in fibo directly to the current namespace; it only adds the module name fibo there. Using the module name you can access the functions:

```
fibo.fib(1000)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

```
fibo.fib2(100)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

```
fibo.__name__
```

```
'fibo'
```

There is a variant of the import statement that imports names from a module directly into the importing module's namespace. For example:

```
from fibo import fib, fib2
fib(500)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

There is even a variant to import all names that a module defines:

```
from fibo import *
fib(500)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module. This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `fib`.

```
import fibo as fib
fib.fib(500)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

It can also be used when utilising `from` with similar effects:

```
from fibo import fib as fibonacci
fibonacci(500)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

### 1.9.2 Standard Modules and the Standard Library

Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings:

```
import fibo
dir(fibo)
```

```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'fib',
 'fib2']
```

For a longer introduction to the standard library, check out: https://docs.python.org/3/tutorial/stdlib.html. Here, we will go through a few basic modules, and we will introduce more during the course.

The math module gives access to the underlying C library functions for floating point math:

```
import math
math.cos(math.pi / 4)
```

```
0.7071067811865476
```

```
math.log(1024, 2)
```

```
10.0
```

The random module provides tools for making random selections:

```
import random
random.choice(['apple', 'pear', 'banana'])
```

```
'banana'
```

```
random.sample(range(100), 10)   # sampling without replacement
```

```
[91, 73, 7, 46, 38, 20, 93, 64, 85, 32]
```

```
random.random()    # random float
```

```
0.2067182414422717
```

```
random.randrange(6)
```

```
1
```

The statistics module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
import statistics
data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
print(statistics.mean(data))
print(statistics.median(data))
print(statistics.variance(data))
```

```
1.6071428571428572
1.25
1.3720238095238095
```

### 1.9.3 NumPy

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

To access NumPy and its functions import it:

```
import numpy as np
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)␣
↪SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel␣
↪Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)␣
↪instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)␣
↪SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel␣
↪Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)␣
↪instructions.
```

Python lists vs NumPy arrays:

NumPy gives you an enormous range of fast and efficient ways of creating arrays and manipulating numerical data inside them. While a Python list can contain different data types within a single list, all of the elements in a NumPy array should be homogeneous. The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren't homogeneous.

NumPy arrays are faster and more compact than Python lists. An array consumes less memory and is convenient to use. NumPy uses much less memory to store data and it provides a mechanism of specifying the data types. This allows the code to be optimized even further.

One way we can initialize NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

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

The elements can be accessed in the same way as lists, e.g.:

```
print(a[0])
```

```
1
```

You can add the arrays together with the plus sign.

```
data = np.array([1, 2])
ones = np.ones(2, dtype=int)
data + ones
```

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

You can, of course, do more than just addition!

```
data - ones
```

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

```
data * data
```

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

```
data / data
```

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

Basic operations are simple with NumPy. If you want to find the sum of the elements in an array, you'd use `sum()`. This works for 1D arrays, 2D arrays, and arrays in higher dimensions.

```
a = np.array([1, 2, 3, 4])
a.sum()
```

```
10
```

There are times when you might want to carry out an operation between an array and a single number (also called an operation between a vector and a scalar) or between arrays of two different sizes:

```
data = np.array([1.0, 2.0])
data * 1.6
```

```
array([1.6, 3.2])
```

With NumPy you can create and manipulate matrices, generate random numbers, and much more! We will discuss specific applications during the course, but see https://numpy.org/doc/stable/index.html for detailed documentation.

### 1.9.4 SciPy

SciPy provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics and many other classes of problems. It is built on NumPy . It adds significant power to Python by providing the user with high-level commands and classes for manipulating and visualizing data.

Some subpackages of interest to physics are:

- Physical and mathematical constants (`scipy.constants`)
- Special functions (`scipy.special`)

- Integration (`scipy.integrate`)
- Optimization (`scipy.interpolate`)
- Fourier transforms (`scipy.fft`)
- Signal processing (`scipy.signal`)
- Linear Algebra (`scipy.linalg`)
- Spatial data structures and algorithms (`scipy.spatial`)

See https://docs.scipy.org/doc/scipy/tutorial/index.html#subpackages for the full list of subpackages.

Some examples:

```
import scipy
from scipy import constants, special, integrate
```

```
scipy.constants.speed_of_light # get the speed of light
```

```
299792458.0
```

```
# Compute the first ten zeros of integer-order Bessel functions Jn.
scipy.special.jn_zeros(2,10)
```

```
array([ 5.1356223 ,  8.41724414, 11.61984117, 14.79595178, 17.95981949,
       21.11699705, 24.27011231, 27.42057355, 30.5692045 , 33.71651951])
```

```
# Calculate the definite integral of sinx/x in [0,1]
scipy.integrate.quad(lambda x: np.sin(x)/x, 0, 1)
```

```
(0.9460830703671831, 1.0503632079297089e-14)
```

### 1.9.5 Matplotlib (Plotting)

"If I can't picture it, I can't understand it." - Albert Einstein

From the Matplotlib page: https://matplotlib.org:

> Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Matplotlib makes easy things easy and hard things possible.

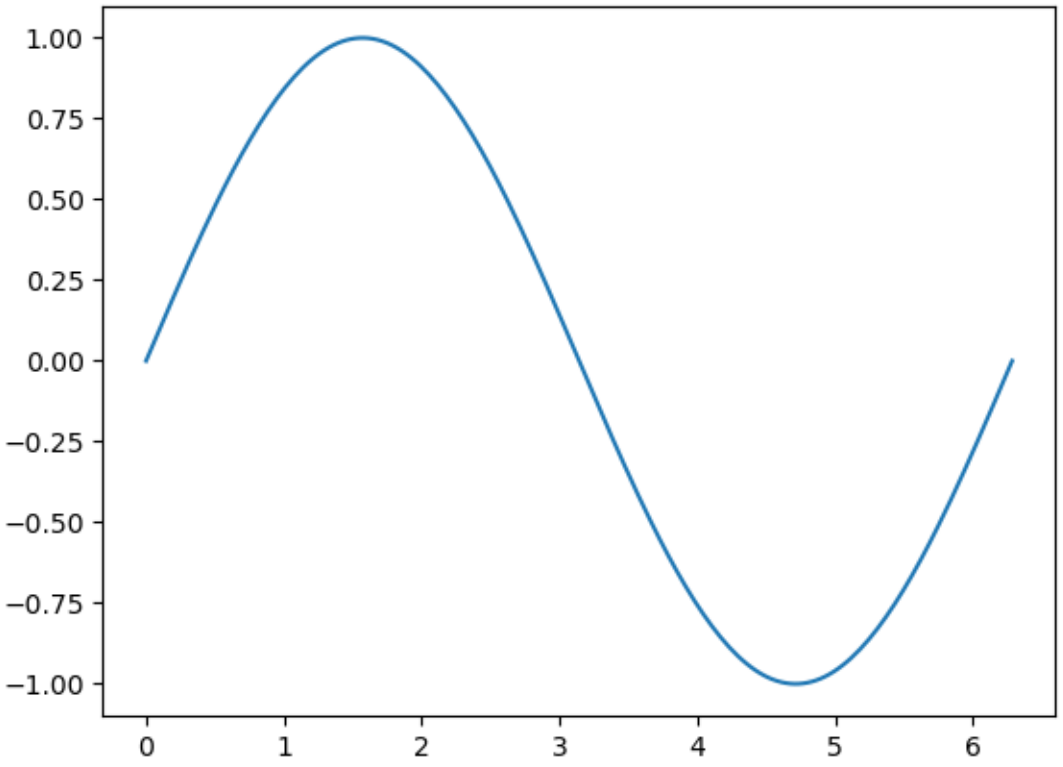
Let's start with a minimal example here (following https://matplotlib.org/stable/users/getting_started/):

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 200) # creates a NumPy array from 0 to 2pi, 200␣
↪equallys-paced points
y = np.sin(x) # take the NumPy array and create another one, where each term is now␣
↪the sine of each of the elements of the above NumPy array

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
↪a figure containing a single set of axes.

ax.plot(x, y) # make a one-dimensional plot using the above arrays
```

(continues on next page)

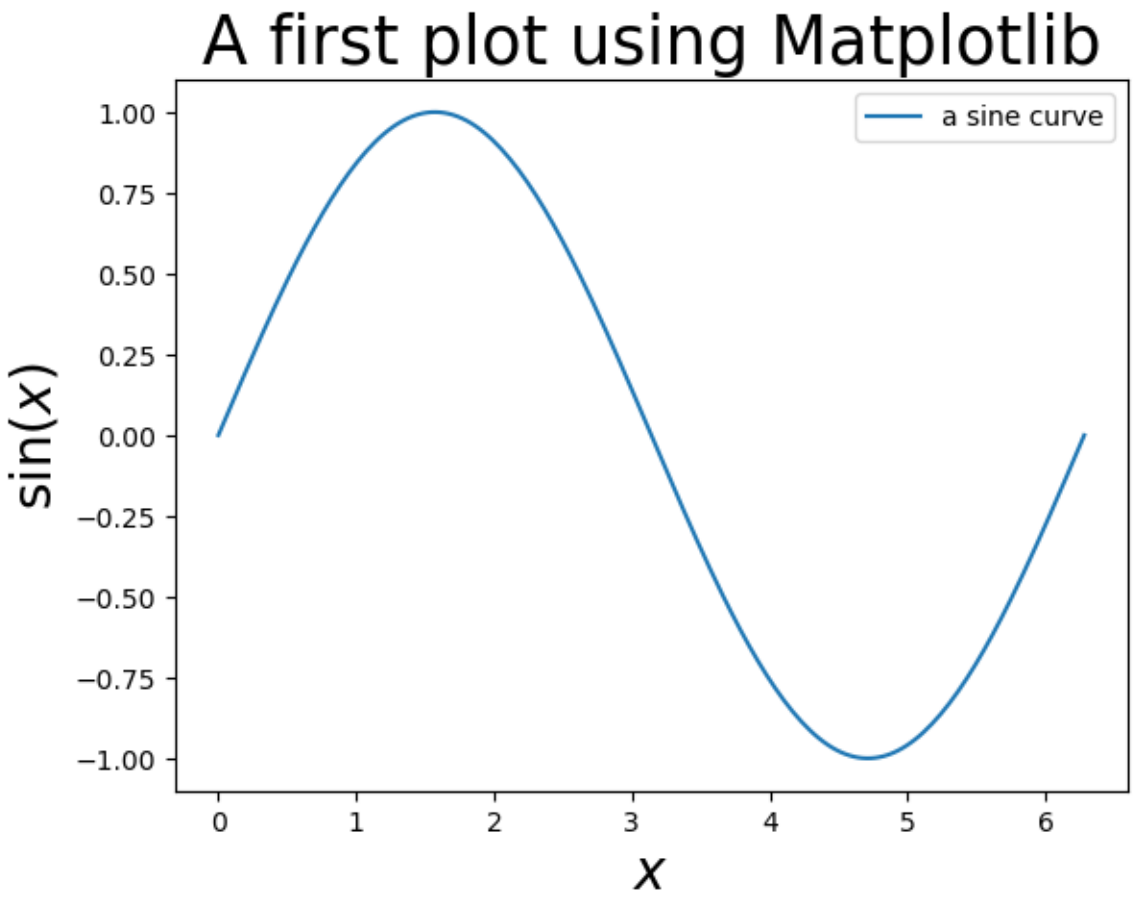
(continued from previous page)

```
plt.show() # show the plot here
```




Let's add a title, axis labels and a legend:

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 200) # creates a NumPy array from 0 to 2pi, 200␣
 ↪equallys-paced points
y = np.sin(x) # take the NumPy array and create another one, where each term is now␣
 ↪the sine of each of the elements of the above NumPy array

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$\sin (x)$', fontsize=20) # set the y label. Note that the 'r' is␣
 ↪necessary to remove the need for double slashes. You can use LaTeX!
ax.set_title('A first plot using Matplotlib', fontsize=25) # set the title

# make a one-dimensional plot using the above arrays, add a custom label
ax.plot(x, y, label='a sine curve')

# construct the legend:
ax.legend(loc='upper right')  # Add a legend

plt.show() # show the plot here
```



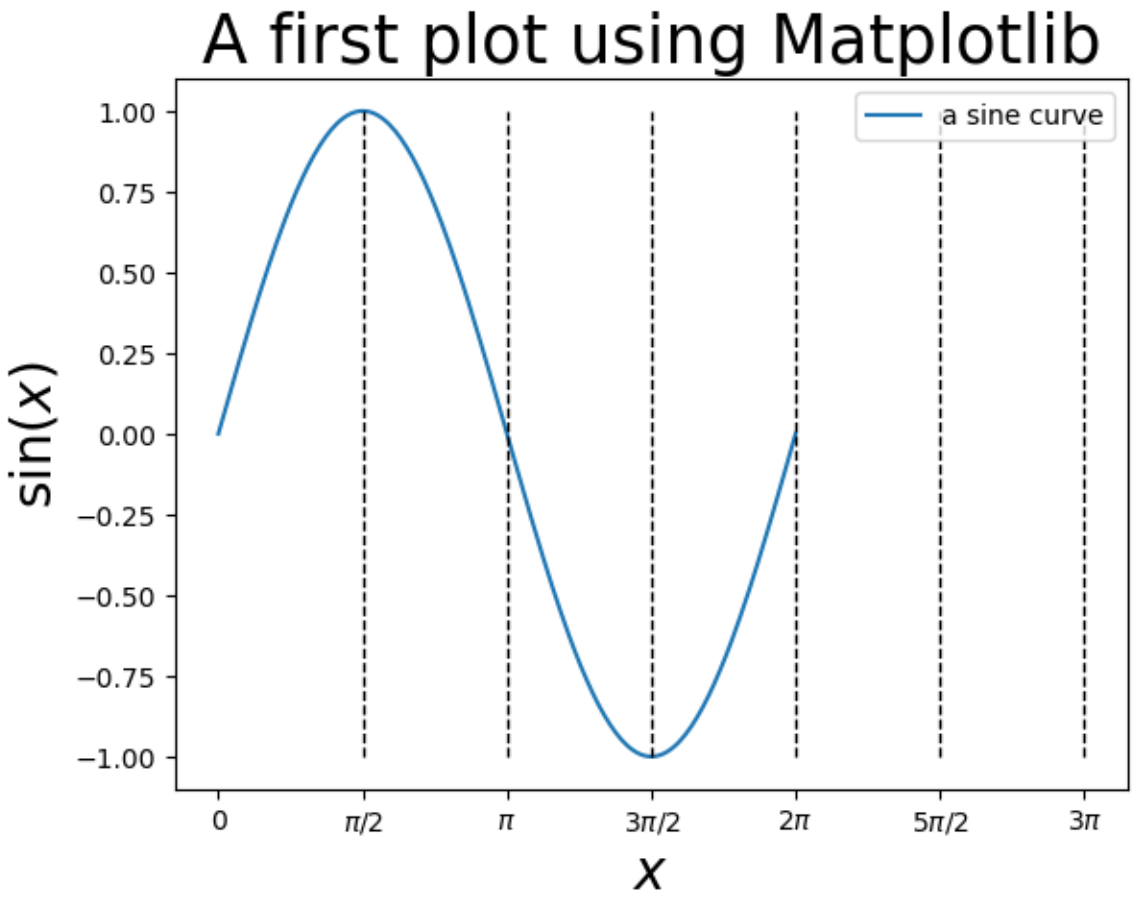

You can also change the labels of the axes to whatever you like, and plot vertical (or horizontal) lines:

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np
from math import pi

x = np.linspace(0, 2 * np.pi, 200) # creates a NumPy array from 0 to 2pi, 200␣
 ↪equallys-paced points
y = np.sin(x) # take the NumPy array and create another one, where each term is now␣
 ↪the sine of each of the elements of the above NumPy array

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$\sin (x)$', fontsize=20) # set the y label. Note that the 'r' is␣
 ↪necessary to remove the need for double slashes. You can use LaTeX!
ax.set_title('A first plot using Matplotlib', fontsize=25) # set the title

# make a one-dimensional plot using the above arrays, add a custom label
ax.plot(x, y, label='a sine curve')

# change the axis labels to correspond to [0, pi/2, pi, 1.5 * pi, 2*pi, 2.5*pi, 3*pi]
ax.set_xticks([0, pi/2, pi, 1.5 * pi, 2*pi, 2.5*pi, 3*pi])
ax.set_xticklabels(['0', '$\\pi/2$', '$\\pi$', '$3\\pi/2$', '$2\\pi$', '$5\\pi/2$', '
```

(continues on next page)

(continued from previous page)

```
↪$3\\pi$'])

# plot vertical lines at pi/2, pi, 3pi/2, 2pi, 5pi/2, 3pi
ax.vlines(x=pi/2, ymin=-1, ymax=1, linewidth=1, ls='--', color='black')
ax.vlines(x=pi, ymin=-1, ymax=1, linewidth=1, ls='--', color='black')
ax.vlines(x=3*pi/2, ymin=-1, ymax=1, linewidth=1, ls='--', color='black')
ax.vlines(x=2*pi, ymin=-1, ymax=1, linewidth=1, ls='--', color='black')
ax.vlines(x=2.5*pi, ymin=-1, ymax=1, linewidth=1, ls='--', color='black')
ax.vlines(x=3.0*pi, ymin=-1, ymax=1, linewidth=1, ls='--', color='black')

# construct the legend:
ax.legend(loc='upper right')  # Add a legend

plt.show() # show the plot here
```



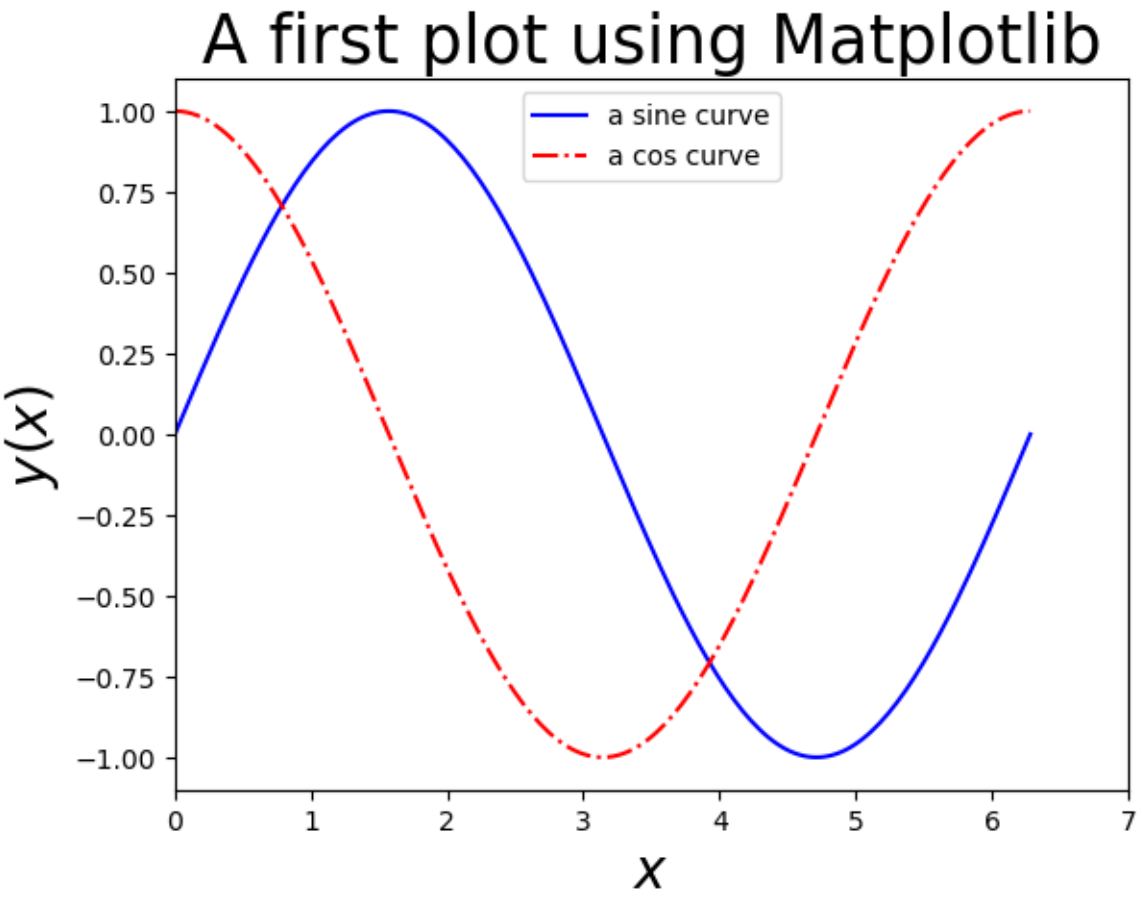

One can plot with different colors and linestyles. See https://matplotlib.org/stable/api/_as_gen/matplotlib.lines.Line2D.html#matplotlib.lines.Line2D.set_linestyle for linestyles and https://matplotlib.org/stable/users/explain/colors/colors.html for colors.

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 200) # creates a NumPy array from 0 to 2pi, 200␣
↪equallys-paced points
```

(continues on next page)

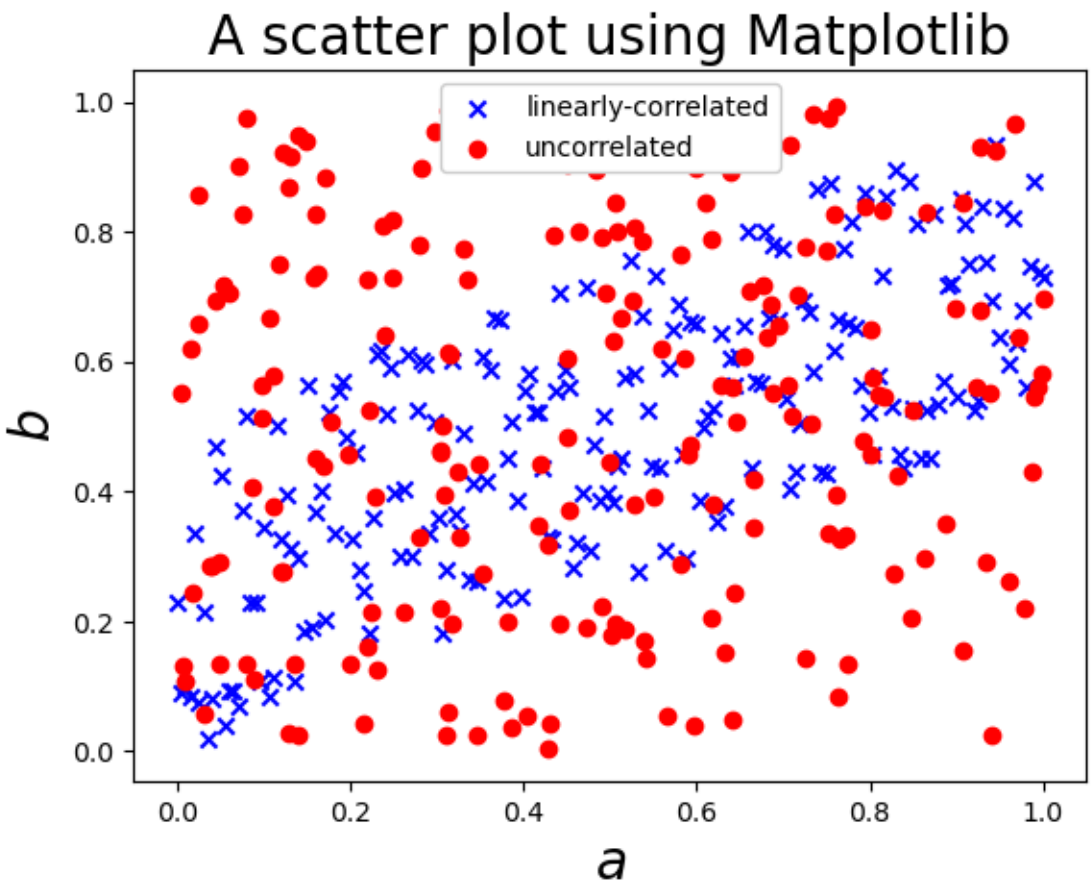
(continued from previous page)

```
y = np.sin(x) # take the NumPy array and create another one, where each term is now␣
 ↪the sine of each of the elements of the above NumPy array
z = np.cos(x) # also get a cosine

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$y(x)$', fontsize=20) # set the y label
ax.set_title('A first plot using Matplotlib', fontsize=25) # set the title

# set the x and y limits:
ax.set_xlim(0, 7)
ax.set_ylim(-1.1,1.1)

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
 ↪and colors:
ax.plot(x, y, color='blue', linestyle='-', label='a sine curve')
ax.plot(x, z, color='red', linestyle='-.', label='a cos curve')

# construct the legend:
ax.legend(loc='upper center')  # Add a legend

plt.show() # show the plot here
```



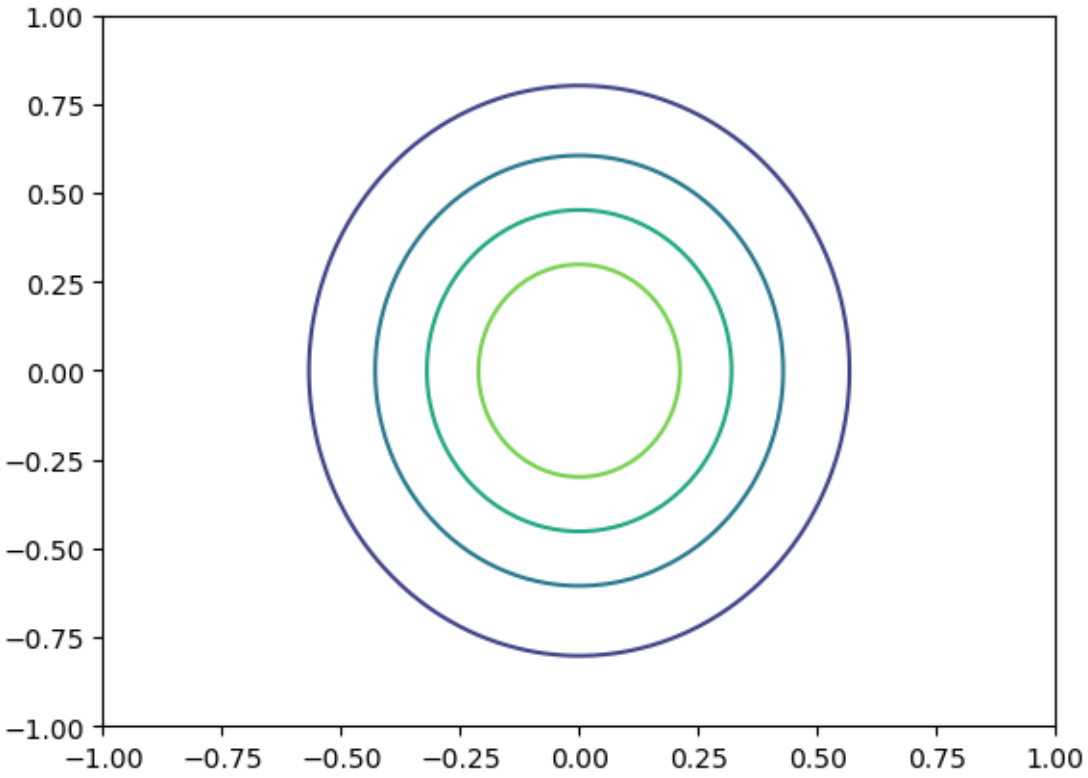
You can also create scatter plots! Here's an example, where we generate a completely uncorrelated set and a slightly correlated set:

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

x = np.linspace(0, 1, 200) # creates a NumPy array from 0 to 2pi, 200 equallys-paced␣
 ↪points
# Now suppose that we have random noise around the curve y = x:
y = 0.5*(x+np.random.random(200)) # generates a NumPy array of size 200 with random␣
 ↪floats in [0,1)

# now generate a completely uncorrelated sample of size 200
z = np.random.random(200)
h = np.random.random(200)

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$a$', fontsize=20) # set the x label
ax.set_ylabel(r'$b$', fontsize=20) # set the y label
ax.set_title('A scatter plot using Matplotlib', fontsize=20) # set the title

# make one-dimensional plots using the above arrays, add a custom label, marker␣
 ↪styles and colors:
ax.scatter(x, y, color='blue', marker='x', label='linearly-correlated')
ax.scatter(z, h, color='red', marker='o',  label='uncorrelated')

# construct the legend:
ax.legend(loc='upper center', framealpha=1.0)  # Add a legend, make it opaque

plt.show() # show the plot here
```



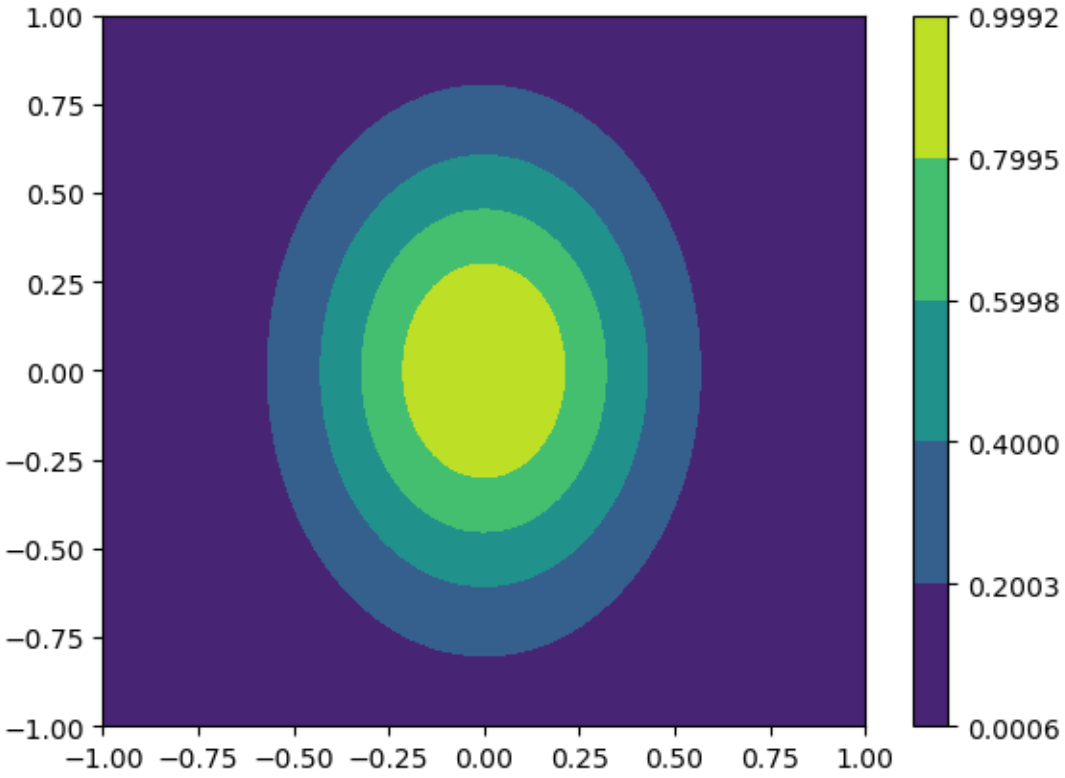

Matplotlib can also generate 2D plots, e.g. contours:

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

# make data: X and Y are defined over a 100x100 grid between (-1,1) in both␣
 ↪dimensions.
X, Y = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))

# now calculate a function over this grid, e.g.:
Z = np.exp(-5*X**2) * np.exp(-2.5*Y**2)
levels = np.linspace(np.min(Z), np.max(Z), 6) # calculate six 'levels' on the contour

# plot
fig, ax = plt.subplots()

# make the contour:
ax.contour(X, Y, Z, levels=levels)

plt.show()
```



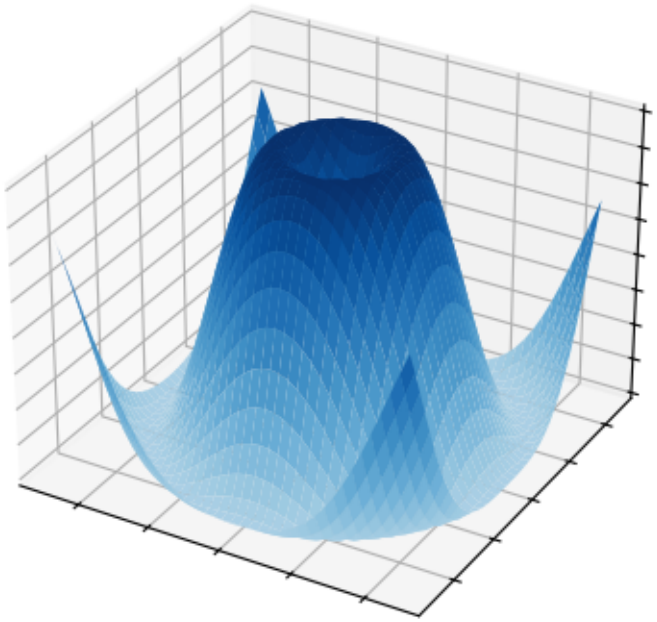

This can also be a "filled" contour, and you can add a color bar to help understand the contour:

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

# make data: X and Y are defined over a 100x100 grid between (-1,1) in both␣
 ↪dimensions.
X, Y = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))

# now calculate a function over this grid, e.g.:
Z = np.exp(-5*X**2) * np.exp(-2.5*Y**2)
levels = np.linspace(np.min(Z), np.max(Z), 6) # calculate six 'levels' on the contour

# plot
fig, ax = plt.subplots()

# make the contour:
cs = ax.contourf(X, Y, Z, levels=levels)

# add a color bar:
cbar = fig.colorbar(cs)

plt.show()
```




You can also plot in three-dimensions:

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

from matplotlib import cm

# Make data
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y) # You need the data to be defined over a grid
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(X, Y, Z, vmin=Z.min() * 2, cmap=cm.Blues)

ax.set(xticklabels=[],
       yticklabels=[],
       zticklabels=[]) # remove tick labels

plt.show()
```



There's tons of functionality in Matplotlib! For examples, check out: https://matplotlib.org/stable/plot_types/index.html and https://matplotlib.org/stable/gallery/index.html.

## 1.10 Other Useful Modules

### 1.10.1 pandas

According to the pandas webpage: https://pandas.pydata.org

> pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

pandas is useful when working with tabular data, such as data stored in spreadsheets or databases. pandas can help you to explore, clean, and process your data. In pandas, a data table is called a `DataFrame`.

We will introduce some pandas functionality during the course.

### 1.10.2 PrettyTable

PrettyTable is "a simple Python library for easily displaying tabular data in a visually appealing ASCII table format". (https://pypi.org/project/prettytable/).

Here's an example:

```
from prettytable import PrettyTable

x = PrettyTable()
x.field_names = ["Particle Name", "Electric Charge", "Spin"]
x.add_row(["Electron", "-1", "1/2"])
```

(continues on next page)

(continued from previous page)

```
x.add_row(["Photon", "0", "1"])
x.add_row(["Higgs Boson", "0", "1"])
x.add_row(["Positron", "+1", "1/2"])
x.add_row(["Graviton", "0", "2"])
print(x)
```

```
+---------------+-----------------+------+
| Particle Name | Electric Charge | Spin |
+---------------+-----------------+------+
|    Electron   |        -1       | 1/2  |
|     Photon    |        0        |  1   |
|  Higgs Boson  |        0        |  1   |
|    Positron   |        +1       | 1/2  |
|    Graviton   |        0        |  2   |
+---------------+-----------------+------+
```

### 1.10.3 tqdm

With `tqdm` you can instantly make your loops show a smart progress meter. e.g.:

```
from tqdm import tqdm
import time
for i in tqdm(range(20)):
    time.sleep(0.1)
```

```
100
 ↪%|████████████████████████████████████████████████████████████████████████████████|␣
 ↪20/20 [00:02<00:00,  9.42it/s]
```

### 1.10.4 SymPy

From the SymPy webpage (https://www.sympy.org/en/index.html):

> SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python.

Here's an example of what SymPy can do:

```
from sympy import *
x, t, z, nu = symbols('x t z nu')

init_printing(use_unicode=True)

# take the derivative of sin(x) * exp(x) wrt. x:
diff(sin(x)*exp(x), x)
```

$$e^{x} \sin\left(x\right) + e^{x} \cos\left(x\right)$$

```
integrate(exp(x)*sin(x) + exp(x)*cos(x), x) # integrate the above to get the function␣
 ↪back!
```

$$e^{x} \sin\left(x\right)$$

```
# calculate the integral of sin(x**2) dx from -infinity to +infinity:
integrate(sin(x**2), (x, -oo, oo))
```

$$\frac{\sqrt{2}\sqrt{\pi}}{2}$$

```
# find the limit of sin(x)/x as x->0:
limit(sin(x)/x, x, 0)
```

$$1$$

```
# solve the equation x**2 - 2 = 0 for x:
solve(x**2 - 2, x)
```

$$\left[-\sqrt{2},\ \sqrt{2}\right]$$

You can also output directly in LaTeX!

```
latex(Integral(cos(x)**2, (x, 0, pi)))
```

```
'\\int\\limits_{0}^{\\pi} \\cos^{2}{\\left(x \\right)}\\, dx'
```

```
init_printing(use_unicode=False)
```

### 1.10.5 Machine Learning with PyTorch

According to Wikipedia (https://en.wikipedia.org/wiki/PyTorch):

> PyTorch is a machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing, originally developed by Meta AI and now part of the Linux Foundation umbrella.

CHAPTER

TWO

# COMPUTER NUMBER REPRESENTATIONS

## 2.1 Introduction to Computer Number Representations

Computers are powerful, but they are *finite*.

A problem in computer design is how to represent an arbitrary number using a finite amount of memory space, and then how to deal with the limitations arising from this representation.

Computer memories are based on magnetic or electronic realizations of a spin pointing up or down. Therefore, the most elementary units of computer memory are two binary integers (bits) 0 and 1.

All numbers are stored in computer memory in the binary form, i.e. as long strings of 0s and 1s.

$N$ bits can store integers in the range $[0, 2^{N-1}]$. $N-1$ since the first bit represents the sign: e.g. 0 for positive numbers.

The description of a particular computer's system or language states the "word length", i.e. the number of bits used to store a number.

The length is often expressed in bytes (i.e. a *mouthful of bits!*), where:

1 byte = 1 B = 8 bits

Memory sizes are measured in bytes, kilobytes, megabytes, gigabytes, terabytes, petabytes.

1 K = 1 kB = $2^{10}$ bytes = 1024 bytes.

1 byte is also the amount of memory needed to store a single letter like "a". This adds up to a typical printed page requiring ~3 kB.

Memory in older personal computers (PCs): uses 8-bit words, and therefore for these systems, the maximum integer was $2^7 = 128$.

With 64-bit (modern PCs): $1 - 2^{63} \simeq 1 - 10^{19}$.

Trying to store a number larger than the hardware/software was designed for (overflow) was common in older machines!

## 2.2 Floating-Point Numbers

Real numbers are represented on computers in either fixed-point or floating-point notations.

In fixed-point notation, numbers are represented with a fixed number of places beyond the decimal point (e.g. integers).

Most scientific computations use double-precision floating-point numbers with 64 bits = 8 B. (Note that in particle physics, and other fields of physics, quadruple precision is necessary for some calculations! There are specialized libraries for this purpose).

64 bit double precision floating-point numbers are called `float` in Python.

In 1987 the Institute of Electrical & Electronics Engineers (IEEE) and the American National Standards Institute (ANSI) adopted the IEEE754 standard for floating point arithmetics.

Bit allocation (IEEE754 standard):

- 1 bit is allocated to the sign indicator, $s$.
- 11 bits are allocated to the exponent, $e$.
- 52 bits are allocated to the fraction, $f$.

A 64-bit (double precision) float can then be represented in decimal (base 10) by the formula:

$n = (-1)^s \times 2^{e-1023} \times (1 + f)$

The value subtracted from the exponent is known as the *bias*, 1023 in this case.

Let's see how this formula can be used to represent a number:

> What is the number 1 10000000010 1000000000000000000000000000000000000000000000000000 (IEEE754) in base 10?
>
> Answer:
>
> The sign of the number is $(-1)^1 = -1$.
>
> The exponent in decimal is $e = 1 \cdot 2^{10} + 1 \cdot 2^1 = 1026$ and therefore the total exponent is $1026 - 1023 = 3$.
>
> The fraction is $f = 1 \cdot \frac{1}{2^1} + 0 \cdot \frac{1}{2^2} + ... = 0.5$.
>
> So:
>
> $n = (-1)^1 \times 2^3 \times (1 + 0.5) = -12.0$ in base 10.

Let's try the opposite operation: given a number in base 10, let's try to find a binary representation:

> What is 15.0 (in base 10) in IEEE754 binary? What is the largest number smaller than 15.0? What is the smallest number larger than 15.0?
>
> The number is positive, so $s = 0$.
>
> We then ask for the exponent $2^{e-1023}$ to give us the last number which is a power of two that is smaller than 15.0. i.e. $2^3 = 8$, and therefore $e = 1026$. In binary form 1026 is: 2^1 + 2^10 or 10000000010.
>
> Then the fraction remains. To obtain this we solve:
>
> $(1+f) = n/2^3$ so: $f = 15/8 - 1 = 0.875$. To get the binary representation try: $0.875 = 1 \cdot \frac{1}{2^1} + 1 \cdot \frac{1}{2^2} + 1 \cdot \frac{1}{2^3}$, so 0.875(base 10) = 1110000000000000000000000000000000000000000000000000 (base 2).
>
> Putting these numbers together gives us the representation of 15.0 in IEEE754:
>
> 0 10000000010 1110000000000000000000000000000000000000000000000000
>
> The next smallest number is: 0 10000000010 1101111111111111111111111111111111111111111111111111, which we can calculate to be:
>
> $n = (-1)^0 \times 2^3 \times (1 \cdot \frac{1}{2^1} + 1 \cdot \frac{1}{2^2} + 1 \cdot \frac{1}{2^4} + 1 \cdot \frac{1}{2^5} + ... + 1 \cdot \frac{1}{2^{52}}) = 14.999999999999999822364316059 97$
>
> and the next largest number is: 0 10000000010 1110000000000000000000000000000000000000000000000001 = 15.000000000000001776356839400 3

From the above, we can conclude that the IEEE754 number representing 15.0 also represents real numbers halfway between its immediate neighbors. Any computation that results in a number within this interval will be assigned the number 15.0

We call the distance from one number to the next the *gap*. Since the fraction is multiplied by $2^{e-1023}$, the gap grows as the number represented grows. The gap at any given number can be computed using numpy:

```
import numpy as np

np.spacing(1e9)
```

```
np.float64(1.1920928955078125e-07)
```

There are special cases:

- When $e = 0$: the leading 1 in the fraction takes the value 0 instead. The result is called a *subnormal* number and is computed by $n = (-1)^s 2^{-1023}(0 + f)$.
- When $e = 2047$ and $f$ is non zero, then the result is "not a number", `NAN`, i.e. undefined.
- When $e = 2047$, $f = 0$, $s = 0$ the result is positive infinity, `+INF`.
- When $e = 2047$, $f = 0$, $s = 1$ the result is negative infinity, `-INF`.

Overflow occurs when numbers exceed the maximum value that can be represented. Python will assign this number to `INF`.

Underflow occurs when small nonzero values become too small, smaller than the smallest subnormal number. Python will assign a zero.

You can check these values using the sys module:

```
import sys
sys.float_info

print(sys.float_info.max)
print(sys.float_info.min)
```

```
1.7976931348623157e+308
2.2250738585072014e-308
```

## 2.3 Errors and Uncertainties in Computations

### 2.3.1 Types of Errors

Four general types of errors exist to plague your computations:

1. **Blunders or bad theory**: typos in program or data, fault in reasoning (i.e. in the theoretical description), using the wrong data files, etc..
2. **Random errors**: imprecision caused by events such as fluctuations in electronics, cosmic rays, etc..
3. **Approximation errors**: Imprecision arising from simplifying the math so that a problem can be solved on a computer. e.g. replacement of infinite series by a finite sums, of infinitesimal intervals by finite ones, by variable functions by constants, e.g.:

$\sin(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!}$ (exact)

vs.

$\sin(x) \simeq \sum_{n=1}^{N} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} + \epsilon(x, N)$

$\epsilon(x, N)$ is the approximation (or algorithmic) error = the series from $N + 1$ to $\infty$.

For any reasonable approximation, this error should decrease as $N$ increases and should vanish in the limit $N \to \infty$.

4. **Round-off errors**: Imprecision arising arising from the finite number of digits used to store floating-point numbers. This is analogous to the uncertainty in the measurement of a physical quantity in an elementary physics lab. Round-off errors accumulate as the computer handles more numbers, i.e. as the number of steps in a computation increases. This may cause some algorithms to become unstable with a rapid increase in error. In some cases the round-off error may become a major component in your answer, leading to "garbage".

e.g. if a computer stores 1/3 as 0.3333 and 2/3 as 0.6667, if we ask the computer to do 2*(1/3) - 2/3 = 0.6666 - 0.6667 = -0.0001. Although small, this error is non-zero. If this is repeated millions of times, the final answer might not even be small!

### 2.3.2 Subtractive Cancelation

Calculations employing numbers that are stored only approximately on the computer can only be expected to yield approximate answers.

We model the computer representation, $x_c$, of the exact number $x$, as:

$x_c \simeq x(1+\epsilon_x)$,

where $\epsilon_x$ is the relative error in $x_c$. We expect this to be of the same order as the "machine precision". In particular, we expect double-precision numbers to have an error in the 15th decimal place.

We can apply this notation to simple subtraction: $a = b - c$.

On a computer, this would be: $a_c \simeq b_c - c_c \simeq b(1+\epsilon_b) - c(1+\epsilon_c)$.

So:

$\frac{a_c}{a} \simeq 1 + \epsilon_b \frac{b}{a} - \frac{c}{a}\epsilon_c$

Observe that: when we subtract two nearly equal numbers, i.e.,

$b \simeq c \gg a$

Then:

$\frac{a_c}{a} \equiv 1 + \epsilon_a \simeq 1 + \frac{b}{a}(\epsilon_b - \epsilon_c) \simeq 1 + \frac{b}{a}(|\epsilon_b| + |\epsilon_c|)$

Even if the relative errors somewhat cancel, they are still multiplied by a large number, $b/a$. This can significantly magnify the error. But since we cannot assume a sign for the errors, we can assume the worst, i.e. that they add up.

As a more explicit mathematical example, consider the constructing the expansion of $e^{-x}$ for **large values of** $x$:

$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + ...$

The first few terms are large, but of alternating sign, leading to an almost total cancelation in order to yield the final small result.

This subtractive cancelation can be eliminated by considering $e^{-x} = 1/e^x$ instead, but round-off errors still remain, which we examine next.

### 2.3.3 Round-off Errors

Error arises from a single division of the computer representations of two numbers:

$a = \frac{b}{c} \rightarrow a_c = \frac{b_c}{c_c} = \frac{b}{c}\left(\frac{1+\epsilon_b}{1+\epsilon_c}\right)$,

so:

$\frac{a_c}{a} = \frac{1+\epsilon_b}{1+\epsilon_c} \simeq (1+\epsilon_b)(1-\epsilon_c) \simeq 1 + \epsilon_b - \epsilon_c$,

where we have ignored terms of order $\epsilon^2$.

We need to assume the worst for the error subtraction, and get:

$\frac{a_c}{a} \simeq 1 + |\epsilon_b| + |\epsilon_c|$.

Can generalize this to estimate the error in the evaluation of a general function $f(x)$:

$\epsilon = \frac{f(x)-f(x_c)}{f(x)} \simeq \frac{\mathrm{d}f(x)/\mathrm{d}x}{f(x)}(x - x_c)$.

E.g. for $f(x) = \sqrt{1+x}$: $\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{1}{2}\frac{1}{\sqrt{1+x}}$ .

$\epsilon = \frac{x-x_c}{2(1+x)}$ .

e.g. Evaluate at $x = \pi/4$ and assume an error in the 4th decimal place of $x$, then we obtain a similar error of $1.5 \times 10^{-4}$ on $f(x) = \sqrt{1+x}$.

## 2.4 Round-off Error Accumulation

If a calculation has a large number of steps, we can view the error as a step in a random walk.

For a random walk (see later), the total distance $R$, covered in $N$ steps of length $r$ is:

$R = \sqrt{N}r$

By analogy, the total relative error after $N$ calculational steps, each with machine precision, $\epsilon_m$, is on average:

$\epsilon_{\text{ro}} = \sqrt{N}\epsilon_m$

If the round-off error does not accumulate in a random manner: a more detailed analysis is needed.

In some cases there may be no cancelation, and the error may increase as $N\epsilon_m$ instead. Even worse, in some recursive algorithms, error generation can be coherent, leading to a $N!$ increase in error.

CHAPTER

# THREE

# RANDOMNESS AND RANDOM WALKS

## 3.1 Deterministic Randomness

Some people are attracted to computing because of its deterministic nature. Nevertheless, many computer cycles are used for **Monte Carlo calculations**.

Monte Carlo calculations include elements of chance at their very core (hence the name). They involve *random-like* numbers generated by a computer to simulate naturally-random processes, such as thermal motion or radioactive decay.

Monte Carlo techniques are able to solve very challenging problems!

## 3.2 Random Sequences

We define a sequence $r_1, r_2, ...$ as random if there are no correlations between the coefficients.

But: being random does not mean all numbers in the sequence are equally likely to occur!

If all numbers in a sequence are equally likely to occur: the sequence is called **uniform**.

Mathematically: the likelihood of a number occurring is described by a distribution $P(r)$, where $P(r)\mathrm{d}r$ is the probability of finding $r$ in the interval $[r, r + \mathrm{d}r]$.

For a **uniform distribution**: $P(r) =$ constant.

The standard random number generator on computers generates *uniform* distributions between 0 and 1, i.e. it outputs numbers in this interval with equal probability, yet independent from the previous number.

By their nature: computers are deterministic devices, and therefore cannot create a random sequence. Computed random number sequences must contain correlations, and therefore cannot be truly random. This implies that if we know $r_m$ and its preceding elements, then it is always possible to figure out $r_{m+1}$. For this reason, computers are said to generate **pseudorandom numbers**.

A primitive alternative to this is to read off a table of truly random numbers, determined by naturally-random process, such as radioactive decay, or to connect the computer to an experimental device that measures random events.

### 3.2.1 Random Number Generation

Sequences of pseudorandom numbers can be generated on a computer via the "linear congruent" or "power residue" method:

To generate a pseudorandom sequence of numbers $0 \leq r_i \leq M-1$ over the interval $[0, M-1]$, starting with $r_1$, to obtain the next random number $r_2$, multiply $r_1$ by a constant $a$, add another constant $c$ and keep the *remainder*. Repeat this with $r_2$ to get $r_3$ and so on. So to get $r_{i+1}$, given $r_i$:

$r_{i+1} = \text{remainder}\left(\frac{ar_i+c}{M}\right)$

The initial number, the *seed*, $r_1$ is frequently supplied by the user.

In Python the remainder is given by the operator "%", i.e. the above equation would read: `rnext = (a * rprevious + c) %M`.

One can then divide the whole sequence by $M$ to get numbers in the interval $[0, 1]$.

If we want random numbers in an interval $[A, B]$ instead, then we can transform the generated sequence in the interval $[0, 1]$, by:

$x_i = A + (B - A)r_i$

Note that the linear congruent method becomes completely correlated if a particular interger comes up a second time. The whole cycle then repeats.

If e.g. 48-bit arithmetic is used, then $M \sim 2^{48} \simeq 3 \times 10^{14}$. If your program would then use approximately these many random numbers (feasible, e.g. if your program is performing Monte Carlo simulations in many dimensions!), then you may need to "re-seed" (i.e. choose a new seed) to reset the sequence during intermediate steps.

In Python, the default `random.random()` algorithm is the "Mersernne Twister". It has an extremely long period: $2^{19937} - 1$ (a prime number) which is approximately $4.3 \times 10^{6001}$. This is many orders of magnitude larger than the estimated number of particles in the observable universe ($\sim 10^{87}$).

See Wikipedia entry for more details, https://en.wikipedia.org/wiki/Mersenne_Twister, as well as original paper: https://dl.acm.org/doi/pdf/10.1145/272991.272995.

### 3.2.2 Generating Random Numbers of Arbitrary Distributions

Supose we possess a random number generator that yields uniform random numbers in the interval $[0, 1]$. We can use it to generate random numbers according to any distribution using the *inversion method*.

The method proceeds as follows:

1. Suppose we want to distribute random numbers according to a function $f(x)$. We first calculate the *cumulative* distribution function (CDF) as: $F(x) = \int_0^x \mathrm{d}x f(x)$.
2. We then find the inverse of the desired CDF, i.e. we solve $y = F(x)$ as $F^{-1}(y) = x$.
3. If then "draw" $y_0$ and calculate $x_0 = F^{-1}(y_0)$, then $x_0$ is distributed according to $f(x)$.

Here's an example for $f(x) = 3x^2$ for $x \in [0, 1]$. Note that $\int_0^1 \mathrm{d}x f(x) = 1$ for $f(x)$ in $[0, 1]$, as expected for a probability distribution.

The CDF is $F(x) = x^3$. Then $x = y^{1/3}$ and so we have:

```
import random
import math
import matplotlib.pyplot as plt # import matplotlib
import numpy as np
```

(continues on next page)

(continued from previous page)

```
x = []

# We have x = (y)**(1/3) and we pick y in [0,1]
for i in range(10000): # generate 1000 points
    y = random.random() # generate random numbers in [0,1]
    x.append( (y)**(1/3) )


# create the real distribution for comparison. Use numpy"
xcomp = np.linspace(0,1,100)
ycomp = 3*np.square(xcomp)

# create a histogram of x:
fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$P(x)$', fontsize=20) # set the y label. Note that the 'r' is␣
 ↪necessary to remove the need for double slashes. You can use LaTeX!
ax.set_title('Histogram', fontsize=20) # set the title

# make a histogram of the values, density=True normalizes the histogram
ax.hist(x, label='generated values via inversion method', density=True, bins=100)

# compare to how we expect it to look like:
ax.plot(xcomp, ycomp, label='expected shape of $f(x) = 3x^2$')

# construct the legend:
ax.legend(loc='upper center')  # Add a legend

plt.show() # show the plot here
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 3
      1 import random
      2 import math
----> 3 import matplotlib.pyplot as plt # import matplotlib
      4 import numpy as np
      6 x = []

ModuleNotFoundError: No module named 'matplotlib'
```

This method predicates on the existence of an analytical expression for the CDF and its inverse, and therefore is not always useful! (see later)

## 3.3 Random Walks

Consider a pizza brought into this room. The molecules from its smell collide randomly with other molecules in the air and eventually reach your nose.

The problem is to determine how many collisions, on average, a scent molecule makes in traveling a distance $R$, given that it travels an average (root-mean-square) distance $r_{\text{rms}}$ between collisions.

### 3.3.1 Random Walks: Theoretical Description

A "walker" takes sequential steps with the direction of each step independent of the direction of the previous step.

Let's consider a two-dimensional model. Start at the origin and take $N$ steps in the $xy$-plane of lengths:

$(\Delta x_1, \Delta y_1), (\Delta x_2, \Delta y_2), ..., (\Delta x_N, \Delta y_N)$

then, we have:

$R^2 = (\Delta x_1 + \Delta x_2 + ... + \Delta x_N)^2 + (\Delta y_1 + \Delta y_2 + ... + \Delta y_N)^2.$

$\Rightarrow R^2 = \Delta x_1^2 + \Delta x_2^2 + \Delta x_N^2 + 2\Delta x_1 \Delta x_2 + 2\Delta x_1 \Delta x_3 + ... + (x \to y).$

If a walk is random, then the walker is equally likely to travel in any direction at each step. If we take the average of $R^2$ over a large number of such random steops, we expect that all cross terms would vanish:

$\langle R^2 \rangle = \langle (\Delta x_1^2 + \Delta y_1^2) + (\Delta x_2^2 + \Delta y_1^2) + .. + (\Delta x_N^2 + \Delta y_N^2) \rangle = N \langle r^2 \rangle = N r_{\text{rms}}^2$

where $r_{\text{rms}} = \sqrt{\langle r^2 \rangle}$.

So:

$R_{\text{rms}} = \sqrt{\langle R^2 \rangle} = \sqrt{N} r_{\text{rms}}.$

Let's investigate this theoretical description through a simulation!

Before we do so, let's introduce the concept of classes in Python (object-oriented programming), and use it in our example.

## 3.4 Digression: Object-Oriented Programming in Python

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made.

The simplest form of a class definition looks like:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

You can read all about classes and object-oriented programming at: https://docs.python.org/3/tutorial/classes.html. Here we are going to proceed by example.

Suppose we wish to create a new object that has the properties we expect from complex numbers. We want this object to have a real and imaginary part, and we want to easily access those, as well as perform various operations on it: e.g. get its modulus, complex conjugate or argument.

We can begin by defining a class as follows:

```
# define a new class called Complex with real and imaginary parts (.r and .i):
class Complex:
    """A simple complex number class"""
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
        print("I have created a new complex variable with r,i=", self.r, self.i)
```

The `__init__()` method is automatically invoked when you create an instance of this class:

```
# Create an "instance" of the Complex class:
x = Complex(3.0, 4.5) # the real part is 3.0 and the imaginary part is 4.5
print('real part=', x.r)
print('imaginary part=', x.i)
y = Complex(1.0, -1.5)
print(y.r, y.i)
```

```
I have created a new complex variable with r,i= 3.0 4.5
real part= 3.0
imaginary part= 4.5
I have created a new complex variable with r,i= 1.0 -1.5
1.0 -1.5
```

We can also add "attribute references", i.e. functions that operate on instances of this function:

```
import math

# define the Complex class, with mod(), arg() and cc() functions to return:
# modulus, argument and complex conjugate
# see https://en.wikipedia.org/wiki/Complex_number for details
class Complex:
    """A more advanced complex number class"""
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

    # the modulus:
    def mod(self):
        return math.sqrt(self.r**2 + self.i**2)

    # the argument
    def arg(self):
        if self.i!=0 and self.r>0:
            arg = 2 * math.atan(self.i/(self.r+self.mod()))
        elif self.r<0 and self.i==0:
            arg = math.pi
        elif self.r==0 and self.i==0:
            arg = float('nan')
        return arg

    # the complex conjugate, return a Complex object with - the imaginary part
    def cc(self):
        return Complex(self.r, -self.i)
```

```
# Create an "instance" of the updated Complex class:
x =Complex(3.0, 4.5) # the real part is 3.0 and the imaginary part is 4.5
```

(continues on next page)

(continued from previous page)

```
print('real part=', x.r)
print('imaginary part=', x.i)
```

```
real part= 3.0
imaginary part= 4.5
```

```
# get the modulus:
x.mod()
```

```
5.408326913195984
```

```
# get the argument:
print(x.arg())
```

```
0.9827937232473289
```

```
# get the complex conjugate, another instance of the Complex class:
y = x.cc()
print(y.r)
print(y.i)
```

```
3.0
-4.5
```

You can also define functions that act on two instantiations of the class. E.g. let's say we want to be able to multiply two complex numbers together correctly.

```
import math

# define the Complex class, with mod(), arg() and cc() functions to return:
# modulus, argument and complex conjugate
# see https://en.wikipedia.org/wiki/Complex_number for details
class Complex:
    """A more advanced complex number class"""
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

    # the modulus:
    def mod(self):
        return math.sqrt(self.r**2 + self.i**2)

    # the argument
    def arg(self):
        if self.i!=0 and self.r>0:
            arg = 2 * math.atan(self.i/(self.r+self.mod()))
        elif self.r<0 and self.i==0:
            arg = math.pi
        elif self.r==0 and self.i==0:
            arg = float('nan')
        return arg

    # the complex conjugate, return a Complex object with - the imaginary part
    def cc(self):
```

(continues on next page)

(continued from previous page)

```
        return Complex(self.r, -self.i)

    # multiply two complex numbers together:
    def multiply(self, other):
        return Complex(self.r * other.r - self.i * other.i, self.r * other.i + self.i␣
↪* other.r)
```

Let's try it out!

```
# define two complex numbers:
x = Complex(3.0, 4.5)
z = Complex(1.0, 2.0)
# multiply them to get another complex number
h = x.multiply(z)
print(h.r)
print(h.i)
```

```
-6.0
10.5
```

(Check the answer!)

Or we can multiply x by its complex conjugate to get a real number:

```
# we can get the c.c. directly:
u = x.multiply(x.cc())
# print the real part and the imaginary parts: the latter should be zero.
print(u.r)
print(u.i)
# the square root of the product x * x complex conjugate should the the modulus!
print(x.mod())
print(math.sqrt(u.r))
```

```
29.25
0.0
5.408326913195984
5.408326913195984
```

## 3.5 Random-Walk Simulation

We can now proceed with a simulation of random walks. We will define a class called "walker", which will allow us to define multiple walkers and make them walk around.

```
import random
import math

class Walker:
    """A random walker class"""
    # instantiations of this class are initialized with an initial position
    def __init__(self, initialx, initialy):
        # initialize the x and y positions:
        self.x = initialx
        self.y = initialy
```

(continues on next page)

(continued from previous page)

```
        # keep the initial positions:
        self.xinit = initialx
        self.yinit = initialy
        # keep all the positions that the walker has moved through in a list:
        self.allx = []
        self.ally = []
        self.allx.append(initialx)
        self.ally.append(initialy)

    # now let's create a function that allows us to take random steps with a certain␣
↪stepsize
    def move(self, stepsize):
        deltax = (2*random.random()-1) # choose random variables in the range [-1,1]
        deltay = (2*random.random()-1)
        # normalize to stepsize:
        L = math.sqrt(deltax**2 + deltay**2)
        deltax = deltax/L
        deltay = deltay/L
        # add these to the current position
        self.x = self.x + deltax
        self.y = self.y + deltay
        self.allx.append(self.x)
        self.ally.append(self.y)

    # get the distance from the starting position
    def distance(self):
        return math.sqrt( (self.x-self.xinit)**2 + (self.y-self.yinit)**2 )
```

Let's test this out with a single walker starting from (0,0):

```
# initialize "Walker1", an instance of the class Walker
Walker1 = Walker(0,0)

# move Walker1 1000 times with stepsize 1:
for i in range(1000):
    Walker1.move(1.0)

# get the distance from the origin:
print(Walker1.distance())
```

```
12.323614832708454
```

Our next task is to initialize N (=9 to start with) walkers at the origin and follow their path visually! We are going to use matplotlib and evolve walkers "dynamically" on a graph.

It also will be interesting to investigate whether our theoretical expectations agree with this simulation.

Let's also calculate, and store, the root-mean-square (rms) distance versus the of number of steps.

```
import numpy as np
import matplotlib.pyplot as plt
import time # various time functions
from tqdm import tqdm # progress bar
import matplotlib.ticker as ticker #

# this allows us to access dynamic updating of the plot:
from IPython import display
```

(continues on next page)

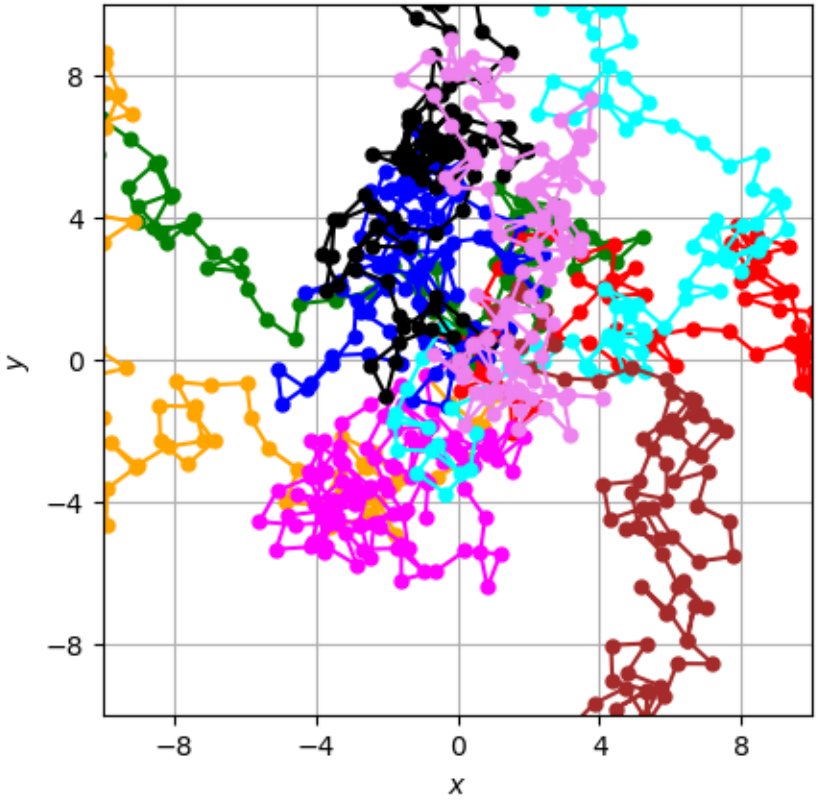
(continued from previous page)

```
from IPython.display import clear_output

# setup the axis:
fig,ax = plt.subplots(1,1)

#get the dynamic display:
dynamicdisplay = display.display("", display_id=True)

# put a grid on the graph:
ax.grid(True, which='both')

# set the aspect ratio to 1:
ax.set_aspect(1)

# set some parameters here:
n = 9 # the number of walkers
N = 100 # the number of steps
step = 1 # the step size
Nupdate = 1 # the frequency of steps used to update the plot

# set the labels and limits:
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
limitsize = math.sqrt(N)*step
ax.set_xlim(-limitsize,limitsize)
ax.set_ylim(-limitsize,limitsize)

# set the major ticks to correspond to twice step size:
ax.xaxis.set_major_locator(ticker.MultipleLocator(4*step))
ax.yaxis.set_major_locator(ticker.MultipleLocator(4*step))

# initialize n walkers and put them in a list:
WalkerList = []
WalkerColor = [] # create a list with the colors of each walker
colors = [ 'green', 'orange', 'red', 'magenta', 'blue', 'cyan', 'black', 'brown',
 ↪'violet'] # 9 colours
j = 0 # counter to make sure we don't go over the 9 colors
for w in range(n):
    WalkerList.append(Walker(0,0))
    # set the color of each walker. If we w exceeds 8 then reset the counter j:
    if w-8*j > 8:
        j = j + 1
    else:
        pass
    WalkerColor.append(colors[w-8*j])

# now go through all the n walkers and get them to perform N steps
for i in tqdm(range(N)):
    # go through all the walkers in the WalkerList
    for j, walker in enumerate(WalkerList):
        # move them one step in a random direction
        walker.move(step)

    if i%Nupdate==0: # only update every Nupdate steps
        # plot them!
        # the list will contain the points so we can remove them later:
        walkers_plot = []
```

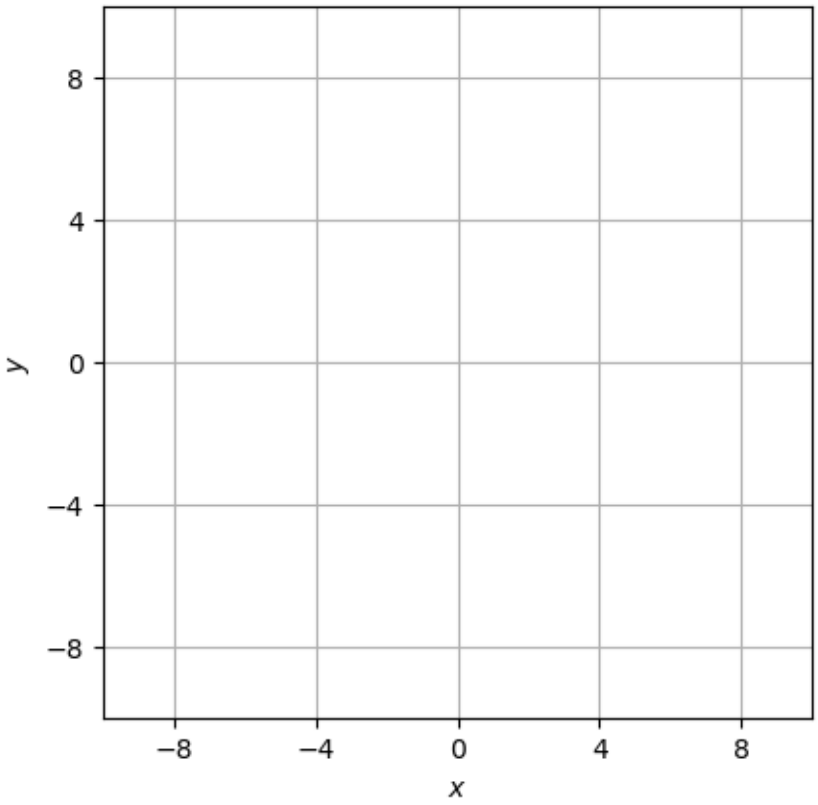
(continues on next page)

(continued from previous page)

```
        for j, walker in enumerate(WalkerList):
            walkers_plot.append(ax.plot(walker.allx, walker.ally, marker='o',
↪color=WalkerColor[j], ls='-', markersize=5))
        dynamicdisplay.update(fig) # update the figure in the notebook
        # remove the walkers from display for next run:
        for j, walker in enumerate(WalkerList):
            wmarker = walkers_plot[j].pop(0)
            wmarker.remove()
        time.sleep(0.1) # "sleep" for half a second to allow us to see the evolution
```




```
100%|██████████| 100/100 [00:15<00:00,  6.33it/s]
```




Let's plot the RMS distance versus the square root of N! First, let's increase the number of walkers and the number of steps, without plotting:

```
import numpy as np
from tqdm import tqdm # progress bar

# set some parameters here:
n = 1000 # the number of walkers
N = 10000 # the number of steps
step = 1 # the step size
Nupdate = 1 # the frequency of steps used to update the plot

# initialize n walkers and put them in a list:
WalkerList = []
for w in range(n):
    WalkerList.append(Walker(0,0))

# list to store R squared average (i.e. the rms) *over all the walkers* after N steps:
R_rms_list = []
# now go through all the n walkers and get them to perform N steps
for i in tqdm(range(N)):
    Rsq_av = 0 # variable used to calculate the square average
    # go through all the walkers in the WalkerList
    for j, walker in enumerate(WalkerList):
        # move them one step in a random direction
        walker.move(step)
        # add to the average:
        Rsq_av = Rsq_av + walker.distance()**2
    # get the average and push it into the list:
```

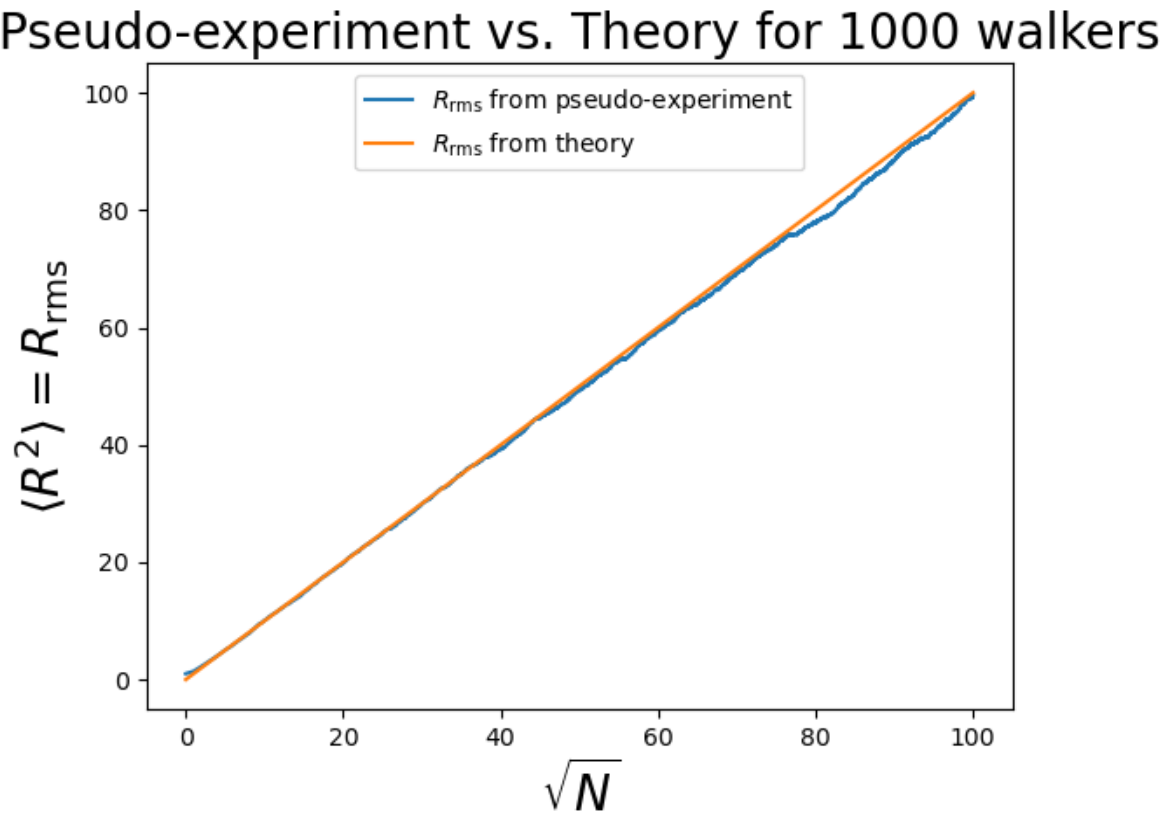
(continues on next page)

(continued from previous page)

```
    R_rms_list.append(math.sqrt(Rsq_av/n)) # divide by the number of walkers to get␣
 ↪the average
```

```
100%|██████████| 10000/10000 [00:10<00:00, 960.48it/s]
```

```
import matplotlib.pyplot as plt
import math

# first, create a list for square root(N):
sqrtN = [math.sqrt(x) for x in range(N)]

# the theoretical expectation is R_rms = sqrt(N) * r_rms (where r_rms = step in this␣
 ↪case):
R_rms_theory = [math.sqrt(y)*step for y in range(N)]

# setup the axis:
fig,ax = plt.subplots(1,1)

# set the labels and titles:
ax.set_xlabel(r'$\sqrt{N}$', fontsize=20) # set the x label
ax.set_ylabel(r'$\left<R^2\right> = R_\mathrm{rms}$', fontsize=20) # set the y label.␣
 ↪Note that the 'r' is necessary to remove the need for double slashes. You can use␣
 ↪LaTeX!
ax.set_title('Pseudo-experiment vs. Theory for ' + str(n) + ' walkers' , fontsize=20)
 ↪# set the title

# plot the pseudo-experiment result
ax.plot(sqrtN, R_rms_list, ls='-', label=r'$R_\mathrm{rms}$ from pseudo-experiment')

# plot the theoretical expectation
ax.plot(sqrtN, R_rms_theory, ls='-', label=r'$R_\mathrm{rms}$ from theory')

# construct the legend:
ax.legend(loc='upper center')  # Add a legend

plt.show() # show the plot here
```




E.g. for `n = 1000` walkers and `N = 10000` steps, the agreement is excellent!

CHAPTER

FOUR

# NUMERICAL DIFFERENTIATION AND INTEGRATION

## 4.1 Numerical Differentiation

### 4.1.1 Introduction

Suppose we have the trajectory of a projectile moving in the vertical direction as a function of time, $y(t)$.

If we wish to determine its speed, we would need to take the derivative of the position with respect to time, i.e. $v(t) = \mathrm{d}y/\mathrm{d}t$.

e.g. suppose the trajectory, *measured experimentally*, looks like the following plot:

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

t = np.linspace(0, 2 * np.pi, 10) # creates a NumPy array for time
y = 5*np.power(t,2) - t * np.exp(-t) - 0.9*np.power(t,3) # the function

fig, ax = plt.subplots() # create the elements required for matplotlib.

# set the labels and titles:
ax.set_xlabel(r'$t$ [s]', fontsize=20) # set the x label
ax.set_ylabel(r'$y(t)$ [m]', fontsize=20) # set the y label.
ax.set_title('The trajectory of a particle moving vertically', fontsize=15) # set the␣
↪title

# make a one-dimensional plot using the above arrays, add a custom label
ax.plot(t, y, lw=0, ms=4, marker='o')

ax.set_ylim([-30, 30]) # set the y limit

plt.show() # show the plot here
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 import matplotlib.pyplot as plt # import matplotlib, a conventional module␣
↪name is plt
      2 import numpy as np
      4 t = np.linspace(0, 2 * np.pi, 10) # creates a NumPy array for time

ModuleNotFoundError: No module named 'matplotlib'
```

Assuming we don't know the analytic form of the function that generated the above, we would like to calculate the speed $v(t)$ numerically.

The starting point is the mathematical definition of the derivative:

$\frac{\mathrm{d}y(t)}{\mathrm{d}t} \equiv \lim_{h\to 0} \frac{y(t+h)-y(t)}{h}$.

Note that a computer would run into errors with this kind of limit:

- as $h$ is made smaller, the numerator would fluctuate between 0 and machine precision, $\epsilon_m$.
- as the denominator, $h$, approaches zero, overflow would occur.

### 4.1.2 The Forward-Difference Derivative

The simplest algorithm for numerical differentiation is called the forward-difference derivative, and it is obtained by simply removing the limit, still taking $h$ reasonably small, i.e.:

$\left.\frac{\mathrm{d}y(t)}{\mathrm{d}t}\right|_{\mathrm{fd}} \equiv \frac{y(t+h)-y(t)}{h}$.

What is the algorithmic (or approximation) error induced by this procedure?

To find out, let's look at the Taylor series of $y(t)$ a small step $h$ away from $y(t)$:

$y(t+h) = y(t) + h\frac{\mathrm{d}y(t)}{\mathrm{d}t} + \frac{h^2}{2!}\frac{\mathrm{d}^2y(t)}{\mathrm{d}t^2} + \frac{h^3}{3!}\frac{\mathrm{d}^3y(t)}{\mathrm{d}t^3} + ....$

Solving for $\frac{y(t+h)-y(t)}{h}$, which is nothing but the forward-difference derivative:

$\left.\frac{\mathrm{d}y(t)}{\mathrm{d}t}\right|_{\mathrm{fd}} = \frac{y(t+h)-y(t)}{h} = \frac{\mathrm{d}y(t)}{\mathrm{d}t} - \frac{h}{2}\frac{\mathrm{d}^2y(t)}{\mathrm{d}t^2} + ...$

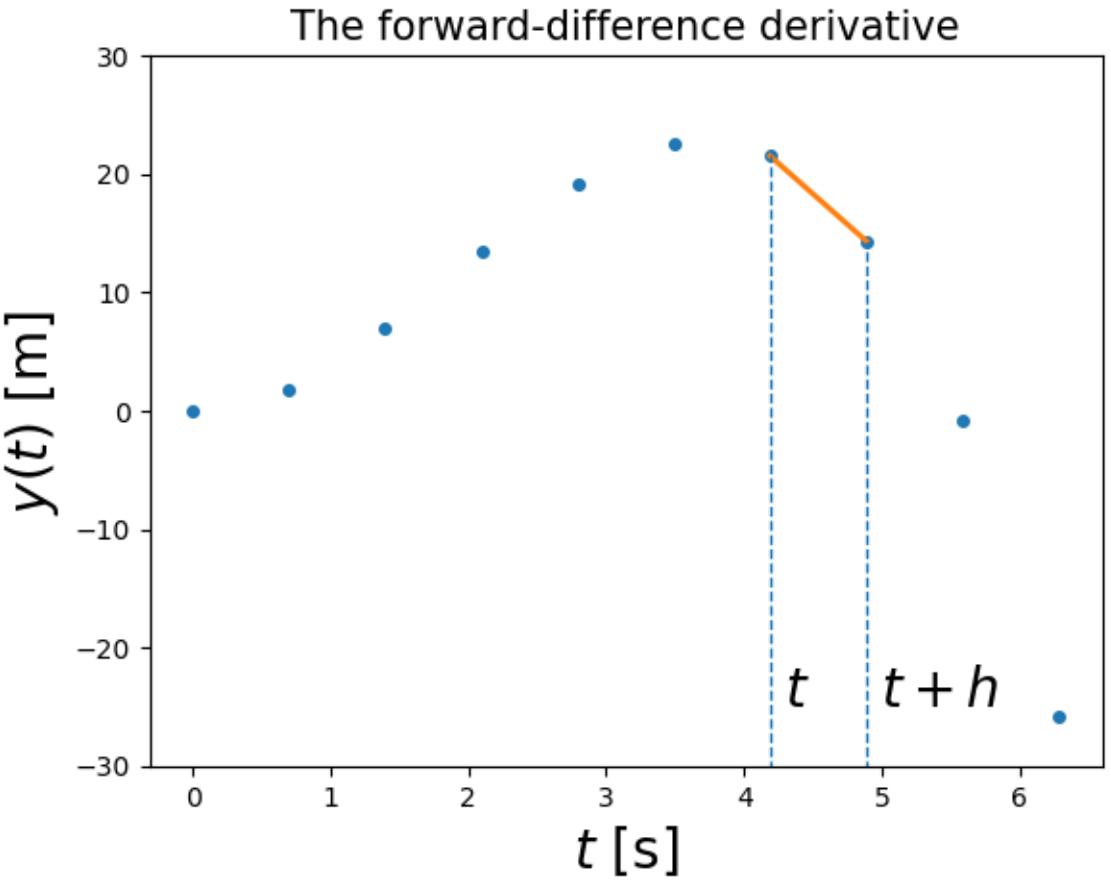
Therefore, the algorithmic error for the forward-difference derivative can be written as:

$\epsilon^{\mathrm{fd}}_{\mathrm{alg}} \simeq \frac{hy''}{2}$, where $y''$ is short-hand for the second derivative.

You can think of the approximation as using two points to represent the function by a straight line in the interval $t$ to $t+h$:

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

t = np.linspace(0, 2 * np.pi, 10) # creates a NumPy array for time
y = 5*np.power(t,2) - t * np.exp(-t) - 0.9*np.power(t,3) # take the NumPy array and␣
 ↪create another one
fig, ax = plt.subplots() # create the elements required for matplotlib.
# set the labels and titles:
ax.set_xlabel(r'$t$ [s]', fontsize=20) # set the x label
ax.set_ylabel(r'$y(t)$ [m]', fontsize=20) # set the y label.
ax.set_title('The forward-difference derivative', fontsize=15) # set the title

ax.set_ylim([-30, 30]) # set the y limit

# make a one-dimensional plot using the above arrays, add a custom label
ax.plot(t, y, lw=0, ms=4, marker='o')

# plot a segment of the line and the corresponding t values:
ax.plot(t[6:8], y[6:8], lw=2, ms=0)
ax.vlines(t[6], ymin=-40, ymax=y[6], lw=1, ls='--')
ax.vlines(t[7], ymin=-40, ymax=y[7], lw=1, ls='--')
ax.text(t[6]+0.1, -25, '$t$', fontsize='20')
ax.text(t[7]+0.1, -25, '$t+h$', fontsize='20')

plt.show() # show the plot here
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)␣
↪SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel␣
↪Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)␣
↪instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)␣
↪SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel␣
↪Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)␣
↪instructions.
```



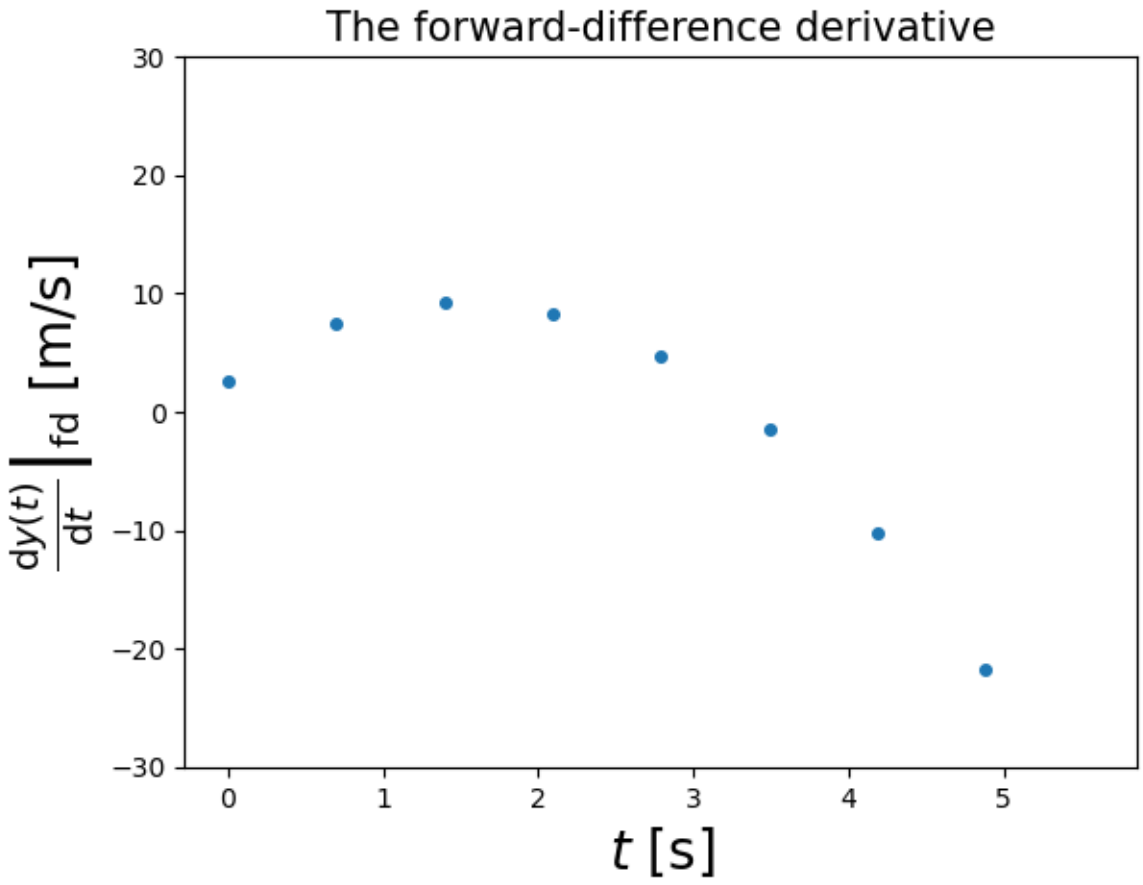

**Example 4.1: Calculate the forward-difference derivative for the above data points and plot it (in a NumPy array called `y`)**

```
# The forward difference for the points "measured" in our pseudo-experiment:

# the forward difference:
dydt_FD = [ (y[i+1] - y[i]) / (t[i+1] - t[i]) for i in range(len(y)-1)]

fig, ax = plt.subplots() # create the elements required for matplotlib.

# set the labels and titles:
ax.set_xlabel(r'$t$ [s]', fontsize=20) # set the x label
ax.set_ylabel(r'$\left. \frac{\mathrm{d} y(t)}{\mathrm{d}t} \right|_\mathrm{fd}$ [m/s]
↪', fontsize=20) # set the y label.
ax.set_title('The forward-difference derivative', fontsize=15) # set the title
```

(continues on next page)

(continued from previous page)

```
ax.set_ylim([-30, 30]) # set the y limit

# make a one-dimensional plot using the above arrays, add a custom label
ax.plot(t[:-1], dydt_FD, lw=0, ms=4, marker='o')

plt.show() # show the plot here
```




### 4.1.3 Example 4.2: Calculate the forward-difference derivatives for f(t) = \cos t and g(t) = \exp(t) at t=0.1, 1., 100, for h=10^{-6}. Compare to the analytic results.

```
import math

# define a function for the forward-difference derivative.
# In Python, you can pass a function as an argument to another function. This is␣
↪known as a higher-order function.
def dfdt_FD(func, t, h):
    """Calculates the forward-difference derivative of a function func at t, with␣
↪step size h"""
    return (func(t+h) - func(t))/h

# calculate the derivatives at various points in an array t:
tarray = [0.1, 1., 100.]
```

(continues on next page)

(continued from previous page)

```
# the step h is taken to be 1E-6:
h = 1E-6

# for the cos function:
for t in tarray:
    print('d(cos(t))/dt at t=', t, '=', dfdt_FD(math.cos,t,h), 'vs analytic=', -math.
 ↪sin(t), 'with error=', abs(dfdt_FD(math.cos,t,h)-(-math.sin(t))))

# for the exp function:
for t in tarray:
    print('d(exp(t))/dt at t=', t, '=', dfdt_FD(math.exp,t,h), 'vs analytic=', math.
 ↪exp(t), 'with error=', abs(dfdt_FD(math.exp,t,h)-(math.exp(t))))
```

```
d(cos(t))/dt at t= 0.1 = -0.09983391413559417 vs analytic= -0.09983341664682815␣
 ↪with error= 4.974887660158522e-07
d(cos(t))/dt at t= 1.0 = -0.8414712548976411 vs analytic= -0.8414709848078965 with␣
 ↪error= 2.7008974456954604e-07
d(cos(t))/dt at t= 100.0 = 0.5063652086523263 vs analytic= 0.5063656411097588 with␣
 ↪error= 4.324574325309527e-07
d(exp(t))/dt at t= 0.1 = 1.105171470561217 vs analytic= 1.1051709180756477 with␣
 ↪error= 5.524855692939923e-07
d(exp(t))/dt at t= 1.0 = 2.7182831874306146 vs analytic= 2.718281828459045 with␣
 ↪error= 1.3589715694983795e-06
d(exp(t))/dt at t= 100.0 = 2.6881184788443713e+43 vs analytic= 2.
 ↪6881171418161356e+43 with error= 1.3370282356702364e+37
```

We will discuss errors in a bit! First let's consider more algorithms for differentiation!

### 4.1.4 The Central-Difference Derivative

Rather than making a single step $h$ forward, we can make half a step forward and half a step backward to get the central-difference derivative:

$\left.\frac{\mathrm{d}y(t)}{\mathrm{d}t}\right|_{\mathrm{cd}} \equiv \frac{y(t+h/2)-y(t-h/2)}{h}$.

To get an estimate of the algorithmic error, we first write down the Taylor series for $y(t+h/2)$ and $y(t-h/2)$:

$y(t+h/2) = y(t) + \frac{h}{2}y'(t) + \frac{h^2}{8}y''(t) + \frac{h^3}{48}y'''(t) + \mathcal{O}(h^4)$

$y(t-h/2) = y(t) - \frac{h}{2}y'(t) + \frac{h^2}{8}y''(t) - \frac{h^3}{48}y'''(t) + \mathcal{O}(h^4)$

and we subtract them to get:

$y(t+h/2) - y(t-h/2) = hy'(t) + \frac{h^3}{24}y'''(t) + \mathcal{O}(h^5)$

All terms with an even power of $h$ vanish!

so:

$\left.\frac{\mathrm{d}y(t)}{\mathrm{d}t}\right|_{\mathrm{cd}} = y'(t) + \frac{1}{24}h^2y'''(t) + \mathcal{O}(h^5)$.

The central-difference algorithm is accurate to order $h^2$ instead of $h$.

Therefore, if the function is smooth, then one would expect $y'''h^2/24 \ll y''h/2$, and we would expect the central-difference error to be much smaller than that with the forward-difference.

Let's compare the results of Example 4.2, calculated with the forward-difference algorithm, now with the central-difference algorithm:

### Example 4.3: Calculate as in Example 4.2, but with the central-difference derivative

```
# define a function for the central-difference derivative.
def dfdt_CD(func, t, h):
    """Calculates the central-difference derivative of a function func at t, with␣
 ↪step size h"""
    return (func(t+h/2) - func(t-h/2))/h

# calculate the derivatives at various points in an array t:
tarray = [0.1, 1., 100.]

# the step h is taken to be 1E-8:
h = 1E-6

# for the cos function:
for t in tarray:
    print('d(cos(t))/dt at t=', t, '=', dfdt_CD(math.cos,t,h), 'vs analytic=', -math.
 ↪sin(t), 'with error=', abs(dfdt_CD(math.cos,t,h)-(-math.sin(t))))

# for the exp function:
for t in tarray:
    print('d(exp(t))/dt at t=', t, '=', dfdt_CD(math.exp,t,h), 'vs analytic=', math.
 ↪exp(t), 'with error=', abs(dfdt_CD(math.exp,t,h)-(math.exp(t))))
```

```
d(cos(t))/dt at t= 0.1 = -0.09983341664465684 vs analytic= -0.09983341664682815␣
 ↪with error= 2.17131868041065e-12
d(cos(t))/dt at t= 1.0 = -0.8414709847803792 vs analytic= -0.8414709848078965 with␣
 ↪error= 2.7517321754544355e-11
d(cos(t))/dt at t= 100.0 = 0.506365639862949 vs analytic= 0.5063656411097588 with␣
 ↪error= 1.246809766541901e-09
d(exp(t))/dt at t= 0.1 = 1.10517091811424 vs analytic= 1.1051709180756477 with␣
 ↪error= 3.859224051439014e-11
d(exp(t))/dt at t= 1.0 = 2.7182818285176324 vs analytic= 2.718281828459045 with␣
 ↪error= 5.858735718788921e-11
d(exp(t))/dt at t= 100.0 = 2.6881171349366646e+43 vs analytic= 2.
 ↪6881171418161356e+43 with error= 6.879470977972417e+34
```

For the specific choice of $h$, the central-difference algorithm is clearly better! (But see below for an analysis of errors in each).

## 4.1.5 Extrapolated-Difference Derivative

We can extend the central-difference approximation to use quarter steps instead of half steps, i.e., calculate:

$\left.\frac{\mathrm{d}y(t,h/2)}{\mathrm{d}t}\right|_{\mathrm{cd}} \equiv \frac{y(t+h/4)-y(t-h/4)}{h/2}$.

As before, expanding via Taylor series $y(t+h/4)$ and $y(t-h/4)$ yields an estimate of the approximation error:

$\left.\frac{\mathrm{d}y(t,h/2)}{\mathrm{d}t}\right|_{\mathrm{cd}} \simeq y'(t) + \frac{h^2}{96}y'''(t) + \mathcal{O}(h^4)$

If we define $D_{\mathrm{cd}}y(t,h) \equiv \left.\frac{\mathrm{d}y(t,h)}{\mathrm{d}t}\right|_{\mathrm{cd}}$, we can combine two central-difference approximations to get the *extended-difference algorithm* as follows:

$\left.\frac{\mathrm{d}y(t)}{\mathrm{d}t}\right|_{\text{ed}} \equiv \frac{4D_{\text{cd}}y(t,h/2)-D_{\text{cd}}y(t,h)}{3}$.

And it can be shown (again via Taylor series expansions) that this algorithm has an error of $\mathcal{O}(h^4)$:

$\left.\frac{\mathrm{d}y(t)}{\mathrm{d}t}\right|_{\text{ed}} \simeq y'(t) - \frac{h^4}{4\cdot16\cdot120}y^{(5)}(t) + ....$

Finally, when working with the aproximations, it is important to remember that they may work for well-behaved functions, they may fail badly for functions containing noise, which will most likely be the case if they are coming from measurements or other computations. In that case, it may be necessary to "smooth" the data or fit them with some analytic function (see later chapter) and then differentiate.

## 4.2 Error Assessment in Numerical Differentiation

### 4.2.1 Introduction

Numerical algorithms play a vital role in computational physics. When you encounter/invent and implement an algorithm, you must assess the following:

1. Does it converge?
2. How precise are the converged results?
3. How fast does it run?

You may think that all algorithms converge if enough terms are used, and that if you want more precision you just use more terms!

But this is not always possible: some algorithms are asymptotic expansions that approximate a function in certain regions of parameter space, and converge only up to a point.

However, even if the series that underlies an algorithm is uniformly convergent, including more terms will decrease the algorithmic error, but will also *increase* the round-off errors that we previously discussed. And because round-off errors eventually diverge to infinity, we need to find the "sweet-spot" for the approximations.

Good algorithms require fewer steps and thus incur less round-off error!

The algorithmic (or approximation) errors, $\epsilon_{\text{alg}}$ in numerical differentiation decrease with decreasing step size, $h$. In turn, round-off errors, $\epsilon_{\text{ro}}$, increase with decreasing step size.

The best value of $h$ has to minimize the total error:

$\epsilon_{\text{tot}} = \epsilon_{\text{alg}} + \epsilon_{\text{ro}}$.

This "sweet spot" occurs when both errors are of the same order, i.e.:

$\epsilon_{\text{alg}} \simeq \epsilon_{\text{ro}}$.

In the case of numerical differentiation, e.g. via the forward-difference algorithm, $\left.\frac{\mathrm{d}y(t)}{\mathrm{d}t}\right|_{\text{fd}} = \frac{y(t+h)-y(t)}{h}$, as $h$ is made continually smaller, we will eventually reach the worst-case round-off error where $y(t+h)$ and $y(t)$ differ by the machine precision $\epsilon_m$. In that worst-case scenarion, the round-off error is:

$\epsilon_{\text{ro}} \simeq \frac{\epsilon_m}{h}$.

Therefore, for the forward-difference algorithm, the sweet-spot $h$ occurs when:

$\frac{\epsilon_m}{h} \simeq \epsilon_{\text{alg}}^{\text{fd}} = \frac{hy''}{2}$ or $h_{\text{fd}} = \sqrt{\frac{2\epsilon_m}{y''}}$.

For the central-difference algorithm:

$\frac{\epsilon_m}{h} \simeq \epsilon_{\text{alg}}^{\text{cd}} = \frac{h^2y'''}{24}$ or $h_{\text{cd}} = \sqrt[3]{\frac{24\epsilon_m}{y'''}}$.

Let's assume that we have a function for which $y' \simeq y'' \simeq y'''$ at the point of interest $t$, which may be crude in general, but works reasonably for $\cos t$ and $\exp(t)$. For double precision floats, $\epsilon_m \simeq 10^{-15}$, and then:

$h_{\text{fd}} \simeq 4 \times 10^{-8}$ which leads to $\epsilon_{\text{fd}} \simeq \frac{\epsilon_m}{h_{\text{fd}}} \simeq 3 \times 10^{-8}$,

and:

$h_{\text{cd}} \simeq 3 \times 10^{-5}$ which leads to $\epsilon_{\text{cd}} \simeq \frac{\epsilon_m}{h_{\text{cd}}} \simeq 3 \times 10^{-11}$,

One can observe that in this case, the central-difference algorithm achieves smaller error for a larger $h$ value! Let's now re-calculate the values of the derivatives of $\cos t$ and $\exp(t)$ using the "sweet-spot" step sizes in the next example.

### Example 4.4: Use the sweet-spot values of the step size to calculate the derivatives of \cos (t) and \exp(t) at the same points as the previous examples.

```
# We can use the functions already defined in this notebook for the forward␣
 ↪difference and central difference.

# Forward difference:
# the step h is taken to be 4E-8:
h = 4E-8

print('Forward difference:')
# for the cos function:
for t in tarray:
    print('d(cos(t))/dt at t=', t, '=', dfdt_FD(math.cos,t,h), 'vs analytic=', -math.
 ↪sin(t), 'with error=', abs(dfdt_FD(math.cos,t,h)-(-math.sin(t))))

# for the exp function:
for t in tarray:
    print('d(exp(t))/dt at t=', t, '=', dfdt_FD(math.exp,t,h), 'vs analytic=', math.
 ↪exp(t), 'with error=', abs(dfdt_FD(math.exp,t,h)-(math.exp(t))))

# Central difference:
# the step h is taken to be 3E-5:
h = 3E-5

print('\nCentral difference:')
# for the cos function:
for t in tarray:
    print('d(cos(t))/dt at t=', t, '=', dfdt_CD(math.cos,t,h), 'vs analytic=', -math.
 ↪sin(t), 'with error=', abs(dfdt_CD(math.cos,t,h)-(-math.sin(t))))

# for the exp function:
for t in tarray:
    print('d(exp(t))/dt at t=', t, '=', dfdt_CD(math.exp,t,h), 'vs analytic=', math.
 ↪exp(t), 'with error=', abs(dfdt_CD(math.exp,t,h)-(math.exp(t))))
```

```
Forward difference:
d(cos(t))/dt at t= 0.1 = -0.09983343596253746 vs analytic= -0.09983341664682815␣
 ↪with error= 1.9315709309797313e-08
d(cos(t))/dt at t= 1.0 = -0.8414709956605648 vs analytic= -0.8414709848078965 with␣
 ↪error= 1.085266831957199e-08
d(cos(t))/dt at t= 100.0 = 0.5063656660642124 vs analytic= 0.5063656411097588 with␣
 ↪error= 2.4954453614611793e-08
d(exp(t))/dt at t= 0.1 = 1.1051709380982544 vs analytic= 1.1051709180756477 with␣
 ↪error= 2.0022606683767208e-08
```

(continues on next page)

(continued from previous page)

```
d(exp(t))/dt at t= 1.0 = 2.7182818884696758 vs analytic= 2.718281828459045 with␣
 ↪error= 6.001063068694634e-08
d(exp(t))/dt at t= 100.0 = 2.6881174087690013e+43 vs analytic= 2.
 ↪6881171418161356e+43 with error= 2.669528657119537e+36

Central difference:
d(cos(t))/dt at t= 0.1 = -0.09983341664465684 vs analytic= -0.09983341664682815␣
 ↪with error= 2.17131868041065e-12
d(cos(t))/dt at t= 1.0 = -0.8414709847803792 vs analytic= -0.8414709848078965 with␣
 ↪error= 2.7517321754544355e-11
d(cos(t))/dt at t= 100.0 = 0.5063656412507278 vs analytic= 0.5063656411097588 with␣
 ↪error= 1.4096901423954478e-10
d(exp(t))/dt at t= 0.1 = 1.10517091811424 vs analytic= 1.1051709180756477 with␣
 ↪error= 3.859224051439014e-11
d(exp(t))/dt at t= 1.0 = 2.718281828576844 vs analytic= 2.718281828459045 with␣
 ↪error= 1.177991038048276e-10
d(exp(t))/dt at t= 100.0 = 2.6881171427769514e+43 vs analytic= 2.
 ↪6881171418161356e+43 with error= 9.608157724429865e+33
```

These are of the same order as we expected from our considerations!

Let's now dig deeper into the errors to get a visual representation of what's happening.

**Example 4.5: Using the known analytical results calculate the *relative error*, \varepsilon, obtained by both methods for the \cos(t) and \exp(t) functions at t=0.1, for varying step size h down to h\simeq \epsilon_m \simeq 10^{-15} and plot \log_{10}|\varepsilon| vs. \log_{10} h. Comment on the behavior at small and large h**

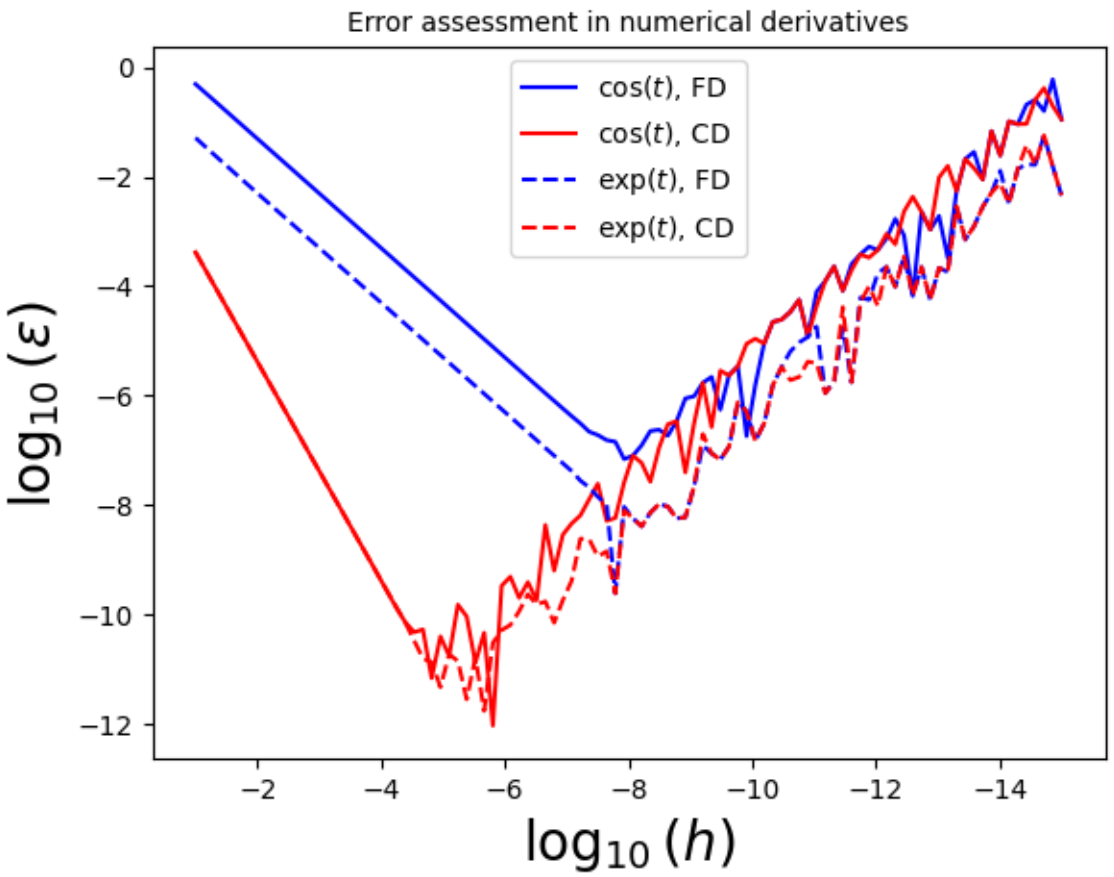
```
import numpy as np
# First, let's create the array of h:
harray = np.logspace(-1, -15, num=100,base=10) # the NumPy logspace returns numbers␣
 ↪spaced evenly on a log scale.
```

```
# Then let's calculate the relative error and take the log (base 10) for each of the␣
 ↪two algorithms:

# fix the point we are examining:
t = 0.1

# Let's use list comprehension!
# cosine:
log10_rel_error_cos_FD = [math.log10( abs( (dfdt_FD(math.cos,t,h) - (-math.sin(t))) /␣
 ↪-math.sin(t) ) ) for h in harray] # Forward-difference
log10_rel_error_cos_CD = [math.log10( abs( (dfdt_CD(math.cos,t,h) - (-math.sin(t))) /␣
 ↪-math.sin(t) ) ) for h in harray] # Centra-Difference

# exp:
log10_rel_error_exp_FD = [math.log10( abs( (dfdt_FD(math.exp,t,h) - (math.exp(t))) /␣
 ↪math.exp(t) ) ) for h in harray] # Forward-difference
log10_rel_error_exp_CD = [math.log10( abs( (dfdt_CD(math.exp,t,h) - (math.exp(t))) /␣
 ↪math.exp(t) ) ) for h in harray] # Centra-Difference


# get the log10 of h:
log10_h = [math.log10(h) for h in harray]
```

(continues on next page)

(continued from previous page)

```
# Now plot! Don't forget the different labels!
fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$\log_{10}(h)$', fontsize=20) # set the x label
ax.set_ylabel(r'$\log_{10}(\varepsilon)$', fontsize=20) # set the y label. Note that␣
↪the 'r' is necessary to remove the need for double slashes. You can use LaTeX!
ax.set_title('Error assessment in numerical derivatives', fontsize=10) # set the␣
↪title

# make a one-dimensional plot using the arrays relevant to the cosine:
ax.plot(log10_h, log10_rel_error_cos_FD, label=r'$\cos(t)$, FD', ls='-', color='blue')
ax.plot(log10_h, log10_rel_error_cos_CD, label=r'$\cos(t)$, CD', ls='-', color='red')
# make a one-dimensional plot using the arrays relevant to the exponential:
# change the linestyle!
ax.plot(log10_h, log10_rel_error_exp_FD, label=r'$\exp(t)$, FD', ls='--', color='blue
↪')
ax.plot(log10_h, log10_rel_error_exp_CD, label=r'$\exp(t)$, CD', ls='--', color='red')

# ivnert the axes to show smaller values of h to the right:
ax.invert_xaxis()

# construct the legend:
ax.legend(loc='upper center')  # Add a legend

plt.show() # show the plot here
```




Now let's do the same exercise in class, with $f(t) = \exp(t)\cos(t)$!

**Example 4.6: Take the numerical derivative of the function f(t) = \exp(t) \cos(t) at t=0.1, and following Example 4.5, perform an error assessment.**

(Solution in a separate notebook, Example4.6)

## 4.3 Numerical Integration

### 4.3.1 Introduction

A traditional way to perform integration by hand is to take a piece of graph paper and count the number of boxes, or *quadrilaterals* lying below the curve.

This is why numerical integration is also known as numerical *quadrature*, even when the method employed is much more sophisticated than simple box counting.

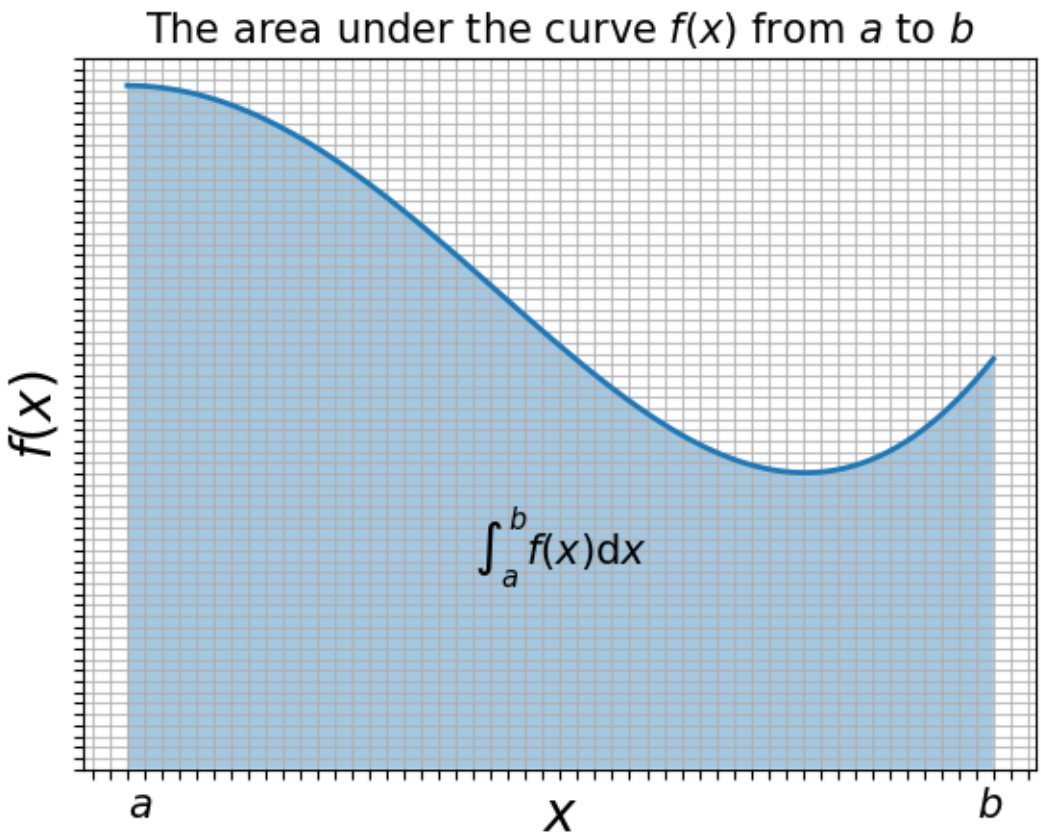
We start of our discussion of numerical integration with the Riemann definition of an integral, as the limit of the sum over boxes under a curve, as the box width $h$ approaches zero:

$\int_a^b f(x)\mathrm{d}x = \lim_{h\to 0}\left[h\sum_{i=1}^{(b-a)/h} f(x_i)\right]$,

where $(b-a)/h = N$ counts the number of boxes.

The figure below is a graphical representation of the quadrature procedure:

```
import matplotlib.ticker as ticker #

# define the x array and calculate over it the function to display
x = np.linspace(0, 1,100)
f = 0.5-np.power(x,2) + np.power(x,4) - 0.2* np.power(x,6)

fig, ax = plt.subplots() # create the elements required for matplotlib.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$f(x)$', fontsize=20) # set the y label.
ax.set_title(r'The area under the curve $f(x)$ from $a$ to $b$', fontsize=15) # set␣
 ↪the title

# put the ticks closer together:
ax.xaxis.set_major_locator(ticker.MultipleLocator(0.02))
ax.yaxis.set_major_locator(ticker.MultipleLocator(0.008))

# set the axis limits:
ax.set_ylim(0,0.520)

# make a one-dimensional plot using the above arrays, add a custom label
ax.plot(x, f, lw=2)

# fill the area under the curve:
ax.fill_between(x, 0, f,alpha=0.4)

# put a grid on the graph:
ax.grid(True, which='both', alpha=0.8)

# hide x and y axes label marks
ax.xaxis.set_tick_params(labelbottom=False)
ax.yaxis.set_tick_params(labelleft=False)

# put text at the right points on the x-axis:
ax.text(0,-0.035, '$a$', fontsize=15)
ax.text(0.98,-0.035, '$b$', fontsize=15)

# put the integral in the shaded region:
ax.text(0.4, 0.15, r'$\int_a^b f(x) \mathrm{d} x $', fontsize=15)


plt.show() # show the plot here
```




The numerical integral of a function $f(x)$ is approximated as a finite sum over boxes of height $f(x_i)$ and width $w_i$:

$\int_a^b f(x)\mathrm{d}x \simeq \sum_{i=1}^{N} f(x_i)w_i$.

In this case the box width is finite. The above formula represents the standard form for all integration algorithms: the function $f(x)$ is evaluated at $N$ points in the interval $[a, b]$, and the function values $f_i \equiv f(x_i)$ are summed with each term in the sum weighted by $w_i$.

Generally, the precision increases as the number of points $N$ gets larger, with the round-off error eventually limiting the increase. This is similar to what we observed in the case of differentiation.

A word of warning: in general, you should not attempt a numerical integration of an integrand that contains a singularity, without first removing the singularity by hand. You may be able to do this very simply by breaking the intervals down into several subintervals so that the singularity is at an endpoint where an integration point is not placed, or by change of variable.

Also, if the integrand has a very slow variation in some region, you can speed up the integration by changing to a variable that compresses that region and places fewer points there, or divides up the integral and performs several integrations. | Conversely, if your integrand has a very rapid variation in some region, you may want to change to variables that expand that region, to ensure that no oscillations (e.g.) are missed.

### 4.3.2 The Trapezoid Rule

The trapezoid (British: trapezium) rule uses *evenly-spaced values* of $x$, i.e. $N$ points $x_i$ ($i = 1, ...N$), spaced at a distance $h$ apart throughout the integration region $[a, b]$ and includes the endpoints of the integration region.

This means that there are $(N - 1)$ intervals of length $h$:

$h = \frac{b-a}{N-1}$, $x_i = a + (i - 1)h$, $i = 1, ..., N$,

where we note that counting starts at $i = 1$ in the above formulae.

The trapezoid rule takes each integration interval $i$ and constructs a trapezoid of width $h$ in it. This approximates $f(x)$ by a straight line in each interval $i$ and uses the average height of the function at the edges of the straight line as the value for $f$, i.e. $(f_i + f_{i+1})/2$.

The area of each such trapzoid is then:

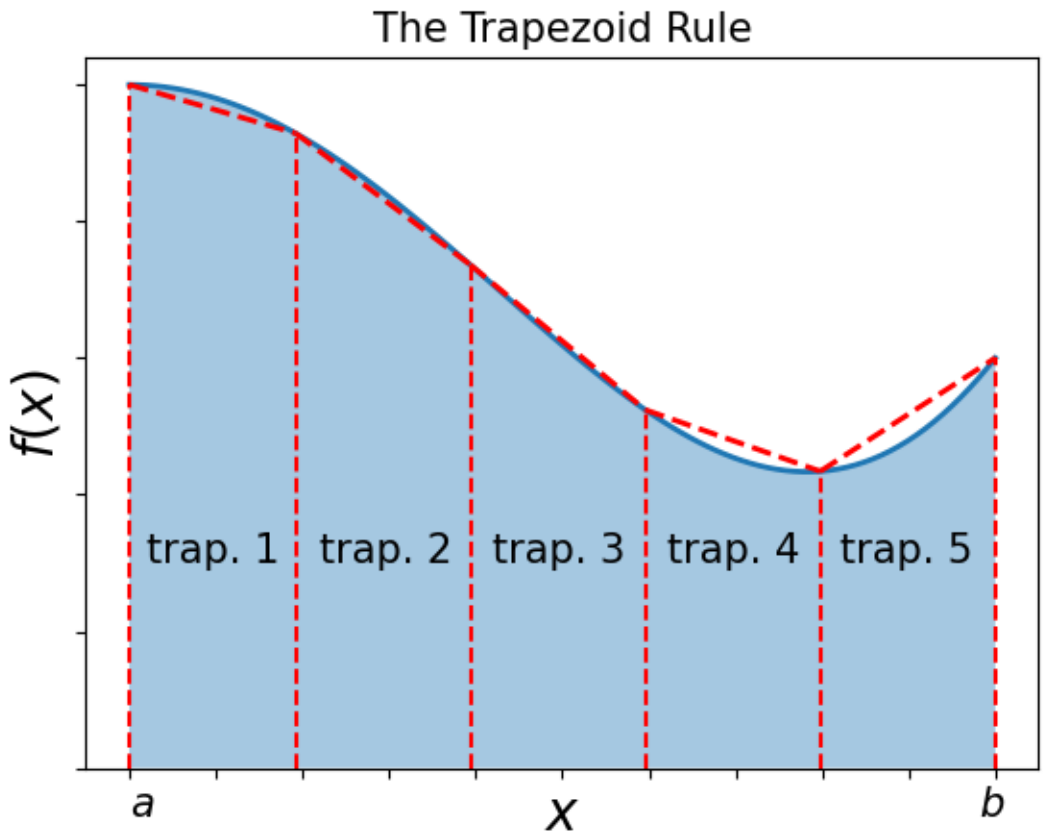
$\int_{x_i}^{x_i+h} f(x)\mathrm{d}x \simeq h\frac{(f_i+f_{i+1})}{2} = \frac{1}{2}hf_i + \frac{1}{2}hf_{i+1}$

```
import numpy as np
import matplotlib.ticker as ticker #

# define the x array and calculate over it the function to display
x = np.linspace(0, 1,100)
f = 0.5-np.power(x,2) + np.power(x,4) - 0.2* np.power(x,6)

fig, ax = plt.subplots() # create the elements required for matplotlib.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$f(x)$', fontsize=20) # set the y label.
ax.set_title(r'The Trapezoid Rule', fontsize=15) # set the title

# put the ticks closer together:
ax.xaxis.set_major_locator(ticker.MultipleLocator(0.1))
ax.yaxis.set_major_locator(ticker.MultipleLocator(0.1))

# set the axis limits:
ax.set_ylim(0,0.520)

# make a one-dimensional plot using the above arrays, add a custom label
ax.plot(x, f, lw=2)

# draw vertical lines to denote the limits of the trapezoids:
ax.vlines(0, 0, f[0], ls='--',color='red')
ax.vlines(x[19], 0, f[19], ls='--', color='red')
ax.vlines(x[39], 0, f[39], ls='--', color='red')
ax.vlines(x[59], 0, f[59], ls='--', color='red')
ax.vlines(x[79], 0, f[79], ls='--', color='red')
ax.vlines(x[99], 0, f[99], ls='--', color='red')

# show the trapezoid edges:
ax.plot([x[0], x[19]], [f[0],f[19]], ls='--', lw=2, color='red')
ax.plot([x[19], x[39]], [f[19],f[39]], ls='--', lw=2, color='red')
ax.plot([x[39], x[59]], [f[39],f[59]], ls='--', lw=2, color='red')
ax.plot([x[59], x[79]], [f[59],f[79]], ls='--', lw=2, color='red')
ax.plot([x[79], x[99]], [f[79],f[99]], ls='--', lw=2, color='red')

# label the trapezoids:
```

(continues on next page)

(continued from previous page)

```
ax.text(0.02, 0.15, 'trap. 1', fontsize=15)
ax.text(0.22, 0.15, 'trap. 2', fontsize=15)
ax.text(0.42, 0.15, 'trap. 3', fontsize=15)
ax.text(0.62, 0.15, 'trap. 4', fontsize=15)
ax.text(0.82, 0.15, 'trap. 5', fontsize=15)


# fill the area under the curve:
ax.fill_between(x, 0, f,alpha=0.4)

# hide x and y axes label marks
ax.xaxis.set_tick_params(labelbottom=False)
ax.yaxis.set_tick_params(labelleft=False)

# put text at the right points on the x-axis:
ax.text(0,-0.035, '$a$', fontsize=15)
ax.text(0.98,-0.035, '$b$', fontsize=15)

plt.show() # show the plot here
```




In order to apply the trapezoid rule to the entire region $[a, b]$, we add the contributions from each subinterval:

$\int_a^b f(x)\mathrm{d}x \simeq \frac{h}{2}f_1 + hf_2 + hf_3 + ... + hf_{N-1} + \frac{h}{2}f_N$.

Note that the "internal" points are counted twice: the end of one interval is the beginning of the next.

The trapezoid rule can be written in terms of the "weight" formula as follows:

$\int_a^b f(x)\mathrm{d}x \simeq \sum_{i=1}^{N} f(x_i)w_i$, $w_i = \{\frac{h}{2}, h, ..., h, \frac{h}{2}\}$.

Let's write a function that implements the trapezoid rule!

**Example 4.7: Construct a function that implements the trapezoid rule on an interval [a,b] using N points, and use to integrate f(x) = \exp(-x) in the interval [0,1]. Compare to the analytic value for N=10^6 intervals.**

```
import math

# Let's write a higher-order function that implements the trapezoid rule:
def trapezoid(func, a, b, N):
    """Calculates the numerical integral of a function in the interval a,b using the␣
 ↪trapezoid rule for N points"""
    # calculate the width:
    h = (b - a)/(N-1)
    # apply the trapezoid rule:
    # the contributions from the first and last points:
    integral = h*(func(a) + func(b))/2
    # the contributions from the remaining points:
    for i in range(1, int(N-1)):
        integral = integral + h*func(a + i * h)
    return integral


# test the function for exp(-t) in [0,1]:
# first define the function we wish to integrate:
def f(x):
    return math.exp(-x)
N = 1E6 # the number of points
trap_int = trapezoid(f,0,1,N)
print('Integral of exp(-t) in [0,1] via the trapezoid rule=', trap_int)

# the analytic integral is simply 1 - exp(-1):
analytic_int = 1 - math.exp(-1)
print('Analytic integral=', analytic_int)

# compare by calculating the fractional error:
print('Fractional error=', abs( (trap_int-analytic_int)/analytic_int ) )
```

```
Integral of exp(-t) in [0,1] via the trapezoid rule= 0.632120558828617
Analytic integral= 0.6321205588285577
Fractional error= 9.378892789826624e-14
```

### 4.3.3 Simpson's Rule

Simpson's rule approximates the integrand, $f(x)$, by a parabola for each interval:

```
import numpy as np
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import matplotlib.ticker as ticker #

# define a function to plot the Simpson rule parabolas:
def SimpsonParabola(func, xinterval):
```

(continues on next page)

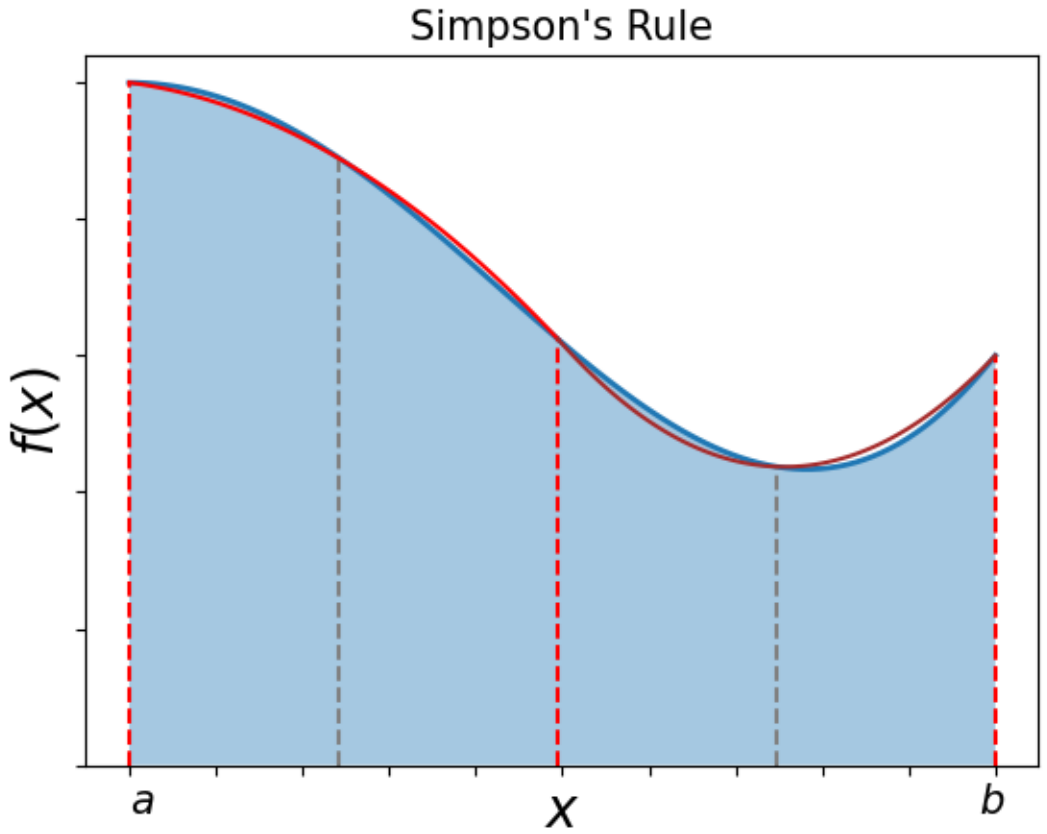
(continued from previous page)

```
    xx = np.linspace(xinterval[0],xinterval[-1],100)
    # calculate the coefficients of the quadratic:
    a = xinterval[0]
    b = xinterval[-1]
    fa = func(a)
    fb = func(b)
    fm = func( (a + b)/2 )
    alpha = (2*fa + 2*fb - 4*fm)/(a**2 - 2*a*b + b**2)
    beta = (-a*fa - 3*a*fb + 4*a*fm - 3*b*fa - b*fb + 4*b*fm)/(a**2 - 2*a*b + b**2)
    gamma = (a**2*fb + a*b*fa + a*b*fb - 4*a*b*fm + b**2*fa)/(a**2 - 2*a*b + b**2)
    return xx, alpha * np.power(xx,2) + beta * xx + gamma

# the "lambda" method allows you to define a function on a single line:
integrand = lambda y: 0.5-np.power(y,2) + np.power(y,4) - 0.2* np.power(y,6)

# define the x array and calculate over it the function to display
x = np.linspace(0, 1,100)
f = integrand(x)

fig, ax = plt.subplots() # create the elements required for matplotlib.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$f(x)$', fontsize=20) # set the y label.
ax.set_title(r"Simpson's Rule", fontsize=15) # set the title

# put the ticks closer together:
ax.xaxis.set_major_locator(ticker.MultipleLocator(0.1))
ax.yaxis.set_major_locator(ticker.MultipleLocator(0.1))

# set the axis limits:
ax.set_ylim(0,0.520)

# make a one-dimensional plot using the above arrays, add a custom label
ax.plot(x, f, lw=2)

# draw vertical lines to denote the limits of the intervals:
ax.vlines(0, 0, f[0], ls='--',color='red')
ax.vlines(x[24], 0, f[24], ls='--', color='grey')

ax.vlines(x[49], 0, f[49], ls='--', color='red')
ax.vlines(x[74], 0, f[74], ls='--', color='grey')
ax.vlines(x[99], 0, f[99], ls='--', color='red')

# Plot the Simpson rule parabolas:
xxp, yyp = SimpsonParabola(integrand, [0,x[49]])
ax.plot(xxp, yyp, color='red')
xxp, yyp = SimpsonParabola(integrand, [x[49],x[99]])
ax.plot(xxp, yyp, color='brown')

# fill the area under the curve:
ax.fill_between(x, 0, f,alpha=0.4)

# hide x and y axes label marks
ax.xaxis.set_tick_params(labelbottom=False)
ax.yaxis.set_tick_params(labelleft=False)
```

(continues on next page)

(continued from previous page)

```
# put text at the right points on the x-axis:
ax.text(0,-0.035, '$a$', fontsize=15)
ax.text(0.98,-0.035, '$b$', fontsize=15)

plt.show() # show the plot here
```




i.e.: $f(x) \simeq \alpha x^2 + \beta x + \gamma$

The intervals are equally spaced (red and grey dashes separate them). The area under the parabola for each interval is:

$\int_{x_i}^{x_i+h}(\alpha x^2 + \beta x + \gamma)\mathrm{d}x = \frac{\alpha x^3}{3} + \frac{\beta x^2}{2} + \gamma x\Big|_{x_i}^{x_i+h}$

For simplicity, consider an interval from -1 to 1:

$\int_{-1}^{1}(\alpha x^2 + \beta x + \gamma)\mathrm{d}x = \frac{2\alpha}{3} + 2\gamma$

But also notice that:

$f(-1) = \alpha - \beta + \gamma$, $f(0) = \gamma$ and $f(1) = \alpha + \beta + \gamma$.

Solving these for $\alpha$, $\beta$ and $\gamma$:

$\alpha = \frac{f(1)+f(-1)}{2} - f(0)$,

$\beta = \frac{f(1)-f(-1)}{2}$,

$\gamma = f(0)$.

We can substitute for $\alpha$ and $\gamma$ to express the integral as the weighted sum over the values of the function at these three points:

$\int_{-1}^{1}(\alpha x^2+\beta x+\gamma)\mathrm{d}x = \frac{f(-1)}{3}+\frac{4f(0)}{3}+\frac{f(1)}{3}$.

Three values of the function are needed in each interval!

However, if we combine two intervals, we only need to evaluate the function at the two endpoints and in the middle:

$\int_{x_i-h}^{x_i+h} f(x)\mathrm{d}x = \int_{x_i-h}^{x_i} f(x)\mathrm{d}x + \int_{x_i}^{x_i+h} f(x)\mathrm{d}x \simeq \frac{h}{3}f_{i-1}+\frac{4h}{3}f_i+\frac{h}{3}f_{i+1}$.

Therefore, Simpson's rule requires integration to be over pairs of intervals (i.e. separated by the grey lines in the figure above).

This implies that the total number of intervals must be even, and the total number of points must be odd!

In order to apply Simpson's rule to the entire interval, we add up the contributions from each *pair of subintervals*, counting all all but the first and last endpoints twice, i.e.:

$\int_a^b f(x)\mathrm{d}x \simeq \frac{h}{3}f_1+\frac{4h}{3}f_2+\frac{2h}{3}f_3+\frac{4h}{3}f_4+...+\frac{4h}{3}f_{N-1}+\frac{h}{3}f_N$.

In terms of the standard integration rule with weights:

$\int_a^b f(x)\mathrm{d}x \simeq \sum_{i=1}^{N} f(x_i)w_i$, $w_i = \left\{\frac{h}{3},\frac{4h}{3},\frac{2h}{3},\frac{4h}{3},...,\frac{4h}{3},\frac{h}{3}\right\}$.

The sum of the weights themselves provides a useful check on your implementation:

$\sum_{i=1}^{N} w_i = (N-1)h$.

And remember that the number of points $N$ must be odd for Simpson's rule!

**Example 4.8: Construct a function that implements Simpson's rule on an interval [a,b] using N points (N is odd!), and use to integrate f(x) = \exp(-x) in the interval [0,1].**

- Check that your sum of weights agrees with $\sum_{i=1}^{N} w_i = (N-1)h$.
- Compare to the analytic value for $N = 2001$ points.
- Also compare to the result obtained in Example 4.7 using the trapezoid rule and $N = 10^6$ points.

You should find that Simpson's rule with $N = 2001$ points performs better than the trapezoid rule with $N = 10^6$ points!

### 4.3.4 Gaussian Quadrature

In a similar manner as we did before, let's rewrite the basic integration formula as follows:

$\int_{-1}^{1} f(x)\mathrm{d}x \simeq \sum_{i=1}^{N} f(x_i)w_i$.

Gaussian quadrature consists of $N$ points and weights that are chosen to make the integration *exact* if $f(x)$ is a $(2N-1)$-degree polynomial!

In contrast to the equally-spaced rules (trapezoid/Simpson's) that we have already seen, there is never an integration point at the extremes of intervals. The values of the points and weights change as the number of points $N$ changes, and the points are *not equally spaced*.

For ordinary (also known as Gauss-Legendre) Gaussian integration, the points, $x_i$ turn out to be the $N$ zeros of the degree $N$ Legendre polynomial $P_N(x)$ on $[-1,1]$. The weights are:

$w_i = \frac{-2}{(N+1)P'_N(x_i)P_{N+1}(x_i)}$

where $P'_N(x)$ is the derivative of the Legendre polynomial. [See the course textbook for more details].

Note that for integrals on a finite interval, the following change of variables is used to reduce the $[a,b]$ interval to the standard interval:

$\int_a^b f(x)\mathrm{d}x = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{(b-a)x+a+b}{2}\right)\mathrm{d}x.$

Let's use Python special functions to calculate the first few Gauss-Legendre "nodes" and weights.

### Example 4.9: Use special functions to get the Gauss-Legendre points and weights for N=2 to N=8 points.

```
import scipy

# set the number of points
N=2
# we can access special functions using scipy.special.
# The roots of Legendre polynomials can be accessed via:
scipy.special.roots_legendre(N)

# the first array is the array of points and the second is the array pf weights:
xi, wi = scipy.special.roots_legendre(N)

# print them separately for N=2:
print('N=2 Gaussian quadrature, evaluated at:', xi)
print('with weights', wi)

# Now let's also print N=2 to N=8:
for n in range(3,9):
    xi, wi = scipy.special.roots_legendre(n)
    print('n=', n, 'Gaussian quadrature, evaluated at:', xi)
    print('with weights', wi,'\n')
```

```
N=2 Gaussian quadrature, evaluated at: [-0.57735027  0.57735027]
with weights [1. 1.]
n= 3 Gaussian quadrature, evaluated at: [-0.77459667  0.          0.77459667]
with weights [0.55555556 0.88888889 0.55555556]

n= 4 Gaussian quadrature, evaluated at: [-0.86113631 -0.33998104  0.33998104  0.
↪86113631]
with weights [0.34785485 0.65214515 0.65214515 0.34785485]

n= 5 Gaussian quadrature, evaluated at: [-0.90617985 -0.53846931  0.          0.
↪53846931  0.90617985]
with weights [0.23692689 0.47862867 0.56888889 0.47862867 0.23692689]

n= 6 Gaussian quadrature, evaluated at: [-0.93246951 -0.66120939 -0.23861919  0.
↪23861919  0.66120939  0.93246951]
with weights [0.17132449 0.36076157 0.46791393 0.46791393 0.36076157 0.17132449]

n= 7 Gaussian quadrature, evaluated at: [-0.94910791 -0.74153119 -0.40584515  0.  ␣
↪        0.40584515  0.74153119
  0.94910791]
with weights [0.12948497 0.27970539 0.38183005 0.41795918 0.38183005 0.27970539
 0.12948497]

n= 8 Gaussian quadrature, evaluated at: [-0.96028986 -0.79666648 -0.52553241 -0.
↪18343464  0.18343464  0.52553241
  0.79666648  0.96028986]
with weights [0.10122854 0.22238103 0.31370665 0.36268378 0.36268378 0.31370665
 0.22238103 0.10122854]
```

Let's now use this to evaluate an integral!

### Example 4.10: Construct a function that implements N-th order Gaussian quadrature in the interval [a,b], and use to integrate f(x) = \exp(-x) in the interval [0,1] for N=8.

Compare to the results obtained through the trapezoid rule with $N = 10^6$ points, and Simpson's rule with $N = 2001$ points.

```
import math
import scipy # we need scipy for the N-th order Legendre polynomials.

# Let's write a higher-order function that implements N-th order Gaussian quadrature:
def gauss(func, a, b, N):
    """Calculates the numerical integral of a function in the interval a,b using N-th␣
 ↪order Gaussian quadrature"""
    # N-th order Gaussian quadrature
    # get the weights and points from the scipy special function:
    xi, wi = scipy.special.roots_legendre(int(N))
    # now use the general formula to get the integral:
    integral = 0
    for j, wj in enumerate(wi):
        # calculate the x value using the xi array:
        xj = 0.5*((b-a) * xi[j] + b + a) # transform the xi[j] to the function␣
 ↪argument, xj
        integral = integral + wj * func(xj)
    return (b-a)/2 * integral

# test the function for exp(-t) in [0,1]:
# first define the function we wish to integrate:
def f(x):
    return math.exp(-x)
N = 8 # the order of the Gaussian quadrature
gauss_int = gauss(f,0,1,N)
print("Integral of exp(-t) in [0,1] via N=", 8, "Gaussian quadrature", gauss_int)

# the analytic integral is simply 1 - exp(-1):
analytic_int = 1-math.exp(-1)
print('Analytic integral=', analytic_int)

# compare by calculating the fractional error:
print('Fractional error=', abs( (gauss_int-analytic_int)/analytic_int ) )
```

```
Integral of exp(-t) in [0,1] via N= 8 Gaussian quadrature 0.6321205588285576
Analytic integral= 0.6321205588285577
Fractional error= 1.7563469643870083e-16
```

We have reached extraordinary precision (essentially machine precision) with just 8 points!

We couldn't reach this precision with $N = 10^6$ points and the trapezoid rule, and we needed $N = 2001$ points to reach this preecision with Simpson's rule!

### 4.3.5 Higher Order Rules (3/8 and Milne Rules)

There exist other interesting "equal intrerval" rules that use higher-order approximations, such as the "3/8" rule and the Milne rule.

In terms of the formula: $\int_a^b f(x)\mathrm{d}x \simeq \sum_{i=1}^{N} f(x_i)w_i$, the weights for the 3/8 rule (third-degree approximation) are:

$w_i = (1, 3, 3, 1)\frac{3}{8}h.$

and for the Milne rule (fourth-degree approximation):

$w_i = (14, 64, 24, 64, 14)\frac{h}{45}.$

A useful check in these cases as well:

$\sum_{i=1}^{N} w_i = b - a.$

### 4.3.6 Integration Error Assessment

Let's now use the functions for the trapezoid rule, Simpson's rule, and N-th order Gaussian integration to evaluate the behavior of the relative error in each case, for the same function as before, $f(x) = \exp(-x)$, integrated over the interval $[0, 1]$, for which we know the analytic form.

As before, we will make a plot of the $\log_{10}$ of the relative error, versus $\log_{10}$ of the number of points used.

```
import math
import numpy as np
import scipy

Narray = np.concatenate([np.array([2,3,4,5,6,7,8]),np.logspace(1, 4, num=20,
 ↪base=10)]) # the NumPy logspace returns numbers spaced evenly on a log scale.

# test the function for exp(-t) in [0,1]:
# first define the function we wish to integrate:
def f(x):
    return math.exp(-x)

# Trapezoid rule result:
I_trapezoid = [trapezoid(f,0,1,N) for N in Narray]

# Simpson's rule result:
# reinstate after we have written the function:
#I_simpson = [simpson(f,0,1,N) for N in Narray]

# Gaussian Quadrature result:
I_gaussian = [gauss(f,0,1,N) for N in Narray]
```

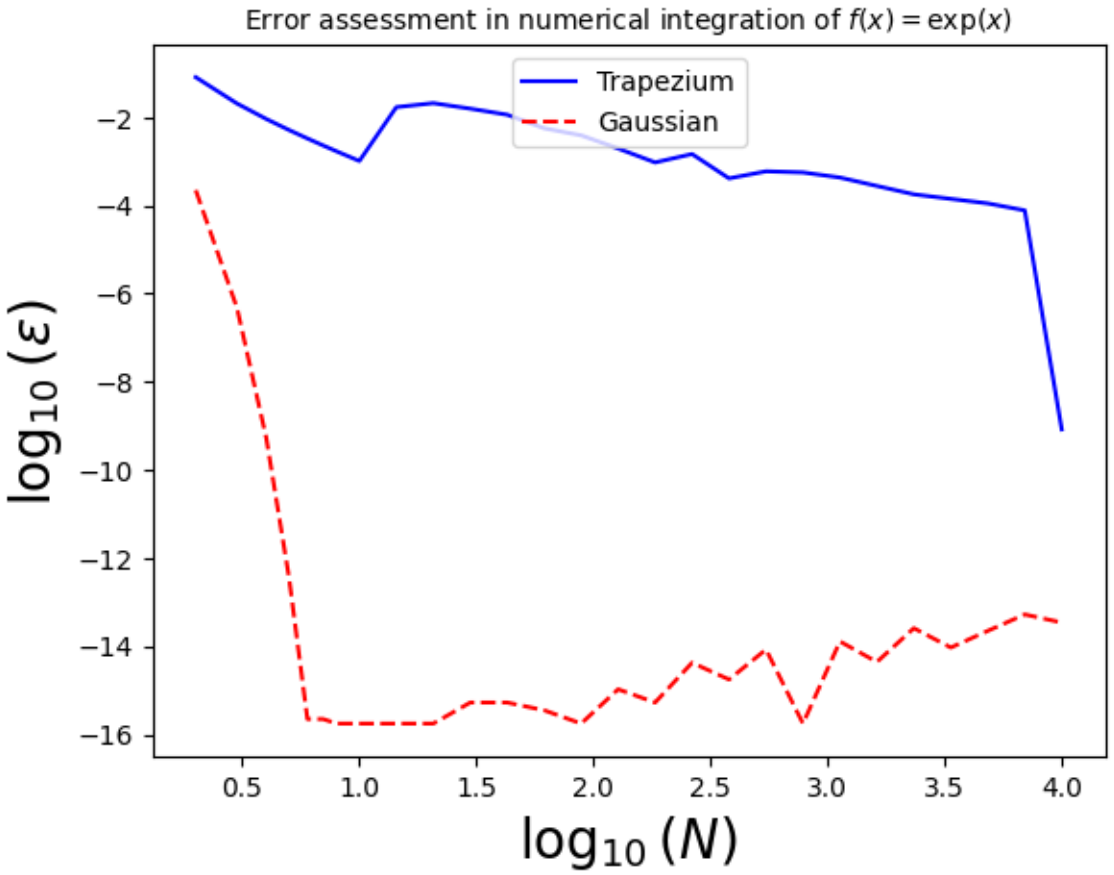
```
# Now let's get the relative errors:
I_analytic = 1-math.exp(-1) # use the analytic result

# get the relative errors:
# trapezoid:
eps_trapezoid = np.abs( np.subtract(I_trapezoid,I_analytic)/I_analytic )
# simpson: (uncomment)
# eps_simpson = np.abs( np.subtract(I_simpson,I_analytic)/I_analytic )

eps_gaussian = np.abs( np.subtract(I_gaussian,I_analytic)/I_analytic )
# if a zero has been found, set it to machine precision:
```

(continues on next page)

(continued from previous page)

```
eps_gaussian[eps_gaussian[:] == 0] = np.finfo(np.float64).eps

# get the log10:
log10_eps_trapezoid = np.log10(eps_trapezoid)
# log10_eps_simpson = np.log10(eps_simpson) # UNCOMMENT!
log10_eps_gaussian = np.log10(eps_gaussian)

# and of the array:
log10_N = np.log10(Narray)

# Now let's plot them!
```

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

# Now plot! Don't forget the different labels!
fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$\log_{10}(N)$', fontsize=20) # set the x label
ax.set_ylabel(r'$\log_{10}(\varepsilon)$', fontsize=20) # set the y label. Note that␣
 ↪the 'r' is necessary to remove the need for double slashes. You can use LaTeX!
ax.set_title(r'Error assessment in numerical integration of $f(x)=\exp(x)$',␣
 ↪fontsize=10) # set the title

# make a one-dimensional plot using the arrays relevant to the cosine:
ax.plot(log10_N, log10_eps_trapezoid, label='Trapezoid', ls='-', color='blue')
#ax.plot(log10_N, log10_eps_simpson, label='Simpson', ls='-', color='green')
ax.plot(log10_N, log10_eps_gaussian, label=r'Gaussian', ls='--', color='red')

# construct the legend:
ax.legend(loc='upper center')  # Add a legend

plt.show() # show the plot here
```




## 4.4 Minor Digression: Python functools and partial functions

The `functools` module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

See https://docs.python.org/3/library/functools.html for details.

Here, I wish to discuss `partial` functions.

Suppose we have a function that takes as arguments a variable and two parameters, e.g.:

```
def myfunction(x, a, b):
    """My function"""
    return a * x + b
```

Suppose we want to do something with this function, e.g. integrate it in the interval $[0, 1]$ for various values of `a` and `b`. We can create a "partial" function and then use it as an argument in our higher-order functions (e.g. `gauss()`) for integration:

```
from functools import partial # partial functions allow us to fix a certain number of␣
 ↪arguments of a function and generate a new function.

a = [0.1, 0.2, 0.3]
b = [0.4, 0.5, 0.6]
```

(continues on next page)

(continued from previous page)

```
for ai in a:
    for bi in b:
        partial_func = partial(myfunction, a=ai, b=bi) # this has created a new␣
↪function with the parameters a and b fixed
        # now we can pass this to our integrator!
        print('Integral of a*x+b in [0,1], for a=',ai, 'b=',bi,'=',gauss(partial_func,
↪0,1,4))
```

```
Integral of a*x+b in [0,1], for a= 0.1 b= 0.4 = 0.45000000000000007
Integral of a*x+b in [0,1], for a= 0.1 b= 0.5 = 0.55
Integral of a*x+b in [0,1], for a= 0.1 b= 0.6 = 0.65
Integral of a*x+b in [0,1], for a= 0.2 b= 0.4 = 0.5
Integral of a*x+b in [0,1], for a= 0.2 b= 0.5 = 0.6000000000000001
Integral of a*x+b in [0,1], for a= 0.2 b= 0.6 = 0.7000000000000001
Integral of a*x+b in [0,1], for a= 0.3 b= 0.4 = 0.55
Integral of a*x+b in [0,1], for a= 0.3 b= 0.5 = 0.6499999999999999
Integral of a*x+b in [0,1], for a= 0.3 b= 0.6 = 0.75
```

**Chapter 5: Monte Carlo Methods.**

CHAPTER

FIVE

# MONTE CARLO METHODS

## 5.1 Introduction

In physics we are often interested in systems with large degrees of freedom, e.g.:

- many atoms in a chunk of matter,
- many electrons in an atom,
- the infinitely-many values of a quantum field at all points in a region of space-time.

The description of such systems often involves the evaluation of integrals of very high dimension.

For example, imagine that we need to integrate atomic wave functions over the three coordinates of each of the 12 electrons in a magnesium atom. This amounts to 3x12=36 dimensions!

If we use 64 points for each integration, this requires about $64^{36} \simeq 10^{65}$ evaluations of the integrand. If you had a very fast computer, capable of $10^6$ evaluations per second, this would take $10^{59}$ s, which is significantly longer than the age of the universe! ($\sim 10^{17}$ s).

Direct quadrature is hopeless, if you still want to be alive when your integration is done!

Monte Carlo methods provide a way to efficiently evaluating integrals of high dimension.

"Monte Carlo": arises from the random or "chance" character of the method, through the famous casino in Monaco.

**Basic idea**: evaluate the integrand at a representative random sample of points. Analogous to predicting the results of an election on the basis of a poll of a small number of voters.

It turns out that the Monte Carlo strategy is very appropriate for a broad class of problems in statistical and quantum mechanics.

## 5.2 The Basic Monte Carlo Strategy

Even though the real power of Monte Carlo methods (henceforth: MC) is in evaluating multi-dimensional integrals, it is easiest to illustrate the basic ideas in a one-dimensional situation.

Suppose we have to evaluate the integral:

$I = \int_0^1 f(x)\mathrm{d}x.$

In Chapter 4, we discussed several quadrature formulas that used values of $f$ at particular values of $x$, e.g. equally-spaced for the trapezium and Simpson's rule.

An alternative way of evaluating the integral is to think of it as the average of $f$ over the interval $[0, 1]$:

$I \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i).$

Here, the average of $f$ is evaluated by considering its values at $N$ points $x_i$, chosen at random, with equal probability anywhere within the interval $[0, 1]$ (i.e. uniformly).

To estimate the uncertainty associated with this formula, $\sigma_I$, consider $f_i \equiv f(x_i)$ as a random variable, and invoke the central limit theorem for large $N$. From the usual laws of statistics, we then have:

$$\sigma_I^2 \approx \frac{1}{N}\sigma_f^2 = \frac{1}{N}\left[\frac{1}{N}\sum_{i=1}^{N} f_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f_i\right)^2\right]$$

where $\sigma_f^2$ is the "variance" in $f$, i.e. a measure of the extent to which $f$ deviates from its average value over the region of integration.

Some important aspects of Monte Carlo integration are revealed by the above integration formula:

- The uncertainty in the estimate of the integral, $\sigma_I$, decreases as $1/\sqrt{N}$. If more points are used, we will get a more precise answer, although the error decreases very slowly with the number of points: you need to do four times more numerical work to halve the error in your answer!
- The trapezium rule error scales as $1/N^2$, therefore having a much greater accuracy for a given amount of numerical work. This advantage vasnishes in the multi-dimensional case, as we will see shortly.
- The precision is greater if $\sigma_f$ is smaller: i.e. if $f$ is as smooth as possible. One limit to consider is that when $f$ is constant, in which case we need its value only at one point to define its average. In the other limit, consider a situation in which $f$ is zero everywhere, except for a very narrow peak above some value of $x$. Then, if we pick $x_i$s with an equal probability in $[0, 1]$, it is probable that all but a few of them will lie outside the peak of $f$, and this will lead to a poor estimate of $I$.

The method is extended to any interval $[a, b]$, by choosing random, uniform $x_i$s in $[a, b]$ and:

$$I = \int_a^b f(x)\mathrm{d}x = \frac{b-a}{N}\sum_{i=1}^{N} f(x_i).$$

Let's begin our numerical Monte Carlo work by evaluating the integral:

$$\int_0^1 \frac{\mathrm{d}x}{1+x^2} = \frac{\pi}{4}.$$

Let's also calculate the error using the formula for $\sigma_I$.

### 5.2.1 Example 5.1: Use the Monte Carlo integration method to calculate the integral \int_0^1 \frac{ \mathrm{d} x } { 1 + x^2 } = \frac{\pi}{4}. Also calculate the error.

```
import math
import random # we need random numbers for the Monte Carlo method!

random.seed(1234)

# Let's define a function that performs one-dimensional MC integration of an␣
 ↪arbitrary for N points
# We'll also make it capable of performing the integration in an interval a,b
def mcint(func, a, b, N):
    sumi = 0
    sumisq = 0
    for i in range(N):
        xi = (b-a) * random.random() + a
        sumi = sumi + func(xi)
        sumisq = sumisq + func(xi)**2
    I = sumi * (b-a) / N
    sumisq = sumisq*(b-a)**2
    # sumisq *= (b-a)**2
    sigmaI = math.sqrt( (1/N) * ( (1/N) * sumisq - I**2 ) )
```

(continues on next page)

(continued from previous page)

```
    return I, sigmaI # return the integral and its error

def fEX51(x):
    return 1/(1+x**2)

Iex51, err_ex51 = mcint(fEX51,0,1,100)
print('The estimate of the integral is=', Iex51, '+-', err_ex51)
print("Let's compare to the analytical value=", math.pi/4)
```

```
The estimate of the integral is= 0.8066530303708649 +- 0.016105612466499625
Let's compare to the analytical value= 0.7853981633974483
```

### 5.2.2 Example 5.2: For the function of example 5.1, calculate the error as a function of the number of points N, from N=10 to N=10^4. Plot the the logarithm of the error versus the log of the number of points. Does it agree with our expectations?

```
# The array of the number of points to integrate over:
Narray = [10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000]

Errs = []
for Ni in Narray:
    I, err = mcint(fEX51,0,1,Ni)
    Errs.append(err/I)

# Let's plot:
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single set of axes.

# get the logs:
x = np.log10(Narray)
y = np.log10(Errs)
yexpect = np.log10(1/np.sqrt(Narray))

# plot:
ax.plot(x, y) # make a one-dimensional plot using the above arrays
ax.plot(x, yexpect, label='theoretical') # make a one-dimensional plot using the␣
 ↪above arrays

# labels:
ax.set_xlabel(r'$\log_{10}(N)$')
ax.set_ylabel(r'$\log_{10}(\epsilon)$')
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[2], line 10
      7     Errs.append(err/I)
      9 # Let's plot:
```

(continues on next page)

(continued from previous page)

```
---> 10 import matplotlib.pyplot as plt # import matplotlib, a conventional module␣
 ↪name is plt
     11 import numpy as np
     13 fig, ax = plt.subplots() # create the elements required for matplotlib.␣
 ↪This creates a figure containing a single set of axes.

ModuleNotFoundError: No module named 'matplotlib'
```

### 5.2.3 Variance Reduction

The uncertainty (squared) in a MC quadrature is proportional to the variance of the integrand:

$\sigma_I^2 \approx \frac{1}{N}\sigma_f^2$,

with the variance of the integrand:

$\sigma_f^2 = \left[\frac{1}{N}\sum_{i=1}^{N} f_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f_i\right)^2\right]$.

Therefore, if we find a way of reducing the variance of the integrand, we will improve the efficiency of the method. One method is known as "importance sampling".

Let's imagine multiplying and dividing the integrand by a positive weight function $w(x)$, normalized such that:

$\int_0^1 w(x)\mathrm{d}x = 1$.

The integral can then be written as:

$I = \int_0^1 w(x)\frac{f(x)}{w(x)}\mathrm{d}x$.

We can then change variables from $x$ to:

$y(x) = \int_0^x w(x')\mathrm{d}x'$.

This implies that:

$\frac{\mathrm{d}y}{\mathrm{d}x} = w(x)$ and $y(x=0) = 0$, $y(x=1) = 1$.

and after the change of variables, the integral becomes:

$I = \int_0^1 \frac{f(x(y))}{w(x(y))}\mathrm{d}y$.

The MC integration proceeds as usual, i.e. averaging the values of $f/w$ using random and uniform points $y$ over the interval $[0, 1]$:

$I \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x(y_i))}{w(x(y_i))}$.

If we choose a $w$ function that behaves approximately as $f$, i.e. it is large where $f$ is large and small where $f$ is small, then the integrand $f/w$ can be made very smooth. This will result in a reduction in the variance of the MC estimate.

This implies of course that we are able to find an appropriate $w$ and that we are able to invert $y(x) = \int_0^x w(x')\mathrm{d}x'$ to find $x(y)$.

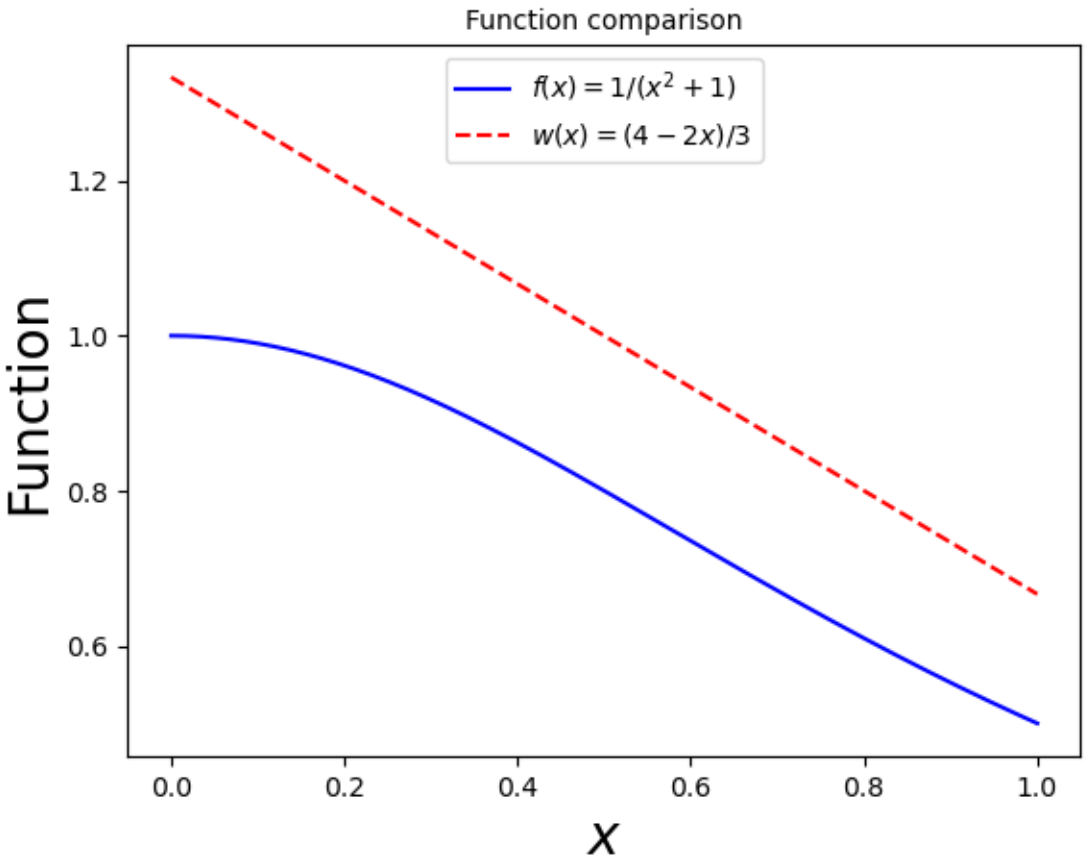
As an example, let's consider the same integral, $I = \int_0^1 \frac{\mathrm{d}x}{1+x^2}$. A good choice for a weight function is:

$w(x) = \frac{1}{3}(4 - 2x)$.

This is positive definite, decreases monotonically and is normalized correctly: $\int_0^1 w(x)\mathrm{d}x = 1$. It also approximates well the behavior of $f$:

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

# Now plot! Don't forget the different labels!
fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'Function', fontsize=20) # set the y label. Note that the 'r' is␣
 ↪necessary to remove the need for double slashes. You can use LaTeX!
ax.set_title(r'Function comparison', fontsize=10) # set the title

# construct the x variable:
x = np.linspace(0,1,1000)

# make a one-dimensional plot:
ax.plot(x, 1/(1+np.power(x,2)), label=r'$f(x) = 1/(x^2 + 1)$', ls='-', color='blue')

# check if this agrees with a line with slope -1/2 (on the log-log plot):
ax.plot(x, (4 - 2 * x)/3, label=r'$w(x) = (4-2x)/3$', ls='--', color='red')

# construct the legend:
ax.legend(loc='upper center')  # Add a legend

plt.show() # show the plot here
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)␣
 ↪SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel␣
 ↪Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)␣
 ↪instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)␣
 ↪SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel␣
 ↪Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)␣
 ↪instructions.
```




The new integration variable is:

$y = \int_0^x w(x')\mathrm{d}x' = \frac{1}{3}x(4-x)$.

And this can be inverted to give:

$x = 2 - (4 - 3y)^{1/2}$.

Let's see this change of variables in action!

### 5.2.4 Example 5.3: Use the Monte Carlo integration method with the variance reduction to calculate the integral \int_0^1 \frac{ \mathrm{d} x } { 1 + x^2 } over the same number of points as in Example 5.2. Make a plot comparing to the error calculated in Example 5.2.

```
# Start by modifying the mcint function to accept the x(y) and w(x) functions as input
import math
import random # we need random numbers for the Monte Carlo method!
```

```
The MC integral of f(x) = 1/(1+x^2) in [0,1] for N=1E4 points is= 0.
↪784742361698107 +- 0.0015987488463171692
The MC integral with variance reduction 0.7855334063802001 +- 0.0001698120319239752
The analytic value of the integral is= 0.7853981633974483
```

## 5.3 Multi-Dimensional Monte Carlo Integration

The one-dimensional discussion above can be readily generalized to $d$-dimensional integrals of the form:

$I = \int \mathrm{d}^d x f(\vec{x})$.

MC integration proceeds as it did in the one-dimensional case:

$I \approx \frac{1}{N} \sum_{i=1}^{N} f(\vec{x}_i)$,

where the components of the vector $\vec{x}_i$ are chosen independently.

Let's use this method to calculate $\pi$ via:

$\pi = 4 \int_0^1 \mathrm{d}y \int_0^1 \mathrm{d}x \theta(1 - x^2 - y^2)$,

i.e. by comparing the area of a quadrant of the unit circle, to that of the unit square.

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

# Now plot! Don't forget the different labels!
fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$y$', fontsize=20) # set the y label. Note that the 'r' is necessary␣
 ↪to remove the need for double slashes. You can use LaTeX!
ax.set_title(r'Area of quadrant of cirlce vs. that of square of side 1', fontsize=10)
 ↪# set the title

# plot the circle
circle1 = plt.Circle((0, 0), 1, color='r', alpha=0.2, ec='r', lw=5)
ax.add_patch(circle1)

# set the aspect ratio of the axes to 1:
ax.set_aspect(1)

plt.show() # show the plot here
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)␣
 ↪SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel␣
 ↪Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)␣
 ↪instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)␣
 ↪SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel␣
 ↪Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)␣
 ↪instructions.
```


### 5.3.1 Example 5.4: Use multi-dimensional Monte Carlo integration to calculate π!

We can also use variance reduction in many dimensions. For a weight function $w(\vec{x})$ normalized so that its integral over the region is unity, the new variable is a vector $\vec{y}$, where the *Jacobian* is:

$\left|\frac{\partial \vec{y}}{\partial \vec{x}}\right| = w(\vec{x})$.

It's generally very difficult to construct $\vec{x}(\vec{y})$ explicitly. We will discuss methods to do this below.

Going back to our MC integration results, although the results were satisfactory in our examples, MC integration does not appear to be particularly efficient, especially in comparison to the other quadrature methods that we previously discussed.

However, consider evaluating a multi-dimensional integral, such as the one used for the calculation of $\pi$.

Suppose that we are willing to invest a given amount of numerical work, e.g. to evaluate the integrand exactly $N$ times, and we wish to compare the efficiencies of various quadrature methods.

For the trapezoid rule, for example, if there are a total of $N$ points, then each dimension of a $d$-dimensional integral is broken up into $\sim N^{1/d}$ intervals, of spacing $h \sim N^{-1/d}$. It turns out that the total error would be:

$\epsilon_{\text{trapezoid}} \sim \mathcal{O}(1/N^{2/d})$.

For large $d$, this decreases very slowly with increasing $N$. For example, for $d = 4$, it decreases as $\sim \mathcal{O}(1/N^{1/2})$.

On the other hand, the uncertainty of Monte Carlo quadrature always decreases as $\sim 1/\sqrt{N}$, *independent* of the number of dimensions $d$!

Therefore, we would expect that the MC method is more efficient than the trapezoid method when $d \simeq 4$.

## 5.4 von Neumann Rejection Sampling ("Hit-or-Miss")

During our integration of $I = \int_0^1 \frac{1}{1+x^2} \mathrm{d}x$, if we had wanted to use the weight function:

$w(x) = \frac{6}{5}(1 - \frac{1}{2}x^2)$,

then we would have been faced with solving a cubic equation to find $x(y)$. While this is certainly possible, choices of the weight that might follow $f$ more generally will lead to more complicated functions that cannot be inverted analytically.

However, it is possible to do this numerically. A convenient method for generating one- or multi-dimensional random variables is the von Neumann rejection:

Suppose we are interested in generating $x$ in $[0, 1]$ with distribution $w(x)$, and that $w'(x)$ is a positive function such that $w'(x) > w(x)$ over the region of integration (Note that this is not the derivative). A convenient choice (but not always useful) for $w'$ is any constant greater than (or equal to) the maximum value of $w$ in the region of integration.

If we generate points in two dimensions that uniformly fill the area under the curve $w'(x)$ and then "accept" for use only those points that are under $w(x)$, then the accepted points will be distributed according to $w(x)$.

Practically:

- Choose $x_i$ distributed proportional to $w'$,
- Choose a random number $\eta$ uniformly in $[0, 1]$.
- $x_i$ is then accepted if $\eta < w(x_i)/w'(x_i)$.
- if a point is rejected, we simply go on and try another $x_i$.

This technique is clearly efficient only if $w'$ is close to $w$ throughout the entire range of integration, otherwise too much time is wasted rejecting "useless" points.

### 5.4.1 Example 5.5: Use von Neumann rejection to sample points in the interval [0,1], distributed as w(x) = \frac{6}{5} ( 1 - \frac{1}{2} x^2). Plot a histogram of the points and verify that they are indeed distributed according to w(x). Evaluate the integral I = \int_0^1 \frac{1}{1+x^2} \mathrm{d} x and its uncertainty with these points.

### 5.4.2 An Introduction to Monte Carlo Simulations

A Monte Carlo simulation contains the following ingredients:

- A probability density function that characterizes the system that we wish to simulate,
- Random numbers,
- A sampling rule,
- An error estimation.

The von Neumann rejection method of sampling provides a simple way to generate "simulations", i.e. samples of "events" that are distributed according to a certain probability density function, and look "realistic".

Let's examine an example in one dimension!

### 5.4.3 Example 5.5: Due to the uncertainty principle, the mass, m of an unstable particle (which can be measured through the momenta of its decay products), is distributed according to the Breit-Wigner distribution (in natural units):

$f(m) = \frac{k}{(m^2-M^2)^2+\Gamma^2M^2}$,

where $k$ is a normalization constant, $M$ is the particle's "nominal" mass and, $\Gamma = 1/\tau$ is its width, defined as the inverse of its lifetime, $\tau$.

Our goal is to generate $N$ decays of the particle in its rest frame, distributed according to the Breit-Wigner distribution.

We can assume that the particle is a top quark, with nominal mass $M = 172.7$ GeV and width $\Gamma = 1.4$ GeV.

We can use the von Neumann rejection method. To do so, we need to find a constant which is larger than $f(m)$ everywhere. We can start by choosing $f_{\max} = \frac{k}{\Gamma^2 M^2}$.

```
import random
import numpy as np
import math

# first let's define our distribution:
def fBW(m,M,Gam):
    """Calculate the Breit-Wigner distribution"""
    k = 1 # set the normalization constant to 1
    return k / (( m**2 - M**2)**2 + Gam**2 * M**2 )

# and perform the von Neumann rejection:
N = 1000 # we will generate N decay "events"
n = 0 # n will be the counter of generated "events"
# set the properties of the particle:
M = 172.7
Gam = 14
# let's also set a minimum and maximum mass (in GeV)
mmax = 200
mmin = 150
# and save the generated masses in an array:
masses = []
while n < N:
    m = random.random()*(mmax-mmin) + mmin
    r = random.random()
    if r < fBW(m,M,Gam) / (1/Gam**2/M**2):
        n = n + 1
        masses.append(m)
    else:
        pass
```

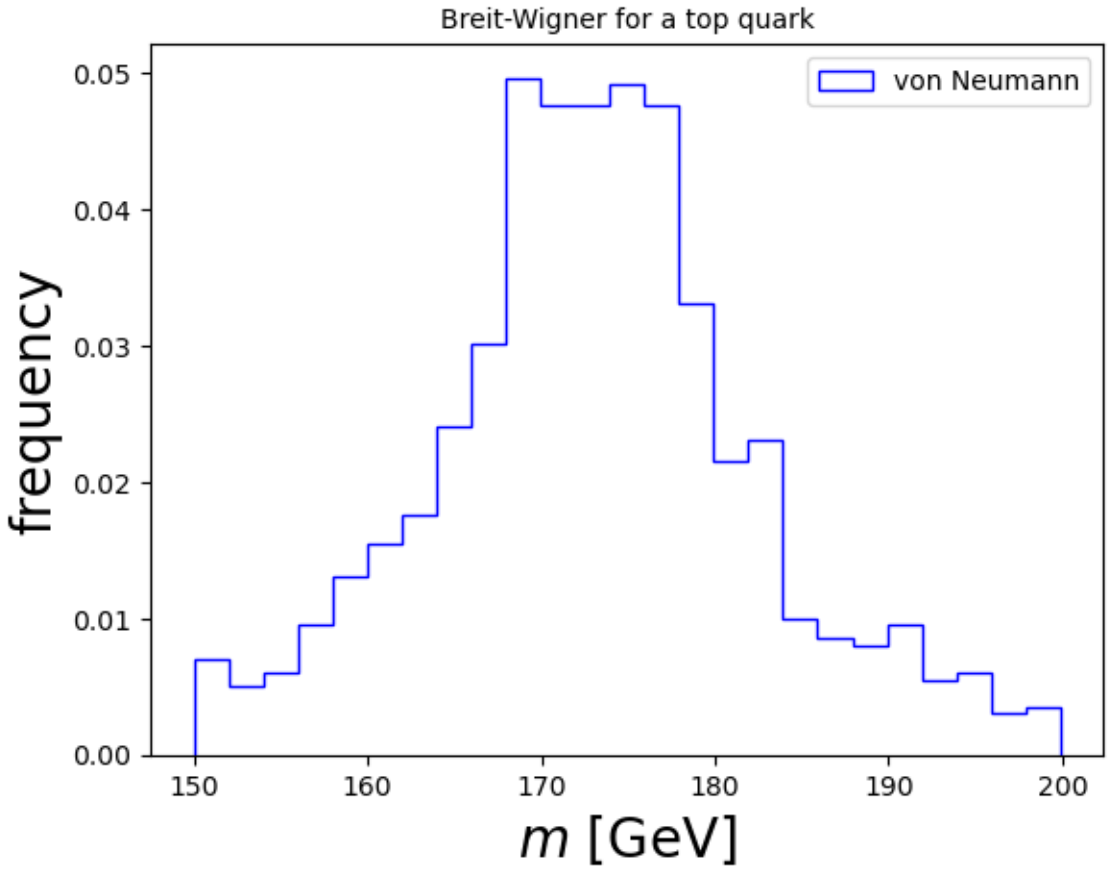
```
# Let's plot a histogram of the masses!
```

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$m$ [GeV]', fontsize=20) # set the x label
ax.set_ylabel(r'frequency', fontsize=20) # set the y label
```

(continues on next page)

(continued from previous page)

```
ax.set_title('Breit-Wigner for a top quark', fontsize=10) # set the title

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
 ↪and colors:
ax.hist(masses, color='blue', bins=25, density=True, fill=False, histtype='step',␣
 ↪label="von Neumann")

# construct the legend:
ax.legend(loc='upper right')  # Add a legend

plt.show() # show the plot here
```




In this case, we can perform a change of variables to obtain an efficiency of 100%:

$m^2 = M\Gamma \tan\eta + M^2$.

and with this:

$\frac{\mathrm{d}m^2}{\mathrm{d}\eta} = M\Gamma \sec^2\eta$, which turns the Breit-Wigner distribution flat, e.g., under the integral:

$I = \int_{m_{\min}}^{m_{\max}} \mathrm{d}m^2 \frac{k}{(m^2-M^2)^2+\Gamma^2M^2}$,

which turns into:

$I = \int_{m_{\min}}^{m_{\max}} \mathrm{d}\eta \left|\frac{\mathrm{d}m^2}{\mathrm{d}\eta}\right| \frac{k}{(m^2-M^2)^2+\Gamma^2M^2} = \int_{\eta_{\min}}^{\eta_{\max}} \frac{k}{\Gamma M}\mathrm{d}\rho$.

### 5.4.4 Example 5.6: Generate events distributed according to a Gaussian distribution:

$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-(x-\mu)^2/(2\sigma^2))$

Plot the results in a histogram.

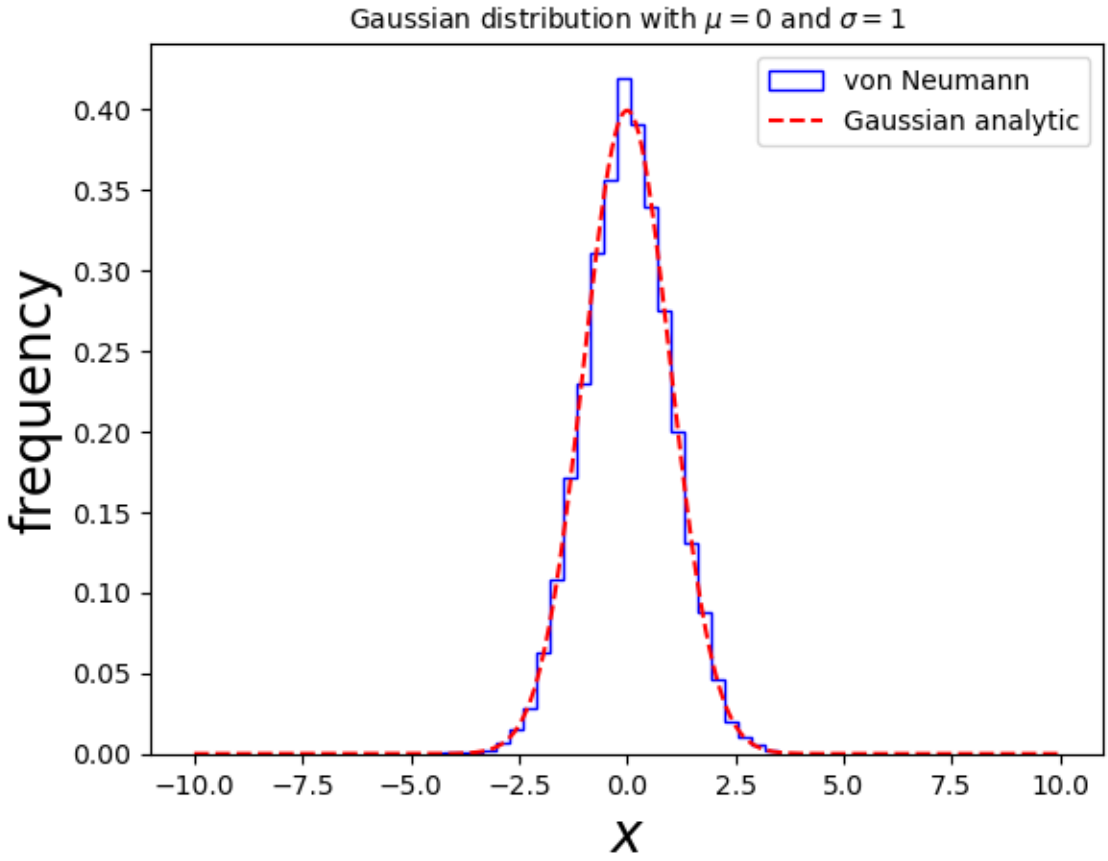
```
import random
import numpy as np
import math

# first let's define our distribution:
def Gaussian(x,mu,sigma):
    """Calculate the Gaussan distribution"""
    return 1/sigma/np.sqrt(2 * math.pi) * np.exp( - (x - mu)**2 / (2 * sigma**2))

# and perform the von Neumann rejection:
N = 10000 # we will generate N poits
n = 0 # n will be the counter of generated "events"
# set the properties of the particle:
mu = 0
sigma = 1
# let's also set a minimum and maximum mass (in GeV)
xmin = -10
xmax = 10
# and save the generated masses in an array:
x = []
while n < N:
    xi = random.random()*(xmax-xmin) + xmin
    r = random.random()
    if r < Gaussian(xi,mu,sigma)/ Gaussian(mu,mu,sigma):
        n = n + 1
        x.append(xi)
    else:
        pass
```

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'frequency', fontsize=20) # set the y label
ax.set_title(r'Gaussian distribution with $\mu=' + str(mu) + r'$ and $\sigma=' +␣
 ↪str(sigma) + '$', fontsize=10) # set the title

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
 ↪and colors:
ax.hist(x, color='blue', bins=25, density=True, fill=False, histtype='step', label=
 ↪"von Neumann")

# plot the distribution itself:
xv = np.linspace(xmin, xmax,1000)
gaussianpoints = Gaussian(xv,mu,sigma)
```

(continues on next page)

(continued from previous page)

```
ax.plot(xv, gaussianpoints, ls='--', color='red', label='Gaussian analytic')

# construct the legend:
ax.legend(loc='upper right')  # Add a legend

plt.show() # show the plot here
```



CHAPTER

SIX

# MATRIX COMPUTING, TRIAL-AND-ERROR SEARCHING AND DATA FITTING

## 6.1 Trial-and-Error Root Finding

### 6.1.1 Introduction

Trial-and-error may not sound very precise, but it is in fact used to solve problems where analytic solutions do not exist, or are not practical.

In trial-and-error algorithms, decisions on what path to follow are made based on the current values of variables, and the program quits only when it thinks it has solved the problem.

Trial-and-error root finding looks for a value of $x$ for which:

$f(x) \simeq 0$.

To put the zero on the right-hand side is conventional, since any equation can be written in this form, e.g.

$10\sin(x) = 3x^3 \Rightarrow 10\sin(x) - 3x^3 = 0$.

Generally, the procedure starts with a *guessed* value for $x$, substitutes that guess into $f(x)$ (the "trial"), and then sees how far the left-hand side is from zero.

The program then changes $x$ based on the error, and tries out a new guess in $f(x)$.

The procedure continues until $f(x) \simeq 0$ to some desired level of precision, or until the changes in $x$ are insignificant, or if the search seems endless.

### 6.1.2 Trial-and-error Roots via Bisection

The most elementary example of trial-and-error root finding is the *bisection algorithm*. It is reliable, but slow.

If you know some interval in which $f(x)$ changes sign, then the bisection algorithm will always converge to the root by finding progressively smaller and smaller intervals within which the zero lies.

The bisection algorithm can be described as follows:

- Start with two values of $x$ between which we know a zero occurs. For example, you may determine these by making a graph, or by stepping through different values of $x$ and looking for a sign change.
- Let's then say that $f(x)$ changes sign from $x_1$ to $x_2$, e.g. if $f(x_1) < 0$ then $f(x_2) > 0$. The bisection algorithm chooses a new $x$ as the bisection of the interval (i.e. the mid point!), and selects as its new interval the half in which the sign change occurs.
- The process continues until the value of $f(x)$ is less than a predefined level of precision, or until a predefined (large) number of subdivisions occurs.

Let's implement the bisection algorith and use it in a concrete example!

### 6.1.3 Example 6.1: Temperature Dependence of Magnetization.

The magnetization of $N$ spin-1/2 particles, each with magnetic moment $\mu$ at a temperature $T$, in an external magnetic field $B$ is given by:

$m(t) = \tanh\left(\frac{m(t)}{t}\right)$,

where $m(t)$ is the "reduced" magnetization and $t$ is the "reduced" temperature.

[For the sake of completeness: $m(T) = M(T)/(N\mu)$, with $M$ being the magnetization, $t = T/T_c$, with $T$ being the temperaturem, and $T_c = N\mu^2\lambda/k_B$ the Curie temperature.]

Our goal is to find the reduced magnetization $m$, for various reduced temperatures: $t = 0.5, 1.0, 2.0$. We will use the bisection method to solve the *transcendetal* equation for $m(t)$.

We should start by plotting the function that we wish to find the zero of, i.e. $f(m) = \tanh(m/t) - m$, to get an indication of the location of the zeros. It will be clear that there are no zeros for some of the values. For those that there will be zeros, let's find the solution to a precision of $\mathcal{O}(10^{-10})$. Let's also print the number of iterations necessary to achieve this.

## 6.2 Newton-Raphson Searching

The Newton-Raphson algorithm finds approximate roots of equations of the same type, $f(x) = 0$, more quickly than the bisection method.

The algorithm is the equivalent of drawing a straight line $f(x) \simeq mx + b$, tangent to the curve at an $x$ value for which $f(x) \simeq 0$ and then using the intrercept of the line with the $x$-axis at $x = -b/m$ as an improved guess for the root.

If the curve was in fact a straight line, the answer would be exact. Otherwise it is a good approximation if the guess is close enough to the root for $f(x)$ to be nearly linear.

The process continues until some set level of precision is reached.

If a guess is in a region where $f(x)$ is nearly linear, then the convergence is much more rapid than for the bisection algorithm.

The formulation of the Newton-Raphson algorithm is as follows:

- Start with a guess $x_0$,
- Find the tangent line to $f(x)$ at $x_0$ via: $y = f(x_0) + (x - x_0)\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x=x_0}$.
- This line crosses the axis at: $f(x_0) + (x - x_0)\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x=x_0} = 0$.
- Solving for $x$: $x = x_0 - f(x_0)/\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x=x_0}$. This will be the new guess.
- The procedure is repeated until some level of precision is reached, or a maximum number of evaluations.

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

# Now plot! Don't forget the different labels!
fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
↪a figure containing a single axes.

# set the labels and titles:
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$f(x)$', fontsize=20) # set the y label. Note that the 'r' is␣
 ↪necessary to remove the need for double slashes. You can use LaTeX!
ax.set_title(r'The Newton-Raphson Method', fontsize=10) # set the title

# construct the variable:
x = np.linspace(0,3.5,350)
# and the function:
f = x**2 - 4

ax.plot(x, f, label='$f(x)$', ls='--', color='red')
x0 = 3.0
y0 = 3.0**2 - 4
x1 = x0 - (x0**2-4)/(2 * x0)
y1 = x1**2 - 4
x2 = x1 - (x1**2-4)/(2 * x1)
y2 = x2**2 - 4

ax.plot(x0, y0, markersize=5, marker='o', color='red')
ax.plot(x1, y1, markersize=5, marker='o', color='red')
ax.plot(x2, y2, markersize=5, marker='o', color='red')
ax.text(x0-0.1,y0+0.4,"$x_0$")
ax.text(x1-0.1,y1+0.4,"$x_1$")
ax.text(x2-0.1,y2+0.4,"$x_2$")

y = y0 + (x-x0) * 2*x0
y2 = y1 + (x-x1) * 2*x1

ax.plot(x,y,ls='--', color='black')
ax.plot(x,y2,ls='--', color='black', alpha=0.5)

# draw horizontal line passing through zero
ax.hlines(xmin=0, xmax=3.2,y=0,color='black')

# draw first vertical line to indicate x_1:
ax.vlines(ymin=0, ymax=y1,x=x1,color='black')

# set the limits:
ax.set_xlim(0,3.2)
ax.set_ylim(-6,6)

# construct the legend:
ax.legend(loc='upper center')  # Add a legend

ax.grid() # show the grid.

plt.show() # show the plot here
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 import matplotlib.pyplot as plt # import matplotlib, a conventional module␣
 ↪name is plt
      2 import numpy as np
      4 # Now plot! Don't forget the different labels!

ModuleNotFoundError: No module named 'matplotlib'
```

It's clear that the Newton-Raphson algorithm requires the evaluation of the derivative of the function at the guess point $x_0$, i.e. $\frac{\mathrm{d}f}{\mathrm{d}x}\Big|_{x=x_0}$.

In many cases you may have an analytic expression for the derivative, and it can be built into the algorithm.

However, it is simpler to use a numerical approximation to the derivative, e.g. using the forward-difference derivative:

$\frac{\mathrm{d}f}{\mathrm{d}x} \approx \frac{f(x+\delta x)-f(x)}{\delta x}$.

A central-difference derivative can also be used, but it would require additional evaluations of the function.

## 6.3 Newton-Raphson with Backtracking

Potential problems may arise while performing Newton-Raphson searching:

- The search can take us to an $x$ value where the function has a local minimum or maximum, i.e.: $\mathrm{d}f/\mathrm{d}x = 0$. This would lead to a horizontal tangent and then the next guess would be $x \to \infty$, which is hard to return from!
- The search can fall into an infinite loop surrounding the zero without ever getting there.

The problem in both cases is that the initial guess is not close enough to a region where the function is approximately linear. This is why a good plot may help produce a good first guess.

When the above occur, you may need to start your initial guess ($x_0$) at a different point in an effort not to fall in these traps, or try *backtracking*.

**Backtracking**: in cases where the new guess leads to an *increase* in the magnitude of the function, i.e. $|f(x_0 + \Delta x)|^2 > |f(x_0)|^2$, you can backtrack somewhat and try a smaller step, say $x_0 + \Delta x/2$. If the magnitude still increases, then backtrack some more, say by trying $x_0 + \Delta x/4$, and so on.

An alternative solution: start with a bisection search algorithm and switch to faster Newton-Raphson when you get closer to zero.

### 6.3.1 Example 6.2: Newton-Raphson applied to the Temperature Dependence of Magnetization.

Implement the Newton-Raphson search algorithm to find the root of the Transcendental equation given in Example 6.1 for $t = 0.5$ and a precision of $\mathcal{O}(10^{-10})$. Use a central-difference derivative while implementing the algorithm and print the number of iterations.

## 6.4 Matrix Computing

### 6.4.1 Why Matrix Computing?

Physical systems are often modeled by systems of simultaneous equations that can be written in matrix form.

As models are made more realistic, matrices become correspondingly bigger. It is therefore important to use a good linear algebra library.

"Industrial-strength" subroutines for matrix computing can be found in well-established scientific libraries. They can be an order of magnitude or more faster than elementary methods found in linear algebra texts.

These packages are usually designed to minimize the round-off error and they are robust, i.e. they have a high chance of being successful for a broad class of problems.

The question is, when is a matrix large enough to require the use of a library routine? A rule of thumb is: "when you have to wait for the answer!".

Note that GPUs (Graphics Processing Units) can perform matrix algebra very efficiently, since they can execute many commands in parallel. This requires programming in specific frameworks, but it's good to keep in mind at this point.

### 6.4.2 Classes of Matrix Problems

The most basic matrix problem: a system of linear equations, e.g.:

$\mathbf{A}\vec{x} = \vec{b}$,

where $\mathbf{A}$ is an $N \times N$ matrix, $\vec{x}$ is an *unknown* vector of length $N$, and $\vec{b}$ is a known vector of length $N$.

The obvious way to solve this: determine the inverse of $\mathbf{A}$, $\mathbf{A}^{-1}$, and multiply both sides of the equation by it to obtain $\vec{x}$:

$\vec{x} = \mathbf{A}^{-1}\vec{b}$.

Calculating the inverse of a matrix is a standard in a matrix subroutine library.

A more efficient way to solve the linear equation is by Gaussian elimination or lower-upper (LU) decomposition. Both of these yield $\vec{x}$ without $\mathbf{A}^{-1}$. (However, sometimes you may want the inverse of a matrix.)

Another form of matrix problems that is frequently encountered is:

$\mathbf{A}\vec{x} = \lambda\vec{x}$,

with an unknown vector $\vec{x}$ and an unknown parameter $\lambda$. This is an *eigenvalue problem*. Solutions only exist for certain (if any) values of $\lambda$. To find a solution, we can use the identity matrix, $\mathbf{I}$, to rewrite the equation as follows:

$[\mathbf{A} - \lambda\mathbf{I}]\vec{x} = 0$.

which yields a non-trivial solution only if the determinant of $[\mathbf{A} - \lambda\mathbf{I}]$ is zero, i.e.:

$\det[\mathbf{A} - \lambda\mathbf{I}] = 0$.

This is known as the "secular equation". The values of $\lambda$ that satisfy this equation are the eigenvalues of the eigenvalue equation.

The traditional way to solve the eigenvalue problem for both eigenvalues and eigenvectors is by *diagonalization* via a transformation matrix $\mathbf{U}$:

$\mathbf{UA}(\mathbf{U}^{-1}\mathbf{U})\vec{x} = \lambda\mathbf{U}\vec{x}$,

$(\mathbf{UAU}^{-1})\mathbf{U}\vec{x} = \lambda\mathbf{U}\vec{x}$,

where $\mathbf{UAU}^{-1} \equiv \mathbf{D}$ is a diagonal matrix:

$$\mathbf{D} = \mathbf{UAU}^{-1} = \begin{pmatrix} \lambda'_1 & 0 & 0 & ... & 0 \\ 0 & \lambda'_2 & 0 & ... & 0 \\ 0 & 0 & \lambda'_3 & ... & 0 \\ 0 & 0 & 0 & ... & \lambda'_N \end{pmatrix},$$

whose diagonal entries are the eigenvalues, corresponding to eigenvectors:

$\vec{x}_i = \mathbf{U}^{-1}\hat{\vec{e}}_i$,

i.e. the eigenvectors are the columns of the matrix $\mathbf{U}^{-1}$.

### 6.4.3 Math Recap: Matrix Multiplication, Inverses and Determinants

Matrix Multiplication:

To multiply two matrices **A** and **B**:

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix},$$

$$\mathbf{B} = \begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{pmatrix},$$

then:

$$\mathbf{AB} = \begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} & a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} & a_{00}b_{02} + a_{01}b_{12} + a_{02}b_{22} \\ a_{10}b_{00} + a_{11}b_{10} + a_{12}b_{20} & a_{10}b_{01} + a_{11}b_{11} + a_{12}b_{21} & a_{10}b_{02} + a_{11}b_{12} + a_{12}b_{22} \\ a_{20}b_{00} + a_{21}b_{10} + a_{22}b_{20} & a_{20}b_{01} + a_{21}b_{11} + a_{22}b_{21} & a_{20}b_{02} + a_{21}b_{12} + a_{22}b_{22} \end{pmatrix},$$

[See https://www.geogebra.org/m/ETHXK756 for an animation!]

Matrix Determinants:

The determinant of a matrix $\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$,

is given by:

$\det[A] = a_{00}(a_{11}a_{22} - a_{21}a_{12}) - a_{01}(a_{01}a_{22} - a_{20}a_{12}) - a_{02}(a_{10}a_{21} - a_{20}a_{11})$

Matrix Inversion (Cramer's rule):

To invert the matrix **A** as defined above we first need the determinant and the matrix of cofactors:

$$\mathbf{C} = \begin{pmatrix} a_{11}a_{22} - a_{21}a_{12} & a_{20}a_{12} - a_{10}a_{22} & a_{10}a_{21} - a_{20}a_{11} \\ a_{21}a_{02} - a_{01}a_{22} & a_{00}a_{22} - a_{20}a_{02} & a_{20}a_{01} - a_{00}a_{21} \\ a_{01}a_{12} - a_{11}a_{02} & a_{20}a_{02} - a_{00}a_{22} & a_{00}a_{11} - a_{10}a_{01} \end{pmatrix},$$

and then the inverse is given by:

$\mathbf{A}^{-1} = \frac{1}{\det[A]}\mathbf{C}^T$, where the superscript $T$ denotes a transposition, i.e. an exchange of rows and columns.

### 6.4.4 Example 6.3: In this problem we will find the inverse of a 3x3 matrix *analytically* and show via direct matrix multiplication that it indeed gives back the identity matrix.

(a) Find the determinant of the matrix:

$$\mathbf{A} = \begin{pmatrix} +4 & -2 & +1 \\ +3 & +6 & -4 \\ +2 & +1 & +8 \end{pmatrix}.$$

(b) Find the inverse of **A**.

(c) Check that the inverse indeed satisfies $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$.

### 6.4.5 Math Recap: Solving Eigenvalue Problems

To find the eigenvalues of a matrix we first subtract $\lambda$ from its diagonal, take the determinant and set it to zero. We then solve the resulting characteristic equation.

E.g. for a 2x2 matrix,

$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$,

the characteristic equation is:

$\det \begin{pmatrix} a_{00} - \lambda & a_{01} \\ a_{10} & a_{11} - \lambda \end{pmatrix} = 0$

or:

$(a_{00} - \lambda)(a_{11} - \lambda) - a_{01}a_{10} = 0$

To find the eigenvectors, i.e. the vectors that satisfy:

$\mathbf{A}\vec{x}_i = \lambda_i \vec{x}_i$, where $\lambda_i$ is the $i$-th eigenvalue, we can write:

$\vec{x}_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}$

and then operate with $\mathbf{A} - \lambda_i \mathbf{I}$

$\mathbf{A} - \lambda_i \mathbf{I} = \begin{pmatrix} a_{00} - \lambda_i & a_{01} \\ a_{10} & a_{11} - \lambda_i \end{pmatrix}$:

$\begin{pmatrix} a_{00} - \lambda_i & a_{01} \\ a_{10} & a_{11} - \lambda_i \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, to get:

$(a_{00} - \lambda_i)x_i + a_{01}y_i = 0$

and:

$a_{10}x_i + (a_{11} - \lambda_i)y_i = 0$,

allowing us to find relations between $x_i$ and $y_i$, i.e. the components of the $i$-th eigenvector. The eigenvectors can then be normalized to unity.

Let's find the eigenvectors and eigenvalues in a concrete example.

### 6.4.6 Example 6.4: Find the eigenvalues and eigenvectors of the matrix:

$\mathbf{A} = \begin{pmatrix} \alpha & \beta \\ -\beta & \alpha \end{pmatrix}$

### 6.4.7 Example 6.5: Given that \lambda_1 = 5 is an eigenvalue, find the remaining eigenvalues of the matrix:

$\mathbf{A} = \begin{pmatrix} -2 & +2 & -3 \\ +2 & +1 & -6 \\ -1 & -2 & 0 \end{pmatrix}$

We will tackle these, and other problems, with Python numerical packages shortly!

## 6.5 Practical Matrix Computing

Many scientific programming bugs arise from improper use of arrays.

These may be a result of the extensive use of matrices, or to the complexity of keeping track of indices and dimensions. Here are some rules of thumb to observe!

- **Computers are finite**: Be careful with matrix memory usage! E.g. if you are storing data in a 4-dimensional array, with each index having a physical dimension of 100, e.g. `A[100][100][100][100]`, then this array will take up $(100)^4$ 64-byte words $\simeq$ 1 GB of memory! Note that modern computers typically have 8-64 GB of memory (RAM).
- **Processing time**: Matrix operations such as inversion require on the order of $N^3$ steps for a square matrix of dimension $N$ (number of rows $\times$ number of columns). E.g. doubling the dimensions of a matrix results in an eightfold increase in the processing time.
- **Matrix Storage**: We think of matrices as multi-dimensional blocks of stored numbers, but the computer stores them as linear strings. For instance, a matrix `a[3,3]` in Python is stored in a row-major order:

```
a00 a01 a02 a10 a11 a12 a20 a21 a22
```

while in Fortran it is stored in a column-major order:

```
a01 a10 a20 a01 a11 a21 a02 a12 a22
```

It is important to keep this in mind in order to write proper code which permits the mixing of Python and Fortran programs!

- **Tests** Always test a library routine on a small problem whose answer you already know. Then you will know if you are using the routine correctly.

## 6.6 Matrices in Python: Python Lists, NumPy Arrays

### 6.6.1 Python Lists

Lists contain sequences of objects that are *mutable* (i.e. changeable) .

Python also has a builtin type called a *tuple*. Its elements are *not* mutable. (See Chapter 1).

Most programming languages require you to specify the size of an array before you can start storing objects in it. On the other hand, Python lists are dynamic, i.e. their sizes are adjusted as needed.

Compound lists can be created in Python by having the individual elements themselves as lists.

```
# Lists are mutable:
VectorA = [1,2,3]
VectorA[2] = 4 # we are changing the last element from 3 -> 4
print(VectorA)
```

```
[1, 2, 4]
```

```
# On the other hand, tuples are not:
TupleB = (1,2,3)
TupleB[2] = 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 3
      1 # On the other hand, tuples are not:
      2 TupleB = (1,2,3)
----> 3 TupleB[2] = 2

TypeError: 'tuple' object does not support item assignment
```

### 6.6.2 NumPy Arrays

Python lists are rather limited. Instead it is recommended to use NumPy arrays for actual mathematical manipulations involving matrices.

NumPy Arrays:

- Can hold up to 32 dimensions!
- But each element must be of the same type, i.e. the array has to be "uniform".
- Elements are not resticted to be floats or integers, but can be any ojbect, as long as the elements are of the same type.
- The data types can be compound.

```
# Let's construct NumPy arrays
import numpy as np

# a NumPy array:
NumpyVectorA = np.array([1,2,3])
print('A=',NumpyVectorA)

# a compound NumPy array:
NumpyVectorB = np.array([['a', 1], ['b', 2], ['c', 3]])
print('B=',NumpyVectorB)
```

```
A= [1 2 3]
B= [['a' '1']
 ['b' '2']
 ['c' '3']]
```

```
# Multiply a matrix by a number multiplies each element by that number:
# e.g.:
NumpyVectorA2 = NumpyVectorA * 2
print('A*2=', NumpyVectorA2)
```

```
A*2= [2 4 6]
```

```
# .shape tells you the shape of an array
# e.g.:
NumpyMatrixC = np.array([[1,2], [3,4], [5,6]])
print('The shape of NumpyMatrixC is', NumpyMatrixC.shape)
# or:
NumpyMatrixD = np.array([[1,2,7], [3,4,8], [5,6,9]])
print('The shape of NumpyMatrixD is', NumpyMatrixD.shape)
```

(continues on next page)

(continued from previous page)

```
# and .ndim tells you the number of dimensions:
print('The number of dimensions for NumpyMatrixD is', NumpyMatrixD.ndim)
# add a dimension:
NumpyMatrixE = np.array([NumpyMatrixD, NumpyMatrixD])
print('The number of dimensions for NumpyMatrixE is', NumpyMatrixE.ndim)
```

```
The shape of NumpyMatrixC is (3, 2)
The shape of NumpyMatrixD is (3, 3)
The number of dimensions for NumpyMatrixD is 2
The number of dimensions for NumpyMatrixE is 3
```

```
# you can get the transpose by the .T operation:
# e.g.:
print('NumpyMatrixD=',NumpyMatrixD)
print('NumpyMatrixD transposed=',NumpyMatrixD.T)
```

```
NumpyMatrixD= [[1 2 7]
 [3 4 8]
 [5 6 9]]
NumpyMatrixD transposed= [[1 3 5]
 [2 4 6]
 [7 8 9]]
```

To obtain a matrix product from two arrays, you should use the dot function. The * operator between two arrays is used for an element-by-element product.

E.g.:

```
# Element-wise multiplication:
print(NumpyMatrixD * NumpyMatrixD)
```

```
[[ 1  4 49]
 [ 9 16 64]
 [25 36 81]]
```

```
# Matrix Multiplication:
print(np.dot(NumpyMatrixD,NumpyMatrixD))
```

```
[[ 42  52  86]
 [ 55  70 125]
 [ 68  88 164]]
```

A major power in NumPy comes from its *broadcasting* operation, an operation in which values are assigned to multiple elements via a single assignment statement.

Broadcasting permits Python to *vectorize* array operations, which means that the same operation can be performed on different array elements in parallel (or nearly so).

Broadcasting also speeds up processing because array operations occur in C instead of Python, and with a minimum of array copies being made.

You've already been using broadcasting, e.g. when you did the following:

```
x = np.linspace(0,2*np.pi,100)
y = np.sin(x) # broadcasting via NumPy is done here
```

## 6.7 NumPy's linalg Package

NumPy's linalg package treates 2D arrays (1D arrays of 1D arrays) as mathematical matrices, and provides an interface to the powerful LAPACK linear algebra library.

There is much to be gained in speed and reliability from using these libraries, rather than writing your own matrix routines.

As a first example, let's look at the standard matrix equation:

$\mathbf{A}\vec{x} = \vec{b}$.

```
import numpy as np
from numpy import linalg # import the linear algebra package

# define the array A:
A = np.array([ [1,2,3], [22, 32, 42], [55,66,100] ])
print('A=',A)

# define the vector b:
b = np.array([1,2,3])
print('b=',b)

# now solve using linalg's solve:
x = linalg.solve(A, b)
print('x=',x)
```

```
A= [[  1   2   3]
 [ 22  32  42]
 [ 55  66 100]]
b= [1 2 3]
x= [-1.4057971  -0.1884058   0.92753623]
```

We can print the residual of the operation $\mathbf{A}\vec{x} - \vec{b}$ to check how far we are from zero!

```
print('Ax-b=', np.dot(A,x) - b)
```

```
Ax-b= [ 0.00000000e+00 -2.66453526e-15 -5.32907052e-15]
```

Very close to zero indeed!

We have solved entire set of linear equations (by elimination) with just a single command, performed a matrix multiplication with the single command `dot`, did a matrix subtraction, and are left with a residual equal to machine precision!

Although there are more efficient numerical approaches, a direct way to solve the equation is to calculate the inverse of $\mathbf{A}$, and then multiply both sides of the equation by the inverse, yielding

$\vec{x} = \mathbf{A}^{-1}\vec{b}$.

Let's follow this approach as well!

```
import numpy as np
from numpy import linalg # import the linear algebra package

# define the array A:
A = np.array([ [1,2,3], [22, 32, 42], [55,66,100] ])
print('A=',A)

# define the vector b:
```

(continues on next page)

(continued from previous page)

```
b = np.array([1,2,3])
print('b=',b)

# get the inverse using the linear algebra package:
invA = linalg.inv(A)
print('A^{-1}=', invA)

# test the inverse:
print('A.A^{-1}=', np.dot(A,invA))

# solve the equation:
x = np.dot(invA,b)
print('x=',x)
```

```
A= [[  1   2   3]
 [ 22  32  42]
 [ 55  66 100]]
b= [1 2 3]
A^{-1}= [[-1.55072464  0.00724638  0.04347826]
 [-0.39855072  0.23550725 -0.08695652]
 [ 1.11594203 -0.15942029  0.04347826]]
A.A^{-1}= [[1.00000000e+00 2.77555756e-17 6.93889390e-18]
 [7.54951657e-15 1.00000000e+00 3.74700271e-16]
 [1.86517468e-14 5.55111512e-16 1.00000000e+00]]
x= [-1.4057971  -0.1884058   0.92753623]
```

Same as via the direct method, as expected!

The second type of matrix equation we wish to solve is the eigenvalue equation:

$\mathbf{I}\vec{\omega} = \lambda\vec{\omega}$,

where in this case, e.g. $\mathbf{I}$ is the inertia matrix (or tensor) and $\omega$ is an unknown eigenvector, $\lambda$ is the unknown eigenvalue.

Then, e.g.:

```
import numpy as np
from numpy import linalg

# the moment of inertia tensor:
I = np.array([[2/3, -1/4], [-1/4, 2/3]])
print('I=',I)

# get the eigenvectors and eigenvalues:
evalues, evectors = linalg.eig(I)
print('Eigenvalues=',evalues)
print('Eigenvectors=',evectors)
```

```
I= [[ 0.66666667 -0.25      ]
 [ 0.25        0.66666667]]
Eigenvalues= [0.66666667+0.25j 0.66666667-0.25j]
Eigenvectors= [[0.70710678+0.j         0.70710678-0.j        ]
 [0.         -0.70710678j 0.         +0.70710678j]]
```

### 6.7.1 Example 6.6: Solve Examples 6.3 to 6.5 numerically!

```
import numpy as np
from numpy import linalg

# Example 6.3:
A = np.array([ [+4, -2, +1], [+3, 6, -4], [+2, +1, +8]])
detA = linalg.det(A)
print('detA=',detA)
invA = linalg.inv(A)
print('A^{-1}=', invA)
print('A.A^{-1}=', np.dot(A,invA))
```

```
detA= 262.99999999999983
A^{-1}= [[ 0.19771863  0.06463878  0.00760456]
 [-0.121673    0.11406844  0.07224335]
 [-0.03422053 -0.03041825  0.11406844]]
A.A^{-1}= [[ 1.00000000e+00  1.38777878e-17  0.00000000e+00]
 [-5.55111512e-17  1.00000000e+00  0.00000000e+00]
 [ 5.55111512e-17  0.00000000e+00  1.00000000e+00]]
```

```
# Example 6.4: Pick numbers alpha = 1, beta = 2:
import numpy as np
from numpy import linalg

A = np.array([[1,2],[-2,1]])

# get the eigenvectors and eigenvalues:
evalues, evectors = linalg.eig(A)
print('Eigenvalues=',evalues)
print('Eigenvectors=',evectors.T[0], evectors.T[1])
```

```
Eigenvalues= [1.+2.j 1.-2.j]
Eigenvectors= [0.         -0.70710678j 0.70710678+0.j        ] [0.         +0.
 ↪70710678j 0.70710678-0.j        ]
```

```
## Example 6.5

A = np.array([ [-2,+2,-3],[+2,+1,-6],[-1,-2, 0]])
# to access elements, e.g. 0,0:
print(A[0][0])

evalues, evectors = linalg.eig(A)
print('Eigenvalues=', evalues)
print('Eigenvectors=', evectors)
```

```
-2
Eigenvalues= [-3.  5. -3.]
Eigenvectors= [[-0.95257934  0.40824829  0.05155221]
 [ 0.27216553  0.81649658  0.82292764]
 [-0.13608276 -0.40824829  0.5658025 ]]
```

## 6.8 N-Dimensional Newton-Raphson

Using what we have learned thus far about matrix computing, we can solve, numerically, any system of $N$ coupled *linear* equations, e.g. by directly asking NumPy's linalg package to solve:

$\mathbf{A}\vec{x} = \vec{b}$.

What about a system of $N$ *nonlinear* equations?

This can also be solved numerically, by combining matrix computing with our methods for searching for roots to equations, such as the Newton-Raphson method.

Let's discuss the general technique for achieving this, and then apply it to a concrete example in statics.

Let's suppose that our system of $N$ equations can be written in terms of $N$ unknowns, $x_i$, $i = 1, 2, ..., N$.

We can write all of these unknowns in a single column vector:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_N \end{pmatrix}$$

and then the $N$ equations can be written as functions of $\vec{x}$ as:

$f_i(x_1, x_2, ..., x_N) = 0$

or:

$f_i(\vec{x}) = 0, i = 1, ..., N.$

The solutions to these $N$ equations are a set of $N$ values of the $x_i$ that make the $f_i$'s vanish simultaneously.

Our search algorithm will guess a set of $N$ solutions, expand the nonlinear equations $f_i$ into linear form, solves the resulting *linear* equations, and continues to improve the guesses, based on how close the previous one was to making all the $f_i(\vec{x}) = 0$.

Explicitly, let the approximate solutions at any stage be called $x_i$'s. To calculate the new set of $x_i$'s, $x_i' = x_i + \Delta x_i$, for which:

$f_i(x_1 + \Delta x_1, x_2 + \Delta x_2, ..., x_N + \Delta x_N) \approx 0, i = 1, ..., N.$

or, by Taylor series expansion:

$f_i(x_1, ..., x_N) + \sum_{j=1}^{N} \frac{\partial f_i}{\partial x_j} \Delta x_j \approx 0$

we now have a solveble set of $N$ *linear* equations, in the $N$ unknown "modifications" to our current solution, $\Delta x_i$, which we can express as a single matrix equation.

$f_1 + \frac{\partial f_1}{\partial x_1}\Delta x_1 + \frac{\partial f_1}{\partial x_2}\Delta x_2 + ... + \frac{\partial f_1}{\partial x_N}\Delta x_N = 0$

$f_2 + \frac{\partial f_2}{\partial x_1}\Delta x_1 + \frac{\partial f_2}{\partial x_2}\Delta x_2 + ... + \frac{\partial f_2}{\partial x_N}\Delta x_N = 0$

...

$f_N + \frac{\partial f_N}{\partial x_1}\Delta x_1 + \frac{\partial f_N}{\partial x_2}\Delta x_2 + ... + \frac{\partial f_N}{\partial x_N}\Delta x_N = 0$

or in matrix form:

$$\begin{pmatrix} f_1 \\ f_2 \\ \ddots \\ f_N \end{pmatrix} + \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \ddots & \ddots & \ddots & \ddots \\ \frac{\partial f_N}{\partial x_1} & \frac{\partial f_N}{\partial x_2} & \cdots & \frac{\partial f_N}{\partial x_N} \end{pmatrix} \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \ddots \\ \Delta x_N \end{pmatrix} = 0$$

The derivatives and the $f_i$'s are all evaluated at *known* values of the previous guesses, $x_i$, so that only the vector of the modifications, $\Delta x_i$, is unknown.

We may write the above equation in matrix notation as:

$\vec{f} + \mathbf{J}\Delta\vec{x} = 0$

where:

$$\vec{f} = \begin{pmatrix} f_1 \\ f_2 \\ \ddots \\ f_N \end{pmatrix}, \Delta\vec{x} = \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \ddots \\ \Delta x_N \end{pmatrix}, \mathbf{J} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \ddots & \ddots & \ddots & \ddots \\ \frac{\partial f_N}{\partial x_1} & \frac{\partial f_N}{\partial x_2} & \cdots & \frac{\partial f_N}{\partial x_N} \end{pmatrix},$$

and where **J** is known as the Jacobian matrix.

Then:

$\mathbf{J}\Delta\vec{x} = -\vec{f}$.

The above is equivalent to the one-dimensional Newton-Raphson formula: $\Delta x = -(1/f')f$.

Comparing this to the standard matrix form of a system of $N$ linear equations with $N$ unknows:

$\mathbf{A}\vec{x} = \vec{b}$,

we can identify $\mathbf{A} \to \mathbf{J}$, $\vec{x} \to \Delta\vec{x}$ and $\vec{b} \to -\vec{f}$.

This can be solved using the techniques of linear algebra, i.e. in our case, using NumPy's `linalg` module.

Note that the solution of such problems requires either analytical or numerical calculation of the $N^2$ partial derivatives $\partial f_i/\partial x_j$, the elements of the Jacobian matrix, **J**. It is usually more straightforward to program a numerical approximation for the derivatives, e.g. a forward-difference approximation:

$\frac{\partial f_i}{\partial x_j} \simeq \frac{f_i(x_j+\delta x_j)-f_i(x_j)}{\delta x_j}$,

where we vary each $x_j$ independently and $\delta x_j$ are some arbitrary changes you input.

As for the case of the one-dimensional Newton-Raphson method, this search can fail if the initial guess is not close enough to the zero of all the $f_i$'s so that they can be approximated by a linear function. The backtracking technique may be applied here as well, in the present case, progressively decreasing the corrections $\Delta x_i$, until $|f|^2 = |f_1|^2 + |f_2|^2 + ... + |f_N|^2$ decreases.

## 6.9 More Matrix Examples

Before we proceed to Exercise 6.2, let's tackle two more matrix problems.

### 6.9.1 Example 6.6: Your model of some physical system results in N=100 coupled linear equations with N unknowns:

$a_{00}y_0 + a_{01}y_1 + ... + a_{0(N-1)}y_{N-1} = b_0$

$a_{10}y_0 + a_{11}y_1 + ... + a_{1(N-1)}y_{N-1} = b_1$

...

$a_{(N-1)0}y_0 + a_{(N-1)1}y_1 + ... + a_{(N-1)(N-1)}y_{N-1} = b_{N-1}$

In this example, take the matrix **a** to be the Hilbert matrix (see https://en.wikipedia.org/wiki/Hilbert_matrix) and **b** its first column:

$$\mathbf{a} = \begin{pmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{3} \\ \frac{1}{4} \\ \cdots \\ \frac{1}{100} \\ \frac{1}{2} \\ \frac{1}{3} \\ \frac{1}{4} \\ \frac{1}{5} \\ \cdots \\ \frac{1}{101} \\ \ddots \\ \ddots \\ \ddots \\ \ddots \\ \ddots \\ \ddots \\ \frac{1}{100} \\ \frac{1}{101} \\ \frac{1}{102} \\ \cdots \\ \cdots \\ \frac{1}{199} \end{pmatrix}$$

or $a_{ij} = \frac{1}{i+j+1}$.

and $\vec{b} = \begin{pmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{3} \\ \ddots \\ \frac{1}{100} \end{pmatrix}$

Solve the matrix equation, $\mathbf{a}\vec{y} = \vec{b}$ numerically, and compare to the analytic solution: $\vec{y} = \begin{pmatrix} 1 \\ 0 \\ \ddots \\ 0 \end{pmatrix}$

### 6.9.2 Example 6.7: Dirac Gamma Matrices

The Dirac equation extends quantum mechanics to include relativity and spin-1/2 particles (fermions). The extension of the Hamiltonian operator for an electron (a fermion), requires it to contain matrices, and those matrices are expressed in terms of $4 \times 4$ ? matrices, that can be represented in terms of the familiar $2 \times 2$ Pauli matrices ?$_i$:

$$?_i = \begin{pmatrix} 0 \\ ?_i \\ -?_i \\ 0 \end{pmatrix}, i = 1, 2, 3,$$

where:

$$?_1 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}, ?_2 = \begin{pmatrix} 0 \\ -i \\ i \\ 0 \end{pmatrix}, ?_3 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}.$$

Confirm the following properties of the ? matrices:

$?_2^\dagger = ?_2^{-1} = -?_2$,

and

$?_1?_2 = -i \begin{pmatrix} ?_3 \\ 0 \\ 0 \\ ?_3 \end{pmatrix}$

### 6.9.3 Example 6.8: 2-Dimensional Newton-Raphson Warmup

As a warmup towards solving Exercise 6.2, implement a 2-Dimensional Newton-Raphson method:

$\vec{f} + \mathbf{J}\Delta\vec{x} = 0$

where:

$\vec{f} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}, \Delta\vec{x} = \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \end{pmatrix}, \mathbf{J} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix},$

and solve via:

$\mathbf{J}\Delta\vec{x} = -\vec{f}$.

using the `linalg` package.

You will need to:

- Create a function that calculates the partial derivatives that enter the Jacobian, $\frac{\partial f_i}{\partial x_j}$. You may use the central-difference derivative.
- Create a function that calculates the actual Jacobian matrix, calling the aforementioned function.
- Create a function that returns the vector of functions $\vec{f}$.
- Use $\mathbf{J}$ and $-\vec{f}$ to solve the system iteratively for $\Delta\vec{x}$, adding it to $\vec{x}$, until a solution is reached, i.e. $|f_i| \sim 0$ within the required precision.

(a) Check your Jacobian with the *linear* system:

$2x + y - 13 = 0$

$x + y - 9 = 0$

(b) Use your code for the 2-D Newton Raphson to solve the linear system above.

(c) Use your code to solve the nonlinear system:

$x^2 + y - 21 = 0$

$x + y^2 - 29 = 0$

Verify that your solutions correspond to the expected ones ($x = 4, y = 5$).

(d) Use your code to solve the system:

$xe^y = 1$

$-x^2 + y = 1$

# 6.10 Data Fitting

## 6.10.1 Introduction to Data Fitting

Data fitting is an art that should be studied by all scientists.

This section gives an introduction to the subject, examining how to interpolate data, and how to perform least-squares fits to data.

A problem that we wish to tackle: Given a table of measured values, e.g. cross sections for the resonant scattering of neutrons from a nucleus at certain energies, we wish to determine the values for the cross sections at energy values lying between those in the table.

e.g.:

| Energy (MeV) | Cross Section (mb) |
|---|---|
| 0 | 10.6 |
| 25 | 16.0 |
| 50 | 45.0 |
| 75 | 83.5 |
| 100 | 52.8 |
| 125 | 19.9 |
| 150 | 10.8 |
| 175 | 8.25 |
| 200 | 4.7 |

This problem can be solved in different ways:

- *Interpolation*: The simplest is to numerically interpolate between the values of the experimental data. This is direct and easy, but does not account for there being experimental noise (i.e. errors) in the data.
- *Least-squares fitting*: A more appropriate solution is to find the best fit of a theoretical function to the data: start with what we believe to be the "correct" theoretical description of the data, e.g.:

$f(E) = \frac{f_r}{(E-E_r)^2+\Gamma^2/4}$,

where $f_r$, $E_r$ and $\Gamma$ are unknown parameters, and then adjust the parameters to obtain the best fit, in the statistical sense. This best fit may not pass through all (or any!) points.

Both interpolation and least-squares fitting are powerful tools that let you treat tables of numbers as if they were analytic functions, and sometimes let you deduce statistically meaningful constraints from measurements.

## 6.10.2 Lagrange Interpolation

Consider a table of data that we wish to interpolate. We call the independent variables $x$ and its tabulated values $x_i$, $(i = 1, 2, ...)$. We assume that the dependent variable is represented by a function $g(x)$, with tabulated values $g_i = g(x_i)$.

In Lagrange interpolation, we assume that $g(x)$ can be approximated as an $(n-1)$-degree polynomial in each interval $i$:

$g_i(x) \simeq a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1}$ for $x \simeq x_i$.

This fit is *local*: we do not assume that one single function can fit all the data in the table, but instead use a different polynomial (i.e. different set of $a_i$'s) for each interval.

If care is taken, the set of polynomials obtained this way will be used in further calculations, without introducing too much unwanted noise or discontinuities.

The classic interpolation formula was created by Lagrange. In this method, the formula for each interval is written as the sum of polynomials:

$g(x) \simeq g_1\lambda_1(x) + g_2\lambda_2(x) + ... + g_n\lambda_n(x)$,

where the $g_i$ are the measured values of the dependent variable (e.g. the cross section) and the $\lambda_i$'s can be calculated via:

$\lambda_i(x) = \prod_{j=1, j\neq i}^{n} \frac{x-x_j}{x_i-x_j}$,

or explicitly:

$\lambda_i(x) = \frac{x-x_1}{x_i-x_1}\frac{x-x_2}{x_i-x_2}...\frac{x-x_n}{x_i-x_n}$.

E.g. with three points, you would get a second-degree polynomial.

E.g. Assume that we are given the pts and function values:

$x_i = (0, 1, 2, 4)$ and $g_i = (-12, -12, -24, -60)$.

With four points, the Lagrange formula determines a third-order polynomial that reproduces each of the tabulated values.

$g(x) = \frac{(x-1)(x-2)(x-4)}{(0-1)(0-2)(0-4)}(-12) + \frac{(x-0)(x-2)(x-4)}{(1-0)(1-2)(1-4)}(-12) + \frac{(x-0)(x-1)(x-4)}{(2-0)(2-1)(2-4)}(-24) + \frac{(x-0)(x-1)(x-2)}{(4-0)(4-1)(4-2)}(-60)$.

$\Rightarrow g(x) = x^3 - 9x^2 + 8x - 12$

Then:

$g(4) = 4^3 - 9(4^2) + 32 - 12 = -60$, $g(0.5) = -10.125$.

If the data contain little noise, the polynomial can be used with some confidence within the range of the data, but with some risk beyond the range of the data.

Note that Lagrange interpolation has no restriction that the points $x_i$ be evenly spaced.

Usually the Lagrange method is applied only to a small region of the table, with a small value of $n$, despite the fact that the formula works perfect well for fitting a high-degree polynomial to the entire table.

The difference between the value of the polynomial and the actual function can be shown to be large if high derivatives exist in $g(x)$, which would be the case if the data is noisy.

Let's apply Lagrange interpolation to the neutron scattering data!

### 6.10.3 Example 6.9: Lagrange Interpolation

Apply $n$-point Lagrange interpolation to the data given in the table above, varying the number of points $n$. Plot both the data points and the resulting fits.

Which number of points "looks" the best?

### 6.10.4 Cubic Spline Interpolation

Through our example of trying to interpolate the resonant cross section with Lagrange interpolation, we saw that fitting parabolas (i.e. 3-point interpolation), within subintervals in the table, may avoid the erroneous and possibly catastrophic deviations of a high-order formula (e.g. 9-point interpolation).

We also saw that a two-point interpolation with straight lines may not lead you far astray, but it is rarely pleasing to the eye, or precise.

A sophisticated variation of an $n = 4$ interpolation, known as *cubic splines*, often leads to surprisingly eye-pleasing fits.

In this approach, cubic polynomials are fit to the function in each interval, with the additonal constraint that the first and second derivatives be continuous from one interval to the next. The continuity of the slope and curvature is what makes the spline interpolation fit particularly eye-pleasing.

The series of cubic polynomials obtained by spline-fitting a table of data can be integrated and differentiated, and is guaranteed to have well-behaved derivative. The existence of meaningful derivatives is an important consideration: e.g. if the function is a potential, you can take the derivative to obtain the force.

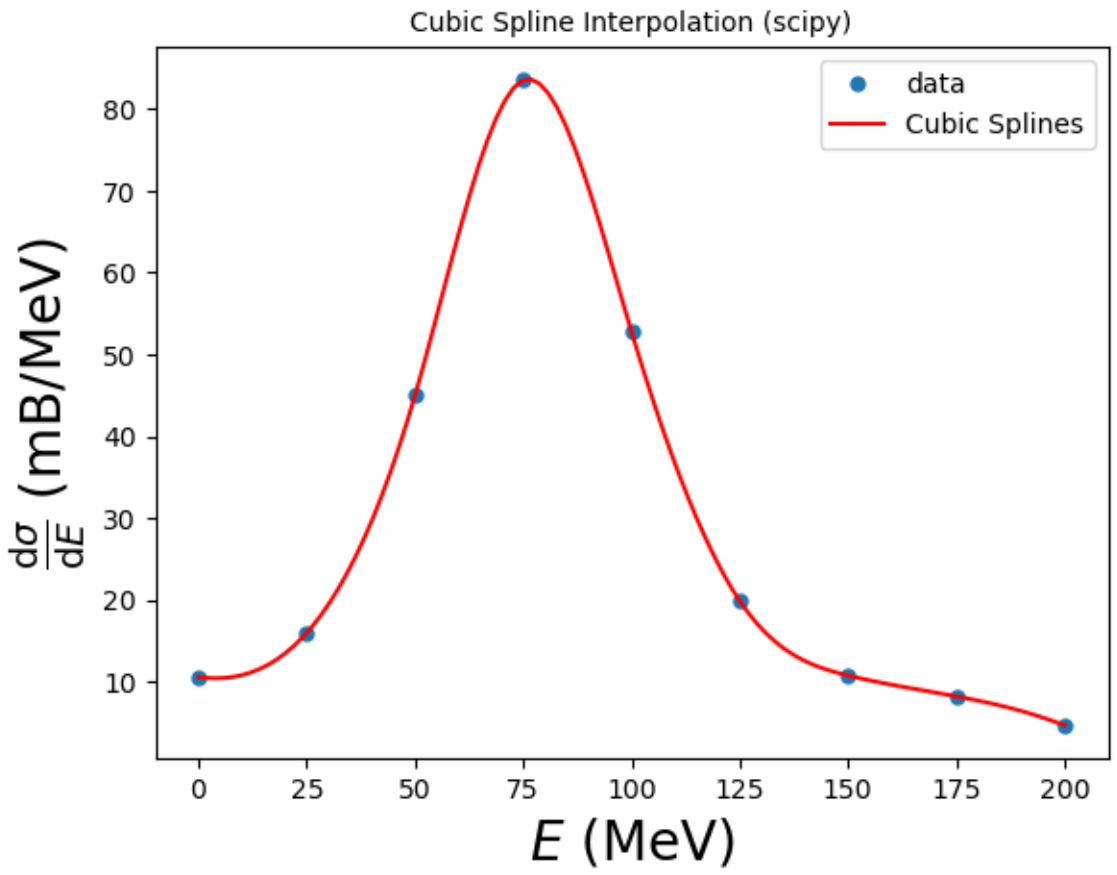
The complexity of simultaneously matching polynomials and their derivatives over all the interpolation points leads to many simultaneous linear equations to be solved. This makes splines unattractive for calculations by hand, yet easy for computers, and popular in both calculations and computer drawing programs.

### 6.10.5 Example 6.10: Cubic Spline Interpolation with scipy

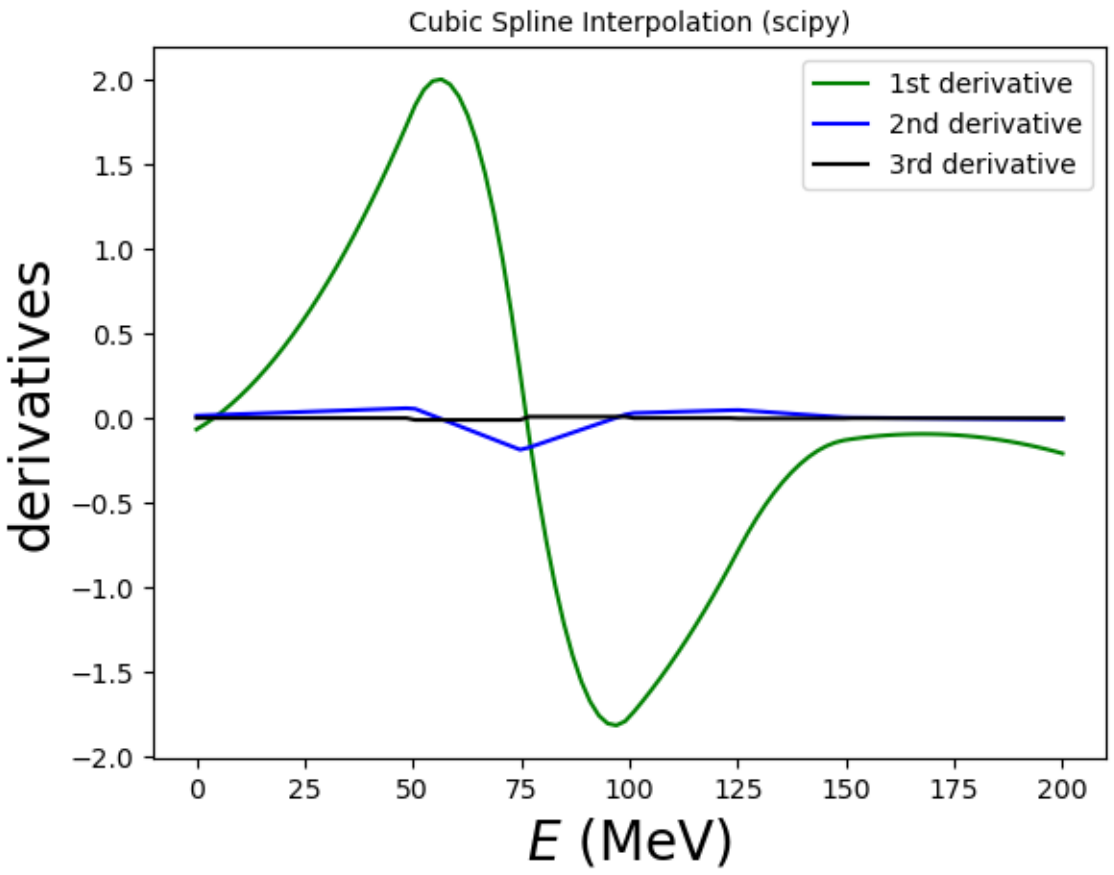
Let's use SciPy's interpolation packages to interpolate the resonant cross section.

```
import scipy
import numpy as np
from scipy import interpolate

# the data:
xdata = np.array([0, 25, 50, 75, 100, 125, 150, 175, 200])
ydata = np.array([10.6, 16.0, 45.0, 83.5, 52.8, 19.9, 10.8, 8.25, 4.7])

# the x coordinates where we want to perform the interpolation:
xinterp = np.linspace(0, 200, 100)

# Cubic Spline Interpolation:
cs = interpolate.CubicSpline(xdata, ydata)
# this creates a function "cs", that we can use at specific points to create the␣
 ↪interpolated y-coordinates
yinterp = cs(xinterp)
```

```
# and now let's plot:
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$E$ (MeV)', fontsize=20) # set the x label
ax.set_ylabel(r'$\frac{\mathrm{d}\sigma}{\mathrm{d}E}$ (mB/MeV)', fontsize=20) # set␣
 ↪the y label
ax.set_title('Cubic Spline Interpolation (scipy)', fontsize=10) # set the title

# plot the data
ax.plot(xdata, ydata, label='data', lw=0,ms=5,marker='o')
# plot the interpolations
ax.plot(xinterp, yinterp, label='Cubic Splines', color='red')

# construct the legend:
ax.legend(loc='upper right')  # Add a legend

plt.show() # show the plot here
```




```
# You can also access the derivatives!

# and now let's plot:
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$E$ (MeV)', fontsize=20) # set the x label
ax.set_ylabel(r'derivatives', fontsize=20) # set the y label
ax.set_title('Cubic Spline Interpolation (scipy)', fontsize=10) # set the title

# Plot the derivatives:
# cs(xinterp, nu=1) calculates the first derivative, etc.
 ax.plot(xinterp, cs(xinterp,nu=2), label='2nd derivative', color='blue')
ax.plot(xinterp, cs(xinterp,nu=3), label='3rd derivative', color='black')

# construct the legend:
ax.legend(loc='upper right')  # Add a legend

plt.show() # show the plot here
```




### 6.10.6 Other SciPy Interpolators

There are many other 1-D interpolating methods in SciPy's interpolate module:

Cubic splines are by construction twice continuously differentiable. This may lead to the spline function oscillating and "overshooting" in between the data points. In these situations, an alternative is to use the so-called monotone cubic interpolants: these are constructed to be only once continuously differentiable, and attempt to preserve the local shape implied by the data. `scipy.interpolate` provides two objects of this kind: PchipInterpolator and Akima1DInterpolator:

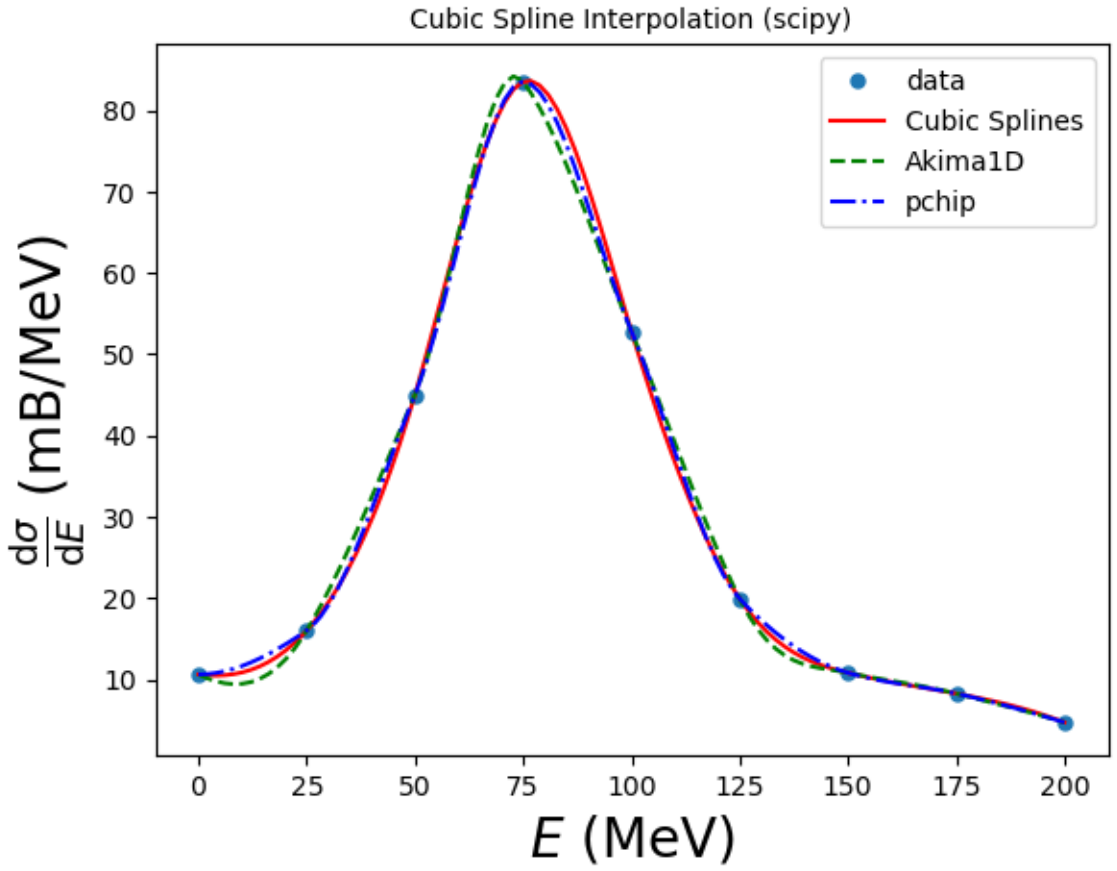
```
ak = interpolate.Akima1DInterpolator(xdata, ydata) # Akima 1D interpolator
pc = interpolate.PchipInterpolator(xdata, ydata) # Pchip interpolator
```

```
# and now let's plot:
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$E$ (MeV)', fontsize=20) # set the x label
ax.set_ylabel(r'$\frac{\mathrm{d}\sigma}{\mathrm{d}E}$ (mB/MeV)', fontsize=20) # set␣
 ↪the y label
ax.set_title('Cubic Spline Interpolation (scipy)', fontsize=10) # set the title
```

(continues on next page)

(continued from previous page)

```
# plot the data
ax.plot(xdata, ydata, label='data', lw=0,ms=5,marker='o')
# plot the interpolations
ax.plot(xinterp, yinterp, label='Cubic Splines', color='red')
ax.plot(xinterp, ak(xinterp), label='Akima1D', color='green', ls='--')
ax.plot(xinterp, pc(xinterp), label='pchip', color='blue', ls='-.')

# construct the legend:
ax.legend(loc='upper right')  # Add a legend

plt.show() # show the plot here
```




B-splines form an alternative (if formally equivalent) representation of piecewise polynomials. This basis is generally more computationally stable than the power basis and is useful for a variety of applications which include interpolation, regression and curve representation. Details are given in the SciPy piecewise polynomials section (https://docs.scipy.org/doc/scipy/tutorial/interpolate/splines_and_polynomials.html#tutorial-interpolate-ppoly), and here we illustrate their usage:

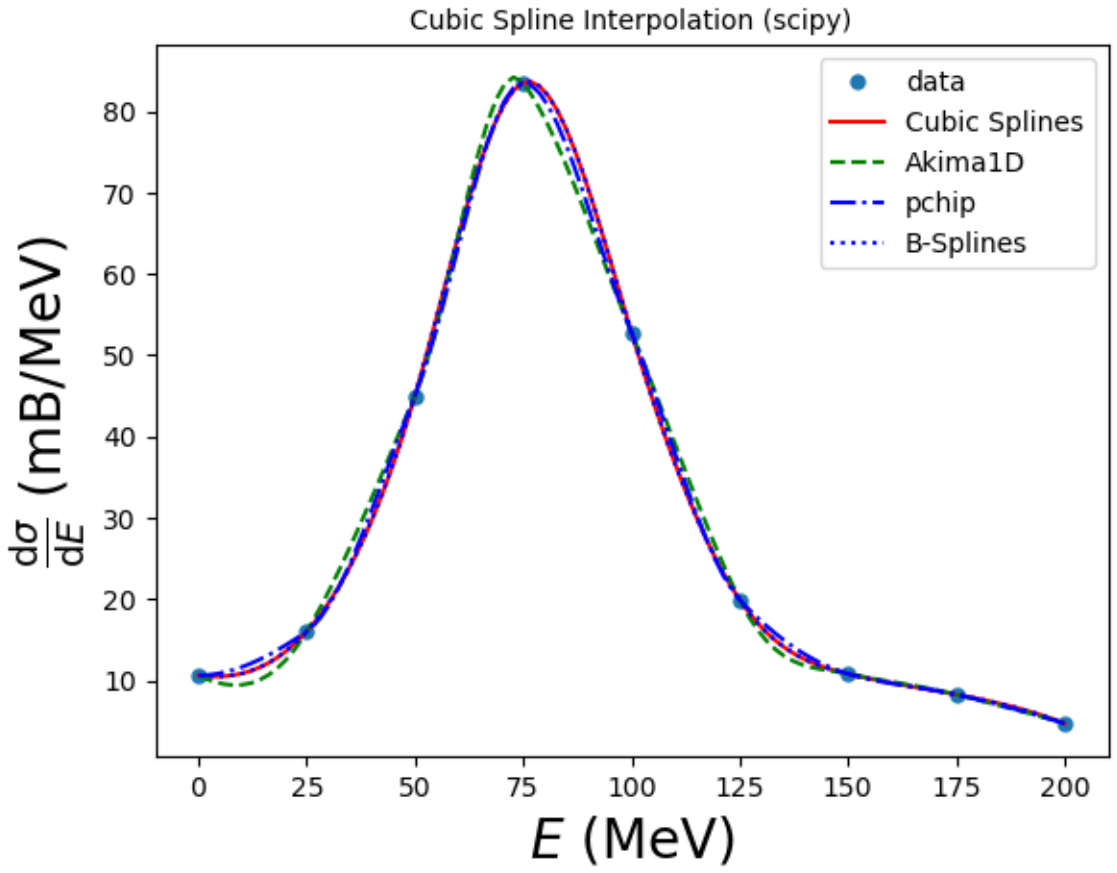
```
# B-spline interpolator
bspl = interpolate.make_interp_spline(xdata, ydata, k=3)
```

```
# and now let's plot:
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$E$ (MeV)', fontsize=20) # set the x label
ax.set_ylabel(r'$\frac{\mathrm{d}\sigma}{\mathrm{d}E}$ (mB/MeV)', fontsize=20) # set␣
 ↪the y label
ax.set_title('Cubic Spline Interpolation (scipy)', fontsize=10) # set the title

# plot the data
ax.plot(xdata, ydata, label='data', lw=0,ms=5,marker='o')
# plot the interpolations
ax.plot(xinterp, yinterp, label='Cubic Splines', color='red')
ax.plot(xinterp, ak(xinterp), label='Akima1D', color='green', ls='--')
ax.plot(xinterp, pc(xinterp), label='pchip', color='blue', ls='-.')
ax.plot(xinterp, bspl(xinterp), label='B-Splines', color='blue', ls=':')

# construct the legend:
ax.legend(loc='upper right')  # Add a legend

plt.show() # show the plot here
```




By default, the result of `make_interp_spline(x, y)` is equivalent to `CubicSpline(x, y)`. The difference is that the former allows several optional capabilities, e.g. it can construct splines of various degrees (via the optional argument `k`).

### 6.10.7 Example 6.11: Global Temperature: Load data with Pandas and Interpolate with SciPy

Taken from: https://climate.nasa.gov/vital-signs/global-temperature/

This graph (data from: "No_Smoothing" column of https://data.giss.nasa.gov/gistemp/graphs/graph_data/Global_Mean_Estimates_based_on_Land_and_Ocean_Data/graph.txt) shows the change in global surface temperature compared to the long-term average from 1951 to 1980. Earth's average surface temperature in 2023 was the warmest on record since recordkeeping began in 1880 (source: NASA/GISS).

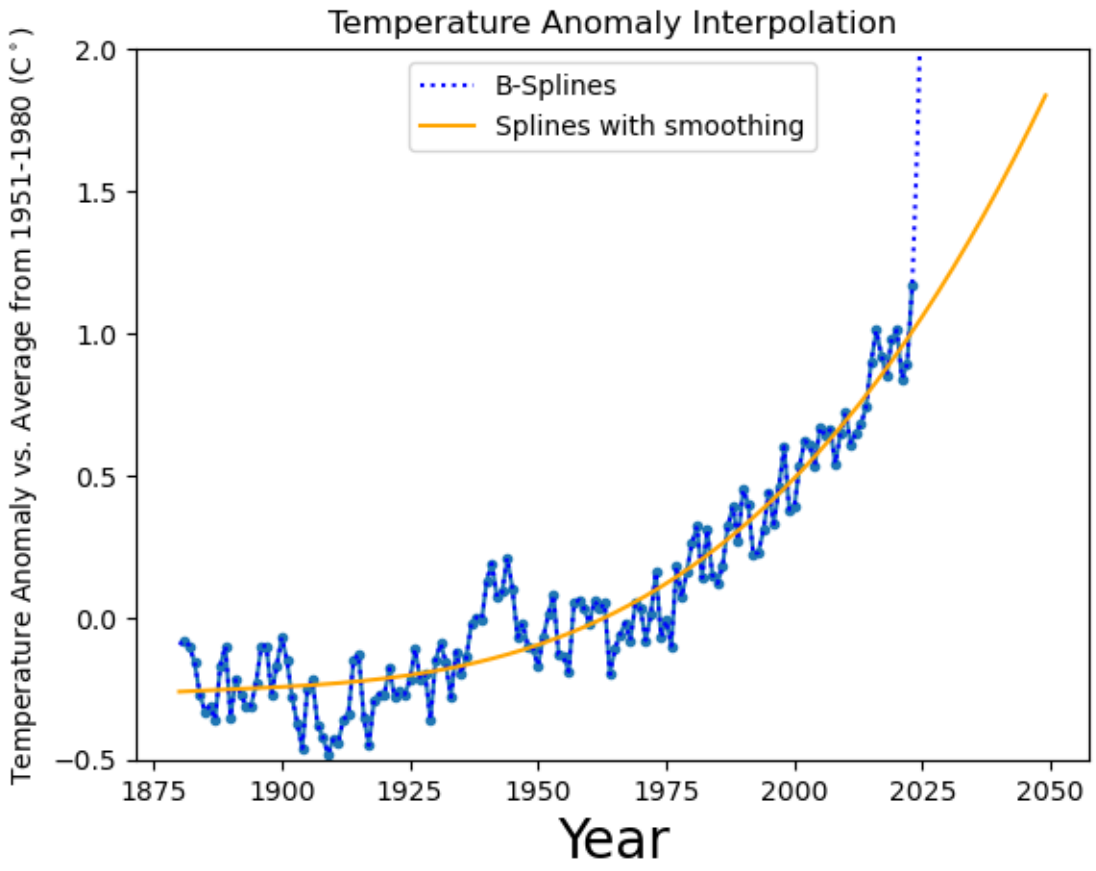
Let's load the data using Python's `pandas` and interpolate them. We can use (very cautiously in this case – this is not a model!) what the temperature *could* be in 2050.

Note that the cubic splines will simply pass through all the points: we don't want that. Here we will see an example of interpolation with "smoothing".

```
import pandas as pd # pandas is a useful and widely-used tool for data analysis
import numpy as np
from scipy import interpolate

# load the data:
df = pd.read_csv('graph.txt', sep='\t') # '\t' represents tabs
# change the names to the expected ones
df.columns = ["Year", "Temp"]
# print to see how it looks like
print(df)

# we can access the columns as df["Year"] and df["Temp"]
# let's interpolate using various methods:
xinterp = np.arange(1880,2050,1)
# interpolators:
bspl = interpolate.make_interp_spline(df["Year"], df["Temp"], k=2)

# This will pass through all the points. We'd rather use something that generates␣
↪smoothing:
spl = interpolate.splrep(df["Year"], df["Temp"], s=10) # s determines the degree of␣
↪smoothing

# get the data points:
yinterp = bspl(xinterp)
ysmooth = interpolate.splev(xinterp, spl, der=0)
```

```
     Year  Temp
0    1881 -0.08
1    1882 -0.10
2    1883 -0.16
3    1884 -0.27
4    1885 -0.33
..    ...   ...
138  2019  0.98
139  2020  1.01
140  2021  0.84
141  2022  0.89
142  2023  1.17

[143 rows x 2 columns]
```

```
# Let's plot them!
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'Year', fontsize=20) # set the x label
ax.set_ylabel(r'Temperature Anomaly vs. Average from 1951-1980 (C$^\circ$)',␣
↪fontsize=10) # set the y label
ax.set_title('Temperature Anomaly Interpolation')

# let's plot the pandas data:
ax.plot(df['Year'], df['Temp'], marker='o', ms=3)

# and the interpolations:
ax.plot(xinterp, yinterp, label='B-Splines', color='blue', ls=':')
ax.plot(xinterp, ysmooth, label='Splines with smoothing', color='orange', ls='-')

# set limits:
ax.set_ylim(-0.5,2)

# construct the legend:
ax.legend(loc='upper center')  # Add a legend

plt.show() # show the plot here
```




```
print("Extrapolated temperature anomaly in 2050:", ysmooth[-1])
```

```
Extrapolated temperature anomaly in 2050: 1.8363666133560805
```

As we will see later in this chapter, data with so much noise is better addressed by least-squares fitting with a theoretical prediction rather than an interpolator.

### 6.10.8 2-D Interpolation

You can also interpolate in multiple dimensions. Let's look at an example in 2D, i.e. two independent variables and one dependent variable that we wish to interpolate.

E.g. we have the data:

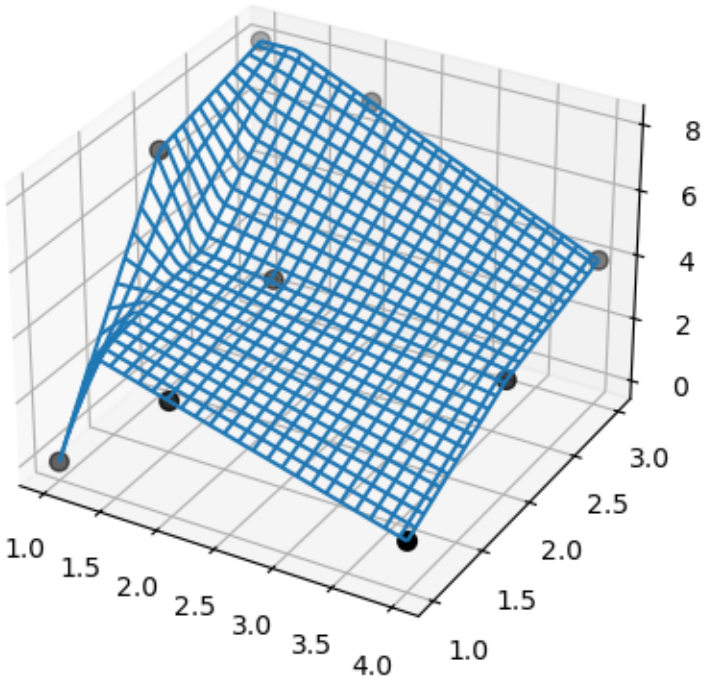
```
# the data:
xdata = np.array([1,1,1,2,2,2,4,4,4])
ydata = np.array([1,2,3,1,2,3,1,2,3])
zdata = np.array([0,7,8,3,4,7,1,3,4])

# interpolating points:
xinterp = np.linspace(min(xdata), max(xdata))
yinterp = np.linspace(min(ydata), max(ydata))

# bivariate B-spline representation of a surface.
tck = interpolate.bisplrep(xdata, ydata, zdata, kx=1, ky=1, s=0)
```

(continues on next page)

(continued from previous page)

```
# evaluate the interpolator:
Z = interpolate.bisplev(xinterp, yinterp, tck)

# Now PLOT!
fig = plt.figure()
ax = fig.add_subplot(projection='3d') # if we want a 3D plot

# necessary for 2-D plot:
X, Y = np.meshgrid(xinterp, yinterp)
Z = Z.reshape(*X.shape).T

# plot the data:
ax.scatter(xdata, ydata, zdata,  'o', color='k', s=48)

# plot a wireframe of the interpolator:
ax.plot_wireframe(X, Y, Z, rstride=2, cstride=2)
```

```
<mpl_toolkits.mplot3d.art3d.Line3DCollection at 0x12f17bc80>
```



### 6.10.9 Least-Squares Fitting

Within the context of data fitting, it is important to emphasize three points:

1. If the data being fit contains errors, the "best-fit", in a statistical sense, should not pass through all the data points.
2. If the theory is not an appropriate one for the data, then its best fit to the data may not be a good fit at all. This is how we know that the theory is not right.
3. Only for the simplest case of a *linear* least-squares fit can we write down a closed-form solution to evaluate and obtain the fit. More realistic problems are usually solved by trial-and-error search procedures.

Imagine that you measured $N_D$ data values of the dependent variable $y$ as a function of the independent variable $x$:

$(x_i, y_i \pm \sigma_i); i = 1, ..., N_D,$

where $\pm\sigma_i$ is the experimental uncertainty in the $i$-th value of $y$.

For simplicity we will assume that errors occur in the dependent variable ($y$), altghough this is hardly ever true.

**Our goal**: to determine how well a mathematical function $y = g(x)$ (i.e. the theory or model) can describe the data.

Additionally: if the theory contains some parameters or constants, our goal can be viewed as determining the best values for these.

Therefore, we assume that the theory function $g(x)$, contains in addition to the functional dependence on $x$, an additional dependence on $N_P$ parameters: $a_1, ..., a_{N_P}$. These are *not* variables, but rather parts of the theoretical model.

With these parameters, we can write the function as:

$g(x) = g(x; \{a_1, a_2, ..., a_{N_P}\}) = g(x; \{a_m\}), m = 1, ..., N_P,$

where $\{a_m\}$ is the set of parameters ($m = 1, ..., N_P$) and $x$ is the independent variable.

Statistics tells us that we can use the so-called "chi-square" variable as a gauge of how well a theoretical function $g$ reproduces the data:

$\chi^2 = \sum_{i=1}^{N_D} \left[\frac{y_i - g(x_i; \{a_m\})}{\sigma_i}\right]^2,$

where the sum $i$ is taken over the $N_D$ experimental data points $(x_i, y_i \pm \sigma_i)$.

Smaller values of $\chi^2$ imply better fits, with $\chi^2 = 0$ occurring if the theoretical curve went through the center of every data point (which never happens in practice!).

The $1/\sigma_i^2$ weighting results in measurements with larger errors contributing less to $\chi^2$.

*Least-squares fitting* involves adjusting the parameters $\{a_m\}$ ($m = 1, ..., N_P$) until a *minimum* $\chi^2$ is found.

This is the best fit possible, and the best way to determine the parameters in a theory.

Therefore, the $N_P$ parameters $\{a_m\}$ that make the $\chi^2$ an extremum are found by solving the $N_P$ equations:

$\frac{\partial \chi^2}{\partial a_m} = 0; m = 1, ..., N_P,$

which imply that:

$\sum_{i=1}^{N_D} \frac{[y_i - g(x_i; \{a_n\})]}{\sigma_i^2} \frac{\partial g(x_i; \{a_n\})}{\partial a_m} = 0; n, m = 1, ..., N_P.$

Often $g(x; \{a_m\})$ has a sufficiently complicated dependence on the values of the $\{a_m\}$ parameters to produce $N_P$ *simultaneous nonlinear* equations for the $\{a_m\}$ values.

But we already know how to solve these! (Trial-and-error searching through the $N_p$-dimensional parameter space).

To be safe, when such a search is completed, you need to check that the minimum of $\chi^2$ is a global one and not a local one. One way to do this is to repeat the search for a whole grid of starting values, and if different minima are found, to pick the one with the lowest $\chi^2$.

When the deviations from theory are a result of random errors described by Gaussian distributions, there are some useful rules of thumb to remember:

- You know your fit is good if the value of $\chi^2$ calculated is approximately equal to the number of degrees of freedom, i.e. $\chi^2 \simeq N_D - N_P$, the difference between the number of data points and the number of parameters in your theory function.
- If $\chi^2 \ll N_D - N_P$, it does not mean you have a great theory or a really precise measurement; instead, you probably have *too many parameters* or have assigned errors ($\sigma_i$) that are too large (i.e. you *overestimated* the errors).
- if $\chi^2 \gg N_D - N_P$, the theory may not be good, or you may have significantly *underestimated* your errors, or you may have errors that are not random.

### 6.10.10 Linear Regression

The $N_P$ simultaneous equations can be simplified considerably if the functions $g(x; \{a_m\})$ depend *linearly* on the parameter values $a_m$, e.g.:

$g(x; a_1, a_2) = a_1 + a_2 x.$

In this case, known as "linear regression", there are two parameters: the slope $a_2$ and the $y$-intercept $a_1$.

Notice that, while there are only two parameters to determine, there still may be an arbitrary number of $N_D$ data points to fit.

*Remember*: A unique solution *is not possible* unless the number of data points is greater than the number of parameters: $N_D \geq N_P$.

For the linear case, there are only two derivatives to consider:

$\frac{\partial g}{\partial a_1} = 1$, $\frac{\partial g}{\partial a_2} = x$.

After substitution, the $\chi^2$ minimization equations can be solved:

$a_1 = \frac{S_{xx} S_y - S_x S_{xy}}{\Delta}$, $a_2 = \frac{S S_{xy} - S_x S_y}{\Delta}$, where:

$S = \sum_{i=1}^{N_D} \frac{1}{\sigma_i^2}$,

$S_x = \sum_{i=1}^{N_D} \frac{x_i}{\sigma_i^2}$,

$S_y = \sum_{i=1}^{N_D} \frac{y_i}{\sigma_i^2}$,

$S_{xx} = \sum_{i=1}^{N_D} \frac{x_i^2}{\sigma_i^2}$,

$S_{xy} = \sum_{i=1}^{N_D} \frac{x_i y_i}{\sigma_i^2}$,

$\Delta = S S_{xx} - S_x^2$.

You can also get an expression for the variance (the square of the uncertainty) in the deduced parameters:

$\sigma_{a_1}^2 = \frac{S_{xx}}{\Delta}$, $\sigma_{a_2} = \frac{S}{\Delta}$.

This is a measure of the uncertainties in the values of the fitted parameters, arising from the uncertainties $\sigma_i$ in the measured $y_i$ values.

A measure of the dependence of the parameters on each other is given by the *correlation coefficient*:

$\rho(a_1, a_2) = \frac{\text{cov}(a_1, a_2)}{\sigma_{a_1} \sigma_{a_2}}$, where:

$\text{cov}(a_1, a_2) = \frac{-S_x}{\Delta}$ is the covariance between $a_1$ and $a_2$.

The above analytic solutions for the parametrs are of the form found in statistics textbooks, but are not optimal for numerical calculations: subtractive cancelation can make the answers unstable. As previously discussed, a rearrangement of the equations can decrease this kind of error.

An improved set of expressions is given by:

$a_1 = \bar{y} - a_2\bar{x}$, $a_2 = \frac{S_{xy}}{S_{xx}}$, $\bar{x} = \frac{1}{N}\sum_{i=1}^{N_D} x_i$, $\bar{y} = \frac{1}{N}\sum_{i=1}^{N_D} y_i$, $S_{xy} = \sum_{i=1}^{N_D} \frac{(x_i-\bar{x})(y_i-\bar{y})}{\sigma_i^2}$, $S_{xx} = \sum_{i=1}^{N_D} \frac{(x_i-\bar{x})^2}{\sigma_i^2}$

### 6.10.11 Example 6.12: Hubble's Law

In 1929, Edwin Hubble examined the data related the radial velocity $v$ of 24 extra-galactic nebulae, to their distance $r$ from our galaxy. He fit them with a straight line:

$v = Hr$, where $H$ is now known as the Hubble constant.

His measurements are given in the file `hubble.txt`, where the first column represents the distance $r$ in Mpc (pc=parsec, defined as the distance at which 1 astronomical unit, the mean Earth-sun distance, subtends an angle of one arcsecond, i.e. 1/3600 of a degree), and the second column is the velocity $v$ in km/s.

We will assume that the errors on $v$ are $\sigma = 1$ km/s, but we will re-evaluate this assumption later on.

a) Load the file using `pandas` and plot the data using `matplotlib`.

b) Compute the least-squares straight-line fit to the data in the form:

$v(r) = a + Hr$,

including the errors on the parameters, $\sigma_a$ and $\sigma_H$.

c) Plot your best fit along with the data.

d) Determine the $\chi^2$ of the fit. Given its value, what would a better estimate of the average error on the measurements be?

### 6.10.12 Quadratic Fits

As long as the function being fitted depends *linearly* on the unknown parameters $a_i$, the condition of minimum $\chi^2$ leads to a set of simultaneous *linear* equations for the $a_m$'s that can be solved by hand or on a computer using matrix techniques. This is true if the function being fitted is a polynomial of any degree.

E.g. suppose we want to fit the quadratic polynomial:

$g(x) = a_1 + a_2 x + a_3 x^2$,

to the experimental measurements $(x_i, y_i \pm \sigma_i)$, $i = 1, ..., N_D$.

Here, $g(x)$ is linear in all the parameters $a_m$, and we still only need to solve linear simultaneous equations, even though $x$ is raised to the second power.

*In contrast*, if we were trying to fit a function of the form $g(x) = (a_1 + a_2 x)e^{-a_3 x}$ to the data, then we would need to solve nonlinear simultaneous equations (see next subsection).

The best fit of a quadratic to the data is obtained by applying the minimum $\chi^2$ condition for $N_P = 3$ parameters and $N_D$ data points.

$\frac{\partial \chi^2}{\partial a_m}$, $m = 1, 2, 3$ leads to three simultaneous linear equations for $a_1$, $a_2$ and $a_3$:

$\sum_{i=1}^{N_D} \frac{[y_i - g(x_i;\{a_m\})]}{\sigma_i^2} \frac{\partial g(x_i;\{a_m\})}{\partial a_n} = 0$; $m, n = 1, 2, 3$, with:

$\frac{\partial g}{\partial a_1} = 1$, $\frac{\partial g}{\partial a_2} = x$, $\frac{\partial g}{\partial a_3} = x^2$.

The $a_m$ dependence only arises from the term in the square brackets in the sums. Since that term only has a linear dependence on the $a_m$'s, all the equations are linear in them.

The equations can be written as:

$Sa_1 + S_x a_2 + S_{xx} a_3 = S_y$,

$S_x a_1 + S_{xx} a_2 + S_{xxx} a_3 = S_{xy}$,

$S_{xx} a_1 + S_{xxx} a_2 + S_{xxxx} a_3 = S_{xxy}$.

with $S = \sum_{i=1}^{N_D} \frac{1}{\sigma_i^2}$, $S_x = \sum_{i=1}^{N_D} \frac{x_i}{\sigma_i^2}$, $S_y = \sum_{i=1}^{N_D} \frac{y_i}{\sigma_i^2}$, $S_{xx} = \sum_{i=1}^{N_D} \frac{x_i^2}{\sigma_i^2}$, $S_{xy} = \sum_{i=1}^{N_D} \frac{x_i y_i}{\sigma_i^2}$, and so on.

If we now define the vector of unknowns:

$$\vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix},$$

the matrix:

$$\mathbf{A} = \begin{pmatrix} S \\ S_x \\ S_{xx} \\ S_x \\ S_{xx} \\ S_{xxx} \\ S_{xx} \\ S_{xxx} \\ S_{xxxx} \end{pmatrix},$$

and:

$$\vec{b} = \begin{pmatrix} S_y \\ S_{xy} \\ S_{xxy} \end{pmatrix},$$

then to get the coefficients $a_1, a_2, a_3$, we need to solve the matrix equation:

$\mathbf{A}\vec{a} = \vec{b}$, for $\vec{a}$, using the techniques that we have previously discussed in this chapter.

### 6.10.13 Nonlinear Fitting

Earlier in this section we saw the Breit-Wigner resonance formula:

$f(E) = \frac{f_r}{(E-E_r)^2 + \Gamma^2/4}$.

If we have a set of data that we wish to describe with such a formula, we would need to determine what values for the parameters $E_r$, $f_r$ and $\Gamma$ provide the best fit to the data.

Since $f$ is a nonlinear function of the parameters, the equations that result from minimizing $\chi^2$ are *nonlinear* as well!

But we already saw how to use the Newton-Raphson algorithm to search for solutions of simultaneous nonlinear equations. The method involved an expansion of the equations about the previous "guess" point, to obtain a set of linear equations, and then solving the linear equations with matrix libraries.

In what follows, we will use the same combination of fitting, trial-and-error searching and matrix algebra, to conduct a nonlinear least-squares fit to the data.

To get the best fit, we need to find values of the $N_P$ parameters $a_m$ in the theory $g(x; \{a_m\})$ that minimize:

$\chi^2 = \sum_{i=1}^{N_D} \left( \frac{y_i - g(x_i)}{\sigma_i} \right)^2$.

This leads to the $N_P$ equations:

$\sum_{i=1}^{N_D} \frac{[y_i - g(x_i;\{a_n\})]}{\sigma_i^2} \frac{\partial g(x_i;\{a_n\})}{\partial a_m} = 0, m = 1, ..., N_P.$

For the sake of simplicity, let's rewrite the theory function using the redefinitions:

$f_r \to a_1$, $E_r \to a_2$, $\Gamma^2/4 \to a_3$, $E \to x$ such that:

$g(x; a_1, a_2, a_3) = \frac{a_1}{(x-a_2)^2+a_3}$.

The derivatives required are then:

$\frac{\partial g}{\partial a_1} = \frac{1}{(x-a_2)^2+a_3}$, $\frac{\partial g}{\partial a_2} = \frac{-2a_1(x-a_2)}{[(x-a_2)^2+a_3]^2}$, $\frac{\partial g}{\partial a_3} = \frac{-a_1}{[(x-a_2)^2+a_3]^2}$.

Substituting into the best-fit condition yields three simultaneous nonlinear equations in $a_1, a_2, a_3$ that we need to solve in order to fit the $N_D$ data points.

If all the errors are equal, we then have the following equations in the form that we require for the Newton-Raphson method, i.e. $f_i(a_1, a_2, ..., a_N) = 0$, $i = 1, ..., N$:

$f_1(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{y_i - g(x_i;a_1,a_2,a_3)}{(x_i-a_2)^2+a_3} = 0,$

$f_2(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{[y_i - g(x_i;a_1,a_2,a_3)](x_i-a_2)}{[(x_i-a_2)^2+a_3]^2} = 0,$

$f_3(a_1, a_2, a_3) = \sum_{i=1}^{9} \frac{y_i - g(x_i;a_1,a_2,a_3)}{[(x_i-a_2)^2+a_3]^2} = 0$

CHAPTER

SEVEN

# ORDINARY DIFFERENTIAL EQUATIONS

## 7.1 Introduction

Many of the laws of physics are formulated in terms of differential equations.

Part of the power of computational tools is that it is easy to solve almost every differential equation. E.g. one can go beyond the small displacement in the description of oscillations, and explore interesting nonlinear phenomena.

Let's begin with a recap of some mathematical preliminaries that pertain to differential equations, and then move on, in this chapter, to to discuss algorithms for solving the ordinary differential equations.

## 7.2 Mathematical Preliminaries

**ORDER**: The order of a differential equation refers to the degree of the derivative.

e.g. the most general form for a *first-order* differential equation is:

$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y)$.

e.g. even if $f(x, y)$ is a nasty function of $x$ and $y$:

$\frac{\mathrm{d}y}{\mathrm{d}x} = x^2 y + x^{17} + y^{13}$,

the equation is still *first order*.

A general form of a *second-order* differential equation is:

$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} + \lambda\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, \frac{\mathrm{d}y}{\mathrm{d}x}, y)$.

The RHS may involve any power of the first derivative as well.

In the above, $x$ is known as the *independent* variable and $y$ is the *dependent* variable.

**ORDINARY VS. PARTIAL**

*Ordinary* diffrential equations contain *just one independent* variable, e.g. $x$ in our discussion.

In contrast, *partial* differential equations contain *several independent variables*.

E.g. the Schrödinger equation in three dimensions:

$i\hbar\frac{\partial\Psi(\vec{x},t)}{\partial t} = -\frac{\hbar^2}{2m}\left[\frac{\partial^2\Psi}{\partial x^2} + \frac{\partial^2\Psi}{\partial y^2} + \frac{\partial^2\Psi}{\partial z^2}\right] + V(\vec{x})\Psi(\vec{x}, t)$,

is a partial differential equation with four independent variables $\vec{x} = (x, y, z)$ and $t$, and one dependent variable $\Psi(\vec{x}, t)$.

**LINEAR VS. NONLINEAR**

Part of the strength of computational methods is that we are no longer limited to solving linear equations.

A *linear* equation is one in which only the first power of $y$ or of $\mathrm{d}^n y/\mathrm{d}x^n$ appears, e.g.:

$\frac{\mathrm{d}y}{\mathrm{d}x} = g(x)y(x)$ is *linear*,

$\frac{\mathrm{d}y}{\mathrm{d}x} = \lambda y(x) - \lambda^2 y^2(x)$ is *nonlinear*.

*WARNING*: An important property of linear equations is no longer valid for nonlinear equations: the principle of superposition, that lets us add solutions together to form new ones: e.g. if $A(x)$ is a solution and $B(x)$ is a solution to a linear differential equation, then their sum:

$y(x) = aA(x) + bB(x)$, is also a solution, for arbritrary values of the constants $a$ and $b$. This would no longer be true for a nonlinear equation!

**Initial and boundary conditions**

The general solution of a first-order differential equation always contains one arbitrary constant. The general solution of a second-order differential equation contains two, and so on.

For any specific problem, these constants are fixed by the initial conditions. E.g. for a first-order equation for $y(x)$, the sole initial condition necessary is $y(x = x_0)$, i.e. the value of $y$ at some value of $x$.

For a second-order equation, two initial conditions are required. These may e.g. be the position and velocity at some time when solving a problem involving Newton's second law of motion:

$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = \frac{1}{m}F(y,t)$.

In addition to the initial conditions, it is possible to further restrict the solutions of differential equations, e.g. through *boundary conditions* that constrain one solution to have fixed values at the boundaries of the solution space.

## 7.3 Dynamic form for ODEs

The most general form of an ODE is a *set* of $M$ *coupled first-order* differential equations:

$\frac{\mathrm{d}\vec{y}}{\mathrm{d}x} = \vec{f}(x,\vec{y})$,

where $x$ is the independent variable and $\vec{y}$ is a vector of $M$ dependent variables. The vector of functions $\vec{f}$ is an $M$-component vector as well.

Differential equations of higher order can be written in this first-order coupled form, by introducing auxiliary functions.

Consider, for example, the one-dimensional motion (in the $z$ direction), of a particle of mass $m$ under a force *field* F(z), described by the second-order differential equation (Newton's second law):

$m\frac{\mathrm{d}^2 z}{\mathrm{d}t^2} = F(z)$.

If we define the momentum as: $p(t) = m\frac{\mathrm{d}z}{\mathrm{d}t}$,

then Newton's law becomes two coupled first-order equations (known as Hamilton's equations):

$\frac{\mathrm{d}p}{\mathrm{d}t} = F(z)$,

$\frac{\mathrm{d}z}{\mathrm{d}t} = \frac{p}{m}$,

which are of the desired general form.

It is therefore sufficient to consider in detail only methods for first-order ODEs!

Since the matrix structure of coupled differential equations is of this natural form, we can focus on the case where there is only one independent variable, which can be generalized readily.

Thus, we will begin by consider the methods for solving:

$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x,y)$, for a single dependent variable $y(x)$.

In this chapter, we will discuss methods for solving ODEs, with emphasis on the initial value problem, i.e.:

Find $y(x)$ given the value of $y$ at some initial point, say $y(x = 0) = y_0$.

This is the kind of problem that occurs, e.g. when we are given the initial position and momentum of a particle and we wish to find its subsequent motion.

Later on, we will discuss the equally important boundary value and eigenvalue problems.

## 7.4 ODE Algorithms

### 7.4.1 Euler's Method

We are interested in the solution of $\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y)$, with the initial condition $y(x = 0) = y_0$.

More specifically, we are usually interested in the value of $y$ at particular value of $x$, say $x = 1$.

The general strategy is to divide the interval $[0, 1]$ into a large number, $N$, of equally-spaced subintervals of length $h = 1/N$ and then to develop a recursion formula relating $y_n$ to $\{y_{n-1}, y_{n-2}, ...\}$, where $y_n$ is our approximation to $y(x_n = nh)$. Such a recursion formula would then allow a step-by-step integration of the differential equation from $x = 0$ to $x = 1$.

One of the simplest algorithms is known as Euler's method, in which we consider the differential equation at the point $x_n$, and replace the derivative on the LHS by its forward-difference approximation:

$\frac{y_{n+1}-y_n}{h} + \mathcal{O}(h) = f(x_n, y_n)$,

so that, if we solve for $y_{n+1}$, we obtain a recursion relation expressing $y_{n+1}$ in terms of the "previous" value, $y_n$:

$y_{n+1} = y_n + hf(x_n, y_n) + \mathcal{O}(h^2)$.

The formula has a "local" error (i.e. the error made in taking a single step from $y_n \to y_{n+1}$) that is $\mathcal{O}(h^2)$. The "global" error made in finding $y(1)$, after taking $N$ such steps in integrating from $x = 0$ to $x = 1$ is then $N\mathcal{O}(h^2) \sim \frac{1}{h}\mathcal{O}(h^2) \sim \mathcal{O}(h)$.

Evidently, this error decreases only linearly with decreasing step size, so that if we halve $h$ (and perform twice as many steps), we halve the error of the final answer.

Let's apply this method to a simple problem.

### 7.4.2 Example 7.1: An Application of Euler's Method

Apply Euler's method to solve:

$\frac{\mathrm{d}y}{\mathrm{d}x} = -xy$ with initial condition $y(0) = 1$ from $x = 0$ to $x = 3$.

Use various step sizes: $h = 0.500, 0.200, 0.100, 0.050, 0.020, 0.010, 0.005, 0.002, 0.001$ and calculate the error relative to the analytical solution $y(x) = \exp -x^2/2$ at the points $x = 1$ and $x = 3$. Does it scale as you expect with $h$?

Plot the $y(x)$ for a few step sizes, e.g. $h = 0.500, 0.050, 0.001$, as well as the analytical function.

### 7.4.3 Example 7.2: Evaluation of the Integration Algorithm

A simple and often stringent test of an accurate numerical integration is to use the final value of $y$ obtained as the initial condition to integrate backward from the final value of $x$ to the starting point. The extent to which the resulting value of $y$ differs from the original initial condition is then a measure of the inaccuracy.

Apply this test to Example 7.1.

Although Euler's method seems to work quite well, it is generally unsatisfactory due to its lower-order accuracy.

The algorithm is not recommended for general use, but it is commonly used to start off more precise algorithms.

## 7.5 Runge-Kutta Methods

There is a lot of freedom in writing down algorithms for integrating differential equations. There exists a large number of them, each having their own pecularities and advantages.

One very convenient and widely used class of methods are the Runge-Kutta algorithms, which come in varying orders of accuracy.

Here, we derive the second-order version (`rk2`) to give the spirit of the approach and then state the equations for the fourth-order method.

We begin by writing down the formal integral of our differential equation:

$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y)$,

as:

$y(x) = \int f(x, y)\mathrm{d}x$,

integrating one step leads to (exactly):

$y_{n+1} = y_n + \int_{x_n}^{x_{n+1}} f(x, y)\mathrm{d}x$.

We then expand $f(x, y)$ in a Taylor series about the midpoint of the integration interval:

$f(x, y) \simeq f(x_{n+1/2}, y_{n+1/2}) + (x - x_{n+1/2}) \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_{n+1/2}} + \mathcal{O}(h^2)$.

Substituting into the expression for $y_{n+1}$, the linear term integrates to zero because it is equally positive and negative in the interval $[x_n, x_{n+1}]$:

$\int_{x_n}^{x_{n+1}} (x - x_{n+1/2}) \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_{n+1/2}} \mathrm{d}x = \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_{n+1/2}} \int_{x_n}^{x_{n+1}} (x - x_{n+1/2})\mathrm{d}x = 0.$

Then:

$y_{n+1} = y_n + \left[f(x_{n+1/2}, y_{n+1/2}) + \mathcal{O}(h^2)\right] \int_{x_n}^{x_{n+1}} \mathrm{d}x$,

which leads to:

$y_{n+1} = y_n + hf(x_{n+1/2}, y_{n+1/2}) + \mathcal{O}(h^3)$,

where the error arises from the quadratic term in the Taylor series.

Although it seems as if we need to know the value of $y_{n+1/2}$, appearing in $f$ in the right-hand side of this equation for it to be of any use, this is not quite true:

Since the error is already $\mathcal{O}(h^3)$, an approximation to $y_{n+1/2}$ whose error is $\mathcal{O}(h^2)$ is good enough. This is provided by the simple Euler's method:

$y_{n+1/2} = y_n + \frac{h}{2} f(x_n, y_n) + \mathcal{O}(h^2)$,

and thus, if we define $k_1 = hf(x_n, y_n)$, we can write:

$y_{n+1} = y_n + hf(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1) + \mathcal{O}(h^3)$.

This is the second-order Runge-Kutta algorithm. It requires the evaluation of $f$ twice for each step.

A fourth-order algorithm, which requires $f$ to be evaluated four times for each integration step and has a local accuracy of $\mathcal{O}(h^5)$ has been found by experience to give the best balance between accuracy and computational effort. It can be written as follows, with the $k_i$s as intermediate variables:

$k_1 = hf(x_n, y_n)$,

$k_2 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1)$,

$k_3 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2)$,

$k_4 = hf(x_n + h, y_n + k_3)$,

$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + \mathcal{O}(h^5)$.

### 7.5.1 Example 7.3: Try out the second- and fourth-order Runge-Kutta methods on the problem defined in Example 7.1.

Compare the accuracy of the methods.

### 7.5.2 Application: Nonlinear Oscillators

With several algorithms for solving ODEs in our possession, we can now examine the behavior of oscillators beyond the harmonic approximation.

Newton's second law provides the equation of motion for an oscillator, which could be, e.g. a mass attached to a spring moving in one dimension, $x$:

$F_k(x) + F_{\text{ext}}(x,t) = m\frac{\mathrm{d}^2x}{\mathrm{d}t^2}$,

where $F_k(x)$ is the restoring force, e.g. exerted by a spring, and $F_{\text{ext}}(x,t)$ is an external (driving) force, that may also depend on time.

Let's examine two models that can describe the departure from linearity, with no external forces being applied ($F_{\text{ext}} = 0$).

For the first model, let's introduce a potential that is a harmonic oscillator for small displacements $x$, but also contains a perturbation that introduces a nonlinear term to the force for large values of $x$:

$V_k(x) \simeq \frac{1}{2}kx^2(1 - \frac{2}{3}\alpha x)$.

Taking he derivative, we can derive $F_k(x)$:

$F_k(x) = -\frac{\mathrm{d}V_k}{\mathrm{d}x} = -kx(1 - \alpha x)$,

and the ODE that we need to consider has the form:

$m\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -kx(1 - \alpha x)$.

For $\alpha x \ll 1$, we recover simple harmonic motion, but as $x \to 1/\alpha$, the *anharmonic* effects would be large.

As long as $x < 1/\alpha$, there will be a restoring force, and the motion will be periodic. In general, there will be an asymmetry in the motion to the right and left of the equilibrium position.

As a second model for nonlinear oscillators, we will consider that the spring's potential function is proportional to some arbitrary *even* power $p$ of the displacement from equilibrium:

$V(x) = \frac{1}{p}kx^p$, ($p$ even),

such that the force is a restoring force:

$F_k(x) = -\frac{\mathrm{d}V}{\mathrm{d}x} = -kx^{p-1}$.

For $p = 2$, we recover the harmonic oscillator. As $p$ increases, the potential becomes more and more like a square well, where the mass almost moves freely inbetween "collisions" with the wall at $x \simeq \pm 1$. Regardless of the value of $p$, the motion will be periodic, but will only be harmonic for $p = 2$.

Newton's law gives the second-order ODE that we need to solve:

$m\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -kx^{p-1}$.

Following our discussion on the numerical solution of ODEs, we can reduce the second-order ODE into two coupled first-order equations via an auxiliary variable, in this case the momentum $p$:

$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{p}{m}$,

$\frac{\mathrm{d}p}{\mathrm{d}t} = F_k(x)$.

Let's begin by examining simple harmonic motion, that we already know a lot about, before we consider the nonlinear cases.

### 7.5.3 Example 7.4: Oscillator warmup: Simple Harmonic Motion

(a) Generalize the single-variable methods that we have derived to the two-variable case relevant for oscillatory motion.

(b) Integrate the equations of motion for single harmonic motion, i.e. with $F_k(x) = -kx$, for values of $k = 4\pi^2$ and $m = 1$ and verify that the period $T$ is what you expect from your Physics I course (recall: $T = 2\pi\sqrt{m/k}$). You may choose the initial conditions (e.g. $p = 0$ and $x = x_0$ at $t = t_0$, where $x_0$ would be the amplitude).

We are now in a position to investigate nonlinear oscillations. Let's consider our two models in Exercise 7.1.

CHAPTER

EIGHT

# AN INTRODUCTION TO NONLINEAR DYNAMICS AND CHAOS

## 8.1 Introduction

We have already had a glimpse at nonlinear systems, e.g. in the case of the pendulum without the small-angle approximation. The aim of this chapter is to provide a brief foray into some of the formal aspects of the analysis of dynamical nonlinear systems, while introducing concepts such as flows in phase space, fixed points, linearization, and chaotic systems.

All of these concepts are part of the subject of dynamics, i.e. the study of systems that evolve in time.

We will use the computational knowledge we have gained thus far to obtain a quantitative understanding of some simple systems.

## 8.2 The Importance of being Nonlinear

There are two types of dynamical systems: differential equations and iterated maps (or difference equations). Here we will study differential equations, since we already have the machinery developed to solve them, even in their nonlinear form.

Why are nonlinear problems harder to solve than linear ones?

The answer lies in the fact that linear systems can be broken down into parts. Each part can be solved sparately and finally recombined to get an answer. This allows for fantastic simplification of complex problems, and underlies methods such as normal modes, Laplace transforms, superposition arguments, and Fourier analysis.

A linear system is precisely equal to the sum of its parts!

But many things in nature don't act this way, e.g. whenever parts of a system interfere, or cooperate, or compete, there are nonlinear interactions going on.

Most of everyday life is nonlinear and the principle of superposition fails spectacularly!

Within the realm of physics, nonlinearity is vital to the operation of a laser, the formation of turbulence in a fluid, and the superconductivity of Josephson junctions.

## 8.3 Flows on the Line

### 8.3.1 Fixed Points and Stability

Let's start by examining a one-dimensional, or first-order system, described by a differential equation of the form:

$\dot{x} = f(x)$,

where, as usual, $\dot{} \equiv \mathrm{d}/\mathrm{d}t$.

Pictures are often more helpful than formulas when analyzing a nonlinear system.

Consider $\dot{x} = \sin x$. The general solution is:

$t = -\ln|\csc x + \cot x| + C$.

If the initial condition is $x = x_0$ at $t = 0$, then:

$t = \ln\left|\frac{\csc x_0 + \cot x_0}{\csc x + \cot x}\right|$.

The result is exact, but it is a headache to interpret!

In contrast, a graphical analysis is clear and simple.

Think of $t$ as time, $x$ as the position of an imaginary particle moving along the real line (i.e. left or right), and $\dot{x}$ the velocity of the particle.

$\dot{x} = f(x)$ represents a *vector field* along the line. The differential equation dictates the velocity $\dot{x}$ at each $x$.

To sketch the vector field, it is convenient to plot $\dot{x}$ versus $x$, then draw arrows on the $x$-axis to indicate the corresponding velocity vector $\dot{x}$ at each $x$.

Arrows would point to the right when $\dot{x} > 0$ and to the left when $\dot{x} < 0$.

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

x = np.linspace(-3 *np.pi+0.2, 3 * np.pi-0.2, 600) #
xdot = np.sin(x)

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$\dot{x}$', fontsize=20) # set the y label
ax.set_title('A vector field on the line', fontsize=10) # set the title

# set the x and y limits:
ax.set_xlim(-3*np.pi+0.2, 3*np.pi-0.2)
ax.set_ylim(-1.1,1.1)

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
 ↪and colors:
ax.plot(x, xdot, color='blue', linestyle='-', label='a sine curve')

# change the axis labels to correspond to [0, pi/2, pi, 1.5 * pi, 2*pi, 2.5*pi, 3*pi]
ax.set_xticks([-3*np.pi, -2.5*np.pi, -2*np.pi, -1.5 * np.pi, -np.pi, -np.pi/2, 0, np.
 ↪pi/2, np.pi, 1.5 * np.pi, 2*np.pi, 2.5*np.pi, 3*np.pi])
ax.set_xticklabels(['$-3\\pi$','$5\\pi/2$', '$-2\\pi$', '$-3\\pi/2$', '$-\\pi$','$-\\
 ↪pi/2$', '0', '$\\pi/2$', '$\\pi$', '$3\\pi/2$', '$2\\pi$', '$5\\pi/2$', '$3\\pi$'])
```

(continues on next page)

(continued from previous page)

```
# plot the fixed points:
xfpu = [-2*np.pi, 0, 2*np.pi]
yfpu = [0, 0, 0]
plt.scatter(xfpu, yfpu, s=80, facecolors='none', edgecolors='r', zorder=11)

# plot the fixed points:
xfps = [-np.pi, np.pi]
yfps = [0, 0]
plt.scatter(xfps, yfps, s=80, facecolors='r', edgecolors='r', zorder=11)


# the axes:
ax.set_aspect(3)
ax.axhline(y=0, color='k', alpha=0.5)
ax.axvline(x=0, color='k')

# Vector origin location
X = [0, 0, -2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi]
Y = [0, 0, 0, 0, 0, 0]

# Directional vectors
U = [np.pi/2-0.1, -np.pi/2+0.1, np.pi/2-0.1, -np.pi/2+0.1, -np.pi/2+0.1, np.pi/2-0.1]
V = [0, 0, 0, 0, 0, 0]
# Creating plot
plt.quiver(X, Y, U, V, color='black', units='xy', scale=1, zorder=10, width=0.08)

plt.show() # show the plot here
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 import matplotlib.pyplot as plt # import matplotlib, a conventional module␣
 ↪name is plt
      2 import numpy as np
      4 x = np.linspace(-3 *np.pi+0.2, 3 * np.pi-0.2, 600) #

ModuleNotFoundError: No module named 'matplotlib'
```

A more physical way to think of the vector field: imagine that a fluid is flowing steadily along the $x$-axis with a velocity that varies from place to place, according to $\dot{x} = \sin x$. The flow is to the right when $\dot{x} > 0$ and to the left when $\dot{x} < 0$.

At points where $\dot{x} = 0$, there is no flow! These are the *fixed points*.

From the graph, we can see that there are two kinds of fixed points: solid dots represent *stable fixed points*, often called attractors or sinks, because the flow is toward them. Open circles represent unstable fixed points, also called repellers or sources.

Armed with this picture, we can easily understand the solutions of the differential equation $\dot{x} = \sin x$ qualitatively:

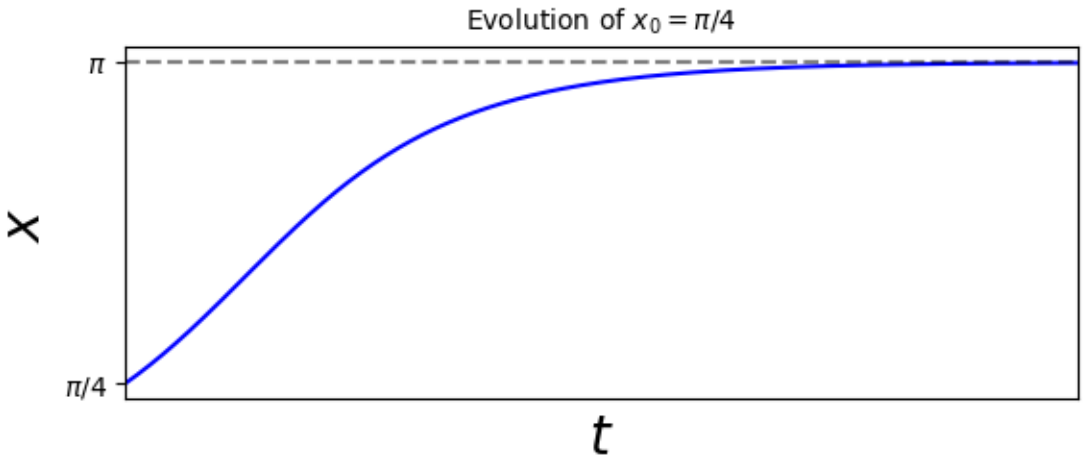
We start an imaginary particle at $x_0$ and watch how it is carried along by the flow. E.g. a particle starting at $x_0 = \pi/4$ moves to the right faster and faster until it crosses $\pi/2$, where $\sin x$ reaches its maximum, and eventually approaches the stable fixed point $x = \pi$ from the left:

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
x = np.linspace(np.pi/4+0.00000001, np.pi-0.000001, 600)
t = np.log( np.abs( (1/np.sin(np.pi/4) + 1/np.tan(np.pi/4)) / (1/np.sin(x) + 1/np.
 ↪tan(x)) ) )
fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$t$', fontsize=20) # set the x label
ax.set_ylabel(r'$x$', fontsize=20) # set the y label
ax.set_title('Evolution of $x_0=\\pi/4$', fontsize=10) # set the title

ax.plot(t, x, color='blue', linestyle='-', label='a sine curve')

# change the axis ticks, labels, and aspect
ax.set_yticks([np.pi/4, np.pi])
ax.set_yticklabels(['$\\pi/4$', '$\\pi$'])
ax.set_xticks([])
ax.set_xlim(0,7)
ax.set_aspect(1)

# the axes:
ax.axhline(y=np.pi, color='k', alpha=0.5, ls='--')


plt.show() # show the plot here
```




A picture cannot tell us certain quantitative things, but can still give us a good understanding of the physical situation. We can then use numerical methods to understand the solutions quantitatively.

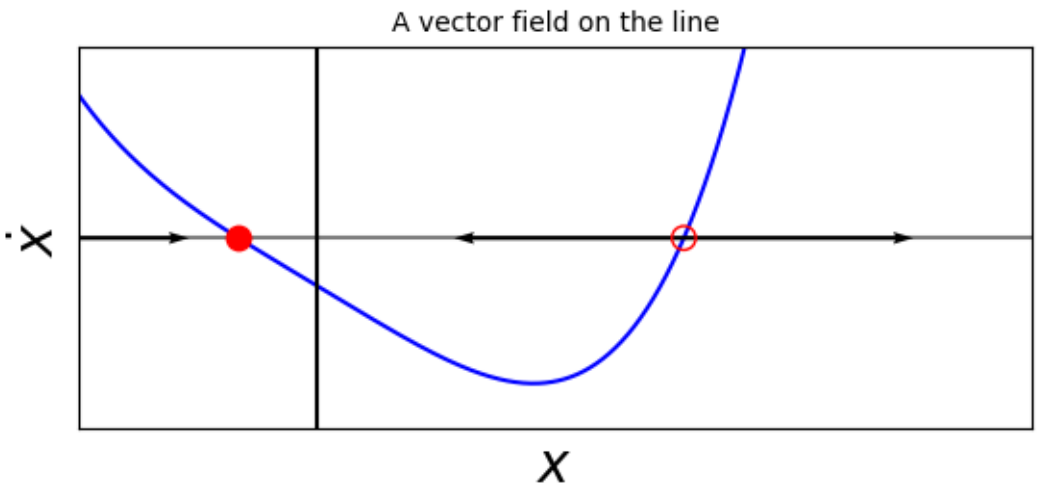
The ideas can be extended to any one-dimensional system of the form $\dot{x} = f(x)$.

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

x = np.linspace(-2, 2, 600) #
xdot = x**4 - 3*x - 1

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.
```

(continues on next page)

(continued from previous page)

```
# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$\dot{x}$', fontsize=20) # set the y label
ax.set_title('A vector field on the line', fontsize=10) # set the title

# set the x and y limits:
ax.set_xlim(-1,3)
ax.set_ylim(-4,4)

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
↪and colors:
ax.plot(x, xdot, color='blue', linestyle='-', label='a sine curve')

# change the axis labels to correspond to [0, pi/2, pi, 1.5 * pi, 2*pi, 2.5*pi, 3*pi]
ax.set_xticks([])
ax.set_yticks([])

# plot the fixed points:
xfpu = [1.5396]
yfpu = [0]
plt.scatter(xfpu, yfpu, s=80, facecolors='none', edgecolors='r', zorder=11)

# plot the fixed points:
xfps = [-0.32941]
yfps = [0]
plt.scatter(xfps, yfps, s=80, facecolors='r', edgecolors='r', zorder=11)


# the axes:
ax.set_aspect(0.2)
ax.axhline(y=0, color='k', alpha=0.5)
ax.axvline(x=0, color='k')

# Vector origin location
X = [1.5396, 1.539, -0.32941-0.8]
Y = [0, 0, 0]

# Directional vectors
U = [2, -2, 0.32941+0.9]
V = [0, 0,  0]
# Creating plot
plt.quiver(X, Y, U, V, color='black', units='xy', scale=1, zorder=10, width=0.035)

plt.show() # show the plot here
```

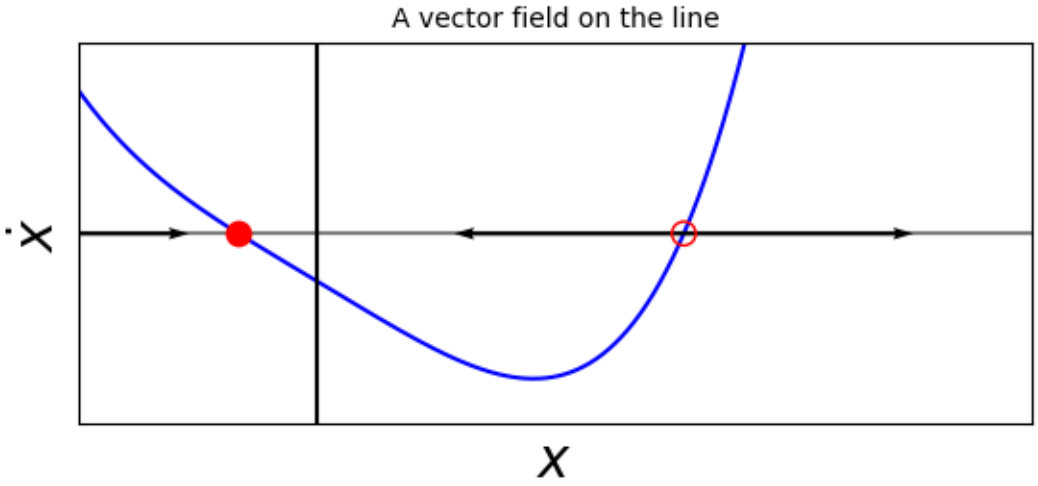



```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

x = np.linspace(-2, 2, 600) #
xdot = x**4 - 3*x - 1

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$\dot{x}$', fontsize=20) # set the y label
ax.set_title('A vector field on the line', fontsize=10) # set the title

# set the x and y limits:
ax.set_xlim(-1,3)
ax.set_ylim(-4,4)

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
 ↪and colors:
ax.plot(x, xdot, color='blue', linestyle='-', label='a sine curve')

# change the axis labels to correspond to [0, pi/2, pi, 1.5 * pi, 2*pi, 2.5*pi, 3*pi]
ax.set_xticks([])
ax.set_yticks([])

# plot the fixed points:
xfpu = [1.5396]
yfpu = [0]
plt.scatter(xfpu, yfpu, s=80, facecolors='none', edgecolors='r', zorder=11)

# plot the fixed points:
xfps = [-0.32941]
yfps = [0]
plt.scatter(xfps, yfps, s=80, facecolors='r', edgecolors='r', zorder=11)


# the axes:
ax.set_aspect(0.2)
```

(continues on next page)

(continued from previous page)

```
ax.axhline(y=0, color='k', alpha=0.5)
ax.axvline(x=0, color='k')

# Vector origin location
X = [1.5396, 1.539, -0.32941-0.8]
Y = [0, 0, 0]

# Directional vectors
U = [2, -2, 0.32941+0.9]
V = [0, 0,  0]
# Creating plot
plt.quiver(X, Y, U, V, color='black', units='xy', scale=1, zorder=10, width=0.035)

plt.show() # show the plot here
```




This imaginary fluid is called the *phase fluid*, and the real line is the *phase space*.

To find a solution to $\dot{x} = f(x)$ starting from an arbitrary initial condition $x_0$, we place an imaginary particle (a *phase point*) at $x_0$ and watch how it is carried along by the flow. As time goes on, the phase point moves along the $x$-axis according to some function $x(t)$. This function is called the *trajectory* based at $x_0$, and it represents the solution to the differential equation starting from $x_0$. A picture like the one above, which shows qualitatively all the trajectories of the system, is called a *phase portrait*.

The appearance of the phase portrait is controlled by the fixed points $x^*$, defined by $f(x^*) = 0$. These are stagnation points of the flow. Solid dots are stable fixed points (the local flow is towards them), and open dots are unstable fixed points (the flow is away from them).

In terms of the differential equation, the fixed points represent equilibrium solutions. An equlibrium is defined to be stable if all sufficiently small disturbances away from it damp out in time. Conversely, unstable equilibria, which disturbances grow in time, are represented by unstable fixed points.

### 8.3.2 Example 8.1: Find all the fixed points for \dot{x} = x^2 -1, and classify their stability.

We note here that the definition of stable equilibrium is based on *small* disturbances: large disturbances may fail to decay. To emphasize this aspect of stability, we may characterize a fixed point as *locally stable* and not globally stable.

## 8.4 Linear Stability Analysis

We would like to have a more quantitative measure of stability, such as the rate of decay to a stable fixed point. To achieve this, we *linearize* about a fixed point.

Let $x^*$ be a fixed point and consider a small perturbation $\eta(t)$ away from $x^*$: $x(t) = x^* + \eta(t)$. Evidently, since $x^*$ is a constant:

$\dot{\eta} = \dot{x}$, and so: $\dot{\eta} = f(x) = f(x^* + \eta)$.

If we now Taylor-expand the right-hand side:

$\dot{\eta} = f(x^*) + \eta f'(x^*) + \mathcal{O}(\eta^2)$,

where $' \equiv \mathrm{d}/\mathrm{d}x$.

Since $f(x^*) = 0$ for the fixed point $x^*$, and neglecting the terms $\mathcal{O}(\eta^2)$, we have:

$\dot{\eta} \approx \eta f'(x^*)$.

This is a linear equation in $\eta$, known as the *linearization* about $x^*$. It shows that the perturbation grows exponentially if $f'(x^*) > 0$, and decays if $f'(x^*) < 0$, since:

$\eta(t) \approx \eta(0) \exp(f'(x^*)t)$.

Therefore, the slope, $f'(x^*)$ at the fixed point determines its stability.

The value $|1/f'(x^*)|$ is a characteristic time scale, and determines the time required for $x(t)$ to vary significantly in the neighborhood of $x^*$.

### 8.4.1 Example 8.2: Use linearization to determine the stability of the fixed points for \dot{x} = \sin x.

## 8.5 Two-Dimensional Systems

Let's now consider the simplest class of a higher-dimensional system, one in two dimensions (2D). We will start with *linear* systems, which are interesting in their own right, but play an important role in the classification of fixed points of nonlinear systems.

### 8.5.1 2D Linear Systems

A 2D linear system is defined by:

$\dot{x} = ax + by$,

$\dot{y} = cx + dy$.

This can be written compactly in matrix form as:

$\dot{\mathbf{x}} = A\mathbf{x}$,

where $\mathbf{x} = (xy)$, and $A = (abcd)$.

In this case, $\dot{\mathbf{x}} = \mathbf{0}$ when $\mathbf{x} = \mathbf{0}$, and therefore $\mathbf{x}^* = \mathbf{0}$ for any choice of $A$.

The solutions of $\dot{\mathbf{x}} = A\mathbf{x}$ can be visualized as trajectories moving on the $(x, y)$ plane, in this context called the *phase plane*.

### 8.5.2 Example 8.3: Analysis of the Simple Harmonic Oscillator.

Vibrations of a mass hanging from a linear spring are governed by the linear differential equation:

$m\ddot{x} + kx = 0$, where $m$ is the mass, $k$ is the spring constant, and $x$ is the displacement of the mass from equilibrium.

Give a phase plane analysis of this simple harmonic oscillator.

### 8.5.3 Classification of Linear Systems

Let's discuss a more general approach of the classification of linear systems, by examining an example system:

$\dot{\mathbf{x}} = A\mathbf{x}$,

where: $A = (a00 - 1)$.

Multiplying out:

$\dot{x} = ax$,

$\dot{y} = -y$.

Therefore, the equations are *uncoupled*: there's no $x$ in the $y$-equation and vice versa. In this simple case, the equation may be solved separately:

$x(t) = x_0 e^{at}$,

$y(t) = y_0 e^{-t}$.

The phase portraits are shown below for different values of the parameter $a$. In each case, $y(t)$ decays exponentially. When $a < 0$, $x(t)$ also decays exponentially, and so all trajectories approach the origin as $t \to \infty$. However, the direction of approach depends on the size of $a$ compared to -1.

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

# the parameter a:
a = -2

# the initial condition combinations:
x0 = [-2, 2, 0, 0, 2, 2, 2, 2, 2, 2, -2, -2, -2, -2, -2, -2]
y0 = [0, 0, 2, -2, 3, 2, 1, -3, -2, -1, 3, 2, 1, -3, -2, -1]
```

(continues on next page)

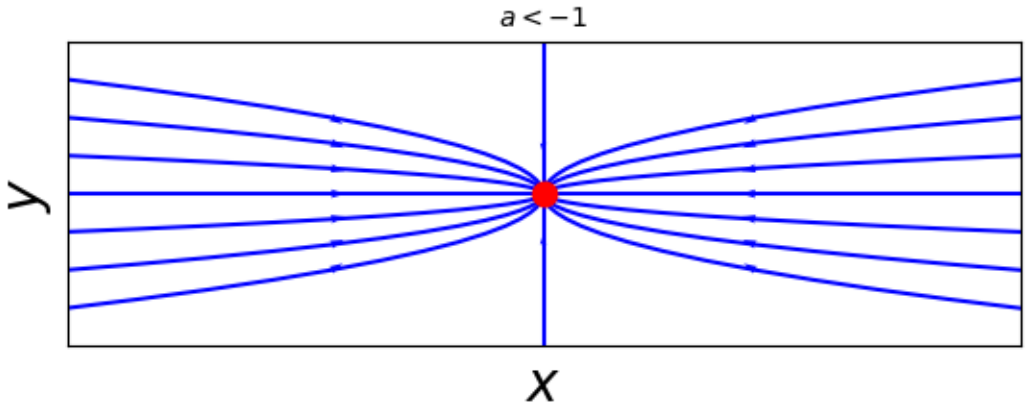
(continued from previous page)

```
#t = np.concatenate((np.linspace(-2, -0.1,100),np.geomspace(-0.1, 1E-6, 1000)))
#t = np.concatenate((t,np.geomspace(1E-6, 0.1, 1000)))
#t = np.concatenate((t,np.linspace(-2, 2,100)))

t = np.logspace(-4,1,100)

x = [xz*np.exp(a*t) for xz in x0]
y = [yz*np.exp(-t) for yz in y0]



fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$y$', fontsize=20) # set the y label
ax.set_title('$a<-1$', fontsize=10) # set the title

# set the x and y limits:
ax.set_xlim(-0.5,0.5)
ax.set_ylim(-2.0,2.0)

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
 ↪and colors:
for i in range(len(x)):
    a1 = 80
    fac = 0.06
    fac2 = 0.045
    wa = 0.04
    #print(x[i][a1], y[i][a1])
    ax.plot(x[i], y[i], color='blue', linestyle='-')
    plt.quiver(x[i][a1], y[i][a1], fac*a*x0[i]*np.exp(a*t[i])/np.sqrt( (a*x0[i]*np.
 ↪exp(a*t[i]))**2 + (y0[i]*np.exp(-t[i]))**2) , -fac2*y0[i]*np.exp(-t[i])/np.sqrt(␣
 ↪(a*x0[i]*np.exp(a*t[i]))**2 + (y0[i]*np.exp(-t[i]))**2), color='b', units='xy',␣
 ↪scale=1, zorder=10, width=wa)


# change the axis labels to correspond to [0, pi/2, pi, 1.5 * pi, 2*pi, 2.5*pi, 3*pi]
ax.set_xticks([])
ax.set_yticks([])

# plot the fixed points:
#xfpu = [1]
#yfpu = [0]
#plt.scatter(xfpu, yfpu, s=80, facecolors='none', edgecolors='r', zorder=11)

# plot the fixed points:
xfps = [0]
yfps = [0]
plt.scatter(xfps, yfps, s=80, facecolors='r', edgecolors='r', zorder=11)


# the axes:
```

(continues on next page)

(continued from previous page)

```
ax.set_aspect(0.08)
#ax.axhline(y=0, color='k', alpha=0.5)
#ax.axvline(x=0, color='k')

plt.show() # show the plot here
```



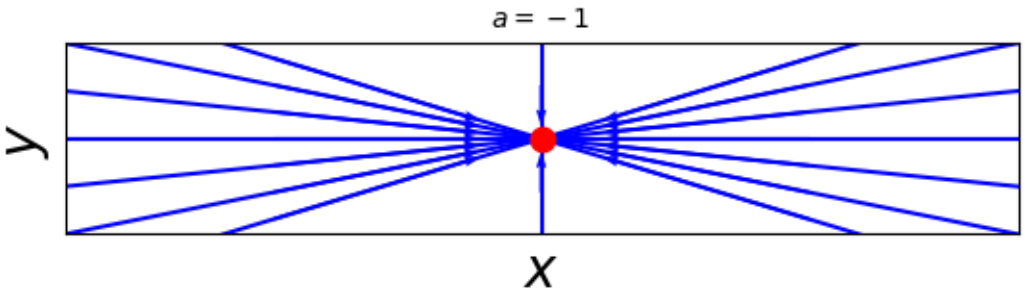

For $a < -1$, $x(t)$ decays more rapidly than $y(t)$. The trajectories approach the origin tangent to the slower direction. $\mathbf{x}^* = \mathbf{0}$ is a *stable node*.

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

# the parameter a:
a = -1

# the initial condition combinations:
x0 = [-2, 2, 0, 0, 2, 2, 2, 2, 2, 2, -2, -2, -2, -2, -2, -2]
y0 = [0, 0, 2, -2, 3, 2, 1, -3, -2, -1, 3, 2, 1, -3, -2, -1]

t = np.logspace(-4,1,100)

x = [xz*np.exp(a*t) for xz in x0]
y = [yz*np.exp(-t) for yz in y0]

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$y$', fontsize=20) # set the y label
ax.set_title('$a=-1$', fontsize=10) # set the title

# set the x and y limits:
ax.set_xlim(-0.5,0.5)
ax.set_ylim(-0.5,0.5)

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
↪and colors:
for i in range(len(x0)):
    a1 = 85
    fac = 5
```

(continues on next page)

(continued from previous page)

```
    wa = 0.005
    #print(x[i][a1], y[i][a1])
    ax.plot(x[i], y[i], color='blue', linestyle='-')
    plt.quiver(x[i][a1], y[i][a1], fac*(x[i][a1+1]-x[i][a1]), y[i][a1+1]-y[i][a1],␣
↪color='b', units='xy', scale=1, zorder=10, width=wa)


# change the axis labels to correspond to [0, pi/2, pi, 1.5 * pi, 2*pi, 2.5*pi, 3*pi]
ax.set_xticks([])
ax.set_yticks([])

# plot the fixed points:
#xfpu = [1]
#yfpu = [0]
#plt.scatter(xfpu, yfpu, s=80, facecolors='none', edgecolors='r', zorder=11)

# plot the fixed points:
xfps = [0]
yfps = [0]
plt.scatter(xfps, yfps, s=80, facecolors='r', edgecolors='r', zorder=11)


# the axes:
ax.set_aspect(0.2)
#ax.axhline(y=0, color='k', alpha=0.5)
#ax.axvline(x=0, color='k')

plt.show() # show the plot here
```




When $a = -1$, all trajectories are straight lines through the origin. This occurs because the decay rates in the two directions are precisely equal. In this case $\mathbf{x}^*$ is called a symmetrical node or a *star*.

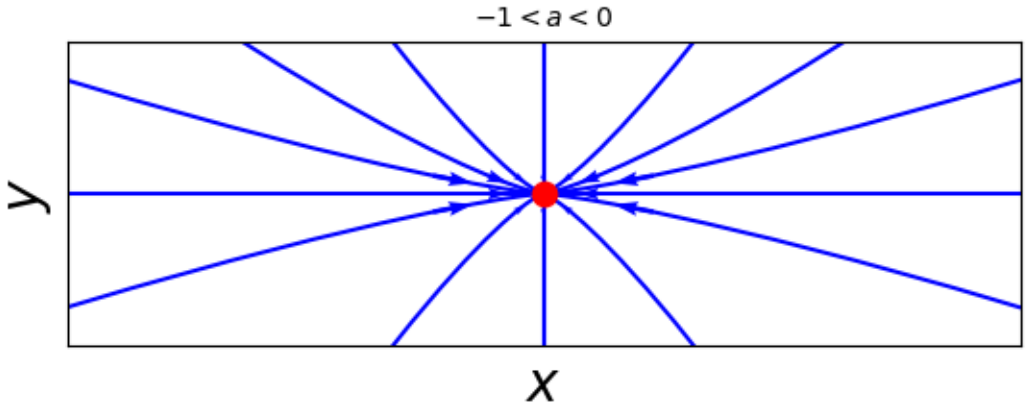
```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

# the parameter a:
a = -0.8

# the initial condition combinations:
x0 = [-2, 2, -2, 2, 0, 0, -4, 4, -1, 1, 0, 0, -4, 4, -1, 1]
y0 = [0, 0, 20, 20, 20, 20, 20, 20, 20, 20, -20, -20, -20, -20, -20, -20]



#t = np.concatenate((np.linspace(-2, -0.1,100),np.geomspace(-0.1, 1E-6, 1000)))
```

(continues on next page)

(continued from previous page)

```
#t = np.concatenate((t,np.geomspace(1E-6, 0.1, 1000)))
#t = np.concatenate((t,np.linspace(-2, 2,100)))

t = np.logspace(-4,1,100)

x = [xz*np.exp(a*t) for xz in x0]
y = [yz*np.exp(-t) for yz in y0]



fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$y$', fontsize=20) # set the y label
ax.set_title('$-1<a<0$', fontsize=10) # set the title

# set the x and y limits:
ax.set_xlim(-0.5,0.5)
ax.set_ylim(-2.0,2.0)

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
 ↪and colors:
for i in range(len(x)):
    a1 = 92
    fac = 4
    fac2 = 1.5
    wa = 0.02
    #print(x[i][a1], y[i][a1])
    ax.plot(x[i], y[i], color='blue', linestyle='-')
    plt.quiver(x[i][a1], y[i][a1], fac*(x[i][a1+1]-x[i][a1]), fac2*y[i][a1+1]-
 ↪y[i][a1], color='b', units='xy', scale=1, zorder=10, width=wa)

# change the axis labels to correspond to [0, pi/2, pi, 1.5 * pi, 2*pi, 2.5*pi, 3*pi]
ax.set_xticks([])
ax.set_yticks([])

# plot the fixed points:
#xfpu = [1]
#yfpu = [0]
#plt.scatter(xfpu, yfpu, s=80, facecolors='none', edgecolors='r', zorder=11)

# plot the fixed points:
xfps = [0]
yfps = [0]
plt.scatter(xfps, yfps, s=80, facecolors='r', edgecolors='r', zorder=11)


# the axes:
ax.set_aspect(0.08)
#ax.axhline(y=0, color='k', alpha=0.5)
#ax.axvline(x=0, color='k')

plt.show() # show the plot here
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)␣
 ↪SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel␣
 ↪Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)␣
 ↪instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)␣
 ↪SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel␣
 ↪Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)␣
 ↪instructions.
```



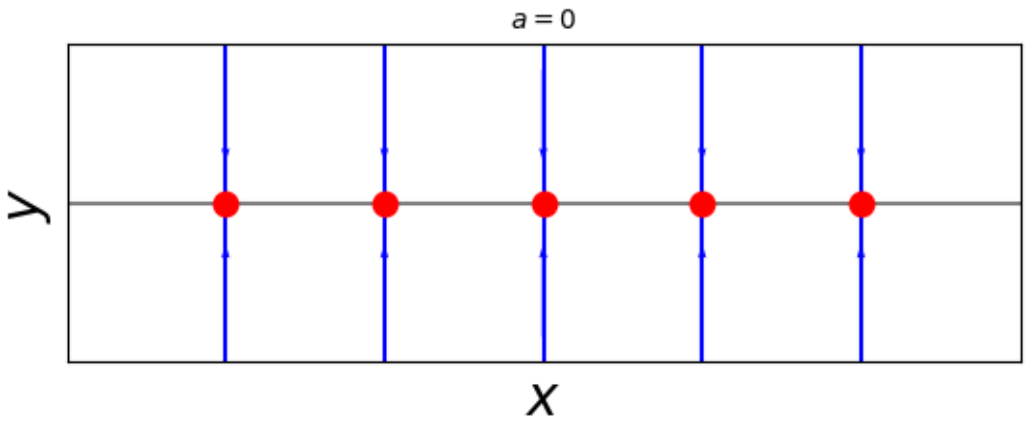

When $-1 < a < 0$, we again have a stable node, but now the trajectories approach $\mathbf{x}^*$ along the $x$-direction, which is the more slowly decaying direction.

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
import numpy as np

# the parameter a:
a = 0

# the initial condition combinations:
x0 = [-2, -1, 0, 1, 2, -2, -1, 0, 1, 2]
y0 = [20, 20, 20, 20, 20, -20, -20, -20, -20, -20]

#t = np.concatenate((np.linspace(-2, -0.1,100),np.geomspace(-0.1, 1E-6, 1000)))
#t = np.concatenate((t,np.geomspace(1E-6, 0.1, 1000)))
#t = np.concatenate((t,np.linspace(-2, 2,100)))

t = np.logspace(-4,1,100)

x = [xz*np.exp(a*t) for xz in x0]
y = [yz*np.exp(-t) for yz in y0]

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
 ↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$y$', fontsize=20) # set the y label
ax.set_title('$a=0$', fontsize=10) # set the title

# set the x and y limits:
ax.set_xlim(-3,3)
```

(continues on next page)

(continued from previous page)

```
ax.set_ylim(-2.0,2.0)

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
 ↪and colors:
for i in range(len(x)):
    a1 = 87
    fac = 4
    fac2 = 1.5
    wa = 0.02
    #print(x[i][a1], y[i][a1])
    ax.plot(x[i], y[i], color='blue', linestyle='-')
    plt.quiver(x[i][a1], y[i][a1], 0, fac2*(y[i][a1+1]-y[i][a1]), color='b', units='xy
 ↪', scale=1, zorder=10, width=wa)

# change the axis labels to correspond to [0, pi/2, pi, 1.5 * pi, 2*pi, 2.5*pi, 3*pi]
ax.set_xticks([])
ax.set_yticks([])

# plot the fixed points:
#xfpu = [1]
#yfpu = [0]
#plt.scatter(xfpu, yfpu, s=80, facecolors='none', edgecolors='r', zorder=11)

# plot the fixed points:
xfps = [-2, -1, 0, 1, 2]
yfps = [0, 0, 0, 0, 0]
plt.scatter(xfps, yfps, s=80, facecolors='r', edgecolors='r', zorder=11)


# the axes:
ax.set_aspect(0.5)
ax.axhline(y=0, color='k', alpha=0.5)
#ax.axvline(x=0, color='k')

plt.show() # show the plot here
```




Something dramatic happens when $a = 0$: there's an *entire line* of fixed points along the $x$-axis. All trajectories approach these fixed points along vertical lines.

```
import matplotlib.pyplot as plt # import matplotlib, a conventional module name is plt
```

(continues on next page)

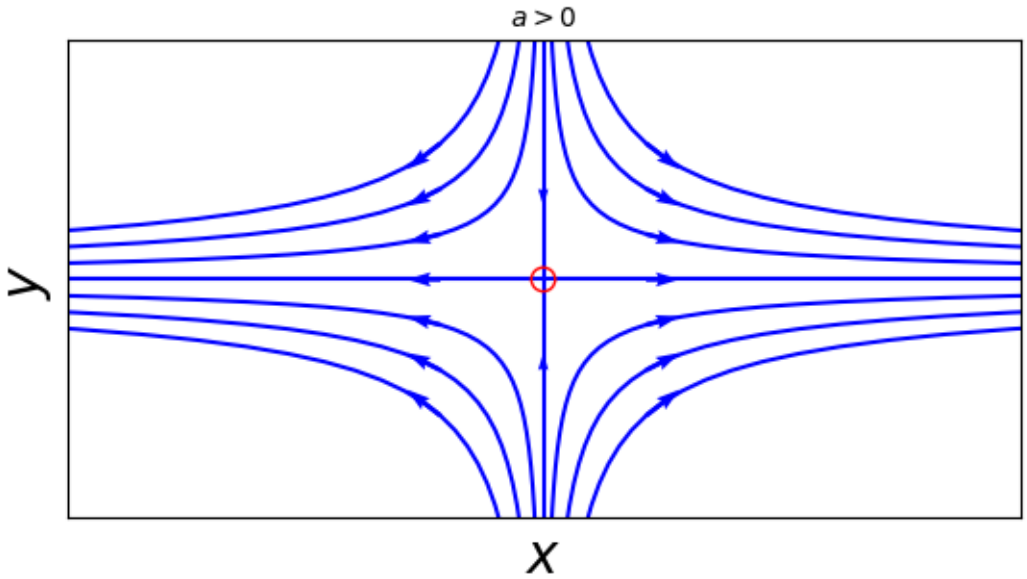
(continued from previous page)

```
import numpy as np

# the parameter a:
a = 1.5

# the initial condition combinations:
x0 = [-2, 2, 0, 0, 2, 2, 2, 2, 2, 2, -2, -2, -2, -2, -2, -2]
y0 = [0, 0, 2, -2, 3, 2, 1, -3, -2, -1, 3, 2, 1, -3, -2, -1]

#t = np.concatenate((np.linspace(-2, -0.1,100),np.geomspace(-0.1, 1E-6, 1000)))
#t = np.concatenate((t,np.geomspace(1E-6, 0.1, 1000)))
#t = np.concatenate((t,np.linspace(-2, 2,100)))

t = np.linspace(-5,5,100)

x = [xz*np.exp(a*t) for xz in x0]
y = [yz*np.exp(-t) for yz in y0]

fig, ax = plt.subplots() # create the elements required for matplotlib. This creates␣
↪a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x$', fontsize=20) # set the x label
ax.set_ylabel(r'$y$', fontsize=20) # set the y label
ax.set_title('$a>0$', fontsize=10) # set the title

# set the x and y limits:
ax.set_xlim(-10,10)
ax.set_ylim(-5.0,5.0)

# make one-dimensional plots using the above arrays, add a custom label, linestyles␣
↪and colors:
for i in range(len(x)):
    a1 = 50
    fac = 2
    wa = 0.10
    ax.plot(x[i], y[i], color='blue', linestyle='-')
    plt.quiver(x[i][a1], y[i][a1], fac*(x[i][a1+1]-x[i][a1]), fac*(y[i][a1+1]-
↪y[i][a1]), color='b', units='xy', scale=1, zorder=10, width=wa)


# change the axis labels to correspond to [0, pi/2, pi, 1.5 * pi, 2*pi, 2.5*pi, 3*pi]
ax.set_xticks([])
ax.set_yticks([])

# plot the fixed points:
xfpu = [0]
yfpu = [0]
plt.scatter(xfpu, yfpu, s=80, facecolors='none', edgecolors='r', zorder=11)

# plot the fixed points:
#xfps = [0]
#yfps = [0]
#plt.scatter(xfps, yfps, s=80, facecolors='r', edgecolors='r', zorder=11)

# the axes:
ax.set_aspect(1)
```

(continues on next page)

(continued from previous page)

```
ax.axhline(y=0, color='b', alpha=0.5)
#ax.axvline(x=0, color='k')

plt.show() # show the plot here
```




Finally, when $a > 0$, $\mathbf{x}^* = \mathbf{0}$ becomes unstable, due to the exponential growth in the $x$-direction. Most trajectories veer away from $\mathbf{x}^*$ and head out to infinity, with the exception of those that start on the $y$-axis. Here $\mathbf{x}^*$ is called a *saddle point*.

In general, if $\mathbf{x}^*$ is an attracting fixed point if all trajectories that start near it approach it as $t \to \infty$. If *all* the trajectories in the phase plane are attracted to it, it is also *globally attracting*

We say that a fixed point $\mathbf{x}^*$ is *Liapunov stable* if all trajectories that start sufficintly close to it, remain close to it for all time. E.g., the case of $a = 0$ in the aboce example is Liapunov stable, but not attractive. When a fixed point is Liapunov stable but not attractive, we say that it is *neutrally stable*. In that case, nearby trajectories are neither attracted nor repelled from a neutrally stable fixed point.

The equilibrium point of a SHO is neutrally stable. Neutral stability is commonly encountered in mechanical systems in the absence of friction.

If a fixed point is Liapunov stable and attracting, we call it *stable* or *asymptotically stable*. If the fixed point is unstable, it can be neither attractive nor Liapunov stable.

Let's study the general case of a two-dimensional linear system i.e. described by a $2 \times 2$ matrix, with the aim of classifying all the possible phase portraits that can occur.

In the example of the preceding section, the $x$- and $y$-axes played a crucial geometric role. They determined the direction of the trajectories as $t \to \pm\infty$. They also contained special straight-line trajectories: a trajectory starting on one of the coordinate axes stayed on that axis forever, and exhibited simple exponential growth or decay along it.

For the general case, we would like to find the analog of these straight line trajectories.

That is, we seek trajectories of the form:

$\mathbf{x}(t) = e^{\lambda t}\mathbf{v}$,

where $\mathbf{v} \neq \mathbf{0}$ is some *fixed* vector to be determined, and $\lambda$ is a growth rate, also to be determined. If such solutions exist, they correspond to exponential motion along the line spanned by the vector $\mathbf{v}$.

To find the conditions on $\mathbf{v}$ and $\lambda$, we substitute $\mathbf{x}(t) = e^{\lambda t}\mathbf{v}$ into $\dot{\mathbf{x}} = A\mathbf{x}$, to obtain:

$\lambda e^{\lambda t}\mathbf{v} = \lambda e^{\lambda t} A\mathbf{v}$,

to obtain:

$\lambda\mathbf{v} = A\mathbf{v}$,

which tells us that the desired straight line solutions exist if $\mathbf{v}$ is an *eigenvector* of $A$ with corresponding *eigenvalue* $\lambda$.

For the case of $A = (abcd)$, the characteristic equation becomes:

$\det(abcd) = 0$,

which yields:

$\lambda^2 + \tau\lambda + \Delta = 0$,

with $\tau = \mathrm{Tr}(A) = a + d$ and $\Delta = \det(A) = ad - bc$.

Then:

$\lambda_{1,2} = \frac{\tau \pm \sqrt{\tau^2 - 4\Delta}}{2}$,

are the solutions of the quadratic equation.

The typical situation is to have distinct eigenvalues $\lambda_1 \neq \lambda_2$, in this case linear algebra tells us that the corresponding eigenvectors $\mathbf{v_1}$ and $\mathbf{v_2}$ are *linearly independent*, and hence they span the entire plane.

For example, any initial condition $\mathbf{x_0}$ can be written as a linear combination of eigenvectors, say:

$\mathbf{x_0} = c_1\mathbf{v_1} + c_2\mathbf{v_2}$. Then, the *general solution* for $\mathbf{x}(t)$ is simply:

$\mathbf{x}(t) = c_1 e^{\lambda_1 t}\mathbf{v_1} + c_2 e^{\lambda_2 t}\mathbf{v_2}$.

(It satisfies the initial condition and it is a linear combination of solutions, hence it is itself a solution.)

### 8.5.4 Example 8.4: Solve the initial value problem \dot{x} = x + y, \dot{y} = 4 x - 2y with initial condition (x_0, y_0) = (2,-3).

Fortunately, we don't need to go through the procedure of Example 8.4 to draw the phase portraot of a linear system: all we need to know are the eigenvalues and eigenvectors:

If $\lambda_1, \lambda_2 < 0$, there exists a stable fixed point at the origin. If on the other hand $\lambda_1, \lambda_2 > 0$, the node is unstable. If one of the eigenvalues is positive and the other negative, the node is a saddle point.

What if the eigenvalues are complex? Then the solution is a *center* or a (stable or unstable) *spiral*.

if the eigenvalues are equal, then we have a *star node*.

(And if there's only one eigenvalue, the fixed point is a *degenerate node*).

## 8.6 The Phase Plane and Phase Portraits

Our goal here of course is to study nonlinear systems. In 2D, the general form of a vector field on the phase plane is given by:

$\dot{x}_1 = f_1(x_1, x_2)$,

$\dot{x}_2 = f_2(x_1, x_2)$,

where $f_{1,2}$ are given functions.

We may also write the system more compactly in vector notation as:

$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$, where $\mathbf{x} = (x_1, x_2)$ and $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}))$.

Here, $\mathbf{x}$ represents a point on the phase plane and $\dot{\mathbf{x}}$ is the velocity vector at that point. By flowing along the vector field, a phase point traces out a solution $\mathbf{x}(t)$, corresponding to a trajectory winding through the phase plane.

Furthermore, the entire phase plane is filled with trajectories, since each point can play the role of an initial condition.

For nonlinear systems, there's typically no hope of finding the trajectories analytically. Even when explicit formulas are available, they are often too complicated to provide much insight. Instead, we wish to determine first the qualitative behavior of the solutions by analytical considerations, and to probe the quantitative aspects numerically.

## 8.7 Fixed Points and Linearization

To examine the nature of fixed points of nonlinear systems, we can apply the *linearization* technique in two dimensions.

Consider the system:

$\dot{x} = f(x, y)$,

$\dot{y} = g(x, y)$,

and suppose that $(x^*, y^*)$ is a fixed point, i.e.:

$f(x^*, y^*) = 0$ and $g(x^*, y^*) = 0$.

Let $u = x - x^*$ and $v = y - y^*$ be small disturbances from the fixed point.

To see whether the disturbance grows or decays, we need to derive differential equations for $u$ and $v$:

$\dot{u} = \dot{x} = f(x^* + u, y^* + v)$.

Taylor-expand the function on the right-hand side about $(x^*, y^*)$ to get:

$\dot{u} = f(x^*, y^*) + u\frac{\partial f}{\partial x} + v\frac{\partial f}{\partial y} + \mathcal{O}(u^2, v^2, uv)$.

Since $f(x^*, y^*) = 0$, we end up with:

$\dot{u} = u\ \frac{\partial f}{\partial x}\Big|_* + v\ \frac{\partial f}{\partial y}\Big|_* + \mathcal{O}(u^2, v^2, uv)$.

Similarly, for $v$, we get:

$\dot{v} = u\ \frac{\partial g}{\partial x}\Big|_* + v\ \frac{\partial g}{\partial y}\Big|_* + \mathcal{O}(u^2, v^2, uv)$,

where the $*$ denotes that the derivatives are evaluated at the fixed point.

Hence, the disturbance $(u, v)$, evolves according to:

$(\dot{u}\dot{v}) = \left(\frac{\partial f}{\partial x}\frac{\partial f}{\partial y}\frac{\partial g}{\partial x}\frac{\partial g}{\partial y}\right)_{(x^*,y^*)} (uv) + \text{quadratic terms}$.

You may recognize: $J = \left(\frac{\partial f}{\partial x}\frac{\partial f}{\partial y}\frac{\partial g}{\partial x}\frac{\partial g}{\partial y}\right)_{(x^*,y^*)}$ as the Jacobian matrix!

Now, since the quadratic terms are tiny, it is tempting to neglect them altogether and obtain a linearized system:

$(\dot{u}\dot{v}) = \left(\frac{\partial f}{\partial x}\frac{\partial f}{\partial y}\frac{\partial g}{\partial x}\frac{\partial g}{\partial y}\right)_{(x^*,y^*)} (uv)$,

which we know how to analyze!

Is it really safe to neglect the quadratic terms? The answer is yes, as long as the fixed point for the linearized system is not a borderline case (center, degenerate node, star or non-isolated fixed point).

If the linearized system predicts a saddle, node, or a spiral, then the fixed point *really* is a saddle, node or spiral for the original nonlinear system.

### 8.7.1 Example 8.5: Consider the system \dot{x} = x + e^{-y} and \dot{y} = -y. First use qualitative arguments to obtain information about the phase portrait. Then, use the Runge-Kutta method to compute several trajectories, and plot them on the phase plane.

## 8.8 The Lorenz Equations and Chaos

We begin our brief introduction to the concept of chaos with the Lorenz equations:

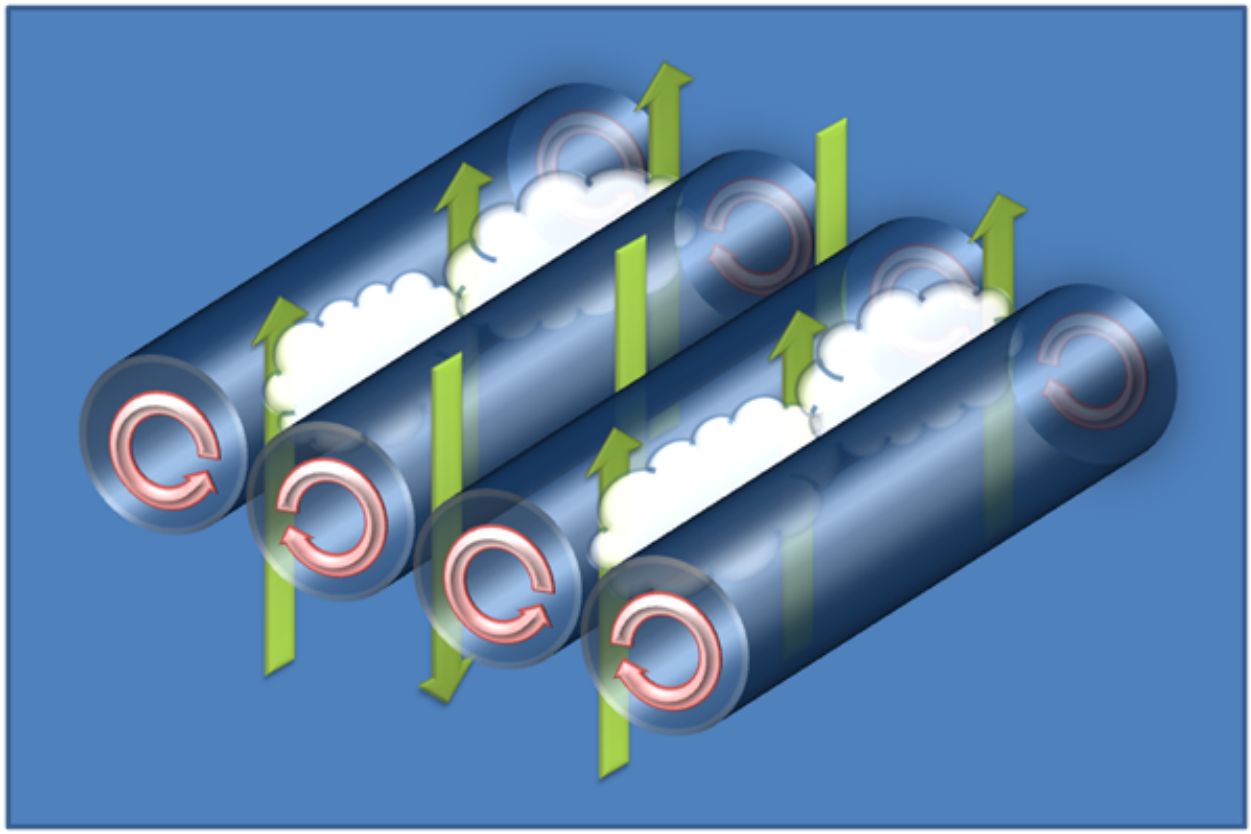
$\dot{x} = \sigma(y - x)$,

$\dot{y} = rx - y - xz$,

$\dot{z} = xy - bz$,

where $\sigma, r, b > 0$ are parameters.

Ed Lorenz (1963) derived this three-dimensional system from a drastically simplified model of convection rolls in the atmosphere.



The same equations also arise in models of lasers and dynamos, and they describe exactly the motion of a certain whater-wheel.

https://en.wikipedia.org/wiki/File:Malkus_Waterwheel_side_by_side.webm

Lorenz discovered that this simple-looking, deterministic system could have extremely erratic dynamics. In particular, over a wide range of parameters, the solutions oscillate irregularly, never exactly repeating, but always remaining in a bounded region of phase space.

When he plotted the trajectories in 3D, he discovered that they settled onto a complicated set, now known as a *strange attractor*, a fractal with dimension between 2 and 3.

## 8.9 Simple Properties of the Lorenz Equations

$\dot{x} = \sigma(y - x)$,

$\dot{y} = rx - y - xz$,

$\dot{z} = xy - bz$.

The $\sigma$ parameter is also known as the Prandtl number, $r$ is known as the Rayleigh number and $b$ has no name (in the convection problem it is related to the aspect ratio of the convection rolls).

The system has two nonlinearities: $xy$ and $xz$. There's also a symmetry: $(x, y) \to (-x, -y)$ which leaves the equations unchanged.

The point $(x^*, y^*, z^*) = (0, 0, 0)$ is a fixed point for all values of the parameters. For $r > 1$, there's also a symmetric pair of fixed points at:

$x^* = y^* = \pm\sqrt{b(r-1)}$, $z^* = r - 1$, called $C^+$ and $C^-$, respectively. As $r \to 1^+$, $C^+$ and $C^-$ coalesce with the origin.

### 8.9.1 Linear Stability of the Origin

Linearizing the Lorenz equations at the origin, we get:

$\dot{x} = \sigma(y - x)$,

$\dot{y} = rx - y$,

$\dot{z} = -bz$.

The equation for $z$ is decoupled and shows that $z(t) \to 0$ exponentially fast. The other two directions are governed by the system:

$(\dot{x}\dot{y}) = (-\sigma\sigma r - 1)\,(xy)$,

with trace $\tau = -\sigma - 1 < 0$ and determinant $\Delta = \sigma(1 - r)$.

if $r > 1$ the origin is a saddle point because $\Delta < 0$. Note that this is a new type of saddle point, since the full system is 3D. If $r < 1$ all the directions are incoming and the origin is a sink and a stable node.

Moreover, it can be shown that for $r < 1$ that every trajectory approaches the origin as $t \to \infty$, and therefore the origin is *globally* stable.

For a range of $r$: $1 < r < r_H = \frac{\sigma(\sigma+b+3)}{\sigma-b-1}$ (assuming $\sigma - b - 1 > 0$), the fixed points $C^\pm$ are linearly stable.

For $r > r_H$, trajectories have a bizarre kind of long-term behavior. Like balls in a pinball machine, they are repelled from one unstable object after another. At the same time, they are confined to a bounded set of zero volume, yet they manage to move on this set forever, without intersecting themselves or others.

### 8.9.2 Exercise 8.1: Chaos on a Strange Attractor

(See file Exercise8.1.ipynb)

### 8.9.3 The Definition of Chaos

No definition of "chaos" is universally accepted, but almost everyone would agree on three ingredients used in the following working definition:

"Chaos is aperiodic long-term behavior in a deterministic system that exhibits sensitive dependence on initial conditions."

1. "Aperiodic long-term behavior" implies that the trajectories do not settle down to fixed points, periodic orbits, or quasiperiodic orbits as $t \to \infty$. For practical reasons, we require that such trajectories are not too rare.
2. "Deterministic" means that the system has no random or noisy inputs or parameters. The irregular behavior of the system arises from the system's nonlinearity, rather than from noisy driving forces.
3. "Sensitive dependence on initial conditions" means that nearby trajectories separate exponentially fast, i.e. the system has positive Liapunov exponent.

## 8.10 Lorenz Attractor Animation

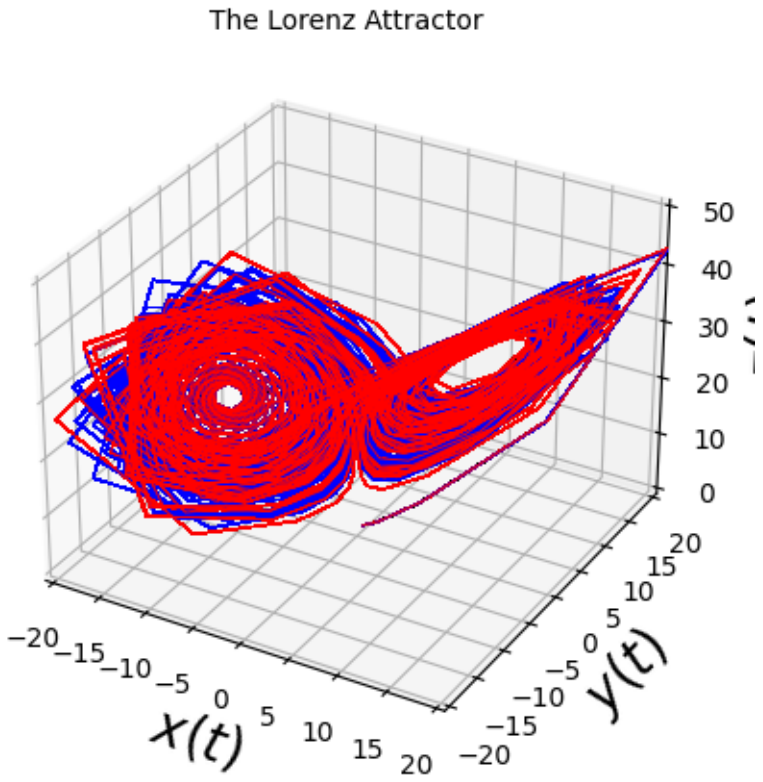
We end this chapter with an animation of the Lorenz Attractor, demonstrating the exponential deviation of two trajectories with similar initial conditions.

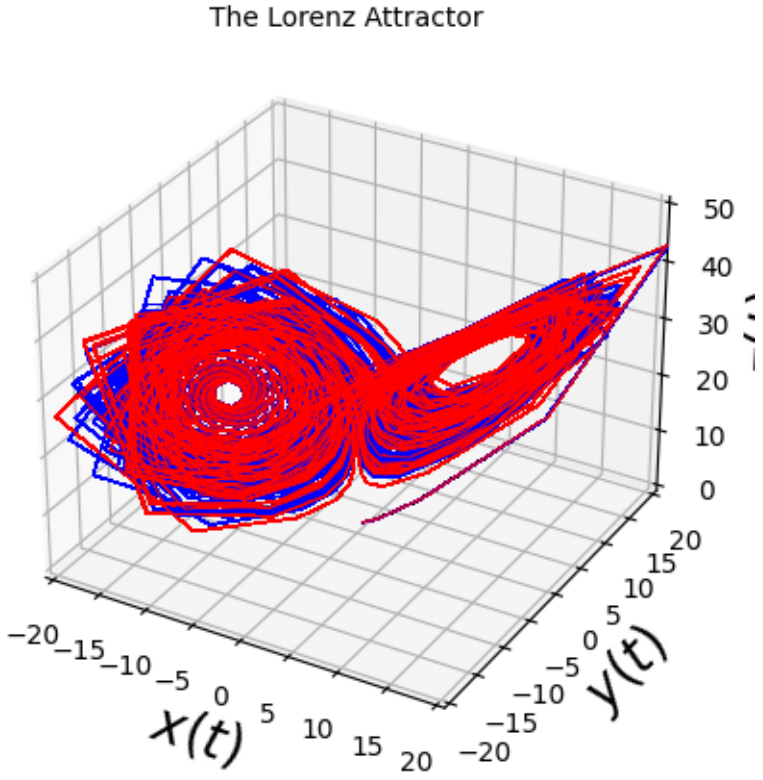
```
import scipy
import numpy as np
from IPython import display
import matplotlib.pyplot as plt

# the name of the function is:
# scipy.integrate.solve_ivp, which by default uses the Runge-Kutta "45" method,
# a modified version of the Runge-Kutta 4-th order algo that has a variable step size.
 ↪

# fix the parameters to have the values that Lorenz used:
sigma = 10
r = 28
b = 8/3

# first define the function vector
# y is a 3D vector here! t is necessary as well, but our function does not depend on␣
 ↪time here
def func(t, y):
    """Returns the function vector"""
    return [sigma * (y[1] - y[0]),
            r * y[0] - y[1] - y[0] * y[2],
            y[0] * y[1] - b * y[2]]

# Now get the solution:
y0 = [0, 1, 0]
solarray = []
tmax = 100
t_eval = np.linspace(0, tmax, 1000)
t_range = (0, tmax)
```

(continues on next page)

(continued from previous page)

```
sol = scipy.integrate.solve_ivp(func, t_range, y0, t_eval=t_eval)
y0 = [0, 1+1E-2, 0]
sol2 = scipy.integrate.solve_ivp(func, t_range, y0, t_eval=t_eval)

# PLOT:

dynamicdisplay = display.display("", display_id=True)

fig, ax = plt.subplots(subplot_kw={"projection": "3d"}) # create the elements␣
↪required for matplotlib. This creates a figure containing a single axes.

# set the labels and titles:
ax.set_xlabel(r'$x(t)$', fontsize=20) # set the x label
ax.set_ylabel(r'$y(t)$', fontsize=20) # set the y label
ax.set_zlabel(r'$z(t)$', fontsize=20) # set the y label

ax.set_title('The Lorenz Attractor', fontsize=10) # set the title

# set the x and y limits:
ax.set_xlim(-20,20)
ax.set_ylim(-20,20)
ax.set_zlim(0,50)


for i, ti in enumerate(sol.t):
    ax.plot(sol.y[0][:i], sol.y[1][:i], sol.y[2][:i], color='blue', linestyle='-',␣
↪marker='o', lw=0.6, ms=0)
    ax.plot(sol2.y[0][:i], sol2.y[1][:i], sol2.y[2][:i], color='red', linestyle='-',␣
↪marker='o', lw=0.6, ms=0)
    dynamicdisplay.update(fig)
```


The Lorenz Attractor
50
40
30
20
10
0
−20
−15
−10
−5
0
5
10
15
20
x(t)
y(t)


The Lorenz Attractor
50
40
30
20
10
0
−20
−15
−10
−5
0
5
10
15
20
x(t)
y(t)

CHAPTER

NINE

# BOUNDARY VALUE AND EIGENVALUE PROBLEMS

## 9.1 Introduction

Many of the important equations of physics can be cast in the form of a linear, second-order differential equation:

$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} + k^2(x)y = S(x)$,

where $S(x)$ is an "inhomogeneous" (or "driving", or "source" term) and $k^2(x)$ is a real function.

When $k^2 > 0$, the solutions of the homogeneous equation (i.e. with $S = 0$) are oscillatory, with wavenumber $k$.

When $k^2 < 0$, the solutions grow or decay exponentially at a rate $\sqrt{-k^2}$.

An example of such a problem is trying to find the electrostatic potential $\Phi$, generated by a localized charge distribution $\rho(\mathbf{r})$.

The starting point would be Poisson's equation: $\nabla^2\Phi = -4\pi\rho$.

(Note that this is written in Gaussian units, where the permittivity of free space is replaced by $1/4\pi$: $\varepsilon_0 \to 1/4\pi$. See https://en.wikipedia.org/wiki/Gaussian_units for further detail.)

For *spherically-symmetric* $\rho$, and hence $\Phi$, the Poisson equation turns into:

$\frac{1}{r^2}\frac{\mathrm{d}}{\mathrm{d}r}\left(r^2\frac{\mathrm{d}\Phi}{\mathrm{d}r}\right) = -4\pi\rho$.

The standard substitution: $\Phi(r) = r^{-1}\phi(r)$ then yields:

$\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2} = -4\pi r\rho$,

which is of the aforementioned form, with $k^2 = 0$ and $S = -4\pi r\rho$.

Another example is the quantum-mechanical wave function for a particle of mass $m$ and energy $E$, moving in a *central* potential $V(r)$. This can be written as:

$\Psi(\mathbf{r}) = r^{-1}R(r)Y_{\ell M}(\theta,\phi)$,

where $Y_{\ell M}$ are the spherical harmonics, and $\ell, M$ are quantum numbers relevant for the angular momentum, and the *radial* wave function satisfies:

$\frac{\mathrm{d}^2R}{\mathrm{d}r^2} + k^2(r)R = 0$,

where $k^2(r) = \frac{2m}{\hbar^2}\left[E - \frac{\ell(\ell+1)}{2mr^2}\hbar^2 - V(r)\right]$.

The above equation is also in the general form stated above, with $S = 0$.

These equations appear unremarkable and readily treated by the methods discussed in Chapter 7, except for two points:

- Boundary conditions imposed by the physics often appear as constraints on the dependent variable at two *separate* points. Therefore, a solution of the problem as an initial value problem is not obviously possible.

- The (time-independent) Schrödinger equation is an eigenvalue problem, in which we must *find* the energies that lead to physically-acceptable solutions satisfying the appropriate boundary conditions.

We begin by deriving an integration algorithm particularly suited to the equations of the desired form, and discuss the boundary value and eigenvalue problems in turn.

## 9.2 The Numerov Algorithm

There is a particularly simple and efficient method for integrating the second-order differential equations of the desired form (i.e. without a first-order derivative). This is known as the Numerov (or Cowling's) method.

Consider the following expansions of the function $y$ about $x$:

$y(x+h) = y(x) + hy'(x) + \frac{h^2}{2}y''(x) + \frac{h^3}{6}y'''(x) + \frac{h^4}{4!}y''''(x) + \mathcal{O}(h^5)$,

$y(x-h) = y(x) - hy'(x) + \frac{h^2}{2}y''(x) - \frac{h^3}{6}y'''(x) + \frac{h^4}{4!}y''''(x) - \mathcal{O}(h^5)$,

Add the two together:

$y(x+h) + y(x-h) = 2y(x) + h^2y''(x) + \frac{h^4}{12}y''''(x) + \mathcal{O}(h^6)$,

since the $\mathcal{O}(h^5)$ terms will cancel out due to opposite signs.

After some minor rearrangements, we get a three-point approximation for the second derivative $y''(x)$:

$y''(x) + \frac{h^2}{12}y''''(x) + \mathcal{O}(h^4) = \frac{y(x+h)+y(x-h)-2y(x)}{h^2}$,

where we have kept the error term $\frac{h^2}{12}y''''(x)$, which we will further manipulate below.

Before we get to that, according to the above, let's write down the the three-point approximation for the second derivative:

$y''(x_n) = \frac{y_{n+1}-2y_n+y_{n-1}}{h^2} + \mathcal{O}(h^2)$,

where we have defined the $n$-th iteration of the independent variable as $x_n = x_0 + nh$.

The differential equation itself tells us that:

$y'' = -k^2y + S$.

If we differentiate this twice more and evaluate it at $x_n$:

$y''''(x_n) \equiv y''''_n = \frac{\mathrm{d}^2}{\mathrm{d}x^2}\left(-k^2y + S\right)\Big|_{x=x_n}$.

We can then use the three-point approximation of the second derivative derived above to deduce that: $y''''_n = -\frac{(k^2y)_{n+1}-2(k^2y)_n+(k^2y)_{n-1}}{h^2} + \frac{S_{n+1}-2S_n+S_{n-1}}{h^2} + \mathcal{O}(h^2)$.

This can be substituted into the "error" term of the three-point approximation for the second derivative $y''(x)$ to get:

$y''_n + \frac{h^2}{12}\left(-\frac{(k^2y)_{n+1}-2(k^2y)_n+(k^2y)_{n-1}}{h^2} + \frac{S_{n+1}-2S_n+S_{n-1}}{h^2} + \mathcal{O}(h^2)\right) + \mathcal{O}(h^4) = \frac{y_{n+1}-2y_n+y_{n-1}}{h^2}$.

Multiplying through by $h^2$ and using the differential equation itself, i.e. $y''_n = -(k^2y)_n + S_n$:

$-(k^2y)_n + S_n + \frac{h^2}{12}\left(-(k^2y)_{n+1} + 2(k^2y)_n - (k^2y)_{n-1} + S_{n+1} - 2S_n + S_{n-1}\right) + \mathcal{O}(h^6) = y_{n+1} - 2y_n + y_{n-1}$.

After further rearrangement, we get a recursion relation for $y_{n+1}$ in terms of $y_n$ and $y_{n-1}$, the previous two steps:

$y_{n+1} = \frac{2(1-\frac{5h^2}{12}k_n^2)y_n-(1+\frac{h^2}{12}k_{n-1}^2)y_{n-1}+\frac{h^2}{12}(S_{n+1}+10S_n+S_{n-1})+\mathcal{O}(h^6)}{1+\frac{h^2}{12}k_{n+1}^2}$.

Note that one can also solve for $y_{n-1}$ in terms of $y_n$ and $y_{n+1}$ to integrate backward in $x$.

We emphasize here that the Numerov method is one more order accurate than the fourth-order Runge-Kutta method, which might be used to integrate the problem as two coupled first-order equations. The Numerov method is also more efficient, as each step requires computation of $k^2$ and $S$ at only the "lattice" points.

### 9.2.1 Example 9.1: Apply the Numerov algorithm to the Simple Harmonic Oscillator problem:

$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} = -4\pi^2 y$, with $y(0) = 1$, $y'(0) = 0$.

Integrate from $x = 0$ to $x = 4$. Note that you will have to use some special procedure to generate the value of $y_1 \equiv y(h)$, needed to start the three-term recursion relation.

Plot the resulting function $y(x)$.

## 9.3 Direct Integration of Boundary Value Problems: the Poisson Equation

Consider trying to solve Poisson's equation for a charge density distribution:

$\rho(r) = \frac{1}{8\pi} e^{-r}$.

If we integrate the charge density over all space, we obtain the total charge:

$Q = \int \rho(r) \mathrm{d}^3 r = 4\pi \int_0^\infty \rho(r) r^2 \mathrm{d}r = 1$.

The exact solution to this problem is:

$\phi(r) = 1 - \frac{1}{2}(r+2)e^{-r}$,

and therefore we can obtain the potential $\Phi = \phi/r$.

The solution has the expected behavior at large $r$: since $\phi \to 1$, we have $\Phi \to 1/r$, the Coulomb potential from a unit charge (*in Gaussian units*).

Let's try to solve this example as an ordinary initial value problem. The charge density $\rho$ has no singular behavior at the origin, and therefore we expect $\Phi$ to be regular there, which implies that $\phi = r\Phi$ vanishes at the origin, when $r = 0$. We can readily check that this is indeed the case for the explicit solution:

$\phi(0) = 1 - \frac{1}{2}(0+2)e^0 = 0$.

We could then integrate $\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2} = -4\pi r\rho$ *outward* from the origin using the Numerov method:

$\phi_{n+1} = 2\phi_n - \phi_{n-1} + \frac{h^2}{12}(S_{n+1} + 10S_n + S_{n-1})$,

with source term $S = -4\pi r\rho = -\frac{1}{2} r e^{-r}$.

However, to be able to accomplish this, we also need to know the value of $\phi_1 = \phi(h)$ (or equivalently, $\mathrm{d}\phi/\mathrm{d}r$ at $r = 0$) in addition to $\phi_0 = 0$. Note that this is also identical to $\Phi(0)$, since:

$\Phi(0) = \lim_{r\to 0} \frac{\phi}{r} = \left.\frac{\mathrm{d}\phi}{\mathrm{d}r}\right|_{r\to 0}$, after applying L'Hôpital's rule.

This is unfortunate, since $\phi_1$ is nominally part of the function we are trying to find, and we do not know it a priori.

We will discuss below what to do in the general case, but for now, let's just take $\phi_1$ from the analytical solution:

$\phi_1 = \phi(r = h) = 1 - \frac{1}{2}(h+2)e^{-h}$.

### 9.3.1 Example 9.2: Apply the Numerov algorithm to solve Poisson's equation with \rho(r) = \frac{1}{8 \pi} e^{-r}.

Assume the exact solution $\phi(r) = 1 - \frac{1}{2}(r+2)e^{-r}$ and plot the error up to $r = 20$.

To start up the recursion, use $\phi_1 = \phi(r = h) = 1 - \frac{1}{2}(h+2)e^{-h}$.

After solving Example 9.2, you will have noticed that the error is getting larger at large $r$!

Let's try to understand the origin of this phenomenon by considering a more general case, where we do not have an analytical formula to give us $\phi$ near the origin that is necessary to get the three-term recursion relation started.

One way to proceed is to find $\Phi(0)$ by direct numerical quadrature of the Coulomb potential. At a point **r**, the potential will be given by an integral over the whole charge distribution:

$\Phi(\mathbf{r}) = \int \frac{\rho(|\mathbf{r}'|)}{|\mathbf{r}-\mathbf{r}'|} \mathrm{d}^3 r'$.

At the origin ($\mathbf{r} = 0$), this would be easy to calculate:

$\Phi(0) = \int \frac{\rho(|\mathbf{r}'|)}{|\mathbf{r}'|} \mathrm{d}^3 r' = 4\pi \int_0^\infty r' \rho \mathrm{d}r'$, where we have assumed a spherically-symmetric $\rho(r)$.

We could achieve this, e.g. by using Simpson's rule. There will be, however, some error associated with the value obtained. Let's say this error is 5%. Let's just impose such an error in our Example 9.2 result and compare the effect to what we had obtained previously.

### 9.3.2 Example 9.3: Apply a 5% assumed error on the \phi_1 input of Example 9.2 (e.g. by rescaling the exact value by 0.95) and compare your error to the previous one.

You will have noticed that disaster has struck: A 5% change on the initial conditions has induced a 50% error on the solution at large values of $r$!

What's going on?

To understand what has happened, consider solutions to the *homogeneous* version of our differential equation:

$\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2} = 0$.

Such solutions can be *added* to any particular solution to give another solution.

There are two linearly-independent homoeneous solutions:

$\phi \sim r$ and $\phi \sim$ constant.

The general solution to $\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2} = -4\pi r\rho$ in the asymptotic region $r \to \infty$, where $\rho$ vanishes and the equation is in fact homogeneous, can be written as a linear combination of these two functions. Of course, the latter, sub-dominant solution (which corresponds to $\Phi \sim 1/r$) is the physical one.

What has occurred in our problem is that an imprecision in the specification of $\Phi$ at the origin, or any numerical round-off error in the integration process, can introduce a small admixture of the $\phi \sim r$ solution, which will eventually dominate at large $r$.

There exists a straightforward cure this issue: subtract a multiple of the "bad", unphysical solution to the homogeneous equation from the numerical result, to guarantee the physical behavior in the asymptotic region. The "bad" results vary linearly with $r$ for large $r$. We can fit the last few points of the numerical solution to the form:

$\phi = mr + b$,

and subtract $mr$ from the numerical results to reinstate the appropriate large-$r$ behavior. Let's do this in the next example.

### 9.3.3 Example 9.4: Correct the "bad" results with the 5% error of Example 9.3.

The "bad" results vary linearly with $r$ for large $r$. Fit the last few points of the numerical solution to the form:

$\phi = mr + b$,

and subtract $mr$ from the numerical results to reinstate the appropriate large-$r$ behavior.

In this relatively simple example, the instabilities are not too severe, and satisfactory results for moderate values of $r$ are obtained with outward integration when the exact (or reasonably-accurate approximation) value of $\phi_1$ is used.

Alternatively, it is also feasible to integrate inward, starting at large $r$, with $\phi = Q$, independent of $r$. This results in a solution that often satisfies accurately the boundary condition at $r = 0$ and avoids having to perform a quadrature to determine the (approximate) starting value $\phi_1$.

## 9.4 Green's Function Solution of Boundary Value Problems

It's possible that the two solutions to the homogeneous equation (i.e. with the RHS=0) have very different behaviors. In that case, some extra precautions must be taken.

For example, consider the equation describing the potential from a charge distribution of multipole order $\ell > 0$:

$\left[\frac{\mathrm{d}^2}{\mathrm{d}r^2} - \frac{\ell(\ell+1)}{r^2}\right]\phi = -4\pi r\rho$,

which has two homogeneous solutions:

$\phi \sim r^{\ell+1}$ and $\phi \sim r^{-\ell}$.

For large $r$, the first of these solutions is much larger than the second, so that ensuring the correct asymptotic behavior by subtracting a multiple of this dominant homogeneous solution from a particular solution obtained by outward integration is subject to large round-off errors. Inward integration would also be unsatisfactory, since the unphysical solution $r^{-\ell}$ is likely to dominate at small $r$.

One possible way to generate an accurate solution is by combining the two methods: inward integration can be used to obtain the potential for $r$ greater than some intermediate radius $r_m$, and outward integration can be used for the potential when $r < r_m$. As long as $r_m$ is chosen so that neither homogeneous solution is dominant, the outer and inner potentials obtained, respectively from these two integrations will match at $r_m$ and, together, will describe the entire solution. If the inner and outer potentials don't quite match, a multiple of the homogeneous solution can be added to the former to correct for any deficiencies in our knowledge of $\phi'(r = 0)$.

Sometimes the two homogeneous solutions have such different behaviors that it is impossible to find a value of $r_m$ that permits satisfactory integration of the inner and outer potentials. Such cases can be solved by the *Green's function* of the homogeneous equation.

To illustrate this, consider our prototypical equation:

$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} + k^2(x)y = S(x)$,

with boundary conditions $\phi(x = 0) = \phi(x = \infty) = 0$.

Since this is a linear problem, the solution to this equation can be written as:

$\phi(x) = \int_0^\infty G(x, x')S(x')\mathrm{d}x'$,

where $G$ is a Green's function that satisfies:

$\left[\frac{\mathrm{d}^2}{\mathrm{d}x^2} + k^2(x)\right]G(x, x') = \delta(x - x')$.

Evidently, $G$ satisfies the homogeneous equation for $x \neq x'$. However, the derivative of $G$ is discontinuous at $x = x'$ as can be seen by integrating the above equation from $x = x' - \varepsilon$ to $x = x' + \varepsilon$ and letting $\varepsilon \to 0$.

We have $\int \frac{\mathrm{d}^2 G}{\mathrm{d}x^2}\mathrm{d}x = \frac{\mathrm{d}G}{\mathrm{d}x}$, the integral of the $\delta$ function that includes $x = x'$ has to be unity, and the second integral $\int_{x'-\varepsilon}^{x'+\varepsilon} G(x, x')\mathrm{d}x$ vanishes as $\varepsilon \to 0$, we are left with:

$\frac{\mathrm{d}G}{\mathrm{d}x}\Big|_{x'-\varepsilon}^{x'+\varepsilon} = 1.$

Then, to construct the Green function, we first solve the homogeneous equation:

$\left[\frac{\mathrm{d}^2}{\mathrm{d}x^2} + k^2(x)\right] G(x, x') = 0$ for $x > x'$ and $x < x'$ and then impose the appropriate boundary conditions, including the discontinuity in the derivative. Let's recap the process by considering a simple analytical example.

### 9.4.1 Example 9.5: A Simple Analytical Green's Function Problem

Solve the equation:

$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} + y = S(x),$

with boundary conditions $y(0) = y(\pi/2) = 0$, using the Green's function method.

There's a general prescription to obtain the Green's function for our differential equation:

$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} + k^2(x)y = S(x),$

- Get the two solutions to the homogeneous equation that satisfy the two boundary conditions: $y_<$ and $y_>$. The first will satisfy the boundary conditions to the left (e.g. at $x = 0$), and the second to the right (e.g. at $x = \infty$).
- These have to be normalized such that their *Wronskian* equals unity: $W = \frac{dy_>}{\mathrm{d}x}y_< - \frac{dy_<}{\mathrm{d}x}y_> = 1$.
- Then the Green's function is given by: $G(x, x') = y_<(x_<)y_>(x_>)$, where $x_<$ and $x_>$ are the smaller and larger of $x$ and $x'$ respectively. (Check that this conforms with Example 9.5!).
- Then the explicit solution is given by: $y(x) = y_>(x)\int_0^x y_<(x')S(x')\mathrm{d}x' + y_<(x)\int_x^\infty y_>(x')S(x')\mathrm{d}x'$.

In general, this expression can be evaluated by a numerical quadrature and is not subject to any of the stability problems we have seen associated with a direct integration of the inhomogeneous equation.

In the case of arbitrary $k^2$, the homogeneous solutions $y_<$ and $y_>$ can be found numerically by outward and inward integrations, respectively, of initial value problems and then normalized to satisfy the Wronskian relation.

For simple forms of $k^2(x)$, they are known analytically. For example, for the problem:

$\left[\frac{\mathrm{d}^2}{\mathrm{d}r^2} - \frac{\ell(\ell+1)}{r^2}\right]\phi = -4\pi r\rho,$

it can be shown that:

$\phi_<(r) = r^{\ell+1}$ and $\phi_>(r) = -\frac{1}{2\ell+1}r^{-\ell}$, are one possible set of homogeneous solutions satisfying the appropriate boundary conditions and the differential equation.

### 9.4.2 Example 9.6: Solve the problem:

$\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2} = -4\pi r\rho,$

for $\rho(r) = \frac{1}{8\pi}e^{-r}$ using the Green's function method.

Compare your results to the exact solution by plotting the error as in the previous examples.

## 9.5 Eigenvalues of the Wave Equation

Eigenvalue problems involving differential equations often arise in finding normal-mode solutions of wave equations.

We begin our discussion of eigenvalue problems with a simple example: that of normal modes of a stretched string of uniform mass density.

After suitable scaling of the physical quantities, the equation and boundary conditions defining these modes can be written as:

$\frac{\mathrm{d}^2\phi}{\mathrm{d}x^2} = -k^2\phi$,

with $\phi(x=0) = \phi(x=1) = 0$.

Here we have:

- $0 < x < 1$ is the coordinate along the string,
- $\phi$ is the transverse displacement of the string,
- $k$ is the constant wavenumber, linearly related to the frequency of vibration.

The equation is an eigenvalue equation in the sense that the solutions satisfying the boundary conditions exist only for particular values of $k$, $\{k_n\}$, which we must find.

Furthermore, it is linear and homogeneous, the normalization of the eigenfunctions corresponding to any $k_n$, which we denote as $\phi_n$, is not fixed, but can be chosen for convenience.

The un-normalized eigenfunctions and eigenvalues of this problem are of course known analytically:

$k_n = n\pi$, $\phi_n = \sin n\pi x$, where $n$ is a positive integer.

The strategy for solving this problem is an *iterative* one:

- Guess a trial eigenvalue and obtain the general solution by integrating the differential equation as an initial value problem (e.g. using the Numerov method).
- If the resulting solution does not satisfy the boundary conditions, we change the trial eigenvalue and integrate again, repeating the process until a trial eigenvalue is found for which the b.c.'s are satisfied to within a predetermined tolerance.

The above method is known as the "shooting method".

For the problem at hand: for each trial value of $k$, we integrate forward from $x = 0$ with the initial conditions:

$\phi(x=0) = 0$ and $\phi'(x) = \delta$,

where $\delta$ is arbitrary and can be chosen for convenience, since the problem we are solving is a homogeneous one, and the normalization of solutions is not specified.

Upon integrating to $x = 1$, we will find, in general, a non-vanishing value of $\phi$, since the trial eigenvalue will not be one of the true eigenvalues. We must then readjust $k$ and integrate again, repeating the process until we find $\phi(x=1) = 0$ to within a specified tolerance.

The problem of finding a value of $k$ for which $\phi(1)$ vanishes is a *root-finding problem*, such as the ones that we have already discussed (Chapter 6). It is safest to use a simple search to locate an approximate eigenvalue, e.g. the bisection method.

### 9.5.1 Example 9.7: Find the lowest eigenvalue of the stretched string problem by employing the shooting method described above.

Start your search at $k = 1$ and terminate the search when the eigenvalue is determined within a precision of $10^{-5}$.

## 9.6 The One-Dimensional Schrödinger Equation

A rich example of the shooting method for eigenvalue problems is the task of finding the stationary quantum states of a particle of mass $m$ moving in a one-dimensional potential $V(x)$.

The time-independent Schroödinger equation is given by:

$\frac{-\hbar^2}{2m}\frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi(x) + V(x)\psi(x) = E\psi(x)$.

If we rescale the $x$ coordinate by a physical length $a$, then $\mathrm{d}x \to a\mathrm{d}x$, and we can rewrite this as:

$\frac{-\hbar^2}{2ma^2}\frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi(x) + V(x)\psi(x) = E\psi(x)$.

We can then divide LHS and RHS by $V_0$, a characteristic scale of the potential:

$\frac{-\hbar^2}{2ma^2V_0}\frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi(x) + \frac{V(x)}{V_0}\psi(x) = \frac{E}{V_0}\psi(x)$,

to reach the form:

$\left[-\frac{1}{z_0^2}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + v(x) - \epsilon\right]\psi(x) = 0$,

where

$z_0^2 = \frac{2ma^2V_0}{\hbar^2}$ ,

which characterizes the "depth" of the potential, and we have defined the dimensionless energy $\epsilon = E/V_0$.

The equation is then of the form that we have previously addressed:

$\frac{\mathrm{d}^2\psi}{\mathrm{d}x^2} - z_0^2\left[v(x) - \epsilon\right]\psi(x) = 0$,

with $k^2(x) = -z_0^2\left[v(x) - \epsilon\right]$.

If $v(x) < 0$ and $v(x)$ is zero at some boundaries ("walls"), our goal then is to find "bound" solutions with $-1 < \epsilon < 0$, which are localized within the potential and decay exponentially outside.

This eigenvalue problem can be solved by the shooting method. Suppose that we are seeking a bound state, and therefore start with a negative trial eigenvalue.

We can integrate toward larger $x$ via the "forward" Numerov algorithm, from some initial value $x_{\text{min}}$ to obtain the wave function $\psi_<(x)$ (the "left" wave function). However, once we reach the "classically forbidden" region, we will start to generate an admixture of the undesirable exponentially-growing solution. Therefore, as a rule, integration *into* a classically forbidden region is likely to be inaccurate.

Therefore, at each energy, it is wiser to generate a second solution, $\psi_>(x)$ (the "right" wave function), by integrating from $x_{\text{max}}$ toward a smaller $x$, using a "backward" Numerov algorithm.

To determine whether the energy is an eigenvalue, $\psi_<(x)$ and $\psi_>(x)$ can be compared at a matching point $x_m$, chosen so that neither integration will be inaccurate. A convenient choice for $x_m$ is the left turning point.

Since both $\psi_<(x)$ and $\psi_>(x)$ satisfy a homogeneous equation, their normalizations can always be chosen so that the two functions are equal at $x_m$:

$\psi_<(x_m) = \psi_>(x_m)$.

Furthermore, the derivative has to be continuous as well:

$\int_{x_m-\varepsilon}^{x_m+\varepsilon}\mathrm{d}x\frac{\mathrm{d}^2\psi}{\mathrm{d}x^2} = 0$, and so:

$\frac{\mathrm{d}\psi_<}{\mathrm{d}x}\Big|_{\mathrm{x_m}} = \frac{\mathrm{d}\psi_>}{\mathrm{d}x}\Big|_{x_m}$.

If we approximate the derivatives by their simplest forward-difference approximations, i.e.:

$\frac{\mathrm{d}\psi(x)}{\mathrm{d}x} \approx \frac{\psi(x)-\psi(x-h)}{h}$,

then an equivalent condition for the continuity condition of the derivatives is:

$f = \frac{\psi_<(x_m-h)-\psi_>(x_m-h)}{\phi} \approx 0$,

where $\phi$ is a normalization factor, chosen to to make $f$ typically of order unity, e.g. $\phi$ could be the maximum value of $\psi_<$ or $\psi_>$.

Note that if there are no turning points, then $x_m$ can be chosen anywhere, while if there are more than two turning points, three or more homogeneous solutions, each accurate in different regions, must be patched together.

We will tackle the particular problem of the potential of the finite square well in Exercise 9.2:

$$V(x) = \begin{cases} -V_0 & \text{if } -a \leq x \leq a; \\ 0 & \text{if } |x| > a. \end{cases}$$

CHAPTER

TEN

# PARTIAL DIFFERENTIAL EQUATIONS

## 10.1 Introduction

Partial differential equations (PDEs) are involved in the description of virtually every physical situation where quantities vary in space, or in space and time.

Examples are diffusion, electromagnetic waves, hydrodynamics, quantum mechanics.

In all but the simplest cases, equations cannot be solved analytically, and so numerical methods must be employed for quantitative results.

The typical numerical approach proceeds as follows: the dependent variables (e.g. temperature, electrical potential), are described by their values at discrete points (a lattice) of the independent variables (e.g. space and time).

Therefore, by appropriate discretization, the PDE is reduced to a large set of *difference equations*.

Although difference equations can be solved by matrix methods, the large size of the matrices involved (dimension $\sim$ number of lattice points), makes this approach impractical.

However, the locality of the original equations (i.e. they involved only low-order derivatives of the dependent variables) makes the resulting differential equations "sparse", which implies that most of the elements of the matrices involved vanish.

For such matrices, iterative methods of inversion and diagonalization can be very efficient.

Most of the physically-important PDEs are of second order, and can be classified into three types:

1. *Parabolic:* Roughly speaking, parabolic equations involve only a first-order derivative in one variable, but have second-order derivatives in the remaining variables. Examples the diffusion equation, or the time-dependent Schrödinger equation: they are first order in time, but second order in space.
2. *Elliptic:* They have second order derivatives in each of the independent variables, each derivative having the *same sign* when all terms of the equation are grouped on one side. Examples are the Poisson equation for the electrical potential and the time-independent Schrödinger equation, in two or more spatial variables.
3. *Hyperbolic:* They involve second derivatives of opposite sign, e.g. the wave equation describing the vibrations of a stretched string.

In this chapter, we will discuss some numerical methods appropriate for *elliptic* equations and then focus on *parabolic* equations. Hyperbolic equations often can be treated by similar methods, with some unique differences (we won't discuss these here).

## 10.2 Elliptic Partial Differential Equations

For concreteness, we will consider particular forms of elliptic boundary value and eigenvalue problems for a field $\phi$ in two spatial dimensions, $(x, y)$.

Specifically, we will tackle the boundary value problem:

$-\left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right] \phi = S(x, y).$

Although this is not the most general elliptic form, it covers a wide variety of situations. For example, in electrostatics, $\phi$ is the potential and $S$ is related to the charge density (i.e., it is a source term). In steady-state heat diffusion, $\phi$ is the temperature, $S$ is the local rate of heat generation or loss.

The discussion can be generalized straightforwardly to other elliptic cases, e.g. in three dimensions.

Of course, the equation by itself is not complete. The boundary conditions are required. We will take the boundary conditions to be of the "Dirichlet" type, i.e. $\phi$ is specified on some closed curve on the $(x, y)$ plane. Conveniently, we will take this to be the unit square, and perhaps some additional curves within it.

The boundary value problem is then to use the PDE to find $\phi$ everywhere within the square.

Other classes of boundary conditions are the "Neumann" type, where the normal derivative of $\phi$ is specified on the surfaces, or the "mixed" type, where a linear combination of $\phi$ and its normal derivatives is specified. Both can be handled by very similar methods.

The eigenvalue problems we will be interested in might involve an equation of the form,

$-\left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right] \phi + V(x, y)\phi = \varepsilon\phi,$

together with a set of Dirichlet boundary conditions.

As an example, the above could be the time-independent Schrödinger equation, with $\phi$ being the wave function, $V(x, y)$ is related to the potential and $\varepsilon$ is related to the energy eigenvalue.

The eigenvalue problem is then to find the values $\varepsilon_\lambda$ and the associated eigenfunctions $\phi_\lambda$, for which the equation and the b.c.'s are satisfied.

### 10.2.1 Discretization

Let's first cast the equation:

$-\left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right] \phi = S(x, y).$

in a form suitable for numerical treatment.

We define a set of lattice points covering the region of interest in the $(x, y)$ plane.

For convenience, we take the lattice spacing $h$ to be uniform and equal in both directions. Therefore, the unit square is covered by $N \times N$ lattice squares, with points labeled by $(i, j)$, each of which runs from $0$ to $N$. The coordinates of point $(i, j)$ are $x_i = ih$ and $y_j = jh$.

We then define $\phi_{ij} = \phi(x_i, y_j)$ and $S_{ij} = S(x_i, y_j)$.

It is straightforward to apply a three-point difference for the second derivative (see, e.g. Chapter 9, section 2 for a derivation using a Taylor series expansion in one dimension):

$\frac{\partial^2\phi}{\partial x^2} \simeq \frac{\phi_{i+1j} + \phi_{i-1j} - 2\phi_{ij}}{h^2},$

to get:

$-\left[\frac{\phi_{i+1j} + \phi_{i-1j} - 2\phi_{ij}}{h^2} + \frac{\phi_{ij+1} + \phi_{ij-1} - 2\phi_{ij}}{h^2}\right] = S_{ij}.$

### 10.2.2 Variational Principle Approach

We will be solving this equation shortly, but first let's derive it in an alternate way, based on a *variational principle*.

This approach is handy in cases where the coordinates are not Cartesian, or when more accurate formulas are needed.

The variational principle also provides some insnight into how the solution algorithm works.

Consider a quantity $\mathcal{E}$, defined to be a *functional* of the field $\phi$, of the form:

$\mathcal{E} = \int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y \, [\frac{1}{2}(\nabla\phi)^2 - S\phi]$.

In some situations, $\mathcal{E}$ has a physical interpretation. For example, in electrostatics, $E = -\nabla\phi$ is the electric field and $S$ is the charge density, making $\mathcal{E}$ the total energy of the system.

In other situations, such as steady-state diffusion, $\mathcal{E}$ should be viewed simply as a useful quantity.

At a solution to $-\left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right]\phi = S(x,y)$, $\mathcal{E}$ is *stationary* under all variations of the solution $\phi$, called $\delta\phi$ that respect the Dirichlet boundary conditions imposed.

$\mathcal{E}$ being stationary implies that:

$\delta\mathcal{E} = 0$.

Let's find an expression for $\delta\mathcal{E}$. The "$\delta$" simply acts as any derivative would, and therefore:

$\delta\mathcal{E} = \int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y \, [\mathbf{\nabla}\phi \cdot \mathbf{\nabla}\delta\phi - S\delta\phi]$.

To perform the integral of the first term by parts, we need to derive an appropriate integration by parts identity in two dimensions.

Starting with the *divergence theorem* in two dimensions for a vector field $\mathbf{V}$:

$\int_C \mathbf{V}\cdot\hat{\mathbf{n}}\mathrm{d}\ell = \int_S \mathbf{\nabla}\cdot\mathbf{V}\mathrm{d}A$,

whre the curve $C$ is the boundary of the surface $S$, and $\hat{\mathbf{n}}$ is a unit vector perpendicular to $C$.

Changing $\mathbf{V} \to u\mathbf{V}$, where $u$ is a scalar function:

$\int_C u\mathbf{V}\cdot\hat{\mathbf{n}}\mathrm{d}\ell = \int_S \mathbf{\nabla}\cdot(u\mathbf{V})\mathrm{d}A$.

Consider the RHS and expand:

$\int_S \mathbf{\nabla}\cdot(u\mathbf{V})\mathrm{d}A = \int_S u\mathbf{\nabla}\cdot\mathbf{V}\mathrm{d}A + \int_S \mathbf{\nabla}u\cdot\mathbf{V}\mathrm{d}A$.

And rearranging:

$\int_S \mathbf{\nabla}u\cdot\mathbf{V}\mathrm{d}A = \int_S \mathbf{\nabla}\cdot(u\mathbf{V})\mathrm{d}A - \int_S u\mathbf{\nabla}\cdot\mathbf{V}\mathrm{d}A$.

By the divergence theorem as written above:

$\int_S \mathbf{\nabla}u\cdot\mathbf{V}\mathrm{d}A = \int_C u\mathbf{V}\cdot\hat{\mathbf{n}}\mathrm{d}\ell - \int_S u\mathbf{\nabla}\cdot\mathbf{V}\mathrm{d}A$.

If we now choose $\mathbf{V} = \mathbf{\nabla}\phi$ and $u = \delta\phi$, we have:

$\int_S \mathbf{\nabla}\delta\phi\cdot\mathbf{\nabla}\phi\mathrm{d}A = \int_C \delta\phi\mathbf{\nabla}\phi\cdot\hat{\mathbf{n}}\mathrm{d}\ell - \int_S \delta\phi\mathbf{\nabla}\cdot\mathbf{\nabla}\phi\mathrm{d}A$,

an integration by parts formula in the necessary form.

Substituting the above into:

$\delta\mathcal{E} = \int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y \, [\nabla\phi\cdot\nabla\delta\phi - S\delta\phi]$,

we get:

$\delta\mathcal{E} = \int_C \delta\phi\mathbf{\nabla}\phi\cdot\hat{\mathbf{n}}\mathrm{d}\ell + \int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y\delta\phi[-\nabla^2\phi - S]$,

where the line integral is over the boundary of the region of interest ($C$). Since we consider only variations that respect the boundary conditions, $\delta\phi$ must vanish on $C$, so that the line integral does as well.

Therefore:

$\delta\mathcal{E} = \int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y \delta\phi[-\nabla^2\phi - S]$,

and demanding that $\delta\mathcal{E} = 0$ for any $\delta\phi$ implies that:

$-\nabla^2\phi - S = 0$, i.e. that $\mathcal{E}$ is stationary when $\phi$ is a solution to our differential equation.

Moreover, it can be shown that $\mathcal{E}$ is not only stationary when $\phi$ is a solution to the PDE, but that it is a minimum as well (left as an exercise!).

Let's also derive a discrete approximation to the $\mathcal{E}$ functional. We employ a two-point difference formula to approximate each first derivative in $(\mathbf{\nabla}\phi)^2$, at the points halfway between the lattice points, and use the trapezoid rule for the integrals.

As a reminder, the trapezoid rule for integration in one dimension is given by:

$\int_a^b \mathrm{d}x f(x) = \frac{h}{2}\left[f(a) + f(b) + 2\sum_{i=1}^{N-1} f(x_i)\right]$.

In two dimensions, it can be shown that the trapezoid rule has the form:

\begin{align} \int_a^b \mathrm{d} x \int_c^d \mathrm{d} y f(x,y) &= \frac{h^2}{4} [f(a,c) + f(b,c) + f(a,d) + f(b,d)\ &+ 2 \sum_{i=1}^{N-1} f(x_i, c) + 2 \sum_{i=1}^{N-1} f(x_i, d)\ &+ 2 \sum_{j=1}^{N-1} f(a, y_j) + 2 \sum_{i=1}^{N-1} f(b, y_j)\ &+ 4 \sum_{i=1}^{N-1} \sum_{j=1}^{N-1} f(x_i, y_j)] \end{align}

(see, e.g., https://math.stackexchange.com/questions/2891298/derivation-of-2d-trapezoid-rule for a derivation)

This leads to:

$\mathcal{E} = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}[(\phi_{ij} - \phi_{i-1j})^2 + (\phi_{ij} - \phi_{ij-1})^2] - h^2 \sum_{i=1}^{N-1}\sum_{j=1}^{N-1} S_{ij}\phi_{ij}$.

Note that setting:

$\frac{\partial\mathcal{E}}{\partial\phi_{ij}} = 0 \forall ij$,

leads to the difference equation derived in the previous section.

Let's now discuss where the boundary conditions enter the set of the linear equations:

$-\left[\frac{\phi_{i+1j}+\phi_{i-1j}-2\phi_{ij}}{h^2} + \frac{\phi_{ij+1}+\phi_{ij-1}-2\phi_{ij}}{h^2}\right] = S_{ij}$.

Unless the coordinate system is well adapted to the geometry of the surfaces on which the boundary conditions are imposed, the lattice points will only roughly describe the geometry. One can always improve the accuracy by using a non-uniform lattice spacing, and place more points in the regions near the surfaces, or by transforming to a coordinate system in which the boundary conditions are expressed more naturally.

In any event, the boundary conditions will then provide the values of the $\phi_{ij}$ at some subset of lattice points.

At a point far away from the boundaries, the boundary conditions do not enter directly. However, consider the equation at a point *just next* to the boundary, say $(i, N-1)$. Since $\phi_{iN}$ is specified as a part of the boundary conditions, we can rewrite the equation as:

$4\phi_{iN-1} - \phi_{i+1N-1} - \phi_{i-1N-1} - \phi_{iN-2} = h^2 S_{ij} + \phi_{iN}$.

Therefore, $\phi_{iN}$ enters *not* as an unknown, but rather as an inhomogeneous, *known* term.

These considerations show that the discrete approximation to the PDE is equivalent to a system of linear equations for the unknown values of $\phi$ at the interior points.

In matrix notation:

$\mathbf{M}\boxed{?} = \mathbf{s}$,

where $\mathbf{M}$ is the matrix appearing in the linear system of equations, and the inhomogeneous term $\mathbf{s}$ is proportional to $S$ at the interior points, and linearly related to the boundary conditions on $\phi$.

In any sort of practical situation, there are a very large number of these equations, e.g. $N^2 = 2500$ for, say, $N = 50$, so the solution by direct matrix inversion can be impractical.

Fortunately, sincew the discrete approximation to the Laplacian involves only neighboring points, most of the elements of **M** vanish (since it is sparse), and there are efficient iterative techniques for solving the matrix equation.

We begin their discussion by considering an analogous, but simpler one-dimensional boundary value problem, and then return to the two-dimensional case.

### 10.2.3 An Iterative Method for Boundary Value Problems

The one-dimensional boundary value problem analogous to the two-dimensional problem we have been discussing can be written as:

$-\frac{\mathrm{d}^2\phi}{\mathrm{d}x^2} = S(x)$,

with $\phi(0)$ and $\phi(1)$ specified.

The related variational principle involves the quantity:

$\mathcal{E} = \int_0^1 \mathrm{d}x \left[\frac{1}{2}\left(\frac{\mathrm{d}\phi}{\mathrm{d}x}\right)^2 - S\phi\right]$.

This can be discretized on a uniform lattice with spacing $h = 1/N$ as:

$\mathcal{E} = \frac{1}{2h}\sum_i^{N-1}(\phi_i - \phi_{i-1})^2 - h\sum_{i=1}^{N-1} S_i\phi_i$.

Considering variations with respect to $\phi_i$ yields the difference equation:

$2\phi_i - \phi_{i+1} - \phi_{i-1} = h^2 S_i$,

which is nothing but the naive discretization of the differential equation.

We already discussed methods for solving the boundary value problem in one dimension in the previous chapter (using the forward/backward Numerov method), but we can also consider the equation together with the known values $\phi_0$ and $\phi_N$, as a set of linear equations.

For a modest number of points, say $N \lesssim 100$, the system can be solved by direct matrix methods.

However, to illustrate the iterative methods appropriate for large sparse matrices of elliptic PDEs in two or more dimensions, let's begin by solving for $\phi$:

$\phi_i = \frac{1}{2}[\phi_{i+1} + \phi_{i-1} + h^2 S_i]$.

This equation is not manifestly useful, since we don't know the $\phi$'s on the RHS. However, it can be interpreted as giving as an "improved" value for $\phi_i$, based on the values of $\phi$ at the neighboring points.

The strategy, known as Gauss-Seidel iteration, is then to guess some initial solution, and then to "sweep" systematically through the lattice (e.g. from left to right), successively replacing $\phi$ at each point by an improved value.

Note that the most "current" values of the $\phi_{i\pm1}$ are to be used in the RHS of the equation. By repeating this sweep many times, an initial guess for $\phi$ can be "relaxed" to the correct solution.

To investigate the convergence of this procedure, we generalize the equation, so that at each step of the relaxation, $\phi_i$ is replaced by a linear mixture of its old value and the new "improved" one, given by:

$\phi_i \to \phi_i' = (1-\omega)\phi_i + \frac{\omega}{2}[\phi_{i+1} + \phi_{i-1} + h^2 S_i]$.

Here, $\omega$ is a parameter that can be adjusted to control the rate of relaxation.

To see that the above procedure leads to an "improvement" in the solution, we calculate the change in the energy functional (in one dimension), remembering that all the $\phi$'s except $\phi_i$ are to be held fixed. After some algebra, one finds:

$E' - E = -\frac{\omega(2-\omega)}{2h}\left[\frac{1}{2}(\phi_{i+1} + \phi_{i-1} + h^2 S_i) - \phi_i\right]^2 \leq 0$,

so that, as long as $0 < \omega < 2$, the energy never increases, and should thus converge to the required minimum value as the sweeps proceed.

### 10.2.4 Example 10.1: The relaxation method for partial differential equations in one dimension.

(a) Use the relaxation method in one dimension to solve the boundary value problem defined by the differential equation:

$-\frac{\mathrm{d}^2\phi}{\mathrm{d}x^2} = S(x)$,

with source term $S(x) = 12x^2$ and boundary conditions $\phi(0) = \phi(1) = 0$. Start with $\phi_i = 0$ as your initial guess. Use $N = 40$ lattice steps and perform a few hundred iterations (e.g. 1000) to reach your solution. Perform all calculations for the values of the relaxation parameter $\omega = 0.5, 1.0, 1.5$.

The relaxation method: Discretize the one-dimensional space and use the discretized "solved" iterative form:

$\phi_i \to \phi_i' = (1-\omega)\phi_i + \frac{\omega}{2}[\phi_{i+1} + \phi_{i-1} + h^2 S_i]$,

where $\phi_i'$ is the updated value of $\phi_i$ at lattice site $i$, $h$ is the lattice spacing, and $\omega$ is a "relaxation" parameter.

Compare to the exact solution: $\phi(\ x) = x(1-x^3)$.

(b) Write a function that calculates the energy functional, defined by:

$\mathcal{E} = \int_0^1 \mathrm{d}x \left[\frac{1}{2}\left(\frac{\mathrm{d}\phi}{\mathrm{d}x}\right)^2 - S\phi\right]$.

Use the discretized form:

$\mathcal{E} = \frac{1}{2h}\sum_{i=1}^{N}(\phi_i - \phi_{i-1})^2 - h\sum_{i=1}^{N-1} S_i\phi_i$.

Calculate the value of the energy at the end of each iteration and plot the energy as a function of iteration.

Compare to the exact solution energy given by: $\mathcal{E} = -9/14 \simeq -0.64286$.

Despite the rather poor initial guess for $\phi$ in Example 10.1, the iterations converge and the converged energy is independent of the relaxation parameter, $\omega$. The rate of convergence clearly depends on $\omega$. A general analysis shows that the best choice for the relaxation parameter depends upon the lattice size and on the geometry of the problem, and it is found to be usually $\omega > 1$. The optimal value can be determined empirically by examining the convergence of the solution for only a few iterations, before choosing a value to be used for many iterations.

### 10.2.5 The Relaxation Method in Higher Dimensions

The generalization of the relaxation method to two-dimensional problems is straightforward.

For the differential equation:

$-\left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right]\phi = S(x,y)$,

we get, for the updated value of the field at a point $(x_i, y_j)$:

$\phi_{ij} \to \phi_{ij}' = (1-\omega)\phi_{ij} + \frac{\omega}{4}[\phi_{i+1j} + \phi_{i-1j} + \phi_{ij+1} + \phi_{ij-1} + h^2 S_{ij}]$.

This algorithm can be applied successively to each point on the lattice, say sweeping the rows in order from top to bottom and each row from left to right. One can again show that the energy functional,

$\mathcal{E} = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}[(\phi_{ij} - \phi_{i-1j})^2 + (\phi_{ij} - \phi_{ij-1})^2] - h^2\sum_{i=1}^{N-1}\sum_{j=1}^{N-1} S_{ij}\phi_{ij}$,

always decreases, and there will be convergence to the required solution.

Several considerations can serve to enhance the convergence in practice:

1. Starting from a good guess at the solution, e.g. perhaps one with similar, but simpler, boundary conditions, will reduce the number of iterations required.
2. An optimal value of the relaxation parameter should be used, either estimated analytically or determined empirically (as described above).
3. It may sometimes be more efficient to concentrate the relaxation process, for several iterations, in some sub-area of the lattice, where the trial solution is known to be particularly poor, thus not wasting effort on already-relaxed parts of the solution.
4. One can always do a calculation on a relatively coarse lattice that relaxes with a small amount of numerical work, and then interpolate the solution found onto a finer lattice, to be used as the starting guess for further iterations.

## 10.3 Parabolic Partial Differential Equations

Typical parabolic PDEs one encounters in physical situations are the diffusion equation:

$\frac{\partial \phi}{\partial t} = \boldsymbol{\nabla} \cdot (D \boldsymbol{\nabla} \phi) + S,$

where $D$ is the diffusion constant (possibly space-dependent) and $S$ is a source function.

Another example is the Schrödinger equation:

$i \frac{\partial \phi}{\partial t} = -\frac{\hbar^2}{2m} \nabla^2 \phi + V \phi.$

In contrast to the boundary value problems, these are generally of the initial value type: we are given the field $\phi$ at an initial time, and seek to find it at a later time. The evolution is subject to certain spatial boundary conditions, e.g. the Schrödinger wave function vanishes at very large distances, or the temperature or heat flux is specified on some surfaces.

Methods by which such problems are solved on a computer are straightforward, although a few subtleties are involved.

### 10.3.1 Naive Discretization and Instabilities

We begin by treating diffusion in one dimension, with a uniform diffusion constant $D = 1$.

We take $x \in [0, 1]$ and assume Dirichlet boundary conditions that specify the value of the field at the end points of the interval.

We will focus on the rescaled equation:

$\frac{\partial \phi}{\partial t} = \frac{\partial^2 \phi}{\partial x^2} + S(x, t).$

As before, we will approximate the spatial derivatives by finite differences on a uniform lattice of $N + 1$ points with spacing $h = 1/N$. The time derivative will be approximated by the simplest first-order difference formula, assuming a time step $\Delta t$.

We will use a *superscript* $n$ to label the time step:

$\phi^n \equiv \phi(t_n)$, with $t_n = n \Delta t$.

As our first approximation to the equation, we approximate the second derivative on the RHS as follows:

$\frac{\partial^2 \phi}{\partial x^2} \equiv -\hat{H} \phi \approx \frac{\phi^n_{i+1} + \phi^n_{i-1} - 2\phi^n_i}{h^2},$

where we have also defined an operator which corresponds to minus the second spatial derivative, with the discretized form:

$\hat{H} \phi^n_i = -\frac{\phi^n_{i+1} + \phi^n_{i-1} - 2\phi^n_i}{h^2}.$

An "explicit" discretization, or differencing scheme, of the differential equation is then given by:

$\frac{\phi_i^{n+1}-\phi_i^n}{\Delta t} = \frac{\phi_{i+1}^n+\phi_{i-1}^n-2\phi_i^n}{h^2} + S_i^n$.

If we then solve the differential equation for $\phi$ at the next time step, we obtain:

$\phi^{n+1} = (1 - \hat{H}\Delta t)\phi^n + S^n\Delta t$,

where we have suppressed the index $i$, essentially turning the above equation to a matrix equation.

### 10.3.2 Example 10.2: Parabolic PDEs: a first example

Solve the differential equation (i.e. find $\phi$ for later times):

$\frac{\partial\phi}{\partial t} = \frac{\partial^2\phi}{\partial x^2} + S(x,t)$,

for $S = 0$ and $\phi(0,t) = \phi(1,t) = 0$, satisfied by the initial condition of a Gaussian centered about $x = 1/2$:

$\phi(x, t = 0) = e^{-20(x-1/2)^2} - e^{-20(x-3/2)^2} - e^{-20(x+1/2)^2}$,

where the latter two "image" Gaussians approximately ensure the boundary conditions at $x = 1$ and $x = 0$, respectively.

Use the naive discretization formula:

$\phi^{n+1} = (1 - \hat{H}\Delta t)\phi^n + S^n\Delta t$,

where $(H\phi)_i \equiv -\frac{1}{h^2}(\phi_i + \phi_{i-1} - 2\phi_i)$.

Integrate up to $t_{\text{max}} = 0.045$ with a time steps $\Delta t = 0.00075$ and $\Delta t = 0.0080$, over a lattice with $N = 25$.

Compare to the analytical solution:

$\phi(x,t) = \tau^{-1/2}\left[e^{-20(x-1/2)^2/\tau} - e^{-20(x-3/2)^2/\tau} - e^{-20(x+1/2)^2/\tau}\right]$, where $\tau = 1 + 80t$.

Both time steps chosen are quite small compared to the natural time scale of the solution, $t \approx 0.01$. Furthermore, as we increase the time step, we see that things go very wrong: an unphysical instability develops in the numerical solution, which quickly acquires violent oscillations from one lattice point to another.

Let's try to understand what is happening here. Consider the differential equation with $S = 0$, as we are trying to solve in the above example:

$\frac{\partial\phi}{\partial t} = \frac{\partial^2\phi}{\partial x^2}$.

The RHS can be replaced by the operator $-\hat{H}$, to obtain:

$\frac{\partial\phi(x,t)}{\partial t} = -\hat{H}\phi(x,t)$.

This can be formally solved to give:

$\phi(x,t) = e^{-t\hat{H}}\phi(x,0)$.

If we now discretize time: $t_n = n\Delta t$, $\phi(t_n, x) = \phi^n$, as before, we get:

$\phi^n = e^{-n\Delta t\hat{H}}\phi^0$.

Now let the set of states $\psi_\lambda$ be the *eigenfunctions* of the operator $\hat{H}$ with eigenvalues $\epsilon_\lambda$.

Since $\hat{H}$ is Hermitian, the eigenvalues $\epsilon_\lambda$ are real and the eigenvectors can be chosen to be orthonormal.

What this implies is that we can expand the solution at any time in terms of this basis:

$\phi^n = \sum_\lambda \phi_\lambda^n\psi_\lambda$,

where $\phi_\lambda^n$ are effectively the coefficients of the expansion.

We also expand the initial condition in terms of the eigenvectors:

$\phi^0 = \sum_\lambda \phi_\lambda^0\psi_\lambda$,

then, substituting into the formal discretized solution, $\phi^n = e^{-n\Delta t\hat{H}}\phi^0$, we obtain:

$\sum_\lambda \phi^n_\lambda \psi_\lambda = e^{-n\hat{H}\Delta t}\sum_\lambda \phi^0_\lambda \psi_\lambda$,

giving us the evolution of each component of the solution:

$\phi^n_\lambda = e^{-n\epsilon_\lambda \Delta t}\phi^0_\lambda$.

This corresponds to the correct behavior of the diffusion equation: short-wavelenght components (i.e. large eigenvalues $\epsilon_\lambda$), disappear more rapidly as the solution "smooths out".

However, the naive discretization (i.e. the "explicit" scheme):

$\phi^{n+1} = (1 - \hat{H}\Delta t)\phi^n$,

would result in:

$\phi^n = (1 - \hat{H}\Delta t)^n \phi_0$,

or:

$\phi^n_\lambda = (1 - \epsilon_\lambda \Delta t)^n \phi^0$.

Recall one of the definitions of the exponential function:

$e^{\alpha t} = \lim_{n\to\infty}(1 + \alpha\Delta t)^n$, with $\Delta t = t/n$.

So the naive discretization in the limit $\Delta t \to 0$ would yield the correct evolution:

$\phi^n_\lambda = e^{-n\epsilon_\lambda \Delta t}\phi^0_\lambda$.

Therefore, as long as $\Delta t$ is chosen to be small, $(1 - \epsilon_\lambda \Delta t)^n$ approximates the exponential, and the short-wavelength (large $\epsilon_\lambda$) components dampen with time.

However, if $\Delta t$ is too large, one or more of the quantities $1 - \epsilon_\lambda \Delta t$ has an absolute value greater than unity. The corresponding components, even if present only due to very small numerical round-off errors, are then amplified with each time step, and soon grow to dominate.

To quantify the limit on $\Delta t$, we have some guidance in that the eigenvalues of $\hat{H}$ are known analytically in this simple model problem.

We have: $\hat{H} = -\frac{\partial^2}{\partial x^2}$. The eigenvalue equation is simply then:

$\hat{H}\psi_\lambda = \epsilon_\lambda \psi_\lambda$

$\Rightarrow \frac{\partial^2 \psi_\lambda}{\partial x^2} = -\epsilon_\lambda \psi_\lambda$,

with $\psi_\lambda(0) = \psi_\lambda(1) = 0$.

The eigenfunctions are then:

$\psi_\lambda = A \sin\sqrt{\epsilon_\lambda}x$, with eigenvalues $\sqrt{\epsilon_\lambda} = \lambda\pi$, with $\lambda = 1, 2, ....$

On a lattice with $x_i = ih$, the discretized solutions are then:

$(\psi_\lambda)_i = A \sin\lambda\pi ih$,

and since $h = 1/N$:

$(\psi_\lambda)_i = A \sin\frac{\lambda\pi i}{N}$.

The eigenvalues have to change when we move to the lattice. To find them, consider the action of the discretized form of $\hat{H}$ on the discretized eigenfunctions:

$(\hat{H}\psi_\lambda)_i = -\frac{1}{h^2}\left[(\psi_\lambda)_{i+1} + (\psi_\lambda)_{i-1} - 2(\psi_\lambda)_i\right]$.

or:

$(\hat{H}\psi_\lambda)_i = -\frac{1}{h^2}\left[\sin\frac{\lambda\pi(i+1)}{N} + \sin\frac{\lambda\pi(i-1)}{N} - 2\sin\frac{\lambda\pi i}{N}\right]$.

Using the trigonometric identities $\sin(A+B) = \sin A \cos B + \sin B \cos A$ and $1 - \cos\frac{\lambda\pi}{N} = 2\sin^2\frac{\lambda\pi}{2N}$, it easy to verify that the eigenvalues after discretization are:

$\epsilon_\lambda = \frac{4}{h^2}\sin^2\frac{\lambda\pi}{2N}$.

The largest eigenvalue corresponds to: $\lambda = N-1$, i.e.:

$\epsilon_\lambda = 4/h^2\left[\sin\left(\frac{N\pi}{2N}\right)\cos\left(\frac{\pi}{2N}\right) - \cos\left(\frac{N\pi}{2N}\right)\sin\left(\frac{\pi}{2N}\right)\right] = 4/h^2\cos(\pi/2N)$,

and to an eigenvector that changes sign from one lattice point to the next:

$(\psi_{N-1})_i = A\sin\left(\frac{(N-1)\pi i}{N}\right) = A\left[\sin(\pi i)\cos\left(\frac{\pi i}{N}\right) - \cos(\pi i)\sin\left(\frac{\pi i}{N}\right)\right] \propto \cos\pi i$.

Requiring that $|1-\epsilon_\lambda\Delta t| < 1$, then yields: $\Delta t \lesssim h^2/2$, which in the case of our example corresponds to $\Delta t \lesssim 0.0008$.

The question of stability is quite distinct from that of accuracy, as the limit imposed on the time step is set up by the spatial step used and not by the characteristic time scale of the solution, which is much larger.

The explicit scheme which we have discussed is unsatisfactory, as the instability forces us to use a much smaller time step than is required to describe the evolution adequately. Indeed, the situation gets even worse if we try to use a finer spatial lattice to obtain a more accurate solution.

Although the restriction on $\Delta t$ that we have derived is rigorous only for the simple case we have considered, it does provide a useful guide for more complicated situations, as the eigenvector of $\hat{H}$ with the largest eigenvalue will always oscillate from one lattice point to the next. Its eigenvalue is therefore quite insensitive to the global features of the problem.

### 10.3.3 Implicit Schemes

One way around the instability of the explicit algorithm discussed above, is to retain the general form:

$\frac{\phi_i^{n+1}-\phi_i^n}{\Delta t} = \frac{\phi_{i+1}^n+\phi_{i-1}^n-2\phi_i^n}{h^2} + S_i^n$,

but to replace the second space derivative by that of the solution, *at the new time*, i.e.:

$\frac{\phi_i^{n+1}-\phi_i^n}{\Delta t} = \frac{\phi_{i+1}^{n+1}+\phi_{i-1}^{n+1}-2\phi_i^{n+1}}{h^2} + S_i^n$.

This defines an *implicit* scheme, since the unknown, $\phi^{n+1}$ appears on both sides of the equation.

In terms of the operator $(\hat{H}\phi^{n+1})_i = -\frac{\phi_{i+1}^{n+1}+\phi_{i-1}^{n+1}-2\phi_i^{n+1}}{h^2}$, the above implicit equation can be written as:

$\frac{\phi_i^{n+1}-\phi_i^n}{\Delta t} = -\hat{H}\phi_i^{n+1} + S_i^n$.

Solving for $\phi^{n+1}$, we get:

$\phi^{n+1} = \frac{1}{1+\hat{H}\Delta t}\left[\phi^n + S^n\Delta t\right]$.

This scheme is equivalent to the explicit scheme to lowest order in $\Delta t$ (as seen by expanding the denominator). However, it is much better in that larger time steps can be used, as the operator $(1+\hat{H}\Delta t)^{-1}$ has eigenvalues $(1+\epsilon_\lambda\Delta t)^{-1}$, all of whose moduli are less than 1 for any value of $\Delta t$. Although this decrease is inaccurate (i.e. not exponential as expected by the "formal" solution of the diffusion equation), for the most rapidly oscillating components, such components should not be large in the initial conditions, if the spatial discretization is accurate.

In any event, there's no amplification of the large-eigenvalue components, which caused the instability found in the explicit scheme. For the slowly-varying components of the solution, corresponding to small eigenvalues, the evolution closely approximates the exponential behavior at each step.

A potential drawback is that it requires the solution of a set of linear equations (albeit tri-diagonal) at each time step to find $\phi^{n+1}$. This is equivalent to the application of the inverse of the matrix $(1+\hat{H}\Delta t)^{-1}$ to the vector appearing in the brackets. Since the inverse itself is time-independent, it might be found only once at the beginning of the calculation and then used for all times, but application still requires $N^2$ operations if done directly.

Fortunately, there exists an algorithm that provides a very efficient solution ($\mathcal{O}(N)$ operations) of a tri-diagonal system of equations, known as "Gaussian elimination and back-substitution".

The algorithm proceeds as follows: let's consider trying to solve the tri-diagonal linear system of equations:

$\mathbf{A}\vec{?} = \mathbf{b}$ for the unknowns $\phi_i$:

$A_i^- \phi_{i-1}^{n+1} + A_i^0 \phi_i^{n+1} + A_i^+ \phi_{i+1}^{n+1} = b_i.$

Here, the $A_i^{\pm,0}$ are the only non-vanishing elements of $\mathbf{A}$ and the $b_i$ are the known quantities. This is exactly the form of the problem posed by the evaluation of our problem:

$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} = \frac{\phi_{i+1}^{n+1} + \phi_{i-1}^{n+1} - 2\phi_i^{n+1}}{h^2} + S_i^n.$

rearranging:

$\phi_i^{n+1} - \frac{\Delta t}{h^2}\left[\phi_{i+1}^{n+1} + \phi_{i-1}^{n+1} - 2\phi_i^{n+1}\right] = \phi_i^n + S_i^n,$

to get:

$-\frac{\Delta t}{h^2}\phi_{i-1}^{n+1} + \left(1 - 2\frac{\Delta t}{h^2}\right)\phi_i^{n+1} - \frac{\Delta t}{h^2}\phi_{i+1}^{n+1} = \phi_i^n + S_i^n \Delta t,$

which corresponds to:

$b_i = \phi_i^n + S_i^n \Delta t,$

$A_i^0 = 1 + 2\frac{\Delta t}{h^2},$

$A_i^{\pm} = -\frac{\Delta t}{h^2}.$

To solve this system of equations, we assume that the solution satisfies a one-term *forward* recursion relation of the form:

$\phi_{i+1}^{n+1} = \alpha_i \phi_i^{n+1} + \beta_i,$

where the coefficients $\alpha_i$ and $\beta_i$ are to be determined.

Substituting the above relation into the linear system of equations:

$A_i^- \phi_{i-1}^{n+1} + A_i^0 \phi_i^{n+1} + A_i^+(\alpha_i \phi_i^{n+1} + \beta_i) = b_i,$

and solving for $\phi_i^{n+1}$, we get:

$\phi_i^{n+1} = \gamma_i A_i^- \phi_{i-1}^{n+1} + \gamma_i(A_i^+ \beta_i - b_i),$

with $\gamma_i = -\frac{1}{A_i^0 + A_i^+ \alpha_i}$.

comparing the above with: $\phi_i^{n+1} = \alpha_{i-1}\phi_{i-1}^{n+1} + \beta_{i-1}$, i.e. the hypothesis with $i \to i-1$, we can identify the following recursion relations for the $\alpha$'s and $\beta$'s:

$\alpha_{i-1} = \gamma_i A_i^-,$

$\beta_{i-1} = \gamma_i(A_i^+ \beta_i - b_i).$

Note that these are *backward* recursion relations.

The strategy to solve the problem then proceeds as follows:

1. We use the $\alpha_{i-1}$, $\beta_{i-1}$ and $\gamma_i$ relations to sweep the lattice *backwards* to determine the $\alpha_i$ and $\beta_i$, running from $N-2$ down to 0. The starting values to be used are: $\alpha_{N-1} = 0$ and $\beta_{N-1} = \phi_N^n$, which will guarantee the correct value of $\phi$ at the last lattice point.
2. Having determined these coefficients, we can then use the recursion relation: $\phi_{i+1}^{n+1} = \alpha_i \phi_i^{n+1} + \beta_i$ in a *forward* sweep from $i = 0$ to $i = N-1$ to determine the solution, with the starting value of $\phi_0^{n+1}$ known from the boundary conditions.

We have then determined the solution in only two sweeps of he lattice, involving $\mathcal{O}(N)$ arithmetic operations. The increase in numerical effort per time step is about a factor of two.

*Note that*: the $\alpha_i$ and $\gamma_i$ are independent of **b**, so that it is more efficient to compute these coefficients only once and store them at the beginning of the calculation. Only the $\beta_i$ are then needed to be computed for each time step.

### 10.3.4 Example 10.3: Application of the Implicit Scheme for Parabolic PDEs

Use the implicit scheme to solve the problem of Example 10.2. Use a lattice with $N = 25$ intervals. Try time steps of $\Delta t = 0.00075$ and $\Delta t = 0.005$.

Compare to the exact solution by graphing the results at $t = 0.045$.

### 10.3.5 Solution by Direct Matrix Multiplication

Modern computers are efficient at matrix manipulations. The number of operations for the solution of parabolic PDEs will be higher than what has been discussed in the previous sections, but nevertheless, for small problems, it should be efficient enough.

Let's write down the form of the $\hat{H}$ operator in the matrix representation. This is easy, since we know its action (up to a factor of $-1/h^2$):

$(\hat{H}\phi)_i \propto \phi_{i+1} + \phi_{i-1} - 2\phi_i.$

E.g. for $i = 1$:

$(\hat{H}\phi)_1 \propto \phi_2 + \phi_0 - 2\phi_1,$

for $i = 2$:

$(\hat{H}\phi)_2 \propto \phi_3 + \phi_1 - 2\phi_2,$

for $i = N - 1$:

$(\hat{H}\phi)_{N-1} \propto \phi_N + \phi_{N-2} - 2\phi_{N-1},$

or in matrix form:

$$\hat{H} \propto \begin{pmatrix} 0 & 0 & ... & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & ... \\ 0 & 1 & -2 & 1 & 0 & ... \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & ... & 1 & -2 & 1 \\ 0 & 0 & ... & 0 & 0 & 0 \end{pmatrix}.$$

Therefore, to obtain a solution in the implicit scheme:

$\phi^{n+1} = [1 + \hat{H}\Delta t]^{-1} [\phi^n + S^n \Delta t],$

with:

$(\hat{H}\phi)_i = -\frac{\phi_{i+1} + \phi_{i-1} - 2\phi_i}{h^2},$

all we need to do is find the inverse of the matrix operator:

$[1 + \hat{H}\Delta t].$

Note that the inverse is calculated in NumPy via:

```
np.linalg.inv()
```

On the other hand, we can also use the *full* time evolution formula derived from the formal solution, via the matrix exponential.

Starting from the PDE:

$\frac{\partial\phi(x,t)}{\partial t} = -\hat{H}\phi(x,t)$.

This can be formally solved to give:

$\phi(x,t) = e^{-t\hat{H}}\phi(x,0)$,

which implies, in the discretized form, that:

$\phi^n = e^{-n\Delta t\hat{H}}\phi^0$.

A single time step therefore corresponds to:

$\phi^{n+1} = e^{-\Delta t\hat{H}}\phi^n$.

The matrix exponential can be calculated in SciPy (via a Padé approximant, see https://epubs.siam.org/doi/10.1137/09074721X):

`scipy.linalg.expm()`,

allowing us to perform the direct time step evolution in the simple case we are considering.

### 10.3.6 Example 10.4: Direct inversion of the matrix and the full evolution

Use direct inversion of the tri-diagonal matrix to calculate the evolution in the problem of Examples 10.3 and 10.4.

(a) Use the implicit formula as the starting point:

$\phi^{n+1} = \frac{1}{1+\hat{H}\Delta t}\left[\phi^n + S^n\Delta t\right]$,

with:

$(\hat{H}\phi)_i = -\frac{\phi_{i+1}+\phi_{i-1}-2\phi_i}{h^2}$.

(b) Use the *full* evolution formula:

$\phi^n = e^{-n\Delta t\hat{H}}\phi^0$,

in the form:

$\phi^{n+1} = e^{-\Delta t\hat{H}}\phi^n$.

You may use the matrix SciPy exponential:

`scipy.linalg.expm`

### 10.3.7 Diffusion and Boundary Value Problems in Higher Dimensions

We saw that diffusion in one dimension is best handled either by an implicit method or by direct inversion of a tri-diagonal matrix.

It is therefore natural to extend this approach to two or more spatial dimensions.

For the two-dimensional diffusion equation:

$\frac{\partial\phi}{\partial t} = \nabla^2\phi$,

the discretization is straightforward, and following our discussion of the one-dimensional problem, the time evolution should be effected by:

$\phi^{n+1} = \frac{1}{1+\hat{H}\Delta t}\phi^n$,

where now:

$(\hat{H}\phi)_{ij} = -\frac{1}{h^2}\left[(\phi_{i+1j} + \phi_{i-1j} - 2\phi_{ij}) + (\phi_{ij+1} + \phi_{ij-1} - 2\phi_{ij})\right]$.

Despite the fact that $\hat{H}$ is very sparse, it is not tri-diagonal, and therefore one-dimensional algorithm does not apply.

However, $\hat{H}$ can be written as a sum of operators that separately involve differences only in the $i$ or $j$ indices:

$\hat{H} = \hat{H}_i + \hat{H}_j$,

where $\hat{H}_i = -\frac{1}{h^2}(\phi_{i+1j} + \phi_{i-1j} - 2\phi_{ij})$ and $\hat{H}_j = -\frac{1}{h^2}(\phi_{ij+1} + \phi_{ij-1} - 2\phi_{ij})$.

This means that we can write down an equivalent expression up to $\mathcal{O}(\Delta t)$ to the original one:

$\phi^{n+1} = \frac{1}{1+\hat{H}_i\Delta t}\frac{1}{1+\hat{H}_j\Delta t}\phi^n$.

This now can be evaluated as before, as each of the required inversions involves a tri-diagonal matrix.

If we define the auxiliary function $\phi^{n+1/2}$, then we can write:

$\phi^{n+1/2} = \frac{1}{1+\hat{H}_j\Delta t}\phi^n$,

and:

$\phi^{n+1} = \frac{1}{1+\hat{H}_i\Delta t}\phi^{n+1/2}$.

Thus, $(1 + \hat{H}_j\Delta t)^{-1}$ is applied by forward and backward sweeps of the lattice in the $j$ direction, independently for each value of $i$. The application of $(1 + \hat{H}_i\Delta t)^{-1}$ is then carried out by forward and backward sweeps in the $i$ direction, independently for each value of $j$. This "alternating-direction" scheme is stable for all values of the time step and is generalized straightforwardly to three dimensions.

## 10.4 Iterative Methods for Eigenvalue Problems

Our analysis of the diffusion equation shows that the net result of time evolution is to enhance those components of the solution with smaller eigenvalues of $\hat{H}$ relative to those with larger eigenvalues.

Indeed, for very long times, it is only that component with the lowest eigenvalue that is significant, although it has a very small amplitude.

This suggests a scheme for finding the lowest eigenvalue of an elliptic operator, solving the problem:

$-\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\phi + V(x,y)\phi = \epsilon\phi$,

by guessing a trial eigenvector and subjecting it to a *fictitious* time evolution that will "filter" it to the eigenvector having lowest eigenvalue.

Since we are dealing with a linear problem, the relentlessly decreasing or increasing magnitude of the solution can be avoided by renormalizing continuously as time proceeds.

Concretely, consider the time-independent Schrödinger equation in one dimension with $\hbar = 2m = 1$. The eigenvalue problem is then:

$\left[-\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x)\right]\phi = \epsilon\phi$,

with the normalization condition:

$\int \mathrm{d}x\phi^2 = 1$.

Note that $\phi$ can always chosen to be real if $V$ is.

The corresponding energy functional is:

$\mathcal{E} = \int \mathrm{d}x\left[\left(\frac{\mathrm{d}\phi}{\mathrm{d}x}\right)^2 + V(x)\phi^2(x)\right]$.

We know that $\mathcal{E}$ is stationary at an eigenfunction with respect to variations of $\phi$ that respect the normalization condition, and that the value of $\mathcal{E}$ at this eigenfunction is the associated eigenvalue.

To derive a discrete approximation to this problem, we first discretize $\mathcal{E}$ as:

$\mathcal{E} = \sum_i h \left[ \frac{(\phi_i - \phi_{i-1})^2}{h^2} + V_i \phi_i^2 \right]$,

and the normalization constraint takes the form:

$\sum_i h \phi_i^2 = 1$.

Variation with respect to $\phi$ gives the eigenvalue problem:

$(\hat{H}\phi)_i \equiv -\frac{1}{h^2}(\phi_{i+1} + \phi_{i-1} - 2\phi_i) + V_i \phi_i = \epsilon \phi_i$,

(Note that $\epsilon$ enters as a Lagrange multiplier that ensures the normalization).

The above defines the problem of finding the real eigenvalues and eigenvectors of a (large) symmetric tri-diagonal matrix representing the operator $\hat{H}$.

Although we have already discussed direct methods for solving this problem (e.g. the Numerov algorithm), they cannot be applied to the very large matrices that arise in two- and three-dimensional problems.

In such cases, one is usually interested in the few highest or lowest eigenvalues of the problem, and for these the diffusion picture is appropriate.

We thus consider the problem:

$\frac{\partial \phi}{\partial \tau} = -\hat{H}\phi$,

where $\tau$ is a "fake" time.

For convenience, we can arrange the problem such that the lowest eigenvalue of $\hat{H}$ is positive definite. This can be guaranteed by shifting $\hat{H}$ by a spatially-independent constant, chosen so that the resultant $V_i$ is positive $\forall i$.

To solve this "fake" diffusion problem, any of the algorithms discussed previously in this chapter can be employed.

The simplest is the "explicit" scheme:

$\phi^{n+1} \sim (1 - \hat{H}\Delta\tau)\phi^n$,

where $\Delta\tau$ is a small, positive parameter. The symbol $\sim$ is used to indicate that $\phi^{n+1}$ is to be normalized to unity according to the discrete normalization condition stated above. For an initial guess, we can choose $\phi^0$ to be anything not orthogonal to the exact eigenfunction, although guessing something close to the solution will speed the convergence.

At each step in this refining process, computation of the energy furnishes an estimate of the true eigenvalue.

As a first example, let's consider the infinite square well.

### 10.4.1 Example 10.5: The Infinite Square Well Using the Fake Diffusion Method

(a) Use the fake time diffusion method on a lattice with 21 points to find the lowest eigenvalue and the corresponding eigenstate, of the infinite square well.

Start with a trial eigenfunction $\phi(x) = x(1-x)$ and perform *100* time evolutions with a time step $\Delta\tau = 0.00075$.

$V(x) = 0$ for $0 \leq x \leq 1$, $V(x) = \infty$ otherwise, with $\phi(0) = \phi(1) = 0$.

You may set $\hbar = 2m = 1$ and solve the problem:

$\left[ -\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x) \right] \phi = \epsilon \phi$.

(b) Plot the lowest eigenfunction and compare to the exact one.

(c) Plot the evolution of the eigenvalue for each time step and show that it converges to the correct lowest one.

(d) Use the explicit method to find the *largest eigenvalue*. Use a time step of $\Delta\tau = 0.0015$. Compare to your expectation from the largest eigenvalue for the discrete case with $N = 21$.

Recall that the exact eigenfunctions and eigenvalues are given by:

$\psi_\lambda = \sqrt{2}^{1/2} \sin \lambda\pi x$ with $\epsilon_\lambda = \lambda^2\pi^2$, where $\lambda$ is a non-zero integer.

Also recall that in the discretized form, $\epsilon_\lambda = \frac{4}{h^2} \sin^2 \frac{\lambda\pi}{2N}$.

Although the procedure works, it is unsatisfactory in that the limitation on the size of $\Delta\tau$ caused by the lattice spacing often results in having to iterate many times to refine a poor trial function. This can be alleviated to some extent by choosing a good trial function. Even better is to use an implicit scheme that does not amplify the large-eigenvalue components for any value of $\Delta\tau$.

Another possibility is to use $\exp(-\hat{H}\Delta\tau)$ to refine the trial function, as we did before.

The method can be used to find other eigenvalues, e.g. the second-lowest eigenvalue:

- First find the lowest eigenvalue and eigenfunction by the method described above.
- A trial function for the second eigenfunction is then guessed and refined in the same way, taking care that at each stage of the refinement, the solution remains orthogonal to the lowest eigenfunction already found. This can be done by projecting out the component of the solution not orthogonal to the lowest eigenfunction.

Having found the 2nd-lowest eigenvalue, the third-lowest can be found similarly, taking care that during its refinement, it remains orthogonal to both of the eigenfunctions with lower eigenvalues.

This process cannot be applied to find more than the few lowest (or highest) eigenvalues, as numerical round-off errors in the orthogonalizations to the many previously-found eigenvectors soon grow to dominate.

Although the methods described have been illustrated by a one-dimensional example, it should be clear that the can be applied directly to find the eigenvalues and eigenfunctions of elliptic operators in two or more dimensions.

CHAPTER

ELEVEN

# MORE MONTE CARLO: THE METROPOLIS ALGORITHM

## 11.1 The Algorithm of Metropolis et al.

We have already discussed algorithms for generating random numbers according to a specified distribution (e.g. via von Neumann rejection).

However, it is difficult, or impossible, to generalize them to sample a complicated weight function in many dimensions, and so an alternate approach is required.

A very general way to produce random variables with a given probability distribution of arbitrary form is the algorithm of Metropolis, Rosenbluth, Rosenbluth, Teller and Teller (https://doi.org/10.1063/1.1699114).

This algorithm requires only the ability to calculate the weight function for a given value of the integration variable.

It proceeds as follows: Suppose we want to generate a set of points in a possibly multi-dimensional space of variables, $\mathbf{X}$, distributed with probability density $w(\mathbf{X})$.

The Metropolis et al. algorithm generates a sequence of points:

$\mathbf{X}_0, \mathbf{X}_1, ...,$

as those visited successively by a random walker moving through $\mathbf{X}$ space.

As the walk gets longer and longer, the points it connects approximate more closely the desired distribution.

The rules of the random walk through configuration space are as follows:

- Suppose a random walker is at point $\mathbf{X}_n$ in the sequence.
- To generate $\mathbf{X}_{n+1}$, it makes a *trial step* to a new point, $\mathbf{X}_t$.
- This new point can be chosen in any convenient manner, e.g. uniformly at random within a multi-dimensional cube of small size $\delta$ about $\mathbf{X}_n$.
- The trial step is then "accepted" or "rejected" according to the ratio:

$r = \frac{w(\mathbf{X}_t)}{w(\mathbf{X}_n)}$.

If $r > 1$ then the step is accepted and we set $\mathbf{X}_{n+1} = \mathbf{X}_t$, while if $r < 1$, the step is accepted with probability $r$.

This is accomplished by comparing $r$ with a random number $\eta$, uniformly distributed in $[0, 1]$, and accepting the step if $\eta < r$.

If the trial step is not accepted, then it is rejected and we put $\mathbf{X}_{n+1} = \mathbf{X}_n$.

- This generates $\mathbf{X}_{n+1}$, and we may then proceed to generate $\mathbf{X}_{n+2}$ by the same process, making a trial step from $\mathbf{X}_{n+1}$.
- Any arbitrary point $\mathbf{X}_0$ can be chosen as the starting point for the random walk.

- During this process, the code could be made more efficient by saving the weight function at the current point of the random walk, so that it need not be computed again when deciding whether or not to accept the trial step. The evaluation of $w$ is often the most time-consuming part of a Monte Carlo calculation using the Metropolis et al. algorithm.

## 11.2 Proof of the Metropolis et al. Algorithm

To prove that the algorithm does indeed generate a sequence of points distributed according to $w$, let us consider a large number of walkers, starting from different initial points and moving independently through $\mathbf{X}$ space.

If $N_n(\mathbf{X})$ is the density of walkers at $\mathbf{X}$ after $n$ steps, then the net number of walkers moving from point $\mathbf{X}$ to a point $\mathbf{Y}$ in the next step is:

$\Delta N(\mathbf{X}) = N_n(\mathbf{X})P(\mathbf{X} \to \mathbf{Y}) - N_n(\mathbf{Y})P(\mathbf{Y} \to \mathbf{X}),$

where $P(\mathbf{X} \to \mathbf{Y})$ is the probability that a walker will make a transition to $\mathbf{Y}$ if it is at $\mathbf{X}$.

Taking out a common factor $N_n(\mathbf{Y})P(\mathbf{X} \to \mathbf{Y})$:

$\Delta N(\mathbf{X}) = N_n(\mathbf{Y})P(\mathbf{X} \to \mathbf{Y}) \left[\frac{N_n(\mathbf{X})}{N_n(\mathbf{Y})} - \frac{P(\mathbf{Y}\to\mathbf{X})}{P(\mathbf{X}\to\mathbf{Y})}\right].$

The above shows that there is equilibrium, corresponding to no net population change, when:

$\frac{N_n(\mathbf{X})}{N_n(\mathbf{Y})} = \frac{N_e(\mathbf{X})}{N_e(\mathbf{Y})} \equiv \frac{P(\mathbf{Y}\to\mathbf{X})}{P(\mathbf{X}\to\mathbf{Y})},$

and that changes in $N(\mathbf{X})$ when the system is not in equilibrium tend to drive it toward equilibrium, i.e. $\Delta N(\mathbf{X}) > 0$ if there are "too many walkers" at $\mathbf{X}$, or if $N_n(\mathbf{X})/N_n(\mathbf{Y})$ is greater than its equilibrium value.

Hence it is possible, and it can be shown, that after a large number of steps, the population of the walkers will settle down to its equilibrium distribution $N_e$.

What remains to be shown is that the transition probabilities of the algorithm lead to an equilibrium distribution of walkers $N_e(\mathbf{X}) \sim w(\mathbf{X})$.

The probability of making a step from $\mathbf{X}$ to $\mathbf{Y}$ is:

$P(\mathbf{X} \to \mathbf{Y}) = T(\mathbf{X} \to \mathbf{Y})A(\mathbf{X} \to \mathbf{Y}),$

where $T$ is the probability of making a trial step from $\mathbf{X}$ to $\mathbf{Y}$, and $A$ is the probability of accepting that step.

If $\mathbf{Y}$ can be reached from $\mathbf{X}$ in a single step (i.e. if it is within a cube of side $\delta$, centered about $\mathbf{X}$), then:

$T(\mathbf{X} \to \mathbf{Y}) = T(\mathbf{Y} \to \mathbf{X}).$

so that the equilibrium distribution of the random walkers satisfies:

$\frac{N_e(\mathbf{X})}{N_e(\mathbf{Y})} = \frac{A(\mathbf{Y}\to\mathbf{X})}{A(\mathbf{X}\to\mathbf{Y})}.$

Now, there are two scenarios for the ratio between $w$'s at $\mathbf{X}$ and $\mathbf{Y}$:

if $w(\mathbf{X}) > w(\mathbf{Y})$, then $A(\mathbf{Y} \to \mathbf{X}) = 1$, and:

$A(\mathbf{X} \to \mathbf{Y}) = \frac{w(\mathbf{Y})}{w(\mathbf{X})},$

while if $w(\mathbf{X}) < w(\mathbf{Y})$, then $A(\mathbf{X} \to \mathbf{Y}) = 1$

$A(\mathbf{Y} \to \mathbf{X}) = \frac{w(\mathbf{X})}{w(\mathbf{Y})}.$

Hence, in either case, the equilibrium population of the walkers satisfies:

$\frac{N_e(\mathbf{X})}{N_e(\mathbf{Y})} = \frac{w(\mathbf{X})}{w(\mathbf{Y})},$

so that the walkers are indeed distributed with the correct distribution.

## 11.3 Applying the Metropolis et al. Algorithm

An obvious question is, how do we choose the step size, $\delta$?

To answer this, suppose that $\mathbf{X}_n$ is at a maximum of $w$, the most likely place for it to be. If $\delta$ is large, then $w(\mathbf{X}_t)$ will likely be very much smaller than $w(\mathbf{X}_n)$, and most trial steps will be rejected, leading to an inefficient sampling of $w$. If $\delta$ is very small, most trial steps will be accepted, but the random walker will never move very far, and so also lead to a poor sampling of the distribution.

A good rule of thumb is that the size of the trial step should be chosen so that about half of the trial steps are accepted.

One bane of applying the algorithm to sample a distribution is that the points that make up the random walk $\mathbf{X}_0, \mathbf{X}_1, ...$ are *not independent* of one another, simply from the way in which they are generated. That is, $\mathbf{X}_{n+1}$ is likely to be in the neighborhood of $\mathbf{X}_n$. Thus, while the points might be distributed properly as the walk becomes very long, they are not statistically independent of one another.

This dictates that some care should be taken in using them to, e.g., calculate integrals.

For example, suppose we wish to calculate:

$I = \int \mathrm{d}\mathbf{X} w(\mathbf{X}) f(\mathbf{X})$,

where $w(\mathbf{X})$ is the normalized weight function.

e.g. in one dimension:

$I = \int \mathrm{d}x w(x) f(x)$, and change variables to $y$, with $\mathrm{d}y/\mathrm{d}x = w(x)$ to get $I = \int \mathrm{d}y f(x(y))$.

We would do this by averaging the values of $f$ over the points of the random walk. However, the usual estimate of the variance is invalid, because the $f(\mathbf{X}_i)$ are *not* statistically independent.

The level of statistical dependence can be quantified by calculating the auto-correlation function:

$C(k) = \frac{\langle f_i f_{i+k} \rangle - \langle f_i \rangle^2}{\langle f_i^2 \rangle - \langle f_i \rangle^2}$,

where $\langle ... \rangle$ indicates average over the random walk, e.g.:

$\langle f_i f_{i+k} \rangle = \frac{1}{N-k} \sum_{i=1}^{N-k} f(\mathbf{X}_i) f(\mathbf{X}_{i+k})$.

The non-vanishing of $C$ for $k \neq 0$ means that the $f$'s are *not* independent.

What can be done in practice is to compute the integral and its variance using points along the random walk separated by a fixed interval, the interval being chosen so that there is effectively no correlation between the points used. An appropriate sampling interval can be estimated from the value of $k$ for which becomes small, e.g., say $C \lesssim 0.1$.

Another issue in applying the algorithm is where to start the random walk, i.e. what to take as $\mathbf{X}_0$. In principle any location is suitable and the results will be independent of this choice, as the walker will "thermalize" after some number of steps. In practice, an appropriate starting point is a probable one, where $w$ is large. Some number of thermalization steps then can be taken before actual sampling begins to remove any dependence on the starting point.

## 11.4 Example 11.1: The Metropolis Algorithm for One-dimensional Integration

(a) Use the algorithm of Metropolis et al. to sample the normal distribution in one dimension:

$w(x) = e^{-x^2/2}/\sqrt{2\pi}$.

For various step sizes, study the acceptance ratio (i.e. the fraction of trial steps accepted), the correlation function (and hence the appropriate sampling frequency), and the overall computational efficiency.

(b) Use the random variables you generate to calculate the integral:

$I = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} \mathrm{d}x x^2 e^{-x^2/2}$,

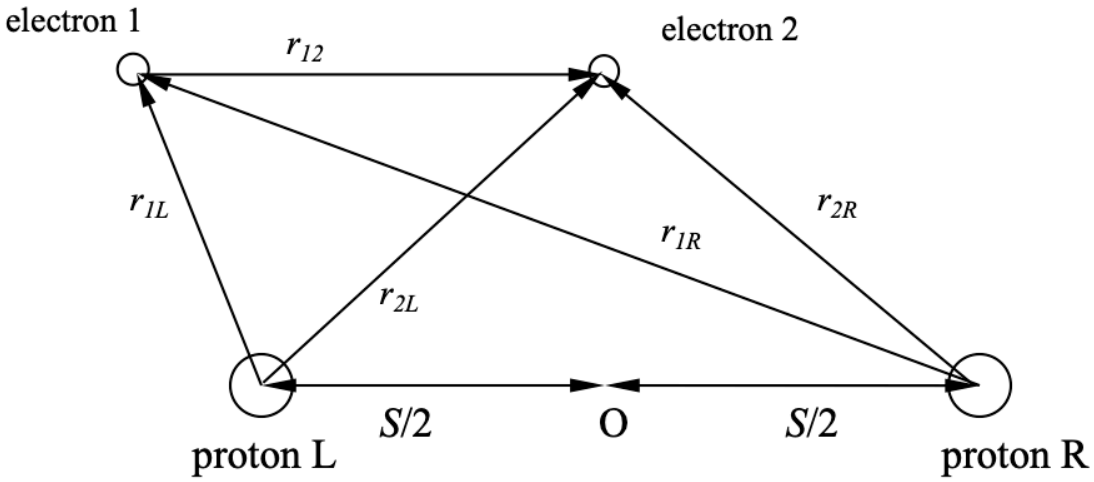
and estimate the uncertainty in your answer.

## 11.5 Quantum Monte Carlo

The algorithm of Metropolis et al. can be combined with a variational method to yield estimates of atomic and molecular ground state energies via quantum mechanics. We will examine one approach that will allow us to calculate an upper bound for the ground state energies of the Helium atom and the Hydrogen molecule ($H_2$).

### 11.5.1 The Model

The system we will consider consists of two protons and two electrons. The protons are considered to be fixed (this is known as the Born-Oppenheimer approximation), and separated by a distance $S$. For $S \neq 0$, this represents a model of the $H_2$ molecule, whereas for $S = 0$, we have a model of the Helium atom. The figure below shows the coordinates defined in the problem, with the position vectors $\mathbf{r}_1$ and $\mathbf{r}_2$ of electrons 1 and 2 are not shown for the sake of clarity.




The Schrödinger equation for the model can be separated into "nuclear" and "electronic" parts in the Born-Oppenheimer approximation.

The electronic part is given by:

$\hat{H}\psi_e(r_1, r_2, S) = \left[-\frac{\hbar^2}{2m_e}(\nabla_1^2 + \nabla_2^2) + V(r_1, r_2)\right] \psi_e(r_1, r_2, S) = E_0(S)\psi_e(r_1, r_2, S)$,

and the nuclear part:

$\left[\frac{\hbar^2}{2m_p}\nabla_S^2 + \frac{e^2}{4\pi\epsilon_0 S} + E_0(S)\right] \psi_n(S) = \epsilon\psi_n(S)$,

where $E_0(S)$ represents the electronic eigenvalue, $\epsilon$ is the total energy of the system, $m_{e,p}$ are the electron and proton masses respectively, and $\psi_{e,n}$ are the electron and proton eigenfunctions, respectively, and $V(r_1, r_2)$ is the electrostatic potential for the electrons.

Adopting "dimensionless atomic units", i.e. setting $a_0 = \frac{\hbar^2}{m_e e^2} \to 1$, where $a_0$ is the Bohr radius, lengths are effectively given in Bohr radii. The resulting unit of energy is sometimes referred to as a "Hartree", where 1 Hartree corresponds to $\sim 27.192$ eV.

In these units, the electrostatic potential for the electrons, $V(r_1, r_2)$, is given by:

$V(r_1, r_2) = -\left[\frac{1}{r_{1L}} + \frac{1}{r_{1R}} + \frac{1}{r_{2L}} + \frac{1}{r_{2R}} - \frac{1}{r_{12}}\right]$.

The potential governing the protons' motion at a separation $S(\neq 0)$, $U(S)$ is the sum of the inter-proton electrostatic repulsion and the eigenvalue $E_0(S)$ of the two-electron Schrödinger equation:

$U(S) = \frac{1}{S} + E_0(S)$.

Our goal here isa to obtain $E_0(S)$, the electronic eigenvalue.

### 11.5.2 Variational Monte Carlo

Consider a Hamiltonian, $\hat{H}$, for which we seek to estimate the ground state. The analytic variational method consists of constructing a "trial" wave function $\psi_a$, which is parameterized by one or more variational parameters, $a$. The expectation value of the energy is then calculated from:

$\langle E(a)\rangle = \frac{\langle\psi_a|\hat{H}|\psi_a\rangle}{\langle\psi_a|\psi_a\rangle}$,

where the denominator is not necessary if the trial wave function is properly normalized.

The expectation value of the energy is then minimized with respect to the variational parameters, $a$. The minimum value of $\langle E(a)\rangle$ yields an upper bound to the ground state energy of the system.

The Variational Monte Carlo (VMC) method seeks to minimize the energy numerically. In realistic systems, a many-body wave function assumes large values in small parts of the configuration space, and hence using homogeneously distributed random points to sample it is not sufficient. The algorithm of Metropolis et al. can be used to sample the required distribution, using an ensemble of random walkers moving through configuration space.

The trial wave function, denoted by $\psi_T(\mathbf{x})$, is used to define the *local energy*:

$E_L(\mathbf{x}) = \frac{\hat{H}\psi_T}{\psi_T}$,

where $\mathbf{x}$ is a multi-coordinate which represents the coordinates of all the particles in the system.

Using the above definition of the local energy, the expectation value of the energy can be written as:

$\langle E\rangle = \frac{\int \mathrm{d}\mathbf{x}\psi_T^2(\mathbf{x})E_L(\mathbf{x})}{\int \mathrm{d}\mathbf{x}\psi_T^2(\mathbf{x})}$.

The algorithm then consists of producing an initial random configuration of walkers that move around configuration space. The walker is moved within a distance $d$ of about its initial position $\mathbf{x}$, to a new position $\mathbf{x}'$. The trial step is accepted or rejected, according to the usual Metropolis algorithm, with the ratio given by:

$\rho = \frac{\psi_T^2(\mathbf{x}')}{\psi_T^2(\mathbf{x})}$.

The expectation value of the energy is calculated by averaging the *local energy* over the random walk, taking into account the correct sampling interval, and excluding the thermalization steps at the start.

The size of the trial step can be chosen so that the acceptance ratio is equal to about $0.5$ for each value of the variational parameter.

As a preparation to the full treatment of the $He$ atom and $H_2$ molecule, we investigate the Hydrogen atom in the following example. Note that a problem arises when attempting to calculate the auto-correlation function, because the trial wave function suggested by the problem is *too* close to the actual wave function. By examining the auto-correlation function, one notices that as $a \to 1$ (i.e. as the trial function approaches the actual wave function), then $C(k)$ becomes indeterminate, since the variance becomes zero at $a = 1$.

### 11.5.3 Example 11.2 Variational Monte Carlo for the Hydrogen Atom

Use the Variational Monte Carlo Method to calculate the ground state of the Hydrogen atom.

The Hamiltonian (energy) operator is:

$\hat{H} = -\frac{1}{2}\nabla^2 - \frac{1}{r}$.

Use a trial wave function of the form:

$\psi_a(r) = e^{-ar}$.

Note that for $a = 1$, the trial wave function becomes proportional to the exact ground state wave function.

The local energy for the H atom is:

$E_L(r) = -\frac{1}{r} - \frac{1}{2}a\left(a - \frac{2}{r}\right)$.

Start with a set of $N_w = 60$ walkers distributed at random in a cube of side $d = 20$.

Each walker should attempt trial steps by adding a random vector $\Delta\mathbf{r}$ to the initial position $\vec{r}$.

The trial step should be accepted or rejected according to the ratio:

$\rho = \psi_a^2(\mathbf{r} + \Delta\mathbf{r})/\psi_a^2(\mathbf{r})$.

The trial step can be varied from 0.5 to 1, adjusted to keep the acceptance ratio to $\sim 0.5$ as we vary the parameter $a$.

You may use a sampling interval of a length of $\sim 25$. The minimum is expected to occur at $a = 1$, with an energy $E = 0.5 = 13.6$ eV, with zero variance!

### 11.5.4 VMC for the H_2 Molecule and He Atom

The trial wave function suitable for the $H_2$ problem is a special case of the Padé-Jastrow wave functions. It consists of a correlated product of molecular orbitals:

$\Phi(\mathbf{r}_1, \mathbf{r}_2) = \phi(\mathbf{r}_1)\phi(\mathbf{r}_2)f(\mathbf{r}_{12})$,

where:

$\phi(\mathbf{r}_i) = e^{-r_{iL}/c} + e^{-r_{iR}/c}$,

and

$f(\mathbf{r}_{12}) = \exp\left[\frac{r_{12}}{b(1+ar_{12})}\right]$,

where $a, b, c$ are variational parameters, and the coordinates are defined in the figure describing the model.

Certain conditions are imposed on these parameters, called "cusp" conditions, which cancel the divergences in the Coulomb potential as $\mathbf{r}_{1L,R}, \mathbf{r}_{2L,R}, \mathbf{r}_{12} \to 0$.

To derive the cusp conditions, what needs to be considered for each vector separately is, e.g.:

$\lim_{r_{1L}\to 0}\left[-\frac{1}{2\phi(\mathbf{r}_1)}\nabla_1^2\phi(\mathbf{r}_1) - \frac{1}{r_{1L}}\right] = \text{finite terms}$.

After performing the differentiation for the Laplacian, the variational parameters $b, c$ are chosen so as to cancel the negative divergence of the Coulomb potential. This procedure yields:

$b = 2$ and $c = \frac{1}{1+e^{-S/c}}$, where $S$ is the inter-proton distance.

The second equation is transcendental and can be solved, e.g. via the Newton-Raphson method, each time the inter-proton separation is varied.

The remaining parameter, $a$ can be varied in order to obtain the minimum variational energy.

What remains is to calculate the local energy for the system:

$E_L(\mathbf{r}_1,\mathbf{r}_2) = \frac{\hat{H}\Phi(\mathbf{r}_1,\mathbf{r}_2)}{\Phi(\mathbf{r}_1,\mathbf{r}_2)} = \frac{1}{\Phi(\mathbf{r}_1,\mathbf{r}_2)}\left[-\frac{\hbar^2}{2m_e}(\nabla_1^2+\nabla_2^2)+V(r_1,r_2)\right]\Phi(\mathbf{r}_1,\mathbf{r}_2).$

The following identity can be employed:

$\nabla_1^2\left[f(\mathbf{r}_{12})\phi(\mathbf{r}_1)\right] = \phi(\mathbf{r}_1)\nabla_1^2 f(\mathbf{r}_{12}) + 2\left[\nabla_1 f(\mathbf{r}_1 2)\right]\cdot\left[\nabla_1\phi(\mathbf{r}_1)\right] + f(\mathbf{r}_{12})\nabla_1^2\phi(\mathbf{r}_1).$

The distances to the "left" (L) and "right" (R) proton, and the inter-electron distance can be written as:

$r_{iL} = \sqrt{x_i^2+y_i^2+(z_1+\frac{1}{2}S)^2},$

$r_{iR} = \sqrt{x_i^2+y_i^2+(z_1-\frac{1}{2}S)^2},$

for $i = 1, 2$, and

$r_{12} = \sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2}.$

One finds that:

$\nabla_i^2\phi(\mathbf{r}_i) = -\frac{2}{c}\left[\frac{e^{-r_{iL}/c}}{r_{iL}}+\frac{e^{-r_{iR}/c}}{r_{iR}}\right]+\frac{\phi(\mathbf{r}_i)}{c^2},$

$\nabla_i^2 f(\mathbf{r}_{12}) = f(\mathbf{r}_{12})\left[\frac{1}{b^2(1+ar_{12})^4}+\frac{2}{b(1+ar_{12})^3 r_{12}}\right],$

and

$2\left[\nabla_1 f(\mathbf{r}_1 2)\right]\cdot\left[\nabla_1\phi(\mathbf{r}_1)\right] = \frac{2(-1)^i f(\mathbf{r}_{12})}{bc(1+ar_{12})^2}\left[\hat{\mathbf{r}}_{iL}\cdot\hat{\mathbf{r}}_{12}e^{-r_{iL}/c}+\hat{\mathbf{r}}_{iR}\cdot\hat{\mathbf{r}}_{12}e^{-r_{iR/c}}\right],$

for $i = 1, 2$, representing each electron.

Putting everything together, we obtain:

$E_L(\mathbf{r}_1,\mathbf{r}_2) = V(\mathbf{r}_1,\mathbf{r}_2)+E_1+E_2$, with:

$E_i = -\frac{1}{2}\left[\frac{1}{(1+ar_{12})^3}+\frac{1}{c^2}+\frac{1}{4}\frac{1}{(1+ar_{12})^4}-\frac{2}{c}\frac{e^{r_{iL}/c}/r_{iL}+e^{r_{iR}/c}/r_{iR}}{e^{r_{iL}/c}+e^{r_{iR}/c}}+\frac{(-1)^i}{c}\frac{1}{(1+ar_{12})^2}\frac{e^{r_{iL}/c}\hat{\mathbf{r}}_{iL}\cdot\hat{\mathbf{r}}_{12}+e^{-r_{iR}/c}\hat{\mathbf{r}}_{iR}\cdot\hat{\mathbf{r}}_{12}}{e^{r_{iL}/c}+e^{r_{iR}/c}}\right].$

The Metropolis algorithm is applied in this case using a distribution of *six-dimensional* walkers, each representing the two electrons in the system. The minimum variational energy for the $H_2$ molecule can be determined by varying the parameter $a$ at each inter-proton separation $S$. Note that the potential energy curve for the protons in this case needs to take into account the protons' electrostatic repulsion.

To determine the upper bound for the ground state of the Helium atom, the inter-proton distance should be set to zero and the variational parameter should be varied.

### 11.5.5 Example 11.3 Variational Monte Carlo for the Helium Atom

Use the Variational Monte Carlo Method to calculate the ground state of the Helium atom (two protons in the nucleus).

Compare to the known value for the ground state energy of He: $\simeq$ 79.005 eV (https://www.nist.gov/pml/atomic-spectra-database)

You may use the local energy provided below:

```
def EnergyLocal(r, S, a):
    """Calculates the local energy for a 6-dimensional position vector r and␣
 ↪interproton distance S and variational parameter a"""
    # get the components:
    r1x = r[0]
    r1y = r[1]
    r1z = r[2]
    r2x = r[3]
    r2y = r[4]
    r2z = r[5]
```

(continues on next page)

(continued from previous page)

```
    # the vector connecting the electrons
    r12x = r1x - r2x
    r12y = r1y - r2y
    r12z = r1z - r2z

    # the unit vector connecting the electrons:
    r12 = np.sqrt(r12x**2 + r12y**2 + r12z**2)
    r12x = r12x / r12
    r12y = r12y / r12
    r12z = r12z / r12

    # get the unit vectors for proton L and R for electron 1
    r1L = np.sqrt(r1x**2 + r1y**2 + (r1z + (0.5 * S))**2)
    r1R = np.sqrt(r1x**2 + r1y**2 + (r1z - (0.5 * S))**2)
    r1Lx = r1x / r1L
    r1Ly = r1y / r1L
    r1Lz = (r1z + (0.5 * S)) / r1L
    r1Rx = r1x / r1R
    r1Ry = r1y / r1R
    r1Rz = (r1z - (0.5 * S)) / r1R

    # get the unit vectors for proton L and R for electron 2
    r2L = np.sqrt(r2x**2 + r2y**2 + (r2z + (0.5 * S))**2)
    r2R = np.sqrt(r2x**2 + r2y**2 + (r2z - (0.5 * S))**2)
    r2Lx = r2x / r2L
    r2Ly = r2y / r2L
    r2Lz = (r2z + (0.5 * S)) / r2L
    r2Rx = r2x / r2R;
    r2Ry = r2y / r2R;
    r2Rz = (r2z - (0.5 * S)) / r2R;
    # get the dot product between the unit vectors from protons L and R to electron 1␣
↪and the unit vector connecting
    # electrons 1 and 2:
    r1Lr12 = r1Lx * r12x + r1Ly * r12y + r1Lz * r12z;
    r1Rr12 = r1Rx * r12x + r1Ry * r12y + r1Rz * r12z;
    # get the dot product between the unit vectors from protons L and R to electron 2␣
↪and the unit vector connecting
    # electrons 1 and 2:
    r2Lr12 = r2Lx * r12x + r2Ly * r12y + r2Lz * r12z;
    r2Rr12 = r2Rx * r12x + r2Ry * r12y + r2Rz * r12z;

        #get c by solving the transcendetal equation
    c = NPsolve(S).x[0]

        #a is the variational param
    E1 = -0.5 * ( (( 1 + a * r12 )**(-3))/r12 + (1/c**2) + 0.25 * ( 1 + a * r12 )**(-
↪4) -
                (2 / c) * ( np.exp(-r1L / c)/r1L + np.exp(-r1R / c)/r1R) / ( np.exp(-
↪r1L / c) + np.exp(-r1R / c) )
                - (1 / c) * (1+ a * r12)**(-2) * ( np.exp(-r1L / c) * r1Lr12 + np.
↪exp(-r1R / c) * r1Rr12) / ( np.exp(-r1L / c) + np.exp(-r1R / c) ))


    E2 = -0.5 * ( (( 1 + a * r12 )**(-3))/r12 + (1/c**2) + 0.25 * ( 1 + a * r12 )**(-
↪4) -
                (2 / c) * ( np.exp(-r2L / c)/r2L + np.exp(-r2R / c)/r2R) / ( np.exp(-
```

(continues on next page)

(continued from previous page)

```
↪r2L / c) + np.exp(-r2R / c) )
                    + (1 / c) * (1+ a * r12)**(-2) * ( np.exp(-r2L / c) * r2Lr12 +␣
↪np.exp(-r2R / c) * r2Rr12) / ( np.exp(-r2L / c) + np.exp(-r2R / c) ))

   V = - 1/r1L - 1/r1R - 1/r2L - 1/r2R + 1/r12;

   # sum up and return
   #localE = V + E12 + E1 + E2
   localE = V + E1 + E2
   return localE
```

### 11.5.6 Example 11.4: Variational Monte Carlo for the Hydrogen Molecule

Use the Variational Monte Carlo Method to calculate the ground state of the Hydrogen molecule ($H_2$).