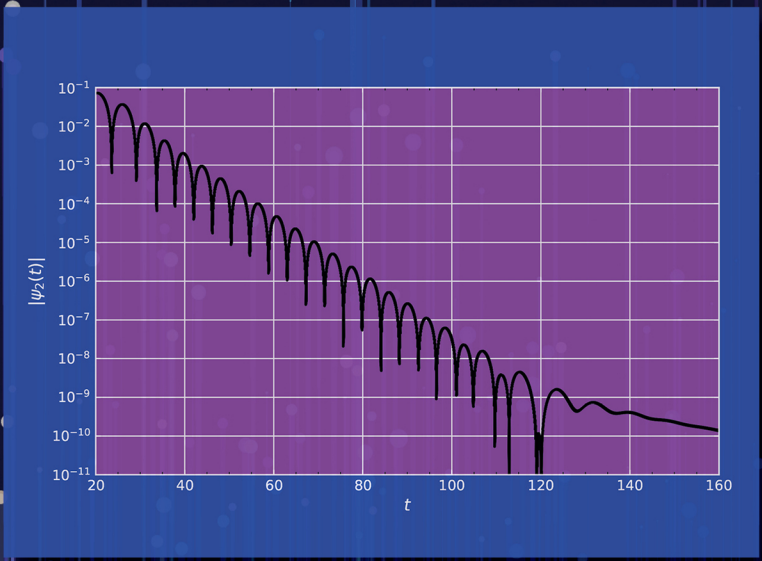
Computational Physics Using Python



Douglas M. Gingrich

CRC Press
Taylor & Francis Group

Computational Physics Using Python

This book provides a practical introduction to using computational (or numerical) methods to solve physics problems using the Python programming language, including differential equations, Fourier transforms, Monte Carlo methods, and data analysis.

It is designed with a two-level approach: topics are introduced at the lowest level, and readers encounter the simplest examples of coding the algorithm themselves before a second level introduced by the problems allows the reader to use library models and take their understanding to a higher level.

The book does not teach Python programming as students traditionally have already learnt those skills before studying computational methods, but it instead teaches readers to apply their knowledge to solve realistic physics problems.

The book is aimed at advanced undergraduate or beginning graduate students in physics or engineering. A junior-level university (or college) physics and mathematics background is assumed. But readers will not be prevented from understanding or applying numerical methods because of a lack of knowledge in a specific physics area.

Key features:

- Explores a wide spectrum of topics, from classical numerical methods to solving ordinary and partial differential equations of physics, plus spectral methods, data analysis, and Monte Carlo methods.
- Includes a chapter on data analysis and statistics, not traditionally covered in related titles on computational methods for scientists.
- Chapters are accompanied by problems and worked solutions (discussions, example code and output). Readers can access the full set of solutions under the support materials tab at: http://www.routledge.com/9781041116288.

Douglas M. Gingrich is a Professor at the University of Alberta, Canada. He obtained his PhD from the University of Toronto and has been teaching physics for over 30 years at the University of Alberta. His main research is in experimental particle physics, where he is an author of over 1700 peer-reviewed journal articles in the fields of particle physics, gravitation, astronomy, and electronics. The publications range from a single author to thousands of co-authors. He has been using computers and a multitude of programming languages to solve physics problems since computers were available to science students. He is now actively employing Python in statistical data analysis in particle physics and numerical solutions in gravity.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Computational Physics Using Python

Douglas M. Gingrich

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Designed cover image: Douglas M. Gingrich

First edition published 2026
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

ISBN: 978-1-041-09300-8 (hbk)
ISBN: 978-1-041-11628-8 (pbk)
ISBN: 978-1-003-66087-3 (ebk)

DOI: 10.1201/9781003660873

Typeset in Latin Modern font
by KnowledgeWorks Global Ltd.

Publisher's note: This book has been prepared from camera-ready copy provided by the authors.

Access the Support Materials at: https://www.routledge.com/9781041116288

Contents

Preface

Computers are an essential tool enabling physics research. Significant advancements over the last decades in processors and software allow the solution of physics problems of unprecedented scale and complexity. It is thus of paramount importance that physics students become confident with numerical methods.

This book is computational physics. The new ingredients of the book include, but are not limited to, model development, simulation, and data analysis. Statistics plays an important role in the book. Numerical methods are important and will be discussed as needed. Nevertheless, we will not discuss algorithms or efficiency in much detail. The book does not include programming, symbolic manipulation, or realtime computing.

The book is divided into roughly three broad computational topics. I begin with a biased review of some classic numerical methods. My hope is that this is not the first time you've heard of these techniques, but some coverage of the basics is necessary so that we all start with similar backgrounds.

The second area I cover is the application of numerical methods to some physics problems. These topics could be considered an advanced set of classic numerical techniques, but you will find that my coverage is hardly advanced. They are the techniques needed to solve some nontrivial physics problems numerically, and include differential equations, the Fourier transform and Monte Carlo methods.

Advancements in computing power have changed what is possible in computational physics. Data analysis techniques are more advanced than just a decade ago. In particular, the ability to apply statistical methods to interpret data and test model hypotheses has developed significantly due to computational capabilities. An introduction to these methods forms the third topic.

Arguably one of the hottest computational physics topics at the time of writing this book is the application of machine learning to solving physics problems. It is my regret that the length and scope of this book do not allow me to even introduce these topics. If you want to be at the cutting edge of computational methods in physics, this is where you want to be. For the future, you want to keep your eye on quantum computing, which has the potential to solve problems too complex for today's most advanced classical computers.

There are many, sometimes seemingly conflicting, approaches to teaching the subject of computational physics. One extreme approach is to code everything yourself, making minimal use of precoded libraries. While this is not the

way most physicists would work, it offers a pedagogical approach to obtain closer insight into the numerical methods. The idea is that the student would begin to use precoded libraries only after understanding what is inside them, knowing what is "under-the-hood". Another extreme approach is to write as little code as possible, making maximum use of precoded libraries and borrowed code from others. This is often cavalierly referred to as "not reinventing the wheel". It brings one fastest to being a working physicist at the expense of a deep understanding of numerical methods. Given enough time and experience, I believe a physicist following either of these extreme approaches will probably reach the same place. If you follow the first approach, you will eventually be making heavy use of libraries that you think you understand. While following the second approach, you will have used the libraries so often that you think you have a good grasp of what they are doing. The preferred approach is highly dependent on how each individual learns.

I will attempt to take a middle-ground approach. For a given topic, we will first code it using the simplest method ourselves. In this way, we become familiar with a bit of the terminology and concepts. Since the simplest techniques are often inadequate for real-world problems, we will then jump into using more complicated techniques offered by libraries. I will thus not discuss in any detail techniques beyond the most basic in any topic, but hopefully you will be set up to attack nontrivial physics problems using more advanced techniques offered by libraries.

This book is about applying computation methods to solving physics problems. It is not a book used to study numerical methods or a book that introduces you to new physics concepts. I will try to provide a balance between interesting physics problems that are just at the level of a senior undergraduate student without having to learn new physics concepts to solve them numerically. Hopefully at times, you, like I, will find it interesting to change the parameters or conditions of a problem, re-spin the program again, and watch what happens—a computational experiment. In this way, I hope you can have some fun and gain some insight into physics using computational methods.

The coding examples in this book are written using the Python programming language in Jupyter Notebooks. Although neither of these is necessary to learning computational physics, I have found Python to be the overweeningly favourite programming language of undergraduate students these days. I thus often take the occasion to briefly discuss the Python implementation of the problem. The use of Jupyter Notebooks is not necessary for this book, although I may have occasionally used some terminology like "cell" to describe a block of code.

This book contains a number of examples and problems. Examples usually appear at the end of a chapter but sometime directly follow the section in which the topic is discussed. The problems appear in a dedicated section at the end of each chapter. Sometimes a problem can appear quite long. This

is necessary to introduce the physics concepts leading to the problem to be solved. The length of the problem is not an indication of its difficulty.

Writing code is often not done in isolation. The codes in the examples in this book is complete. It is also well known that the internet provides abundant code that solves many of the problems in this book. At the time of writing, large language models (LLM) can be used to write code for some of the problems in this book. I recommended that these codes be used as guides for the reader when writing their own programs. At the very least, the reader is asked to test and extend the available code to the problem at hand. I believe there is pedagogical value in using someone else's code which is a valuable workplace skill to develop. But the code should be understood before used.

There is a list of books and some papers in the bibliography. In some cases, I have not explicitly cited these books in the main text. These bibliography items should therefore be regarded less as formal references than acknowledgements of those who originated some of the ideas I use in this book. The debt I owe to these authors will be quite obvious from the text.

Finally, I would like to thank the students at the University of Alberta who worked through my lecture notes for the PHYS420/PHYS580 course which became the initial draft of this book.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 1

Preliminaries

Numerical methods provide a means to solve complex physics problems. They enable us to attack problems which otherwise might not be solvable. It is common to find physical systems for which no solution in closed form exists. One important aspect in computational physics is modelling large complex systems. Some modelling techniques are impossible without computers.

It is important to make the distinction between a physical process and a model. A scientist views that physical processes and systems arise in nature according to underlying principles that we may, or may not, understand. Our job as scientists is to convince ourselves, and others, that we have some comprehension of the physical principles.

As working physicists, we usually construct a mathematical model of the physical system under investigation. This model purports to describe reality as a function of the variables in the problem. Generally, the model will have a domain within which it works well. However, if we permit the variables to go outside this domain, the model may fail.

The computer, when programmed to solve the mathematical model, thus simulates the physical process. If the computer simulation is in agreement with empirical observation, we have gained some valuable insight into the physical process. If the computer results disagree with observation, and we are convinced our simulation is implemented correctly, the model should probably be revisited.

In some cases, the statement of the problem is in effect a statement of the model. In other cases, the identification of the structure of the model may be the most demanding task in the problem. A good deal of effort may be involved in straightening out the model to be as realistic as possible.

How do we know the implementation of our simulation is correct? There is an old adage in computing that nobody writes bug-free code. The best we can achieve is to thoroughly validate our code. Avoid sharing code that does not run (or in some languages would not compile). Furthermore, just because code runs does not mean it is correct, it needs to be debugged and tested. Add debug code, especially at the beginning of code development.

DOI: 10.1201/9781003660873-1

One way that this can be achieved is using lots of print statements that can be removed—commented out—later, or turned off by using logical flags. You should also test that the results agree with an analytically calculable case if possible. Writing simpler code first to solve a similar problem before adding the elements of complexity to suit your specific problem could be beneficial. Look for redundant or alternative ways of doing tasks and compare them. Check the accuracy of your results compared to an estimated uncertainty, and state it. If a problem involves three variables, say, it is not sufficient to just plot or study one variable unless it is clear that the dependence on the other two variables is trivial. You could also validate "end-cases" and ensure that your program can handle "garbage-in" and tells the user the input is invalid. Validation can be more work than formulating the model and implementing and coding it.

When using external functions, such as those in SciPy, I suggest you check the status of completion. It is not good enough to just accept the returned result on faith. Many SciPy library modules return three types of status information: `success`, `status` and `message`. The bool `status` is useful as a condition for an if-elif-else statement to control the flow of the code depending on the success of the algorithm. The int `status` is useful for making decisions based on the specific value corresponding to different types of success or failure of the algorithm. Finally, the str `message` is useful for printing the reason behind `success == False` or if one is using verbose output.

A word of caution when comparing a numerical solution to an analytic calculation of the same problem. Many theorists love analytic calculations and think less of numerical approaches. To enable an analytic calculation of a nontrivial problem often requires making assumptions and approximations. On the other hand, the numerical calculation can sometimes be free of such simplifications and is only limited by small truncation and roundoff errors. In this sense, a numerical calculation can be superior to an analytical calculation.

Before discussing numerical methods proper, I'll say something about a few limitations of computers; computers and their associated languages are finite. Computers store numbers not with infinite precision but rather in some approximation that can be packed into a fixed memory size. A number in integer representation is exact. Arithmetic between numbers in integer representation is also exact, with the provisos that the answer is not outside the range of integers that can be represented and that division is interpreted as producing an integer result, throwing away any integer remainder. In Python, you may encounter `inf` (infinity) or `nan` (not-a-number) in your output. This is usually an indication that something is terribly wrong.

Arithmetic among numbers in floating-point representation is not exact, even if the operands happen to be exactly represented. A floating point number has a limited range. For a Python `float` the range is about $2^{\pm 1023} \equiv 10^{\pm 308}$. While this may seem wide, there are many combinations of constants in physics which are tiny. Such as Planck's constant, the charge and mass of the electron, Boltzmann's constant and so forth, depending on the units used. Be aware when performing operations with them. Using natural units is often

not a bad idea. It is also useful to be aware of potential problems when calculating factorials for large numbers yourself. The above are problems with the finite range of the exponent.

Round-off errors are another potential problem when computing. Round-off error (rounding error) is the difference between the calculated approximation of a number and its exact mathematical value. This is a form of quantization error. Round-off error is a characteristic of computer hardware. When a sequence of calculations subject to rounding error is made, errors many accumulate, sometimes dominating the accuracy of the calculation. In ill-conditioned problems, significant error may accumulate. Round-off errors are problems with the mantissa—the significant digits of the number. The mantissa usually has 16 digits in Python. To test round-off tolerance, we define ϵ_r as the smallest number that, when added to 1, returns a value different from 1. Using NumPy, the statement `numpy.finfo('float').eps` returns $\epsilon_r \equiv 2.22 \times 10^{-16}$. The above discussion is also called the precision error. It is not considering it. It is always present and can only be cleverly reduced by good coding practices.

Many numerical algorithms compute discrete approximations to some desired continuous quantity. The discrepancy between the true answer and the answer obtained in a practical calculation is called the truncation error, or approximation accuracy. It is a characteristic of the program or algorithm used, independent of the hardware. A method is unstable if any round-off or truncation errors that become mixed into the calculation at an early stage are successively magnified until they come to swamp the true answer. Clever minimization of truncation error is the business of the field of numerical analysis.

Now that we have dealt with the preliminaries, we move on to discussing, or reviewing, some classic numerical methods.

CHAPTER 2

Classic numerical methods

We begin by discussing some classic numerical methods. These topics can essentially be found in most introductory computational or numerical methods books. The purpose of this discussion is to build the foundation for subsequent chapters while introducing some solutions to physical problems, as well as, some simple Python code, use of libraries, and plotting.

The topics covered in this chapter include the calculation of series; the solution of nonlinear algebraic equations or root finding; optimization or finding the minimum of a function; interpolation of data; function approximation; matrices and eigensystems; integration of algebraic functions or data; and finite differences. The solution of differential equations is covered in detail in the subsequent chapter.

2.1 CALCULATING SERIES

Series are commonly encounter in physics. My discussion will be restricted to power series, although most of the concepts presented here apply to general series, and even recurrence relations. A typical example of a power series is the binomial expansion:

$$\frac{1}{1-x} \approx 1 + x + x^2 - \dots , \quad \text{where} \quad |x| < 1 . \tag{2.1}$$

Obviously we cannot sum an infinite number of terms, even on a computer. The idea is to stop the calculation when the change in the sum by adding another term is small compare to the previous sum. Or sum until a certain digit of accuracy is not changing any more. Don't forget to consider the convergence; approximating this series for a value of $|x| \geq 1$ will still give a result but it is unlikely to be correct.

DOI: 10.1201/9781003660873-2

I want to pause to discuss what "small" means in the above paragraph as this is a fundamental term used in numerical methods and will occur throughout the book. If s_n is the sum of the series to n terms, and s_{n-1} is the sum of the series to $(n-1)$ terms, we are saying.

$$|s_n - s_{n-1}| < \epsilon\,, \tag{2.2}$$

where ϵ a small absolute number. Choosing ϵ is not too difficult if we know the magnitude $|s_n|$. To take the magnitude $|s_n|$ into account, we often form the relative difference

$$\frac{|s_n - s_{n-1}|}{|s_n|} < \epsilon\,. \tag{2.3}$$

Now ϵ can be though of as a percentage difference, for example, and small is more intuitive. But even this definition can be troublesome for small values of $|s_n|$. In this case, I can suggest the following which will act like a fractional difference for large $|s_n|$ and absolute difference for small $|s_n|$, and be numerically safe for all $|s_n|$:

$$\frac{|s_n - s_{n-1}|}{1 + |s_n|} < \epsilon\,. \tag{2.4}$$

What is most important is communicating which formula has been used.

Series are typically divided into finite and infinite series. In principle, finite series are straight forward. For example,

$$\sum_{n=1}^{N} \frac{1}{n}\,, \quad \text{for} \quad N = 1975 \quad \text{is} \quad 8.2\,. \tag{2.5}$$

This is a real case I encountered and after struggling a bit to calculate it analytically, I just ended up calculated it numerically since the result is effectively exact and is done in negligible time.

For infinite convergent series, it is also sometimes faster to calculate them numerically than figure out the sum analytically. If a series converges rapidly, often only a few terms are needed, depending on the desire accuracy. If not, it's quick to sum hundreds of thousands of terms numerically, which except for pathological cases should give a good approximation to the infinite sum.

I now mention some considerations when calculating series. For this discussion, consider the Taylor series expansion of the function $f(x)$ about x_0:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n = \sum_{n=0}^{\infty} a_n (x - x_0)^n\,, \tag{2.6}$$

where a_n are predetermined constants independent of x. Do not calculate each term afresh. Obtain the nth term from the $(n-1)$ term and some multiplication. For example,

$$\begin{aligned} (x - x_0)^n &= (x - x_0)(x - x_0)^{n-1} \quad \text{and} \\ n! &= n(n-1)!\,. \end{aligned} \tag{2.7}$$

This allows you to use the result of the previous term—the $(n-1)$ factor on the right-hand side—to determine the next term without repeating all the calculation of the previous terms. Since we are dividing the power by the factorial, we probably benefit by calculating the nth term using

$$\frac{(x-x_0)^n}{n!} = \frac{x-x_0}{n}\frac{(x-x_0)^{n-1}}{(n-1)!}\,, \tag{2.8}$$

where the order in the calculation is that the first fraction multiplies the second fraction on the right-hand side, iteratively.

Here are a few other considerations when calculating series. Watch out for round-off errors. If successive terms in the series are similar and are being subtracted, the result could be a small number with a significant round-off error. Regrouping terms can sometimes mitigate this problem. Even if the terms in the series are integers, it may be more accurate to treat them as floating point values when performing operations; although this will add a negligible amount to the computational time. For series with terms that appear to grow, it is sometimes advantageous to reverse the order of the sum.

The main concept to be learned from this section is that it is sometimes advantageous to write the problem differently before numerical computing it. This will be a recurring theme as we go along.

2.2 SOLUTIONS OF NONLINEAR ALGEBRAIC EQUATIONS

Nonlinear algebraic equations are common in physics. Sometime a numerical solution is the only way to solve these equations. If all terms of an equation are brought to one side, we can consider it a function equal to zero: $f(x) = 0$. The interesting numerical case is when $f(x)$ is nonlinear. Numerically we can look for real roots in the interval $x = [a, b]$. I suggest plotting the function first to determine a suitable interval $[a, b]$ to search for the root. For real roots, the function needs to change sign in the interval $[a, b]$. Searching for such roots is called root finding.

I suggest one of the following two, somewhat opposite, approaches to root finding depending on your situation.

1. Root by inspection: scan $f(x)$ over a range of x values printing $f(x)$ until you see it change sign. If more accuracy is needed, reduce your scan range. This method is fast and often good enough; but only useful for one-off root finding.

2. Use a library module[1]: typically when using a library you must supply an interval $[a, b]$ in which the desired root occurs or an estimate of the root. An example root-finding library module in Python is `scipy.optimize.root`.

[1] I cavalierly do not distinguish between Python modules, packages or libraries.

Sometimes it's beneficial to find the root yourself. An example would be were roots are needed multiple times in a tight piece of code that is free of external libraries and is executed repeatedly. Then the overhead of calling a larger library module might have an impact on the performance. In cases like this, I suggest using the simple bisection method as follows.

1. Pick the interval $[a, b]$ where the root is located. This is called bracketing the root. If the root exists in this range, $f(a)f(b) < 0$ somewhere in the range.
2. Divide the range into two equal parts at x_1.
3. If $f(a)f(x_1) < 0$, the root is in $[a, x_1)$, else it is in $[x_1, b]$.
4. Continue bisecting until the desire accuracy is achieved.

In general, the method converges slowly but will always work. With a good initial bracketing, often a few iterations will give a good enough answer. Other classic methods you will hear about are Newton's method, the method of secants, the Newton-Raphson method and so forth. These algorithms are available as library modules but usually the default method in a Python library is the most state-of-the-art and recommended.

An equation can have multiple roots of which some can be degenerate and some complex. I suggest plotting the function over the region of interest to examine the possible roots. If you think making a plot is too pedantic, at least know your function well. When searching for roots, you should also watch out for pathological cases. The function might have an extrema near $y = 0$ but due to finite precision this could cause the function to not actually change sign at the root. Or alternatively, change sign when there is no root. There might be many roots close to each other, i.e. in your bracketed region. Finally, there might be a singularity in the function which is not a root.

2.2.1 Example: Root finding, a pathological case

This example examines the roots of a pathological function. It is inspired by Numerical Recipes Eq. (3.0.10) [1]. In reality my discussion deals more with computational accuracy than root finding. Consider the seemingly harmless function

$$f(x) = 3x^2 + \frac{1}{\pi^4} \ln[(\pi - x)^2] + 1\,. \tag{2.9}$$

It has two real roots that can be approximated analytically as $x = \pi \pm 10^{-\varepsilon}$ with

$$\varepsilon \approx \frac{\pi^4}{2 \ln 10}(3\pi^2 + 1) = 647\,. \tag{2.10}$$

Clearly 10^{-647} is a tiny number, but is non-zero.

In the code below, I first attempt to find the roots using different solver methods in the `scipy.optimize.root` library module. Although 10 different

methods are available, I found only three of them gave results in this case, where I am not yet making any statement about the quality of the result returned. As good practice, I also plot the function over a narrow range around where the roots should be located.

Consider the following code and its outputs. Note that the example code in this book is usually complete and will run as shown.[2] Sometimes the code is broken up into smaller cells to allow pedagogical discussion of logical sections of the example. In this case, previous code often needs to be run first to pickup the imports and definitions—as is also necessary in Jupyter notebooks.

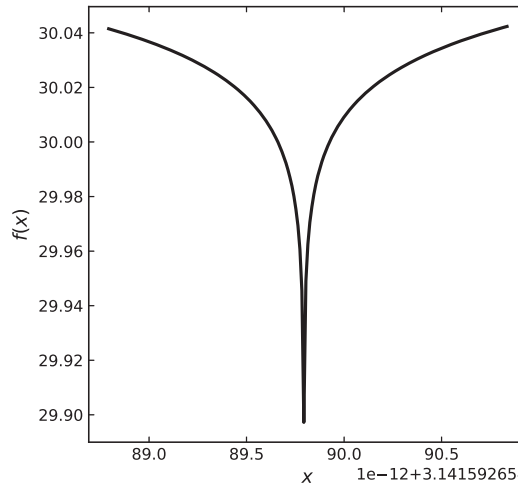
```
"""Attempt to find the roots of the function as written."""

import numpy as np
from scipy import optimize
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

def f(x):
    # Define function as written.
    return 3*x**2 + np.log( (np.pi-x)**2 ) / np.pi**4 + 1

verbose = False # Print-out flag.

# Input the dx step size parameter for plotting.
# Some interesting values.
# dx = 1e-14
# dx = 1e-15
# dx = 3e-16
# dx = 2e-16
dx = float(input("Give dx ~1e-14 "))

width = 100*dx # Half the plot range.
guess = np.pi  # Guess root for the scipy.optimize.root methods.

# Create array of x-values around pi to plot.
x = np.arange(guess-width,guess+width,dx) # Use my step size.

# Print parameters and a few interesting function evaluations.
print("number of points",len(x))
print("step size",dx)
print()
print("Some example function evaluations")
print("f(pi-dx)",f(guess-dx))
print("f(pi)",   f(guess))
print("f(pi+dx)",f(guess+dx))
print()

# Try to find the root 3 different ways.
# Initial guess np.pi + dx to avoid problems with -inf.
sol1 = optimize.root(f,guess+dx,method='hybr')
```

[2]The exception is that the Matplotlib style sheet `mystyle.mplstyle` must be available (see Appendix A), replaced by your own or commented out.

```
sol2 = optimize.root(f,guess+dx,method='lm')

sol3 = optimize.root(f,guess+dx,method='df-sane')
print(sol1.x,'hybr')
print(sol2.x,'lm')
print(sol3.x,'df-sane')
print()
if verbose:
    print(sol1,'\n')
    print(sol2,'\n')
    print(sol3,'\n')

# Plot the function near the root.
fig, ax = plt.subplots()
plt.plot(x,f(x),c='k',lw=2)
plt.xlabel(r"$x$")
plt.ylabel(r"$f(x)$")
plt.locator_params(axis='x',nbins=5)
plt.savefig('2_2_aswritten.pdf')
plt.show()
```

```
Give dx ~1e-14 1e-14
number of points 201
step size 1e-14

Some example function evaluations
f(pi-dx) 29.947375720366356
f(pi) -inf
f(pi+dx) 29.94737572036674

[3.14159265] hybr
[3.14159265] lm
-0.058391156466404255 df-sane

19: RuntimeWarning: divide by zero encountered in log
  return 3*x**2 + np.log( (np.pi-x)**2 ) / np.pi**4 + 1
```

We see that the function evaluates to $-\infty$ at π, as it should. And moving 10^{-14} away from this pole gives function values of about 29.9, which are far from being close to zero. We observe that methods `'hybr'` and `'lm'` returned single roots of π, which actually corresponds to a function value of $-\infty$. Method `'df-sane'` seems to return nonsense. You will see by turning the `verbose` flag on in the code that methods `'hybr'` and `'lm'` believe they converged while method `'df-sane'` gives up due to too many function evaluations. Also notice the `'RuntimeWarning'` of a divide by zero in the log function. This presumably is a result of an attempt to evaluate the log function at zero, or a value of its argument that cannot be represented by other than zero; while searching for a root in this function, it is almost impossible to avoid this warning.



When dealing with the numerical precision in Python, the smallest positive value that is greater than zero, the so called ϵ_m value, is important. We can get the system ϵ_m value using at least three different ways shown in the code below.

```
import sys
print(sys.float_info)
eps1 = sys.float_info.epsilon # Short version of method.
print("system epsilon =",eps1)

# Numpy method.
eps2 = np.finfo('float').eps
print("numpy epsilon =",eps2)

# Another method (ulp = "unit in the last place").
import math
eps3 = math.ulp(1.0)
print("math epsilon =",eps3)
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16,
radix=2, rounds=1)
system epsilon = 2.220446049250313e-16
numpy epsilon = 2.220446049250313e-16
math epsilon = 2.220446049250313e-16
```

Although the x values used in the code never satisfy $x < \epsilon_m$, the NumPy log function needs to be approximated numerically and I assume there are difficulties with the approximation before $x \sim \epsilon_m$.

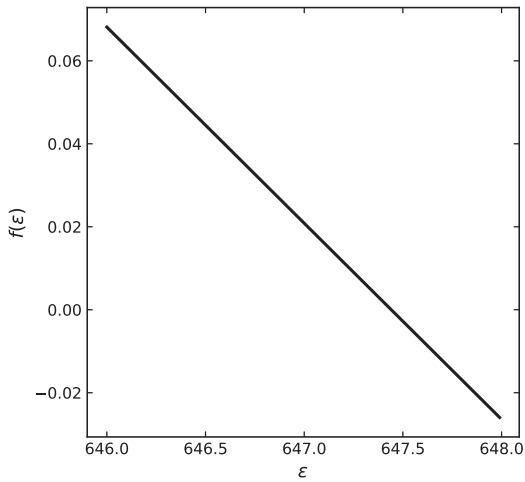
We can do better by expanding the problematic log function. Take $x = \pi \pm 10^{-\varepsilon}$ and write

$$f(\varepsilon) = 3(\pi \pm 10^{-\varepsilon})^2 + \frac{1}{\pi^4}\ln[(\pi - \pi \mp 10^{-\varepsilon}] + 1 \approx 3\pi^2 - \frac{2\ln 10}{\pi^4}\varepsilon + 1\,, \quad (2.11)$$

which is now linear in ε. Notice the approximation in the first term is $10^{-647} \approx 0$, which is valid since Python's `min_10_exp = -307`. The following code implements this safe function. I have not commented this code as it's similar to the previous code.

```
""" Find root with log evaluated."""

def f(x):
    return 3*np.pi**2 - 2*x*np.log(10) / np.pi**4 + 1

verbose = False
dx = float(input("Give dx ~1e-2 "))
width = 100*dx
guess = 647
x = np.arange(guess-width,guess+width,dx)

print("number of points",len(x))
print("step size",dx)
print()
print("some example function evaluations")
print("f("+"{:.0f}".format(guess)+"-dx)",f(guess-dx))
print("f("+"{:.0f}".format(guess),"  )", f(guess))
print("f("+"{:.0f}".format(guess)+"+dx)",f(guess+dx))
print()

sol1 = optimize.root(f,guess+dx,method='hybr')
sol2 = optimize.root(f,guess+dx,method='lm')
sol3 = optimize.root(f,guess+dx,method='df-sane')
print(sol1.x,'hybr')
print(sol2.x,'lm')
print(sol3.x,'df-sane')
print()
if verbose:
    print(sol1,'\n')
    print(sol2,'\n')
    print(sol3,'\n')

fig, ax = plt.subplots()
plt.plot(x,f(x),c='k',linewidth=2)
plt.xlabel(r"$\varepsilon$")
plt.ylabel(r"$f(\varepsilon)$")
plt.locator_params(axis='x',nbins=5)
plt.savefig('2_2_eval.pdf')
plt.show()
```

```
Give dx ~1e-4 1e-4
number of points 200
step size 0.0001

some example function evaluations
f(647.44-dx) 5.899922421903625e-05
f(647.44   ) 5.427156468229555e-05
f(647.44+dx) 4.954390513844942e-05

[647.44114796] hybr
[647.44114796] lm
647.4411479583944 df-sane
```



We now see that all three root finding methods have obtained the same value to eight significant digits: $\pi \pm 10^{-647.44114796}$. Although in this simple case, we could determine it using a calculator. The lesson to be learned from this example is not so much root finding, although you have seen it used, but that some seemingly innocent looking functions can have pathological problems. So be aware. Again, I cannot emphasize enough that plotting the function goes a long way to unveiling potential problems.

As this is the first time you have seen my code, I will make some comments on coding style. I believe that clarity is the most important principle of coding style. It improves the ability of others to understand your code. Even if you are writing code only for yourself, you may return to your code after a prolong absence and be able to understand it again quickly if it is clear. Only in rare situations does the code need to sacrifice clarity to gain an advantage in speed.

Consistency is also an important principle in coding style. I have attempted to be consistent within each example in this book. Nevertheless, I sometimes change the coding style from one example to the next to illustrate that different styles that are consistent and clear are possible.

2.3 OPTIMIZATION

The term optimization is a fancy word to get your attention. Here I will used it to mean either minimization or maximization of a real valued function, where the function does not need to be in a nice analytic form. Minimization or maximization is a common problem in physics, and other fields of science and engineering. We will encounter it when maximizing the likelihood function in Chapter 6.

The maximization can be carried out by minimizing the negative of the function. In general, we could need the minimum, or maximum, of a function in multiple variables. The variable could also be related to each other and restricted to some domain. Here, we specialize to the unconstrained case in which the space over which the minimization is performed is unrestricted. Constraint minimization is usually more difficult, although a function constraint to an interval is common.

The global minimum is the true lowest value of the function. A local minimum is the lowest value in a finite region that is not on the boundary defining the region. Most minimization algorithms are unable to determine the type of minimum they have found.

2.3.1 Univariant optimization

Let's first consider finding the minimum of a function of one variable. One must keep in mind that even if we find a minimum, we do not know if it's a global minimum.

If we could differentiate the function, we could then find the roots of the derivative. We would obtain extrema but we would not know if they were maximum, minimum or saddle points. Such methods typically do not work well.

Like in root finding, we will discuss only the simplest method—similar to a binary search—and sophisticated library functions will serve us better than any algorithms we can implement. In the case of root finding, if one is able to bracket a root with two points on the abscissa a and b such that the function values $f(a)$ and $f(b)$ have different signs, then we know a root exist in the interval $[a, b]$. However, bracketing a minimum requires three points a, b and c to be defined such that $a < b < c$, and if $f(b)$ is less than both $f(a)$ and $f(c)$, then we know a minimum exists in the interval $[a, c]$. The problem becomes how to iterate in the interval $[a, c]$ to find a minimum within some specified tolerance?

The simple algorithm goes as follows:

1. Choose two points x_1 and x_4 that bracket the minimum such that $x_1 < x_4$. Calculate x_2 to be 0.618 of the interval to the right of x_1 (and thus 0.382 to the left of x_4), and x_3 to be 0.618 of the interval to the left of x_4 (and thus 0.382 to the right of x_1). Calculate the function at all four points.

2. If $f(x_2) < f(x_3)$, the minimum lies between x_1 and x_3. In this case x_3 becomes the new x_4 and x_2 becomes the new x_3. The new x_2 is chosen by the 0.618 of the new interval. Evaluate the function at this new point.

3. Otherwise, the minimum lies between x_2 and x_4. Then x_2 becomes the new x_1 and x_3 becomes the new x_2. The new x_3 is given 0.618 of the new interval. Evaluate the function at this new point.

4. If $x_4 - x_1$ is greater than a prespecified tolerance repeat from step 2. Otherwise, calculate $(x_2 + x_3)/2$ and take this as the final estimate of the position of the minimum.

The magic number 0.618 is called the golden ratio. In principle, any number could be used but the golden ratio is the most efficient choice by reducing the interval by a constant factor each iteration.

The method leaves unspecified the choice of the initial bracketing and the tolerance. The tolerance should not be much smaller than about the square-root of the machine floating-point precision. Although robust, the algorithm does not generalize to a function of more than one variable. The above algorithm is implemented in SciPy library module `scipy.optimize.golden` but one would typically use `scipy.optimize.brent`. The Gauss-Newton and gradient decent methods could be advantageous over the above methods for some problem.

2.3.2 Multivariant optimization

Multivariant optimization is considerably more difficult. Let $\mathbf{x}$ denote the vector of independent variables. The gradient of a function f, $\nabla f(\mathbf{x})$, has vector components $\partial f/\partial x_i$. The negative of the gradient $-\nabla f(\mathbf{x})$ defines the direction of steepest decent—direction in which $f(\mathbf{x})$ decreases the most—of f at $\mathbf{x}$.

As a minimization strategy we can move along this direction for a distance γ and then iterate this scheme at the new point. The method is give by the iteration formula

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma \nabla f(\mathbf{x}_k)\,, \tag{2.12}$$

where γ is a small positive—for minimization—parameter. Each component of the gradient can be calculated using forward differences (to be discussed in Sec. 2.8). The question is how to determine γ; it should be about the size of the inverse of the second derivative, or in multiple dimensions the inverse of the Hessian. The method is guaranteed to converge to the minimum of the function, but the convergence can be quite slow. This gradient descent (steepest descent) method is the conceptual basis for many multivariant optimization algorithms.

The convergence can be improved by a modification of the gradient descent method, known as Newton's method. The method can be viewed as using a

local quadratic approximation of the function, which when minimized give an iterative scheme. The method is given by the iterative formula

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H_f^{-1}(\mathbf{x_k})\nabla f(\mathbf{x}_k)\,, \tag{2.13}$$

where we see that γ has been replaced by an inverse Hessian matrix for the function. In practice this alters both the direction and the length of the step. The method may be faster, but it is not guarantee to converge. The method is available in the SciPy library module `scipy.optimize.fmin_ncg`, standing for Newton conjugate gradient algorithm. More generally the `scipy.optimize` module provide many functions for minimization, root finding and curve fitting. Available are unconstraint minimization, as well as constraint minimization. In general, I have found the common use case `scipy.optimize.minimize` to satisfy my needs.

2.4 INTERPOLATION

Measurements usually yield a discrete set of points (x_n, y_n). Given this data, we often want an approximation of y for arbitrary x. We might be interested in only a small local region. A common method to achieve this is interpolation. Interpolation is also useful when an algorithm requires a function as input and we only have a discrete set of points. We would then interpolate the data and present the interpolated function to the algorithm.

We select a function $g(x)$ such that $g(x_n) = y_n$ for each data point. The function is thus exact at the data points. We want $g(x)$ to be a good approximation for other x-values lying between the original data points. This is like drawing a smooth curve through the points. There are many interpolating functions $g(x)$ to choose from.

Typical $g(x)$ is not a single function for all data points. More often it is a piece-wise function with parameters (like the coefficients of a polynomial) that are different for each pair of data points. We could also fit a single function to all the data to allow interpolation. In this case, the function is no longer exact at the data points. We will discuss this method (fitting) in Sec. 6.4.

Linear interpolation is the simplest approach and recommend if good enough for your case. It is often the default method used by graphics programs when drawing a line between the data points. For example, `matplotlib.plot` uses linear interpolation if `linestyle='None'` is not set. Linear interpolation is unambiguous in the sense that we know exactly how it will behave for any data.

For linear interpolation using a straight line approximation between data points $g(x) = a_0 + a_1 x$, where a_0 and a_1 are constants to be determined by the data. Consider two points x_n and x_{n+1} on either side of the point x we wish to interpolate to:

$$\begin{aligned} g(x_n) &= y_n = a_0 + a_1 x_n\,, \\ g(x_{n+1}) &= y_{n+1} = a_0 + a_1 x_{n+1}\,. \end{aligned} \tag{2.14}$$

The two parameters a_0 and a_1 are determined using

$$\begin{aligned} a_1 &= \frac{y_{n+1} - y_n}{x_{n+1} - x_n}\,, \\ a_0 &= \frac{x_{n+1}y_n - x_n y_{n+1}}{x_{n+1} - x_n}\,. \end{aligned} \tag{2.15}$$

The linear interpolating equation can be written on the interval $[x_n, x_{n+1}]$ using only the data points:

$$g(x) = \frac{x - x_n}{x_{n+1} - x_n} y_{n+1} + \frac{x - x_{n+1}}{x_n - x_{n+1}} y_n\,. \tag{2.16}$$

This is only one of a few equivalent ways to write this equation.

For each data interval, one has a different set of coefficients. This is different from fitting a single function to data, which has a single set of coefficients over the entire data range and does not go exactly though the data points.

Linear interpolation works well when the distance to interpolate is small and the function is smooth (i.e. when the second and higher derivatives are small). The linear method uses only two data points. Using a higher-order polynomial for the interpolating function may give a more accurate interpolation. A quadratic interpolation, for example, would require three data points.

$$\begin{aligned} g(x) = {} & \frac{(x - x_n)(x - x_{n+1})}{(x_{n+2} - x_n)(x_{n+2} - x_{n+1})} y_{n+2} + \frac{(x - x_n)(x - x_{n+2})}{(x_{n+1} - x_n)(x_{n+1} - x_{n+2})} y_{n+1} \\ & + \frac{(x - x_{n+1})(x - x_{n+2})}{(x_n - x_{n+1})(x_n - x_{n+2})} y_n\,. \end{aligned}$$

In the limit when all the points are used, the method is called Lagrange interpolation, and can be expressed as

$$g(x) = \sum_{i=1}^{N} \lambda_i(x) y_i\,, \quad \text{where} \quad \lambda_i(x) = \frac{\prod_{j=1\neq i}^{N}(x - x_j)}{\prod_{j=1\neq i}^{N}(x_i - x_j)}\,. \tag{2.17}$$

Using too high a polynomial degree is likely to introduce unphysical wiggles in the interpolation, the so-called Runge's phenomenon.

Other interpolation methods exist. A popular classic method is Neville's algorithm. The method uses an iterative approach of linear interpolations. Each successive interpolation increases the degree of the polynomial. Another method is to use rational function interpolation. That is, a function that is a ratio of two polynomials. The method works wells if poles in the interpolating function are required. But beyond linear interpolation, I recommend using a library module such as `scipy.interpolate.interp1d` with other than the default argument `kind='linear'` rather than coding it yourself.

Piece-wise polynomial interpolation gives discontinuities in the derivatives at the data points. The method of spline interpolation uses all the data points to determine the coefficients. Thus nonlocal information is used to achieve

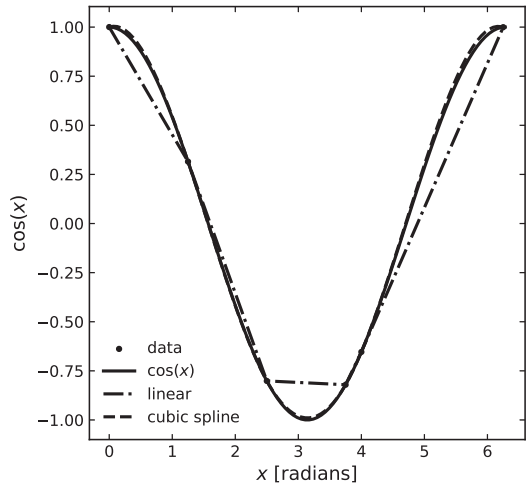
global smoothness up to some order in the derivatives. Cubic splines is a method that produces an interpolating function that is continuous up to, and including, the second derivative.

Sometimes splines don't have much advantage over local methods, but can be useful for sparse data. Note that using splines is not the same as fitting a function to data. There is no such thing as convergence when using splines; the calculation is exact. A Python library class for calculating cubic splines is `scipy.interpolate.CubicSpline`.

2.4.1 Example: Linear interpolation and cubic splines

The prototypical example to demonstrate interpolation is a cosine function which has nontrivial curvature. Consider a set of ideal measurements performed at the unevenly spaced values $x = (0.0, 1.25, 2.5, 3.75, 4.0, 6.25)$. I will plot linear interpolations, cubic splines and theory, where "theory" is my way of saying the true analytical values. The linear interpolation is done automatically by the plotting module `matplotlib.pyplot`. Consider the following code and resulting plot.

```
"""Liner interpolation and cubic splines."""

import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

# Theory.
x1 = np.linspace(0,2*np.pi,100)
y1 = np.cos(x1)

# Linear interpolation between data points (matplotlib default).
x2 = np.array([0.0,1.25,2.5,3.75,4.0,6.25])
y2 = np.cos(x2)

# Cubic splines
f  = CubicSpline(x2,y2)
y3 = f(x1)

# Superimposed on a single plot.
fig, ax = plt.subplots()
plt.scatter(x2,y2,marker='.',c='k',label="data")
plt.plot(x1,y1,ls='-', c='k',label=r"$\cos(x)$")
plt.plot(x2,y2,ls='-.',c='k',label="linear")
plt.plot(x1,y3,ls='--',c='k',label="cubic spline")
plt.xlabel(r"$x$ [radians]")
plt.ylabel(r"$\cos(x)$")
plt.legend()
plt.savefig('2_3_interpolation.pdf')
plt.show()
```



As expected, we see in the figure that linear interpolation is terrible where there is lots of curvature. Cubic splines do better but are still not ideal.

2.5 FUNCTION APPROXIMATION

Function approximation is a method for approximating a given function by another function; often an approximating polynomial. The method is useful when the cost of evaluating the original function is high, such as if the function is a result of a simulation or complex calculation. Every continuous function defined in the interval $[a, b]$ can be approximated to arbitrary accuracy by a polynomial function. The "minmax" polynomial is the one that minimizes the difference between the two functions.

A truncated Chebyshev series closely approximates the minmax polynomial. Chebyshev polynomials are orthogonal over the interval $[-1, 1]$, as shown in Fig. 2.1. Sometimes we may need to recast the function to be approximated in this range. If x is in the range $[-1, 1]$, then

$$x' = \frac{b-a}{2}x + \frac{b+a}{2} \tag{2.18}$$

will be in the range $[a, b]$. The approximation of function $g(x)$ using Chebyshev polynomials $T_n(x)$ is

$$g(x) \approx \sum_{n=0}^{N-1} c_n T_n(x) - \frac{1}{2}c_0\,. \tag{2.19}$$

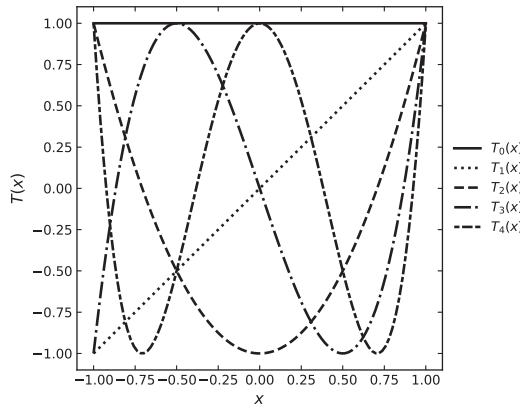


Figure 2.1 The first five Chebyshev polynomials of the first kind.

The numerical problem is to determine the c_n for $n = 0, \ldots, N-1$. Once the c_n are determined, we can truncate the approximation to $M < N$ if we like:

$$g(x) = \sum_{n=0}^{M-1} c_n T_n(x) - \frac{1}{2} c_0 \,. \tag{2.20}$$

This is a feature of Chebyshev polynomials.

The process of function approximation involves two steps. The first step is to construct the interpolating function by calculating the c_n. This should only be done once. The second step is to evaluate the approximating function at a given x using the c_n determined in the previous step and the basis functions $T_n(x)$. The aim is that the second step is fast and can be performed often.

When determining the c_n, we will use library modules that do it for us. Under-the-hood this is done for us by using a series of, so-called, Chebyshev points—or Chebyshev nodes. These are an optimal set of points that occur at the extrema of the Chebyshev polynomials. The point spacing is not uniform, with more points clustered near the ± 1 end points. Using Chebyshev points for approximations using other than Chebyshev polynomials is also a good choice. The Chebyshev points can also be viewed as the equidistant points on a unit half circle projected onto the bisecting diameter, as shown in Fig. 2.2.

Some useful Python library modules for approximating functions using Chebyshev polynomials are `numpy.polynomial.chebyshev.chebinterpolate` and
`numpy.polynomial.chebyshev.chebval`.

2.5.1 Example: Polynomial function approximation

We will write a program that will approximate a function of one variable using Chebyshev polynomials. It will plot the original function and overlay

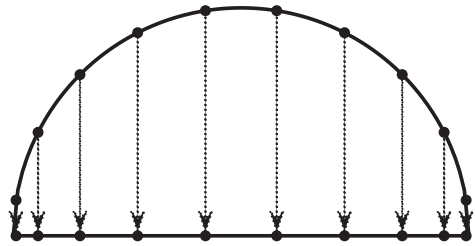


Figure 2.2 Ten Chebyshev nodes in the range $[-1, 1]$ illustrated by the projection of equally spaced points on a unit semicircle onto the diameter.

the approximate function. Often the approximation is too good to view the differences between the real function and the approximate function on an overlay plot, so we will also plot the difference between the real function and the approximate function. Consider the two functions:

$$f(x) = e^{\sin(\pi x)} \quad \text{and} \quad f(x) = \frac{1}{1 + 25x^2}\,. \tag{2.21}$$

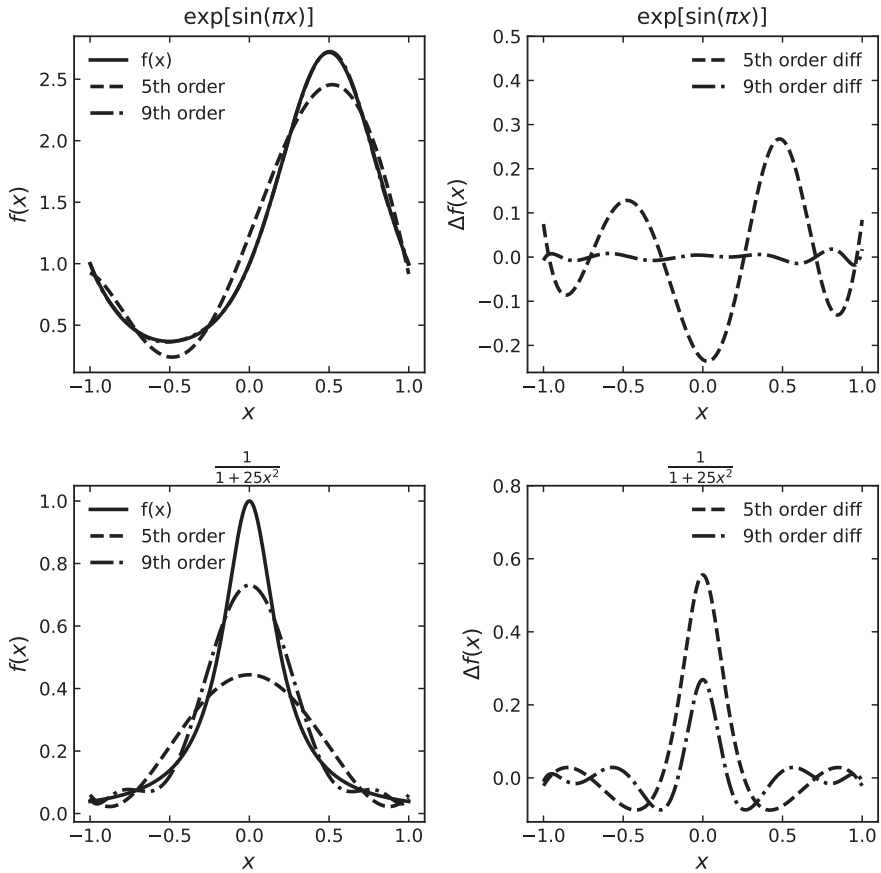
The following code shows results for 5th degree and 9th degree Chebyshev polynomial approximations. A step size of $\Delta x = 0.001$ over the interval $x = [-1, 1]$ was used.

```
"""Chebyshev polynomial function approximation"""

import numpy as np
import numpy.polynomial.chebyshev as ch
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

# First test function.
def f1(x):
    return np.exp(np.sin(np.pi*x))

# Second test function.
def f2(x):
    return 1 / (1 + 25*x**2)

# Interpolate a function at Cheyshev points.
def cheby(f,degree):
    return ch.chebinterpolate(f,degree)

dx = 0.001
x = np.arange(-1,1+dx,dx)
print("Using",len(x),"function values")
# Interpolate the test functions with Chebyshev polynomials of
# degrees 5 and 9.
```

```
ch15 = ch.chebval(x,cheby(f1,5))
ch19 = ch.chebval(x,cheby(f1,9))
ch25 = ch.chebval(x,cheby(f2,5))
ch29 = ch.chebval(x,cheby(f2,9))

fig, ax = plt.subplots(2,2,figsize=(7,7))

# Plot test function 1.
ax[0,0].plot(x,f1(x),ls='solid',  label="f(x)")
ax[0,0].plot(x,ch15, ls='dashed', label="5th order")
ax[0,0].plot(x,ch19, ls='dashdot',label="9th order")
ax[0,0].set_title(r"$\exp[\sin(\pi x)]$")
ax[0,0].set_xlabel(r"$x$")
ax[0,0].set_ylabel(r"$f(x)$")
ax[0,0].legend()

ax[0,1].plot(x,f1(x)-ch15,ls='dashed', label="5th order diff")
ax[0,1].plot(x,f1(x)-ch19,ls='dashdot',label="9th order diff")
ax[0,1].set_title(r"$\exp[\sin(\pi x)]$")
ax[0,1].set_xlabel(r"$x$")
ax[0,1].set_ylabel(r"$\Delta f(x)$")
ax[0,1].set_ylim(top=0.5)
ax[0,1].legend()

# Plot test function 2.
ax[1,0].plot(x,f2(x),ls='solid',  label="f(x)")
ax[1,0].plot(x,ch25, ls='dashed', label="5th order")
ax[1,0].plot(x,ch29, ls='dashdot',label="9th order")
ax[1,0].set_title(r"$\frac{1}{1+25x^2}$")
ax[1,0].set_xlabel(r"$x$")
ax[1,0].set_ylabel(r"$f(x)$")
ax[1,0].legend(loc='upper left')

ax[1,1].plot(x,f2(x)-ch25,ls='dashed', label="5th order diff")
ax[1,1].plot(x,f2(x)-ch29,ls='dashdot',label="9th order diff")
ax[1,1].set_title(r"$\frac{1}{1+25x^2}$")
ax[1,1].set_xlabel(r"$x$")
ax[1,1].set_ylabel(r"$\Delta f(x)$")
ax[1,1].legend()
ax[1,1].set_ylim(top=0.8)

plt.tight_layout()
plt.savefig('2_4_funapprox.pdf')
plt.show()
```

For the first function, the 9th order approximation is significantly better than the fifth-order approximation, as expected. The ninth-order approximation is so good that it is hidden under the function in the top-left figure.

The second test function is not approximated well even up to 9th order. The function is exhibiting the Runge phenomenon, which is a problem of oscillation at the edges of an interval that occurs when using polynomial interpolation with polynomials of high degree over a set of equispaced interpolation points, even though the Chebyshev nodes are used.



2.6 MATRICES AND EIGENSYSTEMS

Physicists often need to manipulate matrices. Applications include systems of linear algebraic equations, transformations (matrix multiplication), the description of a system of particles, calculating principle moments of inertia, evaluating electric circuits, describing coupled harmonic oscillators, eigenvalue problems in quantum mechanics and so forth. In addition, some of the topics we will discuss in differential equations can be solved by a system of equations.

Using the inverse matrix is often not the most accurate method for solving a system of equations and can lead to problems. If each equation in the system of equations is not independent, the solution is not unique. The matrix will have no inverse. A matrix without an inverse is call singular. Sometimes a matrix is not singular but it is close so that round-off errors can make it appear singular.

You should first determine if the matrix is ill-conditioned. A matrix is considered to be ill-conditioned if it is almost singular. The condition of a

matrix indicates how close it is to being singular. The condition number of a matrix estimates how sensitive the answer is to perturbations in the input data and to round-off errors during the solution process. There is more than one definition of the condition number but the use is the same. If the condition number is close to unity it is good, if the condition number is large it is bad. For systems of equations, the condition number cannot be less than unity and values above about 100 can be considered bad [2].

Matrix methods you've previously encounter such as Gaussian elimination with back (or forward) substitution could be used. Pivoting (changing the order of the equations) is a useful technique in many situations. The method of LU decomposition is often essential in physics since it's good for sparse matrices and uses memory efficiently.

When working with matrices, I suggest using Python library modules. For simple analytical matrices, SymPy is a good choice. For numerical matrices `numpy.linalg` and `scipy.linalg` (uses a FORTRAN wrapper) are good choices.

Perhaps even more importance than systems of equations in physics are eigenvalue problems. I'll review the important aspect of eigensystems needed for numerical calculations. The eigenequation is

$$\mathbf{A}\mathbf{x}_i = \lambda_i \mathbf{x}_i \,, \tag{2.22}$$

where $\mathbf{A}$ is an $N \times N$ matrix, $\mathbf{x}_i$ are N eigenvectors and λ_i are N eigenvalues. For computation, the eigenequation is usually written as

$$(\mathbf{A} - \lambda_i \mathbf{I})\mathbf{x}_i = 0 \,, \tag{2.23}$$

where $\mathbf{I}$ is the unit matrix. The equation can only be true if $\det|\mathbf{A} - \lambda\mathbf{I}| = 0$, which is equivalent to an Nth-order polynomial in λ. The roots of this polynomial λ_i are the eigenvalues, which in general are complex numbers.

The eigenvalue problem can also be written as

$$\mathbf{X}^{-1}\mathbf{A}\mathbf{X} = \mathbf{\Lambda} \,, \tag{2.24}$$

where $\mathbf{\Lambda}$ is a diagonal matrix with elements which are the eigenvalues λ_i and $\mathbf{X}$ is a matrix whose columns are the eigenvectors $\mathbf{x}_i$. The matrix $\mathbf{A}$ is said to have been diagonalized since the combined matrix on the left is diagonal. If the matrix is symmetric, elements of $\mathbf{A}$ satisfy $a_{ij} = a_{ji}$, and the eigenvalues are real. If the matrix is Hermitian, $\mathbf{A} = \mathbf{A}^\dagger$ or $a_{ij} = a^*_{ji}$. Eigenvalues of a Hermitian matrix are all real. If $\mathbf{A}$ and $\mathbf{x}$ are complex, we can write the eigenequation as a $2N \times 2N$ matrix:

$$(\mathbf{C} + i\mathbf{D})(\mathbf{u} + i\mathbf{v}) = \lambda(\mathbf{u} + i\mathbf{v}) \,, \tag{2.25}$$

or

$$\begin{pmatrix} \mathbf{C} & -\mathbf{D} \\ \mathbf{D} & \mathbf{C} \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix} = \lambda \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix} . \tag{2.26}$$

The condition number is defined differently when determining eigenvalues or eigenvectors, but the idea is the same; small condition numbers are good.

2.6.1 Example: Matrices and eigensystems

I will present three simple non-physics matrix examples to demonstrate the use of the library modules.

2.6.1.1 System of equations

Consider the system of three equations:

$$\begin{aligned} 4x + 3y + 2z &= 25 \\ -2x + 2y + 3z &= -10 \\ 3x - 5y + 2z &= -4 \end{aligned} \quad . \tag{2.27}$$

In matrix notation

$$\begin{pmatrix} 4 & 3 & 2 \\ -2 & 2 & 3 \\ 3 & -5 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 25 \\ -10 \\ -4 \end{pmatrix} . \tag{2.28}$$

In algebraic notation $\mathbf{Ax} = \mathbf{b} \quad \Rightarrow \quad \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

The following code first determines some properties of the matrix, it then finds the inverse matrix, checks that the matrix times its inverse gives the identity matrix (a validation method I call closure), it then solves for $\mathbf{x}$. I will also show how to solve for $\mathbf{x}$ in one go without calculating the inverse matrix.

```
"""System of equations."""

import numpy as np

# Fill Matrix.
A = np.array([[4,3,2],[-2,2,3],[3,-5,2]])
print("A =")
print(A,"\n")

# Print some characteristics of matrix.
print("rank =",np.linalg.matrix_rank(A))
print("norm =",np.round(np.linalg.norm(A),2))
print("determinant =",np.round(np.linalg.det(A),2))
print("condition =",np.round(np.linalg.cond(A),2),"\n")

# Fill column vector.
b = np.array([25,-10,-4])
print("b =",b,"\n")

# Find matrix inverse.
Ainv = np.linalg.inv(A)
print("A inverse =")
print(np.round(Ainv,3),"\n")

# Check inverse matrix gives idenity matrix (validation).
I = A.dot(Ainv)
```

```
print("check of inverse: I =")
print(I,"\n")

# Calculate solution vector.
x = np.linalg.inv(A).dot(b)
print("solution: x =",x,"\n")

# Calculate solution vector in one go.
x2 = np.linalg.solve(A,b)
print("solution all in one go: x =",x2)
```

```
A =
[[ 4  3  2]
 [-2  2  3]
 [ 3 -5  2]]

rank = 3
norm = 9.17
determinant = 123.0
condition = 1.87

b = [ 25 -10  -4]

A inverse =
[[ 0.154 -0.13   0.041]
 [ 0.106  0.016 -0.13 ]
 [ 0.033  0.236  0.114]]

check of inverse: I =
[[ 1.00000000e+00 -5.55111512e-17  0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00 -5.55111512e-17]
 [ 5.55111512e-17  0.00000000e+00  1.00000000e+00]]

solution: x = [ 5.  3. -2.]

solution all in one go: x = [ 5.  3. -2.]
```

We see that the condition number of about two is small so the matrix should not give a problem. The closure test gives the identity matrix to better than 16 digits of accuracy. The solution, determined by either method, is essentially exact.

2.6.1.2 Analytic problem

This example is a little different than I usually show and is inspired by Robert Johansson [3]. It is a study of the condition number and a comparison between the analytical (using SymPy) and numerical solutions.

Consider the simple system of two equations

$$\begin{pmatrix} 1 & p \\ 1 & 2-p \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \tag{2.29}$$

where p is a parameters. If $p = 1$, the matrix is singular, and for p in the vicinity of one the matrix is ill-conditioned.

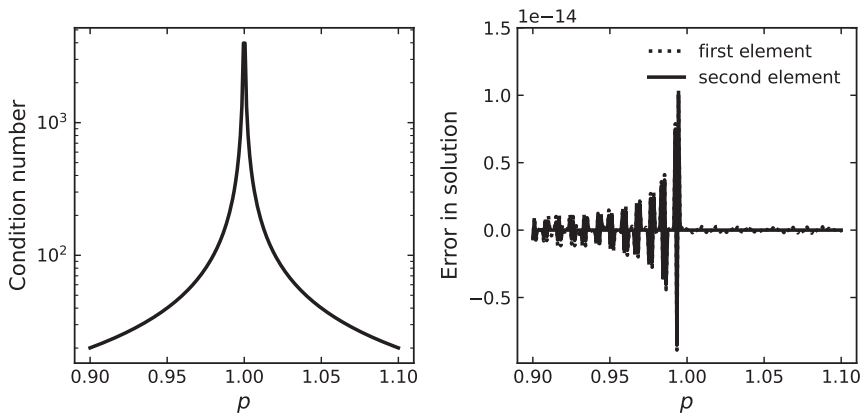
```
"""Analytic example."""

import sympy
sympy.init_printing()

p = sympy.symbols("p",positive=True)
A = sympy.Matrix([[1,p],[1,2-p]])
b = sympy.Matrix([1,2])
x_sym_sol = A.solve(b)
print("Solution:")
x_sym_sol
```

Solution:

$$\begin{pmatrix} \frac{3p-2}{2p-2} \\ \frac{1}{2-2p} \end{pmatrix}$$

```
"""
1) Look at symbolic condition number versus p.
2) Compare numerical solution with analytical solution.
"""

import numpy as np
import matplotlib.pyplot as plt
plt.style.use('./mystyle.mplstyle')

# Symbolic solution (relies on previous cell being executed).
Acond = A.condition_number().simplify()

# Numerical solution.
AA = lambda p: np.array([[1,p],[1,2-p]])
bb = np.array([1,2])
x_num_sol = lambda p: np.linalg.solve(AA(p),bb)

# Make some plots for different p values.
fig, axes = plt.subplots(1,2,figsize=(7,3.5))

# Plot condition number.
p_vec = np.linspace(0.9,1.1,200)
axes[0].plot(p_vec,[Acond.subs(p,pp).evalf() for pp in p_vec],
             c='k',lw=2)
axes[0].set_xlabel(r"$p$")
axes[0].set_ylabel(r"Condition number")
axes[0].set_yscale('log')

# Plot the difference between the symbolic (exact)
```

```
# and numerical results.
# Consider both elements of solution vector separately.
style = ['dotted','solid']
lable  = ['first element','second element']
for i in range(2):
    x_sym = np.array([x_sym_sol[i].subs(p,pp).evalf()
                      for pp in p_vec])
    x_num = np.array([x_num_sol(pp)[i] for pp in p_vec])
    axes[1].plot(p_vec,(x_num-x_sym)/x_sym,
                 ls=style[i],label=lable[i])
axes[1].set_xlabel(r'$p$')
axes[1].set_ylabel("Error in solution")
axes[1].legend()
axes[1].set_ylim(top=1.5e-14)
plt.tight_layout()
plt.savefig('2_5_condition.pdf',bbox_inches='tight')
plt.show()
```



We see that the analytic solution has both components approach infinity as $p \to 1$. The condition number becomes very large at $p = 1$ but is reasonably small for $|p - 1| > 0.1$ The numerical and symbolic results agree well (oscillatory differences). Notice the small y-axis range on the second plot.

2.6.1.3 Eigensystem problem

Consider the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 1 & 0 & -2 \end{pmatrix}. \tag{2.30}$$

What are the eigenvectors and eigenvalues? $\mathbf{Ax} = \lambda\mathbf{x}$
The following code is straightforward

```
"""Eigensystem example."""

import numpy as np
from numpy import linalg as la

debug = False

# Fill matrix.
A = np.array([[1,2,3],[3,2,1],[1,0,-2]])
print("A =")
print(A,"\n")

# Calculate eigenvectors and eigenvalues.
w, v = la.eig(A)

# Print eigenvectors v and eigenvalues w.
if debug: print("eigenvectors",v)
print("eigenvector 1",np.round(v[:,0],3))
print("eigenvector 2",np.round(v[:,1],3))
print("eigenvector 3",np.round(v[:,2],3))
print("\neigenvalues",np.round(w,3))
```

```
A =
[[ 1  2  3]
 [ 3  2  1]
 [ 1  0 -2]]

eigenvector 1 [-0.58  -0.809 -0.093]
eigenvector 2 [-0.516  0.8   -0.307]
eigenvector 3 [-0.664  0.261  0.7  ]

eigenvalues [ 4.267 -0.318 -2.949]
```

We now validate the result by comparing the left-hand and right-hand sides of the eigenequations.

```
u = v[:,0]
print("Ax\u2081",np.dot(A,u))
lam = w[0]
print("\u03BBx\u2081",lam*u,"\n")

u = v[:,1]
print("Ax\u2082",np.dot(A,u))
lam = w[1]
print("\u03BBx\u2082",lam*u,"\n")

u = v[:,2]
print("Ax\u2083",np.dot(A,u))
lam = w[2]
print("\u03BBx\u2083",lam*u,"\n")
```

```
A.x1 [-2.47634761 -3.45191057 -0.39516411]
lambda.x1 [-2.47634761 -3.45191057 -0.39516411]

A.x2 [ 0.16400413 -0.2543268   0.09750198]
lambda.x2 [ 0.16400413 -0.2543268   0.09750198]

A.x3 [ 1.95896437 -0.77030053 -2.06491426]
lambda.x3 [ 1.95896437 -0.77030053 -2.06491426]
```

Which agrees exactly to the number of digits printed.

These are three simple examples, but I again emphasis the need to validate your results. In this case, by performing closure tests.

2.7 INTEGRATION

Integration has many uses in physics. For example, we often need to normalize functions and distributions. We start with the definition of a definite integral. Consider N points $x_0, x_1, \ldots, x_{N-1}$ with $\Delta x = (x_{N-1} - x_0)/(N-1)$, as shown in Fig. 2.3. If $x_0 \leq c_0 \leq x_1, x_1 \leq c_1 \leq x_2, \ldots, x_{N-2} \leq c_{N-2} \leq x_{N-1}$, the fundamental theorem of calculus states

$$\int_a^b f(x)dx = \lim_{\Delta x \to 0} \sum_{i=0}^{N-1} f(c_i)\Delta x\,. \tag{2.31}$$

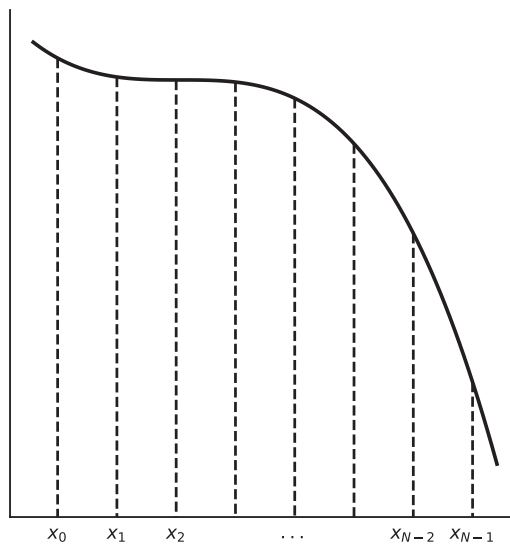


Figure 2.3 Points and slices used when integrating a function from x_0 to x_{N-1}.

This can be generalize to variable $\Delta x \to \Delta x_i$. A two dimensional version can be written as

$$\iint_A f(x,y)dA = \lim_{\Delta A \to 0} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} f(c_i, d_j)\Delta x \Delta y \,. \tag{2.32}$$

Several considerations are necessary when evaluating the sums on the right-hand side. For the integrand, is it a function or data points? Does the integrand have singularities? For the limits of integration, are we evaluating a definite or indefinite integral? Multidimensional integrals are common but they are often treated using Monte Carlo methods (see Sec. 5.3).

The simplest integration method is intuitive. Taking $c_i = x_i$,

$$\int_a^b f(x)dx = \lim_{\Delta x \to 0} \sum_{i=0}^{N-2} f(x_i)\Delta x \,. \tag{2.33}$$

This is the rectangular method with the function evaluated at the lower boundary of the Δx slices (or strips, or panels); the last point is not used. We can also take, $c_i = x_{i+1}$. This is the rectangular method with the function evaluated at the upper boundary of the slices; the first point is not used. Or we can use $c_i = (x_{i+1} + x_i)/2$ which is called the mid-point method. Although the mid-point method is still a rectangular approximation it can turn out to be significantly more accurate for some integrands. The mid-point method is equivalent to taking $\langle f(x_i)\rangle = [f(x_i) + f(x_{i+1})]/2$ which is related to the trapezoid rule, describe below.

In calculations, we want to take Δx small but be cautious to not take it less than floating-point precision. There is also a tradeoff in speed by taking Δx too small, due to the large number of function evaluations, but this is usually insignificant on modern computers.

In developing more elaborative methods, we often develop different elementary interval methods and then generalize them to the entire range of integration. Approximating the integral by a polynomial in each integration slice, we write the following three cases,

$$\begin{aligned}
&\int_{x_i}^{x_{i+1}} f(x)dx \approx \int_{x_i}^{x_{i+1}} C dx && \text{rectangular,} \\
&\int_{x_i}^{x_{i+1}} f(x)dx \approx \int_{x_i}^{x_{i+1}} (Bx + C)dx && \text{trapezoid,} \\
&\int_{x_i}^{x_{i+2}} f(x)dx \approx \int_{x_i}^{x_{i+2}} (Ax^2 + Bx + C)dx && \text{parabolic (Simpson's method).}
\end{aligned} \tag{2.34}$$

Simpson's method requires an even number of slices. The above are referred to as Newton-Cotes quadrature rules and are shown in Fig. 2.4. They use fixed spacing and N-point. The above three elemental forms lead to simple sums you can code yourself.

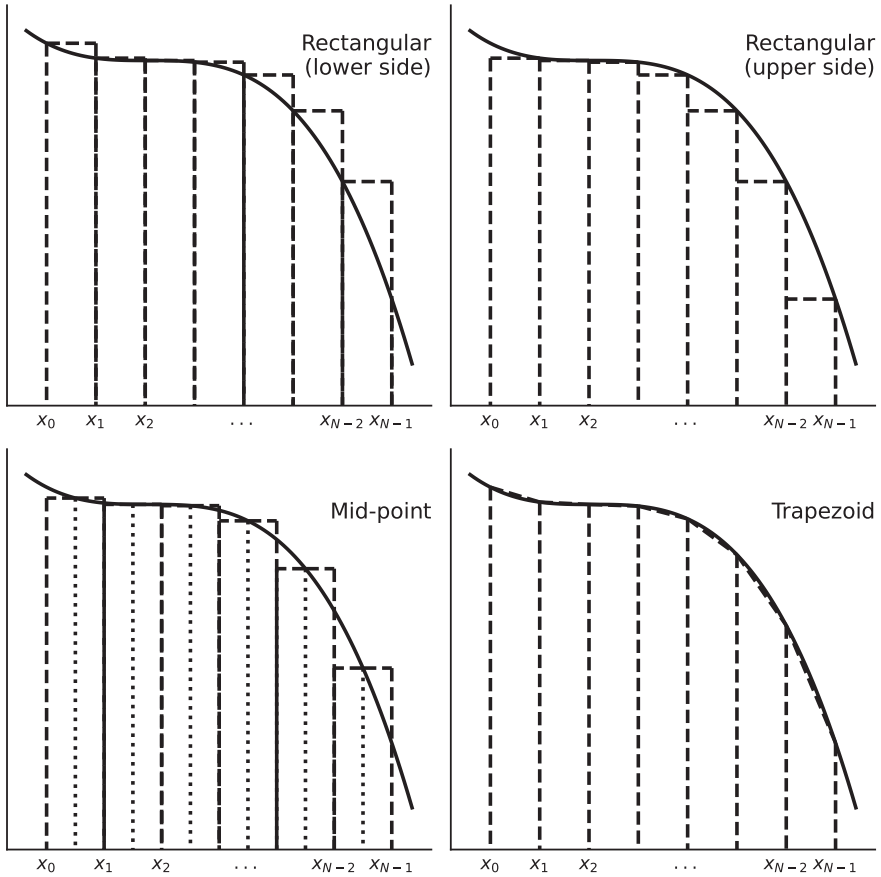


Figure 2.4 Point and slice schemes for the lowest-order Newton-Cotes methods.

For the entire integration range in general,

$$\int_a^b f(x)dx = \sum_{i=0}^{N-1} w_i f(x_i)\,, \tag{2.35}$$

where

rectangular rule ($c_i = x_i$): $w_i = \Delta x(1, 1, 1, \ldots, 1, 1, 0)$.

trapezoid rule: $w_i = \frac{\Delta x}{2}(1, 2, 2, \ldots, 2, 2, 1)$.

Simpson's rule: $w_i = \frac{\Delta x}{3}(1, 4, 2, 4, \ldots, 2, 4, 1)$ odd number of points.

The rectangular rule uses the endpoints once except for the upper boundary which is not used. The trapezoid rule uses the endpoints twice except at the two boundaries. The trapezoid rule is surprisingly good at integrating periodic functions when the difference in the integration limits is not a period [2].

Many more advanced methods have been developed. Adaptive quadrature uses more than one method or changes the step size or both based on some convergence criteria. Some methods approximate the integral with a polynomial over the entire range of integration. Examples are Gaussian integration with Legendre, Laguerre, Hermite or Chebyshev polynomials. The points are usually evaluated at the roots of the polynomial (uneven spacing).

Quadrature by a transformation of variables can be an effective approach. In any method of integration, you may need to remap the range of integration if you are not coding it yourself using $[a, b] \to [-1, +1]$:

$$\int_a^b f(x)dx \quad \longrightarrow \quad \frac{b-a}{2}\int_{-1}^{+1} f\left(\frac{b-a}{2}x + \frac{b+a}{2}\right)dx\,. \tag{2.36}$$

The remapping $[a, b] \to [0, +1]$ is another common occurrence in numerical methods. The choice of which polynomial to use is mostly dictated by the orthogonality range of the polynomial. For an open range such as $[0, \infty]$, the Laguerre polynomial is useful, or Hermite polynomial over $[-\infty, +\infty]$.

An integral over the range $[0, \infty)$ can be mapped into the domain $[0, 1)$ by the transformation $z = x/(1 + x)$, for example. The transformations given here are not unique but are simple examples and may, or may not, lead to numerical issues depending on the integrand. If the lower-bound of integration is a finite value, rather than zero, one may perform a linear transformation to zero first and then apply the above transformation. For integrals in the range $(-\infty, \infty)$, a mapping to the domain $(-1, 1)$ is possible with the transformation $x = z/(1 - z^2)$, for example. For many integrands, a simple approximation is possible by take infinity to be a large number.

A singularity at the endpoint of integration can sometimes be handled by making a change of variable. A singularity within the integration range can be handled by splitting the integral into two integrals and then making a change of variable to both integrals—usually not the same change of variable. A change of variable can be effective in handling improper integrals.

We can also integrate distributions with no closed functional form, and can integrate data points with no function. In all cases, I suggest plotting the integrand to help choose the method.

Table 2.1 show the uncertainties in the Newton-Cotes integration methods discussed above for a single-slice. It is not usual to estimate the uncertainty of a particular integration using these methods as an estimate of the derivative is required. Rather, the uncertainty formula are useful for assessing the gain in accuracy from using a particular method versus, usually, an easier method.

Table 2.1 Uncertainty of various numerical integration methods for d-dimensions and N evaluations, where ξ_i is a point within slice i.

Integration method	Single-slice uncertainty	Uncertainty
rectangle rule	$(1/2)(\Delta x)^2 f'(\xi_i)$	
trapezoidal rule	$(-1/12)(\Delta x)^3 f''(\xi_i)$	$N^{-2/d}$
mid-point rule	$(1/24)(\Delta x)^3 f''(\xi_i)$	
Simpson's rule	$(-1/90)(\Delta x)^5 f^{(4)}(\xi_{i+1})$	$N^{-4/d}$
Gauss' rule (of order m)		$N^{-(2m-1)/d}$
Monte Carlo		$N^{-1/2}$

Estimating the total integration uncertainty is more difficult. The error in a particular quadrature rule can be written as $\mathcal{O}((\Delta x)^p)$, where $\Delta x \ll 1$ is the grid spacing, which is related to the number of function calls, and p is the power of the leading error of the method. The numerical cost to perform a one-dimensional integration scales with the number of function evaluations $n \sim 1/\Delta x$, so we may write $\mathcal{O}(n^{-p})$. For a multidimensional integral, the total number of function evaluations is $N = n^d$, where d is the number of dimensions and assuming the same number of slices are used in each dimension of the integrations. The total error of a d-dimensional integral can thus be written as $\mathcal{O}(N^{-p/d})$. Table 2.1 also shows estimated total uncertainties for several methods. When d is large the uncertainty becomes significant. We will see in Sec. 5.3.6 that the Monte Carlo method of integration scales as $\mathcal{O}(N^{-1/2})$ regardless of the number of dimensions.

A useful Python library module for integration is `scipy.integrate`. Gaussian quadrature requires a function as input. The methods `quad`, `quadrature` and `fixed_quad` can be used, and I suggest `quad`. It supports $\pm\infty$ limits and one can also mask-off singularities. If you only have a set of data points, you can use interpolation or fit a function to the data to obtain the required integrand. The Newton-Cotes methods `traz`, `simps` and `romb` take an array of data as input.[3]

2.7.1 Example: Integration

Consider the normal probability integral

$$\frac{1}{\sqrt{2\pi}} \int_0^1 e^{-x^2/2} dx \,. \tag{2.37}$$

[3]Beginning with SciPy version 1.60 `simps`, `trapz` and `umtrapz` were renamed to `simpson`, `trapezoid` and `cumlative_trapezoid`.

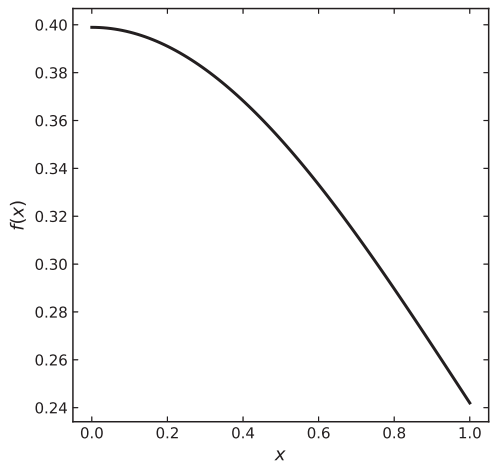
We will attempt to calculate the integral to five-digit accuracy using the rectangle rule and using `scipy.integrate.quad` which uses `qagse` as integrator based on globally adaptive interval subdivision in connection with extrapolation.

To validate the numerical result, we will compare against the analytic integral which uses the error (erf) function.

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-x^2} dx \quad \Rightarrow \quad \frac{1}{\sqrt{2\pi}} \int_0^1 e^{-x^2/2} dx = \frac{1}{2} \mathrm{erf}\left(\frac{1}{\sqrt{2}}\right) . \quad (2.38)$$

```
"""Finite range integration of Gaussian function"""

import numpy as np
import scipy.integrate as integrate
import scipy.special as special
import matplotlib.pyplot as plt
plt.style.use('./mystyle.mplstyle')

# Function to be integrated.
def f(x):
    return (1/np.sqrt(2*np.pi)) * np.exp(-x*x/2)

# Analytical integral.
theory = 0.5 * special.erf(1/np.sqrt(2))

# Lower and upper integral range.
lower = 0
upper = 1

# Calculate and print integral for different panel sizes.
print("dx    theory rectangle")
delta = [1e-2,1e-3,1e-4,1e-5,1e-6]
for dx in delta:
    # Rectangle rule (do not include last point in range).
    rec = np.sum(f(np.arange(lower,upper,dx))) * dx
    print("%.0e" % dx,"%.5f %.5f" % (theory,rec))
print()

# Calculate integral using library module.
val, err, infodict = integrate.quad(f,0,1,full_output=1)
print("quad result",val)
print("quad absolute error",np.format_float_scientific(err,3))
print("number of function evaluatons",infodict['neval'])
print("theory",theory)

# Plot integrand.
x = np.linspace(0,1,100)
fig, ax = plt.subplots()
plt.plot(x,f(x),lw=2,c='k')
plt.ylabel(r"$f(x)$")
plt.xlabel(r"$x$")
plt.savefig('2_6_integrate.pdf',bbox_inches='tight')
plt.show()
```

```
dx    theory rectangle
1e-02 0.34134 0.34213
1e-03 0.34134 0.34142
1e-04 0.34134 0.34135
1e-05 0.34134 0.34135
1e-06 0.34134 0.34134

quad result 0.341344746068543 absolute errror 3.789687964201238e-15
number of function evaluations 21
theory 0.3413447460685429
```



The plot of the integrand is smooth and the range of y is rather small so the integration should be easy. Note the plot comes out last as I separate the plotting code from the calculation code. Nevertheless, during coding development I make the plot first to help guide my coding. We see that we can obtain five-digit accuracy when using $\Delta x = 10^{-6}$. The `scipy.integrate.quad` library obtains at least 15-digit accuracy after only 21 function evaluations. We also see that the estimated absolute error is good to 15 decimal places. Since we are able to calculated the analytical integral in this case, we have validated that the numerical integral given by the library module is identical to the analytic result within the estimated uncertainty.

The lesson to be learn here is plot the integrand and do not ignore the error estimate provided by the integration routine.

2.8 FINITE DIFFERENCES

Physics usually deals with continua. On a computer, we sample the continua and can represent changes with finite differences. Finite differences can be

used in derivatives, interpolated polynomials, integration, regression and so forth.

Suppose we have a set of points $(x_i, y_i), i = 0, 1, \ldots, N-1$ with $x_i - x_{i-1}$ constant.
The backward difference ∇ is defined as

$$\begin{aligned} \nabla y_i &= y_i - y_{i-1}\,, \\ \nabla^2 y_i &= y_i - 2y_{i-1} + y_{i-2}\,. \end{aligned} \tag{2.39}$$

The forward difference Δ is defined as

$$\begin{aligned} \Delta y_i &= y_{i+1} - y_i\,, \\ \Delta^2 y_i &= y_{i+2} - 2y_{i+1} + y_i\,. \end{aligned} \tag{2.40}$$

The central difference δ is defined as

$$\begin{aligned} \delta y_i &= y_{i+1/2} - y_{i-1/2}\,, \\ \delta^2 y_i &= y_{i+1} - 2y_i + y_{i-1}\,. \end{aligned} \tag{2.41}$$

The second-order difference is defined by applying the first-order difference twice. These symbols for finite differences [4] are not universally used. The above definitions will be used extensively in Chapter 3 when solving differential equations.

Let's derive some approximations for derivatives using the Taylor series and the previously defined finite differences. The Taylor series can be written as

$$f(x) = f(a) + f'(x)(x-a) + \frac{f''(x)}{2!}(x-a)^2 + \frac{f'''(x)}{3!}(x-a)^3 + \ldots\,. \tag{2.42}$$

For finite differences, we restrict a to $a \pm n\Delta x$, where Δx is a constant difference in x. If $x_i = a$ and $x - a = \Delta x$ then $x = x_i + \Delta x$. The Taylor series can also be written as

$$f(x_i + \Delta x) = f(x_i) + f'(x_i)\Delta x + \frac{f''(x_i)}{2!}(\Delta x)^2 + \frac{f'''(x_i)}{3!}(\Delta x)^3 + \ldots\,. \tag{2.43}$$

Alternatively, if $x_i = a$ and $x - a = -\Delta x$ then $x = x_i - \Delta x$. The Taylor series can be written as

$$f(x_i - \Delta x) = f(x_i) - f'(x_i)\Delta x + \frac{f''(x_i)}{2!}(\Delta x)^2 - \frac{f'''(x_i)}{3!}(\Delta x)^3 + \ldots\,. \tag{2.44}$$

Subtracting Eq. (2.44) from Eq. (2.43) gives

$$f(x_i + \Delta x) - f(x_i - \Delta x) = 2f'(x_i)\Delta x + \frac{2}{3!}f''(x_i)(\Delta x)^3 + \ldots\,. \tag{2.45}$$

Neglecting terms $(\Delta x)^3$ and higher gives,

$$f'(x_i) = \frac{f(x_i + \Delta x) - f(x_i - \Delta x)}{2\Delta x} \quad \text{or} \quad y'_i = \frac{y_{i+1} - y_{i-1}}{2\Delta x}\,. \tag{2.46}$$

This is a central difference for the first derivative. It is a two-point difference good to second order in Δx.

Now consider adding Eq. (2.43) to Eq. (2.44):

$$f(x_i + \Delta x) + f(x_i - \Delta x) = 2f(x_i) + \frac{2}{2!} f''(x_i)(\Delta x)^2 + \frac{2}{4!} f^{IV}(x_i)(\Delta x)^4 + \ldots\,. \tag{2.47}$$

Neglecting terms $(\Delta x)^4$ and higher gives,

$$f''(x_i) = \frac{f(x_i + \Delta x) - 2f(x_i) + f(x_i - \Delta x)}{(\Delta x)^2} \quad \text{or} \quad y''_i = \frac{y_{i+1} - 2y_i - y_{i-1}}{(\Delta x)^2}\,. \tag{2.48}$$

This is a central difference for the second derivative. It is a three-point difference good to second order in Δx.

We will make heavy use of approximating derivatives with central differences. You can also develop approximations for derivatives using forward and backward difference by using Taylor series with $f(x_i + 2\Delta x), f(x_i - 2\Delta x)$ and so forth. Generally central differences are more accurate than forward or backward differences. Forward differences are useful at the beginning of an iteration when x_{i-1} is unknown, and backward differences at the end of an iteration when x_{i+1} is beyond the region of interest. Forward and backward differences are two-point differences accurate to only first order in Δx.

The above derivatives are two-point functions for first derivatives and three-point functions for second derivatives (requiring knowledge of x_{i+1}, x_i, x_{i-1}). You can also write four-point, five-point and so forth differences that could provide more accuracy. They can be derived using the Taylor series in a more complicated way than we have done above, or they can be derived using Richardson extrapolation [5].

In numerics, its a tradeoff between $\Delta x \to 0$ accuracy versus computing time. Computing time is often not important, but for finite differences the smallness of Δx is limited by the floating-point precision. Finite differences incur a truncation error by dropping higher-order terms. The precision error dominates at small Δx and the truncation error at high Δx, as shown in Fig. 2.5. At small Δx, we observe all methods suffering from about the same precision error which is due to the finite floating-point representation and not the method. At larger Δx, the methods are distinguishable and should scale with the truncation error in the Taylor series. We observe that the first central difference and the second forward difference have about the same accuracy due to the same truncation error.

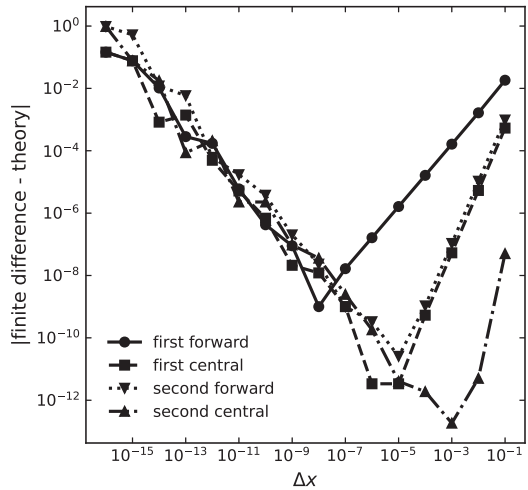


Figure 2.5 Uncertainty of some finite difference methods versus step size Δx for $e^{-x}\cos(x)$ evaluated at $x = 2$.

We could use the curves in Fig. 2.5 to optimize the choice of Δx by using the value when the error is a minimum. Although some dependence on the function and the point at which the function is evaluated might be expected, the minima are probably good representative choices for optimal Δx values.

2.8.1 Example: Finite differences

We will go through three simple examples of using finite differences. The last one deals with propagation of errors which we will revisit when discussing data analysis in Sec. 6.1.

For the first example, we will differentiate the sine function using a step size of $\Delta x = 0.0001$ over the range $[0,2\pi]$ at 1000 points. We calculate the derivative using both forward and central differences. The results are compared by plotting the difference between the analytical derivative of the sine function minus the finite difference results. Consider the following code.

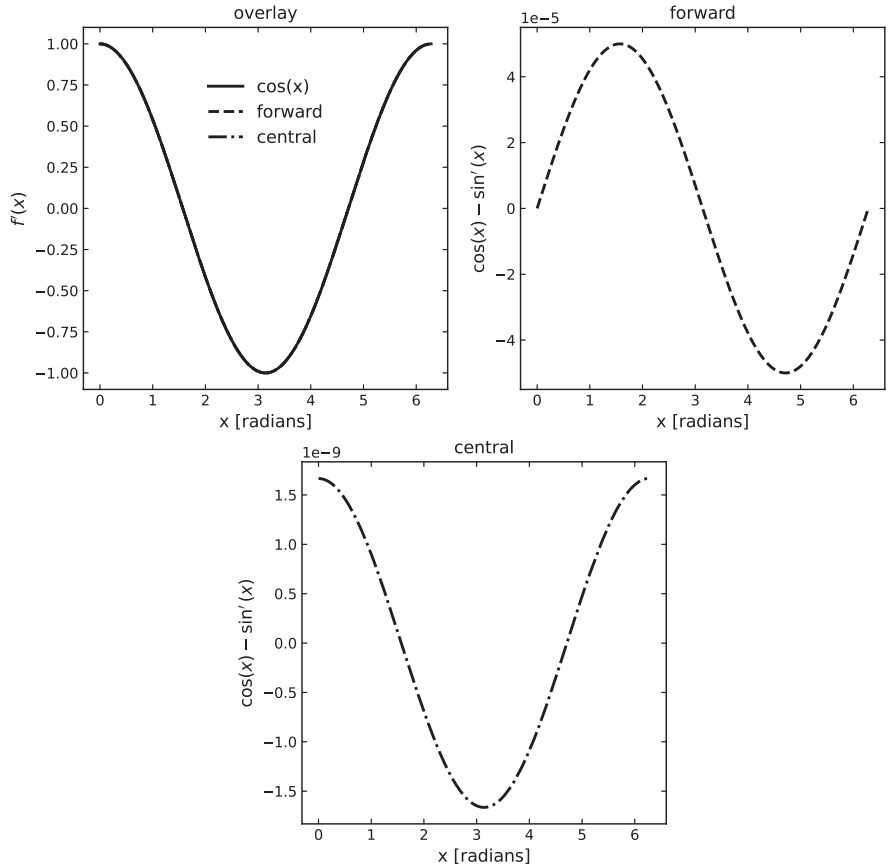
```
""" First derivative of sine function using finite
differences.
"""

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

plt.style.use('./mystyle.mplstyle')
```

```
# Theory.
def f(x):
    return np.sin(x)

# Calculate forward and central differences.
h = 0.0001
x = np.linspace(0,2*np.pi,1000)
print(len(x),"points")
y1 = (f(x+h) - f(x)) / h
y2 = (f(x+h) - f(x-h)) / (2*h)

# Plots
fig = plt.figure(figsize=(10,10))
gs = gridspec.GridSpec(2,4,wspace=0.5,hspace=0.2)

# Plot all 3 on same plot.
ax1 = plt.subplot(gs[0,:2],)
ax1.plot(x,np.cos(x),lw=2,ls='solid',  label="cos(x)")
ax1.plot(x,y1,       lw=2,ls='dashed', label="forward")
ax1.plot(x,y2,       lw=2,ls='dashdot',label="central")
ax1.set_title("overlay")
ax1.set_xlabel("x [radians]")
ax1.set_ylabel(r"$f^\prime(x)$")
ax1.legend(loc="upper center",bbox_to_anchor=(0.5,0.9),
           fontsize='large')

# Theory minus forward.
ax2 = plt.subplot(gs[0,2:])
ax2.plot(x,np.cos(x)-y1,lw=2,ls='dashed')
ax2.set_title("forward")
ax2.set_xlabel("x [radians]")
ax2.set_ylabel(r"$\cos(x) - \sin^\prime(x)$")

# Theory minus central.
ax3 = plt.subplot(gs[1,1:3])
ax3.plot(x,np.cos(x)-y2,lw=2,ls='dashdot')
ax3.set_title("central")
ax3.set_xlabel("x [radians]")
ax3.set_ylabel(r"$\cos(x) - \sin^\prime(x)$")

plt.savefig('2_7_finite_difference_ex1.pdf')
plt.show()
```

The difference between both finite differences and theory are tiny and cannot be seen on the overlay plot on the left. We observe that the central difference method (right plot) is about four orders of magnitude better than the forward difference method (middle plot).

I now discuss the absolute difference plots (middle and right) in the context of a simple error analysis. The step size is $\Delta x = 1 \times 10^{-4}$ and the machine precision is $\epsilon_m \approx 2.2 \times 10^{-16}$. The forward difference estimated error is [2, 6]

$$-\frac{\Delta x}{2} f''(x) + \frac{2}{\Delta x}\epsilon_m |f(x)| = 5.0 \times 10^{-5} \sin(x) + 4.4 \times 10^{-12} |\sin(x)| \,. \quad (2.49)$$



The central difference estimated error is

$$-\frac{(\Delta x)^2}{6}f'''(x)+\frac{1}{\Delta x}\epsilon_m|f(x)| = 1.7\times10^{-9}\cos(x)+2.2\times10^{-12}|\sin(x)|\,. \quad (2.50)$$

The errors are predicted to be dominated by the truncation error. The estimated errors are in surprisingly good agreement with the difference plots; both in magnitude and shape. We notice that the truncation errors follow a smooth functional form.

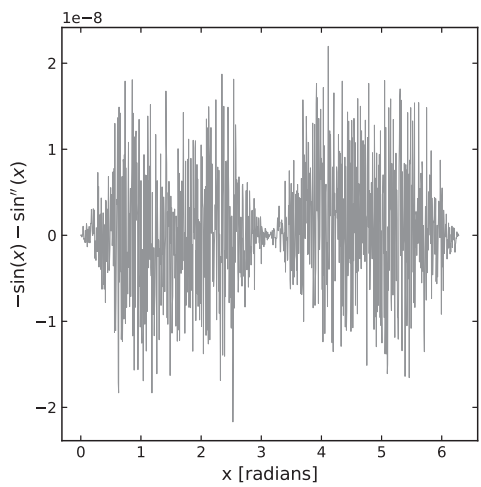
For the second example, we write a Python script to calculate the second derivative of the sine function using central differences with a step size of $\Delta x = 0.0001$ over the range $[0,2\pi]$ for 1000 points. We plot the difference between the analytical second derivative of a sine function and the central difference result. Consider the following code.

```
""" Second derivative of sine function using finite
differences.
"""

import numpy as np
import matplotlib.pyplot as plt
plt.style.use('./mystyle.mplstyle')

# Theory.
def f(x):
   return np.sin(x)

# Calculate second derivative using central difference.
x = np.linspace(0,2*np.pi,1000)
h = 0.0001
y = (f(x+h) - 2.0*f(x) + f(x-h)) / (h*h)

# Plot theory minus central difference.
fig, ax = plt.subplots()
plt.plot(x,-np.sin(x)-y,lw=0.5,c='grey')
plt.xlabel("x [radians]")
plt.ylabel(r"$-\sin(x) - \sin^{\prime\prime}(x)$")
plt.savefig('2_7_finite_difference_ex2.pdf')
plt.show()
```

We observe a small absolute difference again. However, this time the trends are less evident. Indeed, the difference seems to fluctuate about zero indicating we are probably seeing a precision issue compared to a truncation problem. The estimated uncertainty is [2]

$$-\frac{(\Delta x)^2}{12}f''''(x) + \frac{4}{(\Delta x)^2}\epsilon_m|f(x)| = -8.3\times10^{-10}\sin(x) + 8.8\times10^{-8}|\sin(x)|\,, \tag{2.51}$$

which is dominated by the precision uncertainty, as postulated.



For the third example, we plot a unit Gaussian (normal) probability density function with mean 0 and width 1 over the range $x = [-4, 4]$ for 1000 points. Let's say this is the perfect (theoretical) model for your data. Nevertheless, you know that the x resolution of your experimental data measurements has a standard deviation of 0.1. Let's use propagation of uncertainties (errors) and numerical differentiation to plot the one standard deviation envelope—up and down by one standard deviation relative to the nominal distribution.

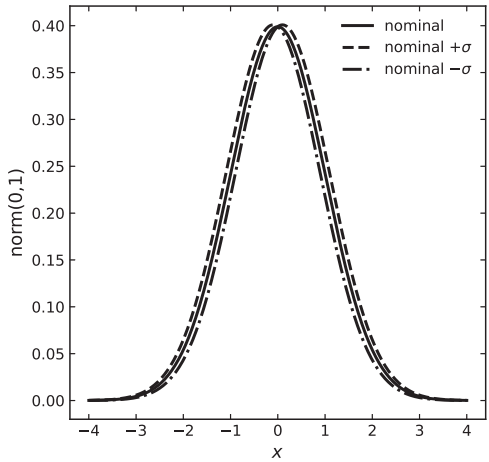
Given $y = f(x)$,

$$y \pm \delta y = f(x) \pm \left| \frac{df(x)}{dx} \right| \delta x \,, \tag{2.52}$$

where δx and δy are the uncertainties in the independent variable x and dependent variable y. Consider the following code

```
"""Resolution smearing of Gaussian function
using finite differences.
"""

import numpy as np
import scipy.stats as stat
import matplotlib.pyplot as plt
plt.style.use('./mystyle.mplstyle')

sigma = 0.1

# Normal pdf.
def f(x):
    return stat.norm.pdf(x,0,1)

# Calculate derivative.
N = 1000
x = np.linspace(-4,4,N)
h = 1/N
fp = abs(f(x+h) - f(x-h)) / (2*h)

# Plots.
fig, ax = plt.subplots()
plt.plot(x,f(x),          lw=2,ls='solid',
         label="nominal")
plt.plot(x,f(x)+fp*sigma,lw=2,ls='dashed',
         label=r"nominal $+\sigma$")
plt.plot(x,f(x)-fp*sigma,lw=2,ls='dashdot',
         label=r"nominal $-\sigma$")
plt.xlabel(r"$x$")
plt.ylabel("norm(0,1)")
plt.legend()
plt.savefig('2_7_finite_difference_ex3.pdf')
plt.show()
```

We observe the uncertainty vanishes at the peak and in the tails, where the derivative of the Gaussian approaches zero. In general, the uncertainty



follows the trend of the derivative as the δx is a constant. Hence, distributions are most smeared where they are steepest.

2.9 EXAMPLE: LAMBDIFICATION

This example evaluates a complicated potential formula three different ways. Our goal would be to plot or use this potential in subsequent code. Sometime a choice needs to be made about how much of the evaluation should be done computationally versus how much should be done analytically. Or, at what point should computation be used. I will examine three different approaches to evaluating the potential:

1. Manually expand and calculate the potential entirely by hand and then code the resulting formula. This approach is unique to the model and if the model is changed the calculation may need to be entirely repeated manually.

2. Expand and calculate the potential as separate functions. This is a middle ground for which one needs to calculate some derivatives manually.

3. Expand and calculate the potential using SymPy lambdification. This approach is general and it's easy to change the model and rerun the code.

Consider the gravitational-perturbation potential for a static spherically symmetric spacetime:

$$V_2(r) = \nu_2 \frac{G(r)}{H(r)} + \frac{[\partial_* H(r)]^2}{2H(r)^2} - \frac{\partial_*^2 \sqrt{H(r)}}{\sqrt{H(r)}}\,, \tag{2.53}$$

where G, F and H are model-dependent functions of r and some parameters; also $\nu_2 = \ell(\ell+1) - 2$. The derivatives are with respect to an r_* variable which is given by

$$\frac{dr_*}{dr} = \frac{1}{\sqrt{FG}}\,. \tag{2.54}$$

Consider a model with

$$F = G \equiv h = 1 - \left(\frac{r_\mathrm{h}}{r}\right)^{n+1} \qquad \text{and} \qquad H = r^2\,, \tag{2.55}$$

where r_h and n are parameters of the model. Let's now perform the manual calculations for each approach.

1. By taking the derivatives of Eq. (2.55) and plugging them into Eq. (2.53), and using Eq. (2.54), we obtain

$$V_2(r) = \frac{\nu_2 + 2}{r^2} - \frac{\nu_2 + n + 5}{r^{n+3}} + \frac{n+3}{r^{2n+4}}\,. \tag{2.56}$$

   This can then be used in further numerical calculations or plotted.

2. But what if h is complicated, but still $F = G \equiv h$ and $H = r^2$? For general h, differentiating and using Eq. (2.54) we obtain

$$V_2(r) = h\left[\frac{\ell(\ell+1)}{r^2} - \frac{h'}{r} + \frac{2(h-1)}{r^2}\right]\,, \tag{2.57}$$

   where the prime is the derivative with respect to r, which we will need to calculate manually. Once we have functions for h and h', we can use Eq. (2.57) in subsequent numerical calculations or plot it.

3. But what if F, G and H are still too complicated? We can perform the chain-rule in Eq. (2.53):

$$\partial_* H(r) = H' \frac{dr}{dr_*} \tag{2.58}$$

   and

$$\partial_*^2 H(r) = \left[H'' \frac{dr}{dr_*} + H' \frac{d}{dr}\left(1 \Big/ \frac{dr_*}{dr}\right)\right] \frac{dr}{dr_*}\,, \tag{2.59}$$

   and then let SymPy do the rest with `lambdify`.

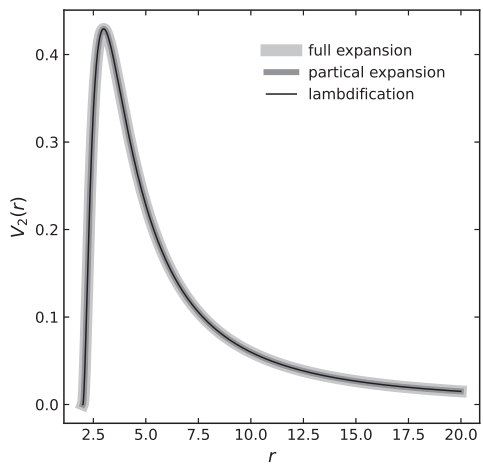
Consider the following code.

```
"""Lambdification"""

import numpy as np
from sympy import symbols, sqrt, diff, lambdify
import matplotlib.pyplot as plt
plt.style.use('./mystyle.mplstyle')
```

```
# Eq. (4). Potential fully precalculated.
def V2_1(r,l,rH,n):
    nu = l*(l+1) - 2
    return (nu+2) / r**2 - (nu+n+5)*rH**(n+1) / r**(n+3) \
           + (n+3)*rH**(2*n+2) / r**(2*n+4)

# Eq. (5). Potential requires h and its derivative.
def h(r,rH,n):
    # Function h.
    return 1 - (rH/r)**(n+1)

def hp(r,rH,n):
    # Derivative of function h: h'.
    return (n+1) * (rH/r)**(n+1) / r

def V2_2(r,l,rH,n):
    # Potential.
    return h(r,rH,n) * ( l*(l+1) / r**2 - hp(r,rH,n) / r \
           + 2*(h(r,rH,n)-1) / r**2 )

# Eq. (1). Potential requires F, G, H, no derivatives.
def H_S(r):
    # Function H.
    return r**2

def F_S(r,rH,n):
    # Function F.
    return 1 - (rH/r)**(n+1)

def G_S(r,rH,n):
    # Function G.
    return 1 - (rH/r)**(n+1)

def V2_3(r,l,rH,n):
    # Potential
    r_ = r #Save numerical value.
    r  = symbols("r")
    F = F_S(r,rH,n)
    G = G_S(r,rH,n)
    H = H_S(r)
    drsdr = 1 / sqrt(F*G)
    nu = l*(l+1) - 2
    v = nu * G / H + (diff(H,r)/drsdr)**2 / (2*H**2) \
        - (diff(sqrt(H),r,2) / drsdr**2 + diff(sqrt(H),r) \
        * diff(1/drsdr,r)/ drsdr) / sqrt(H)
    f = lambdify([r],v) #v is symbolic potential.
    return f(r_)

# Pick some parameters for the problem.
l  = 2
rH = 2
n  = 3
r  = np.arange(rH,rH*10,0.01)

# Superimpose 3 method on a plot.
fit, ax = plt.subplots()
```

```
plt.plot(r,V2_1(r,l,rH,n),lw=8,c='silver',
         label="full expansion")
plt.plot(r,V2_2(r,l,rH,n),lw=4,c='gray',
         label="partical expansion")
plt.plot(r,V2_3(r,l,rH,n),lw=1,
         label="lambdification")
plt.xlabel(r"$r$")
plt.ylabel(r"$V_2(r)$")
plt.legend(loc="upper right",bbox_to_anchor=(0.95,0.95))
plt.savefig('2_8_lambdification.pdf')
plt.show()
```

The plot shows the potential (overlaid) using the three different methods. As they are identical, we have some confidence in our calculations and code. If the model changes, just replacing the functions F, G and H is all that is required in the lambdification approach. Congratulations, you have just calculated the gravitational-perturbation potential for a higher-dimensional Schwarzschild black hole that can be used in calculations of quasinormal modes or transmission coefficients.



2.10 PROBLEMS

1. **Function evaluation, be careful**

   Consider the following function that we might want to evaluate or use in an application:

   $$f(x) = \frac{x^2}{\sqrt{1+x^2}-1}\,.$$

   We would like to be able to evaluate the function at any x.

(a) Code this function as written (see the cell below). Plot the function over the range $x = [-0.1, 0.1]$ and show that you get pathological problems approximately when $|x| < 2 \times 10^{-7}$.

```
%import numpy as np
%def f(x):
%    return x**2 / (np.sqrt(1+x**2) - 1.0)
```

(b) Ignoring numerical issues, is the analytic function above finite at $x = 0$? Show or explain your reasoning?

(c) Do you understand what is happening when you try evaluating the function numerically with $|x| < 2 \times 10^{-7}$?

(d) Rewrite the original analytic function above to be numerically safe in the vicinity of $x = 0$ and good over a range larger than $x = [-1, 1]$.

For pedagogical purposes do not use your result in question (d) for the following questions.

(e) Rewrite the original function above as an infinite series to be numerically safe in the vicinity of $x = 0$. Do not use a Taylor series expansion.

(f) Plot the safe series function and compare it to the original way in question (a) of plotting the function.

(g) Say how you would modify your code to plot a range larger than $x = [-1, 1]$ in question (f) without using your result of question (d).

You may use the NumPy package and Matplotlib library in your code.

2. **Plot single branch**

Consider the following equation:

$$r_{\mathrm{h}} = 2MP\left(\frac{3}{2}, \frac{r_{\mathrm{h}}^2}{4\theta}\right),$$

where M is the independent variable and r_{h} is the dependent variable; θ is a parameter. $P(s, x) = \gamma(s, x)/\Gamma(s)$ is the regularized lower incomplete gamma function (γ is the lower incomplete gamma function and Γ is the gamma function).

Note the library module function `scipy.special.gammainc(s,x)` returns $P(s, x)$ above, i.e. the library function is the regularized version. So we needn't worry about γ and Γ.

(a) Scan through some r_h values for a given M to show that r_h is double valued, i.e. there are two values of r_h for each M in some range of M.

(b) Write a general function, or class, that will return just the upper branch, i.e. the larger r_h values when given the independent variable M. Try to make your function, or class, independent of the function above, i.e. you pass to your code the independent variable, as well as, the function above.

(c) For $\theta = 1$, find the turning point of the function, $[M_\mathrm{min}, r_\mathrm{h}(M_\mathrm{min})]$.

(d) Call your function, or class, with an arbitrary array of M values starting at M_min to return the upper branch of r_h and plot it.

You may use modules `special` and `optimize` from the SciPy library. Congratulations, you have just plotted the event horizon of a noncommutative black hole.

3. **Conic section**

The general equation of a conic section can be written as

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0\,,$$

where A, B, C, D, E, F are constants.

Determine the equation of the conic passing through the points $(x, y) = \left(0, \frac{1}{2}\right)$, $\left(\frac{1}{2}, \frac{4}{9}\right)$, $\left(-2, \frac{57}{2}\right)$, $\left(1, \frac{3}{22}\right)$ and $\left(-\frac{1}{2}, 0\right)$.

(a) First outline the procedure you will use.

(b) Second write the code to determine the coefficients.

(c) Validate your resulting equation.

You may use NumPy only in this problem.

Hint: the coefficient A may be considered to have a magnitude of unity, since the entire equation may be divided by the coefficient of x^2 to establish this value.

4. **Coupled oscillators**

Consider three coupled oscillators of different masses m_i attached by two springs of different spring constants k_j as shown in the diagram.

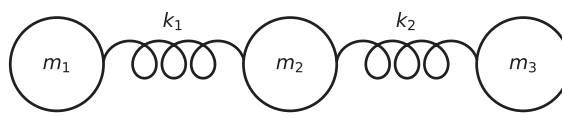


Figure 2.6 Three coupled oscillators.

(a) Write down the three coupled differential equations of motion.

(b) Assume a solution of the form $x_n = A_n e^{i\omega t}, n = 1, 2, 3$. Use this solution to express the set of differential equations as a set of algebraic equations, and write the algebraic equations as a matrix equation.

(c) Solve the matrix equation numerically as an eigenvalue problem. First use three different masses and two different spring constants to show that you get oscillations.

(d) Print out the eigenvectors of the fundamental modes of vibration and the fundamental frequencies of vibration.

(e) Are you able to perform any validation of your results numerically? If so, do so.

(f) Now solve the following three problems:

i. Use $k_1 = k_2 = 1$, $m_1 = 1, m_2 = 2, m_3 = 3$ as a case of three different masses.

ii. Use $k_1 = k_2 = 1$, $m_1 = m_3 = 1, m_2 = 2$ as a case of two equal masses.

iii. Use $k_1 = k_2 = 1$, $m_1 = m_2 = m_3 = 1$ as a case of three equal masses.

(g) Can you validate the three special-case results analytically? Do cases ii. and iii.. You are welcome to try i.

(h) Is there any other validation you can do?

(i) Is it possible by using different spring constants and masses to get three real nonzero frequencies.

You may use the NumPy linear algebra functions `numpy.linalg` in your code. The SymPy library may only be used for validation. You might recognize this problem as a model for a linear molecule with three atoms, such as CO_2.

5. **Integration of data**

Assume the data below describes the acceleration of an object initially at rest at the origin; the data is acceleration a versus time t.

```
import numpy as np
t = np.array([0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0])
a = np.array([0.00,1.15,2.03,2.96,3.97,5.12,
              6.10,7.18,8.02,8.99,10.05])
```

(a) Find the position of the object versus time and plot it.

(b) Also plot the acceleration and velocity on the same plot as the position.

6. **Stefan-Boltzmann constant**

The intensity of radiation emitted from a black-body surface is given by Planck's law

$$I(\nu, T) = \frac{2h\nu^3}{c^2} \frac{1}{e^{h\nu/(kT)} - 1} ,$$

where $I(\nu, T)$ is the amount of power per unit surface area per unit solid angle per unit frequency emitted at a frequency ν by a black body of temperature T. The constants h, c and k we don't care about besides that the Stefan-Boltzmann constant σ is given by the combination

$$\sigma = 40.803 \frac{k^4}{c^2 h^3} .$$

The power radiated by a surface of area A through a solid angle $d\Omega$ in the frequency range between ν and $\nu + d\nu$ is $I(\nu, T) A d\nu d\Omega$.

The Stefan-Boltzmann law is given by the power emitted per unit area of the emitting body

$$\frac{P}{A} = \int_0^\infty I(\nu, T) d\nu \int_0^{2\pi} d\phi \int_0^{\pi/2} \cos\theta \sin\theta d\theta .$$

If the power per unit area is measured to be 460 W/m^2 at a temperature of 300 K, determine the Stefan-Boltzmann constant (in SI units) by numerical integration of the above intensity. You may perform the angular integrals analytically. You may perform the frequency integration using a library module. Do not use numerical values for h, c and k, or natural units.

(a) Show the equation you will solve for the Stefan-Boltzmann constant. Including the numerical coefficient, the two measured values above, and the numerical integral.

(b) Estimate the uncertainty of your numerical integration and the affect on your result.

(c) At what accuracy do you agree with the accepted value of the Stefan-Boltzmann constant?

(d) Can you explain your level of agreement?

7. **Bessel function derivatives**

Sometimes it's not so easy to analytically compute the derivatives of special functions. For the following, use central differences over the range $x = [0.1, 1.0]$ in steps of 0.1.

(a) Calculate the first and second derivatives of the zeroth-order Bessel function of the first kind $J_0(x)$ at these points.

(b) In this case, we can check the accuracy of our results using the first-order Bessel function which is related to the derivative of the zeroth-order Bessel function. Do so.

(c) Can you also compare the second derivative of the zero-order Bessel function to other Bessel functions? If so, do so.

(d) How many decimals of accuracy can you get with central differences?

It is good to present your results in tabular form; no plots needed.

8. **Finite differences with data**

Suppose we have sampled an unknown function $y = f(x)$ at fixed x intervals, and have a set of data $(x_i, y_i), i = 1, 2, \ldots, 10$ as given below.

```
import numpy as np
y = np.array([13.3,21.9,35.2,52.2,72.8,98.4,126.8,160.0,197.1,237.9])
```

(a) Compute the first and second derivatives of $f(x)$ at the points $i = 2, \ldots, 9$ using central differences.

(b) Display your results on a plot so we can check them.

(c) Examine your results and guess the functions for the first and second derivatives.

(d) Can you use these results to qualitatively deduce the original function?

9. **Chain rule**

Consider the following potential $V(r)$ which is a function of the dependent variable r, and three parameters M, s and ℓ:

$$V(r) = \left(1 - \frac{2M}{r}\right)\left(\frac{\ell(\ell+1)}{r^2} + \frac{(1-s^2)2M}{r^3}\right).$$

The dependent variable r which appears in the potential is a function of the independent variable r_*, such that $V(r(r_*))$ and

$$\frac{dr}{dr_*} = \left(1 - \frac{2M}{r}\right).$$

What is needed in this problem is the derivative of $V(r)$ many times with respect to r_* evaluated at a single point $(r_*)_0$ where $V(r)$ is a maximum.

Take $M = 1$ and $s = \ell = 1$.

(a) Numerically differentiate the potential four times with respect to r_* and evaluate each derivative at $(r_*)_0$.

(b) Present your results as a table of five numbers (include 0th, 1st, 2nd, 3rd, 4th derivatives).

(c) For validation, repeat the above using the SymPy library.

When finding the maximum, find it numerically in question (a) and symbolically in question (c). You may use libraries modules to help find the maximum.

No plot is needed. Show any intermediate work as markdown text. Congratulation, you have perform much of the require calculation for determining gravitational quasinormal modes of a Schwarzschild black hole using the WKB (Wentzel-Kramers-Brillouin) approximation.

10. **Deflection of light in a gravitational field**

In this problem, we will calculate the deflection of light by the Sun. This was one of the early tests of general relativity. The agreement of general relativity with measurement helped establish its validity over Newtonian gravity.

Using Schwarzschild spherical symmetry for the Sun, and the conservation of energy and angular moment, the orbital equation for a photon—null geodesic—is

$$\frac{d\phi}{dr} = \pm\frac{1}{r^2}\left[\frac{1}{b^2} - \frac{1}{r^2}\left(1 - \frac{2GM}{c^2 r}\right)\right]^{-1/2} .$$

The sign determines the direction of the orbit around the scattering centre located at the origin. The radial coordinate r is centred at the sun, the motion is in a plane and ϕ is the angle in that plane. Here we are interested in the scattering orbits so b can be interpreted as the impact parameter. G, M, c are constants.

To get the total angle swept out by the photon, we integrate from a photon coming in from $r = -\infty$, scattering, and then going out to $r = \infty$. This can be written as twice the integral from the turning radius r_1 to ∞:

$$\Delta\phi = 2\int_{r_1}^{\infty} \frac{dr}{r^2}\left[\frac{1}{b^2} - \frac{1}{r^2}\left(1 - \frac{2GM}{c^2 r}\right)\right]^{-1/2} . \tag{2.60}$$

The turning point is the radius where

$$\frac{1}{b^2} - \frac{1}{r_1^2}\left(1 - \frac{2GM}{c^2 r_1}\right) = 0 . \tag{2.61}$$

We will now solve Eq. (2.60) with Eq. (2.61). For numerical purposes, we make the change of variable to the dimensionless variable $u = b/r$. Substitution into Eq. (2.60) and Eq. (2.61) gives:

$$\Delta\phi = 2\int_0^{u_1} du\left[1 - u^2\left(1 - \frac{2GM}{c^2 b}u\right)\right]^{-1/2}$$

and

$$1 - u_1^2\left(1 - \frac{2GM}{c^2 b}u_1\right) = 0\,.$$

The only parameter of the problem is $GM/(c^2 b)$. For the bending of light by the Sun, the smallest value of b is approximately the solar radius; thus $GM/(c^2 b) \approx 10^{-6}$. One can then assume $GM/(c^2 b)$ is small and obtain $\Delta\phi \approx \pi + 4GM/(c^2 b)$. From this one get the deflection angle

$$\delta\phi = \Delta\phi - \pi \approx 4\frac{GM}{c^2 b}\,.$$

This is twice the Newtonian limit for the deflection angle for a zero-mass particle.

I find it nontrivial to obtain this approximate deflection angle so this problem calculates it exactly. This is another case where an analytic solution is not necessarily better than a numerical solution; as the analytic solution is approximate and the numerical solution will be exact.

(a) Evaluate the integral exactly to obtain the deflection angle.

(b) Compare your numerical solution with the analytic approximation. At what decimal accuracy do they disagree?

(c) Estimate your numerical accuracy and compare it to the difference between the analytic approximation and the numerical result obtain in the previous part of the question.

11. **Precession of the perihelion of Mercury**

In this problem, we will calculate the precession of the perihelion of Mercury. This was one of the early tests of general relativity. The agreement of general relativity with measurement helped establish its validity over Newtonian gravity.

In Newtonian gravity, planets follow perfect ellipses (Kepler orbits) around the Sun—neglecting perturbations from other planets. The motion is periodic in the equatorial plane ($\theta = \pi/2$). The orbits close if the angle ϕ sweeps out exactly 2π in the passage between two successive inner or two successive outer radial turning points. In general relativity, the planet's major orbital axis rotates slowly and the orbits do not close. The angle ϕ between successive radial turning points is larger than 2π.

The Precession per orbit of the inner turning point (perihelion) is parameterized by $\delta\phi_{\rm prec} = \Delta\phi - 2\pi$.

Using Schwarzschild spherical symmetry for the Sun and the conservation of energy E and angular moment ℓ, the orbital equation for a massive particle—time-like geodesic—is

$$\frac{d\phi}{dr} = \pm\frac{\ell}{r^2}\left[2(E - V_{\text{eff}}(r))\right]^{-1/2} ,$$

where the effective potential is

$$V_{\text{eff}}(r) = -\frac{GM}{c^2 r} + \frac{\ell^2}{2r^2} - \frac{GM\ell^2}{c^2 r^3} .$$

The sign determines the direction of the orbit around the Sun located a the origin. The radial coordinate r is centred at the Sun, the motion is in a plane and ϕ is the angle in that plane. The orbit is given by r as a function of ϕ, or visa versa.

The angular momentum can be obtained approximately by assuming Newtonian orbits:

$$\ell^2 \approx \frac{GM}{c^2} a(1 - e^2) ,$$

where e is the eccentricity and a is the semi-major axis of the orbit.

(a) Calculate the angular momentum ℓ for Mercury's orbit.

The turning points r_1 and r_2 are the perihelion and aphelion, respectively. The energy is determined by calculating the potential at either turning point.

(b) Calculate the energy E for Mercury's orbit.

The change in ϕ per orbit $\Delta\phi$ is obtained by integrating over one orbit, which is twice one-half of the orbit from r_1 to r_2:

$$\Delta\phi = 2\ell \int_{r_1}^{r_2} \frac{dr}{r^2} \left[2(E - V_{\text{eff(r)}})\right]^{-1/2} .$$

Notice that the expression in the square brackets vanishes at the turning points.

The first two terms in $V_{\text{eff}}(r)$ are the Newtonian contribution and the last term is a relativistic correction. Typically one expands in $1/c^2$ to obtain the approximation

$$\delta\phi_{\text{pre}} = \Delta\phi - 2\pi = 6\pi\left(\frac{GM}{c\ell}\right)^2 \approx \frac{6\pi GM}{ac^2(1 - e^2)} .$$

(c) Plug in the values for the Sun and Mercury's orbit to show you get the famous 42 arcsec per century.

As a warm up, let's consider the Newtonian approximation by dropping the third term in $V_{\text{eff}}(r)$.

(d) Make a change of variable, $u = 1/r$, to obtain the following integral

$$\Delta\phi = 2\int_{u_2}^{u_1} \frac{du}{[(u_1 - u)(u - u_2)]^{1/2}} ,$$

where $u_1 = 1/r_1$ and $u_2 = 1/r_2$.

The integral can be evaluated analytically or numerically.

(e) Evaluate the integral numerically. You may validate your numerical result by doing the calculation analytically or symbolically, or both.

(f) Show your result is independent of the choices of r_1 and r_2 as long as $r_1 < r_2$.

This Newtonian result shows there is no precession and that this is true for all elliptical orbits.

(g) Now perform the full relativistic numerical calculation. Direct integration should give 5.02×10^{-7} radian per orbit.

You might consider a change of variable $u = r_2/r$ to make the problem more numerically well-behaved. Care must be taken near the turning points where the denominator of the integral can become undefined due to numerical resolution effects.

This calculation is of intrinsic interest for two reasons: 1) if the numerical result is correct, is the previous approximation accurate enough, and 2) can one obtain enough numerical accuracy to obtain a result close to the correct result?

CHAPTER 3

Differential equations

Since physical laws are commonly expressed as differential equations, their numerical solution is arguably the most important topic in computational physics. As is common practice, I will divide the discussion into ordinary differential equations (ODE) and partial differential equations (PDE). Unlike analytical methods, the discussion will be largely applicable to both linear and nonlinear differential equations, and homogeneous and non-homogeneous differential equations. Solving a differential equation is often called integrating it, which is not to be confused with integrating an analytic function or data.

3.1 ORDINARY DIFFERENTIAL EQUATIONS

ODEs are typically divided into initial-value problems, boundary-value problems and eigenvalue problems. The key method employed in numerical computing is to reduce the problem of an Nth-order differential equation to N coupled first-order differential equations. We start with

$$\frac{d^N y}{dx^N} = f\left(x, y, \frac{dy}{dx}, \ldots, \frac{d^{N-1}y}{dx^{N-1}}\right), \tag{3.1}$$

where x is the independent variable and y the dependent variable. If the nth derivative is written as

$$\frac{d^n y}{dx^n} = y^{(n)} \quad \text{for} \quad n = 0, 1, 2, \ldots, N-1, \tag{3.2}$$

the total system of differential equations can be written as a vector of first-order differential equations whose components are

$$\frac{dy^{(n)}}{dx} = f_n(x, y^{(0)}, y^{(1)} \ldots, y^{(n)}) \quad \text{for} \quad n = 0, 1, \ldots, N-1, \tag{3.3}$$

where $y^{(0)} = y$ and the right-hand side is assumed to be known.

The first-order differential equation can be written as

$$y' = f(x, y); \quad \frac{dy^{(0)}(x)}{dx} = f_0(x, y^{(0)}) \quad \text{or} \quad \frac{dy(x)}{dx} = f(x, y). \tag{3.4}$$

DOI: 10.1201/9781003660873-3

The second-order differential equation as

$$y'' = f(x, y, y'); \quad \frac{dy^{(1)}(x)}{dx} = f_1(x, y^{(0)}, y^{(1)}) \text{ and } \frac{dy^{(0)}(x)}{dx} = f_0(x, y^{(0)}) \,. \tag{3.5}$$

Hopefully, the pattern for third-order and higher is apparent.

Students usually first encounter ODEs in physics in the context of classical mechanics. Consider classical mechanics in one dimension as a simple example:

$$\frac{d^2x}{dt^2} = f\left(t, x, \frac{dx}{dt}\right) \tag{3.6}$$

can be written as

$$\frac{dx}{dt} = v(t) \quad \text{and} \quad \frac{dv}{dt} = f(t, x, v) \,, \tag{3.7}$$

where I have assume v is a function of t only, and not also of x.

Now consider a nonlinear example:

$$xy' + y = 0 \quad \text{gives} \quad y' = -\frac{y}{x} \equiv f(x, y) \,. \tag{3.8}$$

Finally, consider a more general example:

$$\frac{d^2y}{dx^2} + q(x, y)\frac{dy}{dx} + r(x, y) = 0 \tag{3.9}$$

gives

$$\frac{dy}{dx} = f(x) \quad \text{and} \quad \frac{df(x)}{dx} = -q(x, y)f(x) - r(x, y) \,. \tag{3.10}$$

We thus only need to solve 1st-order coupled differential equations. Coupled equations are easy to handle computationally. For simplicity in notation in what follows, let $y_i = f(x_i)$ and $y_{i+1} = f(x_i + \Delta x)$, and so forth.

3.1.1 Euler method

Of all the methods for solving ODEs given initial values, the Euler method is the simplest and most intuitive, but not particularly accurate. This can be expect as the method is over 250 years old and was developed for paper calculations.

To see how the Euler method works, write the Taylor series as

$$y_{i+1} = y_i + y_i'\Delta x + \frac{y_i''}{2!}(\Delta x)^2 + \ldots \,. \tag{3.11}$$

Truncated to 1st order $y_{i+1} = y_i + y_i'\Delta x$, where $y_i' = f(x_i, y_i)$ is usually a known function. For initial-value problems, $y_0' = f(x_0, y_0)$. We then iterate from the initial values:

$$\begin{aligned} y_1 &= y_0 + y_0'\Delta x \,, \quad x_1 = x_0 + \Delta x \quad \Rightarrow \quad y_1' = f(x_1, y_1) \\ y_2 &= y_1 + y_1'\Delta x \,, \quad x_2 = x_1 + \Delta x \quad \Rightarrow \quad y_2' = f(x_2, y_2) \,, \text{ etc.} \end{aligned} \tag{3.12}$$

The above algorithm is called self-starting as we did not have to do anything special in the first iteration. The Euler method is a straight-line estimate for the function at (x_i, y_i) (see Fig. 3.1). The cumulative errors build up rapidly.

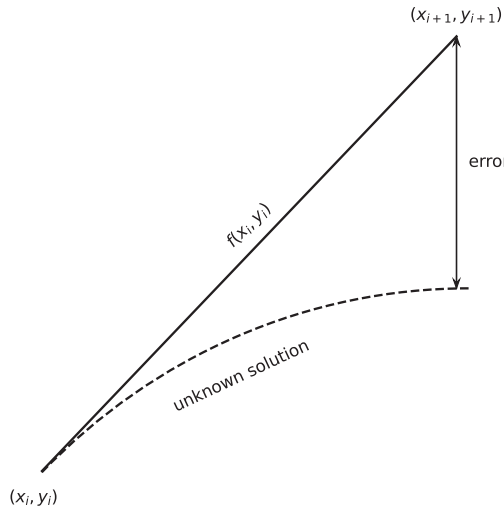


Figure 3.1 Forward Euler method of an unknown solution starting from (x_i, y_i) and predicting at (x_{i+1}, y_{i+1}).

After thinking a bit, you can probably come up with an alternative to the Euler method. Realising that the Euler method is essentially using a forward difference, you could start from a backward difference to come up with the so-called backward-Euler method: $y_{i+1} = y_i + y'_{i+1}\Delta x$. More precisely, we should call the Euler method in Eq. (3.12) the forward-Euler method. We can immediately see a problem with the backward-Euler method in that future positions $(i + 1)$ appear on both sides of the equation. The method is not self-starting.

The astute reader thinking generally, says, "rather than calculate f at position i or $(i + 1)$, I can calculate it at the average position, or I can take the average of f at the two positions". The four possibilities can be written as

$$\begin{aligned}
y_{i+1} &= y_i + f(x_i, y_i)\Delta x && \text{forward Euler,} \\
y_{i+1} &= y_i + f(x_{i+1}, y_{i+1})\Delta x && \text{backward Euler,} \\
y_{i+1} &= y_i + f\left(x_i + \frac{\Delta x}{2}, \frac{y_i + y_{i+1}}{2}\right)\Delta x && \text{midpoint method,} \\
y_{i+1} &= y_i + \left[\frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1})}{2}\right]\Delta x && \text{trapezoid method.}
\end{aligned} \tag{3.13}$$

All these methods, except for the forward-Euler method, are called implicit methods. That is, future positions $(i+1)$ appear on both sides of the equation; they are not self-starting. As an algorithm, the advancing scheme for implicit methods use the value of the derivative at the end of the step rather than the beginning of the step as in explicit methods. To make the midpoint or trapezoid method self-starting, or implicit, we can first calculate y_{i+1} on the right-hand side by using the forward-Euler method. The methods will then become self-starting and explicit as follows:

$$\begin{aligned} y_{i+1} &= y_i + f\left(x_i + \frac{\Delta x}{2}, y_i + f(x_i, y_i)\frac{\Delta x}{2}\right)\Delta x && \text{midpoint method,} \\ y_{i+1} &= y_i + \left[\frac{f(x_i, y_i) + f(x_{i+1}, y_i + f(x_i, y_i)\Delta x)}{2}\right]\Delta x && \text{trapezoid method.} \end{aligned} \tag{3.14}$$

Alternatively, calculating the forward-Euler method first can be considered a prediction. Then using it to recalculate y_{i+1}, is the correction. These explicit methods are sometimes referred to as predictor-corrector methods. Furthermore, the explicit midpoint method is sometimes called the modified-Euler method, and the explicit trapezoid method called the Heun method. The modified-Euler method is demonstrated in Example 3.1.10. The algorithm for the Heun method is as follows:

$$\begin{aligned} y_0' &= f(x_0, y_0) \\ P(y_1) &= y_0 + y_0'\Delta x \quad \text{predict} \quad y_1 \\ P(y_1') &= f(x_1, P(y_1)) \quad \text{predict} \quad y_1' \quad (\text{and} \quad x_1 = x_0 + \Delta x) \\ C(y_1) &= y_0 + \frac{y_0' + P(y_1')}{2}\Delta x\,, \quad \text{corrected, etc.} \end{aligned} \tag{3.15}$$

In the above implicit methods, the error on the prediction and correction are not the same. The predictor accuracy can be made the same accuracy as the corrector by using central differences. But the first step in the method will not be self-starting. We can apply the self-starting predictor-corrector method for the first step, and then after that, the predictor and corrector are of the same accuracy.

I could go on about more methods, such as, the Euler-Cromer method, leap-frog methods, Verlet methods and so forth, and I will do so soon. Some of these are particularly good for certain types of ODE encountered in physics. Using these methods with an adaptive independent-variable step size (see Sec. 3.1.4) also allows making some other ODE solutions tractable. An accurate and popular method that you might code yourself is the fourth order Runge-Kutta method. The method is also commonly offered as a library module. We turn to Runge-Kutta methods now.

3.1.2 Fourth-order Runge-Kutta method

A Rung-Kutta method is one which employs a recurrence formula of the form

$$y_{i+1} = y_i + \sum_{j=1}^{N} a_j k_j \tag{3.16}$$

to calculate successive values of the dependent variable y of the differential equations. The a_j are numerical coefficients and

$$k_j = \Delta x f\left(x_i + p_{j-1}\Delta x, y_i + \sum_{l=1}^{j-1} q_{j-1} k_l\right) \quad \text{for} \quad j = 1, 2, \ldots, N\,, \tag{3.17}$$

where

$$p_0 = 0 \quad \text{and} \quad \sum_{l=1}^{j-1} q_{j-1} k_i = 0 \quad \text{for} \quad j = 1\,. \tag{3.18}$$

The values of k_j represent estimates of the slope of the desire function at various points in the interval x_i to x_{i+1}. These slopes are weighted and averaged by the coefficients a_j.

The explicit midpoint and trapezoidal methods mentioned previously are second-order Runge-Kutta methods. Third-order Runge-Kutta methods exist but fourth-order methods are the most common.

Given $y_i' = f(x_i, y_i)$, the fourth-order Runge-Kutta formula is

$$y_{i+1} = y_i + \frac{\Delta x}{6}[k_1 + 2k_2 + 2k_3 + k_4] + \mathcal{O}((\Delta x)^5) \tag{3.19}$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{\Delta x}{2}, y_i + \frac{\Delta x}{2} k_1\right) \\ k_3 &= f\left(x_i + \frac{\Delta x}{2}, y_i + \frac{\Delta x}{2} k_2\right) \\ k_4 &= f\left(x_i + \Delta x, y_i + \Delta x k_3\right)\,, \end{aligned} \tag{3.20}$$

and k_1, k_2, k_3, k_4 represent estimates of the slope at various points in the interval x_i to x_{i+1}. The four slopes are weighted and averaged.

Runge-Kutta methods are self-starting and do not require explicit derivative approximations, but do require multiple function evaluations per step. The Runge-Kutta methods are single-step methods since they require only knowledge of y_i to determine y_{i+1}. In spite of the small increase in computational time for the function evaluations, they have superior accuracy and stability compared to the previous methods we have discussed. Higher than fourth-order Runge-Kutta methods are possible but become complicated. Fourth-order Runge-Kutta is a good compromise.

3.1.2.1 Two first-order differential equations

First consider the example of two simultaneous first-order differential equations:

$$\frac{dy}{dx} = f\left(x, y(x), u(x)\right) \quad \text{and} \quad \frac{du}{dx} = F\left(x, y(x), u(x)\right) . \tag{3.21}$$

The Runge-Kutta formulae are

$$\begin{aligned} y_{i+1} &= y_i + \frac{\Delta x}{6}[k_1 + 2k_2 + 2k_3 + k_4] \\ k_1 &= f(x_i, y_i, u_i) \\ k_2 &= f\left(x_i + \frac{\Delta x}{2}, y_i + \frac{\Delta x}{2}k_1, u_i + \frac{\Delta x}{2}l_1\right) \\ k_3 &= f\left(x_i + \frac{\Delta x}{2}, y_i + \frac{\Delta x}{2}k_2, u_i + \frac{\Delta x}{2}l_2\right) \\ k_4 &= f\left(x_i + \Delta x, y_i + \Delta x k_3, u_i + \Delta x l_3\right) \end{aligned} \tag{3.22}$$

and

$$\begin{aligned} u_{i+1} &= u_i + \frac{\Delta x}{6}[l_1 + 2l_2 + 2l_3 + l_4] \\ l_1 &= F(x_i, y_i, u_i) \\ l_2 &= F\left(x_i + \frac{\Delta x}{2}, y_i + \frac{\Delta x}{2}k_1, u_i + \frac{\Delta x}{2}l_1\right) \\ l_3 &= F\left(x_i + \frac{\Delta x}{2}, y_i + \frac{\Delta x}{2}k_2, u_i + \frac{\Delta x}{2}l_2\right) \\ l_4 &= F\left(x_i + \Delta x, y_i + \Delta x k_3, u_i + \Delta x l_3\right) . \end{aligned} \tag{3.23}$$

We can see that the order of the evaluation of the formulae is not unique. I have found for most ODEs, the order does not matter. However, I have also encountered ill-conditioned problems where different orders of formulae evaluation lead to different results.

3.1.2.2 One second-order differential equation

Now consider the example of Newton's equation of motion:

$$\frac{d^2x}{dt^2} = f\left(t, x, \frac{dx}{dt}\right) . \tag{3.24}$$

The two first-order equations are

$$\begin{aligned} &\frac{dx}{dt} = v\,; \quad \frac{dv}{dt} = f\left(t, x(t), v(t)\right) \quad \text{or} \\ &\frac{dx}{dt} = \frac{p}{m}\,; \quad \frac{dp}{dt} = F\left(t, x(t), v(t)\right) . \end{aligned} \tag{3.25}$$

In the momentum representation, the Runge-Kutta formulae are

$$x_{i+1} = x_i + \frac{\Delta t}{6}[k_1 + 2k_2 + 2k_3 + k_4] \tag{3.26}$$

$$p_{i+1} = p_i + \frac{\Delta t}{6}[l_1 + 2l_2 + 2l_3 + l_4]\,, \tag{3.27}$$

where

$$\begin{aligned}
mk_1 &= p_i\,; & l_1 &= F\left(t_i, x_i, v_i\right)\\
mk_2 &= p_i + \frac{\Delta t}{2} l_1\,; & l_2 &= F\left(t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2}k_1, v_i + \frac{\Delta t}{2}l_1\right)\\
mk_3 &= p_i + \frac{\Delta t}{2} l_2\,; & l_3 &= F\left(t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2}k_2, v_i + \frac{\Delta t}{2}l_2\right)\\
mk_4 &= p_i + \Delta t l_3\,; & l_4 &= F\left(t_i + \Delta t, x_i + \Delta t k_3, v_i + \Delta t l_3\right)
\end{aligned}$$

In mechanics, often $F\left(t, x(t), v(t)\right) = F\left(x(t), v(t)\right)$, and for the simple harmonic oscillator $F\left(t, x(t), v(t)\right) = -kx$, and the method formulae reduce to

$$\begin{aligned}
l_1 &= -kx_i\\
l_2 &= -k\left(x_i + \frac{\Delta t}{2}k_1\right)\\
l_3 &= -k\left(x_i + \frac{\Delta t}{2}k_2\right)\\
l_4 &= -k\left(x_i + \Delta t k_3\right)\,.
\end{aligned}$$

All the ODE methods we have encountered so far suffer to some degree from round-off errors and truncation errors. There is a trade-off between doing more steps with smaller Δx using a lower-order method as opposed to doing fewer steps with a larger Δx using a higher-order method. The truncation error is a local error; the error made in a single step and is related to the order of the method. The Euler method has a local truncation error of $\mathcal{O}((\Delta x)^2)$. The implicit methods are one order, $\mathcal{O}((\Delta x)^3)$, better but require two function evaluations. The Runge-Kutta local truncation error is $\mathcal{O}((\Delta x)^5)$.

Propagation of errors is also present which includes the error from all the proceeding steps. In a typical problem, we start from some point $x = a$ and want to reach some endpoint at $x = b$. We need to take N steps of Δx. If we reduce Δx we need to take more steps. If the local error is $\mathcal{O}((\Delta x)^n)$, we can estimate the global error as [7]

$$N\mathcal{O}((\Delta x)^n) = \frac{b-a}{\Delta x}\mathcal{O}((\Delta x)^n) = (b-a)\mathcal{O}((\Delta x)^{n-1})\,. \tag{3.28}$$

This is a crude estimate as the local errors could accumulate (constructive errors) or cancel (destructive errors) to varying degrees. The key point is that

the global error is about one order of magnitude lower than the truncation local error. Nevertheless, both the local and global errors approach 0 as $\Delta x \to 0$. So the methods can be considered convergent.

Another important consideration is instability (see Sec. 3.2.6) which occurs when the error grows rapidly. The above global error estimate will not hold if the calculation is not stable. Implicit schemes tend to be more stability. For example, the forward- and backward-Euler methods have the same accuracy, but the backward-Euler method is unconditionally stable, in contrast to the forward-Euler method which may not be stable.

Notice that the step size of the Runge-Kutta method may be changed at any step in the calculation. Using an adaptive step size with the Runge-Kutta method is a common improvement. Other improved algorithms include multi-step methods, Richardson-extrapolation methods, Bulirsch-Stoer methods and so forth.

It is necessary to distinguish between the instability of a method and ill-conditioned problems. If a method cannot solve a well conditioned problem, we say it unstable; a method that can solve it is stable. For an ill-conditioned problem, no method can solve it. The ill-conditioning of a problem is often called the stiffness of the ODE. A stiff ODE causes most methods to be unstable. Stiff equations often contain two different scales. By employing a step size small enough to handle the small scale it takes a prohibitive amount of calculation to handle the large scale. Problem 3.3.6 examines a simple model that leads to an ill-conditioned ODE.

3.1.3 Leap-frog and Verlet algorithms

Current-day research problems often need a faster scheme for solving ODEs than the Runge-Kutta method. The step size can be dictated by considerations other than accuracy, such as the steps for a large number of particles in higher dimensions. This leads us to examine leap-frog and Verlet-type methods.

3.1.3.1 Leap-frog method

In classical physics, we can often take advantage that Newton's second law is second-order and write the equation of motion as two first-order ODEs. We start with the equations of motion

$$\frac{d\mathbf{v}}{dt} = \mathbf{a}(\mathbf{r}(t)) \quad \text{and} \quad \frac{d\mathbf{r}}{dt} = \mathbf{v}(t)\,. \tag{3.29}$$

Using a central difference for the acceleration gives

$$\mathbf{a}(\mathbf{r}_n) = \frac{\mathbf{v}_{n+1} - \mathbf{v}_{n-1}}{2h} + \mathrm{O}(h^2)\,, \tag{3.30}$$

We calculate the velocity using a central difference but advanced in time:

$$\mathbf{v}_{n+1} = \frac{\mathbf{r}_{n+2} - \mathbf{r}_n}{2h} + \mathrm{O}(h^2)\,. \tag{3.31}$$

Rearranging the terms to collect future values on the left-hand side gives:

$$\begin{aligned} \mathbf{v}_{n+1} &= \mathbf{v}_{n-1} + 2h\mathbf{a}(\mathbf{r}_n) + \mathrm{O}(h^3) \\ \mathbf{r}_{n+2} &= \mathbf{r}_n + 2h\mathbf{v}_{n+1} + \mathrm{O}(h^3)\,, \end{aligned} \tag{3.32}$$

which is the leap-frog method. The solution is advanced in steps of $2h$ with the positions evaluated at even values of n, while the velocities are evaluated at odd values of n. This interlacing is necessary because the acceleration, which is a function of position, needs to be evaluated at a time that is centred between the new and old velocities. Sometimes the leap-frog method is formulated in steps of h with an offset of $h/2$, but this method is equivalent to what I have presented.

The leap-frog method is not self-starting. One can take a backward-Euler step to start the algorithm:

$$\mathbf{v}_{-1} = \mathbf{v}_0 - h\mathbf{a}_0\,. \tag{3.33}$$

Leap-frog schemes are simple and, in physics, generally conserve energy which can be an essential requirements for solutions to physics problems.

3.1.3.2 Verlet method

Let's start with the particular case:

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}(t) \quad \text{and} \quad \frac{d^2\mathbf{r}}{dt^2} = \mathbf{a}(\mathbf{r})\,. \tag{3.34}$$

Using central differences for both derivatives gives

$$\begin{aligned} \mathbf{v}_n &= \frac{\mathbf{r}_{n+1} - \mathbf{r}_{n-1}}{2h} + \mathcal{O}(h^2) \\ \mathbf{a}_n &= \frac{\mathbf{r}_{n+1} - 2\mathbf{r}_n + \mathbf{r}_{n-1}}{h^2} + \mathcal{O}(h^2)\,. \end{aligned} \tag{3.35}$$

Rearranging the terms in the last equation to collect future values on the left-hand side gives:

$$\mathbf{r}_{n+1} = 2\mathbf{r}_n - \mathbf{r}_{n-1} + h^2\mathbf{a}_n + \mathcal{O}(h^4)\,. \tag{3.36}$$

These equations are the Verlet method. Again the position and velocity are not calculate at the same time for each iteration.

Because the algorithm needs $\mathbf{r}$ from the previous two steps, it is not self-starting. To get the method started we can use forward differences for the velocity:

$$\mathbf{r}_{-1} = \mathbf{r}_0 - h\mathbf{v}_0 + \frac{h^2}{2}\mathbf{a}(\mathbf{r}_0)\,. \tag{3.37}$$

The position equation has a good truncation error. Part of the efficiency of the Verlet algorithm is the ability to solve for the position of each particle without requiring a separate solution for the particle's velocity. If the force is only a function of position, and we care only about the trajectory of the particle—not its velocity—we can completely skip the velocity calculation.

3.1.3.3 Velocity-Verlet method

Another version of the Verlet algorithm is the velocity-Verlet algorithm. It is recommended because of its stability. It uses forward differences to advance both the average position and average velocity simultaneously:

$$\begin{aligned}\mathbf{r}_{n+1} &= \mathbf{r}_n + h\left(\frac{\mathbf{v}_{n+1}+\mathbf{v}_n}{2}\right) = \mathbf{r}_n + h\mathbf{v}_n + \frac{h^2}{2}\mathbf{a}_n \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + h\left(\frac{\mathbf{a}_{n+1}+\mathbf{a}_n}{2}\right) = \mathbf{v}_n + \frac{h}{2}(\mathbf{a}_{n+1}+\mathbf{a}_n)\,.\end{aligned} \tag{3.38}$$

Although it appears to be lower order, the use of updated positions when calculating velocities, and subsequent use of these velocities, give both Verlet algorithms similar precision. A comparison of these last three scheme will follow in an example. The above three algorithms could be improved by varying the time steps as discussed next.

3.1.4 Adaptive step size

Thus far, we have arbitrarily picked a fixed step size for advancing the independent variable when solving differential equations. It is beneficial to allow the program based on the equation being solve to determine a step size and change it as the behaviour of the function changes; in other words, to adapt the step size to the problem at hand. Adaptive step size is a method for changing the step size dynamically to use a small step size where the curvature is high and a large step size where the curvature is less. I discuss the step-doubling technique and leave fancier methods for the library modules. We will take each step twice; once as a full step $2h$ and independently as two half steps h. The extra calculation will add to the computational overhead, but the investment will usually be worth it.

Consider a step size of $2h$ advancing from x to $x+2h$, where the exact solution is $y(x+2h)$ and the approximate solution is y_1. For a method with accuracy to order $(n-1)$ but error of order n, we write

$$y(x+2h) = y_1 + c(2h)^n + \mathcal{O}(h^{n+1})\,, \tag{3.39}$$

where c is constant over small step sizes. Similarly, for two steps of size h:

$$y(x+2h) = y_2 + 2ch^n + \mathcal{O}(h^{n+1})\,, \tag{3.40}$$

where y_2 is the approximate solution. The difference between the two numerical estimates is an indicator of the current truncation error, $\Delta = y_2 - y_1$. We can eliminate c from the two equations to improve the numerical estimate of the true solution:

$$f(x+2h) = y_2 + \frac{\Delta}{2^{n-1}-1} + \mathcal{O}(h^{n+1})\,. \tag{3.41}$$

This result has one power of h more accuracy than the original method.

Let ϵ as the desire absolute tolerance that we are trying to achieve. The target step size h' to achieve this accuracy can be obtained from

$$\frac{\Delta}{\epsilon} = \left(\frac{h}{h'}\right)^n , \tag{3.42}$$

and hence

$$h' = \alpha h \left|\frac{\epsilon}{\Delta}\right|^{1/n} , \tag{3.43}$$

where $\alpha \approx 0.9$ has been introduced as a safety factor. We might also want to limit the change in step size from one iteration to the next to avoid potential instabilities. Also, limits should be set on the minimum step size to avoid roundoff errors and maximum step size to avoid pathological cases.

Step doubling has been superseded by a more efficient step-size adjustment algorithms based on embedded Runge-Kutta formulae. These algorithms form the default for the library module that we discuss next.

3.1.5 Python libraries for solving ODEs

After you have gained some knowledge and experience with the previous methods for solving ODEs, you will want to solve them using Python library modules. The Python libraries for solving ODEs typically want the set of coupled first-order differential equations, Eq. (3.3), as an array given by

$$\frac{d}{dx}\begin{bmatrix} y^{(0)} \\ y^{(1)} \\ \vdots \\ y^{(N-1)} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \end{bmatrix} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ f(x, y^{(0)}, \ldots, y^{(N-1)}) \end{bmatrix} . \tag{3.44}$$

Advanced methods are available in the `scipy.integrate` Python library modules `solve.odeint` and `solve_ivp`. For most cases, these two libraries are functionally equivalent; `odeint` is older than `solve_ivp`. The `odeint` library uses `lsoda` from the FORTRAN library `odepack` which I believe uses the Adams/BDF method. For `solve_ivp`, an explicit Rung-Kutta method of order 5(4) is used by default. Other methods are available by setting the optional `method` argument. If you are writing your own code, I suggest `solve_ivp`. However, there are still many instances of `odeint` in existence and if you need to understand or modify that code it is useful to be familiar with `odeint`.

The following compares `odeint` on the left with `solve_ivp` on the right.

`odeint(func,y0,t,...)`	`solve_ivp(fun,t_span,y0,...)`
`func = dydt(y,t)`	`fun = dydt(t,y)`
`y0 =` array of initial conditions	`t_span = (t0,tf)` time endpoints
`t =` array of time points	`y0 =` array of initial conditions
`y = odeint(...)`	`t, y = solve_ivp(...)`

The order of the first two arguments in function `dydt` are swapped between the two methods. If the optional argument `tfirst = true` in `odeint`, the first two arguments will be swapped to have the same order as `solve_ivp`. Function `dydx` argument's must include `t` but do not modify it. If you are plotting the results, I suggest using the optional argument `t_eval` to specify the granularity of values returned, similar to `t` in `odeint`.

Consider this simple example to help explain the two methods.

$$\begin{aligned} \theta'(t) &= \omega(t)\,, \\ \theta''(t) &= \omega'(t) = -b\omega(t) - c\sin(\theta(t))\,. \end{aligned} \tag{3.45}$$

In array form,

$$\frac{d}{dt}\begin{bmatrix} \theta \\ \omega \end{bmatrix} = \begin{bmatrix} \omega \\ -b\omega - c\sin\theta \end{bmatrix}. \tag{3.46}$$

For `odeint`

```
def f(t,y,b,c):
    theta, omega = y
    dydt = [omega,-b*omega-c*np.cos(theta)]
    return dydt
```

The returned information is `y`, where `y[:0] = theta` and `y[:1] = omega` with `y[0] = y0`.

For `solve_ivp`

```
def f(y,t,b,c):
    theta, omega = y
    dydt = [omega,-b*omega-c*np.cos(theta)]
    return dydt
```

If optional argument `dense_output = True`, the returned information is a bunch object `sol` with `sol.t` the time series and `sol.y` the solution in the same format as returned by `odeint`. Also `sol.status` (int), `sol.message` (string), and `sol.success` (bool) can be useful and I suggest taking advantage of them.

3.1.6 Boundary-value problems

Thus far we have discussed only initial-value problems in which values of the dependent variable are specified for one value of the independent variable, usually t. In some problems one wishes to solve a system of ODEs subject to values of the dependent variable which are specified at different values of the independent variable x. Such a problem is called a boundary-value problem.

To solve boundary-value problems one possibility is to use finite-difference equations and solve the corresponding system of algebraic equations using matrix methods. This method will be demonstrated in Example 3.1.10.3.

A popular method for solving boundary-value problems reduces to the problem of solving a sequence of initial-value problems. This method of solving boundary-value problems is called the shooting method. Typically only one initial condition is known and the other must be guessed. The initial-value problem is solved to determine the function value at the other boundary. The method is iterated until the solution at the other boundary is close enough, within a tolerance, to the second boundary value. The result at the second boundary can be considered a function of the guess. The iteration method can be optimised by finding the root of this function, using for example, bisection (see Sec. 2.2). The shooting method is illustrated in Sec. 3.1.10.4 and you will encounter it as an eigenvalue problem in Problem 3.3.5.

3.1.7 Pseudo-spectral methods

There are several other methods besides finite differences for solving ODEs: spectral methods, asymptotic iteration methods, Frobenius methods and so forth.

Spectral methods work only for linear differential equations since the methods relies on the principle of superposition. The boundary conditions require the function to be zero at the boundaries—homogeneous boundary conditions—but we will overcome this limitation.

Some aspects of solving differential equations are not commonly discussed in textbooks. Singularities can occur in the differential equation. Zeros are usually also referred to as singularities in this context. These singularities often occur at the boundaries. To solve these cases numerically, the solution can be postulated as a product of a function that obeys the boundary conditions and removes the singularities and an unknown function. The product solution is plugged into the differential equation resulting in a new differential equation in the unknown function which is regular—no singularities or zeros—everywhere. In addition, the independent variable sometimes needs to be transformed into a finite domain to allow a numerical solution. This results in a new differential equation in the new independent variable with a finite range.

Spectral methods for solving differential equations are, in general, powerful and efficient provide the function is smooth. The method approximates the solution we are trying to find, rather than the equation to be solved. The regular function $u(x)$ in the differential equation is decomposed into a truncated sum of basis functions:

$$u(x) \approx u_N(x) = \sum_{n=0}^{N-1} a_n \phi_n(x)\,. \tag{3.47}$$

Different choices for the basis functions $\phi_n(x)$ and methods for computing the coefficients a_n correspond to different types of spectral methods.

The basis functions must form a complete set, each basis function by itself must obey the boundary conditions and be an eigenfunction of the differential

operator in the problem. Only the first condition will be essential. Here we will consider nonperiodic solutions that can be mapped to finite boundary conditions. In this case, the basis functions of choice are Chebyshev polynomials (see Sec. 2.5).

To determine the basis function coefficients, the differential equation and basis functions are evaluated at a set of collection points (grid). Evaluation on a grid of collection points give the pseudo-spectral method its name. We'll use the set of collection points called the Chebyshev grid for which the Gauss-Lobatto version is given by

$$x_i = \cos\left(\frac{i\pi}{N-1}\right) \quad \text{for} \quad i = 1, 2, \ldots, (N-2)\,. \tag{3.48}$$

There are typically two ways to impose the boundary conditions. One approach is to reduce the number of collection points on the residual of the differential equation, and use rows of the pseudo-spectral matrix to explicitly impose the constraints. The other is to modify the problem so that the boundary conditions of the modified problem are homogeneous, and then alter the basis set so the basis functions individually satisfy these conditions. The later method will be described here and is superior for eigenvalue problems [8] .

Consider the equation $Lu = f$, where L is a linear differential operator and f is a function causing the differential equation to be inhomogeneous. Assume this differential equation has inhomogeneous boundary conditions. We replace it with an equation with homogeneous boundary conditions:

$$\begin{aligned} u(x) &= v(x) + B(x)\,, \\ g(x) &= f(x) - LB(x) \end{aligned} \tag{3.49}$$

so that the modified problem is $Lv = g$, where $v(x)$ satisfies homogeneous boundary conditions and $g(x)$ is called a forcing function. The function $B(x)$ is called the shift function and is arbitrary, besides that it must satisfy the inhomogeneous boundary conditions.

Taking $B(x)$ to be linear is a simple choice. For example, if $u(-1) = \alpha$ and $u(+1) = \beta$, by definition, $v(-1) = v(+1) = 0$, and

$$B(x) = \frac{\alpha}{2}(1-x) + \frac{\beta}{2}(1+x)\,. \tag{3.50}$$

This procedure is called homogenization of the boundary conditions.

I now discuss basis recombination. This is the procedure of choosing simple linear combinations of the original basis functions so that these combinations, the new basis functions, individually satisfy the homogeneous boundary conditions.

Since for Chebyshev polynomials

$$T_{2n}(\pm 1) = 1 \quad \text{and} \quad T_{2n+1}(\pm 1) = \pm 1 \quad \text{for} \quad n = 1, 2, \ldots\,, \tag{3.51}$$

a good choice of basis functions are

$$\phi_{2n}(x) = T_{2n} - 1 \quad \text{and} \quad \phi_{2n+1} = T_{2n+1} - x \quad \text{for} \quad n = 1, 2, \ldots . \tag{3.52}$$

Since $T_0(x) = 1$ and $T_1(x) = x$, these two functions are no longer independent basis functions and will not be included in the set of $(N-2)$ basis functions.

The collection points are the $(N-2)$ interior points of an N-point Gauss-Lobatto grid. We do not include the end points since the boundary conditions are simply imposed there.

We write

$$v(x) \approx \sum_{n=2}^{N-1} b_n \phi_n(x) \,. \tag{3.53}$$

Using the collection points, we can now write the problem in matrix notation

$$\mathbf{Hb} = \mathbf{G} \,, \tag{3.54}$$

where $G_i = g(x_i)$ and

$$H_{ij} = P(x_i)\frac{d^2\phi_{j+1}(x_i)}{dx^2} + Q(x_i)\frac{d\phi_{j+1}(x_i)}{dx} + R(x_i)\phi_{j+1}(x_i) \,, \tag{3.55}$$

for $i, j = 1, \ldots, (N-2)$.

Sometimes the $\mathbf{H}$ matrix can be ill-conditioned. But more often, this is an indication that you have a bug in your code. If you are convinced the matrix is calculated correctly and it is ill-conditioned, you can try using QR decomposition to solve the matrix equation. First calculate the QR decomposition of the transpose: $\mathbf{H}^T = \mathbf{QR}$. Since $\mathbf{Q}$ is an orthogonal matrix, a little algebra gives

$$\mathbf{b} = \mathbf{Q}(\mathbf{R}^T)^{-1}\mathbf{G} \,. \tag{3.56}$$

After solving the matrix equation to compute the coefficients b_n of $\phi_n(x_i)$, we solve Eq. (3.49) for $u(x_i)$. If we need to evaluate $u(x)$ at points other then the collection points, we evaluate the basis function at the new x values and calculate $u(x)$ using the previously determined b_n coefficients.

Sometimes calculating the new basis functions at different values of x can be much more complicated than calculating the original basis functions at x. In this case, we can use the original basis functions to write

$$v(x) = \sum_{n=0}^{N-1} a_n T_n(x) \,, \tag{3.57}$$

where

$$a_n = b_n \text{ for } n \leq 2 \,, \; a_0 = -\sum_{n=1}^{(2n)\leq(N-1)} b_{2n} \text{ and } a_1 = -\sum_{n=1}^{(2n+1)\leq(N-1)} b_{2n+1} \,. \tag{3.58}$$

The saving for the case of Chebyshev polynomials is questionable. Problem 3.3.7 has you write a general class or function for solving ODEs using the pseudo-spectral method.

3.1.8 Asymptotic iteration method

Consider the homogeneous linear second-order differential equation for the function $u(x)$:

$$u'' = \lambda_0(x)u' + s_0(x)u\,, \tag{3.59}$$

where primes are differentiation with respect to x, and $\lambda_0(x)$ and $s_0(x)$ are defined in some interval, not necessarily bounded, and have sufficiently many continuous derivatives. In order to find a general solution to this equation, we rely on the form of the right-hand side. If we differentiate with respect to x, we find that

$$u''' = \lambda_1(x)u' + s_1(x)u\,, \tag{3.60}$$

where

$$\lambda_1 = \lambda_0' + s_0 + (\lambda_0)^2 \quad \text{and} \quad s_1 = s_0' + s_0\lambda_0\,. \tag{3.61}$$

Taking the second derivative, we get

$$u'''' = \lambda_2(x)u' + s_2(x)u\,, \tag{3.62}$$

where

$$\lambda_2 = \lambda_1' + s_1 + \lambda_0\lambda_1 \quad \text{and} \quad s_1 = s_0' + s_0\lambda_0\,. \tag{3.63}$$

Iteratively, for the $(n+1)$th derivatives, $n = 1, 2, \ldots$, we have

$$u^{(n+1)} = \lambda_{n-1}(x)u' + s_{n-1}(x)u\,. \tag{3.64}$$

We thus see that differentiating Eq.(3.64) n times with respect to x leaves a symmetric form for the right-hand side:

$$u^{(n+2)} = \lambda_n(x)u' + s_n(x)u\,, \tag{3.65}$$

where

$$\begin{aligned} \lambda_n(x) &= \lambda_{n-1}'(x) + s_{n-1}(x) + \lambda_0(x)\lambda_{n-1}(x) \quad \text{and} \\ s_n(x) &= s_{n-1}'(x) + s_0(x)\lambda_{n-1}(x)\,. \end{aligned} \tag{3.66}$$

For sufficiently large n, the asymptotic aspect of the method is [9, 10]

$$\frac{s_n(x)}{\lambda_n(x)} = \frac{s_{n-1}(x)}{\lambda_{n-1}(x)} \equiv \beta(x)\,, \tag{3.67}$$

where the eigenvalues are obtained from the quantization condition:

$$\delta_n = s_n\lambda_{n-1} - s_{n-1}\lambda_n = 0\,, \tag{3.68}$$

which is equivalent to imposing a termination in the number of iterations.

To calculate the eigenfunctions, we take the ratio of the $(n+2)$th derivative to the $(n+1)$th derivative to obtain

$$\frac{d}{dx}\ln\left(u^{(n+1)}\right) = \frac{u^{(n+2)}}{u^{(n+1)}} = \frac{\lambda_n(u' + (s_n/\lambda_n)u)}{\lambda_{n-1}(u' + (s_{n-1}/\lambda_{n-1})u)}\,. \tag{3.69}$$

From our asymptotic limit, this reduces to

$$\frac{d}{dx}\ln\left(u^{(n+1)}\right) = \frac{\lambda_n}{\lambda_{n-1}}\,, \tag{3.70}$$

which yields

$$u^{(n+1)}(x) = C_1 \exp\left(\int^x \frac{\lambda_n(x')}{\lambda_{n-1}(x')}dx'\right) = C_1\lambda_{n-1}\exp\left(\int^x(\beta+\lambda_0)dx'\right)\,, \tag{3.71}$$

where C_1 is the integration constant and the right-hand side of the definition of $\beta(x)$ has been used. Substituting this into Eq. (3.59), we obtain the first-order differential equation

$$u' + \beta u = C_1 \exp\left(\int^x(\beta+\lambda_0)dx'\right)\,, \tag{3.72}$$

which leads to the general solution:

$$u(x) = \exp\left[-\int^x \beta(x')dx'\right]\left[C_2 + C_1\int^x \exp\left(\int^{x'}[\lambda_0(x'') + 2\beta(x'')]dx''\right)\right]\,. \tag{3.73}$$

The integration constants, C_1 and C_2, can be determined by an appropriate choice of normalisation. Note that for an exact solution $C_1 = 0$.

An unappealing feature of the recurrence relations is that in each iteration one must take the derivative of the s and λ terms of the previous iteration. This can slow the numerical implementation of the method down considerably and can also lead to problems with numerical precision. To circumvent these issues an improved version of the method which bypasses the need to take derivatives at each step is used. We expand λ_n and s_n in a Taylor series around an arbitrary value ξ:

$$\begin{aligned}\lambda_n(\xi) &= \sum_{i=0}^{\infty} c_n^j (x-\xi)^i \quad \text{and} \\ s_n(\xi) &= \sum_{i=0}^{\infty} d_n^j (x-\xi)^i\,,\end{aligned} \tag{3.74}$$

where the c_n^i and d_n^i are the ith Taylor series coefficients of $\lambda_n(\xi)$ and $s_n(\xi)$, respectively. Substituting these expressions into Eq. (3.66) leads to a set of recurrence relations for the coefficients:

$$\begin{aligned}c_n^i &= (i+1)c_{n-1}^{i+1} + d_{n-1}^i + \sum_{k=0}^{i} c_0^k c_{n-1}^{i-k} \quad \text{and} \\ d_n^i &= (i+1)d_{n-1}^{i+1} + \sum_{k=0}^{i} d_0^k d_{n-1}^{i-k}\,.\end{aligned} \tag{3.75}$$

In terms of these coefficients, the quantization condition can be expressed as

$$d_n^0 c_{n-1}^0 - d_{n-1}^0 c_n^0 = 0\,, \tag{3.76}$$

and thus we have reduced the method into a set of recurrence relations which no longer require derivative operators. We begin at $n = 0$ and calculating the $(n + 1)$ coefficients sequentially until the desired number of recursions is reached.

3.1.8.1 Examples: Asymptotic iteration method

The asymptotic iteration method (AIM) generally has three steps, with simple problems skipping some of these steps. First, an asymptotic solution at $\pm\infty$ needs to be substituted into the ODE to obtain another well-behaved ODE over a finite range. This is general for all methods and not specific to AIM. Sometimes an additional change of variable is needed. Second, the recurrence relations need to be solved. Usually an analytic form in the large number-of-iterations limit is required. This can be done on paper or using SymPy, depending on your abilities. Finally, the quantization condition needs to be solved for the eigenvalues. This is usually performed using root finding and often is done numerically. Now consider three examples.

Differential equation with constant coefficients First let's solve the general homogeneous linear second-order differential equation with constant coefficients. The recurrence relations Eq (3.66) simplify to

$$\lambda_n = s_{n-1} + \lambda_0 \lambda_{n-1} \quad \text{and} \quad s_n = s_0 \lambda_{n-1}\,. \tag{3.77}$$

The ratio can be written as

$$\frac{s_n}{\lambda_n} = \frac{s_0}{s_{n-1}/\lambda_{n-1} + \lambda_0} \tag{3.78}$$

and thus Eq. (3.67) becomes

$$\beta = \frac{s_n}{\lambda_n} = \frac{s_0}{s_n/\lambda_n + \lambda_0}\,. \tag{3.79}$$

Solving for β, which is a constant, we obtain a quadratic equation:

$$\beta^2 + \lambda_0 \beta - s_0 = 0\,. \tag{3.80}$$

The solutions is

$$\beta = \frac{-\lambda_0 \pm \sqrt{\lambda_0^2 + 4s_0}}{2}\,. \tag{3.81}$$

The solution to the differential equation becomes

$$\begin{aligned} u(x) &= \exp\left(-\beta x\right)\left[C_2 + C_1 \int^x \exp\left[(\lambda_0 + 2\beta)x\right)\right] \\ &= C_2 \exp(-\beta x) + C_1' \exp[(\lambda_0 + \beta)x]\,, \end{aligned} \tag{3.82}$$

where C_1' is a new constant. All that remains to do is determine β for the specific case at hand.

Consider the specific case of $\lambda_0 = 4$ and $s_0 = -3$. Substitution into the quadratic equation gives $\beta = -1$ or -3. The general solution Eq. (3.73) becomes

$$u(x) = C_1 e^x + C_2 e^{3x}\,, \tag{3.83}$$

where C_1 and C_2 are not necessarily the same constants as in Eq. (3.73). The two different solutions for β just swap the constants in the two terms above and thus give the same solution for $u(x)$.

Harmonic oscillator in one dimension This example of a harmonic oscillator potential in one dimension adds another level of sophistication onto the previous example. The Schrödinger equation is

$$\left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{k}{2}x^2\right]\psi = E\psi\,. \tag{3.84}$$

We take units in which $\hbar^2/(2m) = 1$ and $k/2 = 1$. When $x \to \pm\infty$, the wave function ψ must vanish. Asymptotically ψ behaves like a Gaussian function, suggesting a factorized solution

$$\psi(x) = \exp(-x^2/2)u(x)\,. \tag{3.85}$$

The differential equation for $u(x)$ becomes

$$u'' = 2xu' + (1 - E)u\,. \tag{3.86}$$

This is the Hermite equation and we set $1 - E = -2k$ for convenience later.

The Hermite equation has $\lambda_0 = 2x$ and $s_0 = -2k$. Next we need to solve the quantization condition Eq. (3.68). I have done it using SymPy code below.

```
from sympy import *
l0, l, ln, s0, s, sn = symbols('l0 l ln s0 s sn')
x, k, delta = symbols('x k delta')
l0 =  2*x
s0 = -2*k
l = l0
s = s0
for n in range(0,3):
    ln = diff(l,x) + s + l0*l
    sn = diff(s,x) + s0*l
    a = sn/ln
    delta = factor(simplify(ln*s - l*sn))
    print(f"n: {n+1}\t delta: {delta}")
    l = ln
    s = sn
```

```
n: 0  delta: 4*k*(k - 1)
n: 1  delta: 8*k*(k - 2)*(k - 1)
n: 2  delta: 16*k*(k - 3)*(k - 2)*(k - 1)
```

We deduce the general form

$$\delta = 2^{n+2} \prod_{i=0}^{n+1} (k-i)\,, \quad n = 0, 1, 2, \ldots\,. \tag{3.87}$$

For $\delta = 0$, k is a non-negative integer. Therefore the energy eigenvalues are $E_k = 2k + 1, k = 1, 2, 3, \ldots$. We leave it as an exercise for the reader to write down the eigenfunctions. It helps to recognize that k is the order of the Hermite equation.

Pöschl-Teller potential Now an example requiring root-finding, albeit symbolically. The Pöschl-Teller potential is a common example in quantum mechanics:

$$V(x) = \frac{1}{2}\mathrm{sech}^2 x\,, \tag{3.88}$$

giving the Schrödinger equation

$$\frac{d^2\psi}{dx^2} + \left[\omega^2 - \frac{1}{2}\mathrm{sech}^2 x\right]\psi = 0\,, \tag{3.89}$$

where ω is the energy.

To work with the equation in the finite domain, we make a transformation $y = \tanh x$, which gives

$$\begin{aligned} \left(1-y^2\right)\frac{d}{dy}\left[\left(1-y^2\right)\frac{d\psi}{dy}\right] + \left[\omega^2 - \frac{1}{2}\left(1-y^2\right)\right] = 0\,, \\ \frac{\partial^2\psi}{dy^2} - \left(\frac{2y}{1-y^2}\right)\frac{\partial\psi}{\partial y} + \left[\frac{\omega^2}{(1-y^2)^2} - \frac{1}{2(1-y^2)}\right]\psi = 0\,. \end{aligned} \tag{3.90}$$

Since $-\infty < x < \infty$, we have $-1 < y < 1$. We will require, so called, quasinormal-mode boundary conditions:

$$x \to \infty \Rightarrow \psi \to e^{-i\omega x} \quad \text{and} \quad x \to -\infty \Rightarrow \psi \to e^{i\omega x}\,. \tag{3.91}$$

Recognize that

$$x = \tanh^{-1} y = \frac{1}{2}\ln\left(\frac{1+y}{1-y}\right)\,, \tag{3.92}$$

$$y \to 1 \Rightarrow \psi \sim -\frac{1}{2}\ln(1-y)\,, \tag{3.93}$$

$$y \to -1 \Rightarrow \psi \sim \frac{1}{2}\ln(1+y)\,. \tag{3.94}$$

Including both boundary conditions as factorizable contributions, a trial solution is

$$\psi = (1-y)^{-i\omega/2}(1+y)^{-i\omega/2}\phi\,. \tag{3.95}$$

After some algebra—which you might consider doing using SymPy—we obtain

$$\frac{\partial^2\phi}{dy^2} = \frac{2y(1-i\omega)}{1-y^2}\frac{d\phi}{dy} + \frac{1-2i\omega-2\omega^2}{2(1-y^2)}\phi\,, \tag{3.96}$$

where

$$\lambda_0 = \frac{2y(1-i\omega)}{1-y^2} \quad \text{and} \quad s_0 = \frac{1-2i\omega-2\omega^2}{2(1-y^2)}\,. \tag{3.97}$$

Requiring $\delta = 0$, gives a polynomial in complex ω to order $2(n+1)$. We thus need to find $2(n+1)$ roots for each n. I have found them using the following SymPy code.

```
from sympy import *
l0, l, ln, s0, s, sn = symbols('l0 l, ln, s0, s, sn')
y, w, delta = symbols('y w delta')
l0 =  2*y*(1 - 1j*w) / (1-y**2)
s0 = (1 - 2j*w - 2*w**2) / 2 / (1-y**2)
l = l0
s = s0
for n in range(0,3):
    ln = diff(l,y) + s + l0*l
    sn = diff(s,y) + s0*l
    delta = simplify(ln*s - l*sn)
    eig = solve(delta,w)
    for e in eig:
      print(f"n: {n+1}\t w: {complex(e).real: .2f} {complex(e).
    imag}j")
    l = ln
    s = sn
```

```
n: 1  w: -0.50 -1.5j
n: 1  w: -0.50 -0.5j
n: 1  w:  0.50 -1.5j
n: 1  w:  0.50 -0.5j
n: 2  w: -0.50 -2.5j
n: 2  w: -0.50 -1.5j
n: 2  w: -0.50 -0.5j
n: 2  w:  0.50 -2.5j
n: 2  w:  0.50 -1.5j
n: 2  w:  0.50 -0.5j
n: 3  w: -0.50 -3.5j
n: 3  w: -0.50 -2.5j
n: 3  w: -0.50 -1.5j
n: 3  w: -0.50 -0.5j
n: 3  w:  0.50 -3.5j
n: 3  w:  0.50 -2.5j
n: 3  w:  0.50 -1.5j
n: 3  w:  0.50 -0.5j
```

We see that for each $n > 1$, two new roots are obtain, in addition to all

previous $2n$ roots. We deduce the general form

$$\omega_n = \pm\frac{1}{2} - i\left(n + \frac{1}{2}\right) \quad n = 0, 1, 2, \ldots . \tag{3.98}$$

3.1.9 Frobenius method

I will illustrate the Frobenius method by using the example of calculating the quasinormal modes from a perturbed Schwarzschild black hole. If the problem is simple enough, the method is often referred to as a series solution method. However, if the ODE contains singularities and has complicated boundary conditions, the full power the Frobenius method is required.

Consider the following wave equation which we will solve as a boundary-value eigenvalue problem.

$$\left[\frac{\partial^2}{\partial r_*^2} + \omega_{\ell,s}^2 - V_{\ell,s}(r)\right]\psi_{\ell,s}(r_*, t) = 0\,, \tag{3.99}$$

where

$$\frac{dr_*}{dr} = \frac{1}{f(r)} \quad \text{and} \quad V_{\ell,s}(r) = f(f)\left[\frac{\ell(\ell+1)}{r^2} + \frac{1-s^2}{r^3}\right], \tag{3.100}$$

such that $f(r) = 1 - 2M/r$ and $s = 0, 1, 2$, $\ell \geq s$. The parameters $\omega_{\ell,s}$ are a set of complex eigenvalues to be determined. Without loss of generality, we may pick $M = 1/2$.

For the Frobenius method, we will find it convenient to first transform the differential equation to the r variable:

$$r(r-1)\psi''_{\ell,s} + \psi'_{\ell,s} + \left[\frac{\omega_{\ell,s}^2 r^3}{r-1} - \ell(\ell+1) + \frac{s^2-1}{r}\right]\psi_{\ell,s} = 0\,, \tag{3.101}$$

where primes denote differentiation with respect to r. We will be interested in the solution in the range $r = [1, \infty)$. The ODE has regular singularities at $r = 0$ and $r = 1$, and an irregular singularity at $r = \infty$.

We postulate a product solution consisting of the asymptotic wavefunction $\phi(r)$ and a series approximation. For the asymptotic solution, we need to consider the boundary conditions. The boundary conditions for this problem are (dropping subscripts ℓ, s and considering them to be implicit)

$$\psi(r \to \infty) = e^{i\omega r_*} \quad \text{and} \quad \psi(r \to 1) = e^{-i\omega r_*}\,, \tag{3.102}$$

where the waves are purely outgoing and ingoing, respectively. We change the r_* dependence to r by solving dr_*/dr:

$$r_* = r + \ln(r-1)\,, \tag{3.103}$$

which has asymptotic limits

$$\begin{aligned} r \to \infty \quad &\Rightarrow \quad r_* = r + \ln r \quad \text{and} \\ r \to 1 \quad &\Rightarrow \quad r_* = \ln(r-1)\,. \end{aligned} \tag{3.104}$$

The boundary conditions can be written as

$$\psi(r \to \infty) \to e^{i\omega r} r^{i\omega} \quad \text{and} \quad \psi(r \to 1) \to (r-1)^{-i\omega}\,. \tag{3.105}$$

Normalization gives

$$\psi^{r\to\infty}(r \to 1) \to e^{i\omega(r-1)} r^{i\omega} \quad \text{and} \quad \psi^{r\to 1}(r \to \infty) \to (r-1)^{-i\omega} r^{i\omega}\,. \tag{3.106}$$

The asymptotic solution can be written as the produce of the two asymptotic solutions

$$\phi(r) = (r-1)^{-i\omega} r^{2i\omega} e^{u\omega(r-1)}\,. \tag{3.107}$$

We plug the product solution into the ODE in r and operate on $\phi(r)$ only. This give an ODE for the remaining series solution part. Next, the singularities at $(0, 1, \infty)$ are mapped to $(\infty, 0, 1)$ by the change of variable

$$u = \frac{r-1}{r}\,. \tag{3.108}$$

The domain $[1, \infty)$ now maps to $[0, 1)$ which is advantageous for numerical computation. The total solution can be written as

$$\psi(r) = \phi(r) \sum_{n=0}^{\infty} a_n \left(\frac{r-1}{r}\right)^n\,. \tag{3.109}$$

The change of variable $r \to u$ is now applied to the ODE that operates on the series solution only. Differentiating the series gives a sequence of expansion coefficients a_n which are determined by a three-term recurrence relation

$$\begin{aligned} \alpha_0 a_1 + \beta_0 a_0 = 0 \quad &\text{and} \\ \alpha_n a_{n+1} + \beta_n a_n \gamma_n a_{n-1} = 0\,, \ &\text{for} \quad n = 1, 2, \ldots\,. \end{aligned} \tag{3.110}$$

The recurrence coefficients α_n, β_n and γ_n are simple functions of n and the parameters of the differential equation:

$$\begin{aligned} \alpha_n &= n^2 + 2(1 - i\omega)n - 2i\omega + 1\,, \\ \beta_n &= -[n^2 + (2 - 8i\omega)n - 8\omega^2 - 4i\omega + \ell(\ell+1) + 1 - s^2]\,, \\ \gamma_n &= n^2 - 4i\omega n - 4\omega^2 - s^2\,. \end{aligned} \tag{3.111}$$

The boundary conditions at spatial infinity will be satisfied for those values of $\omega = \omega_n$ for which the series is absolutely convergent, i.e. for which $\sum a_n$ exists and is finite.

The ratio of successive a_n is given by the infinite continued fraction

$$\frac{a_{n+1}}{a_n} = \frac{-\gamma_{n+1}}{\beta_{n+1} - \frac{\alpha_{n+1}\gamma_{n+2}}{\beta_{n+2} - \frac{\alpha_{n+2}\gamma_{n+3}}{\beta_{n+3} - \cdots}}} . \tag{3.112}$$

The usual notation for such a continued fraction is

$$\frac{a_{n+1}}{a_n} = \frac{-\gamma_{n+1}}{\beta_{n+1}-} \frac{\alpha_{n+1}\gamma_{n+2}}{\beta_{n+2}-} \frac{\alpha_{n+2}\gamma_{n+3}}{\beta_{n+3}-} \cdots . \tag{3.113}$$

The recurrence relations give

$$\frac{a_1}{a_0} = -\frac{\beta_0}{\alpha_0} \quad \text{and} \quad \frac{a_1}{a_0} = -\beta_0 \frac{\alpha_0\gamma_1}{\beta_1-} \frac{\alpha_1\gamma_2}{\beta_2-} \frac{\alpha_2\gamma_3}{\beta_3-} \cdots . \tag{3.114}$$

Combined the two expressions give the desired (implicit) characteristic equation for the eigenvalues:

$$0 = \beta_0 - \frac{\alpha_0\gamma_1}{\beta_1-} \frac{\alpha_1\gamma_2}{\beta_2-} \frac{\alpha_2\gamma_3}{\beta_3-} \cdots . \tag{3.115}$$

This equation may be inverted an arbitrary number of times n to give

$$\beta_n - \frac{\alpha_{n-1}\gamma_n}{\beta_{n-1}-} \frac{\alpha_{n-2}\gamma_{n-1}}{\beta_{n-2}-} \cdots - \frac{\alpha_0\gamma_1}{\beta_0} = \frac{\alpha_n\gamma_{n+1}}{\beta_{n+1}-} \frac{\alpha_{n+1}\gamma_{n+2}}{\beta_{n+2}-} \frac{\alpha_{n+2}\gamma_{n+3}}{\beta_{n+3}-} \cdots . \tag{3.116}$$

The expression on the left-hand side is a finite sum which can be calculated using back calculation starting with $\alpha_0\gamma_1/\beta_0$. The right-hand side is an infinite sum which can be calculated using standard continued fraction algorithms.

For any method of continued fraction solution, the determination of the eigenvalues is now reduced to the numerical problem of finding roots of the above equation. The nth eigenvalue is usually found to be numerically the most stable root of the nth inversion [11].

3.1.10 Examples: Ordinary differential equations

We'll now consider two examples comparing the Euler and modified Euler methods. Then we will consider a simple shooting-method problem. Lastly, I'll introduce streamlines and how to visualize the electric field and potential of a collection of static point charges.

3.1.10.1 Nonlinear equation

For the first example, we'll solve the following nonlinear differential equation numerically and compare it with the analytical solution by showing their ratio. Consider the first-order nonlinear differential equation

$$\frac{dy}{dx} = y^2 e^x . \tag{3.117}$$

The analytic solution is

$$y = \frac{-1}{e^x + C}\,, \tag{3.118}$$

where C is a constant of integration obtained from an initial condition or boundary value. We take $C = 2$.

Numerically we'll solve for y in the range $0 \leq x \leq 10$ with step size $h = 0.01$. Let's calculate the solution ourselves using the Euler method and modified Euler method. In the code below

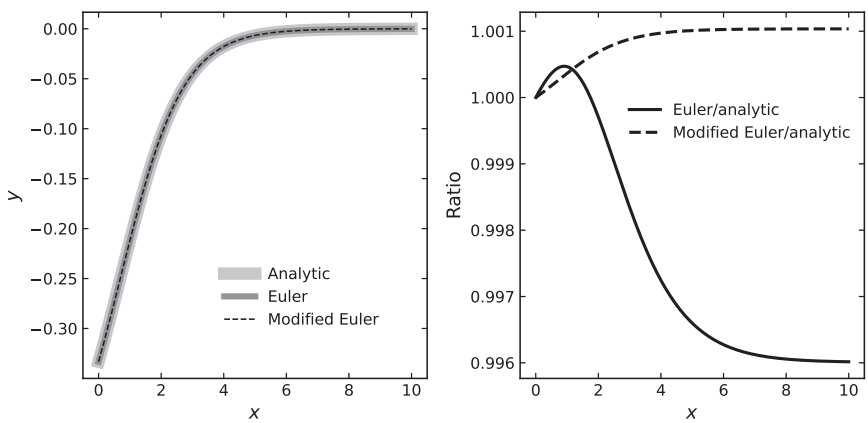
$$f(x_i, y_i) = y_i^2 e^{x_i}\,, \tag{3.119}$$

$$\begin{aligned} x_{i+1} &= x_i + h \\ y_{i+1} &= y_i + hf(x_i, y_i) \quad \text{Euler} \\ y_{i+1} &= y_i + hf\left(x_i + \frac{h}{2}, y_i + \frac{h}{2} f(x_i, y_i)\right) \quad \text{modified Euler} \end{aligned} \tag{3.120}$$

```
"""Solve nonlinear ODE using Euler method."""

import numpy as np
from matplotlib import pyplot as plt

plt.style.use('./mystyle.mplstyle')

# Define function.
def f(x,y):
    return y**2*np.exp(x)

# Parameters of problem.
xmin =  0.0
xmax = 10.0
h = 0.01
N = int((xmax-xmin)/h)

# Calculate analytic integral.
C = 2.0
x1 = np.linspace(xmin,xmax,N)
y1 = -1.0/(np.exp(x1)+C)
print("min x",x1[0],"max x",x1[-1],"step size",h,
      "number of points",N)

# Storage.
y2 = np.empty(N) # Euler
y3 = np.empty(N) # Modified Euler

# Initial condition.
y0 = -1.0/(np.exp(xmin)+C);
y2[0] = y0
y3[0] = y0

# Iterate.
```

```
for i in range(N-1):
    # Euler
    y2[i+1] = y2[i] + h*f(x1[i],y2[i])
    # Modified Euler
    P = y3[i] + (h/2)*f(x1[i],y3[i])
    y3[i+1] = y3[i] + h*f(x1[i]+h/2,P)

plt.figure(figsize=(8,4))
# Plot theory, Euler and modified Euler on top of each other.
plt.subplot(1,2,1)
plt.plot(x1,y1,lw=8,ls='solid',c='silver',label="Analytic")
plt.plot(x1,y2,lw=4,ls='solid',c='grey',  label="Euler")
plt.plot(x1,y3,lw=1,ls='dashed',          label="Modified Euler")
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")
plt.legend(loc="lower right",bbox_to_anchor=(0.9,0.1))

# Plot ratios.
plt.subplot(1,2,2)
plt.plot(x1,y2/y1,lw=2,ls='solid',
         label="Euler/analytic")
plt.plot(x1,y3/y1,lw=2,ls='dashed',
         label="Modified Euler/analytic")
plt.xlabel(r"$x$")
plt.ylabel("Ratio")
plt.legend(loc="center right",bbox_to_anchor=(1,0.7))

plt.tight_layout()
plt.savefig('3_1_ode_1.pdf')
plt.show()
```

```
min x 0.0 max x 10.0 step size 0.01 number of points 1001
```



The plot on the left-hand side indicates that the analytic solution is hidden under the numerical methods. Thus the modified-Euler method is exact to the eye. By taking the ratio of the numerical solution to the analytic solution in

the right-hand plot, we see that the modified-Euler method over estimates the solution by much less than a fraction of a percent. The Euler method overestimates, then underestimates the solution by about 0.5% at the highest value of x.

3.1.10.2 Simple harmonic oscillator

The simple harmonic oscillator is one of the prototypical examples in all areas of physics. Using numerical methods we are able to solve cases of a so-called physical oscillator including the effects of friction, size, driving forces and so forth. Nevertheless, in this example we will look at a simple harmonic oscillator once more, emphasizing the numerical method and validation. It should be clear how to extend the method to more realistic models.

The coupled first-order differential equations for a simple harmonic oscillator in one dimension are

$$\frac{dx}{dt} = v \quad \text{and} \quad m\frac{dv}{dt} = -kx\,. \tag{3.121}$$

We will calculate the solution ourselves using the Euler method and modified-Euler method. We plot the position and velocity as a function of time, as well as the total energy. Take the initial conditions to be $x(t = 0) = 0$ and $v(t = 0) = 5$.

The analytical solution is

$$x(t) = A\sin(\omega t)\,, \quad v(t) = \omega A\cos(\omega t)\,, \quad \omega = \sqrt{\frac{k}{m}}\,. \tag{3.122}$$

From energy considerations

$$E = \frac{1}{2}kx^2(t) + \frac{1}{2}mv^2(t)\,. \tag{3.123}$$

In the following code one needs to set `euler = True` to use the Euler method, else the modified-Euler method is used. The values of the `parm` variables choose between three differ values of (k, m).

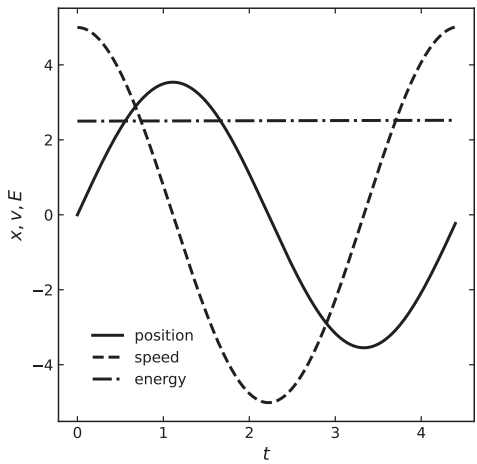
```
"""Simple harmonic oscillator"""

import numpy as np
from matplotlib import pyplot as plt

plt.style.use('./mystyle.mplstyle')

def energy(x,v,k,m):
    # Calculate total energy.
    e = 0.5*k*x**2 + 0.5*m*v**2
    e /= 5.0 # Arbitrary scaling to fit on plot.
    return e

# Pick between Euler or modified Euler.
euler = True
if euler:
```

```
    print("Using Euler method")
else:
    print("Using modified-Euler method")

# Oscillator parameters (3 choices).
parm = 2
k, m = np.array([[1,1],[1,2],[2,1]])[parm]
print("k",k,"m",m)

# Step size (3 choices).
step = 2
h = np.array([0.1,0.01,0.001])[step]
print("Step size",h)

# Plot range (in units of pi).
R = 1.4

# Storage.
N = int(R*np.pi/h)
t = np.empty(N)
x = np.empty(N)
v = np.empty(N)
e = np.empty(N)

# Initial values.
t[0] = 0.0
x[0] = 0.0
v[0] = 5.0
e[0] = energy(x[0],v[0],k,m)

for i in range(int(R*np.pi/h)-1): # Time range of R*pi.
    if euler:
        v[i+1] = v[i] - h * (k/m) * x[i]
        x[i+1] = x[i] + h * v[i]
    else:
        v_mid = v[i] - (h/2) * (k/m) * x[i]
        x_mid = x[i] + (h/2) * v[i]
        v[i+1] = v[i] - h * (k/m) * x_mid
        x[i+1]= x[i] + h * v_mid
    t[i+1] = t[i] + h
    e[i+1] = energy(x[i+1],v[i+1],k,m)

# Make plots. y-axis represents position, speed, and energy
fig, ax = plt.subplots()
plt.plot(t,x,lw=2,ls='solid',  label="position")
plt.plot(t,v,lw=2,ls='dashed', label="speed")
plt.plot(t,e,lw=2,ls='dashdot',label="energy")
plt.xlabel(r"$t$")
plt.ylabel(r"$x, v, E$")
plt.legend(loc="lower left",bbox_to_anchor=(0.05,0.05))
plt.savefig('3_1_ode_2.pdf')
plt.show()
```

```
Using Euler method
k 2 m 1
Step size 0.001
```

Theory calculations

k	m	ω	T	$v_{\max}$	$x_{\max}$	$\frac{E}{5}$
1	1	1	2π	5	5	$\frac{5}{2}$
1	2	$\frac{1}{\sqrt{2}}$	$2\sqrt{2}\pi$	5	$5\sqrt{2}$	5
2	1	$\sqrt{2}$	$\sqrt{2}\pi$	5	$\frac{5}{\sqrt{2}}$	$\frac{5}{2}$



Runs

```
Euler: step = 0.1 is terrible
Euler: step = 0.01 energy increasing over short time
Euler: step = 0.001 energy increasing over long time

Modified Euler: step = 0.1 energy increases over long time
Modified Euler: step = 0.01, energy constant over long time
Modified Euler: step = 0.001, energy constant over long time
```

The code examines three different choices of parameters (m, k) for three different step sizes; this is what I mean by numerical experiments. A theoretical (analytical) calculation table is presented showing the expected frequency, period, maximum speed, maximum displacement and energy for each choice of parameters. Notice I have calculated everything I could think of for comparison with the numerical results. It is often not good enough to just compare one quantity, if more can be compared.

Following the table are comments on the runs for different step sizes. The modified-Euler method is superior and the Euler method is unsatisfactory even for a step size of $h = 0.001$.

We have plotted three quantities with different units on the same plot. Typically not a good thing to do, especially without labeling the y-axis. The

plot is of course only for one of the run conditions. The results are validated by the theory calculations, and the energy is constant.

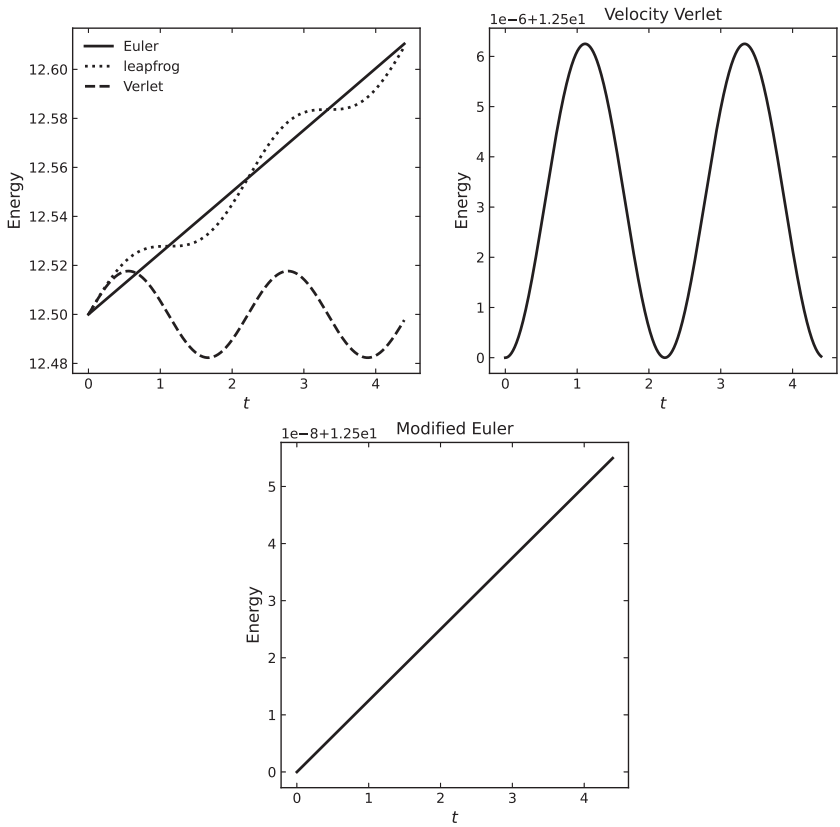
This is an example of where validation has benefited from physics. One can also validate by using maths. That is, calculating simple or approximate cases analytically and comparing with the numerical solution. The hope is then that we will have some confidence in the results when solving cases numerically that have no analytical information. One can also validate using numerical methods; solving parts, or all, of the numerical problem using alternative methods. But if the problem is a physical process with underlying symmetry principles, such as conservation of energy, then this can be a powerful tool for validation. By ensuring the energy is constant over a long time range in the above problem, we have some confidence in our implementation—namely stability—for calculating the position and speed.

Energy comparison We now use the calculation of the energy to compare the different leapfrog-type schemes. The following code shows how to iterate the different schemes.

```
"""Simple harmonic oscillator energy comparison:
Euler, modifed Euler, leapfrog, Verlet, velocity Verlet.
"""
import numpy as np
from matplotlib import pyplot as plt
import matplotlib.gridspec as gridspec

plt.style.use('./mystyle.mplstyle')

# Calculate total energy.
def energy(x,v,k,m):
    return 0.5*k*x**2 + 0.5*m*v**2

# Oscillator parameters (3 choices).
parm = 2
k, m = np.array([[1,1],[1,2],[2,1]])[parm]
print("k",k,"m",m)

# Step size (3 choices).
step = 2
h = np.array([0.1,0.01,0.001])[step]
print("Step size",h)

# Plot range (in units of pi).
R = 1.4

fig = plt.figure(figsize=(10,10))
gs = gridspec.GridSpec(2,4,wspace=0.5,hspace=0.2)

# Loop over different schemes.
methods = ["Euler","Modified Euler","leapfrog",
           "Verlet","Velocity Verlet"]
```

```
for meth in methods:
    # Storage.
    N = int(R*np.pi/h)
    t = np.empty(N)
    x = np.empty(N)
    v = np.empty(N)
    e = np.empty(N)

    # Initial values.
    t[0] = 0.0
    x[0] = 0.0
    v[0] = 5.0
    e[0] = energy(x[0],v[0],k,m)

    # Loop over time range.
    for i in range(int(R*np.pi/h)-1): # Time range of R*pi.
        if meth == "Euler":
            if i == 0: print(meth)
            v[i+1] = v[i] - h*(k/m)*x[i]
            x[i+1] = x[i] + h*v[i]
        elif meth == "Modified Euler":
            if i == 0: print(meth)
            v_mid = v[i] - (h/2)*(k/m)*x[i]
            x_mid = x[i] + (h/2)*v[i]
            v[i+1] = v[i] - h*(k/m)*x_mid
            x[i+1]= x[i] + h*v_mid
        elif meth == "leapfrog":
            if i == 0: print(meth)
            if i == 0:
                v_half = v[0] - 0.5*h*(k/m)*x[0]
            x[i+1] = x[i] + h*v_half
            v[i+1] = v_half - 0.5*h*(k/m)*x[i]
            v_half = v_half - h*(k/m)*x[i]
        elif meth == "Verlet":
            if i == 0: print(meth)
            if i == 0:
                x_minus = x[0] - h*v[0] - 0.5*h*h*x[0]
                x[i+1] = 2*x[i] - x_minus - h*h*(k/m)*x[i]
                v[i+1] = 0.5*(x[i+1] - x_minus)/h
            else:
                x[i+1] = 2*x[i] - x[i-1]- h*h*(k/m)*x[i]
                v[i+1] = 0.5*(x[i+1] - x[i-1])/h
        elif meth == "Velocity Verlet":
            if i == 0: print(meth)
            x[i+1] = x[i] + h*v[i] - 0.5*h*h*(k/m)*x[i]
            v[i+1] = v[i] - 0.5*h*(k/m)*(x[i] + x[i+1])
        else:
            print("Invalid scheme.")
        t[i+1] = t[i] + h
        e[i+1] = energy(x[i+1],v[i+1],k,m)

    # Make plots.
    if meth == "Euler" or meth == "Verlet" or \
       meth == "leapfrog":
      ax1 = plt.subplot(gs[0,:2],)
      if meth == "Euler":
          ax1.plot(t,e,ls='solid', label=meth)
```

```
      if meth == "Verlet":
          ax1.plot(t,e,ls='dashed',label=meth)
      if meth == "leapfrog":
          ax1.plot(t,e,ls='dotted',label=meth)
      ax1.set_xlabel(r"$t$")
      ax1.set_ylabel(r"Energy")
      ax1.legend()
    elif meth == "Velocity Verlet":
      ax2 = plt.subplot(gs[0,2:])
      ax2.plot(t,e,label=meth)
      ax2.set_title("Velocity Verlet")
      ax2.set_xlabel(r"$t$")
      ax2.set_ylabel(r"Energy")
    elif meth == "Modified Euler":
      ax3 = plt.subplot(gs[1,1:3])
      ax3.plot(t,e,label=meth)
      ax3.set_title("Modified Euler")
      ax3.set_xlabel(r"$t$")
      ax3.set_ylabel(r"Energy")

plt.savefig('3_1_leapfrog.pdf')
plt.show()
```



The figures shows a comparison of the energy determined by the five different schemes. A common step size of $h = 0.001$ has been used to allow fair comparison. Theoretically, the energy is constant at a values of 12.5 (no scaling). Three different plots have been made to accommodate the different energy ranges. The Verlet and velocity-Verlet schemes are superior in that the energy does not appear to be growing with time, and hence these scheme are stable for this example. The Euler, modified-Euler and leapfrog schemes show growing energy with time, and are thus unstable for this example. The Verlet and velocity-Verlet schemes show energy oscillations at different frequencies. The velocity-Verlet oscillation amplitude is about a factor of 10^{-3} smaller than the Verlet amplitude. The modified-Euler scheme has energy grow about a factor of 10^{-5} smaller than the Euler or leapfrog schemes. Nevertheless, over a long enough time period, the growth in the energy in the modified-Euler scheme might be an issue in the solution.

3.1.10.3 Boundary value problem

This is an example of solving a simple boundary value problem using a system of equations. Consider the one-dimensional ODE,

$$\frac{\partial^2 \phi}{\partial x^2} = -10\,, \tag{3.124}$$

in the interval $x = [0, 1]$. In this examples, we will consider Dirichlet boundary conditions at both boundaries, and a Dirichlet boundary condition at one boundary and a Neumann boundary condition at the other.

First consider Dirichlet boundary conditions $\phi(x = 0) = 1$ and $\phi(x = 1) = 2$. Divide the interval $[0, 1]$ into $N = 5$ discrete points x_n excluding the boundaries. We will find the function $\phi(x_n) = \phi_n$ at these points by write the ODE problem in finite-difference form for every interior points n and the boundary conditions $\phi_0 = 1$ and $\phi_{N+1} = 2$. Discretize the interval $[0, 1]$ into $(N + 2)$ evenly-spaced points, including the boundary points, with separation $\Delta x = 1/(N+1)$. Then we write the set of equations in matrix form as $A\phi = b$, where $\phi = \phi_1, \ldots, \phi_N$, taking into account the boundary conditions.

More generally,

$$\frac{\partial^2 \phi}{\partial x^2} = C\,, \tag{3.125}$$

where $C = -10$, and $\phi(0) = 1$, $\phi(1) = 2$. Using central differences for $i = 2, 3, 4$,

$$\frac{\phi_{i-1} - 2\phi_i + \phi_{i+1}}{(\Delta x)^2} = C\,. \tag{3.126}$$

For $i = 1$,

$$\frac{\phi(0) - 2\phi_1 + \phi_2}{(\Delta x)^2} = C \quad \Rightarrow \quad \frac{-2\phi_1 + \phi_2}{(\Delta x)^2} = C - \frac{\phi(0)}{(\Delta x)^2}\,. \tag{3.127}$$

For $i = 5$,

$$\frac{\phi_4 - 2\phi_5 + \phi(1)}{(\Delta x)^2} = C \quad \Rightarrow \quad \frac{\phi_4 - 2\phi_5}{(\Delta x)^2} = C - \frac{\phi(1)}{(\Delta x)^2}\,. \tag{3.128}$$

In matrix form,

$$\frac{1}{(\Delta x)^2}\begin{pmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}\begin{pmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \phi_5 \end{pmatrix} = \begin{pmatrix} C - \frac{\phi(0)}{(\Delta x)^2} \\ C \\ C \\ C \\ C - \frac{\phi(1)}{(\Delta x)^2} \end{pmatrix}. \tag{3.129}$$

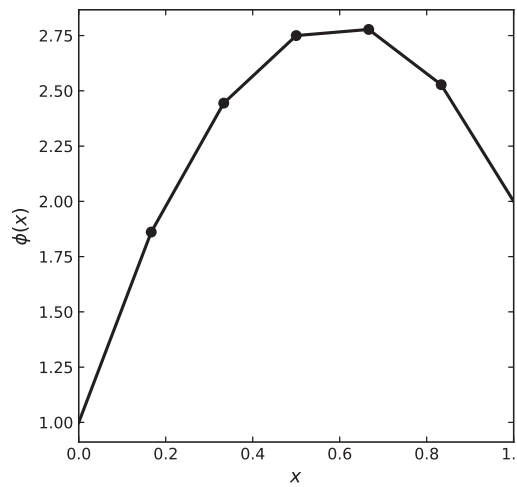
Aside: the analytical solution to the differential equation is $\phi(x) = -5x^2 + 6x + 1$. The maximum is at $x = \frac{3}{5} = 0.6$ and has a value $\phi\left(x = \frac{3}{5}\right) = \frac{14}{5} = 2.8$.

Consider the following code that solves the system of linear equations for the unknown vector ϕ. I have used Python library modules to solve this system of equations. A plot of $\phi(x)$ versus x including the boundary values is shown.

```
"""Solve boundary value problem using a system of equations.
Both Dirichlet boundary conditions.
"""
import numpy as np
import scipy.sparse as sp
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

N = 5
C = -10
phi0, phi1 = 1, 2
dx = 1.0 / (N + 1)

A = (np.eye(N,k=-1) -2*np.eye(N) + np.eye(N,k=1)) / dx**2
print("A =")
print(A)
print()
print("condition number",np.round(np.linalg.cond(A),1))
print()

b = C * np.ones(N)
b[0]   -= phi0 / dx**2
b[N-1] -= phi1 / dx**2
print("b =",b)

u = np.linalg.solve(A,b)
U = np.hstack([[phi0],u,[phi1]])
x = np.linspace(0,1,N+2)

fig, ax = plt.subplots()
ax.plot(x,U)
ax.plot(x[1:-1],u,'ko')
ax.set_xlabel(r"$x$")
```

```
ax.set_ylabel(r"$\phi(x)$")
ax.set_xlim(0,1)
plt.savefig('3_1_bvp_direchlet.pdf')
plt.show()
```

```
A =
[[-72.  36.   0.   0.   0.]
 [ 36. -72.  36.   0.   0.]
 [  0.  36. -72.  36.   0.]
 [  0.   0.  36. -72.  36.]
 [  0.   0.   0.  36. -72.]]

condition number 13.9

b = [-46. -10. -10. -10. -82.]
```



The condition number is under 100, so the matrix is probably not ill-conditioned. The plot follows the analytical solution and has a maximum in the correct location.

We now solve the case of mixed Neumann-Dirichlet boundary conditions. Replace the Dirichlet boundary condition $\phi(x = 0) = 1$ with the Neumann boundary condition $\partial\phi/\partial x = 0$ at $x = 0$. I will create a ghost-cell with negative index -1 outside the physical domain to express the derivative at the boundary. Note that the requirement of a zero derivative relates the value of the ghost-cell to a value inside the physical domain.

Using central differences for $i = 1, 2, 3, 4$,

$$\frac{\phi_{i-1} - 2\phi_i + \phi_{i+1}}{(\Delta x)^2} = C\,. \tag{3.130}$$

For $i = 0$,

$$\frac{\phi_{-1} - 2\phi_0 + \phi_1}{(\Delta x)^2} = C\,. \tag{3.131}$$

For Neumann boundary condition,

$$\frac{\phi_1 - \phi_{-1}}{2\Delta x} = 0 \quad \Rightarrow \quad \phi_{-1} = \phi_1 \quad \text{and} \quad \frac{-2\phi_0 + 2\phi_1}{(\Delta x)^2} = C\,. \tag{3.132}$$

For $i = 5$,

$$\frac{\phi_4 - 2\phi_5 + \phi(1)}{(\Delta x)^2} = C \quad \Rightarrow \quad \frac{\phi_4 - 2\phi_5}{(\Delta x)^2} = C - \frac{\phi(1)}{(\Delta)^2}\,. \tag{3.133}$$

In matrix form,

$$\frac{1}{(\Delta x)^2}\begin{pmatrix} -2 & 2 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}\begin{pmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \phi_5 \end{pmatrix} = \begin{pmatrix} C \\ C \\ C \\ C \\ C - \frac{\phi(1)}{(\Delta x)^2} \end{pmatrix}. \tag{3.134}$$

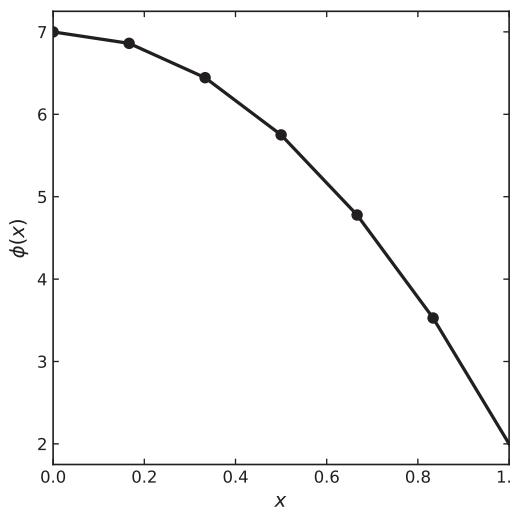
Aside: the analytical solution to the differential equation is $\phi(x) = -5x^2 + 7$. The maximum is at $x = 0$ and has a value $\phi(x = 0) = 7$.

Consider the following code to solve the system of linear equations for the unknown vector ϕ. I have used Python library modules to solve this system of equations. A plot $\phi(x)$ versus x including the boundary value is shown.

```
"""Solve boundary value problem using a system of equations.
Mixed Dirichlet and Neumann boundary conditions.
"""

import numpy as np
import scipy.sparse as sp
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

N = 6
C = -10
phi1 = 2
dx = 1.0 / N

A = (np.eye(N,k=-1) -2*np.eye(N) + np.eye(N,k=1)) / dx**2
A[0,1] = 2 / dx**2
print("A =")
print(A)
print()
print("condition number",np.round(np.linalg.cond(A),1))
print()

b = C * np.ones(N)
```

```
b[N-1] -= phi1 / dx**2
print("b =",b)

u = np.linalg.solve(A,b)
U = np.hstack([u,[phi1]])
x = np.linspace(0,1,N+1)

fig, ax = plt.subplots()
ax.plot(x,U)
ax.plot(x[0:-1],u,'ko')
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$\phi(x)$")
ax.set_xlim(0,1)
plt.savefig('3_1_bvp_newmann.pdf')
plt.show()
```

```
A =
[[-72.  72.   0.   0.   0.   0.]
 [ 36. -72.  36.   0.   0.   0.]
 [  0.  36. -72.  36.   0.   0.]
 [  0.   0.  36. -72.  36.   0.]
 [  0.   0.   0.  36. -72.  36.]
 [  0.   0.   0.   0.  36. -72.]]

condition number 60.6

b = [-10. -10. -10. -10. -10. -82.]
```



The condition number is under 100, so the matrix is probably not ill-conditioned. The plot follows the analytical formula and has a maximum in the correct place.

For pedagogical purposes, this simple example is solved using a system of finite-difference equation. Given a choice, one would probably use `solve_bvp` from the `scipy.integrate` sub-package. The following code snippet reproduces the previous results; I have removed the plotting code for compactness.

```
import numpy as np
from scipy import integrate

def fun(x,y):
    return np.vstack([y[1],-10*np.ones_like(y[0])])

# Both Dirichlet conditions.
def bc_d(ya,yb):
    return np.array([ya[0]-1,yb[0]-2])

# Mixed Neumann and Dirichlet conditions.
def bc_n(ya,yb):
    return np.array([ya[1],yb[0]-2])

x = np.linspace(0,1,6)
y = np.zeros((2,len(x)))
sol_d = integrate.solve_bvp(fun,bc_d,x,y)
y_d = sol_d.sol(x)[0]
sol_n = integrate.solve_bvp(fun,bc_n,x,y)
y_n = sol_n.sol(x)[0]
```

3.1.10.4 Shooting method

This example applies the shooting method to one-dimensional motion in the Earth's gravitational field with air resistance. Consider the model

$$\frac{d^2y}{dt^2} = -g - b\frac{dy}{dt}\,. \tag{3.135}$$

Written as two first order ODEs gives

$$\frac{dy}{dt} = v \quad \text{and} \quad \frac{dv}{dt} = -g - bv\,. \tag{3.136}$$

Let's take $y = 0$ at the initial and final times, $t_i = 0$ and $t_f = 10\,\text{s}$, respectively. We want to know the initial velocity satisfying these boundary conditions when the coefficient of resistance is $b = 0.1$.

Consider the following code. First we solve the problem without air resistance and match the results to our expectations. Then we add air resistance and describe the change in motion. The differential equation is integrated using fourth-order Runge-Kutta for each iteration of the shooting method. A binary search is used to determine the optimal height (boundary value) at the final time. The trajectory in time-height space is calculated using `scipy.integrate.solve_ivp`. From this, we also calculate the maximum height and the time at which it occurs.

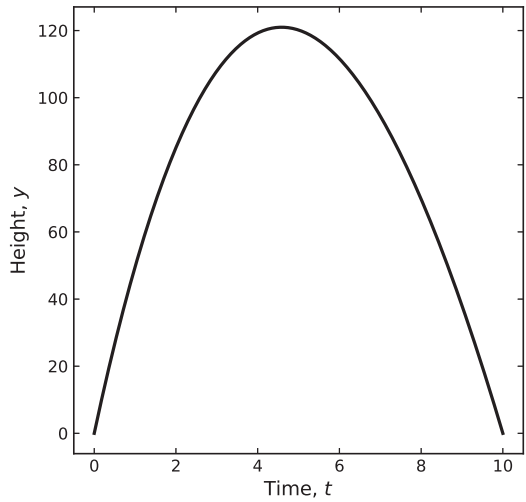
```
"""Motion with air resistance using shooting method"""

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

plt.style.use('./mystyle.mplstyle')

# Function to integrate.
def f(t,z,g,b):
    y    = z[0]
    v    = z[1]
    dydt =  v
    dvdt = -g - b*v
    return np.array([dydt,dvdt])

# Solve the equation of motion to obtain final height.
def height(v,g,b):
    t = 0 # Dummy, not used.
    z = np.array([y0,v],float)
    for t in np.arange(ti,tf,h):
        k1 = h * f(t,z,g,b)
        k2 = h * f(t,z+0.5*k1,g,b)
        k3 = h * f(t,z+0.5*k2,g,b)
        k4 = h * f(t,z+k3,g,b)
        z += (k1 + 2*k2 + 2*k3 + k4) / 6
    return z[0]

g   =  9.81 # Acceleration due to gravity.
b   =  0.1  # Resistance coefficient.
y0  =  0.0  # Initial and final height.
ti  =  0.0  # Initial time.
tf  = 10.0  # Final time.

N   = 1000      # Number of Runge-Kutta steps.
h   = (tf-ti)/N # Size of Runge-Kutta steps.
eps = 1e-10     # Target accuracy of binary search.

# Bracket the velocities (arbitrary) for a binary search.
v1 = 0.01
v2 = 1000
y1 = height(v1,g,b)
y2 = height(v2,g,b)
verbose = False
if verbose:
    print("v =",v1,"gives height",y1)
    print("v =",v2,"gives height",y2)

# Perform a binary search.
# (Too fast give positive heights and too slow negative heights.)
while abs(y2-y1) > eps:
    # Bisect range.
    vp = (v1 + v2) / 2
    yp = height(vp,g,b)
    # Above midpoint.
    if y1*yp > 0:
```

```
        v1 = vp
        y1 = yp
    # Below midpoint.
    else:
        v2 = vp
        y2 = yp
# Take final average.
v0 = (v1 + v2) / 2

# Solve using scipy.integrate.solve_bvp.
BVP = False
if BVP:
    def bc(ya,yb):
        return np.array([ya[0]-y0,yb[0]-y0])
    x = np.linspace(ti,tf)
    y = np.zeros((2,x.size))
    sol = integrate.solve_bvp(lambda t,z: f(t,z,g,b),bc,x,y)
    v0 = sol.yp[0,0]
    print(sol)

# Determine trajectory.
t = np.linspace(ti,tf,1000)
sol = integrate.solve_ivp(f,[ti,tf],[y0,v0],t_eval=t,args=(g,b))

# Print the results.
print(f"Required initial velocity {v0:.2f} m/s")
print(f"Final height with this velocity {height(v0,g,b):.1e} m")
print(f"Maximum height {sol.y[0].max():.2f} m"
      f" at time {sol.t[sol.y[0].argmax()]:.2f} s")

# Plot the trajectory.
fig, ax = plt.subplots()
plt.plot(sol.t,sol.y[0])
plt.xlabel(r"Time, $t$")
plt.ylabel(r"Height, $y$")
plt.savefig('3_1_shooting.pdf')
plt.show()
```

```
Required initial velocity 57.09 m/s
Final height with this velocity -5.9e-12 m
Maximum height 120.99 m at time 4.58 s
```

The initial velocity is significantly higher than the 49.05 m/s value without resistance. The final height or tolerance of the shooting method is tiny and hence the method is probably reliable. The trajectory with air resistance is no longer symmetric in time since the maximum height occurs before 5 s.

Since this is such a simple example, it is just as easy to use `scipy.integrate.solve_bvp`. If you set the logical flag `BVP = True` in the code, you will do just that.



3.1.10.5 Streamlines

Ever wonder how the plots of electric field lines from static charges in first year undergraduate physics textbooks are made? Single point charges of either sign, dipoles and quadrapoles are common examples. A requirement of our education is that we can sketch these field lines but let's examine how to draw them exactly.

At first glance it would seem that Matplotlib has all the tools and we just need to call them. For example, `matplotlib.pyplot.quiver` is widely advertised as a method for viewing vector fields. While it gives us a visualization of a vector field, it is far from what we are looking for. We have better luck by using `matplotlib.pyplot.streamplot`. It can be made to give accurate fields but never in the way shown in textbooks. After much playing around with the settings, I gave up and decided to write my own streamplot code, by first understanding streamlines under-the-hood.

In visualizing vector fields, streamlines are a concept that is closely related to vector fields. Mathematically speaking streamlines are continuous lines whose tangent at each point is given by a vector field. Each line, and therefore also streamline, can be parameterized by some parameter t. A streamline $\mathbf{r}(t)$ satisfies the equation

$$\frac{\mathrm{d}\mathbf{r}(t)}{\mathrm{d}t} = g(t)\mathbf{E}(\mathbf{r}(t))\,, \tag{3.137}$$

where $\mathbf{E}(\mathbf{r}(t))$ is the vector field and $g(t)$ some scaling function. The scaling function is arbitrary but must not be zero. It basically determines how fast one moves along the streamline as a function of the parameter t. Convenient choices are

$$g(t) = \frac{1}{|\mathbf{E}(\mathbf{r}(t))|} \quad \text{or} \quad g(t) = 1\,. \tag{3.138}$$

In the code below, some charges are specified and then the total electric field is calculated by summing over the electric fields of the individual charges. As a bonus, I also calculate the much simpler electric potential—scalar field. I believe the code works for an arbitrary number and position of static point charges. The charge configuration is specified by the variable `charges`. Feel free to experiment and test your intuition of what the plot should look like.

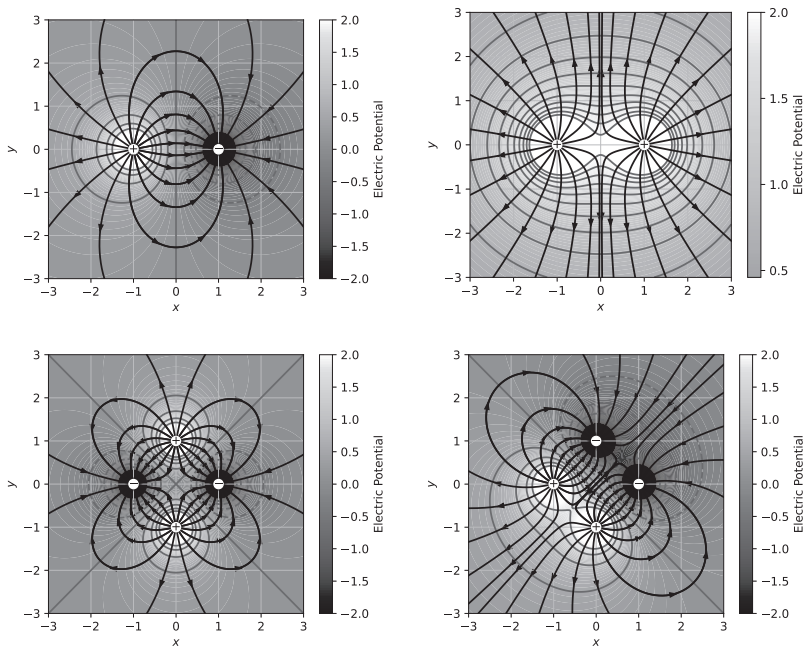
```
"""Make electric field and potential maps for point charges."""

import numpy              as np
from itertools            import product
from scipy.integrate      import ode
import matplotlib.pyplot as plt
from matplotlib           import cm

class charge:
    # Class specifying and storing point charge configuration.
    def __init__(self,q,pos):
      self.q   = q
      self.pos = pos

def E_point_charge(q,a,x,y):
    # Magnitude of electric field of one charge.
    return q * (x-a[0]) / ((x-a[0])**2 + (y-a[1])**2)**(3/2), \
           q  *(y-a[1]) / ((x-a[0])**2 + (y-a[1])**2)**(3/2)

def E_total(x,y,charges):
    # Component magnitude of total electric field from all
    charges.
    Ex, Ey = 0, 0
    for C in charges:
        E = E_point_charge(C.q,C.pos,x,y)
        Ex += E[0]
        Ey += E[1]
    return [Ex,Ey]

def E_dir(t,y,charges):
    # Direction of total electric field from all charges.
    # Function that will be integrated by odeint.
    Ex, Ey = E_total(y[0],y[1],charges)
    E = np.sqrt(Ex*Ex + Ey*Ey)
    return [Ex/E,Ey/E]

def V_point_charge(q,a,x,y):
    # Potential of one charge.
    return q / ((x-a[0])**2 + (y-a[1])**2)**(1/2)

def V_total(x,y,charges):
    # Total potential from all charges.
    V = 0
    for C in charges:
        Vp = V_point_charge(C.q,C.pos,x,y)
        V += Vp
    return V
```

```
# Create charge configuration.
# 1) Positive charge at origin
#charges = [charge( 1,[0,0])]
# 2) Negative charge at origin
#charges = [charge(-1,[0,0])]
# 3) Dipole
#charges = [charge(1,[-1,0]),charge(-1,[1,0])]
# 4) Two positive charges
#charges = [charge(1,[-1,0]),charge(1,[1,0])]
# 5) Quadruple
#charges = [charge(-1,[-1,0]),charge(-1,[1,0]),
#           charge( 1,[0,1]),charge(1,[0,-1])]
# 6) Four charges
charges = [charge( 1,[-1,0]),charge(-1,[1,0]),
           charge(-1,[0,1]),charge(1,[0,-1])]

# Grid size for calculation and plot.
x0, x1 = -3, 3
y0, y1 = -3, 3

# Calculate field lines
R = 0.008 # Minimum distance to charge (to avoid singularities).
Ex, Ey, Esign = [], [], []
for C in charges:
    # Plot field lines starting at current charge.
    dt = R
    if C.q < 0:
        dt = -dt
    # Loop over field lines starting in different
    # direction around current charge.
    Nlines = 16
    for alpha in np.linspace(0,2*np.pi*(Nlines-1)/Nlines,Nlines):
        # Use interface class to scipy.integrate.ode
        r = ode(E_dir)
        r.set_integrator('vode')
        r.set_f_params(charges)
        offset = 2*np.pi/(2*Nlines)
        x = [C.pos[0] + np.cos(alpha+offset)*R]
        y = [C.pos[1] + np.sin(alpha+offset)*R]
        r.set_initial_value([x[0],y[0]],0)
        while r.successful():
          r.integrate(r.t+dt)
          x.append(r.y[0])
          y.append(r.y[1])
          hit_charge = False
          # Check if field line left drawing area or
          # end at some charge.
          for C2 in charges:
              if np.sqrt((r.y[0]-C2.pos[0])**2
                         + (r.y[1]-C2.pos[1])**2) < R:
                  hit_charge = True
          if hit_charge or (not (x0 < r.y[0] and r.y[0] < x1)) \
            or (not (y0 < r.y[1] and r.y[1] < y1)): break
        Ex.append(x)
        Ey.append(y)
        Esign.append(C.q)
```

```
# Calculate electric potential.
potential = True
if potential:
    Vx = []
    Vy = []
    V  = []
    numcalcv = 300
    xrange = np.linspace(x0,x1,numcalcv)
    yrange = np.linspace(y0,y1,numcalcv)
    for x,y in product(xrange,yrange):
        Vx.append(x)
        Vy.append(y)
        V.append(V_total(x,y,charges))
    Vx = np.array(Vx)
    Vy = np.array(Vy)
    V  = np.array(V)

# Begin plotting (order of field lines, charge, potential
# is important to get overlay correct).
if potential:
    fig, ax = plt.subplots(figsize=(4.9,4))
else:
    fig = plt.figure(figsize=(3.89,4),facecolor="w")
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.xlim(x0,x1)
plt.ylim(y0,y1)
plt.grid()

# Plot electric potential.
if potential:
    clim0, clim1 = -2*0.9999, 2*0.9999 # 0.9999 to avoid edge
    effects
    V = np.clip(V,clim0,clim1)
    plt.tricontour(Vx,Vy,V,10,colors="0.3")
    plt.tricontourf(Vx,Vy,V,100,cmap=cm.jet)
    cbar = plt.colorbar()
    plt.clim(clim0,clim1)
    cbar.set_ticks([-2,-1.5,-1,-0.5,0,0.5,1,1.5,2])
    cbar.set_label("Electric Potential")

# Plot field line.
for x, y, sign in zip(Ex,Ey,Esign):
    plt.plot(x,y,color="k")
    mid = int(0.6*len(x))
    if sign > 0:
        plt.arrow(x[mid],y[mid],x[mid]-x[mid-1],y[mid]-y[mid-1],
                  head_width=0.1,color='black')
    if sign < 0:
        plt.arrow(x[mid-1],y[mid-1],
                  x[mid-1]-x[mid],y[mid-1]-y[mid],
                  head_width=0.1,color='black')

# Draw point charges.
for C in charges:
    if C.q > 0:
        plt.plot(C.pos[0],C.pos[1],'ow',ms=8*np.sqrt(C.q))
```

```
        plt.text(C.pos[0],C.pos[1],'+',
                 ha='center',va='center',fontsize='medium')
    if C.q < 0:
        plt.plot(C.pos[0],C.pos[1],'ow',ms=8*np.sqrt(-C.q))
        plt.text(C.pos[0],C.pos[1],'\N{MINUS SIGN}',
                 ha='center',va='center',fontsize='medium')

plt.gray()
#plt.savefig('3_1_streamlines_dipole.pdf',bbox_inches='tight',
#plt.savefig('3_1_streamlines_quad.pdf',bbox_inches='tight',
#plt.savefig('3_1_streamlines_2.pdf',bbox_inches='tight',
plt.savefig('3_1_streamlines_4.pdf',bbox_inches='tight',
            pad_inches=0.2,dpi=300)
plt.show()
```

Shown below are plots for the cases of a dipole (top left), two positive charges (top right), a quadrupole (bottom left) and two dipoles (bottom right).



3.2 PARTIAL DIFFERENTIAL EQUATIONS

Partial differential equations (PDE) occur in almost all fields of physics. They are continuous and analytic. The solutions are usually continuous and analytic. Numerical methods allow a richer set of PDEs to be solved over a wider class of initial conditions.

The general second order PDE in two independent variables is of the form

$$a\frac{\partial^2\phi}{\partial x^2} + b\frac{\partial^2\phi}{\partial x\partial y} + c\frac{\partial^2\phi}{\partial y^2} + d\frac{\partial\phi}{\partial x} + e\frac{\partial\phi}{\partial y} + f\phi(x,y) + g = 0\,. \tag{3.139}$$

The equation is classified as hyperbolic if $b^2-4ac > 0$, parabolic if $b^2-4ac = 0$ and elliptical if $b^2 - 4ac < 0$. In general, a, b, c, d, e, f, g are functions of (x, y) and the classification depends on the (x, y) coordinate considered; a given equation may be one type at one point and another type at another point.

We consider three types of PDEs:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 \qquad \text{Laplace equation (elliptical)}, \tag{3.140}$$

$$\frac{\partial^2\phi}{\partial x^2} = k\frac{\partial\phi}{\partial t} \qquad \text{diffusion equation (parabolic)}, \tag{3.141}$$

$$\frac{\partial^2\phi}{\partial x^2} = k\frac{\partial^2\phi}{\partial t^2} \qquad \text{wave equation (hyperbolic)}, \tag{3.142}$$

where k is a constant. These are some of the most common PDEs in physics. These equations are not restricted to a single space dimension—or two in the Laplace case—and a function of one (or two) variables may be added, or a constant, to give a nonhomogeneous equation. For example, adding a function to the Laplace equation gives

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = f(x,y) \quad \text{Poisson equation}\,. \tag{3.143}$$

Partial derivatives are functions of more than one variable and can be expanded in a multidimensional Taylor series. We consider the case when only one variable is changing at a time and the others are held fixed. This is equivalent to a Taylor series in one variable.

In this book, we only consider PDEs up to two variables in second order. Solutions to PDEs can be obtained by using finite differences. These are known as grid methods. Recall the central difference

$$\frac{\partial^2\phi(x,y)}{\partial x^2} = \frac{\phi(x_i+\Delta x, y_i) - 2\phi(x_i,y_i) + \phi(x_i-\Delta x, y_i)}{(\Delta x)^2}\,. \tag{3.144}$$

We simplify the notation using $x \to i$ and $y \to j$:

$$\frac{\partial^2\phi}{\partial x^2} = \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{(\Delta x)^2}\,. \tag{3.145}$$

Similarly for the forward difference

$$\frac{\partial\phi}{\partial y} = \frac{\phi_{i,j+1} - \phi_{i,j}}{\Delta y}\,. \tag{3.146}$$

These are the only two finite differences we will use when looking at the simplest cases. Using other finite difference are possible and could help improve the accuracy and stability.

The space variable(s) are typically bounded so the domain is closed. The time variable is usually in an open-ended domain. We now consider each of the three types of PDEs in turn.

3.2.1 Laplace equation

The Laplace equation is an example of an elliptical PDE, in two dimensions the equation is

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0\,. \tag{3.147}$$

We change $x \to i$ and $y \to j$ and consider a grid (i, j). Using central differences, the second order derivatives are

$$\frac{\partial^2\phi}{\partial x^2} = \frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{(\Delta x)^2} \quad \text{and} \quad \frac{\partial^2\phi}{\partial y^2} = \frac{\phi_{i,j-1} - 2\phi_{i,j} + \phi_{i,j+1}}{(\Delta y)^2}\,. \tag{3.148}$$

The Laplace equation written as a difference equation is

$$\frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{(\Delta x)^2} + \frac{\phi_{i,j-1} - 2\phi_{i,j} + \phi_{i,j+1}}{(\Delta y)^2} = 0\,. \tag{3.149}$$

Usually without loss of generality $\Delta x = \Delta y$ is taken so that it drops out on both sides of the equation. Solving for $\phi_{i,j}$ gives

$$\phi_{i,j} = \frac{1}{4}\left[\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1}\right]\,. \tag{3.150}$$

This solution is the average value of the four closest neighbours. The Laplace equation minimizes the curvature.

It can be illustrative to represent the difference equation in tabular form:

$j+1$		1	
j	1	−4	1
$j-1$		1	
	$i-1$	i	$i+1$

If we had not set $\Delta x = \Delta y$, the table would look like

$j+1$		$\frac{1}{(\Delta y)^2}$	
j	$\frac{1}{(\Delta x)^2}$	$-2\left(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2}\right)$	$\frac{1}{(\Delta x)^2}$
$j-1$		$\frac{1}{(\Delta y)^2}$	
	$i-1$	i	$i+1$

The geometry of the problem can lead to the Laplace equation in other coordinate systems. In polar coordinates

$$\nabla^2\psi = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial\psi}{\partial r}\right) + \frac{1}{r^2}\frac{\partial\psi^2}{\partial\theta^2} = 0\,. \tag{3.151}$$

Using $r_i = i\Delta r$ and $\theta_j = j\Delta\theta$, the tabular form of the finite difference equation becomes [12]

	$i-1$	i	$i+1$
$j+1$		$\frac{1}{(r_i\Delta\theta)^2}$	
j	$\left(1-\frac{1}{2i}\right)\frac{1}{(\Delta r)^2}$	$-2\left(\frac{1}{(\Delta r)^2}+\frac{1}{(r_i\Delta\theta)^2}\right)$	$\left(1+\frac{1}{2i}\right)\frac{1}{(\Delta r)^2}$
$j-1$		$\frac{1}{(r_i\Delta\theta)^2}$	

Returning to Cartesian coordinates, the grid can be visualized using Fig. 3.2. The square is the location being solved for and the circles are the locations contributing to the calculation.

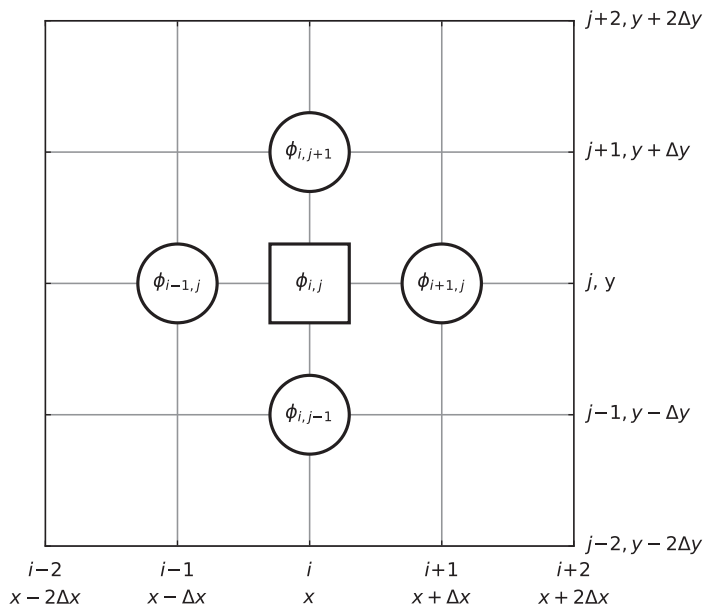


Figure 3.2 Grid for solution to the Laplace equation.

In general, solving the Laplace equation requires us to determine the simultaneous solution of a set of linear algebraic equations. We could use standard matrix methods to solve the system of equations. An example of the matrix method will appear in Problems 3.3.15. There are several other techniques that can be used to obtain the solution. We can use iterative methods of which the Jacobian method will be discussed below. The Gauss-Seidel iteration method is an improvement on the Jacobian method by using information in the iteration immediately as it becomes available, rather than waiting to use it in the next iteration. There are improved iteration methods such as over relaxation, and even non-grid methods such as Fourier methods and cyclic reduction methods.

First consider the Jacobian iterative approach. To solve the equation, we use a finite grid with values of $\phi_{i,j}$ specified at the boundaries. We call these boundaries the exterior region. In the interior region, we initially guess values of the function at all grid points. We iterate—replace $\phi_{i,j}$ by the average of its four neighbours—the equation until the function values stabilize. The view is that the solution flows inward from the boundaries. We want the steady-state solution that is independent of time—iteration step. Different boundary conditions are possible. For now, consider the case where ϕ is constant on all boundary points—the exterior region.

For stability, we will compare the new solution with the old solution until the change is small with each iteration. We could require stability at all points simultaneously or a sum of all points, or some other difference-minimization metric. We consider the later. At each grid point

$$\phi_{i,j}^{\text{new}} = \frac{1}{4}\left[\phi_{i+1,j}^{\text{old}} + \phi_{i-1,j}^{\text{old}} + \phi_{i,j+1}^{\text{old}} + \phi_{i,j-1}^{\text{old}}\right] . \tag{3.152}$$

The difference is

$$\begin{aligned}\Delta_{i,j} &= \phi_{i,j}^{\text{new}} - \phi_{i,j}^{\text{old}} \\ &= \frac{1}{4}\left[\phi_{i+1,j}^{\text{new}} + \phi_{i-1,j}^{\text{new}} + \phi_{i,j+1}^{\text{new}} + \phi_{i,j-1}^{\text{new}} - 4\phi_{i,j}^{\text{old}}\right] .\end{aligned} \tag{3.153}$$

Then we require $s \equiv \sum_{i,j} |\Delta_{i,j}| < \epsilon$, where ϵ is a small tolerance.

The Jacobian approach has the advantage of keeping the symmetry of the initial guess and the boundary conditions. It should always work. However, two arrays are needed for the old and new $\phi_{i,j}$. The Gauss-Seidel method usually converges in fewer iterations and only requires one array to store $\phi_{i,j}$. Although the Gauss-Seidel method does not keep the above mentioned symmetry; it usually doesn't matter. The new ϕ values at $(i+1)$ and $(j+1)$ can be used to calculate the (i,j) component of ϕ since they have already been determine at the (i,j) calculation as shown in Fig. 3.2. Now Eq. 3.150 becomes

$$\phi_{i,j}^{\text{new}} = \frac{1}{4}\left[\phi_{i-1,j}^{\text{new}} + \phi_{i+1,j}^{\text{old}} + \phi_{i,j-1}^{\text{new}} + \phi_{i,j+1}^{\text{old}}\right] . \tag{3.154}$$

Example 3.2.7.1 uses the Gauss-Seidel method.

A less intuitive algorithm is called successive over-relaxation (SOR) and can be written as

$$\phi_{i,j}^{\text{new}} = (1-\omega)\phi_{i,j}^{\text{old}} + \frac{\omega}{4}\left[\phi_{i-1,j}^{\text{new}} + \phi_{i+1,j}^{\text{old}} + \phi_{i,j-1}^{\text{new}} + \phi_{i,j+1}^{\text{old}}\right], \tag{3.155}$$

where ω is called the over-relaxation parameter. For $\omega = 1$, we have the Gauss-Seidel method. For $\omega < 1$, we have under-relaxation and the convergence is slow. For $\omega > 2$, the method is unstable That leaves us to select a best value in the range $1 < \omega < 2$. An adaptive program will adjust ω depending on how well the solution is converging.

3.2.1.1 Boundary conditions

In physics, we commonly encounter two types of boundary conditions, or a mixture of the two. Dirichlet boundary conditions specify values of the function (solution) at the boundaries. Neumann boundary conditions specify normal derivatives at the boundaries, which are often taken as zero. The value of zero derivative at the boundary is referred to as an insulated boundary condition.

For the case of insulating boundary conditions at the boarders of a rectangular grid:

$$\begin{aligned}
\phi_{i,j} &= \frac{1}{4}\left[\phi_{i,j+1} + \phi_{i,j-1} + 2\phi_{i+1,j}\right] \quad \text{left},\\
\phi_{i,j} &= \frac{1}{4}\left[\phi_{i,j+1} + \phi_{i,j-1} + 2\phi_{i-1,j}\right] \quad \text{right},\\
\phi_{i,j} &= \frac{1}{4}\left[\phi_{i-1,j} + \phi_{i+1,j} + 2\phi_{i,j-1}\right] \quad \text{upper},\\
\phi_{i,j} &= \frac{1}{4}\left[\phi_{i-1,j} + \phi_{i+1,j} + 2\phi_{i,j+1}\right] \quad \text{lower}.
\end{aligned} \tag{3.156}$$

3.2.1.2 Grid points near irregular boundaries

What do we do when the boundaries are not rectangular? If the values at the boundary are known, we can use linear interpolation for points less than a grid spacing from a boundary [13].

Considering Fig. 3.3, we can calculate the average

$$\phi_p = \left(\frac{\alpha}{1+\alpha}\right)\phi_A + \left(\frac{1}{1+\alpha}\right)\phi_C \tag{3.157}$$

and

$$\phi_p = \left(\frac{\beta}{1+\beta}\right)\phi_D + \left(\frac{1}{1+\beta}\right)\phi_D\,. \tag{3.158}$$

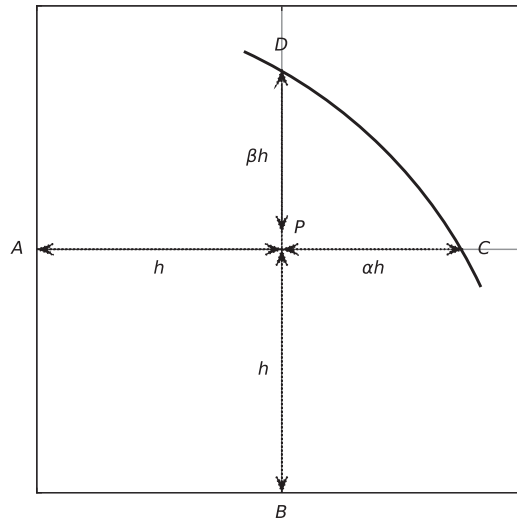


Figure 3.3 Non-grid boundary.

The case of normal derivatives at the boundary can also be handled. Alternatively, non-rectangular grid-coordinates could be used (eg. polar, conformal and so forth).

3.2.2 Diffusion equation

The diffusion equation is an example of a parabolic partial differential equation that can be written as

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2} \,. \tag{3.159}$$

Changing the variables $x \to i$ and $t \to j$, we write the differences

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{(\Delta x)^2} \qquad \text{central difference,} \tag{3.160}$$

$$\frac{\partial T}{\partial t} = \frac{T_{i,j+1} - T_{i,j}}{\Delta t} \qquad \text{forward difference.} \tag{3.161}$$

Using forward time and central space differences is known as the FTCS scheme. Substituting the finite differences into the diffusion equation gives

$$\frac{T_{i,j+1} - T_{i,j}}{\Delta t} = \kappa \frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{(\Delta x)^2} \,. \tag{3.162}$$

For convenience, let

$$\alpha = \kappa \frac{\Delta t}{(\Delta x)^2} \,. \tag{3.163}$$

Notice the position of κ is in correspondence with its position in the original diffusion equation, Eq. (3.159).

The discrete diffusion equation can be written as

$$T_{i,j+1} = \alpha(T_{i-1,j} + T_{i+1,j}) + (1 - 2\alpha)T_{i,j} \,. \tag{3.164}$$

The right-hand side is at single j (time). The left-hand side is at a future time. This case is referred to as an explicit method. The equation is self-starting. The grid for the solution of the diffusion equation is shown in Fig. 3.4. The solution at a point depends on the solution at three points in the previous times step.

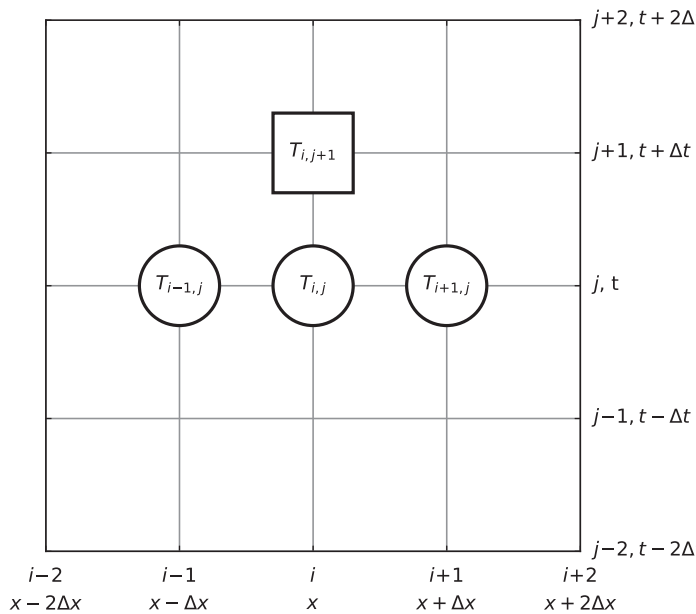


Figure 3.4 Grid for solution to the diffusion equation.

We need boundary conditions at the two ends $x = 0$ and $x = L$, say, for all time t. In addition, we need initial conditions at all positions at time $t = 0$. If an end is insulated, an imaginary column of grid points beyond the end can be used. In this case, to ensure the derivative is zero

$$\begin{aligned} \frac{T_{i,1} - T_{i,-1}}{2\Delta x} = 0 \Rightarrow T_{i,1} = T_{i,-1} \quad \text{or} \\ \frac{T_{i,N} - T_{i,N-2}}{2\Delta x} = 0 \Rightarrow T_{i,N} = T_{i,N-2} \,. \end{aligned} \tag{3.165}$$

The solution is stable if $\alpha \leq 1/2$. If we divide the length $x = [0, L]$ into a

grid of size Δx, then given $\kappa, \Delta x$ and $\alpha \leq 1/2$, we then take

$$\Delta t \leq \frac{(\Delta x)^2}{2\kappa} \tag{3.166}$$

for stability. The FTCS scheme is conditionally stable as will be demonstrated in Sec. 3.2.6. If we were to formulate a backward time and central space scheme (BTCS) it would be unconditionally stable. But then we would have the problem that it was not self-starting.

I suggest you do not store T_i in an array to avoid storing a large two-dimensional grid, unless you want to plot T_i as function of time.

3.2.3 Wave equation

The wave equation is an example of a hyperbolic PDE and can be written as

$$\frac{\partial^2 \psi}{\partial x^2} = k \frac{\partial^2 \psi}{\partial t^2}, \quad \text{where} \quad k = \frac{1}{c^2} \tag{3.167}$$

is related to the wave speed c. Substituting $x \to i$ and $t \to j$, the central differences are

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{\psi_{i-1,j} - 2\psi_{i,j} + \psi_{i+1,j}}{(\Delta x)^2}. \tag{3.168}$$

$$\frac{\partial^2 \phi}{\partial t^2} = \frac{\psi_{i,j-1} - 2\psi_{i,j} + \psi_{i,j+1}}{(\Delta t)^2}. \tag{3.169}$$

The discrete wave equation can be written as

$$\psi_{i,j+1} = \alpha(\psi_{i-1,j} + \psi_{i+1,j}) + 2(1-\alpha)\psi_{i,j} - \psi_{i,j-1}, \tag{3.170}$$

where

$$\alpha = \frac{1}{k}\left(\frac{\Delta t}{\Delta x}\right)^2. \tag{3.171}$$

The solution is stable if $\alpha \leq 1$ as will be shown in Sec. 3.2.6. The solution gets better for smaller time steps but worse for smaller grid spacing. The accuracy decreases when α is smaller, contrary to most people's intuition. The stability criterion can also be written as $c \leq \Delta x/\Delta t$ known as the Courant condition.

The grid for the solution of the wave equation is shown in Fig. 3.5. In this case, the solution at a point not only depends on the solution at three points in the previous times step, but the solution at a point two time steps back.

Notice that the right-hand side of Eq. (3.170) is not evaluated at a single j value. If we used forward differences for time, the right-hand side would be at a single time but the left-hand side would be at two times. In any case, FTCS schemes for the wave equation are unconditionally unstable. Because of the second order time derivatives, we cannot use Eq. (3.170) to start the solution. We need a separate algorithm to start the solution or values of $\psi_{i,0}$

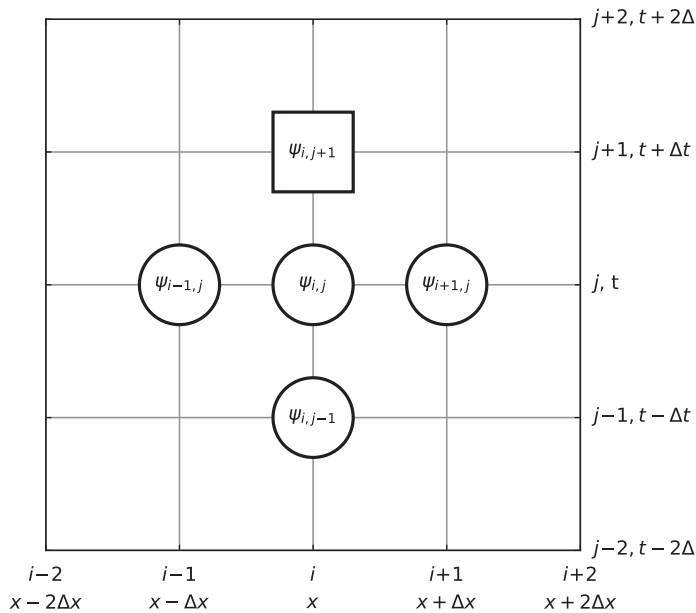


Figure 3.5 Grid for solution to the wave equation.

and $\psi_{i,1}$. Assume $d\psi_{0,1}/dt = 0$ at $t = 0$—or use an imaginary row—the central difference is

$$\psi'_{i,0} = \frac{\psi_{i,1} - \psi_{i,-1}}{2\Delta t} = 0 \quad \Rightarrow \quad \psi_{i,-1} = \psi_{i,1}\,. \tag{3.172}$$

Therefore

$$\psi_{i,1} = \frac{1}{2}\left[\alpha(\psi_{i-1,0} + \psi_{i+1,0}) + 2(1-\alpha)\psi_{i,0}\right]\,. \tag{3.173}$$

This is a starting formula that can be used once.

This completes our survey of numerical grid methods to solve the three most popular PDEs in physics. We now turn to some other PDEs in physics and engineering, and finely the PDE of quantum mechanics.

3.2.4 Continuity equation

The continuity equation appears in may fields of physics:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{j} = 0\,. \tag{3.174}$$

It describes how the time rate of change of a density ρ is related to the divergence of the flux $\mathbf{j}$ of the density. This is another example of a hyperbolic PDE.

Often $\mathbf{j} = \rho\mathbf{v}$, where $\mathbf{v}$ is the velocity describing the movement of ρ. If the quantity flowing is incompressible, $\nabla \cdot \mathbf{v} = 0$, and we can write

$$\frac{\partial \rho}{\partial t} + \mathbf{v} \cdot \nabla \rho = 0\,. \tag{3.175}$$

This is the linear convection equation or advection equation.

You probably think this is just a first-order version of the wave equation. However, there is one physical difference due to the order of the derivatives. Unlike the wave equation which has positive-moving and negative-moving waves, the advection equation only has positive-moving waves—if v is positive. To see this, consider the advection equation as an operator for positive-moving waves and another for negative-moving waves as $\partial/\partial t - \mathbf{v}\cdot\nabla$ and $\partial/\partial t - \mathbf{v}\cdot\nabla$, respectively. Using the operators to advect in both directions give

$$\begin{aligned}\left(\frac{\partial}{\partial t} - \mathbf{v}\cdot\nabla\right)\left(\frac{\partial}{\partial t} + \mathbf{v}\cdot\nabla\right)\psi &= \left(\frac{\partial^2}{\partial t^2} - (\mathbf{v}\cdot\nabla)^2\right)\psi \\ &= \left(\frac{\partial^2}{\partial t^2} - v^2\nabla^2\right)\psi = 0\,,\end{aligned} \tag{3.176}$$

where $\nabla\mathbf{v} = 0$ has been used and the final result is the wave equation.

3.2.4.1 Advection equation

The advection equation can be solved analytically but modifications to it lead to interesting solutions. Before tackling the interesting problems let's tune up on the advection equation. It is natural for us to try using forward-time differences and central-space differences, FTCS. In one dimension for a scalar function $u(x,t)$, discretizing $t = n\Delta t$ and $x = j\Delta x$ gives

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + v\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} = 0\,. \tag{3.177}$$

Arranging for future times on the left-hand side of the equation gives

$$u_j^{n+1} = u_j^n - \frac{v\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right)\,. \tag{3.178}$$

This difference equation is simple: its explicit, first-order accurate in time and second-order accurate in space. However, implementation of the FTCS scheme will fail; the waveform does not maintain its shape. The FTCS scheme is unstable for all values of Δt using this PDE, as we will see in the figures below.

The Lax-Fredrichs method fixes this instability by replacing the u_j^n term in Eq. (3.178) by the average value of its left and right neighbours:

$$u_j^{n+1} = \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) - \frac{v\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right)\,. \tag{3.179}$$

The Lax scheme is stable if $\alpha \equiv v\Delta t/\Delta x \leq 1$. This is called the Courant-Freiedrichs-Lewy (CFL) condition. It applies in general to numerical schemes that solve hyperbolic equations. One gets the best results for the advection equation when $\Delta t = \Delta x/v$. As in the wave equation, small time steps do not necessarily lead to better solutions.

We now discuss a scheme that is second-order in time. Consider the Taylor expansion:

$$u(x, t + \Delta t) = u(x, t) + \Delta t \frac{\partial u}{\partial t} + \frac{(\Delta t)^2}{2} \frac{\partial^2 u}{\partial t^2} + \mathcal{O}((\Delta t)^2)\,. \tag{3.180}$$

Using the advection equation (3.175) in one dimension to replace the derivatives in time with derivatives in space:

$$\begin{aligned} \frac{\partial u}{\partial t} &= -v \frac{\partial u}{\partial x} \\ \frac{\partial^2 u}{\partial t^2} &= -v \frac{\partial}{\partial t} \frac{\partial u}{\partial x} = -v \frac{\partial}{\partial x} \frac{\partial u}{\partial t} = v^2 \frac{\partial^2 u}{\partial x^2} \end{aligned} \tag{3.181}$$

leads to

$$u(x, t + \Delta t) = u(x, t) - v\Delta t \frac{\partial u}{\partial x} + \frac{(v\Delta t)^2}{2} \frac{\partial^2 u}{\partial x^2}\,. \tag{3.182}$$

The Lax-Wendroff discretizing scheme gives

$$u_j^{n+1} = u_j^n - \frac{v\Delta t}{2\Delta x} \left(u_{j+1}^n - u_{j-1}^2\right) + 2 \left(\frac{v\Delta t}{2\Delta x}\right)^2 \left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right)\,. \tag{3.183}$$

The result is equivalent to the two-step approach presented elsewhere [1, 12]. The CFL condition is also the condition for stability of the Lax-Wendroff scheme. The averaging in the Lax scheme introduces a numerical diffusion or viscosity. The amplitude of the wave decreases spuriously. If the time step is too large, the diffusion is too weak to stabilize the solution. If the time step is too small the diffusion is too strong and it damps out the true solution.

We see that for $\alpha = 1$, both the Lax and Lax-Wendroff scheme give $u_j^{n+1} = u_{j_1}^n$, which is a solution that maintains its waveform—is exact–and moves in increasing x one Δx step for each $v\Delta t$.

Figure 3.6 shows the solution to the advection equation using three different methods. Space has been discretized into 50 points. The initial waveform is a sine wave. The boundary conditions require the ends of the waveform to be zero. The CFL number is 0.1 and waveforms are shown after 200 time steps of $\Delta t = 0.001$. The FTCS scheme is unstable, showing oscillations at high x. However, the solution does maintain the original waveform shape at low x. The Lax scheme shows considerable amplitude damping but is stable. The Lax-Wendroff scheme is stable, maintains the shape and only has minor amplitude damping.

The above methods used for the advection equation apply equally to other hyperbolic PDEs—including nonlinear ones which we turn to next.

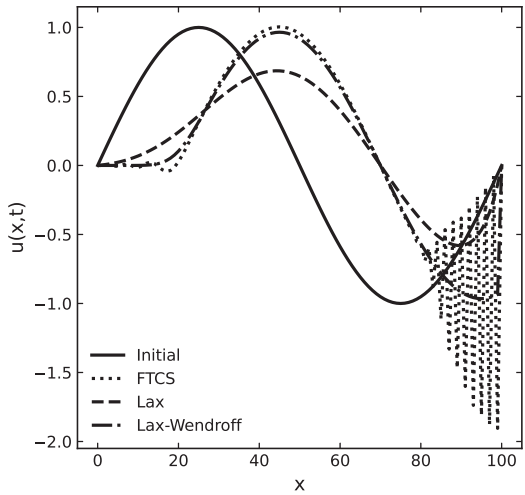


Figure 3.6 Solution to the advection equation using three different methods. Space has been discretized into 50 points. The initial waveform is a sine wave. The boundary conditions require the ends of the waveform to be zero. The CFL number is 0.1 and waveforms are shown after 200 time steps of $\Delta t = 0.001$.

3.2.4.2 Burger equation

The Bateman-Burger equation is popular in some branches of physics and engineering. Neglecting the diffusion term in the viscous Burgers equation give the inviscid Burger equation which can be written as

$$\frac{\partial u}{\partial t} = u\frac{\partial u}{\partial x} = \frac{\partial (u^2/2)}{\partial x} = 0\,. \tag{3.184}$$

The second version is the conservative form. This equation can be considered as a variation of the advection equation. The second term is nonlinear. In the advection equation, all points on the wave moved at the same speed and so the shape of the wave remains unchanged in time. For the Burger equation, the points on the wave move such that the local speed depends on the local wave amplitude; higher parts of the wave move faster than lower parts. This changes the shape of the wave with time. The high parts speed up and push their way to the front of the wave packet forming a sharp leading edge known as a shock wave. Shock waves are a form of nonlinear instability. They can be controlled by numerical viscosity.

We write down the difference equations in the previous three scheme using $t = i\Delta t$ and $x = j\Delta x$. The FTCS scheme is

$$u_{i+1,j} = u_{i,j} - \frac{\alpha}{2}\left(u^2_{i,j+1} - u^2_{i,j-1}\right) . \tag{3.185}$$

The superscript 2 is a square. The Lax scheme is

$$u_{i+1,j} = \frac{1}{2}\left(u_{i,j+1} + u_{i,j-1}\right) - \frac{\alpha}{2}\left(u^2_{i,j+1} - u^2_{i,j-1}\right) . \tag{3.186}$$

The Lax-Wendroff scheme is

$$u_{i+1,j} = u_{i,j} - \frac{\alpha}{4}\left(u^2_{i,j+1} - u^2_{i,j-1}\right) + \frac{\alpha}{8}\left[(u_{i,j+1} + u_{i,j})(u^2_{i,j+1} - u^2_{i,j}) - (u_{i,j} + u_{i,j-1})(u^2_{i,j} - u^2_{i,j-1})\right] , \tag{3.187}$$

where α is the CFL number. For stability $\alpha \leq 1$. The schemes are explicit and centred on the grid points. The second derivative term in the Lax-Wendroff scheme has an artificial diffusion that stabalizes the numerical solution.

Figure 3.7 shows the solution to the Burger equation using two different methods. Space has been discretized into 50 points. The initial waveform is a

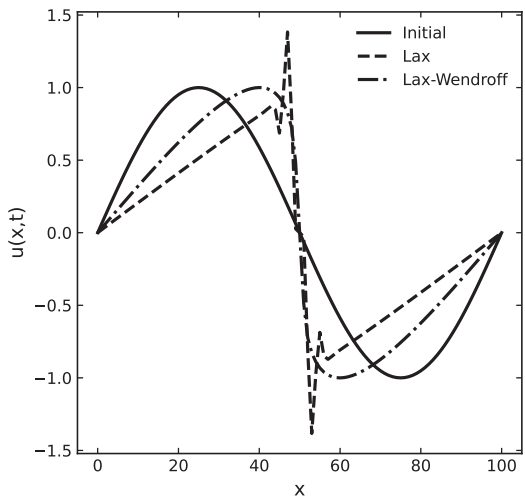


Figure 3.7 Solution to the Burger equation using two different methods. Space has been discretized into 50 points. The initial waveform is a sine wave. The boundary conditions require the ends of the waveform to be zero. The CFL number is 1 and waveforms are shown after 15 time steps of $\Delta t = 0.15$.

sine wave. The boundary conditions require the ends of the waveform to be zero. The CFL number is 1 and waveforms are shown after 15 time steps of $\Delta t = 0.15$. The FTCS scheme is not shown as its oscillations dominate the plot and we are already told that the scheme is unstable. The shock wave is apparent for the Lax and Lax-Wendroff schemes. Instabilities are already apparent in the Lax scheme. The Lax-Wendroff scheme is maintaining the amplitude while still show a shock wave. This example shows that the type of Lax scheme matters.

Advecting the wave further in time will cause the amplitude in the Lax scheme to vanish and oscillates at the shock-wave front will form in the Lax-Wendroff scheme. If more accuracy is need, one might consider staggered leapfrog methods or unwinding differencing [1].

3.2.5 Schrödinger equation

At first glance the time-dependent Schrödinger equation may appear to be a diffusion equation but with the same sign between the two terms. However, there is an imaginary factor i in the equation and the wavefunction ψ is complex. The physics also requires the wavefunction to remain unitary. Which means the modulus-square norm of ψ remains less than unity. This requires the time-evolution operator of the wavefunction to be Hermitian. Because of these differences from the diffusion equation a special treatment of the Schrödinger equation is needed.

The one-dimensional Schrödinger equation for a particle of mass m is

$$i\hbar\frac{\partial\psi}{\partial t} = \mathcal{H}\psi\,, \tag{3.188}$$

where the Hamiltonian operator is

$$\mathcal{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x)\,. \tag{3.189}$$

We represent the discretized wavefunction as $\psi_j^n = \psi(x_j, t_n)$. It is tempting to use the forward difference for the time derivative and central difference for the space derivative:

$$i\hbar\frac{\psi_j^{n+1} - \psi_j^n}{\Delta t} = -\frac{\hbar^2}{2m}\frac{\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n}{(\Delta x)^2} + V_j\psi_j^n\,. \tag{3.190}$$

This is an explicit FTCS scheme which is conditionally stable. Unfortunately, it is not unitary, as we will see.

The difference equation can be written as

$$i\hbar\frac{\psi_j^{n+1} - \psi_j^n}{\Delta t} = \sum_{k=0}^{N-1} H_{jk}\psi_k^n\,, \tag{3.191}$$

where

$$H_{jk} = -\frac{\hbar^2}{2m}\frac{\Delta_{j+1,k} - 2\Delta_{j,k} + \Delta_{j-1,k}}{(\Delta x)^2} + V_j\Delta_{jk}\,. \tag{3.192}$$

In matrix notation,

$$\Psi^{n+1} = \left(\mathbf{I} - \frac{i\Delta t}{\hbar}\mathbf{H}\right)\Psi^n\,. \tag{3.193}$$

We see that the time-evolution operator is not unitary. This equation looks to be first order in Δt of the continuous time evolution operator $\exp(-it\mathcal{H}/\hbar)$.

Applying the discrete Hamiltonian to the future wavefunction ψ^{n+1} give

$$\Psi^{n+1} = \left(\mathbf{I} + \frac{i\Delta t}{\hbar}\mathbf{H}\right)^{-1}\Psi^n\,. \tag{3.194}$$

This is an implicit FTCS scheme. It is unconditionally stable but again not unitary. Neither operator is unitary and they both appear to be first order in some Taylor series expansion. The explicit and implicit schemes are equivalent as $\Delta t \to 0$.

Taking the average between the explicit and implicit FTCS schemes gives

$$i\hbar\frac{\Psi^{n+1} - \Psi^n}{\Delta t} = \frac{\mathbf{H}\Psi^{n+1} + \mathbf{H}\Psi^n}{2}\,, \tag{3.195}$$

resulting in

$$\left(\mathbf{I} + \frac{i\Delta t}{2\hbar}\mathbf{H}\right)\Psi^{n+1} = \left(\mathbf{I} - \frac{i\Delta t}{2\hbar}\mathbf{H}\right)\Psi^n \tag{3.196}$$

or

$$\Psi^{n+1} = \left(\mathbf{I} + \frac{i\Delta t}{2\hbar}\mathbf{H}\right)^{-1}\left(\mathbf{I} - \frac{i\Delta t}{2\hbar}\mathbf{H}\right)\Psi^n\,. \tag{3.197}$$

This method is unconditionally stable and second-order accurate in space and time. What's more, the time-evolution operator is now unitary. It can be shown (see Sec. 3.2.6) if the two methods are added together with different weights, the minimum degree of implicitness that guarantees stability for all sizes of the time step is equal weighting. This is essential a Crank-Nicolson method. Since the method is still implicit we can solve it using matrix methods. The problem becomes a complex tridiagonal system, which can be represented as a sparse matrix.

3.2.6 von Neumann stability analysis

It is important to know if the method of solution to a PDE will result in a stable solution, that is, one that does not grow exponentially with time. Stability depends on both the type of PDE and the method of solution for

that PDE. For finite difference schemes, it is usual to apply a von Neumann stability analysis to the scheme being used to solve the desired PDE.

The von Neumann method recognizes that the spatial variation of the solution ψ to a PDE at any time can be represented by a Fourier series $\psi(x,t) = \sum_k A_k(t)e^{ikx}$, where k is a real spatial wave number in the x direction. Since the equation is linear, we can consider each term in the Fourier series separately, then add them later. The von Neumann stability analysis turns a PDE into an algebraic equation by examining separately all the Fourier components of the spatial variation. The independent solutions, or eigenmodes, of the difference equations with discretization $x = j\Delta x$ and $t = n\Delta t$ are of the form

$$\psi(x_j, t_n) = \psi_j^n = \xi(k)^n e^{ikj\Delta x}\,, \tag{3.198}$$

where ξ can be complex in general. We view ψ_j^n as a function that oscillates in space (the exponential) with the amplitude or amplification factor $\xi(k)$ that gets multiplied by the power of ξ for each time step; the time dependence of a single eigenmode is a successive integer power of the complex number ξ. The difference equations are unstable—have exponentially growing modes—if $|\xi(k)| > 1$ for any k.

The method of the stability analysis is to insert the trial solution Eq. (3.198) into the numerical scheme and solve for the amplification factor in terms of the grid spacing, time step and any parameters of the PDE. Technically the method only applies to equally spaced spatial grids and PDEs with constant coefficient, but is consider to be approximately correct if these two criteria are relaxed [1].

We will now perform von Neumann stability analysis on the different schemes used to solve the previous PDEs, thus justifying stability statements made in Sec. 3.2.2, 3.2.3 and 3.2.5.

3.2.6.1 Diffusion equation stability

First, consider the diffusion equation:

$$\frac{\partial \psi}{\partial t} = D\frac{\partial \psi}{\partial x}\,. \tag{3.199}$$

Represented as the FTCS scheme,

$$\psi_j^{n+1} = \psi_j^n + \frac{D\Delta t}{\Delta x^2}\left(\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n\right)\,. \tag{3.200}$$

Applying the von Neumann stability analysis, we can write

$$\xi = 1 + \frac{D\Delta t}{\Delta x^2}\left(e^{ik\Delta x} - 2 + e^{-ik\Delta x}\right) = 1 - 4\frac{D\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)\,. \tag{3.201}$$

For stability, we require $|\xi| \leq 1$ for all k and thus

$$-1 \leq 1 - 4\frac{D\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right) \leq 1\,,$$
$$0 \leq 4\frac{D\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right) \leq 2\,. \tag{3.202}$$

If D is positive and constant, or approximately constant locally, the lower inequality is satisfied and the remaining conditional inequality is

$$\frac{D\Delta t}{\Delta x^2} \leq \frac{1}{2}\,. \tag{3.203}$$

This is the conditional stability requirement of the diffusion equation stated in Sec. 3.2.2.

Now consider the BTCS scheme,

$$\psi_j^n = \psi_j^{n+1} + \frac{D\Delta t}{\Delta x^2}\left(\psi_{j+1}^{n+1} - 2\psi_j^{n+1} + \psi_{j-1}^{n+1}\right)\,. \tag{3.204}$$

Stability analysis gives

$$\xi = 1 - 4\xi\frac{D\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)\,,$$
$$1 = \xi\left[1 + 4\xi\frac{D\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)\right]\,. \tag{3.205}$$

Stability requires $|\xi| \leq 1$ and thus

$$1 + \frac{4D\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right) \geq 1\,, \tag{3.206}$$

which is unconditionally satisfied. This is the unconditional stability requirement of the diffusion equation stated in Sec. 3.2.2.

3.2.6.2 *Wave equation stability*

Next, let's consider the wave equation

$$\frac{\partial^2\psi}{\partial t^2} = c^2\frac{\partial^2\psi}{\partial x^2}\,. \tag{3.207}$$

Discretization by central differences gives

$$\psi_j^{n+1} - 2\psi_j^n + \psi_j^{n-1} = \frac{c^2\Delta t^2}{\Delta x^2}\left(\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n\right)\,. \tag{3.208}$$

Stability analysis gives the quadratic equation

$$\xi - 2 + \xi^{-1} = -4\frac{c^2\Delta t^2}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)\,, \tag{3.209}$$

which can be written as

$$\xi^2 - 2b + 1 = 0\,, \quad \text{where} \quad b = 1 - 2\frac{c^2\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)\,. \tag{3.210}$$

The solutions are given by

$$\xi = b \pm \sqrt{b^2 - 1}\,. \tag{3.211}$$

If the two roots are ξ_1 and ξ_2, we can write

$$(\xi - \xi_1)(\xi - \xi_2) = \xi^2 - \xi(\xi_1 + \xi_2) + \xi_1\xi_2 = 0\,, \tag{3.212}$$

and notice that $\xi_1\xi_2 = 1$ and $\xi_1 + \xi_2 = 2b$. Requiring $|\xi_1| \leq 1$ and $|\xi_2| \leq 1$ for stability, gives $|\xi_1| = |\xi_2| = 1$. The inequalities can be written as $-1 \leq b \leq 1$ giving

$$\begin{aligned} -1 &\leq 1 - 2\frac{c^2\Delta t^2}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right) \leq 1\,, \\ 0 &\leq \frac{c^2\Delta t^2}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right) \leq 1\,. \end{aligned} \tag{3.213}$$

Since all the quantities are positive, the lower inequality is satisfied and only the upper inequality is conditional:

$$\frac{c^2\Delta t^2}{\Delta x^2} \leq 1\,. \tag{3.214}$$

This is the conditional stability requirement of the wave equation stated in Sec. 3.2.3.

3.2.6.3 Advection equation stability

The FTCS scheme for the advection equation was stated to be unconditionally unstable. Applying the stability analysis gives

$$\xi = 1 - \frac{v\Delta t}{2\Delta x}\left(e^{ik\Delta x} - e^{-ik\Delta x}\right) = 1 - i\frac{v\Delta t}{\Delta x}\sin(k\Delta x)\,. \tag{3.215}$$

The magnitude of the amplification factor is

$$|\xi| = \sqrt{1 + \left(\frac{v\Delta t}{\Delta x}\right)^2 \sin^2(k\Delta x)}\,. \tag{3.216}$$

The magnitude is always great than unity. The solution is unstable since the amplitude grows with each time step.

In the Lax-Friedrichs scheme the stability analysis gives

$$\begin{aligned} \xi &= \frac{1}{2}\left(e^{ik\Delta x} + e^{-ik\Delta x}\right) - \frac{v\Delta t}{2\Delta x}\left(e^{ik\Delta x} - e^{-ik\Delta x}\right)\,, \\ &= 1 + \cos(k\Delta x) - i\frac{v\Delta t}{\Delta x}\sin(k\Delta x)\,. \end{aligned} \tag{3.217}$$

The magnitude of the amplification factor is

$$|\xi| = \sqrt{\cos^2(k\Delta x) + \left(\frac{v\Delta t}{\Delta x}\right)^2 \sin^2(k\Delta x)}\,. \tag{3.218}$$

The stability criterion is satisfied if and only if $|c\Delta t/\Delta x| \leq 1$ This is the Courant-Friedrichs-Lewy stability criterion.

3.2.6.4 Crank-Nicolson method stability

Finally, consider the Crank-Nicolson method for solving the Schrödinger equation. The discretized explicit and implicit scheme (θ-implicit scheme) to solve the Schrödinger equation in Sec. 3.2.5 is

$$\begin{aligned}\psi_j^{n+1} - \psi_j^n = \frac{i\hbar\Delta t}{2m\Delta x^2}\left[\theta\left(\psi_{j+1}^{n+1} - 2\psi_j^{n+1} + \psi_{j-1}^{n+1}\right)\right.\\ \left. + (1-\theta)\left(\psi_{j+1}^{n} - 2\psi_j^{n} + \psi_{j-1}^{n}\right)\right]\,,\end{aligned} \tag{3.219}$$

where θ is the weight of implicitness. Applying the stability analysis and performing the previous trigonometric simplifications leads to the solution

$$\xi = \frac{1-(1-\theta)\alpha}{1+\alpha}\,, \tag{3.220}$$

where $\alpha = 4i\hbar\Delta t/(2m\Delta x^2)\sin^2(k\Delta x/2)$ is imaginary. The modules-squared of the amplification factor is require to satisfy

$$\frac{1+(1-\theta)^2|\alpha|^2}{1+|\alpha|^2} \leq 1\,, \tag{3.221}$$

which leads to $\theta \leq 1/2$ unconditionally independent of α. Thus $\theta = 1/2$ is the maximum amount of implicitness as mentioned in Sec. 3.2.5 and gives the Crank-Nicolson method, which is also required by unitarity. Note when the above θ-scheme is applied to the diffusion equation, the equivalent of α is real leading to a conditional stability criteria for $\theta < 1/2$ and an unconditional stability criteria for $\theta \geq 1/2$

3.2.7 Examples: Partial differential equations

3.2.7.1 Laplace equation

We will solve the Laplace equation in two dimensions:

$$\frac{\partial^2\phi(x,y)}{\partial x^2} + \frac{\partial^2\phi(x,y)}{\partial y^2} = 0\,. \tag{3.222}$$

Consider an 8×8 grid of (x,y) points. Take fixed boundary conditions of 100 at each grid point on the top and both sides of the grid, and fixed values

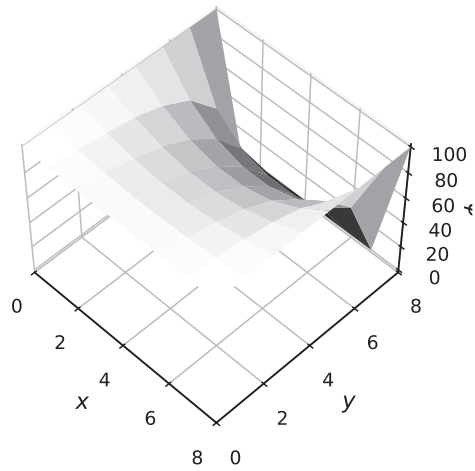
of zero at each grid point at the bottom of the grid. We'll use the iterative Gauss-Seidel method and require a tolerance of 0.1.

Consider the following code. I explicitly type into an array an initial starting grid of $\phi_{i,j}$ values. This is for pedagogical purposes to allow easy visualization of the initial grid. Normally one would fill the grid of initial values using assignment statements according to some simple algorithm. Also notice the use of the `while` statement. One should be cautious against possible infinite loops. I include a bail-out statement `iteration < giveUp` to prevent an infinite loop, and print out the cause of leaving the `while` statement so we know if the program finished successfully.

```
"""Solve the Laplace equation using the Jacobian method."""

import numpy as np
from matplotlib import pyplot as plt
np.set_printoptions(precision=0,suppress=True)

plt.style.use('./mystyle.mplstyle')

debug = False

# The function phi is modeled by a two-dimensional array
# of grid points: phi[0:7,0:7].
# x increases left to right; y increases top to bottom.
# The boundary values are fixed and the internal values
# are initial guesses.
phi = np.array([
    [100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0],
    [100.0, 90.0, 90.0, 90.0, 90.0, 90.0, 90.0,100.0],
    [100.0, 80.0, 80.0, 75.0, 75.0, 80.0, 80.0,100.0],
    [100.0, 75.0, 69.0, 50.0, 50.0, 60.0, 75.0,100.0],
    [100.0, 60.0, 50.0, 30.0, 30.0, 50.0, 60.0,100.0],
    [100.0, 50.0, 30.0, 15.0, 15.0, 30.0, 50.0,100.0],
    [100.0, 40.0, 20.0,  5.0,  5.0, 20.0, 40.0,100.0],
    [100.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,100.0]
])

# Decide when to stop.
iteration = 0
giveUp = 1000
tolerance = 0.1
print("Tolerance =",tolerance)
print("Will give up after",giveUp,
      "iterations if tolerance not achieved.")
print()

# Loop until tolerance is satisfied.
sum = np.inf
while (sum > tolerance) and (iteration < giveUp):
    # Iterate over grid points not on bounaries.
    sum = 0
    for i in range(1,7): # Do not loop over boundaries
```

```
        for j in range(1,7): # Do not loop over boundaries.
            # Calculate difference of new and old phi[i,j]
            d = (phi[i+1,j] + phi[i-1,j] + phi[i,j+1] \
                 +  phi[i,j-1] - 4.0*phi[i,j]) / 4
            phi[i,j] += d # Update phi[i,j]
            sum += abs(d) # Use sum of differences to determine
    tolerance.
    iteration += 1
    if debug: print(sum)

# Print out results.
if iteration == giveUp:
    print("Failed to converge after",iteration,"iterations.")
else:
    print("Convergence achieved after",iteration,"iterations")
print()
print("(x,y) grid of phi values.")
print(phi)

# Make contour plot.
x = np.linspace(0,8,8)
y = np.linspace(0,8,8)
X, Y = np.meshgrid(x,y)
fig = plt.figure(figsize=(4,4))
ax = plt.axes(projection='3d')
ax.plot_surface(X,Y,phi,cmap='gray')
ax.view_init(50,-45) # Rotates plot (elev,azim).
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$y$")
ax.set_zlabel(r"$\phi$",labelpad=1)
ax.xaxis.pane.fill = False
ax.yaxis.pane.fill = False
ax.zaxis.pane.fill = False
plt.xlim([0,8])
plt.ylim([0,8])
plt.savefig('3_2_laplace.pdf')
plt.show()
```

```
Tolerance = 0.1
Will give up after 1000 iterations if tolerance not achieved.

Convergence achieved after 37 iterations

(x,y) grid of phi values.
[[100. 100. 100. 100. 100. 100. 100. 100.]
 [100.  98.  96.  95.  95.  96.  98. 100.]
 [100.  95.  91.  89.  89.  91.  95. 100.]
 [100.  91.  84.  81.  81.  84.  91. 100.]
 [100.  85.  74.  69.  69.  74.  85. 100.]
 [100.  74.  59.  52.  52.  59.  74. 100.]
 [100.  52.  35.  29.  29.  35.  52. 100.]
 [100.   0.   0.   0.   0.   0.   0. 100.]]
```



Only 37 iterations were required to get a believable and clear plot. The initial values at the grid points were probably reasonable choices.

3.2.7.2 Diffusion equation

Heat flow with temperature T along the x direction at time t is given by

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2} \,, \tag{3.223}$$

where κ is the thermal diffusivity of the medium. We'll take one end of the material $x = 0$ to be held at $T(x = 0) = 100$ and the other end $x = L$ to be held at $T(x = L) = 0$.
Let $x \to i$ and $t \to j$;

$$T_{i,j+1} = \alpha(T_{i-1,j} + T_{i+1,j}) + (1 - 2\alpha)T_{i,j} \,, \tag{3.224}$$

where $\alpha = \kappa \Delta t/(\Delta x)^2$. Take $\kappa = 1.14$, boundary conditions $T_{0,j} = 100$, $T_{N-1,j} = 0$, and initial conditions $T_{i,0} = (100, 0, \ldots, 0)$.

The following code is self-explanatory except for perhaps the output. Since we are usually only interested in the steady-state solution, I simply print out the temperatures at each time step. One is then meant to look at the temperatures and notice when they stop changing. When this occurs, the solution has been found; usually the time at which the steady-state solution is reached is not important. Then for purposes of this book, I delete intermediate values of the output to keep the presentation of the output a manageable length. This approach is fine for one-off applications but is totally unsatisfactory for serious coding. One should state a convergence criteria—similar to what was done in

the solution to the Laplace equation—and print nothing, or only information messages, until convergence is reached.

```
"""Solve diffusion equation."""

import numpy as np
np.set_printoptions(formatter={'float': '{:6.3f}'.format})

# Parameters of problem.
kappa = 1.14
dt = 0.2
dx = 1.0
print("kappa",kappa,"dt",dt,"dx",dx)
alpha = kappa*dt/(dx*dx)
if (alpha > 0.5):
    print("Unstable value of alpha %.3f" % alpha)
else:
    print("Stable value of alpha %.3f" % alpha)
print()

# Boundary condition and initial conditions.
gridSize = 8
f = np.array([100.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0])
v = f # Do not save result in 2D array: v is new and f is old.
print(f)

# Loop over time.
sec = int(1.0/dt)
for _ in range(60*sec):
    # Loop over space.
    for i in range(1,gridSize): # Step over interior of grid.
        v[i] = alpha * (f[i-1] + f[i+1]) \
        + (1.0 - 2.0*alpha) * f[i]
    f = v
    print(f)
```

```
kappa 1.14 dt 0.2 dx 1.0
Stable value of alpha 0.2280

[100.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000]
[100.000 22.800  5.198  1.185  0.270  0.062  0.014  0.003  0.000]
[100.000 36.388 11.395  3.304  0.914  0.245  0.064  0.016  0.000]
[100.000 45.193 17.256  5.940  1.908  0.583  0.172  0.048  0.000]
[100.000 51.320 22.443  8.784  3.173  1.080  0.351  0.106  0.000]
[100.000 55.835 26.942 11.645  4.627  1.722  0.608  0.196  0.000]
[100.000 59.317 30.836 14.420  6.198  2.489  0.943  0.322  0.000]
[100.000 62.099 34.221 17.060  7.829  3.354  1.351  0.483  0.000]
[100.000 64.384 37.185 19.544  9.479  4.294  1.824  0.679  0.000]
[100.000 66.303 39.802 21.868 11.122  5.287  2.352  0.906  0.000]
[100.000 67.944 42.129 24.037 12.736  6.317  2.926  1.160  0.000]
[100.000 69.367 44.215 26.061 14.311  7.366  3.536  1.437  0.000]
```

```
[100.000 70.617 46.095 27.950 15.837  8.424  4.172  1.733  0.000]
[100.000 71.725 47.802 29.714 17.311  9.481  4.826  2.043  0.000]
[100.000 72.717 49.359 31.365 18.730 10.528  5.492  2.364  0.000]
[100.000 73.612 50.786 32.912 20.094 11.561  6.162  2.691  0.000]
[100.000 74.424 52.100 34.365 21.402 12.574  6.833  3.022  0.000]
[100.000 75.166 53.315 35.730 22.656 13.564  7.498  3.353  0.000]
[100.000 75.846 54.443 37.016 23.857 14.528  8.156  3.684  0.000]

[100.000 87.499 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.499 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.499 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 24.999 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 25.000 12.500  0.000]
[100.000 87.500 74.999 62.499 49.999 37.499 25.000 12.500  0.000]
[100.000 87.500 75.000 62.499 49.999 37.499 25.000 12.500  0.000]
[100.000 87.500 75.000 62.499 49.999 37.499 25.000 12.500  0.000]
[100.000 87.500 75.000 62.499 49.999 37.499 25.000 12.500  0.000]
[100.000 87.500 75.000 62.499 49.999 37.499 25.000 12.500  0.000]
[100.000 87.500 75.000 62.499 49.999 37.500 25.000 12.500  0.000]
[100.000 87.500 75.000 62.500 49.999 37.500 25.000 12.500  0.000]
[100.000 87.500 75.000 62.500 50.000 37.500 25.000 12.500  0.000]
[100.000 87.500 75.000 62.500 50.000 37.500 25.000 12.500  0.000]
```

3.2.7.3 Wave equation

The wave equation for a string where y is the transverse displacement and x is the distance along the string is

$$\frac{\partial^2 y}{\partial x^2} = \frac{1}{v^2}\frac{\partial^2 y}{\partial t^2}, \tag{3.225}$$

where $v = \sqrt{T/\rho}$ is a constant consisting of the linear mass density ρ and the tension T. Take $T = 10^4$ and $\rho = 1$, with $0 \le x \le 9$.
The starting formula is ($x \to i$ and $t \to j$)

$$y_{i,2} = \frac{1}{2}\left[\alpha\left(y_{i-1,1} + y_{i+1,1}\right) + 2(1-\alpha)y_{i,1}\right]. \tag{3.226}$$

The main formula is ($x \to i$ and $t \to j$)

$$y_{i,j+1} = \alpha\left(y_{i-1,j} + y_{i+1,j}\right) + 2(1-\alpha)y_{i,j} - y_{i,j-1}\,, \tag{3.227}$$

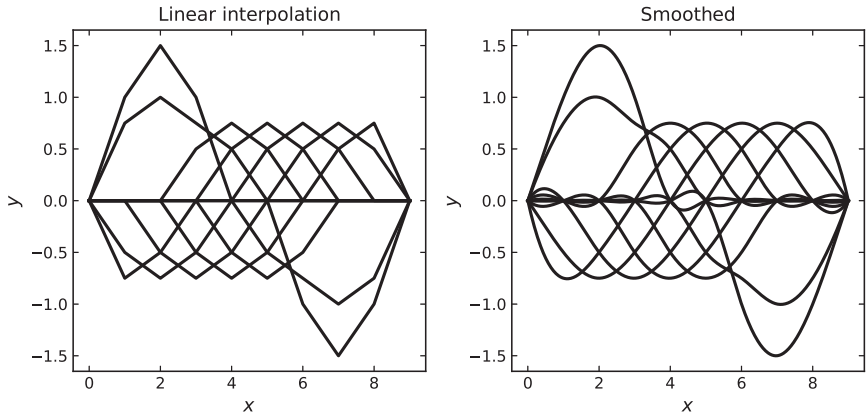
where $\alpha = v^2\left(\Delta t/\Delta x\right)^2$. The solution is stable if $\alpha \leq 1$. Take $\Delta x = 1$ and $\Delta t = 0.01$. We will give the string an initial triangle pulse over three points centred on the second grid point beyond the fixed end at $x = 0$.

Consider the following code. One can experiment with the time `dt` step and total time `tmax`. For `dt = 1`, I have picked values for `tmax` so that the waveform does not repeat, has exactly one repeat or lots of repeats. I used these values and my expectations to validate my code.

```
"""Solve the wave equation in one dimension."""

import numpy as np
from scipy.interpolate import make_interp_spline
from matplotlib import pyplot as plt

plt.style.use('./mystyle.mplstyle')

verbose = False # Lots of output.
debug   = False # A little bit more output.
T = 1e4
rho = 1
v = np.sqrt(T/rho)
dt = 0.01
grid = 9 # maximum internal grid point
#dt = 0.001 # interesting case for energy
dx = 1 # fixed
tmax = 9 # No repeats if dt = 0.01.
#tmax = 10 # 1 repeat.
#tmax = 100 # Lots of repeat, also good for energy
print("T",T,"rho",rho,"v",v,"dx",dx,"dt",dt)
alpha = (v*dt/dx)**2
if alpha > 1:
    print("unstable alpha =",alpha)
else:
    print("stable alpha =",alpha)

# Initially zero the grid. Set the endpoints to 0.
y = np.zeros([10,tmax+1])

# Start with a tringle-like pulse over 3 points.
y[0,0] = 0.0
y[1,0] = 1.0
y[2,0] = 1.5
y[3,0] = 1.0
y[4,0] = 0.0
y[5,0] = 0.0
y[6,0] = 0.0
y[7,0] = 0.0
y[8,0] = 0.0
y[9,0] = 0.0
```

```
# Starting equation.
for i in range(1,grid):
    y[i,1] = 0.5 * (alpha * (y[i-1,0] + y[i+1,0])
             + 2.0*(1.0-alpha) * y[i,0])

# Main equations.
for j in range(1,tmax):
    for i in range(1,grid):
        y[i,j+1] = alpha * (y[i-1,j] + y[i+1,j]) \
                   + 2.0*(1.0-alpha) * y[i,j] - y[i,j-1]

# Print results.
if (verbose):
    print()
    for j in range(tmax+1):
        print("At",j*dt,"sec, the displacement is",end=" ")
        for i in range(grid+1):
            print(y[i,j],end=" ")
        print()

# Swap time and space coordinates for plotting.
w = np.swapaxes(y,0,1)
x = np.arange(0,grid+1)
if (debug):
    print(x)
    print(w[0])

plt.figure(figsize=(8,4))

# Linear interpolation.
plt.subplot(1,2,1)
for j in range(tmax+1):
    plt.plot(x,w[j],label='{}'.format(j))
plt.title("Linear interpolation")
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")

# Smooth plot.
plt.subplot(1,2,2)
xs = np.linspace(0,grid,500) # Need more points for smooth plot.
for j in range(tmax+1):
    model = make_interp_spline(x,w[j])
    ys = model(xs)
    plt.plot(xs,ys,label='{}'.format(j))
plt.title("Smoothed")
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")

plt.tight_layout()
plt.savefig('3_2_wave_1.pdf')
plt.show()
```

```
T 10000.0 rho 1 v 100.0 dx 1 dt 0.01
stable alpha = 1.0
```



The left-hand plot shows the string profile at discrete times. The graphics uses linear interpolation. The right-hand plot shows the same information with interpolation using cubic splines.

If the program is run for longer time durations, the wave appears to return to its original position, as it should for a conserved system. In addition, we will use conservation of energy to validate the results. We can calculate the period, and kinetic and potential energy analytically.

Period: Since

$$v = \sqrt{\frac{T}{\rho}} = \sqrt{\frac{10^4}{1}} = 100\,, \tag{3.228}$$

the period is

$$\tau = \frac{L}{v} = \frac{9}{100} = 0.09\,. \tag{3.229}$$

Energy density:

$$u(x,t) = \frac{1}{2}\left[\rho\left(\frac{\partial y}{\partial t}\right)^2 + T\left(\frac{\partial y}{\partial x}\right)^2\right]\,. \tag{3.230}$$

Total energy

$$U(t) = \int_0^L u(x,t)dx\,. \tag{3.231}$$

Kinetic energy

$$KE(t) = \frac{1}{2}\rho \int_0^L \left(\frac{\partial y}{\partial t}\right)^2 dx\,. \tag{3.232}$$

Potential energy

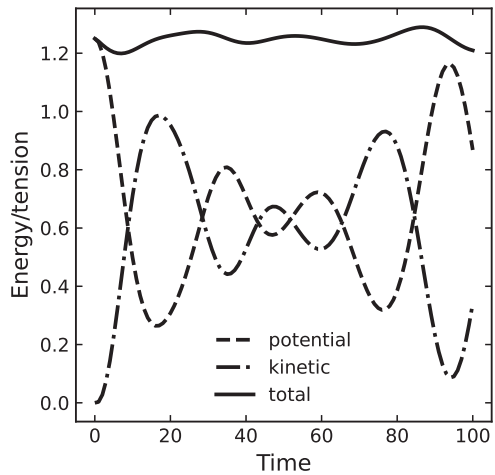
$$PE(t) = \frac{1}{2}T \int_0^L \left(\frac{\partial y}{\partial x}\right)^2 dx\,. \tag{3.233}$$

The energy per string tension is $U(t)/T = 1.25$, which is a constant.

The code below was run for a smaller time step and a longer total time than the previous waveform display. This helps reduce the fluctuations.

```
""" Calculate energy assuming y[i,j] is prefilled.
Try small dx and big tmax.
"""
print("T",T,"rho",rho,"v",v,"dx",dx,"dt",dt)

# Calculate energy density.
t  = np.empty(tmax+1)
uK = np.empty(tmax+1)
uP = np.empty(tmax+1)
u  = np.empty(tmax+1)
for j in range(tmax+1):
    K = 0
    U = 0
    for i in range(grid):
        if j == 0:
            Kp = 0
        else:
            Kp = ((y[i,j] - y[i,j-1])/dt)**2
        Up = ((y[i+1,j] - y[i,j])/dx)**2
        K += Kp
        U += Up
    K = 0.5*rho * K
    U = 0.5*T   * U
    t[j]  = j
    uK[j] = K/T
    uP[j] = U/T
    u[j]  = K/T + U/T
    if debug: print("%.2f %.2f %.2f" % (u,K,U))

fig, ax = plt.subplots(figsize=(4,4))
plt.plot(t,uP,lw=2,ls='dashed', label="potential")
plt.plot(t,uK,lw=2,ls='dashdot',label="kinetic")
plt.plot(t,u, lw=2,ls='solid',  label="total")
plt.xlabel("Time")
plt.ylabel("Energy/tension")
plt.legend(loc='lower center',frameon=False)
plt.savefig('3_2_wave_2.pdf')
plt.show()
```

```
T 10000.0 rho 1 v 100.0 dx 1 dt 0.001
```



The plot shows that the total energy per tension is not particularly constant, but it is roughly centred on the analytical value of 1.25 and must importantly, does not diverge.

3.2.7.4 Schrödinger equation

In this example, we solve the time-dependent Schrödinger equation for a Gaussian wavepacket moving in a simple harmonic oscillator potential. The Hamiltonian is

$$H_{jk} = -\frac{\hbar^2}{2m}\frac{\delta_{j+1,k} - 2\delta_{j,k} + \delta_{j-1,k}}{(\Delta x)^2} + V_j\delta_{jk}\,, \tag{3.234}$$

with simple harmonic oscillator potential

$$V_j = \frac{1}{2}kx_j^2\,. \tag{3.235}$$

The initial Gaussian wavepacket is

$$\Psi(t=0,x) = \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}\,. \tag{3.236}$$

The boundary conditions are $\Psi = 0$ at $x = a$ and $x = b$, which implies the first and last rows of H_{ik} are zero: $H_{0k} = H_{(N-1)k} = 0$, but $H_{00} = H_{(N-1)(N-1)} = 1$ so the row does not vanish.

We use the Crank-Nicolson scheme:

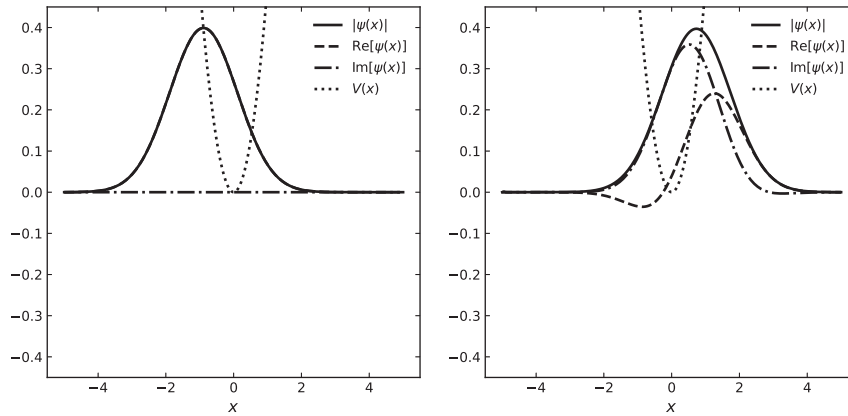
$$\Psi^{n+1} = \left(\mathbf{I} + \frac{i\Delta t}{2\hbar}\mathbf{H}\right)^{-1}\left(\mathbf{I} - \frac{i\Delta t}{2\hbar}\mathbf{H}\right)\Psi^n\,. \tag{3.237}$$

Consider the following code. For simplicity, we take $\hbar = m = k = 1$ and work over the range $[a, b] = [-5, 5]$. We achieve a small saving by using the Python finite difference package `findiff` for the central difference piece of the Hamiltonian. In addition, we are solving the Schrödinger equation using spare matrices and have imported the appropriate `scipy.spare` package.

```
"""Time-dependent Schroedinger equation."""

import numpy as np
from findiff import FinDiff # Finite difference equations.
from scipy.sparse.linalg import inv
from scipy.sparse import csc_matrix, eye, diags
# Poor person's movie.
from IPython.display import display, clear_output
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

# Grid in space and time.
a = -5
b =  5
x = np.linspace(a,b,200)
t = np.linspace(0,10,200)
dx = x[1] - x[0]
dt = t[1] - t[0]

# Parameters.
hbar = 1
m = 1
k = 1

# Discretize Hamiltonian as a scipy sparse matrix.
V = 0.5*k*x**2
H = -(hbar**2/(2*m)) * FinDiff(0,dx,2).matrix(x.shape) + diags(V)

# At the boundary, we have to modify the matrix to keep the
# boundary conditions. H = 1 (unchanged) at x = a and x = b.
HL = H.tolil() # convert matrix to list of list (no copy)
# Clear first and last rows.
HL[0,:] = HL[-1,:] = 0
# Set diagonal element in first and last positions to unity.
H[0,0]  = H[-1,-1] = 1

# Build the Crank-Nicholson propagator.
I_plus  = csc_matrix(eye(len(x)) + 1j * dt/(2*hbar) * H)
I_minus = csc_matrix(eye(len(x)) - 1j * dt/(2*hbar) * H)
U = inv(I_plus).dot(I_minus)

# Initial Gaussian wave packet.
sigma = 1
```

```
mu = -0.89 # Visually appealing initial position.
psi = np.exp(-(x-mu)**2/(2*sigma**2)) / (np.sqrt(2*np.pi)*sigma)
psi[0] = psi[-1] = 0

# Graphics.
def snapshot():
    ax.plot(x,abs(psi),    ls='solid',  label=r'$|\psi(x)|$')
    ax.plot(x,np.real(psi),ls='dashed', label=r'Re[$\psi(x)$]')
    ax.plot(x,np.imag(psi),ls='dashdot',label=r'Im[$\psi(x)$]')
    ax.plot(x,V,           ls='dotted', label=r'$V(x)$')
    ax.set_ylim(-0.45,0.45)
    ax.set_xlabel(r'$x$')
    ax.legend()

# Initial configuration.
fig, ax = plt.subplots()
snapshot()

# Motion picture.
movie = True
if movie:
    fig, ax = plt.subplots()
    # Start the time propagation.
    for it, tp in enumerate(t):
        psi = U.dot(psi)
        psi[0] = psi[-1] = 0
        # Boundary conditions: psi = 0 at x = a, b.
        if (it % 4) == 1:
            ax.cla()
            snapshot()
            display(fig)
            clear_output(wait=True)
            plt.pause(0.01)

# Final configuration.
if movie:
    fig, ax = plt.subplots()
    snapshot()

plt.savefig('3_2_schroedinger.pdf')
plt.show()
```

The code uses the Python `Ipython.display` tools to create a movie of the Gaussian wavepacket propagating in the potential. Since viewing a movie is not possible in a book, I have created snapshots of the initial and final waveform configuration. The magnitude, real part and imaginary part of the wavefunction are shown, along with the potential. The total magnitude of the wavefunction maintains its shape since the system is conservative. I started the wavefunction initially as purely real. The wavefunction oscillates back and forth in the potential approximately twice before I stopped the movie. A sizeable imaginary component of the wavefunction is present when stopped.



3.3 PROBLEMS

1. **Integrate tortoise equation**

   In general relativity, it is sometimes beneficial to transform the radial coordinate r to the tortoise coordinate r_*. The general transformation equation between the two coordinates is given by

   $$\frac{dr_*}{dr} = \frac{1}{\sqrt{f(r)g(r)}}\,, \tag{3.238}$$

   where $f(r)$ and $g(r)$ are metric coefficient functions of r. For the Schwarzschild metric outside a black hole,

   $$f(r) = g(r) = 1 - \frac{r_\mathrm{S}}{r}\,,$$

   where $r_\mathrm{S} = 2M$, and M is the mass of the black hole.

   The first order ODE, Eq. (3.238), defining the relationship between the coordinates has boundary conditions:

   $$\text{as} \quad r \to +\infty, \quad r_* \to +\infty\,; \quad \text{and as} \quad r \to r_\mathrm{S}, \quad r_* \to -\infty\,.$$

   (a) Setup the problem be writing the first-order ODE that needs to be solved as a function of the single parameter M. State the initial value you will use to solve the first order ODE. Your code need not implement the equation this way.

   In your code, take $M = 1$.

   (b) Write a function, or class, that returns r_* for arbitrary (r, M), i.e. numerically solves the first-order ODE to obtain $r_* = r_*(r)$.

(c) Write a function, or class, that returns r for arbitrary (r_*, M), i.e. numerically inverts your result in part (b), to find $r = r(r_*)$.

(d) Plot $r_*(r)$ and $r(r_*)$ on the same graph using different axis labels. i.e. for $r_*(r)$ the x-axis is r and for $r(r_*)$ the x-axis is r_*.

The r values used in the plots should not be dependent on the r values used to solve the ODE, i.e. I should be able to plot your solution for any arbitrary r and r_* within your range of integration.

(e) In your solution, integrate the differential equation to as near $2M$ as possible without Python giving any RuntimeWarning. State the minimum r you achieved.

You may use the sub-package `integrate` and functions in `special` from the SciPy library. Do not solve the problem analytically, although you may do so for comparison and validation.

2. **Newtonian gravitational orbits**

In this problem there are no explicit instructions on how to present your results. I view that as part of the problem. Use whatever tables, graphs, drawing and so forth that you feel appropriate to clearly convey your results to an undergraduate physics student. No constants or parameters are given. Be creative and choose them to give interesting results.

(a) Consider an object of mass m at a point (x, y) attracted by a gravitational force to another body of mass M fixed at the origin. Ignore the third dimension. Only considering gravity, write the force resolved into x- and y-components, and the corresponding equations of motion.

(b) Choose an initial position and velocity of m in the (x, y)-plane. Write a program to compute the subsequent motion under the force of gravity. Use any method you like to solve the ODE, including a Python library if you like.

(c) Can you obtain a closed elliptical orbit? Do you know how to make it a circular orbit? For the case of the elliptical orbit, determine the period. Calculate over 10 periods to show that your orbits are stable.

(d) Modify your code to simulate the transfer of orbits by burning of a rocket for a fixed length of time. You should show the orbit before and after the burn for sufficient time to demonstrate they are stable. During the burn you can ignore the effects of gravity and give the mass m a constant acceleration, if you like. Can you control the burn to get a desired orbit?

Alternative theories of gravity have postulated higher dimensional space. In three space dimensions, the gravitational force is proportional to $1/r^2$, but in higher dimensions the gravitational force can be proportional to $1/r^n$, where $(n+1)$ is the number of space dimensions.

(e) Use your program to investigate the motion for various values of n. Is it possible to obtain closed orbits for any value of n other than two? If so, are these orbits stable?

3. **Tolman-Oppenheimer-Volkoff equation: neutron stars**

The structure of a spherically symmetric body of isotropic material which is in static gravitational equilibrium can be modelled using general relativity. The resulting differential equation for the pressure in the body versus radial distance is called the Tolman-Oppenheimer-Volkoff (TOV) equation. We apply the TOV equation to nonrotating neutron stars and compute the maximum mass for a sequence of such stars.

The TOV equations is

$$\frac{dP(r)}{dr} = -\frac{Gm(r)\epsilon(r)}{r^2}\left[1+\frac{P(r)}{\epsilon(r)c^2}\right]\left[1+\frac{4\pi r^3 P(r)}{m(r)c^2}\right]\left[1-\frac{2Gm(r)}{c^2 r}\right]^{-1},$$

where $P(r)$ is the pressure, $\epsilon(r)$ is the energy density—not mass density—and $m(r)$ is the mass inside a sphere of radius r. In addition, the mass is given by

$$\frac{dm(r)}{dr} = 4\pi r^2 \epsilon(r)\,.$$

An equation of state (EOS) that connects $P(r)$ to $\epsilon(r)$ is needed. For the EOS, we choose the polytropic model:

$$P(\rho) = \kappa\rho^\gamma$$

and

$$\epsilon(r) = \rho(r) + \frac{P(r)}{\gamma - 1}\,,$$

where $\gamma = 1 + 1/n$, and $n \approx 1$ for neutron stars.

Consider a star of mass M and radius R. Since $P(r)$, $\epsilon(r)$ and $m(r)$ are equations of r only, we can solve the above system of equations by integrating r from $r \approx 0$ to $r = R$. At $r = 0$, $m(0) = 0$ and $\rho(0) = \rho_c$, where ρ_c is the central (maximal) rest-mass density. As $r = R$, $m(R) = M$ and $P(R) = 0$.

(a) Work in natural units with $c = G = M_\odot = 1$. This means masses are in units of the Sun's mass $M_\odot$ and distances are in units of $GM_\odot/c^2$.

(b) Take $\kappa = 100$ and $n = 1$.

(c) Take $\rho_c = 8 \times 10^7$ kg/m^3. To use this, you need to convert it to natural units.

(d) Since R is unknown in this problem, you will need to stop the integration when the pressure vanishes. You may handle this by writing your own integration function or using `scipy.integrate.solve_ivp` with the `events` argument.

(e) Draw a mass versus radius diagram.

(f) What is the maximum mass (in Sun units) of a neutron star and its radius in km?

Inspired by Sebastiano Bernuzzi.

4. **Friedmann equation: age and destiny of the Universe**

Let's consider a model of an expanding Universe based on the cosmological principle. The cosmological principle is the notion that the Universe is homogeneous and isotropic when viewed at a large enough scale. It is natural to investigate the age of the Universe and its ultimate destiny based on this model and currently measured, or predicted, model parameters.

We start with the Friedmann equation, assuming a matter-dominated Universe in which the radiation (plus neutrino) density can be ignored:

$$H^2 = \left(\frac{\dot{a}}{a}\right)^2 = \frac{\Omega_M}{a^3} + \frac{\Omega_k}{a^2} + \Omega_\Lambda \,, \tag{3.239}$$

where H is the Hubble parameter relative to its value today. The scale factor $a(t)$ is taken to be one at the present time $t = 1$, and is a measure of the size of the Universe. The parameter Ω_k is the spatial curvature density: $\Omega_k > 0$ for positive curvature, $\Omega_k < 0$ for negative curvature and $\Omega_k = 0$ gives a flat Universe. The parameter Ω_M is the matter (baryonic and dark matter) density today and Ω_Λ is the cosmological constant referring to the vacuum energy density today.

We could obtain a first-order differential equation by taking the square-root of Eq. (3.239) but we need to allow $\dot{a}$ to be either positive or negative—corresponding to either expansion or contraction of the Universe—and that information would be lost when taking the square-root.

(a) Instead, obtain a second-order differential equation by differentiating Friedmann's equation (assuming $\dot{a} \neq 0$).

(b) Using your second-order differential equation, write down the pair of first-order differential equations in matrix form.

We will solve the equations forward and backward in time from today to predict the future state of the Universe and its past, respectively. We start at the present time since these are the conditions we know; using $a = 1$ and $\dot{a} = 1$, by definition.

(c) Write code to solve the Friedmann equation for these initial conditions.

(d) Plot the scale factor versus time. Mark on your plot the present time, and use different colours for the forward evolution and backward evolution. Also, draw on your plot the curve for constant growth.

According to the standard cosmological model (ΛCDM), estimates for the parameters are $\Omega_M = 0.315 \pm 0.007$ and $\Omega_\Lambda = 0.685 \pm 0.007$. Hint: step (a) should be done in such a way as to eliminate Ω_k.

(e) Run your code for the following values of $(\Omega_M, \Omega_\Lambda)$ and comment on the results for each case.

(1,0)	present total density but with no vacuum density contribution,
(0.6,0)	60% of the present total density and no vacuum density,
(0,1)	no matter density and only vacuum density,
(0.95,0.5)	95% of the present matter density and a small vacuum density,
(0.3,0.7)	approximately the estimated values.

5. **Eigenvalue problem: electron in a potential**

Consider the time-independent Schrödinger equation for the wavefunction $\psi(x)$ in one dimension x:

$$\left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} - E + V(x)\right]\psi(x) = 0\,.$$

In this problem, we will consider four different potentials [2] :

$$V_1(x) = \begin{cases} 0 & \text{for } -a_0 < x < a_0 \\ \infty & \text{elsewhere} \end{cases},$$

$$V_2(x) = \begin{cases} V_0\left(\frac{x^2}{a_0^2} - \frac{1}{4}\right) & \text{for } -a_0 < x < a_0 \\ \infty & \text{elsewhere} \end{cases},$$

$$V_3(x) = V_0\frac{x^2}{a_0^2}\,,$$

$$V_4(x) = V_0\frac{x^4}{a_0^4}\,.$$

For boundary conditions, we will take $\psi(x = x_{\min}) = \psi(x = x_{\max}) = 0$. The wavefunction $\psi(x)$ can be normalized to

$$1 = \int_{x_{\min}}^{x_{\max}} dx|\psi(x)|^2 .$$

Notice that $\psi(x)$ should always be real in this problem.

We will solve an atomic physics problem by considering the wavefunction for a non-spinning electron of mass m and charge e. Use SI units for all physical quantities and constants, but express energies in units of electron volt, rather than Joules.

(a) Write a general program to solve the Schrödinger equation as a eigenvalue problem using the shooting method; $\psi(x)$ are the eigenfunctions and E are the eigenvalues. Write your own fourth-order Runge-Kunta method to solve the differential equation to obtain the eigenvalues. You may use whatever method you like (include library modules) to find the eigenfunctions. If you need to do any definite integrals, you may use library modules.

(b) For potential $V_1(x)$, find the lowest eigenvalue and corresponding eigenfunction for a_0 the Bohr radius. State your eigenvalue to two decimal places in units of eV. Make a plot of the corresponding eigenfunction properly normalized over the allowed range. For this question, you could probably solve it analytically or look up the results or both, so this is considered a warm up and to help validate your code.

(c) For a question that might be harder to find, and solve analytically, repeat the previous question but for potential $V_2(x)$, with $V_0 = 9\,\text{eV}$.

(d) Moving to something more realistic, repeat the previous question but for potential $V_3(x)$ in the range $-10a_0 \leq x \leq 10a_0$, which is meant to approximate the infinite range. This time find the lowest three eigenvalues and their corresponding eigenfunctions. If you recognize this as a quantum simple harmonic oscillator, you will be able to validate your results.

(e) Finally, repeat the previous question for the quantum anharmonic oscillator potential $V_4(x)$.

6. **Moving point charge in a static electric field**

The magnitude of the electric field due to an infinite line of positive charge per unit length λ at a perpendicular distance r from the line is

$$E = \frac{\lambda}{2\pi\epsilon_0}\frac{1}{r} .$$

The electric field is directed away from the line. If an electron is released at a distance r from the line it will be attracted to the line with a force $F = eE$ (ignoring gravity).

(a) Write the equation of motion for an electron in this field. Next, set all the constants and numerical coefficients equal to unity.

(b) Code the standard fourth-order Runge-Kutta method to investigate the motion of the electron. Write the code yourself and do not call any libraries that implement the Runge-Kutta method. You may only use NumPy and Matplotlib.

(c) Use a step size of $\Delta t = 0.0001$ and take the initial conditions $t = 0$, $r = 1$ and $v = 0$, where v is the speed of the electron. Assume there is a negligibly small hole in the line through which the electron can pass. Plot your solution over the range $t = [0, 93.346]$.

(d) Do you obtain oscillations? If so, how many times does the electron cross the line?

(e) What happens if you increase the time range to about $t = [0, 94]$, or more?

(f) In working the problem are there any other plots you could make? If so, explain which ones will be useful and show them. Explain what you see in the plots you have shown.

(g) Are you able to solve this problem analytically, or approximately analytically, to compare with your numerical solution? If you cannot solve it analytically, can you write something analytic about it?

(h) You could try to improve your results by coding an adaptive Runge-Kutta method to reduce the step size near the line and pass through it slowly. Alternatively, you could increase the step size near the line and skip the singularity. Do you believe either of these methods would help obtain more oscillations, and if so, why?

(i) Do you think the solution should be oscillatory?

(j) Discuss how you could change the model to better describe physical reality.

7. **Pseudo-spectral solution to second order ODE**

Write a class or function that can solve a general second-order linear differential equation as a boundary value problem using the pseudo-spectral method. Set it up so that you pass to your code the function, the directlet boundary values, and the order of the calculation—number of basis functions. Use Chebyshev polynomials and the Chebyshev points over the range $-1 \leq x \leq 1$.

(a) Test your code on the homogeneous ODE

$$xy'' - (x+1)y' + y = 0$$

with boundary conditions $y(-1) = y(1) = 1$. The analytic solution is

$$y(x) = c_1(x+1) + c_2 e^x \,.$$

Plot your pseudo-spectral solution and theory, and show the differences.

(b) Repeat the previous question with boundary conditions $y(-1) = 1$ and $y(1) = 0$.

(c) Test your code on the inhomogeneous ODE

$$y'' - \frac{1}{x}y' + \frac{1-x}{x}y = 4xe^x$$

with boundary conditions $y(-1) = y(1) = 0$. The analytic solution is

$$y(x) = x^2 e^x + c_1(-1-2x)e^{-x} + c_2 e^x \,.$$

Plot your pseudo-spectral solution and theory, and show the differences.

8. **Teukolsky angular equation**

The Teukolsky equation describes field perturbations of a Kerr (rotating) black hole [14]. Spin perturbations of $s = 0, 1/2, 1, 2$ are incorporated in the same second-order differential equation in the variables (t, r, θ, ϕ). The differential equation is usual solved by separation of variables, leaving nontrivial ODEs in r and θ. Here we will be interested in solving the angular eigenvalue equation.

The eigenvalues can be obtained iteratively by the following set of coupled first-order differential equations using a continued method [15]:

$$\begin{aligned} \frac{d_s A^m_{\ell\ell'}}{d(a\omega)} &= -\sum_{\alpha,\beta,\gamma\neq l} \frac{{}_sA^m_{\gamma\alpha}\,{}_sA^m_{\ell\beta}}{{}_sE^m_\ell - {}_sE^m_\gamma}\langle\alpha,\beta\rangle\,{}_sA^m_{\gamma,\ell'} \,, \\ \frac{d_s E^m_\ell}{d(a\omega)} &= -\sum_{\alpha,\beta} {}_sA^m_{\ell\alpha}\,{}_sA^m_{\ell\beta}\langle\alpha,\beta\rangle \,, \end{aligned} \tag{3.240}$$

The expectations values are given by

$$\langle\alpha,\beta\rangle = \left\langle s\alpha m \left| \frac{d\mathcal{H}_1}{d(a\omega)} \right| s\beta m \right\rangle ,$$

where the Hamiltonian is given by

$$\mathcal{H}_1 = a^2\omega^2\cos^2\theta - 2a\omega s\cos\theta \,.$$

The bra and ket represent the angular integration of pairs of spin-weighted spherical harmonics. They can be related to the rotation matrix elements of quantum mechanics. The ones we need are

$$\langle s\ell' m|\cos^2\theta|slm\rangle = \frac{1}{3}\delta_{\ell\ell'} + \frac{2}{3}\sqrt{\frac{2\ell+1}{2\ell'+1}}\langle \ell 2m0|\ell' m\rangle\langle \ell 2{-}s0|\ell'{-}s\rangle$$

and

$$\langle s\ell' m|\cos\theta|slm\rangle = \sqrt{\frac{1\ell+1}{2\ell'+1}}\langle \ell 1m0|\ell' m\rangle\langle \ell 2{-}s0|\ell'{-}s\rangle\,,$$

where $\langle j_1 J_2 m_1 m_2|JM\rangle$ are Clebsch-Gordon coefficients. Note that in the above $-s$ is negative s, not subtraction.

We will only be interested in the energy eigenvalues for this problem but you must also determine the eigenfunctions to solve the problem. You thus don't need to determine the spin-weighted spherical harmonics.

Solve the pair of coupled differential equations (3.240) to obtain the energy eigenvalues as a function of $a\omega$ for $s = \pm 2$ and $(\ell, m) = (2,-2),(2,1),(2,0),(2,1),(2,2),(3,0),(4,0),(5,0),(6,0)$.

The initial conditions can be taken as ${}_sS_\ell^m(\theta,0) = {}_sY_\ell^m(\theta)$ and ${}_sE_\ell^m(0) = \ell(\ell+1)$, which implies ${}_sA_{\ell\ell'}^m = \delta_{\ell\ell'}$.

You may consider the following symmetries. These are not important for the physics but sometimes the numerical solution is more stable in one particular representation versus another.

$$\begin{aligned}
{}_{-s}S_\ell^m(\theta,a\omega) &= {}_sS_\ell^m(\pi-\theta,a\omega)\,,\\
{}_{-s}E_\ell^m(a\omega) &= {}_sE_\ell^m(a\omega)\,,\\
{}_sS_\ell^m(\theta,-a\omega) &= {}_sS_\ell^{-m}(\pi-\theta,a\omega)\,,\\
{}_sE_\ell^m(-a\omega) &= {}_sE_\ell^{-m}(a\omega)\,.
\end{aligned}$$

I don't know of a Python library for calculating Clebsch-Gordon coefficients, however, there is a SymPy library to calculate them. So you can either write a lambdification of the SymPy library or calculate them yourself.

9. **Transmission coefficients**

Consider the following differential equation.

$$\frac{d^2\psi(r_*)}{dr_*^2} = \left[V(r(r_*)) - \omega^2\right]\psi(r_*)\,,$$

where ω is a real frequency and

$$V(r) = \left(1-\frac{2M}{r}\right)\left(\frac{\ell(\ell+1)}{r^2} + \frac{(1-s^2)2M}{r^3}\right)\,,$$

with M, s and ℓ three parameters.

We will be solving the differential equation as a boundary value problem between $r = (2M, \infty)$, but using the independent variable r_* given by the following equation

$$r_*(r) = r + 2M \ln\left(\frac{r}{2M} - 1\right) .$$

Note that now $r_* = (-\infty, \infty)$.

You might find the inverse transform useful:

$$r(r_*) = 2M\left[1 + W(e^{\frac{r_*}{2M}} - 1)\right] ,$$

where W is the Lambert function.

The boundary conditions are such that

$$\psi(r_* \to -\infty) = \exp[-i\omega r_*(2M)] ,$$
$$\psi(r_* \to \infty) = B_{\text{in}}(\omega)\exp[-i\omega r_*] + B_{\text{out}}(\omega)\exp[i\omega r_*] .$$

Take $M = 1$ and $s = \ell = 2$. For a given ω, integrate the equation to determine amplitudes B_{in} and B_{out}.

From these amplitudes, determine the transmission coefficient

$$T = 1 - \left|\frac{B_{\text{out}}(\omega)}{B_{\text{in}}(\omega)}\right|^2 .$$

Plot the transmission coefficient versus ω in the range $(0, 1]$.

You may use any integration method you wish. You may use `scipy.special.lambertw` for the Lambert function.

Hint: The coefficients B_{in} and B_{out} can be solved using two equations with two unknowns.

Congratulations, you have just calculated a transmission coefficient for Hawking radiation (evaporation) of a Schwarzschild black hole.

10. **Central configuration**

Place three point particles of equal mass initially at rest in a plane. Assume gravity is the only force acting between them. Configure the three particles such that after some time they will all simultaneously meet at a single point

(a) Write a program as general as possible to solve this three-body problem. Show the trajectories of all three particles in Cartesian coordinates (i.e. a single 2D (x, y) graph). Show the initial particle positions and make it clear which particle is which by different colours.

(b) Make a second plot expanding the region where the particles meet.

(c) Do not let them pass each other but bring them together as close as you can. Make it clear how close they can get before problems occur.

(d) Bonus: repeat for six particles.

You may use any integration method you wish, including libraries.

11. **General relativity gravitational orbits**

This problem determines the orbital trajectories of a particle of energy per unit mass E in a Schwarzschild geometry with mass parameter M. Conservation of angular momentum L confines the particle motion to a plane, which conveniently may be chosen as the equatorial plane with $\theta = \pi/2$.

The radial and angular equations of motion are

$$E = \frac{1}{2}\left(\frac{dr}{d\tau}\right)^2 + V_{\text{eff}}(r) \quad \text{and} \quad \frac{d\phi}{d\tau} = \frac{L}{r},$$

where

$$V_{\text{eff}}(r) = -\frac{M}{r} + \frac{L^2}{2r^2} - \gamma\frac{ML^2}{r^3}$$

and $\gamma = 1$ for general relativity (GR) and $\gamma = 0$ for Newtonian mechanics. The conserved quantities E and L are constants of the motion.

The orbits can be classified into four types: circular, elliptical, scattering and plunging. These different types of orbits can be distinguished by the energy of the particle E relative to the potential $V_{\text{eff}}(r)$. Figures 3.8 below show the potential and possible energies (left) that give the different orbits (right).

The different types of orbits behave differently in GR and Newtonian gravity. Circular orbits occur when the particle energy is equal to the extreme of the potential energy function. Newtonian mechanics gives one stable circular orbit, while GR gives a stable and an unstable circular orbit. Bound elliptical orbits occur when the particle energy intersects the potential at two points. These are turning points that correspond to the outer and inner radii of the elliptical orbit—recall that these theories are classical. The GR case has bound elliptical orbits that precess. The scattering orbit occurs when the particle approaches from infinity and has an energy less than the maximum of the potential. The amount of scattering is different for GR and Newtonian gravity. The plunging orbits occur when the particle approaches from infinity and has an energy greater than the maximum of the potential. This is not possible in Newtonian gravity.

(a) Plot the GR and Newtonian potentials for $M = 1$ and $L = 4.3$. Restrict the range to $r > 2M$.

(b) We could solve the ODE for r by taking the square-root but this would involve a $\pm$ sign that makes it difficult to integrate. A better approach is to take the derivative of the radial ODE to obtain a second order ODE. In this case E drops out of the ODE and only appears in the boundary conditions. Code the systems of coupled ODEs as a function that can be be used for integration. Try to make your potential and ODEs as general as possible so they can be used for all the different orbits.

(c) For circular orbits, first calculate the extreme of the potential; either analytically or numerically. Do this for both the GR and Newtonian cases. Integrate the circular orbits over at least one period.

(d) For elliptical orbits, pick an E value that has two turning points. Use the same E value for both GR and Newtonian gravity orbits. Integrate the elliptical orbit over at lease one period and the precessing orbit over a few precessions.

(e) For the scattering orbits, pick an E value that has a single turning point. Integrate the scattering orbit starting at $r = 100$ which is meant to represent infinity. Include the Newtonian case for the same value of E.

(f) For the plunging orbit, pick an E value above the GR potential. Integrate the plunging orbit starting at $r = 100$ which is meant to represent infinity. Stop your integration as close to $r = 2M$ as possible.

The objective of this problem is to obtain the plots shown in Fig. 3.8.

12. **Gravitational lensing**

Gravitational lensing is when a massive celestial body like a galaxy causes a sufficient curvature of spacetime for the path of light around it to be visibly bent, thus acting as a lens.

We start from the same photon orbit equation of Problem 10, Sec. 2.10.

$$\frac{d\phi}{dr} = \pm\frac{1}{r^2}\left[\frac{1}{b^2} - \frac{1}{r^2}\left(1 - \frac{2GM}{c^2 r}\right)\right]^{-1/2}.$$

Rather than integrate this equation to find the total change in angle, we will integrate it over ϕ to find the trajectory of photons.

(a) To remove the troublesome $\pm$ signs, square both sides of the ODE.

(b) To remove the troublesome inverse, take the inverse of the ODE to obtain $(dr/d\phi)^2$.

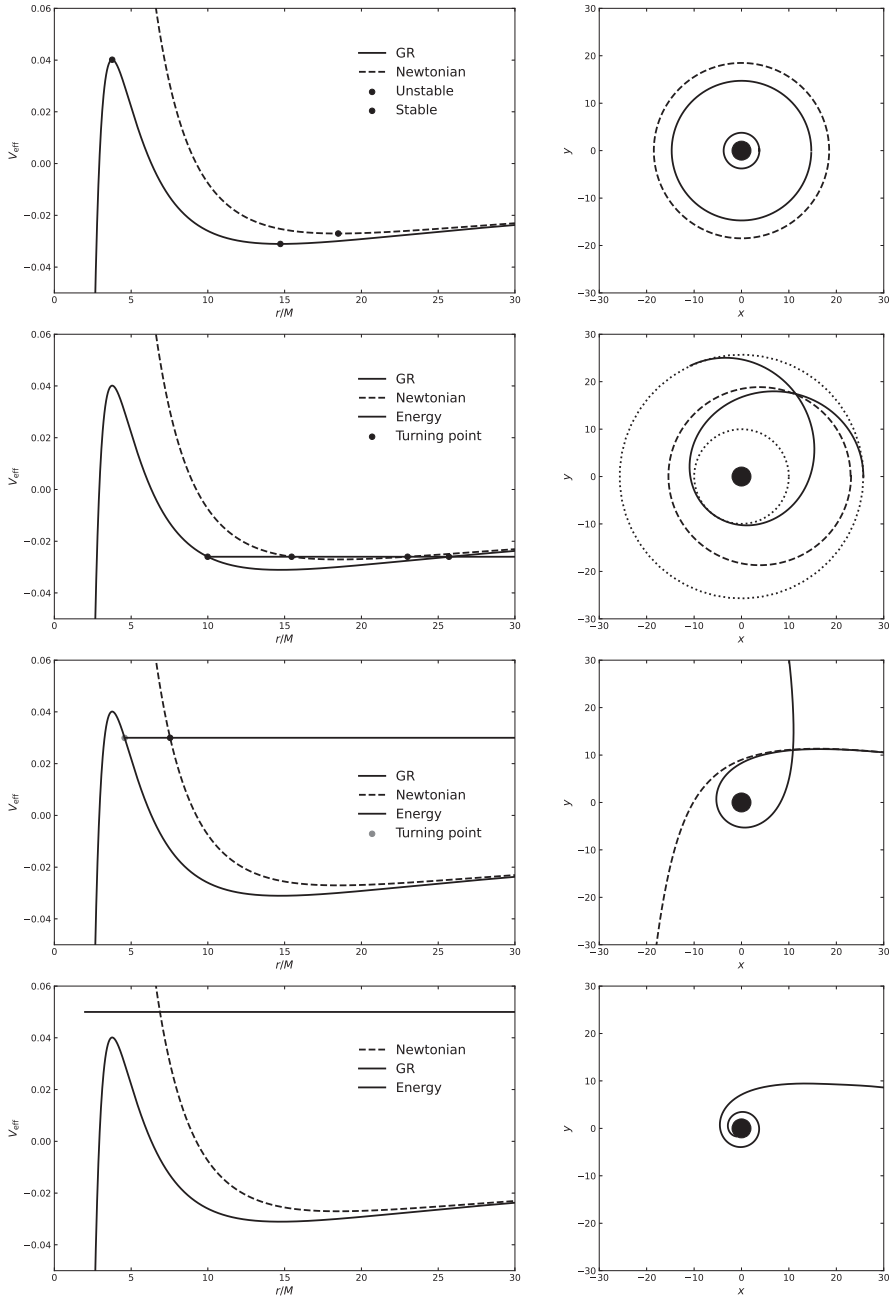


Figure 3.8 Left: Gravitational potentials and particle energies. Right: particle trajectories corresponding to the energies.

(c) To remove the square of the derivative, differentiate both sides of the ODE by ϕ. You now have a nice second-order ODE—albeit nonlinear.

(d) To avoid infinite values of r, apply the transformation $u = 1/r$ to the ODE.

Use units such that mass is measured in meters: $GM/c^2 = 1477.1\,\mathrm{m}$. Take the galaxy mass to be $M = 28M_\odot$.

Consider a point light source, a point galaxy at $(x, y) = (0, 0)$, and an observer (focus); all in a line with the source and observer equidistant on each side of the galaxy. For the position of the source, take $r = 10^6$, which corresponds to $u(\phi = \pi) = 10^{-6}$, and an initial angle given by $-du(\phi = \pi)/d\phi = 10^{-6}$.

(e) Solve the resulting ODE. You will be integrating ϕ over the range $[\pi, 0]$, π representing the position of the source and 0 the position of the observer.

(f) Convert u back to r, and convert (r, ϕ) to (x, y). Make a plot of (x, y) showing the source, galaxy and focus. Assuming symmetry, draw the up-and-down light-paths around the galaxy from the source to the focus. Also mark undeflected light-rays that reach the focus with the same angle as the deflected light-rays and the apparent source positions.

Your plot should contain equivalent information to Fig. 3.9. The apparent source positions represent the position of an Einstein ring in two dimensions.

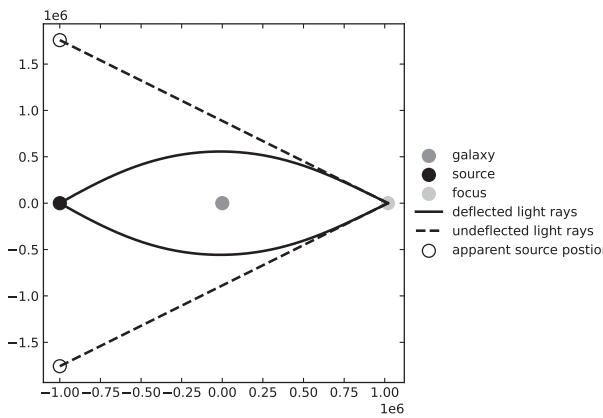


Figure 3.9 Two trajectories showing the bending of light-rays caused by a galaxy's mass. The apparent source positions represent the position of an Einstein ring in two dimensions.

13. **Poisson equation**

The potential $\phi(x, y)$ of a region containing a charge density $\rho(x, y)$ is given by

$$\frac{\partial^2\phi(x,y)}{\partial x^2} + \frac{\partial^2\phi(x,y)}{\partial y^2} = -4\pi\rho(x,y)\,.$$

Consider a region

$$0 \le x \le 25 \quad \text{and} \quad 0 \le y \le 25\,,$$

with

$$\phi(0,y) = \phi(25,y) = \phi(x,0) = \phi(x,25) = 0\,,$$

and

$$\rho(x,y) = x(x-25)y(y-25)\,.$$

Compute the potential in the interior region.

(a) Write down the difference equation.
(b) Solve the difference equation to a tolerance of two, and plot your result as a 3D contour plot.

14. **Poisson equation with two sources**

The potential $\phi(x, y)$ of a region containing a charge density $\rho(x, y)$ is given by the Poisson equation

$$\frac{\partial^2\phi(x,y)}{\partial x^2} + \frac{\partial^2\phi(x,y)}{\partial y^2} = -4\pi\rho(x,y)\,.$$

Consider a region $-10 \le x \le 10$ and $-10 \le y \le 10$, with $\phi(-10, y) = \phi(10, y) = \phi(x, -10) = \phi(x, 10) = 0$, and

$$\rho(x,y) = \begin{cases} 100 & \text{for } (x,y) = (5,5)\,, \\ -100 & \text{for } (x,y) = (-5,-5) \\ 0 & \text{elsewhere.} \end{cases}$$

Devise a numerical iteration scheme to find the potential in the interior of the square.

(a) State the method you will use.
(b) What parameters will you use?
(c) Write the code and show a three-dimensional plot.

15. **The Laplace equation as a boundary value problem**

Consider the one-dimensional boundary-value problem,

$$\frac{\partial^2 u}{\partial x^2} = -5\,,$$

in the interval $x = [0, 1)$ with boundary conditions $u(x = 0) = 1$ and $u(x = 1) = 2$. Divide the interval $[0, 1]$ into $N = 5$ discrete points x_n. Find the function $u(x_n) = u_n$ at these points. Write the boundary-value problem in finite-difference form for every interior point n and the boundary conditions $u_0 = 1$ and $u_{N+1} = 2$. Discretize the interval $[0, 1]$ into $(N + 2)$ evenly spaced points, including the boundary points, with separation $\Delta x = 1/(N + 1)$. Write the set of equation in matrix form as $Au = b$, where $u = u_1, \ldots, u_N$.

(a) Solve the linear system of equations for the unknown vector u.

(b) Plot $u(x)$ versus x including the boundary values.

(c) Copy and modify your code to solve the two dimensional case.

Now consider the Laplace equation in two dimensions:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y} = 0\,.$$

Take as boundary conditions $u(x = 0) = 3$, $u(x = 1) = -1$, $u(y = 0) = -5$, $u(y = 1) = 5$. Use 50 points along both x and y.

(d) Clearly display your result by using an appropriate plot.

16. **Diffusion equation with constant source**

The temperature distribution $T(x, t)$ in a wire with perfectly insulated sides which is heated by an electric current is

$$\frac{\partial T(x,t)}{\partial t} = \kappa \frac{\partial^2 T(x,t)}{\partial x^2} + C\,,$$

where $C = j\frac{I^2 R}{\rho c \omega}$ is a complicated set of parameters that we will not worry about, and κ is the thermal diffusivity of the wire material.

Devise a difference equation to calculate $T(x, t)$ numerically. Assume that the value of C does not change as the wire heats up.

Take $\kappa = 1.14\ \mathrm{cm}^2/\mathrm{s}$, $C = 1^\circ\mathrm{C/s}$, and the length of the wire to be 8 cm. Take the entire wire to initially have a temperature of 0°C. Assume the ends of the wire are fixed at a temperature of 0°C for all time. What is the peak temperature of the wire and at what location along its length does the peak temperature occur?

17. **Neutron diffusion**

Assume the neutron density $n(\mathbf{r}, t)$ obeys the diffusion equation:

$$\frac{\partial}{\partial t} n(\mathbf{r}, t) = D\nabla^2 n + Cn\,,$$

where D is the diffusion constant and C is the creation rate for neutrons. To simplify the problem, consider a one-dimensional system of length L,

$$\frac{\partial}{\partial t}n(x,t) = D\frac{\partial^2 n}{\partial x^2} + Cn\,,$$

with $-L/2 \leq x \leq L/2$. If all neutrons that reach the boundary escape from the system, then

$$n(x = -L/2, t) = 0 \quad \text{and} \quad n(x = L/2, t) = 0$$

are the Dirichlet boundary conditions.

(a) Use the FTCS scheme to discretize the neutron diffusion equation in one-dimension.

(b) What is the criterion for numerical stability?

(c) Write a program to solve the neutron diffusion equation as an initial value problem using the FTCS scheme.

(d) Take the $D = 10^5$ m^2/s and $C = 10^8$ s^{-1} for U^{235}. Choose the initial neutron density to be a delta function at the centre of the system:

$$n(0,0) = \begin{cases} 1/h \text{ for } i = N/2 \\ \text{otherwise} \end{cases},$$

where h is the space discretization size and N is the number of space divisions in the discretization.

(e) Calculate the average density

$$\bar{n}(t) = \frac{1}{L}\int_{-L/2}^{L/2} n(x,t)dx\,.$$

(f) Pick a length that gives a subcritical system—the neutron density is finite for all time. Plot $n(x,t)$ and $\bar{n}(t)$.

(g) Pick a length that gives a supercritical system—the neutron density will continue to increase for all time. Plot $n(x,t)$ and $\bar{n}(t)$.

(h) Can you estimate the length at which the system is critical—between subcritical and supercritical—numerically?

18. **Wave equation for massive scalar particles**

The Klein-Gordon wave equation is

$$\left[\frac{\partial^2}{\partial t^2} - \frac{\partial^2}{\partial x^2} + m^2\right]\phi(x,t) = 0\,,$$

where m is a mass parameter and natural units of $\hbar = c = 1$ have been used.

(a) Derive a numerical method using difference equations to solve the Klein-Gordon equation.

(b) Choose your own values for Δt, Δx and m (non-zero). State what they are and why you made this choice. Consider a range of 10 x-values and set the initial conditions for $t = 0$ to $x_i = (0, 1, 1.5, 1, 0, 0, 0, 0, 0, 0)$ and any other conditions you may need.

(c) Plot the solution over the range of x for several time values that show significant propagation and the effects of the mass term.

(d) Is there some physical principle you can use to validate your results? Just state it, you don't need to actually show it.

(e) Comment on your results.

For visualization, you might want to smooth the solution using cubic splines, for example.

19. **Gravitational ringdown**

Consider the partial differential equation

$$\left[4\frac{\partial^2}{\partial u \partial v} + V_\ell(u, v)\right] \psi(u, v) = 0\,.$$

The light-cone variables (u, v) are related to the spacetime coordinates (t, r) by

$$u = t - r_* \quad \text{and} \quad v = t + r_*\,,$$

where

$$r_* = r + 2M \ln\left(\frac{r}{2M} - 1\right) \quad \text{and} \quad r = 2M\left[1 + W\left(\exp\left(\frac{r_*}{2M} - 1\right)\right)\right]\,,$$

with $W(x)$ the Lambert W function and M is a free mass parameter. The potential V_ℓ is given by

$$V_\ell(r) = \left(1 - \frac{2M}{r}\right)\left(\frac{\ell(\ell+1)}{r^2} - \frac{6M}{r^3}\right)\,,$$

where ℓ is an angular momentum quantum number.

(a) Write the PDE as a difference equation on a grid $(u, v) \to (i, j)$, and state what you will use for the argument of the potential V_ℓ.

You are free to pick the grid size and step size. You may use `scipy.special` for the Lambert function.

(b) Write code that will solve the differential equation in general.

(c) Use the following parameters in your code: $M = 1/2$ and $\ell = 2$. Place your coordinates system at $u_0 = v_0 = 0$. Take the following initial conditions: $\psi_{0,v}$ is given by a Gaussian function with mean 10 and width 3, and $\psi_{u,0} = \psi_{0,0}$. Use a Gaussian normalized to unity.

(d) Plot the absolute value of the wavefunction versus time t at $r = 10$. Use a log-log scale with a time range of $t = [20, 160]$.

Congratulations, you have just calculated the gravitational waveform for the ringdown that could have resulted a black-hole merger.

20. **Advection equation**

(a) Write code to reproduce Fig. 3.6.

(b) Make runs with a CFL values $\alpha = 1$ and α small. Explain what you see? You may need to remove the FTCS scheme result from the plot to see the others.

(c) For your different CFL values, try extending the total time. Explain what you see?

21. **Burger equation**

(a) Derive the Burger difference equation in the Lax-Wendroff scheme Eq. (3.187) using the method in Sec. 3.2.4.1.

(b) Write code to reproduce Fig. 3.7.

(c) Make runs with CFL values $\alpha = 1$ and α small. Explain what you see? You may need to remove the FTCS scheme result from the plot to see the others.

(d) For your different CFL values try extending the total time. Explain what you see?

22. **Time-dependent Schrödinger equation**

If a Hamiltonian $\mathcal{H}$ is time independent, we can write the general solution to the Schrödinger equation as

$$\Psi(x,t) = \exp\left[-i\mathcal{H}(t-t_0)/\hbar\right]\Psi(x,t_0)\,.$$

It is common in analytical calculations to expand the exponential of the Hamiltonian operator (time-evolution operator) in a Taylor series and use it as such. Numerically, a different approach is to treat the time-evolution operator as a matrix $\mathbf{H}$. We can always diagonalize the Hamiltonian with some matrix $\mathbf{D}$. Then the exponential of the Hamiltonian is equal to

$$e^{[-i\mathbf{H}(t-t_0)]}\mathbf{D}e^{[-i\varepsilon_0(t-t_0)]}e^{[-i\varepsilon_1(t-t_0)]}\ldots\mathbf{D}^\dagger\,,$$

where $\varepsilon_0, \varepsilon_1, \ldots$ are the eigenvalues of the Hamiltonian. While one can calculate this numerically yourself, I recommend the library module `scipy.linalg.expm`.

In this problem we will study the time-evaluation of a Gaussian wavepacket in free space. From the above equation, we see that we need the initial Gaussian wavepacket profile at $t = t_0$, and the Hamiltonian which in this case is simply the kinetic energy term, which is proportional to the second spacial derivative. We have previously seen how to calculate the kinetic energy term; here we need to exponentiate it. The initial Gaussian wavepacket can be taken as [16]

$$\Psi(x, t = 0) = A \exp\left[-\frac{\sigma_p^2 (x - x_0)/hbar^2}{1 - 2j\sigma_p^2 \tau/(\hbar m)} + jp_0 x/\hbar \right],$$

where

$$A^2 = \frac{\sqrt{2}\sigma_p/\hbar}{\sqrt{\pi}(1 - 2j\sigma_p^2 \tau/(\hbar m)}.$$

This is a very general form where x_0 is the packets initial mean position, τ is the time at which the packet is the narrowest and p_0 is the mean momentum. The other parameters are as usual, except that σ_x has been replace by the width of the momentum distribution σ_p by using the uncertainty relation $\sigma_x \sigma_p = \hbar/2$.

For this problem take $m = \hbar = 1$ and use $x_0 = -20, \sigma_p = 0.2, p_0 = 3, \tau = 5$. Take a total length of 100 units over the range $[-L/2, L/2]$, with 256 equal size divisions. For time, take the initial time $t_0 = 0$ and propagate for a total time of 10 units, using a time step of 0.1.

(a) Plot the squared modules of the wavepacket every 10 time steps.

(b) Compare with theory, which is given by

$$|\Psi(x, t)|^2 = B \exp\left[-\frac{2}{\hbar^2} \frac{\sigma_p^2 (x - x_0 - p_0 t/m)^2}{1 + r\sigma_p^4 (t - \tau) * *2/(m\hbar)^2} \right],$$

where

$$B^2 = \sqrt{\frac{2}{\pi}} \frac{\sigma_p/hbar}{\sqrt{1 + r\sigma_p^4 (t - \tau)^2/(m\hbar)^2}}.$$

(c) Assuming you used a three-point central difference of second order for the second derivative, now try a five-point central differences of fourth order.

(d) Add a Heaviside step-function potential

$$V(x) = \begin{cases} 0, & \text{if } x < 0\,, \\ V_0 & \text{if } x > 0 \end{cases}$$

to the Hamiltonian. Be careful of how you handle the region close to $x = 0$ numerically.

(e) Add the motion of a classical particle in this potential to the plot.

(f) Calculate the fraction of the wavepacket transmitted and reflected. Also determine this for the classical particle.

CHAPTER 4

Fourier transforms

Fourier transforms are a powerful approach to gaining insights into some physics problems, such as solving differential equations and expanding solutions into their fundamental modes in field theory, to name only two examples. In terms of data analysis, spectral analysis and signal processing, which are based on the Fourier transform, are two common techniques used in several fields of physics.

Since the Fourier transform is continuous and infinite on the domain, we first discretize the Fourier transform to develop the discrete Fourier transform (DFT) that can be implemented on a computer. This allows a better understanding of the fast-Fourier transform (FFT) algorithm which will be used exclusively as library modules and thus treated as a black box. Lastly, I will discuss some numerical considerations which either arise from experience or are encountered in nonidealized problems.

Consider a physical process in the time t domain that has values $h(t)$. In general, h is complex and typically t is taken as real time $t \geq 0$. In the frequency f domain, the process can be represented as $H(f)$. In general, H is complex and $-\infty < f < +\infty$. Our discussion will be phrased in terms of the time/frequency domains, but other reciprocal domains are common in physics such as length/energy, for example. It is common to transform from configuration space to energy-momentum space to make certain calculations easier.

We can go back and forth between the time and frequency domains with Fourier transforms:

$$H(f) = \int_{-\infty}^{\infty} dt\, h(t) e^{2\pi i f t} \quad \text{and} \quad h(t) = \int_{-\infty}^{\infty} df\, H(f) e^{-2\pi i f t}\,. \tag{4.1}$$

Different conventions for the sign of the exponential and normalizations are possible. One can also use angular frequency $\omega = 2\pi f$ or period

DOI: 10.1201/9781003660873-4

$T = 1/f$ instead of frequency. In angular frequency, the Fourier transforms become

$$H(\omega) = \int_{-\infty}^{\infty} dt\, h(t) e^{i\omega t} \quad \text{and} \quad h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} d\omega H(\omega) e^{-i\omega t}\,. \tag{4.2}$$

Again, other normalizations are possible.

In physics $h(t)$ is usually real, which means $H(-f) = [H(f)]^*$. Other symmetries allow simplifications. The power spectral density $|H(f)|^2$ is particularly useful and will often be used in what follows.

4.1 DISCRETE FOURIER TRANSFORM

For computational work, $h(t)$ is hardly ever continuous. Often $h(t)$ is sampled over an evenly spaced time interval Δ called a time series. The sampling rate, or frequency, is $1/\Delta$. The discrete function, due to the discrete time, is $h_k = h(k\Delta)$, where $k = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$. Consider the typical case of $h_k \equiv h(t_k), t_k \equiv k\Delta$, where $k = 0, 1, 2, \ldots, N-1$ (N even). Since t_k is finite, f is finite; the number of degrees of freedom in either representation must be the same. There are N values of information; $f_n \equiv n/(N\Delta)$, where $n = -N/2, \ldots, N/2$. The $\pm N/2$ end points are the Nyquist frequencies—they have the same absolute value.

Discretizing the integral over a finite time T gives

$$H(f_n) = \frac{1}{T} \int_0^T dt\, h(t) e^{2\pi i f_n t} \approx \frac{1}{T} \sum_{k=0}^{N-1} \Delta h_k e^{2\pi i f_n t_k} = \frac{1}{N} \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N}\,. \tag{4.3}$$

The discrete Fourier transform (DFT) is

$$H_n = \frac{1}{N} \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N}\,. \tag{4.4}$$

There are no dimensionful parameters in the argument of the exponential. The functions h_k and H_n are complex in general.

The inverse DFT can be written as

$$h_k = \sum_{n=0}^{N-1} H_n e^{-2\pi i k n/N}\,. \tag{4.5}$$

Since $H_{(-n)} = H_{(N-n)}$ for $n = 1, 2, 3, \ldots$, we typically take H_n for $n = 0, \ldots, N-1$. The positive frequencies are over the range $1 \le n \le N/2 - 1$. The negative frequencies are over the range $N/2 + 1 \le n \le N - 1$, and $n = N/2$ is used for the $\pm$ Nyquist frequency. The zero frequency $n = 0$ is usually ignored as it just gives a constant.

For real h_k and complex H_n, H_n has twice the information of h_k. In other words, the output is twice redundant. For real h_k, only positive (or negative,

but not both) frequencies should be used. The function H_n is identical for positive and negative frequencies.

The time series can by specified by its beginning time, ending time, sampling frequency, sampling interval, and number of samples. The sampling frequency is the inverse of the sampling interval, and often, the beginning time of zero is implied. This leaves three parameters—of which two are independent—to specify the time series.

4.2 FAST-FOURIER TRANSFORM

The fast-Fourier transform (FFT) is one of the brilliant classic algorithms of numerical computing. It was first conceptualized by Gauss in 1805, then rediscovered a few times before being popularized by Cooley and Turkey in 1965 [17]. Here I just sketch out its key features since we will use library modules in practice.

The FFT algorithm makes use of the periodicity in the definition of the DFT and rearranges the order of the calculation to reduce the number of multiplications. Define $w = e^{2\pi i/N}$, so that the DFT becomes

$$H_n = \sum_{k=0}^{N-1} w^{nk} h_k \,. \tag{4.6}$$

Think of h_k as a vector and w^{nk} as a matrix. The calculation of all the H_n typically takes of $\mathcal{O}(N^2)$ multiplications—ignoring additions. We can write $H_n = H_n^e + w^n H_n^o$, where the sub-transforms are

$$\begin{aligned} H_n^e &= \sum_{k=0}^{N/2-1} e^{2\pi ink/(N/2)} h_{2k} \qquad \text{even terms}\,, \\ H_n^o &= \sum_{k=0}^{N/2-1} e^{2\pi ink/(N/2)} h_{2k+1} \qquad \text{odd terms}\,. \end{aligned} \tag{4.7}$$

The even- and odd-term DFTs each have half the data points which are separated twice as far apart. Conceptually, the algorithm divides the input data into two equal groups and transforms each group. It then divides each subgroup of data in half and transforms these four groups separately. The process is recursive until a single sample is reached, at which the Fourier transform is equal to the sample. The actual algorithm is the reverse of the procedure just described. Starting from a single sample combining them in pairs, recursively in fourths, eighths, etc., until all the data is used in a single transform.

The FFT algorithm also applies to the inverse DFT. Typically N should be a power of two in the algorithm to ensure the recursive splitting into even and odd terms. If this is not possible, it can be achieved by padding the time series with zero values up to the closest power of two. The library functions

do not enforce the power-of-two requirement but are most efficient for powers of two.

The DFT requires $\mathcal{O}(N^2)$ operations, while the FFT requires $\mathcal{O}(N \log_2 N)$ operations, a significant savings. For example, the FFT for $N = 10^3$ data points requires a computational time of about 10^{-2} less compared to the DFT, only getting better for larger N.

4.3 SOME NUMERICAL CONSIDERATION

Presentation of the results of Fourier transforms take some consideration, as they are in general complex. One could plot the real and imaginary parts separately. Plotting just the modulus can be useful, but the phase information will be lost; but perhaps the phase information is of no use in the problem. A common result to plot is the modulus squared, often called the power spectrum.

In some fields of physics, plotting a spectrogram which is a two-dimensional plot of frequency versus time is common. If the results are only periodic on short time scales, applying the FFT on sliding-window time domains can be an effective visualization. This is usually what an equalizer graph on sound equipment displays.

I now discuss some considerations which are equally applicable to FFT as DFT. Data points can correspond to more than one periodic function, particularly for sparse data. The time series is not unique to a particular periodic function, and this leads to issues of aliasing. Problem 4.6.3 has you work through an example of aliasing, and it will be illustrated in Sec. 4.3.1.

In addition, there can be spectral leakage between nearby frequency bins, which occurs when the signal contains components with periods that are not exactly divisible by the sampling period. A useful technique to mitigate some of these problems is windowing. That is, multiplying the signal by a function that reduces the signal to zero at the beginning and end of the sampling duration. Spectral leakage will be illustrated in Sec. 4.3.2.

Realistic data can include random noise and background and also suffer from resolution effects. Sometimes the data can be filtered to increase the signal-to-noise ratio. Some examples are band-pass filtering and convolution filters which make use of the FFT.

While it's not unheard of to code your own FFT, there is a nice set of Python library functions for calculating FFTs and the inverse FFTs:
`Y = numpy.fft.fft(y)` and `y = numpy.fft.ifft(Y)`.
There is also `scipy.fftpack.fft`. When not coding your own DFT (or FFT) be aware of the normalization definitions. Check the returned frequency format. Helper functions exist to make the frequencies more accessible. Please see the example code.

4.3.1 Aliasing

Sampling a signal with a finite time interval in the DFT limits the accuracy of the high-frequency components. A small time interval is need to sample the rapid variations of the high-frequency components. Inaccurate high-frequency components may contaminate low-frequency components in the signal.

Figure 4.1 shows an extreme example that illustrates the problem of sampling with a large time interval. Shown are two sine waves of frequency 0.8 and 0.2; the latter has a phase shift of π. When the sampling interval is one, the two data sets for these sine waves (dots in Fig. 4.1) are identical. The high-frequency component (0.8 frequency wave) has been aliased by the low-frequency component (0.2 frequency wave).

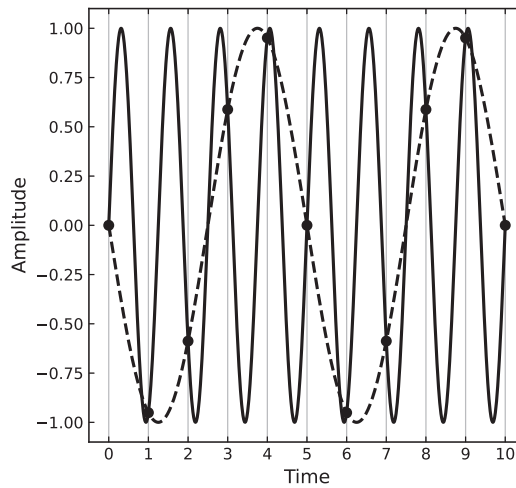


Figure 4.1 Example of aliasing. One sine wave (dashed line) has a frequency of 0.2 and a phase shift of π, and the other sine wave (solid line) has a frequency of 0.8 and no phase shift. For a sampling interval of one (vertical lines), the points sampled from each waveform (dots) are identical.

For other time intervals, in this example, some high-frequency values that are not included in the low-frequency waveform may be included in the sampling, but there may not be enough of them to separate the high-frequency components properly. Some of the high-frequency components may occur in the low-frequency spectrum as spurious frequency components.

Because of aliasing, there is an upper bound to how high a frequency we may resolve for a given sampling time interval Δ. This critical frequency is

called the Nyquist frequency:

$$f_c = \frac{1}{2\Delta}, \tag{4.8}$$

which for this example is $f_c = 0.5$. To avoid aliasing, no frequency greater than the Nyquist frequency can appear in the input signal. This is the Nyquist criterion. To satisfy the Nyquist criterion, we must increase the sampling frequency.

4.3.2 Spectral leakage

Spectral leakage occurs when frequencies other than the signal frequencies occur in the frequency spectrum. The leakage is due to the finite time duration used in sampling the signal. Signal components with frequencies that cannot fit into a full number of cycles in the sampling duration, and since DFT assumes the signal is periodic, cause discontinuity at the sampling duration boundary that can lead to spectral leakage.

Multiplying the signal with a window function reduces this problem. A window function is a function that when multiplied with the signal modulates its magnitude so that it approaches zero at the beginning and end of the sampling duration. The resulting signal can be viewed as a periodic function with smooth transitions between sampling duration boundaries, and as such, the DFT of the windowed signal can be better behaved.

I now attempt to explain spectral leakage and windowing using the example in Fig 4.2. A sine wave of unit amplitude and frequency of two is added to a sine wave of amplitude 1/5 and frequency of four as shown in Fig. 4.2a). This signal is meant to continue forever, but only the first 20 time units are shown; thus, 20 is representing infinity in this example. The power spectrum in Fig. 4.2b) clearly shows these fundamental frequencies and the correct power ratio.

In practice, we cannot sample the waveform over an infinite time duration and must choose a finite time duration: four units of time in this example. We are effectively multiplying the infinite waveform by a window function in time, one that is zero except during the sampling time duration for which it is unity. This rectangular-window function is shown in Fig. 4.2c) along with its power spectrum in Fig. 4.2d). The power spectrum has a peak (main lobe) with oscillatory lobes in the tails that decrease in amplitude and periodically are zero. The data are windowed by this rectangular window. By the convolution theorem, the Fourier transform of the product of the waveform with this rectangular window function is equal to the convolution of the waveform's Fourier transform with the window's Fourier transform. This is illustrated in Fig. 4.2e) which is the product of the waveform times the rectangular-window function, along with the power spectrum in Fig. 4.2f). Also shown is the non-windowed power spectrum as dotted lines. The reason for the leakage at frequencies different from the fundamentals is that the rectangular

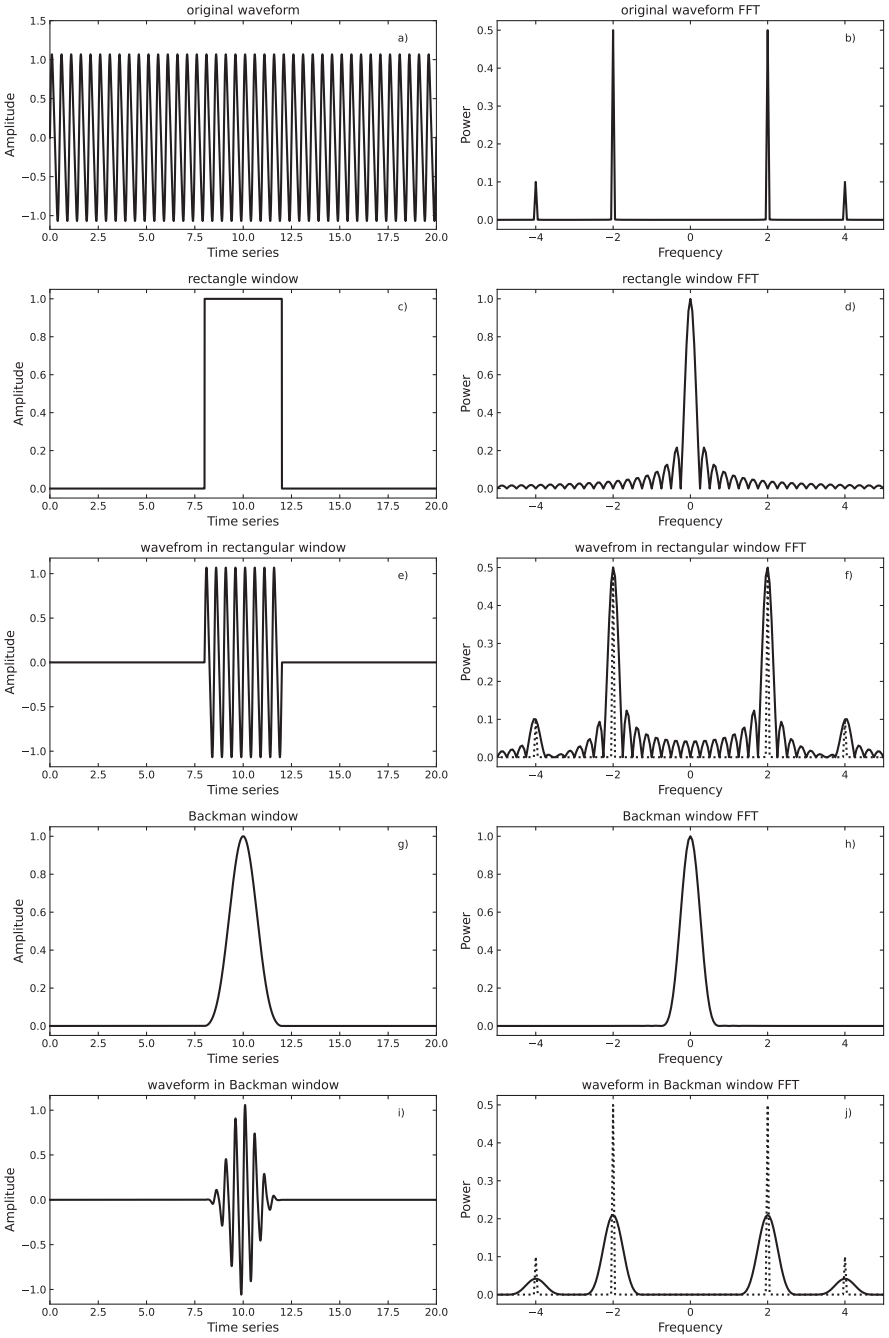


Figure 4.2 Spectral leakage and windowing. The time series are in the left column and their corresponding power spectrum in the right column.

window function turns on and off rapidly leading to its Fourier transform having substantial high-frequency components.

To remedy this situation, we can multiply the input waveform by a window function that changes more gradually from zero to a maximum and then back to zero as the time bins increase. There are many such window functions on the market, and it is not my business here to promote one or the other. I have arbitrarily picked the Blackman window function shown in Fig. 4.2g) and its power spectrum in Fig. 4.2h). The power spectrum tapers off quickly with no ringing. The price to pay is that the width of the main lobe has increased. Finally, Fig. 4.2i) and 4.2j) show the convolution of the signal waveform with the Blackman-window function and its power spectrum, respectively. There is considerable widening in the width and decrease in amplitude of the peaks using the Blackman window, but the oscillatory lobes are gone.

In the above demonstration, I have attempted to focus on the spectral leakage problem while being unrealistic in other aspects, such as using a very large sampling rate and, thus, number of samples. I also ensured an integer number of samples per cycle of the wave, and no aliasing.

Leakage can cause smaller signals to be lost in the leakage caused by larger signals. Windowing helps mitigate this issue. But, in general, there is a trade-off between attenuation of the side frequencies and the width of the main lobe. The best approach, if possible, is to increase the sampling time duration which gives a bigger improvement than implementing any windowing scheme.

4.4 DIFFERENTIATION WITH FOURIER TRANSFORM

Finite differences are not the only way to differentiate a function numerically. The Fourier transform—numerically the FFT—can be used. Consider differentiating the function $h(t)$, n times:

$$\frac{d^n}{dt^n}h(t) = \frac{d^n}{dt^n}\int_{-\infty}^{\infty} d\omega e^{-i\omega t}H(\omega) = \int_{-\infty}^{\infty} d\omega e^{-i\omega t}(-i\omega)^n H(\omega)\,. \tag{4.9}$$

In words, differentiating n times can be performed by taking the Fourier transform of the function to be differentiate, multiply by $(-i\omega)^n$ and then taking the inverse transform of the resulting product.

The same approach can be performed numerically using the FFT and its inverse. One must be aware that the maximum frequency magnitude is $\omega_{\max} = \pi/\Delta t$. The values of ω are in the range $[-\omega_{max}, \omega_{max})$ in n steps of $\Delta\omega = 2\omega_{\max}/N$, where $N = n + 1$ is the number of points, and n should be a power of two. You are asked to apply the FFT differentiation method in Problem 4.6.1.

The above differentiation explanation has been in terms of t and ω space; its also often—perhaps more often—performed in x and k space.

4.5 EXAMPLE: FOURIER TRANSFORM

In this example, we will calculate the power spectrum of some simple functions that we know the analytical solution for and can compare the numerical results with expectations. We first write a program that adds two sine waves of arbitrary amplitudes and periods and plots the power spectrum. I call the first function (sine wave) model 1, the second function (sine wave) model 2, and the sum of the two functions model 3.

Let's take a sampling frequency of 100 and a time range of $[0, 10]$. We will normalize the power to the sum of the squares of the amplitudes of each sample. For the first sine wave, let's take the amplitude to be 2 and a period of 0.1, and for the second sine wave an amplitude of 5 and period of 0.025.

The code below plots all three models at once for comparison. We plot the input time series on the left and the power spectrum on the right. Three different cases can be addressed depending on the value of the logical variables `mode0, mode1, mode2` (setting one, and only, of them to `True`) in a run. The three cases correspond to the original problem described above `mode0`, adding a constant to the sine waves `mode1`, and exponentially decaying the sine waves `mode2`. More details on the three cases are given after displaying the code output below.

```
"""FFT for two sine waves."""

import numpy as np
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

# Initialize ranges, sampling, etc.
samplingFrequency = 100
samplingInterval  = 1 / samplingFrequency;
tBegin =  0;
tEnd   = 10;
t = np.arange(tBegin,tEnd,samplingInterval);
print("time sampling frequency",samplingFrequency)
print("time sampling interval",samplingInterval)
print("time interval [%.1f,%.1f]"% (tBegin,tEnd))
print()

# Create two signal models.
T1 = 0.1
T2 = 0.025
A1 = 2
A2 = 5

# Pick question to answer.
mode0 = True
mode1 = False
mode2 = False
if mode0:
    offset1 = 0
    offset2 = 0
```

```
    # Arbitrary big number to effectively turn off exponential.
    tau = np.inf
if mode1:
    # Turn on offsets: question 3.
    offset1 = A1
    offset2 = A2
    # Arbitrary big number to effectively turn off exponential.
    tau = np.inf
if mode2:
    # Decay exponential: question 4.
    offset1 = A1
    offset2 = A2
    tau = 2
print("model 1: Amplitude %.1f period %.3f offset %.1f \
      time constant %.1f" % (A1,T1,offset1,tau))
print("model 2: Amplictue %.1f period %.3f offset %.1f \
      time constant %.1f" % (A2,T2,offset2,tau))
print()
model1 = offset1 + np.exp(-t/tau) * A1 * np.sin(2*np.pi*t/T1)
model2 = offset2 + np.exp(-t/tau) * A2 * np.sin(2*np.pi*t/T2)
model3 = model1 + model2

# Calculate normalizations.
norm1 = sum(model1**2)
norm2 = sum(model2**2)
norm3 = sum(model3**2)
print("power normalization %.1f %.1f %.1f" % (norm1,norm2,norm3))
print()

# Calculate FFT.
DFT1 = np.fft.rfft(model1)
DFT2 = np.fft.rfft(model2)
DFT3 = np.fft.rfft(model3)

# Get frequencies.
Tcount = model3.size
fValues = np.arange(1,int(Tcount/2)+1) # Remove 0 frequency.
tRange  = Tcount / samplingFrequency
f = fValues / tRange
print("number of time samples",Tcount)
print("number of frequency values",len(fValues))
print("frequency range [%.1f,%.1f]"% (f[0],f[-1]))

# Create the plots.
log = False # Flag to turn on log-scale plotting.
figure, axis = plt.subplots(3,2,figsize=(8,12))

# Time domain representation for model 1.
if (log):
    axis[0,0].set_yscale('log')
    axis[0,0].set_xscale('log')
axis[0,0].plot(t,model1,lw=0.5,c='grey')
axis[0,0].set_title('Model 1')
axis[0,0].set_xlabel('Time')
axis[0,0].set_ylabel('Amplitude')

# Time domain representation for model 2.
```

```
if (log):
    axis[1,0].set_yscale('log')
    axis[1,0].set_xscale('log')
axis[1,0].plot(t,model2,lw=0.5,c='grey')
axis[1,0].set_title('Model 2')
axis[1,0].set_xlabel('Time')
axis[1,0].set_ylabel('Amplitude')

# Time domain representation for model 2.
if (log):
    axis[2,0].set_yscale('log')
    axis[2,0].set_xscale('log')
axis[2,0].plot(t,model3,lw=0.5,c='grey')
axis[2,0].set_title('Model 3')
axis[2,0].set_xlabel('Time')
axis[2,0].set_ylabel('Amplitude')

# Frequency domain representation for model 1.
if (log): axis[0,1].set_yscale('log')
axis[0,1].plot(f,np.abs(DFT1[1:])**2/norm1,lw=2)
axis[0,1].set_title('Fourier transform of model 1')
axis[0,1].set_xlabel('Frequency')
axis[0,1].set_ylabel('Power')

# Frequency domain representation for model 2.
if (log): axis[1,1].set_yscale('log')
axis[1,1].plot(f,np.abs(DFT2[1:])**2/norm2,lw=2)
axis[1,1].set_title('Fourier transform of model 2')
axis[1,1].set_xlabel('Frequency')
axis[1,1].set_ylabel('Power')

# Frequency domain representation for model 3.
if (log): axis[2,1].set_yscale('log')
axis[2,1].plot(f,np.abs(DFT3[1:])**2/norm3,lw=2)
axis[2,1].set_title('Fourier transform of model 3')
axis[2,1].set_xlabel('Frequency')
axis[2,1].set_ylabel('Power')

plt.tight_layout()
if mode0: plt.savefig('4_FFT_1.pdf')
if mode1: plt.savefig('4_FFT_2.pdf')
if mode2: plt.savefig('4_FFT_3.pdf')
plt.show()
```

Let's now determine our expectations for the power spectra. The Fourier transform of a sine wave of amplitude A and frequency f_0 is

$$H(f) = NA\frac{\delta(f - f_0)}{2i}\,, \tag{4.10}$$

where N is the number of samples. The power spectrum is

$$|H(f)|^2 = \left(\frac{NA}{2}\right)^2 \delta(f - f_0)\,. \tag{4.11}$$

The normalization is given by

$$\text{Norm} = NA^2\langle\sin^2\omega t\rangle = N\frac{A^2}{2}\,. \tag{4.12}$$

The normalized power spectrum is

$$\frac{|H(f)|^2}{\text{Norm}} = \frac{N}{2}\delta(f - f_0)\,. \tag{4.13}$$

For the three models, the expected power spectrum frequency normalization and normalized amplitudes for $N = 1000$ are

Model	T_0	f_0	Norm	Amplitude
1	0.1	$1/0.1 = 10$	$1000 \times 2^2/2 = 2000$	$(1000 \times 2/2)^2/2000 = 500$
2	0.025	$1/0.025 = 40$	$1000 \times 5^2/2 = 12500$	$(1000 \times 5/2)^2/12500 = 500$
3	0.1, 0.025	10, 40	$2000 + 12500 = 14500$	$(1000 \times 2/2)^2/14500 = 69.0$ $(1000 \times 5/2)^2/14500 = 431.0$

When setting `mode = 0` in the code, the plots show that the power spectra for all three models are as expected.

Now we add a constant 2 to the first sine wave and a constant 5 to the second sine wave. Our expectations are that a constant offset does not change the frequencies but the normalization changes. The normalization of the single sine wave becomes

$$\text{Norm} = N(A^2\langle\sin^2\omega t\rangle + A^2) = \frac{3}{2}A^2N\,. \tag{4.14}$$

For the two sine waves

$$\begin{aligned}\text{Norm} &= N\left(A_1^2\langle\sin^2\omega_1 t\rangle + A_2^2\langle\sin^2\omega_2 t\rangle + (A_1 + A_2)^2\right)\\ &= \left[\frac{3}{2}(A_1^2 + A_2^2) + 2A_1A_2\right]N\,.\end{aligned} \tag{4.15}$$

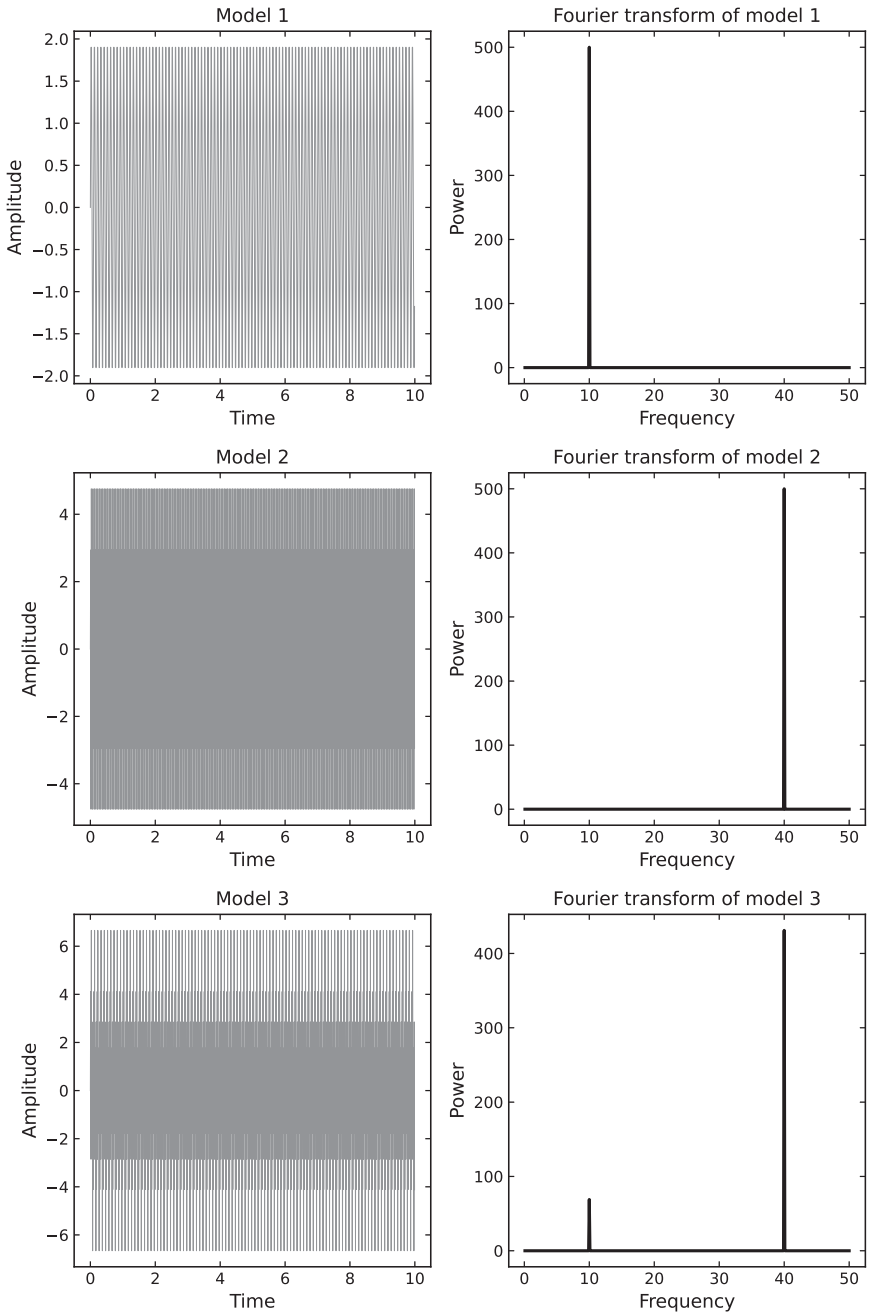
The new power spectrum normalization and amplitudes are

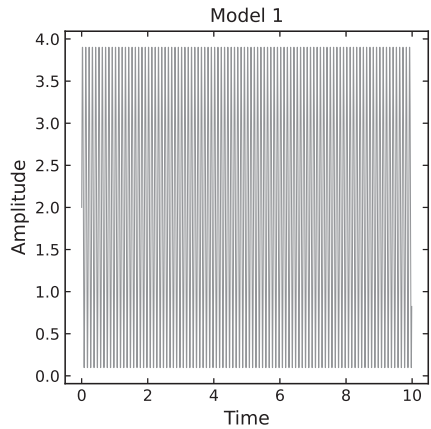
Model	Norm	Amplitude
1	6000	$(1000 \times 2/2)^2/3000 = 167$
2	37500	$(1000 \times 5/2)^2/37500 = 167$
3	63500	$(1000 \times 2/2)^2/63500 = 15.7$, $(1000 \times 5/2)^2/63500 = 98.4$

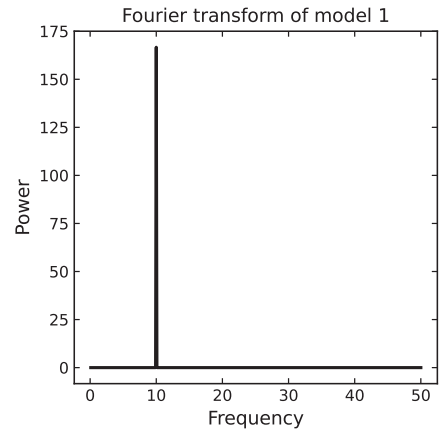
Note that for a constant offset, the power spectrum amplitudes without normalization do not change.

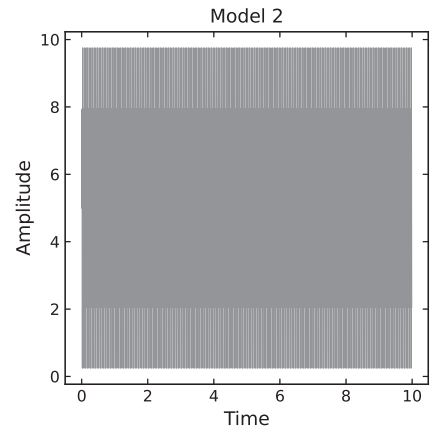
When setting `mode = 1` in the code, the plots show the power spectra for all three models are as expected.

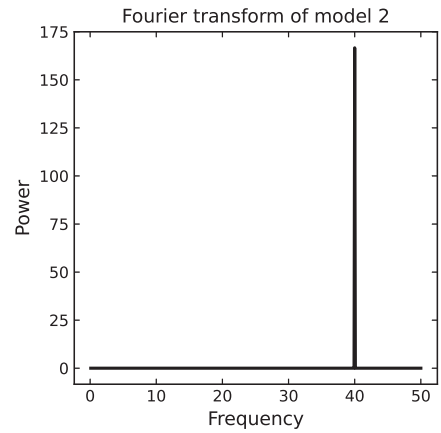
Finally, we will simulate an exponentially decaying amplitude by multiplying the sine wave term by an exponential with a time constant of 2. When including the exponential decay, the amplitudes will decrease and the frequencies will have wider distributions. The exponential Fourier transforms as a Cauchy function. The exponential times sine wave in the time space gives a convolution of a delta function with a Cauchy function in frequency space. Thus the delta-function spikes are smoothed out. When setting `mode = 2` in the code, the plots show the power spectra for all three models are as expected.

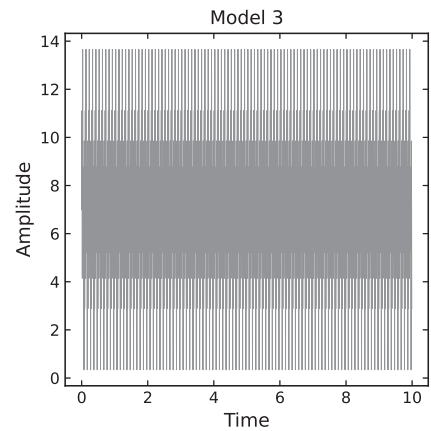

Model 1
Amplitude
Time
Fourier transform of model 1
Power
Frequency
Model 2
Amplitude
Time
Fourier transform of model 2
Power
Frequency
Model 3
Amplitude
Time
Fourier transform of model 3
Power
Frequency

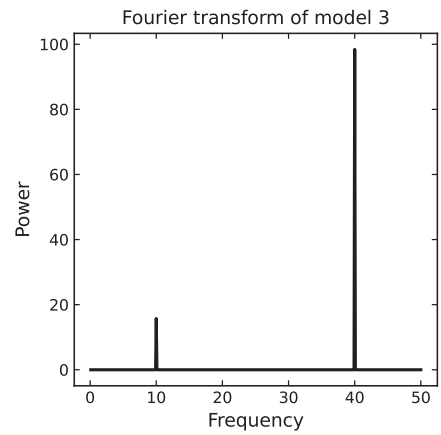

Model 1
Amplitude
Time
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
2
4
6
8
10

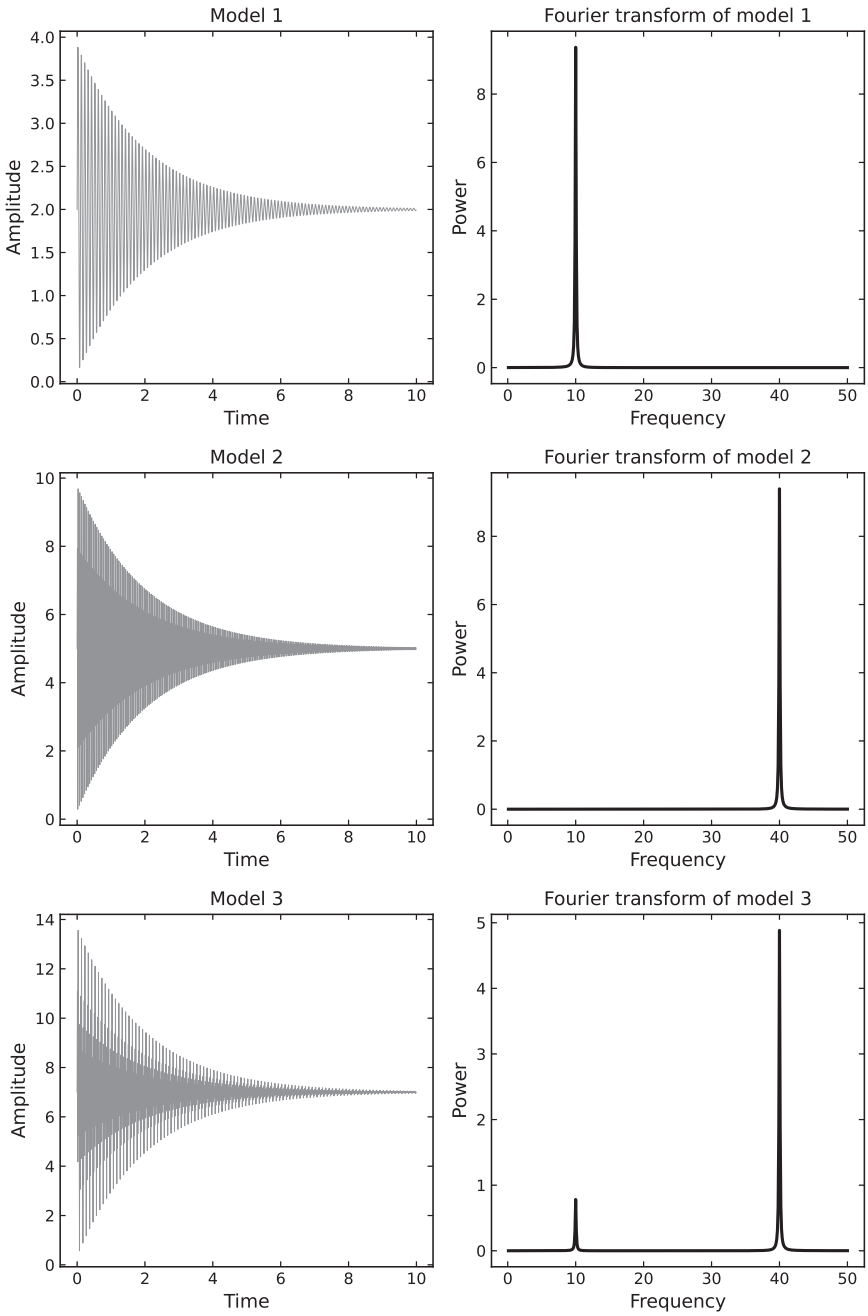

Fourier transform of model 1
Power
Frequency
0
25
50
75
100
125
150
175
0
10
20
30
40
50


Model 2
Amplitude
Time
0
2
4
6
8
10
0
2
4
6
8
10


Fourier transform of model 2
Power
Frequency
0
25
50
75
100
125
150
175
0
10
20
30
40
50


Model 3
Amplitude
Time
0
2
4
6
8
10
12
14
0
2
4
6
8
10


Fourier transform of model 3
Power
Frequency
0
20
40
60
80
100
0
10
20
30
40
50


Model 1
Amplitude
Time
Fourier transform of model 1
Power
Frequency
Model 2
Amplitude
Time
Fourier transform of model 2
Power
Frequency
Model 3
Amplitude
Time
Fourier transform of model 3
Power
Frequency

4.6 PROBLEMS

1. **Schrödinger kinetic energy operator**

   Repeat problem 3.3.22 but this time calculate the kinetic energy operator using the fast-Fourier transform.

2. **Fourier transform: convolution**

   The Fourier transform (FT) of the convolution of two functions is the product of their corresponding Fourier transforms. Consider a periodic signal that is smeared by measurement resolution effects. Simplistically the resolution function can be thought of as a Gaussian distribution. So the signal is convoluted with a Gaussian. However, sometimes things are not so ideal, and the tails of the resolution function are bigger than Gaussian tails. One way to model this resolution function is to use a crystal-ball function. A crystal-ball function is a Gaussian core changing to exponential tails at some junction values. The function and its first derivative are both continuous where the Gaussian and exponentials match at the junction values.

   Consider the convolution of a signal-pulse train with a crystal-ball resolution function. The FT of a pulse train in the time domain is a pulse train in the frequency domain. Use `scipy.signal.unit_impulse` which is a unit impulse signal—or discrete delta function. The FT of a Gaussian in the time domain is a Gaussian in the frequency domain. The FT of an exponential in the time domain is a Lorentzian function in the frequency domain. A Lorentzian function is also called a Cauchy function or Breit-Wigner function. Thus the FT of a crystal ball should approximate a Gaussian with a width and tails that are not exactly Gaussian; it is not a Voigt profile. The problem is thus, in principle, analytically calculable.

   (a) Convolute a signal-pulse train with a crystal-ball resolution function.

   (b) Then take the FFT of the signal pulse train, crystal ball function, and the convolution separately.

   (c) To test the Fourier transform convolution theorem, multiply the FFT of the pulse train by the FFT of the crystal-ball function, and compare with the FFT of the convolution.

3. **Fourier transform: Aliasing**

   (a) Create four different sine-wave models with frequency f and phase ϕ as in the following table. The numerical values used in this problem were inspired by Ref. [7].

   (b) Sample the first three models with 50 data points and a sampling interval of 1 starting at 0.

Model	f	ϕ
1	0.8	0
2	0.2	0
3	0.2123	0
4	0.2	π

(c) For the first two models, plot the time series (two plots) and the real and imaginary parts of the discrete Fourier transforms (two more plots, each with real and imaginary curves labelled).

(d) Comment on all four plots. Also point out any observations that might be counterintuitive to the idealized expectations of the transforms. Can you explain what you see in your numerical transform?

(e) On a separate plot, plot model 1 and 4 with 1000 data points over the range $[0, 10]$.

(f) Are you able to use this plot to help explain what you see in (c)?

(g) For model 3, plot the time series (one plot) and the real and imaginary parts of the discrete Fourier transforms (one more plot, with real and imaginary curves labelled), and a third plot of the power spectrum.

(h) Comment on all three plots in (g). Can you explain what you see in your numerical transform?

In the end, you should be showing a total of eight plots.

4. **Fourier transform with noise**

(a) Write a program that generates a vector of 1024 standard Gaussian distributed random values, computes the discrete Fourier transform of it using the FFT, and plots the time series and power spectrum. We will call this spectrum white noise. We have not yet discussed randomly distributed variables. So you may use `numpy.random.normal` or `scipy.stats.norm` to obtain Gaussian-distributed random values. Note: you should normalize your power spectrum so it's independent of the number of time-series samples. Please normalize the FFT to $1/\sqrt{N}$.

(b) Assemble a time series consisting of white noise plus a sinusoid of amplitude A and a period of 20 data points. Compute the power spectrum for an arbitrary value of A. Plot the time series and power spectrum for the sinusoid only and sinusoid plus white noise: you are now showing six plots in total. State and justify your choice of frequency values?

(c) Calculate the mean and standard deviation of the white-noise power spectrum. You may use library functions if you like.

(d) It can be shown that a good criterion for a significant signal in a power spectrum above noise with a mean of one and a standard deviation of one in the power spectrum is about 10. Determine the value of A that gives this power amplitude of 10.

You may use `numpy.fft` or `scpy.fftpack`. Please don't code the discrete FT or FFT yourself.

5. **Average and difference filters**

Averaging a waveform over adjacent sampling times smooths the waveform. Large changes in the waveform within a sampling interval will be lost. Averaging thus suppresses the high-frequency component in the waveform, acting like a low-pass filter. Conversely, taking a difference of the waveform between adjacent sampling times suppresses small changes within a sampling interval, acting like a high-pass filter.

(a) Write a program that accepts a signal composed of a sum of four sine waves of various frequencies and plots the power spectrum.

(b) Now average and take the difference of the time series as follows

$$Y_i^{\mathrm{A}} = \frac{1}{2}(y_i + y_{i+1}) \quad \text{and}$$
$$Y_i^{\mathrm{D}} = \frac{1}{2}(y_i - y_{i+1}) \quad \text{for } i = 0, \ldots, N\,,$$

where $y_N = y_0$, i.e. the time series wraps around.

(c) Plot the power spectrum of the resulting two filters. Choose sine-wave amplitudes and frequencies that clearly demonstrate the filtering process.

(d) State the amplitudes and periods used for your four sine waves. State the sampling frequency, sampling interval, sampling range, number of samples, number of frequency values, and frequency range. Also state any normalization you might have used, including those used by any library modules.

You may use the FFT libraries in NumPy or SciPy.

6. **Gravitational ringdown**

(a) Read in the data from file ringdown.txt which contains two NumPy arrays: `(t,y) = numpy.loadtxt("ringdown.txt")`

(b) Plot all the data using a log y-axis (linear x-axis). The x-axis is time.

Globally, the data decreases with increasing time. Locally, the data is quasi-periodic with a late-time tail which shows power-law behavior.

(c) Calculate the FFT and plot the power spectrum for the data. The normalization is not important, as we are only interested in determining the fundamental frequency. You may use library functions for the FFT.

Since the data has a variable oscillation period but looks constant in a central time region, exclude the low-time and high-time data regions in your DFT to improve the quality of your frequencies.

(d) Indicate on the plot with vertical lines your FFT time range.

(e) Determine the fundamental frequency in units of inverse time.

The low frequencies corresponding to the falling continuum background are not of interest even though the power may be highest at low frequencies. Likewise, we are not interested in the high frequencies corresponding to fluctuations. After ignoring low and high frequencies and estimating the fundamental frequency by eye on the time plot, you should see a unique peak in the power spectrum. Congratulations, you have calculated the fundamental frequency of a Schwarzschild black hole after it has been gravitationally perturbed.

CHAPTER 5

Monte Carlo methods

When we speak of Monte Carlo, we refer to a large group of numerical methods for solving physics problems by means of random sampling.[1] In particular, the Monte Carlo method allows us to simulate any process whose development is affected by random factors, and many mathematical problems not affected by any random influences can be connected to the Monte Carlo method by an artificially constructed probabilistic model.

The Monte Carlo method has at least two features. The first is the simple structure of a computational algorithm. For example, a program is prepared to perform one random trial. The trial is repeated N times, and the results of all trials are analyzed. The second feature is the uncertainty, which as a rule is proportional to $1/\sqrt{N}$. The Monte Carlo method is thus especially efficient in solving those problems which do not require a high degree of accuracy.

Since Monte Carlo methods are stochastic processes, we first review the required probability and statistics topics. We then develop the basic techniques. These basic techniques are used to perform integration and sample distributions using the basic methods of uniform, importance, and stratified sampling. Finally, I introduced the method of Markov chain Monte Carlo.

5.1 BASIC PROBABILITY AND STATISTICS

Before beginning the discussion of Monte Carlo methods in numerical computing, I feel we must review some concepts of probability and statistics. This background is essential for understanding the Monte Carlo methods and will also serve us well in Chapter 6.

5.1.1 Probability

Some of this section will be review for some students. But since not all students have the same level of background or a formal course in statistics, I introduce

[1] Never use the words Monte Carlo as a noun. Then you would be referring to an administrative area of the principality of Monaco.

DOI: 10.1201/9781003660873-5

the concepts that we will make use of. Advancements in computing technology have enabled the practical use of a high level of statistics which would have been considered academic and beyond practical computation only a decade ago.

A random variable is the outcome of a repeatable experiment. Note, this is frequentist interpretation. The idea is that all the repeatable experiments are identical. Practically, we usually do not repeat an experiment many times. At best, we have a few groups of scientists attempt to measure the same physical process with less than identical experiments—often complementary experiments. Nevertheless, computers allow the concept of identical pseudo-experiments. A probabilistic model can be simulated with the run of some program. The random numbers can be changed and the code can be run again. Each run of the code is modelling an experiment. We thus build up the statistical distribution of outcomes by running many pseudo-experiments, sometimes called toys.

Let x be the outcome of an observation. If x is continuous, $p(x;\theta)dx$ is the probability that x is between x and $x + dx$. The probability density function (pdf) is $p(x;\theta)$, and θ are parameters of the function. If x is discrete, $p(x;\theta)$ is the probability to get x. A pdf is always normalized to unity. The arguments of the function $(x;\theta)$ can have multiple components. Often $p(x;\theta)$ is unknown and must be determined. If the parameters θ are unknown, θ can be estimated using statistical methods.

The cumulative distribution function (CDF) is the probability of $x \leq a$:

$$P(a) = \int_{-\infty}^{a} dx\, p(x) \quad \text{and} \quad 0 \leq P(a) \leq 1\,. \tag{5.1}$$

$P(a)$ is non-decreasing because $p(x)$ is positive. The probability of x in $a < x < b$ can be written in terms of cumulative distributions:

$$\begin{aligned}\int_{a}^{b} dx\, p(x) &= 1 - \int_{-\infty}^{a} dx\, p(x) - \int_{b}^{\infty} dx\, p(x) \\ &= \int_{-\infty}^{b} dx\, p(x) - \int_{-\infty}^{a} dx\, p(x) \\ &= P(b) - P(a)\,. \end{aligned} \tag{5.2}$$

Any function u of a random variable x, $u(x)$, is also a random variable but with a different pdf. This is a key concept of which we will make extensive use.

The expectation value of $u(x)$ is

$$E[u(x)] \equiv \int_{-\infty}^{\infty} dx\, u(x) p(x)\,. \tag{5.3}$$

The nth moments of x are

$$\alpha_n \equiv E[x^n] = \int_{-\infty}^{\infty} dx\, x^n p(x)\,. \tag{5.4}$$

A common moment is the mean (first moment),

$$\mu \equiv \alpha_1 = \int_{-\infty}^{\infty} dx x p(x)\,. \tag{5.5}$$

The nth central moment of x (moment about mean α_1) is

$$m_n \equiv E[(x-\alpha_1)^n] = \int_{-\infty}^{\infty} dx (x-\alpha_1)^n p(x)\,. \tag{5.6}$$

A common case is the variance (second central moment),

$$\begin{aligned}\sigma^2 \equiv V[x] \equiv m_2 &= \int_{-\infty}^{\infty} dx (x-\alpha_1)^2 p(x) \\ &= \int_{-\infty}^{\infty} dx\, x^2 p(x) - 2\alpha_1 \int_{-\infty}^{\infty} dx\, x p(x) + \alpha_1^2 \int_{-\infty}^{\infty} dx\, p(x) \\ &= \alpha_2 - 2\alpha_1^2 + \alpha_1^2 = \alpha_2 - \mu^2\ .\end{aligned} \tag{5.7}$$

The standard deviation is σ.

We will make use of a few other probability definitions. The quantile x_α is the value of x at which the cumulative distribution is α:

$$\alpha = P(x_\alpha) = \int_{-\infty}^{x_\alpha} dx p(x)\,. \tag{5.8}$$

The inverse of the cumulative distribution is $x_\alpha = P^{-1}(\alpha)$. A special case, $P(x_{\rm med}) = 1/2$ defines the median.

5.1.2 Probability distributions

I will now switch to a different notation which does not confuse probability p with p-values, defined soon. Consider $f(x;\boldsymbol{\theta})$ as a probability distribution function. The function is normalized to unity by definition, and $\boldsymbol{\theta}$ is a possible set of parameters—known or unknown. We will consider the case of both discrete and continuous probability. Moments can be defined for each distribution. The cumulative distribution $F(x <) = F(x;\boldsymbol{\theta})$ is the probability of a value $\leq x$. The survival function is $1 - F(x <)$, but we will not use it. Also, we can set $p = F(x;\boldsymbol{\theta})$ and take the inverse. The inverse, or quantile, is $F^{-1}(p;\boldsymbol{\theta})$. This is referred to as the percent point function in `scipy.stats`. The p-value p is the probability of $\leq x$ in the interval $[0 < p < 1]$. Later we will see that the significance of a hypothesis $f(x;\boldsymbol{\theta})$ of x has a high p-value if the hypothesis is probable and a low p-value if unlikely.

The quantile divides the total range of probability into intervals of equal probability. The two-quantile is the median (50% division). The 100-quantiles are percentiles. The quantiles 0.5, 0.84, 0.16, 0.98, and 0.02 are used for the mean, $+\sigma$, $-\sigma$, $+2\sigma$, and -2σ, respectively, uncertainties (two sided). Quantiles 0.1, 0.05, and 0.01 are used for hypothesis testing and represent the 90%, 95%, and 99%, respectively, confidence level limits (CL) on the hypothesis.

We now discuss six common probability distributions that we will encounter later. There are, of course, many more well-studied probability distributions, and which are most common for you depends on your area of physics.

5.1.2.1 Uniform distribution

The uniform distribution is fundamental to our discussion since it typically is the distribution of the random numbers we use in our code. The uniform distribution on the interval $[a, b]$ is

$$f(x; a, b) = \begin{cases} 1/(b-a); & a \le x \le b \\ 0; & \text{otherwise} \end{cases} \tag{5.9}$$

Based on our previous discussion of moments, you should be able to show that the

$$\text{mean is} \quad \mu = \frac{a+b}{2} \quad \text{and variance is} \quad \sigma^2 = \frac{(b-a)^2}{12}\,. \tag{5.10}$$

5.1.2.2 Binomial distribution

The binomial distribution describes random processes with exactly two outcomes (discrete):

$$f(r; N, p) = \frac{N!}{r!(N-r)!} p^r q^{N-r}\,, \tag{5.11}$$

where p is the probability of success and $q = 1-p$ is the probability of failure. The number of independent trials is N, and r is number of successes. The distribution describes the probability to obtain exactly r successes each of probability p in N independent trials, without regard for order. The mean is $\mu = Np$ and the variance is $\sigma^2 = Npq$. The binomial distribution can be generalized to a multinomial distribution which can be used to describe the distribution of entries across bins in a histogram, for example.

5.1.2.3 Poisson distribution

The binomial distribution in the limit of $p \to 0, N \to \infty, Np = \nu$ (discrete) is a Poisson distribution:

$$f(n; \nu) = \frac{\nu^n e^{-\nu}}{n!}\,; \quad n \ge 0, \nu > 0\,. \tag{5.12}$$

The Poisson distribution is the probability of finding exactly n events when the events occur independent of one another at an average rate ν per a given interval (space or time). The distribution is most commonly used in counting experiments to describe the fluctuations in a bin of histogrammed data. The mean is $\mu = \nu$ and the variance is $\sigma^2 = \nu$. The standard deviation is $\sigma = \sqrt{\nu}$.

5.1.2.4 Normal (Gaussian) distribution

The normal distribution is the Poisson distribution in the limit of large ν:

$$f(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right], \tag{5.13}$$

where $-\infty < x < \infty$, $-\infty < \mu < \infty$, and $\sigma > 0$. Note that the normalization is not a constant and depends on σ. The parameter μ can be thought of as a translation in x. The mean is μ and the variance is σ^2. The cumulative distribution $F(x<) = F(x;\mu,\sigma)$ is

$$F(x;\mu,\sigma) = \frac{1}{2}\left\{1 + \text{erf}\left[-\frac{1}{\sqrt{2}}\left(\frac{x-\mu}{\sigma}\right)\right]\right\}. \tag{5.14}$$

The inverse (quantile) is

$$F^{-1}(p;\mu,\sigma) = \mu + \sigma\,\text{erf}^{-1}(2p-1), \tag{5.15}$$

where $p = [0,1]$ or $2p-1 = [-1,+1]$.

5.1.2.5 Exponential distribution

The exponential probability distribution is

$$f(x;\beta) = \beta\exp(-\beta x); \quad x \geq 0,\ \beta > 0. \tag{5.16}$$

The mean is $\mu = 1/\beta$ and the variance is $\sigma^2 = 1/\beta^2$. The cumulative distribution $F(x<) = F(x;\beta)$ is

$$F(x;\beta) = 1 - \exp(-\beta x). \tag{5.17}$$

The inverse (quantile) is

$$F^{-1}(p;\beta) = -\frac{1}{\beta}\ln(1-p), \tag{5.18}$$

where $p = [0,1]$ and $1-p = [0,1]$.

5.1.2.6 Chi-squared distribution

The χ^2-probability distribution is

$$f(z;n) = \frac{z^{n/2}e^{-z/2}}{2^{n/2}\Gamma(n/2)}; \quad z \equiv \chi^2 > 0, n > 0, \tag{5.19}$$

where for the cases we are concerned with, the χ^2-function is

$$z = \sum_{i=1}^{n}\frac{(x_i-\mu_i)^2}{\sigma_i^2}. \tag{5.20}$$

For $x_1, \ldots, x_n$ independent Gaussian random variables, $f(z; n) = \chi^2$ is a pdf with n degrees of freedom. The χ^2-pdf is often used to evaluate the level of compatibility between observed data and a hypothesis for the pdf that the data might follow. The mean is $\mu = n$ and the variance is $\sigma^2 = 2n$.
The cumulative distribution is the incomplete gamma function $P(n/2, z/2)$. Sometimes the complement of P is used:

$$Q(z|n) \equiv 1 - P(z|n) = 1 - P\left(\frac{n}{2}, \frac{z}{2}\right) \equiv Q\left(\frac{n}{2}, \frac{z}{2}\right) . \tag{5.21}$$

The inverse (quantile) is

$$F^{-1}(p; n) = 2P^{-1}\left(\frac{n}{2}, p\right) . \tag{5.22}$$

Note that the same symbol, χ^2, is used for both the pdf and the function in its exponential.

If the range in x of the continuous distributions described above is changed, the distributions must be renormalized to unity. Figure 5.1 shows three example probability distributions.

5.1.2.7 Non-analytic distributions

Probability distributions are not limited to analytic functions; we can form a probability distribution based on a set of data. Consider the graph (x_i, y_i), where $i = 0, \ldots, N-1$. If the spacing between the points is small, we can think of the data as just function evaluations at discrete points. We can histogram the data to have n_j entries in bin j for $j = 0, \ldots, M-1$. If the bin width is a small $\Delta x_j, n_j/\Delta x_j \to (dn/dx)_j$ and we recover the function.

We can fit the data to a model (analytic function) or interpolate, and we can integrate by summing. We can also normalize and calculate cumulative distributions. We can invert $y_i(x_i) \to x_i(y_i)$ by changing $(x_i, y_i) \to (y_i, x_i)$. The mean and variance can be calculated to gain some understanding of the underlying probability distribution.

Histograms give a sense of the density of the data and are often used for density estimation, estimating the probability density function of the underlying variable. To construct a histogram, the first step is to bin the range of values—that is, divide the entire range of values into a series of smaller intervals—and then count how many values fall into each interval (bin). A rectangle is drawn with height proportional to the count and width equal to the bin size so that the rectangles abut each other. The total area of a histogram used for probability density is always normalized to unity. It then shows the proportion of cases that fall into each of several categories, with the sum of heights times bin width equal to one. Sometimes it is useful to think of a histogram as the derivative of a plot or function. Histograms should not be confused with bar charts.

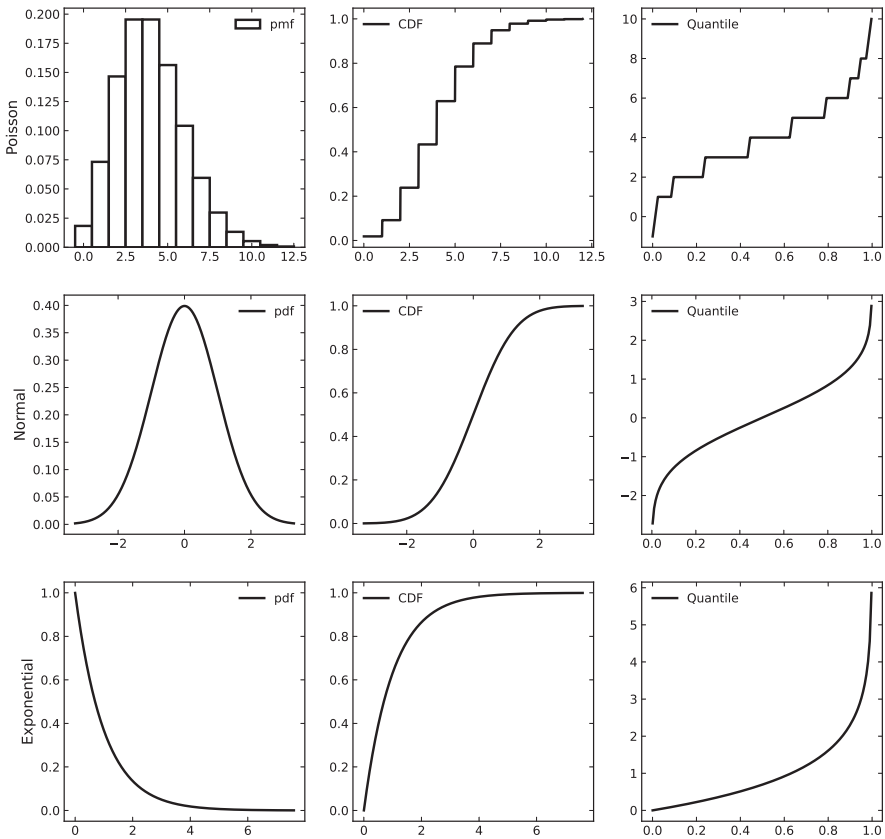


Figure 5.1 Example probability distribution functions (left column), cumulative distribution functions (middle column), and quantile (right column) for a Poisson distribution (top row), a normal distribution (middle row), and an exponential distribution (bottom row).

5.2 MONTE CARLO TECHNIQUES

Monte Carlo (MC) methods are important in numerical computing in physics and in some subfields are arguably the most important. MC methods technically allow the evaluation of difficult integrals. The methods are usually superior for multidimensional integration. Sampling random variables from a pdf falls under the topic of MC methods. MC methods are essential for the simulation of stochastic processes, for example, quantum mechanics.

5.2.1 Sampling uniform distributions

Assume a uniform random number generator exists. This function generates uniform statistically independent values u in the open interval $u = [0, 1)$. Do not use random number generators blindly. If in doubt, histogram the distribution of random variables.

The only thing we may do is seed the generator.

1. Set the seed once before generation.
2. Set the seed to a constant to reproduce the same random sequence for each run of your code. This is great for debugging, as it provides reproducibility.
3. When making multiple runs of your code, use the current system time to avoid reproducibility. This is the Python default if the seed is not set by the coder.

If you desire a different range and the generator does not provide uniform random values in this range, you can make the range transformation $[a, b) \rightarrow (b - a)u + a$. Also note that $(1 - u)$ is in the same $[0, 1)$ interval.

5.2.2 Random variable generation in Python

A wealth of random variable generators are available to Python. The three common ones are `random`, `numpy.random` and `scipy.stats`. Other specialized packages exist. In principle, given a uniform random number generator, any random variable distribution can be generated. As we will below. For simple random number generation, I recommend `random` or `numpy.random`. I will use both—not at the same time—in what follows. If one's code does not import NumPy, I would suggest using the Python native generator `random`; else, use `numpy.random`. Like many NumPy libraries, they can be called as functions or classes. In what follows, I will simply call the random generator functions. A more modern user should probably instantiate the Generator object using `numpy.random.default_rng()`.

5.2.3 Random variables and integration

Given a uniform random number, there are typically two methods to generate a general random variable that follows the distribution $f(x)$.

5.2.3.1 Inverse transformation method

Consider the probability density function $f(x)$ on the interval $-\infty < x < \infty$. If a is chosen with probability $f(a)$, then the integrated probability up to point a, $F(a)$, is itself a random variable which will occur with uniform probability

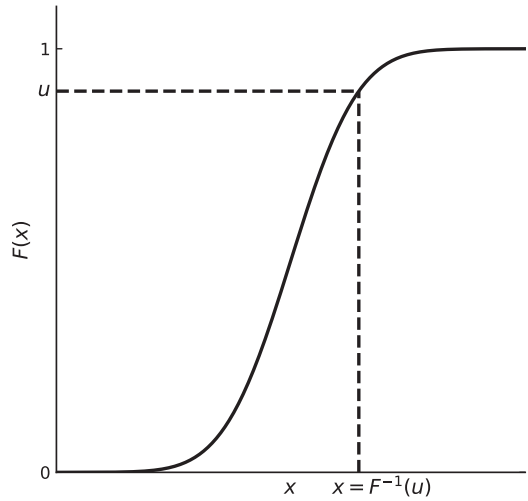


Figure 5.2 Cumulative distribution.

density on $[0, 1)$. Then $u = F(x) \Rightarrow x = F^{-1}(u)$. This is shown in Fig. 5.2 for the continuous case.

If the above does not make sense to you, I'll illustrate the method in detail. We use two related random variables; call them u and v. Variable u is uniformly distributed in [0,1). It is often called a uniform deviate. Variable v is related to u through a one-to-one functional relation. If we take a particular sample value drawn from the uniform deviate u, there is a corresponding value v. The fraction of drawn values that are in a particular u-element du is equal to the fraction of values that are in the corresponding v-element dv. Those fraction are $p_u(u)du$ and $p_v(v)dv$, respectively, where p_u and p_v are the respective probability distributions of u and v. We have

$$p_u(u)du = p_v(v)dv \quad \Rightarrow \quad p_v(v) = p_u(u)\left|\frac{du}{dv}\right| = \left|\frac{du}{dv}\right|, \tag{5.23}$$

where $p_u = 1$ has been used in the last expression for a uniform deviate. If we are required to find random values v from a probability distribution p_v, we simply have to solve $p_v(v) = |du/dv|$. Using the cumulative probability $P_v(v) = \int^v p_v(v')dv'$, we may write $u = P_v(v)$, which can be inverted to give $v = P_v^{-1}(u)$.

It is not always possible to invert the function analytically but is always possible to do it numerically. This can be done by root finding for a set of evenly spaced u values and using interpolation afterward. The inverse transform method is useful for analytic functions that can be inverted. Python libraries modules of pdfs, cumulative pdfs, and inverse (quantile) pdfs exist. Histogramming x produces the required random distribution. The

distribution can also be calculated numerically when $p(x)$ is not an analytic function. If the pdf is not used over the range $[-\infty, \infty]$, renormalize it over the range $[a, b]$.

5.2.3.2 Acceptance-rejection method

In this method (due to von Neumann), we want to generate a distribution according to $f(x)$, which need not be normalized. Consider $a < x < b$ and $0 < f(x) < f_{\max} \equiv C$. The algorithm consists of

1. generate random $(b-a)u_1 + a$ and calculate $f\left((b-a)u_1 + a\right)$,
2. generate random Cu_2.
3. if $Cu_2 < f\left((b-a)u_1 + a\right)$, accept it, else reject it,
4. repeat as necessary,

where u_1 and u_2 are two uniform random variables on $[0, 1)$. The accepted functional values can be used as random variables distributed according to $f(x)$. The method is also known as the accept-reject method or hit-and-miss method.

If the range or function maximum is unknown, use a wider range and higher $f_{\max}$. The efficiency of hits to trials will just be lower. Figure 5.3 shows the choice of the most efficient single maximum (dashed box) and how multiple maximums (left dashed and right dash-dotted boxes) can be used in different ranges to gain further efficiency.

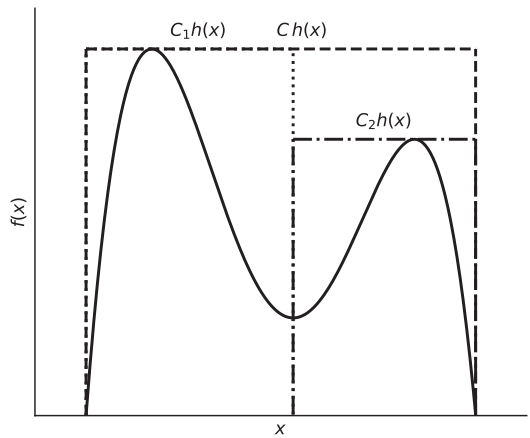


Figure 5.3 Hit and miss method.

One can also use this method to integrate the function $f(x)$. As a concrete example, consider $0 \leq f(x) \leq 1$ for $0 \leq x \leq 1$. Generate uniform random

numbers u_1 and u_2. Increment N. If $u_2 \leq f(u_1)$, increment m. The integral of $f(x)$ is

$$\int_0^1 f(x)dx = \lim_{N\to\infty} \left(\frac{m}{N}\right) . \tag{5.24}$$

The method is easily generalized to higher dimensions and will be discussed further in subsequent sections.

5.2.3.3 Efficiency statistical uncertainty

Since the concept of efficiency often plays a role in simulation and data analysis, I want to spend some time discussing its uncertainty. In fact, we have already seen an efficiency calculation in Eq. (5.24). Besides wanting to know the efficiency of the integration algorithm, we should also have an estimate of its accuracy.

For m accepted events from N tries, the efficiency is

$$\epsilon = \frac{m}{N} \equiv p\,. \tag{5.25}$$

If m and N are uncorrelated, propagation of errors gives

$$\frac{(\delta\epsilon)^2}{\epsilon^2} = \left(\frac{\delta m}{m}\right)^2 + \left(\frac{\delta N}{N}\right)^2 . \tag{5.26}$$

A typical estimate of the uncertainty is to take $\delta N = \sqrt{N}$ and ignore δm:

$$\frac{\delta\epsilon}{\epsilon} = \frac{1}{\sqrt{N}}\,. \tag{5.27}$$

Or conversely, to view N as fixed and take $\delta m = \sqrt{m}$:

$$\frac{\delta\epsilon}{\epsilon} = \frac{1}{\sqrt{m}} = \frac{1}{\sqrt{p}}\frac{1}{\sqrt{N}}\,. \tag{5.28}$$

If we assume the distribution of efficiency is Poisson distributed $\delta m = \sqrt{m}$ and $\delta N = \sqrt{N}$:

$$\frac{\delta\epsilon}{\epsilon} = \sqrt{\frac{1+p}{p}}\frac{1}{\sqrt{N}}\,. \tag{5.29}$$

If we assume the distribution of efficiency is binomially distributed $m = pN$ and $V[m] = pqN$ with N fixed:

$$\frac{\delta\epsilon}{\epsilon} = \sqrt{\frac{1-p}{p}}\frac{1}{\sqrt{N}}\,. \tag{5.30}$$

The standard deviation estimator is

$$\sigma = \left(\frac{\bar{f^2} - \bar{f}^2}{N-1} \right)^{1/2} \approx \frac{1}{\sqrt{N}} \left(\bar{f^2} - \bar{f}^2 \right)^{1/2} . \tag{5.31}$$

For the hit-and-miss method,

$$f(x) = \begin{cases} 1 & \text{if } x \text{ inside volume,} \\ 0 & \text{elsewhere.} \end{cases} \tag{5.32}$$

Thus

$$\bar{f} = \frac{m}{N} = p \quad \text{and} \quad \bar{f^2} = \frac{m}{N} = p \tag{5.33}$$

giving

$$\begin{aligned} \delta\epsilon &= \sqrt{\frac{p - p^2}{N}} = \sqrt{\frac{p(1-p)}{N}} , \\ \frac{\delta\epsilon}{\epsilon} &= \sqrt{\frac{1-p}{p}} \frac{1}{\sqrt{N}} , \end{aligned} \tag{5.34}$$

which is the binomial result.

Figure 5.4 shows the efficiency relative uncertainty using the different statistical distribution mentioned above. Calculating the statistical uncertainty assuming a binomial distribution is probably most correct. All methods approach the same vanishing relative uncertainty for large N. All methods give vanishing absolute uncertainty as $p \to 0$. The first naive estimate Eq. (5.27) equals the binomial estimate when $p = 1/2$. The second naive estimate Eq. (5.28) and the Poisson estimate never equal the binomial estimate for finite uncertainty. We also note that the binomial estimate vanishes when $p \to 1$. More on this topic in Sec. 6.3.

5.2.4 Example: Random variables

We will discuss three simple examples of generating or using random variables.

5.2.4.1 Exponential distribution

We will demonstrate generating random variables with a $(1/\tau)\exp(-t/\tau)$ distribution over the range [a,b). Let's pick $\tau = 2$ and $t = [0, 5)$, but the code will be general enough to use any values for these three parameters. For this example, we will use the inverse transform method, not the accept-reject method. We will display the results in a finely binned histogram of frequency of occurrence versus t and superimpose a plot of $(1/\tau)\exp(-t/\tau)$ versus t to show agreement.

Since we are generating random variables over a finite range, we need to renormalize the probability distribution. The probability distribution is $1/[\tau(\alpha-\beta)]exp(-t/\tau)$, where $\alpha = \exp(-a/\tau)$ and $\beta = \exp(-b/\tau)$. The inverse

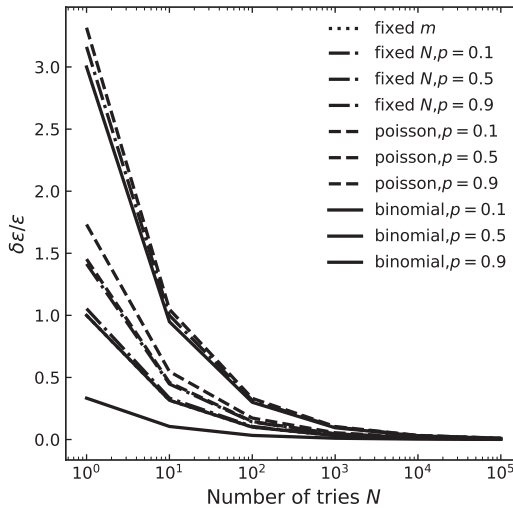


Figure 5.4 Relative efficiency uncertainties using different statistical distributions.

cumulative distribution is $t = -\tau \ln[\alpha + (\beta - \alpha)u]$, where u is a uniform random number on $[0, 1)$.

Consider the following code and the histogram it produces.

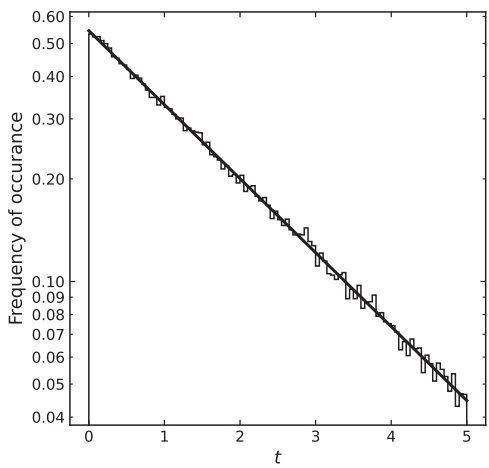
```
"""Generate exponential random variables
using transform mathod.
"""

import numpy as np
from numpy.random import seed, rand
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from matplotlib.ticker import FormatStrFormatter

plt.style.use('./mystyle.mplstyle')

# Common parameters.
debug = False
a = 0.0
b = 5.0
tau = 2
alpha = np.exp(-a/tau)
beta  = np.exp(-b/tau)

# Perform inverse transform.
N = 100000 # Number of random variables to generate.
binWidth = 0.05
Nbins = int((b-a)/binWidth) # Number of bins in histograms.
# Good practice (we will all get same string of random numbers).
```

```
seed(0)

if debug: print (b,binWidth,Nbins)
x1 = -tau*np.log(alpha+(beta-alpha)*rand(N)) * np.ones(N)
# Normalize by dividing by the number
# of samples times the bin width.
y1 = np.ones(N) / (binWidth*N)

fig, ax = plt.subplots()
plt.hist(x1,bins=Nbins,weights=y1,histtype='step')

# Plot the analytic function for comparison.
x2 = np.linspace(a,b,1000)
y2 = np.exp(-x2/tau) / (tau*(alpha-beta))
plt.plot(x2,y2,lw=2)

# Make plot pretty.
plt.xlabel(r"$t$")
plt.ylabel("Frequency of occurance")
plt.yscale('log') # Makes comparison easy.
ax.yaxis.set_minor_formatter(FormatStrFormatter("%.2f"))
ax.yaxis.set_major_formatter(FormatStrFormatter("%.2f"))
plt.savefig('5_2_exp.pdf')
plt.show()
```

Note that by plotting the distribution on a log-y scale, the distribution should be linear. I typically never plot exponential or power-law distribution on a linear scale. Too much information is lost.

5.2.4.2 Circular distribution

In this example, we generate random variables on a circle bound between $0 \leq x \leq 10$ and $0 \leq y \leq 10$.



Standard method: for uniform random number u on $[0, 1)$,

$$\begin{aligned} \theta &= \frac{\pi}{2} u \,, \\ x &= R \cos\theta \,, \\ y &= R \sin\theta \,. \end{aligned} \tag{5.35}$$

Inverse transform method: pdf
Motivated from above $x/y = \cot\theta$, and the normalized pdf can be written as

$$p(x) = \frac{1}{R}\frac{x}{y} = \frac{1}{R}\frac{x}{\sqrt{R^2 - x^2}} \,. \tag{5.36}$$

The CDF is

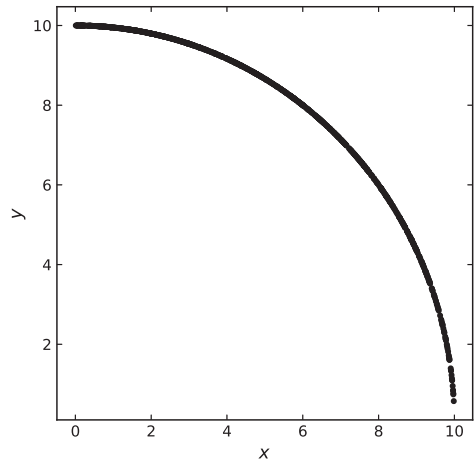
$$P(\alpha) = \frac{1}{R}\int_0^{\alpha} \frac{x}{\sqrt{R^2 - x^2}} = 1 - \sqrt{1 - \left(\frac{\alpha}{R}\right)^2} \,. \tag{5.37}$$

The inverse CDF is

$$y = R\sqrt{1 - u^2} \,. \tag{5.38}$$

Consider the following code that uses the inverse-transform method and the plot it produces.

```
"""Sample circumfrance of quarter circle."""

import numpy as np
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

R = 10
N = 1000
x = np.empty(N)
y = np.empty(N)

u = np.random.rand(N)
x = R*u
y = R*np.sqrt(1-u**2)

fig, ax = plt.subplots()
plt.scatter(x,y,marker='.',c='k')
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")
plt.savefig('5_2_circle.pdf')
plt.show()
```



5.2.4.3 Integrate quarter circle

This next example calculates an area using the accept-reject method. Consider the following algorithm,

$x = (R - 0)u_1 + 0,$

$y = y_{\max} u_2 = Ru_2,$

test $y < f(x) = \sqrt{R^2 - x^2}$.

A modified algorithm (effectively identical) is

$x = Ru_1,$

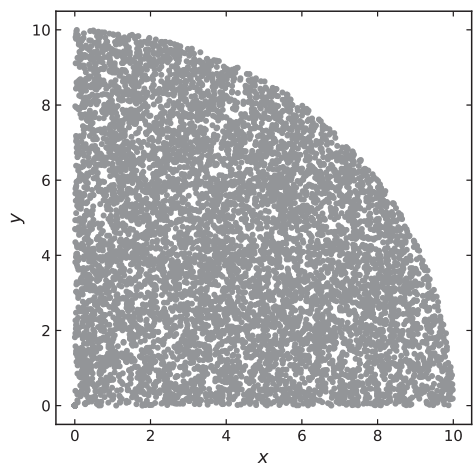
$y = Ru_2,$

test $\sqrt{x^2 + y^2} < R$; or test $x^2 + y^2 < R^2$.

Consider the following code that uses the modified method.

```
"""Random distibution in quarter circle."""

import numpy as np
from random import seed, random
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

R = 10
N = 10000
m = 0 # Accept.
```

```
seed(0)
x = np.empty(N)
y = np.empty(N)

for i in range(N):
    rx = R*random()
    ry = R*random()
    r2  = rx*rx + ry*ry
    if r2 < R*R:
        m += 1
        x[i] = rx
        y[i] = ry

fig, ax = plt.subplots()
plt.scatter(x,y,marker='.',c='grey')
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")
plt.savefig('5_2_circle_MC.pdf')
plt.show()
```



The above code performs the sampling. The following code calculates the area and estimate the uncertainty.

```
"""Calculate area and estimate the uncertainty."""

import numpy as np
from random import random

iterate = [100,10000,1000000]
for N in iterate:
    R = 10
    m = 0
    for _ in range(N):
```

```
        rx = R*random()
        ry = R*random()
        r2  = rx*rx + ry*ry
        if r2 < R*R: m += 1
    theory = R*R * np.pi / 4.0
    prob   = m/N
    area   = R*R * m /N
    diff   = np.abs(theory - area)
    pred   = R*R * np.sqrt(1.0/N)

    print ("-----------------------------")
    print ("{:.1f}".format(theory),"analytical area")
    print ("{:.1f}".format(area),  "numerical area")
    print ("{:.0e}".format(N),     "tries/",m,"accepted")
    print ("{:.1f}".format(diff),  "area difference")

    print("uncertainties")
    print ("{:.2f}".format(pred),
           "1/sqrt(N)")
    print ("{:.2f}".format(np.sqrt(prob)*pred),
           "1/sqrt(m)")
    print ("{:.2f}".format(np.sqrt(prob*(1+prob))*pred),
           "Poisson")
    print ("{:.2f}".format(np.sqrt(prob*(1-prob))*pred),
           "Binomial")
    print ()
```

```
78.540 analytical area
82.000 calculated area
1.000000e+02 tries/ 82 accepted
3.460 difference
-----------------------------
10.0000 1/sqrt(N)
9.0554 1/sqrt(m)
12.2164 Poisson
3.8419 Binomial

78.540 analytical area
78.450 calculated area
1.000000e+04 tries/ 7845 accepted
0.090 difference
-----------------------------
1.0000 1/sqrt(N)
0.8857 1/sqrt(m)
1.1832 Poisson
0.4112 Binomial

78.540 analytical area
78.567 calculated area
1.000000e+06 tries/ 785674 accepted
0.028 difference
-----------------------------
```

```
0.1000 1/sqrt(N)
0.0886 1/sqrt(m)
0.1184 Poisson
0.0410 Binomial
```

For a proper uncertainty estimate, we should calculate the area many times for fixed N. Then each area value can be entered into a histogram to obtain a distribution of areas. By calculating the mean and variance of the distribution, we can attempt to determine the probability distribution. The probability distribution can then be used to determine an estimate of the uncertainty, or else the standard deviation of the distribution can be used.

5.3 MONTE CARLO INTEGRATION AND SAMPLING

We now discuss MC integration in detail. Often the same procedure can be used to sample the probability distribution of the function being integrated at the same time as integrating it. In the next two sections, I follow the notation of Alex Gezerlis [2] which I find sufficiently rigorous, although my explanations will be terser.

5.3.1 Monte Carlo integration: uniform sampling

We use the uniform distribution to derive an expression for the MC estimate of an integral, along with an estimate of its uncertainty. We will also introduce the distinction between population and sampling moments.

5.3.1.1 Population mean

If $f(X)$ is a continuous function of a random variable X, the expectation value of f is

$$\langle f(X)\rangle = \int_{-\infty}^{\infty} p(x)f(x)dx\,. \tag{5.39}$$

For a uniform pdf,

$$p(x) = \frac{1}{b-a}\,, \tag{5.40}$$

we and obtain

$$\langle f(X)\rangle = \frac{1}{b-a}\int_a^b f(x)dx\,, \tag{5.41}$$

which is the population mean. We don't know $\langle f(X)\rangle$; it can only be estimated.

5.3.1.2 Sample mean

Let X_i be random variables drawn from a uniform distribution. The function values $f(X_i)$ are random variables by acting on X_i with function f. The

arithmetic average is

$$\overline{f} = \frac{1}{n}\sum_{i=0}^{n-1} f(X_i)\,. \tag{5.42}$$

This is the sample mean. It can be shown that $\langle\overline{f}\rangle = \langle f(X)\rangle$, where $\overline{f}$ is an unbiased estimator of $\langle f(X)\rangle$. An estimator is a useful approximation. The estimator becomes unbiased as $\overline{f} \to \langle f(X)\rangle$ for $n \to \infty$. Thus an estimate of the integral is

$$\int_a^b f(x)dx \approx \frac{b-a}{n}\sum_{i=0}^{n-1} f(X_i)\,. \tag{5.43}$$

5.3.1.3 Population variance

The variance of the population mean is

$$\mathrm{var}[f(X)] = \langle[f(X) - \langle f(X)\rangle]^2\rangle = \langle f^2(X)\rangle - \langle f(X)\rangle^2\,. \tag{5.44}$$

For a uniform distribution

$$p(x) = \frac{1}{b-a}\,, \tag{5.45}$$

and we obtain

$$\mathrm{var}[f(X)] = \frac{1}{b-a}\int_a^b f^2(x)dx - \left(\frac{1}{b-a}\int_a^b f(x)dx\right)^2\,. \tag{5.46}$$

This is the population variance and the population standard deviation is $\sqrt{\mathrm{var}[f(X)]}$. Again, $\mathrm{var}[f(X)]$ is unknown; it can only be estimated.

5.3.1.4 Sample variance

The variance of the sample mean is

$$\mathrm{var}[\overline{f}] = \mathrm{var}\left[\frac{1}{n}\sum_{i=0}^{n-1} f(X_i)\right]\,. \tag{5.47}$$

To evaluate this, consider the identity for two random variables Y and Z:

$$\mathrm{var}[\lambda_1 Y + \lambda_2 Z] = \lambda_1^2\mathrm{var}[Y] + \lambda_2^2\mathrm{var}[Z] + 2\lambda_1\lambda_2\mathrm{cov}[Y,Z]\,. \tag{5.48}$$

If Y and Z are independent random variables, $\mathrm{cov}[Y,Z] = 0$. In our case, $f(X_i)$ are function value random variables from the same sample distribution so $\mathrm{var}[f(X_i)] = \mathrm{var}[f(X)]$. Then

$$\mathrm{var}[\overline{f}] = \frac{1}{n^2}\sum_{i=0}^{n-1}\mathrm{var}[f(X_i)] = \frac{1}{n^2}\sum_{i=0}^{n-1}\mathrm{var}[f(X)] = \frac{n}{n^2}\mathrm{var}[f(X)] = \frac{1}{n}\mathrm{var}[f(X)]\,. \tag{5.49}$$

Also

$$\sqrt{\text{var}[\overline{f}]} = \frac{1}{\sqrt{n}}\sqrt{\text{var}[f(X)]}\,. \tag{5.50}$$

We still don't know var$[f(X)]$.

Consider the following estimator for the population variance,

$$e_{\text{var}} = \overline{f^2} - \overline{f}^{\,2} = \frac{1}{n}\sum_{i=0}^{n-1} f^2(X_i) - \left(\frac{1}{n}\sum_{i=0}^{n-1} f(X_i)\right)^2 . \tag{5.51}$$

It can be shown that the expectation value is

$$\langle e_{\text{var}}\rangle = \frac{n-1}{n}\text{var}[f(X)]\,. \tag{5.52}$$

Thus e_{var} is a biased estimator, but $ne_{\text{var}}/(n-1)$ will be an unbiased estimator of the population variance. Thus

$$\text{var}(\overline{f}) = \frac{1}{n}\text{var}[f(X)] \approx \frac{1}{n}\frac{n}{n-1}e_{\text{var}} = \frac{1}{n-1}(\overline{f^2} - \overline{f}^{\,2})\,. \tag{5.53}$$

Notice that this is where the infamous $1/\sqrt{n-1}$ factor comes from.

Putting it all together

$$\int_a^b f(x)dx \approx (b-a)\overline{f} \pm (b-a)\sqrt{\frac{\overline{f^2} - \overline{f}^{\,2}}{n-1}}\,, \tag{5.54}$$

where

$$\overline{f} = \frac{1}{n}\sum_{i=0}^{n-1} f(X_i) \quad \text{and} \quad \overline{f^2} = \frac{1}{n}\sum_{i=0}^{n-1} f^2(X_i)\,, \tag{5.55}$$

and the X_i are chosen uniformly from a to b. The factor $1/\sqrt{n-1}$ slowly decreases the variance as the number of samples n increases. This method is sometimes called the mean-value method of MC integration.

5.3.2 Monte Carlo integration: importance sampling

The method of importance sampling—or weighted sampling—aims to make the function to integrate (integrand) as flat as possible. It is a variance reduction technique. The idea is to use a simple weighting function from which we can draw nonuniform random deviants concentrated where the integrand is the largest—the important regions.

The expectation value of the general function $g(x)$ distributed according to probability $p(x) = w(x)/\int_a^b w(x)dx$ is

$$\langle g(X)\rangle_w = \int_a^b p(x)g(x)dx = \frac{\int_a^b w(x)g(x)dx}{\int_a^b w(x)dx}\,, \tag{5.56}$$

where $w(x)$ is a positive weight function—not necessarily a pdf.

Taking $g(x) = f(x)/w(x)$ gives

$$\left\langle \frac{f(X)}{w(X)} \right\rangle_w = \frac{\int_a^b f(x)dx}{\int_a^b w(x)dx}\,, \tag{5.57}$$

where $f(x)$ is another general function.

After rearranging, we have a formula for the integral of $f(x)$:

$$\int_a^b f(x)dx = \left\langle \frac{f(X)}{w(X)} \right\rangle_w \int_a^b w(x)dx \approx \frac{1}{N}\sum_{i=1}^{N-1} \frac{f(X_i)}{w(X_i)} \int_a^b w(x)dx\,. \tag{5.58}$$

For the sampling mean, $(X_0, X_1, \ldots, X_{n-1})$ are drawn from the probability density function $p(x) = w(x)/\int_a^b w(x)dx$, which is not uniform.

The standard deviation is

$$\sigma = \sqrt{\frac{\mathrm{var}_w(f/w)}{N-1}} \int_a^b w(x)dx\,, \tag{5.59}$$

where $\mathrm{var}_w(f/w) = \langle (f/w)^2 \rangle_w - \langle f/w \rangle_w^2$. By a strategic choice of weight function $w(x)$, $f(x)/w(x)$ can be made flatter, and the variance will be reduced. This flattening of the integrand is the key to the success of the importance sampling method.

The method has two general implementations, and we will work with both of them in the examples and problems: 1) sometimes the integrand has the apparent form of f/w, and w is a simple function from which we can randomly sample; 2) other times the integrand has no apparent ratio of functions, or the denominator is too complicated to sample from easily—thus we make a change of variable to introduce such a function in the denominator. Hence, sometimes the method is called a change of variable.

Important sampling effectively divides an integrand by a similar known integrand to reduce the variance. There are other variance reduction techniques. For example, we could also consider subtracting a similar known integrand from the integrand being considered to reduce the variance or a combination of division and subtraction.

5.3.3 Example: importance sampling

Let's now work through an example using importance sampling. We will integrate the relativistic Breit-Wigner distribution that is often used in quantum mechanics to describe a particle resonance. The relativistic Breit-Wigner pdf is

$$f(x;\rho) = \frac{k}{(x^2-\rho^2)^2 + \rho^2}\,, \tag{5.60}$$

where k is a constant give by

$$k = \frac{2\sqrt{2}\rho\gamma}{\pi\sqrt{\rho^2+\gamma}} \quad \text{and} \quad \gamma = \sqrt{\rho^2(\rho^2+1)}\,. \tag{5.61}$$

Physically x would be the centre-of-mass energy and ρ the mass of the resonance divided by its intrinsic decay width, Γ. In this case, Γ is the full-width at half-maximum (FWHM). Here I will refer to the relativistic Breit-Wigner distribution as just the Breit-Wigner distribution.

Because of the peaked nature of the Breit-Wigner distribution, we will consider Cauchy and Gaussian distributions for weight functions. The Cauchy distribution pdf is

$$f(x; x_0, \gamma) = \frac{1}{\pi\gamma}\left[\frac{\gamma^2}{(x-x_0)^2+\gamma^2}\right], \tag{5.62}$$

where x_0 is the location representing the position of the peak and γ is the scale parameter which is the half-wide at half-maximum (HWHM). The range of the pdf is $-\infty < x < +\infty$. Although the distribution has an undefined mean and variance, it is commonly found in physics. The Cauchy distribution is sometimes called the Lorentz or Breit-Wigner (non-relativistic) distribution. I will refer to it here as the Cauchy distribution. For the Gaussian (normal) pdf, we have already encountered it Eq. (5.13) and will not repeat the equation here. However, we need to know that the FWHM for a Gaussian distribution is $2\sqrt{2\ln 2}\sigma$.

We will integrate over a finite range $a \leq x \leq b$ and thus must renormalize the pdf weight functions. For the sampling, we will analytically calculate the inverse CDFs.

To generate random variables according to a Cauchy distribution over a limited range $[a, b]$:

$$p(x; x_0, \gamma) = \frac{w(x; x_0, \gamma)}{\int_a^b w(x'; x_0, \gamma)dx'}. \tag{5.63}$$

The cumulative distribution is

$$\begin{aligned} u = \int_a^x p(x'; x_0, \gamma)dx' &= \frac{\int_a^x w(x'; x_0, \gamma)dx'}{\int_a^b w(x'; x_0, \gamma)dx'} \\ &= \frac{\int_{-\infty}^x w(x'; x_0, \gamma)dx' - \int_{-\infty}^a w(x'; x_0, \gamma)dx'}{\int_a^b w(x'; x_0, \gamma)dx'} \\ &= \frac{\frac{1}{\pi}\tan^{-1}\left(\frac{x-x_0}{\gamma}\right) + \frac{1}{2} - P(a; x_0, \gamma)}{\int_a^b w(x'; x_0, \gamma)dx'}, \end{aligned} \tag{5.64}$$

where P is the cumulative Cauchy distribution. The inverse cumulative distribution is

$$\begin{aligned} x &= x_0 + \gamma\tan\left[\pi\left(u\int_a^b w(x';, x_0, \gamma)dx' + P(a; x_0, \gamma) - \frac{1}{2}\right)\right] \\ &= x_0 + \gamma\tan\left[\pi\left(u' - \frac{1}{2}\right)\right], \end{aligned} \tag{5.65}$$

where u' is a uniform deviate in the range $[a, b]$. The normalization of the Cauchy distribution over the limited range is

$$\int_a^b w(x'; x_0, \gamma)dx' = \frac{1}{\pi}\left[\tan^{-1}\left(\frac{b - x_0}{\gamma}\right) - \tan^{-1}\left(\frac{a - x_0}{\gamma}\right)\right] . \tag{5.66}$$

To generate random numbers according to a Gaussian distribution over a limited range $[a, b]$:

$$p(x; \mu, \sigma) = \frac{w(x; \mu, \sigma)}{\int_a^b w(x'; \mu, \sigma)dx'} . \tag{5.67}$$

The cumulative distribution is

$$\begin{aligned} u &= \int_a^x p(x'; \mu, \sigma)dx' = \frac{\int_a^x w(x'; \mu, \sigma)dx'}{\int_a^b w(x'; \mu, \sigma)dx'} \\ &= \frac{\int_{-\infty}^x w(x'; \mu, \sigma)dx' - \int_{-\infty}^a w(x'; \mu, \sigma)dx'}{\int_a^b w(x'; \mu, \sigma)dx'} \\ &= \frac{\frac{1}{2}\left[1 + \text{erf}\left(\frac{x-\mu}{\sqrt{2}\sigma}\right)\right] - P(a; \mu, \sigma)}{\int_a^b w(x'; \mu, \sigma)dx'} , \end{aligned} \tag{5.68}$$

where P is the cumulative Gaussian distribution. The inverse cumulative distribution is

$$\begin{aligned} x &= \mu + \sqrt{2}\sigma\ \text{erf}^{-1}\left[2\left(u\int_a^b w(x'; \mu, \sigma)dx' + P(a; \mu, \sigma)\right) - 1\right] \\ &= \mu + \sqrt{2}\sigma\ \text{erf}^{-1}\left[2u' - 1\right] , \end{aligned} \tag{5.69}$$

where u' is a uniform deviate in the range $[a, b]$. The normalization of the Gaussian is

$$\int_a^b w(x'; \mu, \sigma)dx' = \frac{1}{2}\left[\text{erf}\left(\frac{b - \mu}{\sqrt{2}\sigma}\right) - \text{erf}\left(\frac{a - \mu}{\sqrt{2}\sigma}\right)\right] . \tag{5.70}$$

Consider the following code. We will integrate the Breit-Wigner distribution over the range $0 < x < 6$ centred on $x = 3$. We will take the FWHM to be two. We use the probability distributions `norm, cauchy` and `rel_breitwigner` from the `scipy.stats` library module rather than code the distributions ourselves. This is a small saving.

```
"""Importance sampling of Beit-Wigner distribution."""

import numpy as np
from scipy.integrate import quad
```

```
from scipy.stats import norm, cauchy, rel_breitwigner
from scipy.special import erf, erfinv
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

# Cauchy distribution: normalize and random deviate.
def pcauchy(a,b,N,mu,g):
    cdfa = (1.0/np.pi) * (np.arctan((a-mu)/g)) + 0.5
    cdfb = (1.0/np.pi) * (np.arctan((b-mu)/g)) + 0.5
    pnorm = cdfb - cdfa
    x = np.random.uniform(cdfa,cdfb,N)
    ran = mu + g*np.tan(np.pi*(x-0.5))
    if np.any((ran < a)|(ran > b)):
        print("generated numer out of range")
    return pnorm, ran

# Normal distribution: normalize and random deviate.
def pgaussian(a,b,N,mu,sigma):
    cdfa = 0.5 * (1.0 + erf((a-mu)/(sigma*np.sqrt(2))))
    cdfb = 0.5 * (1.0 + erf((b-mu)/(sigma*np.sqrt(2))))
    pnorm = cdfb - cdfa
    x = np.random.uniform(cdfa,cdfb,N)
    ran = mu + np.sqrt(2)*sigma*erfinv(2*x-1)
    if np.any((ran < a)|(ran > b)):
        print("generated numer out of range")
    return pnorm, ran

# Mean value integral.
def MCmean(a,b,N,mu,FWHM):
    x = np.random.uniform(a,b,N)
    y = rv_bw.pdf(x)
    s = np.sum(y)
    s2 = np.sum(y**2)
    est = (b-a) * s / N
    expe = s / N
    expe2 = s2 / N
    std = (b-a) * np.sqrt( (expe2-expe**2) / (N-1) )
    return est, std

# Important sampling.
def MCimport(a,b,N,mu,FWHM):
    if CAUCHY:
      pnorm, x = pcauchy(a,b,N,mu,FWHM/2)
      weight = rv_cauchy.pdf(x)
    else:
      sigma = FWHM / (2*np.sqrt(2*np.log(2)))
      pnorm, x = pgaussian(a,b,N,mu,sigma)
      weight = rv_norm.pdf(x)
    y = rv_bw.pdf(x) / weight
    s = np.sum(y)
    s2 = np.sum(y**2)
    est = pnorm * s / N
    expe = s / N
    expe2 = s2 / N
    std = pnorm * np.sqrt( (expe2-expe**2) / (N-1) )
    return est, std
```

```
# Constants and parameters.
DEBUG   = False
VERBOSE = False
CAUCHY  = True
np.random.seed(0)
a = 0
b = 6
mu = 3
FWHM = 2.0

# Create frozen RV objects.
rho = mu/FWHM
rv_bw = rel_breitwigner(rho,0,FWHM)
rv_cauchy = cauchy(mu,FWHM/2)
sigma = FWHM / (2*np.sqrt(2*np.log(2)))
rv_norm = norm(mu,sigma)

# Check integral.
Itheory = quad(rv_bw.pdf,a,b)
print("Integral by quadurature:",Itheory)
if DEBUG:
  print("Integral: Cauchy",
        (1/np.pi)*(np.arctan((b-mu)/sigma))
       -(1/np.pi)*(np.arctan((a-mu)/sigma)))
  print("Integral: Gaussian",
       0.5*(erf((b-mu)/(sigma*np.sqrt(2)))
           -erf((a-mu)/(sigma*np.sqrt(2)))))

# MC methods.
xn = []
ym = []
em = []
ys = []
es = []
Nruns = 7
for n in range(Nruns):
    N = 8*4**n
    # Sample with uniform deviate.
    I = MCmean(a,b,N,mu,FWHM)
    if VERBOSE: print("Integral: mean value",N,I)
    xn.append(N)
    ym.append(Itheory[0]-I[0])
    em.append(I[1])
    # Importance sampling.
    I = MCimport(a,b,N,mu,FWHM)
    if VERBOSE: print("Integral: importance",N,I)
    ys.append(Itheory[0]-I[0])
    es.append(I[1])

# Make plots.
fig, axes = plt.subplots(2,2,figsize=(7,7))
x = np.linspace(a,b,1000)

# Plot distributions normalized to peak.
A = np.pi*FWHM/2
axes[0,0].plot(x,A*rv_cauchy.pdf(x),ls='dashed',label="Cauchy")
```

```
A = np.sqrt(2*np.pi)*sigma
axes[0,0].plot(x,A*rv_norm.pdf(x),ls='dashdot',label="Gaussian")
axes[0,0].set_xlabel(r"$x$")
axes[0,0].set_ylabel(r"$f(x)$")
gamma = rho*np.sqrt(rho**2+1)
k = (2*np.sqrt(2)*rho*gamma) / (np.pi*np.sqrt(rho**2+gamma))
A = rho**2*FWHM/k
axes[0,0].plot(x,A*rv_bw.pdf(x),ls='solid',label="Breit-Wigner")
axes[0,0].legend(loc='lower center',bbox_to_anchor=(0.55,0))

# Plot the ratio of Breit-Wigner to weight function.
axes[0,1].plot(x,rv_bw.pdf(x)/rv_cauchy.pdf(x),
               ls='dashed', label="weight Cauchy")
axes[0,1].plot(x,rv_bw.pdf(x)/rv_norm.pdf(x),
               ls='dashdot',label="weight Gaussian")
axes[0,1].set_xlabel(r"$x$")
axes[0,1].set_ylabel(r"$f(x)/w(x)$")
axes[0,1].set_yscale('log')
axes[0,1].legend(loc='upper right',bbox_to_anchor=(0.9,0.9))

# Plot quadrature minus MC.
axes[1,0].errorbar(xn,ys,yerr=es,marker='o',ls='solid',
                   label="importance")
axes[1,0].errorbar(xn,ym,yerr=em,marker='s',ls='dashed',
                   label="mean-value")
axes[1,0].set_xscale('log')
axes[1,0].set_xlabel("Samples")
axes[1,0].set_ylabel("Quad-MC")
axes[1,0].legend(loc='lower right',bbox_to_anchor=(0.9,0.1))

# Plot uncertainties,
axes[1,1].plot(xn,es,marker='o',ls='solid', label="importance")
axes[1,1].plot(xn,em,marker='s',ls='dashed',label="mean-value")
axes[1,1].set_xscale('log')
axes[1,1].set_xlabel("Samples")
axes[1,1].set_ylabel("Uncertainty")
axes[1,1].legend(loc='upper right',bbox_to_anchor=(0.9,0.9))

fig.tight_layout()
plt.savefig("5_5_importance.pdf")
plt.show()
```

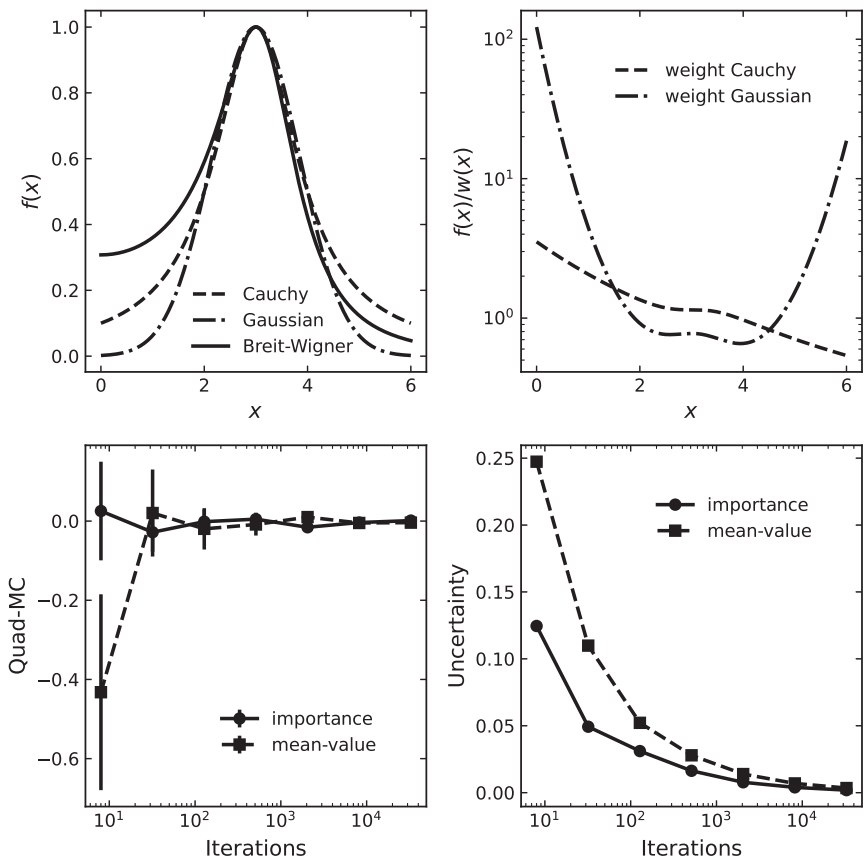
```
Integral by quadurature: (0.9720442215451264, 1.130867904563288e-08)
```

We first print out the result of the integral, and its error using quadrature. Recall that although quadrature can be used to integrate the distribution, it cannot be used to sample it; our code could be easily adapted to sample the Breit-Wigner distribution.

The top-left figure shows the three distributions superimposed with a common FWHM. For comparison purposes, the distributions have all been normalized to have a maximum of unit. The important region around the peak is similar for all three distributions. The main difference in the distributions is the tails.

The top-right figure shows the Breit-Wigner distribution divided by the Cauchy distribution or the Gaussian distribution. The variance is reduced more by using the Cauchy distribution as the weight function. For the remaining calculations and plots, the Cauchy distribution is used as the weight function. If `CAUCHY = False`, the code uses the Gaussian distribution as the weight function; I encourage the reader to play with this alternative weight function.

Integrating the Breit-Wigner by uniform sampling is not particularly difficult, so the improvement by using importance sampling is expected to be marginal. The bottom-left figure shows the difference between quadrature integration and the mean-value MC integration or importance sampling methods. Importance sampling appears to do better than unweighted sampling only when a low number of 10 samples are used. Be reminded that running the code for a different random seed will give a different result, within the statistical uncertainty.



The bottom-right figure shows that the estimated uncertainty decreases significantly for a large number of samples. The importance sample uncertainty is about one-half the mean-value sampling uncertainty for 10 samples. For a large number of samples, importance sample give no benefit for this simple example.

5.3.4 Monte Carlo integration: stratified sampling

Stratified sampling can be used to concentrate the sampling points where the variance of the function is the largest. It introduces a deterministic procedure into MC estimation and is thus referred to as a quasi Monte Carlo (QMC) method.

If the range of integration is divided into M subdomains (strata):

$$\int_a^b f(x)dx = \sum_{h=0}^{M-1} \frac{b_h - a_h}{n_h} \sum_{i=0}^{n_h-1} f(X_{i,h}) \,, \tag{5.71}$$

where $X_{i,h}$ is uniform random variable i in subregion h, and $a_0 = a, b_h = a_{h+1}, b_{M-1} = b$. Drawing random deviates from other distributions would also be possible. Equation (5.71) is an unbiased estimator of the integral.

The standard deviation is

$$\sigma = \sqrt{\sum_{h=0}^{M-1} \frac{(b_h - a_h)^2}{n_h} \sigma_h^2(f(x))} \,, \tag{5.72}$$

where σ_h is the standard deviation of the sample in the hth subdomain. The variance is reduced if not all the integrals in the subdomains are the same. One may also stratify with a different number of samples in each subdomain; concentrating more samples in the high-variance subdomains. A poor allocation of subdomains and number of samples per subdomain could actually increase the variance. One subdomain per sample may appear optimal but the calculation of the variance would not be possible.

An alternative approach to stratified sampling would be to run the code multiple times for different strata, or slices, with different MC integration settings, and then stitch the resulting slice information together afterwards. This approach is beneficial when one has access to a large number of computing cores. The method is introduced in Problem 5.5.6.

5.3.5 Example: stratified sampling

We will consider an example that uses uniform stratification in four regions. To exploit the full power of stratification, one should sample more in the high variance regions; the reader is encourage to do so. Consider the function to be integrated in Fig. 5.5, which is a simple Higgs potential that allows symmetry breaking in the standard model of particle physics. Although this function

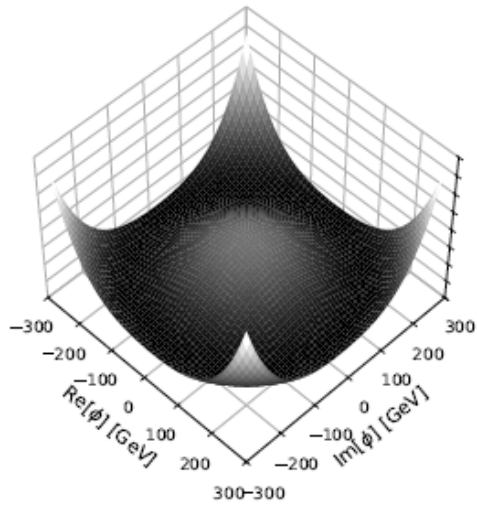


Figure 5.5 Quartic function to integrate in one dimension.

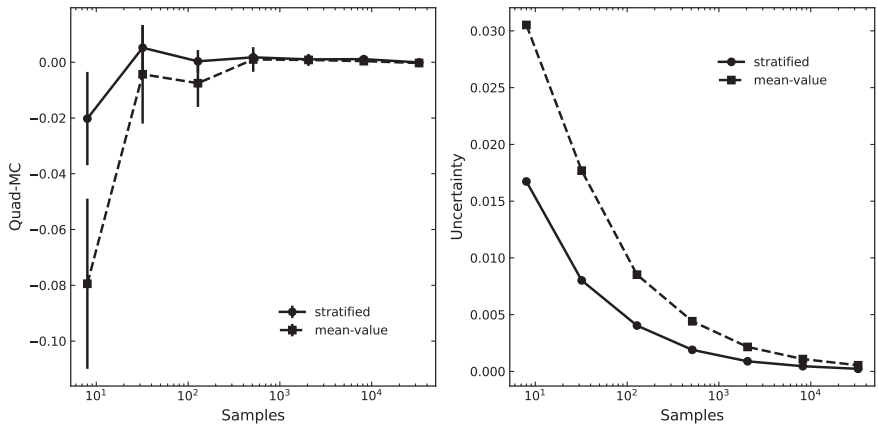
is two dimensional—real and imaginary parts—because of the symmetry we only need to integrate in one dimension. Consider the following code.

```
"""Stratified sampling of Higgs potential"""

import numpy as np
from scipy.integrate import quad
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

plt.style.use('./mystyle.mplstyle')

# Function to be integrated.
def f(x):
    lam = 0.13
    lam = 0.13e-12 # convert to TeV^4
    v = 246.0 # GeV
    return lam*(x**2 - v**2)**2

# Same function for plotting.
def V(X,Y):
    lam = 0.13
    #lam = 0.13e-12 # convert to TeV^4
    v = 246.0 # GeV
    R = np.sqrt(X**2+Y**2)
    return lam*(R**2 -v**2)**2

# Mean value integral.
def MCmean(a,b,N):
```

```
    x = np.random.uniform(a,b,N)
    y = f(x)
    s = np.sum(y)
    s2 = np.sum(y**2)
    est = (b-a) * s / N
    expe = s / N
    expe2 = s2 / N
    std = (b-a) * np.sqrt( (expe2-expe**2) / (N-1) )
    return est, std

# Stratified sampling.
def MCstrat(a,b,N,M):
    inc = (b-a) / M
    est = 0
    std = 0
    for i in range(M):
        x0 = a + i*inc
        x = np.random.uniform(x0,x0+inc,N)
        y = f(x)
        s = np.sum(y)
        s2 = np.sum(y**2)
        est += inc * s / N
        expe = s / N
        expe2 = s2 / N
        stds = inc * np.sqrt( (expe2-expe**2) / (N-1) )
        std += stds**2
    return est, np.sqrt(std)

# Constants.
verbose = False
np.random.seed(0)
a = -280
b =  280

# Run parameters.
M = 4
print("subregions",M)

# Check total integral.
Itheory = quad(f,a,b)
print("Integral by quadrature:",Itheory)

# MC methods.
xn = []
ym = []
em = []
ys = []
es = []
Nruns = 7
for n in range(Nruns):
    N = 8*4**n
    # Sample with uniform deviate.
    I = MCmean(a,b,N)
    if verbose: print("Integral: mean value",N,I)
    xn.append(N)
    ym.append(Itheory[0]-I[0])
    em.append(I[1])
```

```
    # Stratified sampling.
    I = MCstrat(a,b,N//M,M)
    if verbose: print("Integral: stratified",N,I)
    ys.append(Itheory[0]-I[0])
    es.append(I[1])

# Make plots.
fig = plt.figure(figsize=(5,5))
ax = plt.axes(projection='3d')

x = np.linspace(a,b,1000)
y = np.linspace(a,b,1000)
X, Y = np.meshgrid(x,y)
Z = V(X,Y)
ax.plot_surface(X,Y,Z,cmap='grey')
ax.view_init(50,-45) # Rotates plot (elev,azim).
ax.xaxis.pane.fill = False
ax.yaxis.pane.fill = False
ax.zaxis.pane.fill = False
ax.set_xlabel(r"Re$[\phi]$ [GeV]")
ax.set_ylabel(r"Im$[\phi]$ [GeV]")
ax.set_zticks(ax.get_zticks())
labels = [item.get_text() for item in ax.get_zticklabels()]
empty_string_labels = ['']*len(labels)
ax.set_zticklabels(empty_string_labels)
plt.xlim([-300,300])
plt.ylim([-300,300])
plt.savefig("5_7_Higgs.pdf",bbox_inches='tight')
plt.show()

fig = plt.figure(figsize=(10,5))

ax = fig.add_subplot(1,2,1)
ax.errorbar(xn,ys,yerr=es,marker='o',ls='solid',
            label="stratified")
ax.errorbar(xn,ym,yerr=em,marker='s',ls='dashed',
            label="mean-value")
ax.set_xscale('log')
ax.set_xlabel("Samples")
ax.set_ylabel("Quad-MC")
ax.legend(loc='lower right',bbox_to_anchor=(0.9,0.1))

ax = fig.add_subplot(1,2,2)
ax.plot(xn,es,marker='o',ls='solid', label="stratified")
ax.plot(xn,em,marker='s',ls='dashed',label="mean-value")
ax.set_xscale('log')
ax.set_xlabel("Samples")
ax.set_ylabel("Uncertainty")
ax.legend(loc='upper right',bbox_to_anchor=(0.9,0.9))

fig.tight_layout()
plt.savefig("5_7_stratified.pdf")
plt.show()
```

```
subregions 4
Integral: quad (0.1258368861568, 1.3970700835841398e-15)
```



The plots shows the gain due to stratification. The left figure shows the difference between quadrature and MC methods versus the number of samples. The right figure shows the uncertainty versus number of samples. As expected, the two MC methods are equivalent for high number of samples, but uncertainties of less than a factor of one-half are obtained with only 10 samples using stratified sampling compared to mean-value sampling. When referring to the number of samples in stratified sampling, we are sampling each of the four subdomains one-quarter of that of the mean-value method. Thus the total number of random numbers used in the two methods are the same.

5.3.6 Monte Carlo in multiple dimensions

The power of the MC integration method becomes most apparent in higher dimensions. For uniform sampling in multiple dimensions

$$\int f(\mathbf{x}) d^d x \approx V\overline{f} \pm \frac{V}{\sqrt{\mathcal{N}-1}} \left(\overline{f^2} - \overline{f}^{\,2}\right) \,, \tag{5.73}$$

where

$$\overline{f} = \frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} f(\mathbf{X}_i) \quad \text{and} \quad \overline{f^2} = \frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} f^2(\mathbf{X}_i) \,, \tag{5.74}$$

and $\mathcal{N}$ is the total number of samples. The vector $\mathbf{X}_i$ is d uniform random numbers. The standard deviation goes as $1/\sqrt{\mathcal{N}-1}$, not as d. This last statement emphasizes the usefulness of the MC method for calculating multidimensional integrals.

When there are d-dimensions, the total number of points in a quadrature integration whose size is M in each coordinate direction is $N = M^d$. The fractional uncertainty in estimating a one-dimensional integral of a function based upon evaluating M grid points, may be estimated as $\mathcal{O}(1/M)$. If this estimate applies to multiple-dimensional integration, then the fractional uncertainty is

$\mathcal{O}(1/M) = \mathcal{O}(N^{-1/d})$. The uncertainty in a MC estimate of the integral based on N evaluations is $\mathcal{O}(N^{-1/2})$. When $d > 2$, the MC square-root convergence scales better than the quadrature estimate. For quadrature, the uncertainty really goes as $\mathcal{O}(N^{-2/d})$, for $d > 1$. Only at $d \sim 4$ does the MC integration begin to have a significant advantage.

5.3.7 Example: Monte Carlo

This is an example of using weights to flatten the integrand of a multidimension integral. Recall the numerical integration formula:

$$I = \int_{x_1}^{x_2} dx f(x) = (x_2 - x_1)\langle f(x)\rangle \approx (x_2 - x_1)\frac{1}{N}\sum_{i=0}^{N-1} f(X_i)\,, \tag{5.75}$$

where $f(X_i)$ are values of $f(x)$ at N randomly chosen values of X in the range $[x_1, x_2]$. As an example, consider black hole production in proton–proton collisions. The cross section can be written as

$$\sigma = \int_{E_{\min}}^{E_{\max}} dE\hat{\sigma}(E)\mathcal{L}(E)\,, \tag{5.76}$$

where E is the centre-of-mass energy of the colliding partons (quarks, antiquarks and gluons) within the protons; $\hat{\sigma}$ is the parton-level cross section and $\mathcal{L}$ is the parton luminosity which takes us from protons to partons.

To flatten the integrand, we make a change of variable.

$$\frac{dY}{dE} = y(E)\,; \quad \sigma = \int_{Y_1=Y(E_{\min})}^{Y_2=Y(E_{\max})} dY\frac{\hat{\sigma}(E)\mathcal{L}(E)}{y(E)}\,, \tag{5.77}$$

where $y(E)$ is a weight function.

In the physics of this example, $\hat{\sigma}(E) \sim E^{\beta}$, where $\beta = 2/(n+1)$ and n is an integer. In addition, $\mathcal{L}(E) \sim E^{-8}$. Thus a sensible choice of the weight function is $y(E) = \alpha E^{\alpha-1} \Rightarrow Y(E) = E^{\alpha}$, where $\alpha = \beta - 7$. We now write the integral as

$$\sigma = [Y(E_{\max}) - Y(E_{\min})]\left\langle \frac{\hat{\sigma}(E)\mathcal{L}(E)}{y(E)} \right\rangle = \frac{E_{\max}^{\alpha} - E_{\min}^{\alpha}}{\alpha}\left\langle \frac{\hat{\sigma}(E)\mathcal{L}(E)}{E^{\alpha-1}} \right\rangle, \tag{5.78}$$

where $Y(E)$ is chosen at random from a uniform distribution between the minimum and maximum values.

The parton luminosity $\mathcal{L}(E)$ is most commonly written as a function of

$$\tau = \frac{\hat{s}}{s} = \frac{E^2}{s} \quad \text{and} \quad x_a = x, x_b = \frac{\tau}{x}\,, \tag{5.79}$$

where s is the square of the proton–proton centre-of-mass energy. Thus

$$\mathcal{L}(E) = \frac{2\tau}{E}\mathcal{L}(\tau) = \frac{2\tau}{E}\int_{\tau}^{1}\frac{dx}{x}f_1(x)f_2\left(\frac{\tau}{x}\right)\,. \tag{5.80}$$

This integral can also be evaluated by an MC procedure. Since $xf(x)$ are predetermined functions, we write

$$\mathcal{L}(\tau) = \frac{1}{\tau}\int_{\tau}^{1} \frac{dx}{x} x f_1(x) \frac{\tau}{x} f_2\left(\frac{\tau}{x}\right) = \frac{1}{\tau}\int_{\tau}^{1} \frac{dx}{x} h(x,\tau)\,. \tag{5.81}$$

To flatten the integrand, we make a change of variable

$$\frac{dZ}{dx} = z(x)\,; \quad \mathcal{L}(\tau) = \frac{1}{\tau}\int_{Z(\tau)}^{Z(1)} dZ \frac{h(x,\tau)}{xz(x)}\,, \tag{5.82}$$

where $z(x)$ is a weight function.

Since $f(x) \sim 1/x \Rightarrow h(x,\tau)$ is fairly flat. For efficiency, we pick $z(x) = 1/x \Rightarrow Z(x) = \ln x$. Therefore

$$\mathcal{L}(\tau) = \frac{Z(1) - Z(\tau)}{\tau}\langle h(x,\tau)\rangle = -\frac{\ln\tau}{\tau}\langle h(x,\tau)\rangle\,, \tag{5.83}$$

where $Z(x)$ is chosen at random from a uniform distribution between the minimum and maximum values.

Combining the two MC integrals gives

$$\sigma \approx \frac{E_{\max}^{\alpha} - E_{\min}^{\alpha}}{\alpha}\left\langle \frac{\hat{\sigma}}{E^{\alpha-1}} \frac{2}{E}[-\ln\tau] h(x,\tau) \right\rangle, \tag{5.84}$$

where first $Y(E)$ is chosen at random, then E is determined from it, and hence τ. Then given this value of τ, $Z(x)$ is chosen at random, then x is determined from it.

If we want to generate events drawn from this distribution, E_i and x_i are chosen for the ith event and the cross section σ_i is computed. Then σ_i is accepted or rejected against a maximum $\sigma_{\max}$ which is found from a previously performed search. An event i will be accepted with probability $\sigma_i/\sigma_{\max}$.

5.4 MARKOV CHAIN MONTE CARLO

In discussing Markov chain Monte Carlo, we will consider the Metropolis-Hastings algorithm. The Metropolis-Hastings algorithm is another one of the classic landmark algorithms that have enabled solutions to otherwise intractable problems. We previously discussed random-sampling techniques to reducing the integrand variance and sample the important regions. These algorithms required a knowledge of the integrand. We now discuss a sampling algorithm that interrogates the integrand without us giving it this knowledge; integrand-aware sampling.

5.4.1 Multidimensional weighted Monte Carlo integration

Consider the weighted sampling version of the multidimensional integral Eq. (5.73).

$$\int w(\mathbf{x}) f(\mathbf{x}) d^d x \approx V\overline{f} \pm \frac{V}{\sqrt{\mathcal{N}-1}} \left(\overline{f^2} - \overline{f}^{\,2}\right)^{1/2}, \tag{5.85}$$

where

$$\overline{f} = \frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} f(\mathbf{X}_i) \quad \text{and} \quad \overline{f^2} = \frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} f^2(\mathbf{X}_i)\,. \tag{5.86}$$

The $\mathbf{X}_i$'s are drawn from $w(\mathbf{x})$ in which the x_i are uniform.

Using this approach is not practical when the number of dimensions is large for at least the following reasons.

1. The change of variable needs to be replaced by a Jacobian matrix.
2. The many CDF integrations would probably need to be done numerically.
3. Most inverse CDFs could not be done analytically.

We will now discuss a method that circumvents these technical difficulties.

5.4.2 Markov chains and detailed balance

We consider an alternative approach to multidimensional weighted MC integration called Markov chain Monte Carlo (MCMC). In this section, I borrow from and follow the notation of Alex Gezerlis [2]. We will produce a sequences of random samples $\mathbf{X}_0, \mathbf{X}_1, \ldots, \mathbf{X}_{N-1}$, where the $\mathbf{X}_i$ are not independent. The sample $\mathbf{X}_i$ depends only on sample $\mathbf{X}_{i-1}$. The sequences are referred to as random walks. As $\mathcal{N} \to \infty$, the distribution of $\mathbf{X}_i$ produced will follow $w(\mathbf{x})$.

We want a stationary distribution of the sampling chain. Using the principle of detailed balance we write

$$w(\mathbf{X}) T(\mathbf{X} \to \mathbf{Y}) = w(\mathbf{Y}) T(\mathbf{Y} \to \mathbf{X})\,, \tag{5.87}$$

where $T(\mathbf{X} \to \mathbf{Y})$ is the conditional probability density that you move to $\mathbf{Y}$ if you start at $\mathbf{X}$; also called the transition probability. The function $w(\mathbf{X})$ is the probability density of being near $\mathbf{X}$. Thus $w(\mathbf{X})T(\mathbf{X} \to \mathbf{Y})$ is how likely it is to start at $\mathbf{X}$ and move to $\mathbf{Y}$.

Consider going from one probability density function to another.

$$p_i(\mathbf{X}) = p_{i-1}(\mathbf{X}) + \int \left[p_{i-1}(\mathbf{Y}) T(\mathbf{Y} \to \mathbf{X}) - p_{i-1}(\mathbf{X}) T(\mathbf{X} \to \mathbf{Y})\right] d^d\mathbf{Y}\,. \tag{5.88}$$

For $p_{i-1}(\mathbf{X}) = w(\mathbf{X})$,

$$p_i(\mathbf{X}) = w(\mathbf{X}) + \int \left[w(\mathbf{Y}) T(\mathbf{Y} \to \mathbf{X}) - w(\mathbf{X}) T(\mathbf{X} \to \mathbf{Y}) \right] d^d\mathbf{Y} \,. \tag{5.89}$$

The integrand vanishes by detailed balance. Therefore $p_i(\mathbf{X}) = p_{i-1}(\mathbf{X}) = w(\mathbf{X})$ is the stationary distribution we are looking for. If at some point of the iteration our distribution becomes the desired weight distribution, then it will stay there. This ensure stability of our solution.

How do we know we are approaching the stationary distribution? Consider the ratio

$$\frac{p_i(\mathbf{X})}{w(\mathbf{X})} = \frac{p_{i-1}(\mathbf{X})}{w(\mathbf{X})} + \int \left[p_{i-1}(\mathbf{Y}) \frac{T(\mathbf{Y} \to \mathbf{X})}{w(\mathbf{X})} - p_{i-1}(\mathbf{X}) \frac{T(\mathbf{X} \to \mathbf{Y})}{w(\mathbf{X})} \right] d^d\mathbf{Y} \,. \tag{5.90}$$

Then p_{i-1}/w is the ratio of actual to desired distributions. Notice that $T(\mathbf{X} \to \mathbf{Y})$ is non-negative. So using detailed balance

$$\frac{p_i(\mathbf{X})}{w(\mathbf{X})} = \frac{p_{i-1}(\mathbf{X})}{w(\mathbf{X})} + \int T(\mathbf{X} \to \mathbf{Y}) \left[\frac{p_{i-1}(\mathbf{Y})}{w(\mathbf{Y})} - \frac{p_{i-1}(\mathbf{X})}{w(\mathbf{X})} \right] d^d\mathbf{Y} \,. \tag{5.91}$$

The first term is a ratio that is driven closer to one. The first term in the integrand drives the ratio up. The second term in the integrand drives the ratio down. If

$$\frac{p_{i-1}(\mathbf{X})}{w(\mathbf{X})} > \frac{p_{i-1}(\mathbf{Y})}{w(\mathbf{Y})} \,, \tag{5.92}$$

the expression wants to lower the ratio, and visa versa.

To summarize. If we have a Markov chain that obeys detailed balance,

1. $w(\mathbf{X})$ is a stationary distribution, and
2. $p_i(\mathbf{X})$ asymptotically approaches that stationary distribution.

5.4.3 Metropolis-Hasting algorithm

The Metropolis-Hasting algorithm satisfies detail balance and splits the transition probability into two factors:

$$T(\mathbf{X} \to \mathbf{Y}) = \pi(\mathbf{X} \to \mathbf{Y}) \alpha(\mathbf{X} \to \mathbf{Y}) \,, \tag{5.93}$$

where $\pi(\mathbf{X} \to \mathbf{Y})$ is the probability of proposing a step and $\alpha(\mathbf{X} \to \mathbf{Y})$ is the probability of accepting the step. In other words, in an iteration, the system can move or stay where it is. Different choices for π exist and α is picked to satisfy detailed balance.

Consider the, so-called, Metropolis-Hasting ratio

$$R(\mathbf{X} \to \mathbf{Y}) = \frac{w(\mathbf{Y}) \pi(\mathbf{Y} \to \mathbf{X})}{w(\mathbf{X}) \pi(\mathbf{X} \to \mathbf{Y})} \,. \tag{5.94}$$

The right-hand side is known.

The Metropolis—not Metropolis-Hasting—algorithm picks $\pi(\mathbf{X} \to \mathbf{Y}) = \pi(\mathbf{Y} \to \mathbf{X})$ so that

$$R(\mathbf{X} \to \mathbf{Y}) = \frac{w(\mathbf{Y})}{w(\mathbf{X})} \tag{5.95}$$

is the ratio of analytically known desired weights. It can be shown that the acceptance probability $\alpha(\mathbf{X} \to \mathbf{Y}) = \min[1, R(\mathbf{X} \to \mathbf{Y})]$ will satisfy detailed balance.

Enough about theory, let's get on to implementing the above discussion into the following algorithm.

1. Start in a random location $\mathbf{X_0}$.

2. Propose a uniform distributed step. $\mathbf{Y}_i = \mathbf{X}_{i-1} + \theta \mathbf{U}_i$, where $\mathbf{U}_i = [-1, 1]$ are uniformly distributed numbers, and θ a parameter that controls the step size. The step $\mathbf{Y}_i$ is the proposed walker configuration. This is the simple Metropolis algorithm.

3. Evaluate
$$\alpha(\mathbf{X}_i \to \mathbf{Y}_i) = \min\left[1, \frac{w(\mathbf{Y}_i)}{w(\mathbf{X}_{i-1})}\right], \tag{5.96}$$
where w need not be normalized because the ratio cancels any normalization constants.

4. Evaluate
$$\mathbf{X}_i = \begin{cases} \mathbf{Y}_i & \text{if } \alpha(\mathbf{X}_{i-1} \to \mathbf{Y}_i) \geq U_i \\ \mathbf{X}_{i-1} & \text{if } \alpha(\mathbf{X}_{i-1} \to \mathbf{Y}_i) < U_i \,, \end{cases} \tag{5.97}$$
where U_i is another uniform random number $[0, 1)$. The optimal acceptance is 23% [18]. An acceptance of 15-50% is good.

5. A measurement of $f(\mathbf{X}_i)$ can be made at every step. But if the uncertainty is to be estimated using Eq. (5.85) and (5.86), correlations occur because the Markov chain samples are not independent and these correlations should be avoid. The correlations can be reduced by making a measurement of $f(\mathbf{X}_i)$ every n_m steps, where n_m is less than the total number of steps to be taken.

6. Increment i by one and go back to step 2. Terminate when the variance is small.

When starting from a random location in step 1, the probability at that location can be small. It can take some number of steps to reach a high-probability location in the distribution. Often one does not record the first n_b steps to avoid including these possible low-probability points in the calculation. This is referred to as a burn-in period.

5.4.4 Example: Random walk

The MC method applied to a two dimensional random walk provides an unusual solution to the Laplace equation in two dimensions. Consider a region R bounded by the curve C within which $\phi(x, y)$ satisfies the Laplace equation. Fixed Dirichlet boundary values $U(x, y)$ are specified at every point on C. The problem is to determine the value of ϕ at some interior point (m, n) on the solution grid which has been constructed over R.

We start a two dimensional random walk at the point (m, n). At each stage in the walk, the probability of a step to any one of the four closest neighbouring points on the grid is equal. We keep up the random walk until a step crosses C. The boundary value $U(x, y)$ at the point on the curve C that is crossed is recorded. This process is repeated many times, beginning each time at the point (m, n) and continuing each random walk until the curve C is crossed.

Let U_i be the boundary value where the ith walk terminated. It can be shown that

$$\phi(m, n) = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} U_i \,, \tag{5.98}$$

or, the value of ϕ is the mean of the terminal boundary values, averaged over the number of random walks made.

Consider the solution grid $x = 1, 2, 3, \ldots, 6$ and $y = 1, 2, 3, \ldots, 6$. Suppose the boundary conditions are $U(1, y) = U(x, 6) = 100$ and $U(x, 1) = U(6, y) = 0$. We want to find the solution of the Laplace equation at the point $(3, 3)$ using this MC random walk method. We will run 10,000 random walks to evaluate the potential function at the point $(3, 3)$. Consider the following code.

```
"""Solving the Laplace equation with a random walk."""

import numpy as np

N = 10000       # Number of random walks.
maxSteps = 2000 # Arbitrary large number to avoid infinite loop.
p, q = 3, 3     # Point to calculate the solution.
U = 0           # Sum of boundary values.
np.random.seed(0)

# Boundary conditions
xmin, xmax = 1, 6
ymin, ymax = 1, 6
Uxmin, Uxmax, Uymin, Uymax = 100, 0, 0, 100

# Take N random walks.
for n in range(N):
    # Starting position.
    x = p
    y = q
    step = 0
```

```
    # Continue to step until boundary is reached.
    while step < maxSteps:
        # Take a step.
        updown = np.random.choice([0,1])
        if updown:
            y += np.random.choice([-1,1])
        else:
            x += np.random.choice([-1,1])
        # Check is crossed boundary.
        if x < xmin:
            U += Uxmin
            break
        if x > xmax:
            U += Uxmax
            break
        if y < ymin:
            U += Uymin
            break
        if y > ymax:
            U += Uymax
            break
        step += 1
    # Boundary reached or maximum steps exceeded.
    if step== maxSteps: print("Warning, maximum steps reached")

# Print solution.
print("Solution",U/N,
      "at point ({},{}) in {} walks".format(p,q,N))
```

```
Solution 50.18 at point (3,3) in 10000 walks
```

5.4.5 Example: Variational Monte Carlo

This example is inspired by Alex Gezerlis [2] and Sølve Selstø [16]. We will apply the Metropolis algorithm to a helium atom, and produce an upper bound on the ground-state energy.

The helium atom consists of two electrons in orbit around a nucleus containing two protons and two neutrons. An approximate Hamiltonian for the system is

$$\hat{H} = -\frac{\hbar^2}{2m}\left(\nabla_1^2 + \nabla_2^2\right) - \frac{e^2}{4\pi\epsilon_0}\left(\frac{Z}{r_1} + \frac{Z}{r_2} - \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}\right), \tag{5.99}$$

where $Z = 2$ for helium. Using the Bohr radius a, we form the dimensionless distance $\bar{r} = r/a$ and write

$$\hat{H} = -\alpha^2 mc^2\left[\frac{1}{2}\left(\bar{\nabla}_1^2 + \bar{\nabla}_2^2\right) - \left(\frac{Z}{\bar{r}_1} + \frac{Z}{\bar{r}_2} - \frac{1}{|\bar{\mathbf{r}}_1 - \bar{\mathbf{r}}_2|}\right)\right], \tag{5.100}$$

where $e^2/(4\pi\epsilon_0) = \alpha^2 amc^2$ and $\hbar^2/m = \alpha^2 a^2 mc^2$ have been used. In terms of the hydrogen atom binding energy $-\alpha^2 mc^2 = 2E_1 = 27.2114\,\text{eV}$.

Let's now determine a trial wave function with which to try and solve the problem. We put each of the two particles with different spin states into the same single-particle state ϕ:

$$\psi_T(\mathbf{X}) = \phi(\bar{\mathbf{r}}_1)\phi(\bar{\mathbf{r}}_2)\,. \tag{5.101}$$

The position of the 1st particle, for example, is $(\bar{\mathbf{r}}_1)_x$, $(\bar{\mathbf{r}}_1)_y$ and $(\bar{\mathbf{r}}_1)_z$. The position vector $\mathbf{X}$ has $2 \times 3 = 6$ components.

For the wavefunction, we can start with the non-interacting case and include a variational parameter. We chose a hydrogen atom wavefunction

$$\phi(\bar{\mathbf{r}}) = \exp\left[-\alpha Z \bar{r}\right]\,, \tag{5.102}$$

where α—not to be confused with the fine-structure constant—is a variation parameter that allows the wavefunction to be different from the non-interacting case. Notice that the normalization has not been include since it will cancel when using the ratio of wavefunctions later on.

We define the local energy evaluated at the position of all the particles $\mathbf{X}$ as

$$E_L(\mathbf{X}) = \frac{\hat{H}\psi_T(\mathbf{X})}{\psi_T(\mathbf{X})}\,. \tag{5.103}$$

This is the local total energy. It is called local because it can be evaluated from all the particles at a specific location $\mathbf{X}$.

For convenience, we will divide this into kinetic and potential energy terms: $\hat{H} = \hat{T} + \hat{V}$. The local kinetic energy is

$$\hat{T}_L(\mathbf{X}) = \frac{\hat{T}\psi_T(\mathbf{X})}{\psi_T(\mathbf{X})}\,, \tag{5.104}$$

where

$$\hat{T} = -\frac{1}{2}\sum_{j=1}^{2} \bar{\nabla}_j^2 \tag{5.105}$$

will be evaluated using central differences. The local potential energy is

$$\hat{V}_L(\mathbf{X}) = \frac{\hat{V}\psi_T(\mathbf{X})}{\psi_T(\mathbf{X})} = V(\mathbf{X})\,. \tag{5.106}$$

The following code specifies the trial wavefunction and the local energy.

```
"""Calculate the helium atom ground-state energy
using the variational quantum Monte Carlo method.
"""

import numpy as np

# Trial wave function.
```

```
def psi(alpha,Z,r):
    r1 = np.linalg.norm(r[0,:])
    r2 = np.linalg.norm(r[1,:])
    return np.exp(-alpha*Z*(r1+r2))

# Calculate local kinetic energy term in Hamiltonian.
def Ekin(alpha,Z,r,h=0.001):
    npart, ndim = r.shape
    psiold = psi(alpha,Z,r)
    kin = 0
    for i in range(npart):
        numer = 0.0
        for j in range(ndim):
            rs = r[i,j] # Save position.
            r[i,j] = rs + h
            psip = psi(alpha,Z,r)
            r[i,j] = rs - h
            psim = psi(alpha,Z,r)
            r[i,j] = rs # Restore position.
            numer += psip - 2.0*psiold + psim
        laplace = numer / h**2
        kin += -0.5 * laplace / psiold
    return kin

# Calculate local potential energy term in Hamiltonian.
def Epot(Z,r):
    npart, ndim = r.shape
    noninteract = 0
    interact = 0
    for i in range(npart):
        noninteract += - Z/np.linalg.norm(r[i,:])
        for j in range(i+1,npart):
            interact += 1/np.linalg.norm(r[i,:]-r[j,:])
    return noninteract + interact
```

The variational energy is

$$E_V = \frac{\langle \psi_T | \hat{H} | \psi_T \rangle}{\langle \psi_T | \psi_T \rangle} \geq E_0 \,, \tag{5.107}$$

where E_0 is the ground state energy. By taking $\psi_T = \psi_T(\alpha)$ and $E_V = E_V(\alpha)$, and by minimizing $E_V(\alpha)$ with respect to α, we try to get close to E_0. We will not actually minimize E_V. We will calculate $E_V(\alpha)$ for different α and observe the minimum on a plot.

In coordinate space, Eq. (5.107) can be written as

$$E_V = \int d^d x \left(\frac{|\psi_T(\mathbf{x})|^2}{\int d^d |\psi_T(\mathbf{x})|^2} \right) \frac{\hat{H}\psi_T(\mathbf{x})}{\psi_T(\mathbf{x})} \,. \tag{5.108}$$

Identifying factors according to multidimensional weighted Monte Carlo integration gives

$$w(\mathbf{x}) = \frac{|\psi_T(\mathbf{x})|^2}{\int d^d |\psi_T(\mathbf{x})|^2} \quad \text{and} \quad f(\mathbf{x}) = \frac{\hat{H}\psi_T(\mathbf{x})}{\psi_T(\mathbf{x})} \,, \tag{5.109}$$

where the later expression is the local energy $E_L(\mathbf{x})$.

The procedure now is to draw samples from $w(\mathbf{x})$ using the Metropolis algorithm and use them to evaluate the local energy. The acceptance probability is determined by the ratio of the weight function at the proposed and current walker configurations. The denominator (integral) of the weight function cancels in the ratio and need not be considered. The algorithm will lead to a random walk that should asymptotically approach the desire weight distribution. The variational energy is approximated as

$$E_V = \frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} E_L(\mathbf{X_i}) \pm \frac{1}{\sqrt{\mathcal{N}-1}} \sqrt{\frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} E_L^2(\mathbf{X_i}) - \left[\frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} E_L(\mathbf{X}_i)\right]^2}. \tag{5.110}$$

The $\mathbf{X_i}$ are drawn from $|\psi_T(\mathbf{x})|^2$ via the Metropolis algorithm.

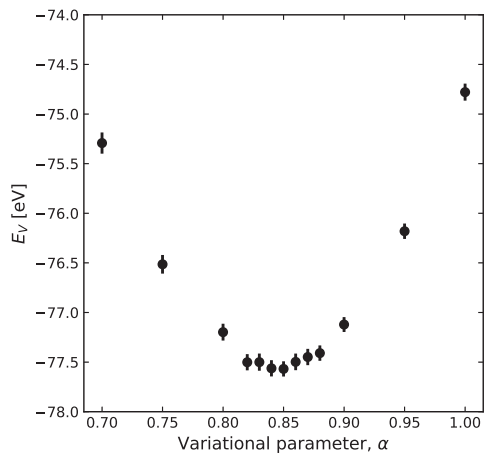
Consider the following code.

```
# Estimate mean and standard error of the mean.
def stats(x):
    n = len(x)
    xbar = np.sum(x) / n
    xsq  = np.sum(x**2) / n
    varxbar = (xsq - xbar**2) / (n-1)
    return xbar, np.sqrt(varxbar)

# Metropolis algorithm.
def vmc(npart,ndim,alpha,Z,iseed=8735):
    # Total number of variational energy measurements.
    Ncal= 10**4
    # Frequency which variational energy measurements are made.
    nm = 100
    # Step size.
    theta = 0.8
    # Calculate a random position for each particle
    # in the d-dimensional cube.
    np.random.seed(iseed)
    rolds = np.random.uniform(-1,1,(npart,ndim))
    # Trial wave function at initial positions
    # (initial walker configuration).
    psiold = psi(alpha,Z,rolds)
    # Initialization before loop.
    iacc, imeas = 0, 0
    eners = np.zeros(Ncal)
    for itot in range(nm*Ncal): # Total number of steps.
        # Propose step (walker configuration).
        rnews = rolds \
                + theta*np.random.uniform(-1,1,(npart,ndim))
        # Calculate trail wave function at
        # new walker configuration.
        psinew = psi(alpha,Z,rnews)
        # Calculate ratio of weights
        # (ratio of wave function modulus squared).
        psiratio = (psinew / psiold)**2
        # Test acceptance ratio.
        if psiratio >= np.random.uniform(0,1):
            rolds = np.copy(rnews)
```

```
            psiold = psinew
            iacc += 1
        # Record an energy measurement
        if (itot%nm) == 0:
            eners[imeas] = Ekin(alpha,Z,rolds) + Epot(Z,rolds)
            imeas += 1
    return eners, iacc/(nm*Ncal)

# Driving code.
iseed = 0
# Atomic number.
Z = 2
# Energy units in eV.
UNITS = 2*13.6057
# 2 particles in 3 dimensions.
npart, ndim = 2, 3
# Variational parameter alpha.
alpha = 0.85

# Call the algorithm.
energy, accrate = vmc(npart,ndim,alpha,Z,iseed)
print("acceptance rate",accrate)

# Calculate the MC integral.
mean, error = stats(energy)
print(f"alpha {alpha}",
      f"energy {mean*UNITS:.2f} +- {error*UNITS:.2f} eV")
```

```
acceptance rate 0.304572
alpha 0.85 energy -77.59 +- 0.24 eV
```

The figure shows the results of multiple runs with different values of the variational parameter. The minimum value of $-77.59 \pm 0.24\,\text{eV}$ can be compared to the experimentally measured value of $-78.98\,\text{eV}$.



5.5 PROBLEMS

1. **Expectation values**

   It is common to display the distribution of a random variable in a histogram.

   (a) Plot histograms of random variables distributed uniformly, according to a normal distribution and according to an exponential distribution.

   (b) Consider the standard normal distribution ($\mu = 0$ and $\sigma = 1$) and the exponential distribution with decay constant $\beta = 1$

   (c) Use the same number of bins in each histogram. Use histogram ranges $[0, 1)$, $[0, 5\beta)$ and $[-5\sigma, 5\sigma)$ for the uniform, normal and exponential distributions, respectively.

   (d) Calculate the expectation value in each bin and superimpose these values on the histograms.

   (e) Repeat the procedure for $10^3, 10^4$ and 10^5 random variables each and comment on the agreements between the histograms and the expectation values.

   You may use the `scipy.stats` statistics module in this problem.

2. **Random variable: angular distribution**

   Generate random variables having a $(1 + \cos^2\theta)$ distribution. For this question, use the inverse transform method, not the accept-reject method. Display your results in a finely binned histogram of frequency of occurrence versus $\cos\theta$. Superimpose on the histogram a plot of $(1 + \cos^2\theta)$ versus $\cos\theta$ to show you get agreement. You may only used a uniform random number generator.

   (a) Explain your method and any maths before your code.

   (b) Write code.

   (c) Show the histogram with the plot superimposed.

3. **Random variable: data distribution**

   Read in the file `function.csv` which contains a list of (x, y) values in csv format. Making this file is not part of the problem. You may only used a uniform random number generator.

   (a) Plot the data.

   (b) Calculate and plot the CDF.

   (c) Calculate and plot the inverse CDF.

(d) Using the inverse transform method, generate random variables distributed according to the input data and histogram the frequency of occurrence.

(e) Using the Monte Carlo method (von Neumann), generate random variables distributed according to the input data and histogram the frequency of occurrence.

(f) Discuss any differences in your results between question (d) and (e). In particular, the number of samples needed to get roughly the same smoothness in the distributions generated by the two methods.

4. **Drell-Yan proton–proton cross section**

Inspired by Klein and Godunov [19].

This is the simplist nontrivial particle physics Monte Carlo cross section you can calculate. It simulates the Drell-Yan process: $p + p \rightarrow \gamma^* \rightarrow \ell^+ + \ell^- + X$, where X is unobserved hadrons. But we will consider only the production process, not the decay.

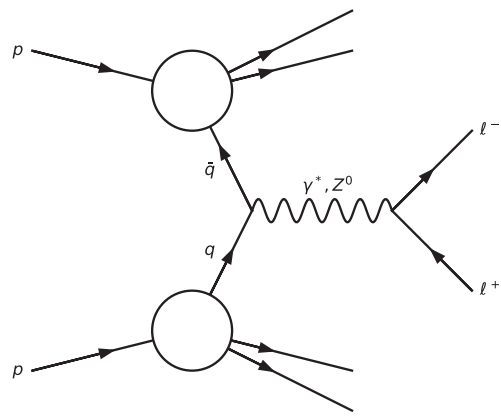


Figure 5.6 Drell-Yan process in proton–proton scattering.

The first complication we encounter in simulating this process is protons p do not interact to produce the photon γ^*, but rather the quarks q within the protons interact to product the photon. We consider the quarks that interact to have a fraction of the protons energy x.

To carry out the calculation, we need experimentally determined probability density functions (pdfs) for the probabilities that a quark of flavour i has proton fractional energy x. There are many such libraries of parton distribution functions available, all with different measurement, theoretical assumptions, approximations and modelling techniques. They can-

not currently be calculated from first principles. For simplicity, we will consider only the u-quark, d-quark, and a single contribution from the sea quarks (antiquarks). We will take the quarks to be massless. The following three analytic models for the pdf's will be used.

$$q_u(x) = 2.13\sqrt{x}(1-x)^{2.8}\,,$$
$$q_d(x) = 1.26\sqrt{x}(1-x)^{3.8}\,,$$
$$\bar{q}(x) = \bar{q}_u(x) = \bar{q}_d(x) = 0.27(1-x)^{8.1}\,.$$

(a) Plot the pdfs and calculate their normalizations. Notice that the pdf normalizations do not sum to one as we are not considering the gluon or other types of quarks in the proton.

In general, the quark-antiquark scattering cross section needs to be convoluted with each parton distribution function and summed over the different quark types (flavours). The simplicity of the Drell-Yan process is that the convolution is non-existent since the quark-antiquark scattering cross section does not depend directly on x. The Drell-Yan proton–proton differential cross section can be written as

$$\frac{d^2\sigma(x_1,x_2,q^2)}{dx_1dx_2} = \frac{4\pi\alpha^2}{9q^2}\sum_{a=u,u,d} e_a^2\left[q_a(x_1)\bar{q}_a(x_2) + \bar{q}_a(x_1)q_a(x_2)\right]\,,$$

where we will take the convention that x_1 is for the quark from the proton travelling in the positive z-direction and x_2 is for the quark from the proton travelling in the negative z-direction. The coupling factor $4\pi\alpha^2$ shows that the process is electromagnetic (the α) and a quark colour coupling factor (the 1/9) has been included. e_a^2 is the electric charge of the quark of type a in units of e, i.e. it is a simple fraction. $e_u = 2/3, e_d = -1/3, e_{\bar{u}} = -2/3, e_{\bar{d}} = 1/3$. The sum is over the valence quarks in a proton: two u-quarks and a d-quark.

(b) Write a program to calculate this differential cross section $d^2\sigma/dx_1dx_2$. Sample the cross section over x_1 and x_2 to calculate the cross section as a function of $q^2 = sx_1x_2$, were $\sqrt{s} = 13600\,\text{GeV}$. Take $\alpha = 1/137$. Plot the cross section over the range [1300,2000] GeV. What is your Monte Carlo efficiency?

The above cross section is an electromagnetic process in which a virtual photon γ^* is exchanged. Now include the electroweak interaction involving Z-boson exchange (Z^0 in the diagram above).

(c) Add the following Briet-Wigner distribution to the $1/q^2$ factor in the formula above so that the differential cross section has two terms:

$$\sigma_{\text{BW}}(M) \sim \frac{M_Z\Gamma_Z}{(M^2-M_Z^2)^2 + (M_Z\Gamma_Z)^2}\,,$$

where $M_Z = 91.2$ GeV and $\Gamma_Z = 2.5$ GeV. Notice that $q^2 = M^2$; these are just different symbol conventions. We will (incorrectly) take the coupling for the two processes to be equal and ignore the quantum mechanical interference between them. To make the Z-boson contribution prominent, use $\sqrt{s} = 1000\,\text{GeV}$, and plot over the range [5,200] GeV. What is your Monte Carlo efficiency?

5. **Importance sampling integration**

   Calculate a value for the integral

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

   using the importance sampling technique with weight function $w(x) = x^{-1/2}$.

6. **Event generation with importance sampling**

   (a) Consider the function

$$f(x) = x^{-4}$$

   from which we wish to generate events. Normally we would not have an analytic form for this function, and you should treat it as unknown.

   i. Sample the function $f(x)$ using the von Neumann method over the range $[1, 10)$. Histogram the accepted events for 10^5 trials. Explain your distribution.

   We will now sample the function $f(x)$ using importance sampling. Pick three different power-law weight functions

$$\omega(x) = x^{p_w}$$

   where $p_w = -3, -4, -5$.

   ii. Calculate the cumulative distribution function needed to draw random variables from the power-law distribution.

   iii. Plot the function $f(x)$ and the three weight functions on the same plot.

   iv. On another plot, overlay the three integrands $f(x)/\omega(x)$ on the same plot.

   v. Sample $f(x)/\omega(x)$ using the von Neumann method over the range $[1, 10)$. Histogram the accepted events for 10^5 trials. Discuss your distributions for the three cases.

   There is more than one way to normalize the distributions. One is to normalize by area. The histogram is first normalized to unity by dividing the content of each bin by the product of the total number

of entries m times the bin width Δx. Then the histogram pdf can be normalized by the integral of $f(x)$ which can be calculated at the same time as sampling. For the uniform sampling method we can calculate the integral using either the von Neumann method or the mean value method. For the importance sampling method we can calculate the integral using the mean value method.

vi. Normalize the distributions to the area of $f(x)$.

You might notice that using this normalization cause the first bin of the histogram to be less than the maximum function value.

vii. Can you explain why this is the case?

An alternative normalization is to scale the distribution by ratio of the function $f(x)$ maximum divided by the content in the first bin of the histogram.

viii. Normalize the distributions to the maximum of $f(x)$.

(b) Another method to improve sampling is to split the abscissa into slices or regions and sample each one separately—often in parallel. This can reduce the variation of the function and allow better sampling. The trick is to stitch the slices back together properly. For this problem, take $f(x) = 1/x$ and $\omega(x) = 1/x$.

i. Sample the function $f(x)$ using the von Neumann method (no weighting) over two ranges $[1, 6)$ and $[6, 11)$. Histogram the accepted events for 10^4 trials each on the same histogram. Normalize to the area of $f(x)$.

ii. Comment on how smooth the distribution is at $x = 6$.

iii. Calculate the cumulative distribution function needed to draw random variables from the power-law distribution.

iv. Sample $f(x)/\omega(x)$ using the von Neumann method over the ranges $[1, 6)$ and $[6, 11)$. Histogram the accepted events for 10^4 trials. Comment on the smoothness at $x = 6$.

(c) We now consider a less perfect case. We will again consider two slices $[1, 6)$ and $[6, 11)$, but this time will take

$$f(x) = \left(\frac{x}{2} + \frac{x^2}{2}\right)^{-3}$$

to not be a simple power-law—have more degrees of curvature. We will also importance sample with different power-law weight functions in the two slices.

i. Sample the function $f(x)$ using the von Neumann method (no weighting) over two ranges $[1, 6)$ and $[6, 11)$. Histogram the accepted events for 10^5 trials each on the same histogram. Normalize to the area of $f(x)$.

ii. Comment on how smooth the distribution is at $x = 6$.
iii. Calculate the cumulative distribution functions needed to draw random variables from the power-law distributions.
iv. Sample $f(x)/\omega(x)$ using the von Neumann method over the ranges $[1, 6)$ and $[6, 11)$. Histogram the accepted events for 10^5 trials. Comment on the smoothness at $x = 6$.

Once you have your code working, try to pick powers for the two weight function that optimize the sampling by giving high von Neumann efficiency values. I found indices of -4.5 and -5 worked well.

7. **Markov chain Monte Carlo integration**

Calculate the following integral using Markov chain Monte Carlo integration.

$$\int_0^1 \int_0^1 e^{\sin(xy)} dxdy\,.$$

Take the weight to be

$$\text{pdf}(x, y) = \text{pdf}(x)\text{pdf}(y) = 4xy\,.$$

8. **All methods of integration**

Consider the following integral

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

(a) Calculate the integral analytically.
(b) Calculate the integral using a library module.
(c) Calculate the integral and an estimate of the uncertainty using the mean value Monte Carlo method.
(d) Calculate the integral and an estimate of the uncertainty using the importance sampling Monte Carlo method.
(e) Calculate the integral and an estimate of the uncertainty using the Metropolis algorithm.

In all iterative cases, consider 10^6 iterations. No plot is needed in this problem.

9. **Markov Chain Monte Carlo: Ideal gas**

This problem is inspired by Mark Newman [20]. For a boson of mass m in a cubic box of length L,

$$E(n_x, n_y, n_z) = \frac{\pi^2 \hbar^2}{2mL^2} \left(n_x^2 + n_y^2 + n_z^2\right)\,,$$

where $n_x, n_y, n_z = 1, \ldots, \infty$ are positive quantum numbers.

For an ideal gas of N bosons that do not interact, the total energy is

$$E = \sum_{i=1}^{N} E\left(n_x^{(i)}, n_y^{(i)}, n_z^{(i)}\right) ,$$

where $n_x^{(i)}$ is the value of the quantum number n_x for the ith particle, and so forth.

You will calculate the total internal energy of the gas versus temperature using Markov chain Monte Carlo (MCMC).

Each particle can change state when one of its quantum numbers changes by ± 1. If all the other particles in the gas remain unchanged, this represents the minimal change of the gas which we call a MCMC step.

(a) Calculate the change in energy of the gas for a single step.

The Metropolis acceptance probability in going from state i to state j can be given by

$$P_{i\to j} = \begin{cases} 1 & \text{if} \quad E_j \leq E_i \\ \exp[-\beta(E_j - E_i)] & \text{if} \quad E_j > E_i \end{cases} ,$$

where $\beta = 1/(k_{\mathrm{B}}T)$. The temperature of the gas is T and k_{B} is the Boltzmann constant.

(b) Write a program to calculate the total internal energy of the gas using the MCMC method. You may take $m = \hbar = L = 1$ for simplicity. Take the number of particles to be $N = 1000$.

i. Start with all particles in the ground state and take $k_{\mathrm{B}}T = 10$.

ii. Select at random a particle, axis and sign for the first move (state change).

iii. Calculate the energy change and then either accept or reject the move according to the above Metropolis acceptance probability. If a particle is in an $n = 1$ state and we happen to choose to decrease the value of the quantum number n, then the move is always rejected because there is no lower value of n.

iv. Assume the ergodic theorem so that the energy for all steps are saved to estimate the average energy of the gas. However, include a burn-in period by not saving the energies for the first 100 000 steps.

v. Repeat the whole process for a total of 250 000 steps.

(c) Plot the internal energy of the gas versus step number.

(d) Run your simulation for different values $k_{\mathrm{B}}T$ and

i. Plot the equilibrium energy versus k_BT to see if you get the ideal gas law.
ii. Plot your data with statistical error bars.
iii. Indicate on the plot the ground state.
iv. Comment on your results.

10. **Variational Monte Carlo: Rubber-band helium**

Rubber-band helium is a Hamiltonian in which the Coulomb forces in Helium are replaced by Hooke's law forces:

$$H = -\frac{\hbar}{2m}\left(\nabla_1^2 + \nabla_2^2\right) + \frac{1}{2}m\omega^2(r_1^2 + r_2^2) - \frac{\lambda}{4}m\omega^2|\mathbf{r}_1 - \mathbf{r}_2|^2\,.$$

Pretend we do not know the solution and solve for a lower bound on the ground-state energy using the variational Monte Carlo method. You may compare your result with the exact ground state energy

$$E_0 = \frac{3}{2}(1 + \sqrt{1-\lambda})\hbar\omega\,.$$

If one takes the ground-state wavefunction for a three-dimensional harmonic oscillator as the trial wavefunction, the expectation value of the Hamiltonian can be shown to be

$$\langle H\rangle = 3\hbar\omega\left(1 - \frac{\lambda}{4}\right)\,.$$

Your ground-state energy bound should be lower than this.

11. **Path integral harmonic oscillator**

Inspired by G. Peter Lepage [21].

The evolution of a position eigenstate $|x_i\rangle$ from time t_i to time t_f can be computed using the path integral

$$\langle x_f|e^{-\hat{H}(t_f-t_i)/\hbar}|x_i\rangle = \int \mathcal{D}x(t)e^{iS[x]/\hbar}\,,$$

where $\hat{H}$ is the Hamiltonian operator and $\int \mathcal{D}x(t)$ is the sum over all possible particle paths from $x(t_i) = x_i$ to $x(t_f) = x_f$. The classical action is

$$S[x] = \int_{t_i}^{t_f} dtL(x,\dot{x}) = \int_{t_i}^{t_f} dt\left[\frac{1}{2}m\dot{x}(t)^2 - V(x(t))\right]\,,$$

evaluated for each path $x(t)$.

Since the Euclidean path integral is preferred for numerical computations, we make a Wick rotation $it \to t$ and write

$$\langle x_f | e^{-\hat{H}(t_f - t_i)/\hbar} | x_i \rangle = \int \mathcal{D}x(t) e^{-S[x]/\hbar}$$

with

$$S[x] = \int_{t_i}^{t_f} dt L(x, \dot{x}) = \int_{t_i}^{t_f} dt \left[\frac{1}{2} m \dot{x}(t)^2 + V(x(t)) \right] .$$

We now discretize the problem. For an arbitrary path $x(t), t_i \leq t \leq t_f$, discretizing t, gives $x = \{x(t_0), x(t_1) \ldots x(t_N)\}$, where $\Delta t = (t_f - t_i)/N$ and $T = t_f - t_i = N\Delta t$.

The discretized integral over all paths is

$$\int \mathcal{D}x(t) \to A \int_{-\infty}^{\infty} dx_1 dx_2 \ldots dx_{N-1} ,$$

where we do not integrate over the endpoints which are held fixed. The normalization factor is given by

$$A = \left(\frac{m}{2\pi\hbar\Delta t} \right)^{N/2} ,$$

for a Euclidean path integral.

The discretized action is

$$S_{\text{lat}}[x] = \sum_{n=0}^{N-1} \left[\frac{1}{2} m \frac{(x_{n+1} - x_n)^2}{\Delta t} + V(x_n)\Delta t \right] ,$$

where for the two endpoints $x_0 = x_N \equiv x$.

Putting it all together, we have

$$\langle x | e^{-\hat{H}T/\hbar} | x \rangle \approx A \int_{-\infty}^{\infty} dx_1 \ldots dx_{N-1} e^{-S_{\text{lat}}[x]/\hbar} , \tag{5.111}$$

which is an $(N-1)$-dimensional integration problem.

Take $\Delta t = 1/2$ and $N = 8$ so the right-hand side of Eq. (5.111) becomes a seven-dimensional integral. Solve the integral on the right-hand side of Eq. (5.111) for a one-dimensional harmonic oscillator potential with $\omega = m = \hbar = 1$ using multidimensional integration. For enough accuracy, I found it necessary to use the `vagas` [22] adaptive multidimensional integration package.

Compare your result with the standard quantum mechanics result given by

$$\langle x|e^{-\hat{H}T}|x\rangle \approx |\langle x|E_0\rangle|^2 e^{-E_0 T}$$

with $E_0 = 1/2$ and

$$\langle x|E_0\rangle = \frac{e^{-x^2/2}}{\pi^{1/4}}$$

for the choices of $\omega = m = \hbar = 1$ used above.

12. **Radioactive decay chains**

In this problem, you will write a general function or class that can simulate simple radioactive decay chains to give the number of atoms of each isotope in the chain versus time.

The left diagram in Fig. 5.7 shows a chain reaction starting from ^{213}Bi and ending at ^{209}Bi. This is the alpha decay of Bismuth-213 which can be consider for targeted radionuclide therapy. The right diagram in Fig. 5.7 shows a chain reaction starting from ^{220}Rn and ending at ^{208}Pb. Radon is responsible for the majority of the mean public exposure to ionizing radiation. The vertical arrows represent alpha decay in which the number

Figure 5.7 Left: Bismuth-213 alpha decay chain. Right: Radon-220 alpha decay chain.

of neutrons and protons in the isotope decreases by two each. While the upper-right directed arrows represent beta decay in which a neutron changes into a proton, and emits a beta particle and antineutrino. For our purposes here, the type of decay does not matter. Also shown in the diagrams are the half-life of each isotope along with branching fractions were more than one type of decay of an isotope is possible.

The probability for a single atom to decay in a time interval of length t is given by

$$p(t) = 1 - 2^{-t/\tau}\,,$$

where τ is the half-life of the decaying atom.

(a) Simulate the decay of the atoms over time, mimicking the randomness of the decay using random numbers. Do not use an exponential decay model. Take a time interval of $t = 1\,\text{s}$. Do not allow a given atom to decay twice in a single time interval. Consider at time zero, a sample of 10 000 ^{213}Bi atoms or 10 000 ^{220}Rn atoms, and zero of the other atoms.

(b) For each decay chain, make a single log-plot of the number of each atom versus time over 10 hours. Label the graph for each atom.

(c) Write a general function or class that can perform the decays for an arbitrary chain reaction, then test your code on the two decay chains in Fig. 5.7. You can do this by creating a record for each isotope containing the half-life and branching ratio. The particular decay chain can then be formed by stringing the isotope records together in the proper order.

Isotopes with half-life less than the time-step can be removed from the chain. You might find it easier to write specific code for one of the decay chains first and then generalize it to be decay chain independent.

CHAPTER 6

Data analysis

Data analysis techniques can appear specific to the field of physics they are applied to. Formal techniques are generally needed in the domain of experimental physics. A large part of data analysis in all experimental science should be the estimation of uncertainties on measurements, or the determination of uncertainties from measurements on physical quantities. Throughout, we may sometimes replace the word uncertainty with the easier-to-say word error with the understanding that we do not make errors.

First, we discuss the propagation of uncertainties and the effectiveness of the Monte Carlo method in their determination. We briefly mention the presentation of uncertainties and some pathological issues that can arise. The main discussion covers the likelihood function and parameter estimation—with a small discussion of the goodness of fit. We conclude with the topic of hypothesis testing.

6.1 PROPAGATION OF UNCERTAINTIES

Consider the propagation of uncertainties for the case of a known function. For simplicity, consider a function of just two variables $z = f(x, y)$, where the independent variables (x, y) have been measured N times. What is the uncertainty on z? To first order

$$\sigma_z^2 = \left(\frac{\partial f}{dx}\right)^2 \sigma_x^2 + 2\frac{\partial f}{\partial x}\frac{\partial f}{\partial y}\sigma_{xy} + \left(\frac{\partial f}{dy}\right)^2 \sigma_y^2 \,, \tag{6.1}$$

where σ_x and σ_y are the uncertainties on x and y, respectively, and

$$\sigma_{xy} = \frac{1}{N}\sum_{i=0}^{N-1}(x_i - \bar{x}_i)(y_i - \bar{y}_i) \tag{6.2}$$

is the covariance.

DOI: 10.1201/9781003660873-6

If (x, y) are independent and random, the covariance vanishes and

$$\sigma_z = \left[\left(\frac{\partial f}{\partial x} \right)^2 \sigma_x^2 + \left(\frac{\partial f}{\partial y} \right)^2 \sigma_y^2 \right]^{1/2} . \tag{6.3}$$

We refer to this as adding the errors in quadrature.

When (x, y) are not independent, that is, they are correlated, $\sigma_{xy} \neq 0$. The quantity

$$\frac{\partial f}{\partial x} \frac{\partial f}{\partial y} \sigma_{xy} \tag{6.4}$$

can be positive or negative. In either case—due to the Schwarz inequality—

$$\sigma_z \leq \left| \frac{\partial f}{dx} \right| \sigma_x + \left| \frac{\partial f}{dy} \right| \sigma_y \, . \tag{6.5}$$

Often we take equality in Eq. (6.5) to avoid underestimating the uncertainty; this is called being-conservative. The above equality formula is referred to as adding the errors linearly.

The uncertainties on the variables σ_x and σ_y can be Gaussian distributed, but σ_z can be highly asymmetric and not well behaved. What happens when $f(x, y)$ is not simple or known? For example, the function $M^2 = m_1^2 + m_2^2 + 2(E_1 E_2 - \mathbf{p}_1 \cdot \mathbf{p}_2)$ that we will soon encounter. This is a case where MC simulation can be used. The method will take account of all correlations and nonlinearities automatically. But we require an accurate model for the process f. The MC method is also a method of choice for implementing or studying resolution-function smearing.

6.2 KINEMATICS

Particle kinematics provides an example of a complicated multivariable formula and nontrivial propagation of uncertainties. Often the laboratory (lab) frame in which the particles interact and the centre-of-mass (CoM) frame are not the same. Typically a physical theory or model is developed in the CoM frame, where there is a high degree of symmetry, then the formula are Lorentz transformed to the lab frame where the measurements or observations are made. The resulting formula in the lab frame, for example, an angular distribution can be complicated. We will take the approach of starting from the Lorentz four-vectors in the CoM frame and Lorentz boosting and, possibly, rotating them into the lab frame.

We will make use of the following energy E momentum $\mathbf{p}$ four-vector P notation.

$P = (E, p_x, p_y, p_z)$	Cartesian coordinates,
$P = (E, p, \theta, \phi)$	spherical coordinates,
$P = (E, p_T, \phi, p_z)$	cylindrical coordinates,
$P = (E, p_T, \phi, y)$	where y is the rapidity.

For a particle of mass m, $m^2 = P \cdot P$ is a constraint that reduces the number of independent components.

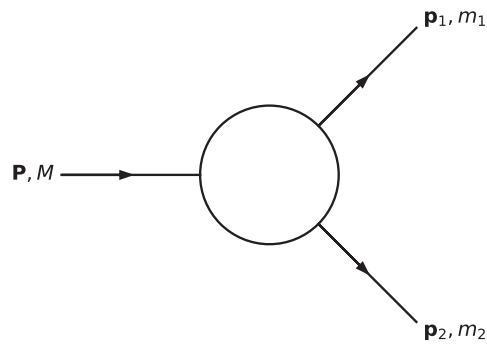


Figure 6.1 Two-body kinematics.

Consider the two-body decay process as shown in Fig. 6.1. In the rest frame of a particle of mass M, $\mathbf{p} = 0$ ($E = M$). From conservation of energy and momentum

$$E_1 = \frac{M^2 - m_1^2 + m_2^2}{2M}, \quad E_2 = \frac{M^2 - m_2^2 + m_1^2}{2M}, \quad M = E_1 + E_2 . \tag{6.6}$$

$$|\mathbf{p}_1| = |\mathbf{p}_2| = \frac{[(M^2 - (m_1 + m_2)^2)(M^2 - (m_1 - m_2)^2)]^{1/2}}{2M} . \tag{6.7}$$

In terms of four-vectors in the decaying particle's rest frame

$$\begin{aligned} P &= (M, 0, 0, 0) , & P^2 &= M^2 , \\ P_1 &= (E_1, \mathbf{p}_1) , & P_1^2 &= m_1^2 = E_1^2 - \mathbf{p}_1 \cdot \mathbf{p}_1 , \\ P_2 &= (E_2, \mathbf{p}_2) , & P_2^2 &= m_2^2 = E_2^2 - \mathbf{p}_2 \cdot \mathbf{p}_2 . \end{aligned}$$

The angles are not determined from two-body decay kinematics. They are determined by physics. They depend on the particle's spin and structure of the decay vertex. The angular distributions typically includes powers of $\sin\theta$, $\cos\theta$, and constants. The azimuthal angle ϕ is also present when not summing over particle polarizations.

For isotropic (phase-space) decay, the angular density is proportional to the solid angle:

$$d\Omega = d(\cos\theta) d\phi . \tag{6.8}$$

In this case, $\cos\theta$ is uniform over $(2u_1 - 1)$ and ϕ uniform over $2\pi u_2$, where u_1 and u_2 are uniform random variables on the interval $[0, 1)$.

The momentum components can be calculated as follows. Generate $\cos\theta$, calculate $\sin\theta = (1 - \cos^2\theta)^{1/2}$, then determine

$$\begin{aligned} p_z &= p\cos\theta , \\ p_T &= p\sin\theta . \end{aligned} \tag{6.9}$$

Generate ϕ and calculate

$$\begin{aligned} p_x &= p_T \cos\phi\,, \\ p_y &= p_T \sin\phi\,. \end{aligned} \tag{6.10}$$

For an arbitrary boost direction $\hat{\beta}$, the Lorentz transformation is

$$\begin{aligned} E' &= \gamma(E + \mathbf{p}\cdot\beta)\,, \\ \mathbf{p}' &= \mathbf{p} + (\gamma - 1)(\mathbf{p}\cdot\hat{\beta})\hat{\beta} + \gamma E\beta\,, \end{aligned} \tag{6.11}$$

where

$$\gamma = \frac{E'}{M} \quad \text{and} \quad \beta = \frac{\mathbf{p}'}{E'}\,, \quad \text{with} \quad \gamma^2 = (1-\beta^2)^{-1}\,. \tag{6.12}$$

The Lorentz invariant mass is

$$M^2 = (P_1 + P_2)^2 = P_1^2 + P_2^2 + 2P_1\cdot P_2 = m_1^2 + m_2^2 + 2(E_1E_2 - \mathbf{p}_1\cdot\mathbf{p}_2) \tag{6.13}$$

with $m_1^2 = E_1^2 - \mathbf{p}_1^2$ and $m_2^2 = E_2^2 - \mathbf{p}_2^2$.

6.2.1 Example: Propagation of uncertainties

This example applies propagation of uncertainties using the inverse transform technique to a relativistic kinematics problem: two-body decay. We will write a program to calculate the four-vectors of particles A, B and C for 100,000 decays. Let's take the mass of the particles to be $M = 80.4\,\text{GeV}$, $m_1 = 4.2\,\text{GeV}$ and $m_2 = 1.3\,\text{GeV}$. These masses are for the $W^- \to b\bar{c}$ decay process, which is arbitrary for this problem. I suggest you histogram all four components of each of the three four-vectors to validate your code. Also as a check, histograms of the sum of each four-vector component of the two daughter particles should equal the four-vector components of the mother particle. For conciseness of presentation these histograms are not presented here.

First, we generate a pseudo-data set to which we will apply the uncertainties. Consider the two-body decay of particle A in its rest frame: $A \to B + C$. In the CoM frame, particle A (mother) is at rest and has energy equal to it's rest mass. The energy of particles B and C (daughters) are determined by conservation of energy. The momentum of particles B and C are determined by conservation of momentum.

The physics of the decay process determines the angular distribution of the momentum vectors. For simplicity, we will take this distribution to be isotropic in three dimensions (isotropic phase-space).

Now we boost the system of particles into the lab frame, i.e. the frame of the source of the mother particles. Let's consider a source that emits the mother particles with a single velocity $\boldsymbol{\beta}$ isotropic in all directions. Boost the three-particle system to the frame with $-\boldsymbol{\beta}$, using $|\boldsymbol{\beta}| = 0.5$.

I suggest you histogram all the components of each four-vector to validate your code. Also, the Lorentz invariant quantities should be identical in all frames. I suggest you check them.

Once we have the boost working, we simulate the source producing the mother particle with a direction still isotropic but with a velocity magnitude distributed according to a beta function with $\alpha = 2$ and $\beta = 5$.

Now comes the propagation of errors (uncertainties). First histogram the invariant mass of the two particle system. The idea is that we do not observe the decaying mother particle and must infer it from the two final-state daughter particles. The invariant mass histogram should have all entries in a single bin at the decaying particle's mass.

Clearly, any uncertainty in the energy, momentum or angles of the decay particles propagates nontrivially to an uncertainty in the invariant mass. Hence, we are using this MC method for propagation of errors.

Let's work in cylindrical coordinates. For simplicity, we consider only an uncertainty in p_T; assuming E, p_z, θ, ϕ are measure exactly. Because of uncertainties, the four-vector invariant mass values will no longer be equal to the particle's mass. Let's take the uncertainty in p_T of both daughter particles to be $\sigma = 0.1 p_T$. We also assume the uncertainty is distributed with a Gaussian distribution of mean p_T and standard deviation σ. We often refer to this type of uncertainty as a Gaussian smearing of the p_T component of momentum.

We histogram the two-particle invariant mass before and after p_T resolution smearing, and calculate the mean and standard deviation of the invariant mass distribution.

Quantum mechanics does not allow a resonance particle to have a single mass. The energy E of a resonance is distributed according to a relativistic Breit-Wigner distribution:

$$p(E) = \frac{2E}{\pi} \frac{M\Gamma}{(E^2 - M^2)^2 + (M\Gamma)^2} \Theta(E)\,, \tag{6.14}$$

where M and Γ are the resonance's natural mass and width. Note that the Breit-Wigner distribution cannot be normalized and it is not the Cauchy distribution. You will need to employ an approximate method to smear the resonance.

The following code is run for different settings of three logical variables: `distributed` (the mother particle's velocity is distributed according to a beta distribution), `resonance` (include the mother particle's natural width) and `jer` (include p_T smearing).

```
"""Monte Carlo propogation of uncertainties."""

import numpy as np
from scipy import stats # for beta function

# This cell just contains function definitions.
# Four-vector convention is (E,px,py,pz).
# Metric convention is (+1,-1,-1,-1).

np.random.seed(1)

def m(p):
    # Calculate invariant mass of a 4-vector.
    m2 = p[0]*p[0] - np.dot(p[1:],p[1:])
    if m2 <= 0:
```

```
        print("m Warning: negative mass squared",m2,"found for 4-vector",p)
        m2 = 0.0
    return np.sqrt(m2)

def boost(gamma,beta,p):
    # Boost 4-vector to lab frame.
    b = np.sqrt(np.dot(beta,beta))
    b_hat = beta / b
    Ep = p[0]
    pp = p[1:]
    E = gamma * (Ep + np.dot(pp,beta))
    P = pp + (gamma-1.0) * np.dot(pp,b_hat) * b_hat + gamma*Ep*beta
    if E < np.sqrt(np.dot(P,P)): print("boost problem")
    return np.concatenate((E,P),axis=None)

def res(pT):
    # pT resolution function.
    sigma = 0.1 * pT
    u = np.random.normal()
    return pT + sigma*u

def bw(M,G):
    # Breit-Wigner distribution using hit and miss.
    # Take range to be 20 times the width.
    r = 20*G
    if M-r < 0: print("bw: range less than zero",M-r)
    while True:
        u1 = np.random.rand()
        u2 = np.random.rand()
        E = (2*u1 - 1) * r + M
        max = 2*E / (np.pi*M*G)
        fmax = u2*max
        f = (2*E/np.pi) * (M*G) / ((E*E - M*M)**2 + (M*G)**2)
        if (f > max): print("bf: function larger than maximum",f,max)
        if (f > fmax): break
    return E

def gencms(resonance,debug):
    # Generate an isotropic decay in CoM.
    # Return 4-vectors of all three particles.

    # Particles masses and resonance width.
    M  = 80.4
    G  =  2.1
    m1 =  4.2
    m2 =  1.3

    # Smear resonance mass if required.
    if resonance: M = bw(M,G)

    # Set the 4-vector of the mother resonance particle.
    Pcms = [M,0.0,0.0,0.0]

    # Calculate the kinematics of the daughter particles.
    E1 = (M*M - m2*m2 + m1*m1) / (2*M)
    E2 = M - E1
    p1 = np.sqrt( (M*M - (m1+m2)**2) * (M*M - (m1-m2)**2) ) / (2*M)
    p2 = p1

    # Isotropic angles in rest frame.
    u1 = np.random.rand()
    u2 = np.random.rand()
    cos = 2*u1 - 1
    sin = np.sqrt(1-cos*cos)
    phi = 2*np.pi * u2
    pT = p1 * sin
    px = pT * np.cos(phi)
```

```
    py = pT * np.sin(phi)
    pz = p1 * cos
    p1cms = [E1, px, py, pz]
    p2cms = [E2,-px,-py,-pz]

    # Check everything looks OK. This will produce way too
    # much output unless only a few events are simulated.
    if debug:
        print("particle 1: E = ",E1,"p = ",p1)
        print("particle 2: E = ",E2,"p = ",p2)
        print("particle 1:",p1cms)
        print("particle 2:",p2cms)
        print("particle 1 mass",m(p1cms))
        print("particle 2 mass",m(p2cms))
        P = np.add(p1cms,p2cms)
        print("p1 + p2",P)
        print("p1 + p2 mass",m(P))
    #
    return Pcms, p1cms, p2cms

# This stores all 4-vectors and creates no output.
debug = False
p  = []
p1 = []
p2 = []

# Particles masses and resonance width.
m1 = 4.2
m2 = 1.3
N = 100000
distributed = True  # Distribution of boost velocities.
resonance   = True  # Give resonance BW width.
jer         = True  # Jet energy resolution.
print("distribution:",distributed)
print("width:       ",resonance)
print("smearing:    ",jer)

# Loop over events.
for _ in range(N):

    # Generate isotropic decay in CoM frame.
    Pcms, p1cms, p2cms = gencms(resonance,debug)

    # Pick boost magnitude.
    b = 0.5
    if distributed:
        u = np.random.rand()
        b = stats.beta.ppf(u,2,5)

    # Pick boost direction (isotropic phase space).
    u1 = np.random.rand()
    u2 = np.random.rand()
    cos = 2*u1 - 1
    b3 = b*cos
    bT = np.sqrt(b*b-b3*b3)
    phi = 2*np.pi * u2
    b1 = bT * np.cos(phi)
    b2 = bT * np.sin(phi)
    beta = np.array([b1,b2,b3])
    gamma = 1.0 / np.sqrt(1.0-np.dot(beta,beta))
    if debug: print("gamma",gamma,"beta",beta)

    # Boost to lab frame.
    Plab  = boost(gamma,beta,Pcms)
    p1lab = boost(gamma,beta,p1cms)
    p2lab = boost(gamma,beta,p2cms)
    #  Check Lorentz invariants still invariant.
```

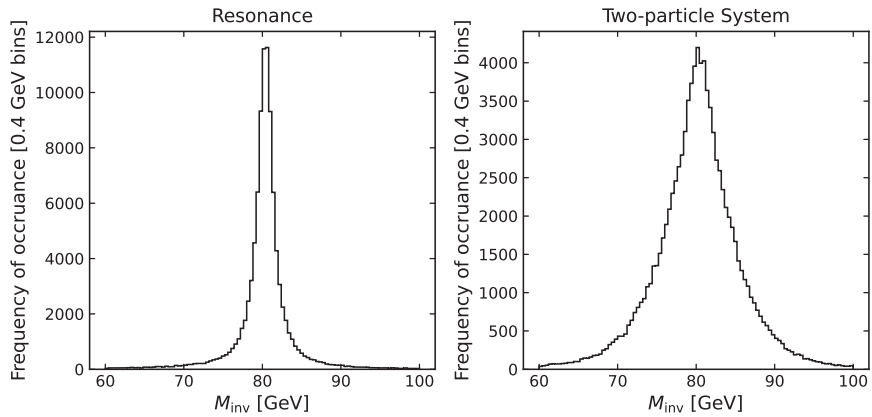
```
    if debug:
        print("decay particle mass",m(Plab))
        print("particle 1 mass",m(p1lab))
        print("particle 1 mass",m(p2lab))

    # Smear transverse momentum of final-state particles.
    if jer:
        pT = np.sqrt(p1lab[1]*p1lab[1]+p1lab[2]*p1lab[2])
        cos = p1lab[1] / pT
        sin = p1lab[2] / pT
        pT = res(pT)
        p1lab[1] = pT * cos
        p1lab[2] = pT * sin
        p1lab[0] = np.sqrt(m1**2+np.dot(p1lab[1:],p1lab[1:]))

        pT = np.sqrt(p2lab[1]*p2lab[1]+p2lab[2]*p2lab[2])
        cos = p2lab[1] / pT
        sin = p2lab[2] / pT
        pT = res(pT)
        p2lab[1] = pT * cos
        p2lab[2] = pT * sin
        p2lab[0] = np.sqrt(m2**2+np.dot(p2lab[1:],p2lab[1:]))
    p.append(Plab)
    p1.append(p1lab)
    p2.append(p2lab)
# End of loop.
```

```
distribution: True
width:        True
smearing:     True
```

```
"""This code makes the plots."""

from matplotlib import pyplot as plt
plt.style.use('./mystyle.mplstyle')

# Stack the arrays to make them more accessible.
p  = np.vstack(p)
p1 = np.vstack(p1)
p2 = np.vstack(p2)

# Calculate the invariant mass of the resonance and
# the two-particle system.
minv  = []
minvp = []
for i in range(len(p)):
    ptot = np.add(p1[i,:],p2[i,:])
    minvp.append(m(p[i,:]))
    minv.append(m(ptot))

print(f"Resonance  invariant mass mean {np.mean(minvp):.2f} TeV"
      f" and standard deviation {np.std(minvp):.2f} TeV")
print(f"2-particle invariant mass mean {np.mean(minv ):.2f} TeV"
      f" and standard deviation {np.std(minv ):.2f} TeV")

# Make the plots.
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1,2,1)
n, bins, patchs = plt.hist(minvp,range=(60,100),bins=100,histtype='step')
plt.title("Resonance")
plt.ylabel("Frequency of occruance [0.4 GeV bins]")
plt.xlabel(r"$M_\mathrm{inv}$ [GeV]")

ax = fig.add_subplot(1,2,2)
```

```
n, bins, patchs = plt.hist(minv,range=(60,100),bins=100,histtype='step')
plt.title("Two-particle System")
plt.ylabel("Frequency of occruance [0.4 GeV bins]")
plt.xlabel(r"$M_\mathrm{inv}$ [GeV]")

plt.tight_layout()
plt.savefig('6_2_propagation.pdf')
plt.show()
```

```
Resonance  invariant mass mean 80.07 TeV and standard deviation 5.44 TeV
2-particle invariant mass mean 80.02 TeV and standard deviation 6.83 TeV
```



The figure on the left shows the mother particle's mass. The width of the distribution is due to the intrinsic width of the resonance. The figure on the right shows the invariant mass of the two daughter particles. The width of the distribution includes the width of the mother particle and the p_{T} smearing. Notice that the shape of the distribution is neither Breit-Wigner nor Gaussian.

We see that the width and smearing has changed the mean by a fraction of a percent. Making runs without the width and without smearing show the means in both cases to be identical to what is input. We notice the two-particle width which includes the extra smearing to be about 1.4 GeV wider. The table shows the mass and RMS width for all possible permutations of the three logical flags. Some setting are useful for validation.

$M = 80.4\,\mathrm{GeV}$, $\Gamma = 2.1\,\mathrm{GeV}$ (FWHM), $\sigma = 0.1 p_T$.

Mass [GeV]	RMS [GeV]	Distribution	Width	Smearing
80.400	0	F	F	F
80.400	0	T	F	F
80.066	5.411	T	T	F
80.333	4.143	T	F	T
80.018	6.834	T	T	T

6.3 STATISTICAL ERROR BARS

In this section, we begin a discussion of statistical uncertainties. First, we will consider the question of how to present graphical error bars [23]. Graphical error bars can be considered a matter of presentation or the convention of a statistical practice. Sometimes the error bars also include systematic uncertainties or theory uncertainty bars. Uncertainty bands can be used instead of bars, the information is the same.

For plots, we can show error bars along the abscissa x and ordinate y. The error on y often come from the propagation of the error on the independent variable x to the dependent variable y. For histograms, the error in x is the bin width which is usually implied. The y-error is often statistical only.

One approach is simply not to show any error bars. This could be the case if the observed data are exact. But more often, it's when the uncertainty is not the main point of the plot, although intrinsic errors are still present.

6.3.1 Poisson $\sqrt{n}$ and binomial methods

The $\sqrt{n}$-method of displaying error bars is the most common and is often performed automatically by the graphics library module if the option to show error bars is selected. Say we measure integer n and assume it follows a Poisson distribution with some unknown mean μ. We can estimate $\mu = n$, which gives $\sigma^2 = \mu = n \Rightarrow \sigma = \sqrt{n}$. The data and error bars can be represented as $n \pm \sqrt{n}$. This is the most common approach. It is simple and easy to communicate. This choice of error bar is usually called the standard deviation.

This method can potentially be misleading for two important cases: 0 ± 0 and 1 ± 1. The case 0 ± 0 is not a good description if $n \neq 0$ has some finite probability. Sometimes the standard deviation is not a good description of fluctuations. Poisson errors are not recommended for $n = 0, 1$ unless the tails of the distribution, where there are few events, are not the region of interest.

If the observed values do not follow a Poisson distribution, use the probability model that the data does follows. An example is the measurement of efficiency. Consider the case of m events passing from a total of N tries. Then m should be binomially distributed. An estimate of the efficiency is $\epsilon = m/N$. The variance of a binomial distributed variable m is $V[m] = N\epsilon(1-\epsilon)$. The standard deviation of the efficiency is

$$\sigma_\epsilon = \frac{\sqrt{N\epsilon(1-\epsilon)}}{N} = \frac{1}{N}\sqrt{m\left(1 - \frac{m}{N}\right)}\,. \tag{6.15}$$

The above expression suffers from the same pathologies as the Poisson distribution for $m = 0$ and $m = 1$. In addition, it also has problems with $m = N$

and $m = (N-1)$. Consider each case of $\sigma(m)$ in turn:

$$\begin{aligned}
\sigma(0) &= 0\,. \\
\sigma(1) &= \frac{1}{N}\left(1-\frac{1}{N}\right)^{1/2} \approx \frac{1}{N}\left(1-\frac{1}{2N}\right) \approx \frac{1}{N}\,. \\
\sigma(N-1) &= \frac{1}{N}\left[(N-1)\left(1-\frac{N-1}{N}\right)\right]^{1/2} \approx \frac{1}{N}\,. \\
\sigma(N) &= 0\,.
\end{aligned}$$

I do not recommended using binomial error estimation for $\epsilon \approx 0$ or $\epsilon \approx 1$.

6.3.2 Confidence intervals

A more statistically sound approach to representing error bars is to use confidence intervals. This is a frequentist concept. The idea is to extend the error bars only over the physical range; $n = 0$ (Poisson) and $m = 0$, $m = N$ (binomial) are bad. Using confidence intervals, the error bars will generally be asymmetric about the measured value. The method will approach $\pm\sigma$ standard error bars for Poisson and binomial distributed uncertainties in the case of large m or $(N - m)$.

Again, take n as a Poisson distributed observation with mean μ. The confidence level of the lower limit is $1-\alpha$, while the confidence level of the upper limit is $1-\beta$. The $F^{-1}_{\chi^2}(x,n) \equiv \chi^2$ distribution is the quantile of a gamma distribution which is the CDF of the Poisson distribution (see Sec. 5.1.2.6). The error bars can be calculated using

$$\begin{aligned}
\nu_{\text{lo}} &= \frac{1}{2}F^{-1}_{\chi^2}(\alpha; 2n)\,, \\
\nu_{\text{up}} &= \frac{1}{2}F^{-1}_{\chi^2}(1-\beta; 2(n+1))\,.
\end{aligned} \tag{6.16}$$

The function $F^{-1}_{\chi^2}$ is available in the Python library module `scipy.stats.chi2`.

It is common to take the 68.3% central confidence interval. In which case $\alpha = \beta$ and $1 - \alpha - \beta = 0.683$ so that $\alpha = \beta = 0.159$. You then display

$$\bar{\nu}^{+(\nu_{\text{up}}-\bar{\nu})}_{-(\bar{\nu}-\nu_{\text{lo}})}\,. \tag{6.17}$$

For the case of a binomial process, F^{-1}_F is quantile of the F-distribution. A useful Python library module in this case is `scipy.stats.binom`.

Figure 6.2 shows the confidence interval method applied to three simple distributions.

6.4 PARAMETER ESTIMATION

Sometimes a statistical model for the data is unknown. We can employ statistical methods to help estimate the parameters of a parametric representation of the model.

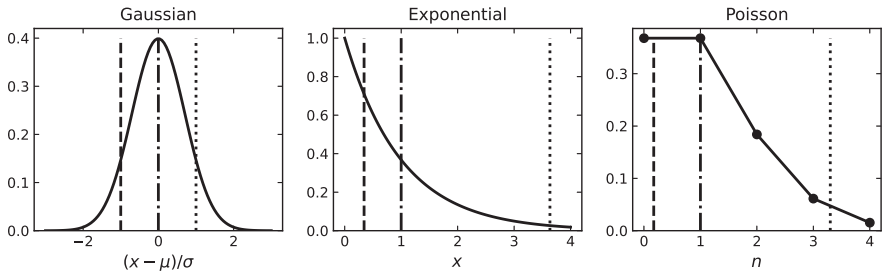


Figure 6.2 Confidence-level intervals for the Gaussian, exponential and Poisson distributions. On each plot the dashed-dotted line is the mean, dashed the lower-confidence band and dotted the upper-confidence band.

Consider an experiment producing data (measurements) where $\mathbf{x} = (x_0, x_1, x_2, \ldots)$ are one or more data values. A hypothesis is a statement about the probability of the data. If the probability $P(\mathbf{x}|H)$ for data $\mathbf{x}$ is regarded as a function of the hypothesis H it is called the likelihood of H, and we can write $L(H) = P(\mathbf{x}|H)$.

If the hypothesis is characterized by one or more parameters $\boldsymbol{\theta}$, $L(\boldsymbol{\theta}) = P(\mathbf{x}|\boldsymbol{\theta})$ is the likelihood function. An estimator $\hat{\theta}$ is a function of the data used to estimate the value of the parameter θ. When forming estimators it is informative to be aware of the following:

1. Consistency is when $\hat{\theta}$ converges to the true value θ as the amount of data increases.

2. The bias of an estimator can be written as $b = E[\hat{\theta}] - \theta$. An unbiased estimator gives $b = 0$. If $\hat{\theta}$ is an unbiased estimator, $\hat{\theta}^2$, for example, is not, in general, an unbiased estimator of θ^2.

3. Efficiency is the ratio of the minimum possible variance for any estimator of θ to the variance $V[\hat{\theta}]$ of the estimator $\hat{\theta}$.

4. Robustness is when an estimator is insensitive to departures from assumptions in the pdf.

6.4.1 Maximum likelihood

The maximum likelihood estimators for $\boldsymbol{\theta}$ are defined as the values that give the maximum of L. These estimators have many of the desired properties previously mentioned.

Mathematically, it is easier to work with $\ln L$. The likelihood equations are

$$\frac{\partial \ln L}{\partial \theta_i} = 0, \quad \text{for} \quad i = 1, \ldots, N\,. \tag{6.18}$$

For the simplest case, these equations can be solved analytically. In other cases, a system of equations can be solved numerically. For nontrivial cases, solving these equations becomes a minimization problem on a computer.

Consider the situation of n statistically independent quantities $\mathbf{x} = (x_1, \ldots, x_n)$ with each component following the same pdf $f(x; \boldsymbol{\theta})$. This case allows the joint pdf of the data sample to factorize:

$$L(\boldsymbol{\theta}) = \prod_{i=1}^{n} f(x_i; \boldsymbol{\theta}) \,, \tag{6.19}$$

where the number of events n is fixed.

If n follows a Poisson distribution of mean μ, we write the extended likelihood as

$$L(\boldsymbol{\theta}) = \frac{\mu^n}{n!} e^{-\mu} \prod_{i=1}^{n} f(x_i; \boldsymbol{\theta}) \,. \tag{6.20}$$

This is the unbinned maximum likelihood. Normalizations depending on $\boldsymbol{\theta}$ must be included but other constants can be dropped. The compute time is $\sim \mathcal{O}(n)$, where n is the number of data measurements.

6.4.2 Binned Likelihood

To reduce the amount of computation, the data is often binned in a histogram. If possible, I recommend not binning the data. Consider the vector of binned data $\mathbf{n} = (n_1, \ldots, n_N)$. The compute time will be $\sim \mathcal{O}(N)$, for N bins. Obviously N will be less than n thus reducing the computational requirements, usually significantly. Define the expectation values $\boldsymbol{\mu} = E[\mathbf{n}]$ and probability $f(\mathbf{n}; \mu)$. If $\mu = \mu(\boldsymbol{\theta})$, we can maximize the likelihood based on the bin contents. In the following, we consider the likelihood for three probability distributions: multinomial, Poisson and normal (Gaussian).

6.4.2.1 Multinomial

If the total number of events is fixed $n_{\text{tot}} = \sum n_i$. The probability is

$$f(\mathbf{n}; \boldsymbol{\theta}) = \frac{n_{\text{tot}}!}{n_1! \ldots n_N!} p_1^{n_1} \ldots p_N^{n_N} \,, \tag{6.21}$$

where $p_i = p_i(\boldsymbol{\theta})$ and $\mu_i = n_{\text{tot}} p_i$. This probability distribution can be used in the same way we use the Poisson and normal distributions below.

6.4.2.2 Poisson

If n_i are independent and Poisson distributed, the probability distributed over N bins is

$$f(\mathbf{n}; \boldsymbol{\theta}) = \prod_{i=1}^{N} \frac{\mu_i^{n_i}}{n_i!} e^{-\mu_i} \,, \tag{6.22}$$

where $\mu_i = \mu_i(\boldsymbol{\theta})$ and $\mu_{\text{tot}} = \sum \mu_i$.

Rather than maximize the likelihood, it is usually to maximize the likelihood ratio

$$\lambda(\boldsymbol{\theta}) = \frac{f(\mathbf{n};\boldsymbol{\theta})}{f(\mathbf{n};\hat{\mu})}\,, \tag{6.23}$$

where $f(\mathbf{n};\mu)$ is a model with adjustable parameters. For each bin $\mu = (\mu_1,\ldots,\mu_N)$, the corresponding estimators are $\hat{\mu} = (n_1,\ldots,n_N)$. The, so-called, saturated model is

$$f(\mathbf{n};\mathbf{n}) = \prod_{i=0}^{N} \frac{n_i^{n_i}}{n_i!} e^{-n_i}\,. \tag{6.24}$$

This ratio allows use to obtain a statistic usable for a test of the goodness of fit (see Sec. 6.5).

For computational and notational reasons, typically one minimizes $-2\ln\lambda(\boldsymbol{\theta})$ rather than maximizes $\lambda(\boldsymbol{\theta})$. The factor of two will become apparent later. Proceeding, we have

$$\begin{aligned}
\ln f(\mathbf{n};\boldsymbol{\theta}) &= \ln \prod_{i=1}^{N} \frac{\mu_i^{n_i}}{n_i!} e^{-\mu_i} = \sum_{i=1}^{N} \left[\ln \frac{\mu_i^{n_i}}{n_i!} e^{-\mu_i}\right] \\
&= \sum_{i=1}^{N} \left[-\mu_i + n_i \ln \mu_i - \ln n_i!\right]\,, \qquad (6.25)\\
\ln f(\mathbf{n};\mathbf{n}) &= \sum_{i=1}^{N} \left[-n_i + n_i \ln n_i - \ln n_i!\right]\,, \qquad (6.26)\\
-2\ln\lambda(\boldsymbol{\theta}) &= 2\sum_{i=1}^{N} \left[\mu_i(\boldsymbol{\theta}) - n_i - n_i \ln \mu_i(\boldsymbol{\theta}) + n_i \ln n_i\right] \\
&= 2\sum_{i=1}^{N} \left[\mu_i(\boldsymbol{\theta}) - n_i + n_i \ln \frac{n_i}{\mu_i(\boldsymbol{\theta})}\right]\,. \qquad (6.27)
\end{aligned}$$

If you are not using a ratio, you can get the same form by using Stirling's approximation using $\ln n! \approx n\ln n - n$. Notice $-2\ln\lambda(\boldsymbol{\theta})$ does not contain any uncertainty estimates. Also, if $n_i = 0$, the last two terms in Eq. (6.27) vanish.

6.4.2.3 Gaussian (least-squares)

If the measurements y_i are assumed Gaussian distributed with mean $\mu(x_i;\boldsymbol{\theta})$ and known σ_i^2,

$$\chi^2(\boldsymbol{\theta}) = -2\ln L(\boldsymbol{\theta}) + \text{constant} = \sum_{i=1}^{N} \frac{[y_i - \mu(x_i;\boldsymbol{\theta})]^2}{\sigma_i^2}\,. \tag{6.28}$$

This expression can be useful even when the data are not Gaussian distributed, as long as the y_i are independent.

If the data are not independent but have a covariance matrix $V_{ij} = \text{cov}[y_i, y_j]$,

$$\chi^2(\boldsymbol{\theta}) = [\mathbf{y} - \mu(\boldsymbol{\theta})]^T V^{-1} [\mathbf{y} - \mu(\boldsymbol{\theta})]\,, \tag{6.29}$$

for measurements $\mathbf{y} = (y_i, \ldots, y_N)$ and predicted values $\mu = (\mu_i(\boldsymbol{\theta}), \ldots, \mu_N(\boldsymbol{\theta}))$.

In Eq. (6.28), one requires the variances σ_i^2 which are often not known a priori and must be estimated from the data. If using $\sigma_i^2 = y_i$, the estimate is poor for small y_i and undefined for $y_i = 0$. Often we set $\sigma_i^2 = 1$ and call it a least-squares fits.

For least-squares, if the model is a straight line parameterized by $\boldsymbol{\theta}$, the minimization can be calculated analytically and is called linear regression. Python library module `scip.la.lstsq` can be useful in this case. On the other hand, linear regression is available on most programmable calculators, and was commonly taught in school as one of the first programmable calculator feature. If the model is a polynomial or linear function of the parameters $\boldsymbol{\theta}$, an analytical calculation is also possible, but the matrix of equation coefficients can be ill-conditioned.

For the general case, numerical minimization is the best alternative. Minimization is iterative and is effectively a multivariant optimization (see Sec. 2.3). Python library module `scipy.optimize.leastsq` can be useful. Weighted least-squares fitting—including σ_i—is called χ^2 fitting.

6.4.3 Variance of the parameter estimators

Given a set of measurements of a random variable and a model for the pdf, we have discussed how to estimate its parameters. Now we will discuss how to assign statistical uncertainties to these estimated parameters. Ideally we would like to determine the estimates of the parameters and variance analytically but here we will assume the problem at hand is not so trivial and that numerical methods are required.

If the model has a single parameter, it is a special case of the general discussion that follows. Non-trivial models will depend on more than one parameter, and there can be correlations between them. Correlation is a measure that determines the degree to which two or more random variables move due to uncertainty. It is given by

$$\rho_{x,y} = \text{correlation}(x, y) = \frac{\text{covariance}(x, y)}{\sigma(x)\sigma(y)}\,. \tag{6.30}$$

The covariance was first mentioned in Eq. (6.2). The diagonal elements will be one. Covariance is a systematic relationship between two variables in which a change in one reflects a change in the other. Positive values indicate a direct relationship and negative values an inverse relationship. The covariance is great for defining the type of relationship between variables but is terrible in interpreting the magnitude which is better done with the correlation matrix.

Let's first discuss the variance for the least-squares method. The covariance matrix U^{-1} for the parameters can be written as [24]:

$$(U^{-1})_{ij} = \frac{1}{2} \left. \frac{\partial \chi^2}{\partial \theta_i \partial \theta_j} \right|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}} . \tag{6.31}$$

If the model is linear in the parameters, than χ^2 will be quadratic in the parameters and we can write

$$\chi^2(\boldsymbol{\theta}) = \chi^2(\hat{\boldsymbol{\theta}}) + \frac{1}{2} \sum_{i,j=1}^{m} \left. \frac{\partial^2 \chi^2}{\partial \theta_i \partial \theta_j} \right|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}} (\theta_i - \hat{\theta}_i)(\theta_j - \hat{\theta}_j) . \tag{6.32}$$

Combining Eq. (6.32) and Eq. (6.31) yields

$$\chi^2(\boldsymbol{\theta}) = \chi^2(\hat{\boldsymbol{\theta}}) + 1 = \chi^2_{\text{min}} + 1 . \tag{6.33}$$

This equation give the one standard deviation departure from the least-squares estimates. The contours in parameter space has tangents $\hat{\theta}_i \pm \hat{\sigma}_i$. The n standard deviation departure from the estimator is given by replacing 1 by n in Eq. (6.33).

Figure 6.3 shows a representative case for two parameters. The values of $\chi^2_{\text{min}} + 1$ are plotted versus the two parameters θ_0 and θ_1. Also shown

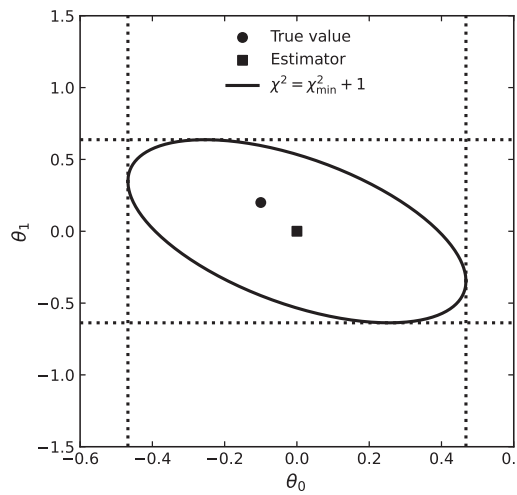


Figure 6.3 Log-likelihood function contour versus two parameters θ_0, θ_1. Shown is the true value (circle), best estimator (square) and the one standard-deviation uncertainties contour. The contour has been shifted so the best estimator is at the origin. The horizontal and vertical lines are tangents to the contour.

are the estimators $(\hat{\theta}_0, \hat{\theta}_1)$ (square) and the unknown true values (circle). The contour has been centred on the estimators for convenience of illustration. For a model with linear parameters, the contour is an ellipse with its major axis rotated with respect to the parameter axes. The amount of rotation is related to the covariance between the two parameters. For the case drawn here, the parameters are anticorrelated. Uncorrelated parameter uncertainties would cause the major axis of the ellipse to be along one of the two parameter axis. The contour would only be circular if the uncertainties on the two parameters were identical. The dotted horizontal and vertical lines represent the tangents to the contour and determine the uncertainties $\hat{\theta}_0 \pm \hat{\sigma}_0$ and $\hat{\theta}_1 \pm \hat{\sigma}_1$. For least-squares fitting, the library modules typically return estimates for the model parameters and a covariance matrix.

If the model is not linear in the parameters, then the contour is not in general elliptical, and we can no longer obtain the standard deviations from the tangents [24, 25]. However, it defines a region in parameter space which can be interpreted as a confidence region, the size of which reflects the statistical uncertainty of the fit parameters.

Figure 6.4 shows the likelihood function versus a single parameters. The one standard deviation up and down uncertainties on the parameter are indicated on the figure. In general, the likelihood is not symmetric and the up and down uncertainties are different. In the non-ideal situation, one must perform a search in the higher dimensional parameter space, as you will encounter in Problem 6.7.4.

Now we come to the variance of the maximum likelihood method. First consider the case of a single parameter. The Taylor series expansion about the

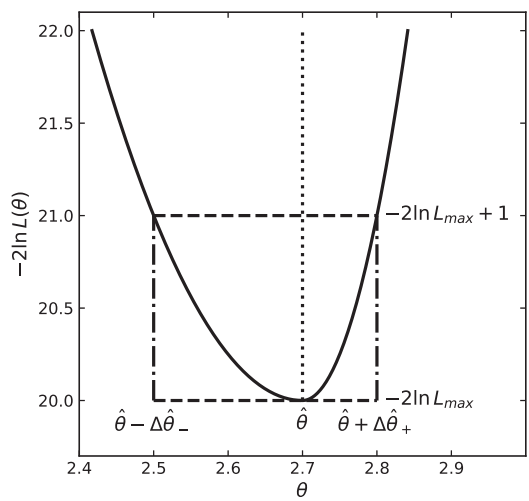


Figure 6.4 Scaled log-likelihood function versus a single parameter. Shown is the best estimator and the one standard-deviation uncertainties.

maximum-likelihood estimator $\hat{\theta}$ is

$$\ln L(\theta) = \ln L(\hat{\theta}) + \left.\frac{\partial \ln L}{\partial \theta}\right|_{\theta=\hat{\theta}} (\theta - \hat{\theta}) + \frac{1}{2!} \left.\frac{\partial^2 \ln L}{\partial \theta^2}\right|_{\theta=\hat{\theta}} (\theta - \hat{\theta})^2 + \ldots . \quad (6.34)$$

Since the likelihood is a maximum at $\hat{\theta}$, the derivative in the second term of Eq. (6.34) evaluate at $\theta = \hat{\theta}$ vanishes. Ignoring higher-order terms and using $\ln L(\hat{\theta}) = \ln L_{\max}$, we have

$$\ln L(\theta) = \ln L_{\max} + \frac{(\theta - \hat{\theta})^2}{2\hat{\sigma^2}_{\hat{\theta}}} \quad (6.35)$$

where we have used

$$\widehat{V^{-1}}_{ij} = - \left.\frac{\partial^2 \ln L}{\partial \theta_i \theta_j}\right|_{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}} \quad \Rightarrow \quad \widehat{\sigma^2}_{\hat{\theta}} = \left.\frac{-1}{\frac{\partial^2 \ln L}{\partial \theta^2}}\right|_{\theta=\hat{\theta}} . \quad (6.36)$$

For $\theta = \hat{\theta} \pm \hat{\sigma}_{\hat{\theta}}$,

$$\ln L(\hat{\theta} \pm \hat{\theta}_{\hat{\theta}}) = \ln L_{\max} - \frac{1}{2} . \quad (6.37)$$

The method can be generalized to more than one parameter. We now see the point of our earlier scaling by -2 of the log-likelihood in Eq. (6.27). The maximum becomes a minimum, and the offset of $-1/2$ in Eq. (6.37) becomes $+1$ to look similar to the least-squares case Eq. (6.33).

In case the contour is too difficult to solve numerically, the distribution of estimates can approximate using the Hessian or investigated using a MC method. Typically the estimator is found by minimization. Estimating uncertainty on parameters is not straightforward and depends on the specific method used. However, there are approaches to estimate the uncertainty based on the Hessian matrix. If the Hessian is available from the minimization method, it can be used to estimate the covariance matrix of the parameters, and thus, their uncertainties. The inverse of the Hessian matrix at the solution point approximates the covariance matrix. The square-root of the diagonal elements of the covariance matrix give the standard errors (uncertainties) in the parameters. If the Hessian is not obtainable from the minimization method, numerical differentiation techniques can be used to approximate the Hessian after the optimization is complete. For the least-squares method, the covariance matrix can be estimated directly from the residuals and the Jacobian matrix of the model. Using Python library modules like `scipy.optimize.curve_fit` the uncertainties are determined by this method and provided to the user.

For the MC method, one must generate a large number of pseudo-experiments computing the estimates each time and look at how the values are distributed. Since we don't know the true parameters, the estimated values from the real experimental measurements can be used for the true parameters in the MC program. We can compute the sampling variance for the estimates

obtained from the MC pseudo-experiments and give this as the statistical error on the parameters estimated from the real measurement. Obviously this method requires a significant amount of work.

6.4.4 Practical consideration

I suggest you at least consider the following when performing χ^2-fitting.

1. Perform local fits unless there is a good reason to require a global fit. Sometime the data covers a large range and the model being fit has no particular reason to cover that large of a range. While it may be aesthetically appealing to have a single curve drawn through the entire data plot, it might not be physically or statistically justified. When only local information is required, just fit that local region. You might even consider fitting different models to different local regions as separate fits.

2. If the bin widths are not constant, the model might not be a good fit to data. Consider weighting the data by the bin width. Also weight the errors the same way; do not recalculated the errors.

3. Considering how the χ^2 is calculated, sometimes it's advantageous to use bin widths that give the same number of events in each bin—although it's not visually appealing.

4. If some bins have no events, a log-likelihood fit is often better.

5. For least-squares fitting, you should determine the area under the fitted curve directly using the number of entries in histogram.

Which x-value in the bin should be used when fitting? Fitting a function to steeply falling (or raising) data using bin centres can have problems. Some improvements you might try are:

1. Weight the x-value to be closer to higher edge.

2. Integrate the fit function over the bin width then divide by the bin width and fit this integrated function to the data [26].

Binning is more of an art than a science. Some considerations are:

1. Do not use too many bins. Using too many bins emphasizes fluctuations in the data. Certainly do not approach the unbinned limit or there is no point in binning the data.

2. Make sure to use enough bins. Significant information might be lost if too few bins are used. The histogram will be too smooth.

3. As a rule of thumb, the number of bins b can be estimated using $b = 1 + \log_2 N = 1 + \log_{10} N / \log_{10} 2 \approx 1 + 3.3 \log_{10} N$.

4. Take the histogram range into consideration. The number of bins should be a nice integer number. From a presentation/communication point of view, this is important.

Now a few words about the choosing a model to fit the data. If possible, it is more ecstatically pleasing if the model used to fit the data is based on some physical principles. Nevertheless, there are times when this is not possible and we must choose a purely empirical model description. If the model is too simple (not enough parameters) we will overfit the data. If the model is too complex (too many parameters) we will fit the fluctuations (noise) in the data. This situation is shown in Fig. 6.5. The fit to a straight line is probably under fitting as I believe I see a nonlinear trend in the data; and a straight line does not approximate well the true polynomial, which we just happen to know in this example. The fit to a 15th order polynomial over fits the data, as we see the wiggles in the polynomial have nothing to do with the true polynomial. The best fit will be between 1st and 15th order, namely 2nd order.

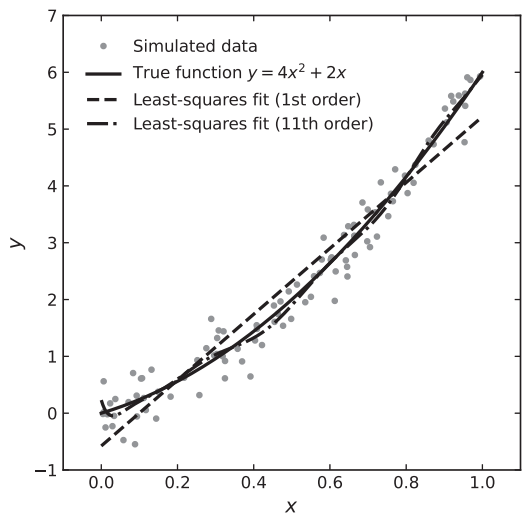


Figure 6.5 Under fitting and over fitting of data.

6.4.5 Example: Plotting, error bars and fitting

This example uses binned data with a range in bin entries spanning over six orders of magnitude. Thus the example is nontrivial. We first read in the pairs of numbers from the csv file `dijet.csv`. These numbers represent a histogram of invariant mass values of a dijet system (the meaning of the numbers is not important, think of it as your favourite thing to measure).[1] The first column

[1] Derived from https://doi.org/10.17182/hepdata.91126.

of numbers are the lower bin edges and the second column the number of entries in each bin. Notice that the bin widths are not constant. The following code reads in the data and determines the number of bins of data in the file.

```
"""Plotting and fitting of dijet data."""

import numpy as np
from scipy.stats import chi2
from scipy.optimize import curve_fit
from scipy.special import gammaincc
from scipy.special import erfinv
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')

debug = False

# Read in the histogram from the csv file.
data = np.genfromtxt("dijet.csv",delimiter=',',dtype=int)
data = data.flatten('F')
edges, counts = np.split(data,2)

if (debug):
    print (edges,'\n')
    print (counts,'\n')
print (len(counts),"data values found")
```

```
121 data values found
```

The following code determines the lower bin edge of the first and the upper bin edge of the last bins with nonzero entries. We are not going to consider the region below and above the region with entries, assuming they contain no information by having zero entries. Also, to avoid some turn-on effects we are only going to consider x-values above 1200. The code also determines the bin at which this happens.

```
# Determine bin number of non-zero bins.
a = np.where(counts!=0)[0]
imin = a[0]
imax = a[-1]
print ("lower non-zero bin number",imin,
       "upper non-zero bin number",imax)

# Determine bin edges of non-zero bins.
min = edges[imin]
max = edges[imax+1]
print ("lower non-zero bin edge",min,
       "upper non-zero bin edge",max)

# Determine index of threshold bin.
istart = np.argmax(edges>=1200)
print("bin",istart,"is >= 1200 GeV")
```

```
lower nonzero bin number 7 upper nonzero bin number 95
lower nonzero bin edge 1166 upper nonzero bin edge 8208
bin 8 is >= 1200 GeV
```

The bin edges in the input data file are in units of GeV. We want to work in TeV. The following code convert them.

```
# Convert from GeV to TeV.
GeV2TeV = 1.0e-3
nedges = edges * GeV2TeV
```

The following code plots the binned data using the bin centres as the x-values. For the y-values the number of entries are used. For the y-value error bars the square root of the y-values are used. The data is plotted over the range [1.2,8.3] TeV.

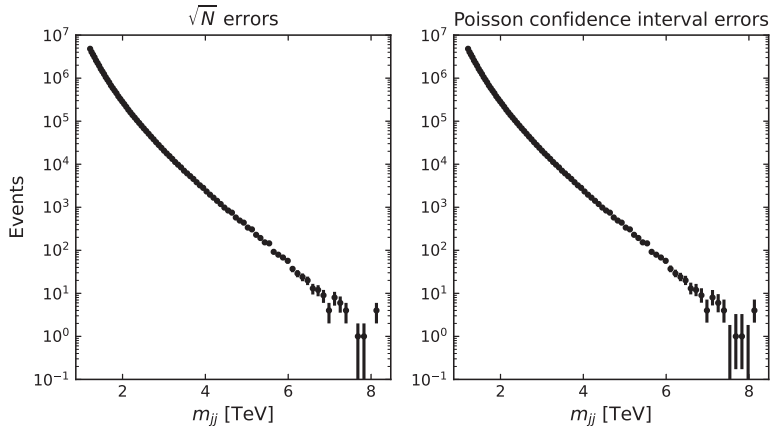
I have used a small plot marker type and simple symmetric error bars. The x-axis label has been set to (r'm_jj [TeV]'). What would be an appropriate y-axis label? Since the number of entries spans many orders of magnitude on the y-axis, I use a log-y scale.

If the minimum y-axis was set to 10^{-6} what would be the usefulness of showing data with one entry and an error bar going down to 10^{-6} versus data with zero entries showing no error bars at all? What would be a reasonable y-axis minimum showing the most useful information, and why?

The code adds a second plot using Poisson 68.3% confidence level error bars.

```
debug = False

# Truncate histogram bin range.
tedges = nedges[istart:imax+2]
ydata  = counts[istart:imax+1]
if debug:
    print(tedges)
    print(ydata)

# Calculate bin centres and widths.
xdata = (tedges[:-1] + tedges[1:]) / 2
bw    =  tedges[1:]  - tedges[:-1]
if debug:
    print(xdata)
    print (bw)

def CL(n,alpha,beta):
    # Return the Poisson confidence interval errors.
    # np.where statement used to assign lower bound of 0
    # for case of n = 0.
    errlo = np.where(n != 0,n -0.5 * chi2.ppf(alpha,2*n),0)
    errup = 0.5 * chi2.ppf(1-beta,2*(n+1)) - n
    return (errlo,errup)

# Calculate square-root errors.
err = np.sqrt(ydata)

# Calculate Poisson confidence interval errors.
```

```
alpha = 0.158655
beta  = 0.158655
errmp = CL(ydata,alpha,beta)

# Plot the binned data with error bars.
fig, ax = plt.subplots(1,2,figsize=(8,4))
#
ax[0].errorbar(xdata,ydata,yerr=err,fmt='.',c='k')
ax[0].set_title(r"$\sqrt{N}$ errors")
ax[0].set_xlabel(r"$m_{jj}$ [TeV]")
ax[0].set_ylabel("Events")
ax[0].set_ylim([1e-6,1e7])
ax[0].set_ylim([1e-1,1e7])
ax[0].set_yscale('log')
#
ax[1].errorbar(xdata,ydata,yerr=errmp,fmt='.',c='k')
ax[1].set_title("Poisson confidence interval errors")
ax[1].set_xlabel(r"$m_{jj}$ [TeV]")
ax[1].set_ylim([1e-6,1e7])
ax[1].set_ylim([1e-1,1e7])
ax[1].set_yscale('log')
#
plt.savefig('6_4_dijet_1.pdf')
plt.show()
```



Since the number of entries in a bin is an integer there is little information to be had by extending the y-axis to much lower than one; I suggest one order of magnitude lower, 0.1. With this choice for the minimum, all the data values will be on the plot. If the errors are calculated using $\sqrt{n}$, entries of one will have their lower error bars clipped. But this will happened regardless of what the lower value of the y-axis is on a log-scale. Entries of zero simply do not appear. For Poisson confidence interval errors, the full error bars will be visible except for bins with zero entries for which only part of the upper error bar will be visible on a log-scale; this is the important information. Notice that the asymmetry in confidence interval errors is sometime difficult to see on a log-scale.

We now turn to fitting the data. Define the following models (fit functions):

$$\begin{aligned} f(x;p_0,p_1,p_2) &= p_0(1-x)^{p_1}x^{p_2} \\ f(x;p_0,p_1,p_2,p_3) &= p_0(1-x)^{p_1}x^{p_2+p_3\ln x} \\ f(x;p_0,p_1,p_2,p_3,p_4) &= p_0(1-x)^{p_1}x^{p_2+p_3\ln x+p_4(\ln x)^2}\,, \end{aligned} \tag{6.38}$$

where $x = m_{jj}/\sqrt{s}$ and $\sqrt{s} = 13\,\text{TeV}$. We will use these semiempirical models to fit the data. They are referred to as the 3-, 4-, 5-parameter models. They are implemented in the following code.

```
# Model functions.
def bg3(m,p0,p1,p2):
    sqrts = 13.0
    x = m / sqrts
    return p0 * np.power(1-x,p1) * np.power(x,p2)
def bg4(m,p0,p1,p2,p3):
    sqrts = 13.0
    x = m / sqrts
    return p0 * np.power(1-x,p1) * np.power(x,p2+p3*np.log(x))
def bg5(m,p0,p1,p2,p3,p4):
    sqrts = 13.0
    x = m / sqrts
    return p0 * np.power(1-x,p1) \
           * np.power(x,p2+p3*np.log(x)+p4*np.log(x)*np.log(x))
```

The following code fits one of the models to the data using the $\sqrt{n}$-type errors. The resulting fit function is plotted on top of the data. The fit parameters and their uncertainties are printed. They can be checked by printing the entire covariance matrix as an option. If you notice anything strange about the covariance matrix, it might be due to the data with zero errors. Try setting these errors to one.

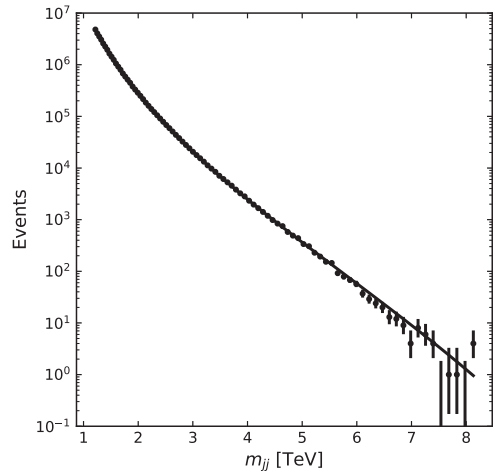
```
# Weight by bin width (change histogram to function).
weight = False
if weight:
    ydatap = ydata * bw
    errp   = err   * bw
    errmpp = errmp * bw
else:
    ydatap = ydata
    errp   = err
    errmpp = errmp

# Fudge the error bars.
for i, e in enumerate(errp):
    if (e == 0): errp[i] = 1.0 #Avoid 0 error bars.

# Fit the data.
mode = 3
if mode == 3:
    guess = [1.50e2,7.38e0,-4.68e0]
    parm, cov = curve_fit(bg3,xdata,ydatap,guess,errp)
if mode == 4:
    guess = [1.66e2,7.50e0,-4.62e0,1.24e-2]
    parm, cov = curve_fit(bg4,xdata,ydatap,guess,errp)
```

```
if mode == 5:
    guess = [1.40e5,1.26e1,1.86e0,2.39e0,3.16e-1]
    parm, cov = curve_fit(bg5,xdata,ydatap,guess,errp,
                          maxfev=100000)

# Superimpose the fit on the data.
fig, ax = plt.subplots(1,figsize=(5,5))
plt.errorbar(xdata,ydatap,yerr=errmpp,marker='.',ls='None')
if mode == 3: plt.plot(xdata,bg3(xdata,*parm))
if mode == 4: plt.plot(xdata,bg4(xdata,*parm))
if mode == 5: plt.plot(xdata,bg5(xdata,*parm))

# Make pretty plot.
plt.xlabel(r"$m_{jj}$ [TeV]")
plt.ylabel("Events")
plt.ylim([1e-1,1e7])
plt.yscale('log')
plt.savefig('6_4_dijet_2.pdf')
plt.show()

# Print the fit results.
perr = np.sqrt(np.diag(cov))
np.set_printoptions(linewidth=160,
                    formatter={'float': '{:.2e}'.format})
debug = True
if debug:
    print ("covarance matrix:")
    for i in range(mode):
        print (cov[i])
    stds = np.array([[perr[i]*perr[j] for j in range(mode)]
                     for i in range(mode)])
    print ("correlation matrix:")
    corr = cov / stds
    for i in range(mode):
        print (corr[i])
print("fit parameters and uncertainties")
print("p0 =",'{0: 1.2e}'.format(parm[0]),"+-",'{0:1.2e}' \
      .format(perr[0]))
print("p1 =",'{0: 1.2e}'.format(parm[1]),"+-",'{0:1.2e}' \
      .format(perr[1]))
print("p2 =",'{0: 1.2e}'.format(parm[2]),"+-",'{0:1.2e}' \
      .format(perr[2]))
if (mode > 3): print("p3 =",'{0: 1.2e}'.format(parm[3]),"+-",
                     '{0:1.2e}'.format(perr[3]))
if (mode > 4): print("p4 =",'{0: 1.2e}'.format(parm[4]),"+-",
                     '{0:1.2e}'.format(perr[4]))
```

```
covarance matrix:
[1.26e+02 1.80e+00 2.84e-01]
[1.80e+00 2.63e-02 4.02e-03]
[2.84e-01 4.02e-03 6.43e-04]
correlation matrix:
[1.00e+00 9.88e-01 9.99e-01]
[9.88e-01 1.00e+00 9.79e-01]
[9.99e-01 9.79e-01 1.00e+00]
```

```
fit parameters and uncertainties
p0 =  1.50e+02 +- 1.12e+01
p1 =  7.38e+00 +- 1.62e-01
p2 = -4.68e+00 +- 2.54e-02
```

Notice in the plot with the fit that the error bars have been drawn using confidence intervals, but the fit algorithm uses $\sqrt{n}$-type errors; the least-squares algorithm `scipy.optimize.curve_fit` requires symmetric error bars. In these types of global fits, the fit algorithm will optimize the fit at low mass where the number of entries is the highest, and thus, relative uncertainty the lowest. At high masses, the fit goes through the points on average with no change in curvature; the curvature has already been determined by the bins with a large number of entries. If one is most interested in the low-statistics tail of the distribution, one should fit to just the high-mass region.



Parameter	Value
p_0	$(1.50 \pm 0.1124) \times 10^2$
p_1	$(7.38 \pm 0.1624) \times 10^0$
p_2	$(-4.68 \pm 0.0254) \times 10^0$
χ^2/dof	$2327/85 = 27.4$
p_0	$(1.66 \pm 1.15) \times 10^2$
p_1	$(7.50 \pm 0.765) \times 10^0$
p_2	$(-4.62 \pm 0.456) \times 10^0$
p_3	$(1.24 \pm 8.30) \times 10^{-2}$
χ^2/dof	$2329/84 = 27.7$
p_0	$(1.39 \pm 6.31) \times 10^5$
p_1	$(1.26 \pm 0.349) \times 10^1$
p_2	$(1.86 \pm 4.31) \times 10^0$
p_3	$(2.38 \pm 1.56) \times 10^0$
p_4	$(3.16 \pm 2.06) \times 10^{-1}$
χ^2/dof	$2294/83 = 27.6$

The table presents the result of fitting the data to the three different models and requires three separate runs of the code with different values for `mode` which represents the number of parameters in the model.

We will now calculate the χ^2, degrees of freedom (dof), and χ^2-per-dof of the fits. This topic is not discussed until Sec. 6.5. You can either try to get the gist of it now, or return to it later.

```
if mode == 3: f = bg3(xdata,*parm)
if mode == 4: f = bg4(xdata,*parm)
if mode == 5: f = bg5(xdata,*parm)
chi2 = 0
for data,model,errpp in zip(ydatap,f,errp):
    if weight:
        chi2 += (data-model)*(data-model)/(errpp*errpp)
    else:
        chi2 += (data-model)*(data-model)/model
print("chi-squared",np.round(chi2,1))
print("degrees of freedom",len(f),"-",mode,"=",len(f)-mode)
print("chi-squared/dof",np.round(chi2/(len(f)-mode),1))
```

```
chi-squared 2327.1
degrees of freedom 88 - 3 = 85
chi-squared/dof 27.4
```

We examine the quality of the fit by making the following goodness-of-fit plots in the code below.

1. Plot of data minus the model versus mass (absolute difference—residuals).
2. Plot of data minus the model all divided by the model versus mass (relative difference).
3. Plot of the ratio of the data to the model versus mass.
4. Plot of data minus the model all divided by the square-root of the model versus mass (pull).
5. Plot of p-value versus mass (see Sec. 6.6).
6. Plot of significance versus mass (see Sec. 6.6).

```
fig, ax = plt.subplots(3,2,figsize=(6,9))

# Absolute difference (residuals).
ax[0,0].plot(xdata,ydatap-f,marker='.',ls='None',c='k')
ax[0,0].set_title("y-f")

# Relative difference.
ax[0,1].plot(xdata,(ydatap-f)/f,marker='.',ls='None',c='k')
ax[0,1].set_title("(y-f)/f")
```

```
# Ratio.
ax[1,0].plot(xdata,ydatap/f,marker='.',ls='None',c='k')
ax[1,0].set_title("y/f")

# Pulls.
ax[1,1].plot(xdata,(ydatap-f)/errp,marker='.',ls='None',c='k')
ax[1,1].set_title(r"(y-f)/$\sqrt{f}$")

# p-values and significances.
z = []
p = []
xsig = []
for d,b,x in zip(ydatap,f,xdata):
    if d > b:
        pval = 1 - gammaincc(d,b)
        if pval < 2.87e-7: pval = 2.87e-7
        p.append(pval)
        if pval < 0.5:
            Z = np.sqrt(2)*erfinv(1-2*pval)
            z.append(Z)
            xsig.append(x)
    else:
        pval = gammaincc(d+1,b)
        if pval < 2.87e-7: pval = 2.87e-7
        p.append(pval)
        if pval < 0.5:
            Z = np.sqrt(2)*erfinv(1-2*pval)
            z.append(-Z)
            xsig.append(x)

ax[2,0].plot(xdata,p,marker='.',ls='None',c='k')
ax[2,0].set_title("p-value",fontsize=16)
ax[2,0].set_yscale('log')

ax[2,1].plot(xsig,z,marker='.',ls='None',c='k')
ax[2,1].set_title("significance")

plt.tight_layout()
plt.savefig('6_4_dijet_3.pdf',bbox_inches='tight')
plt.show()
```

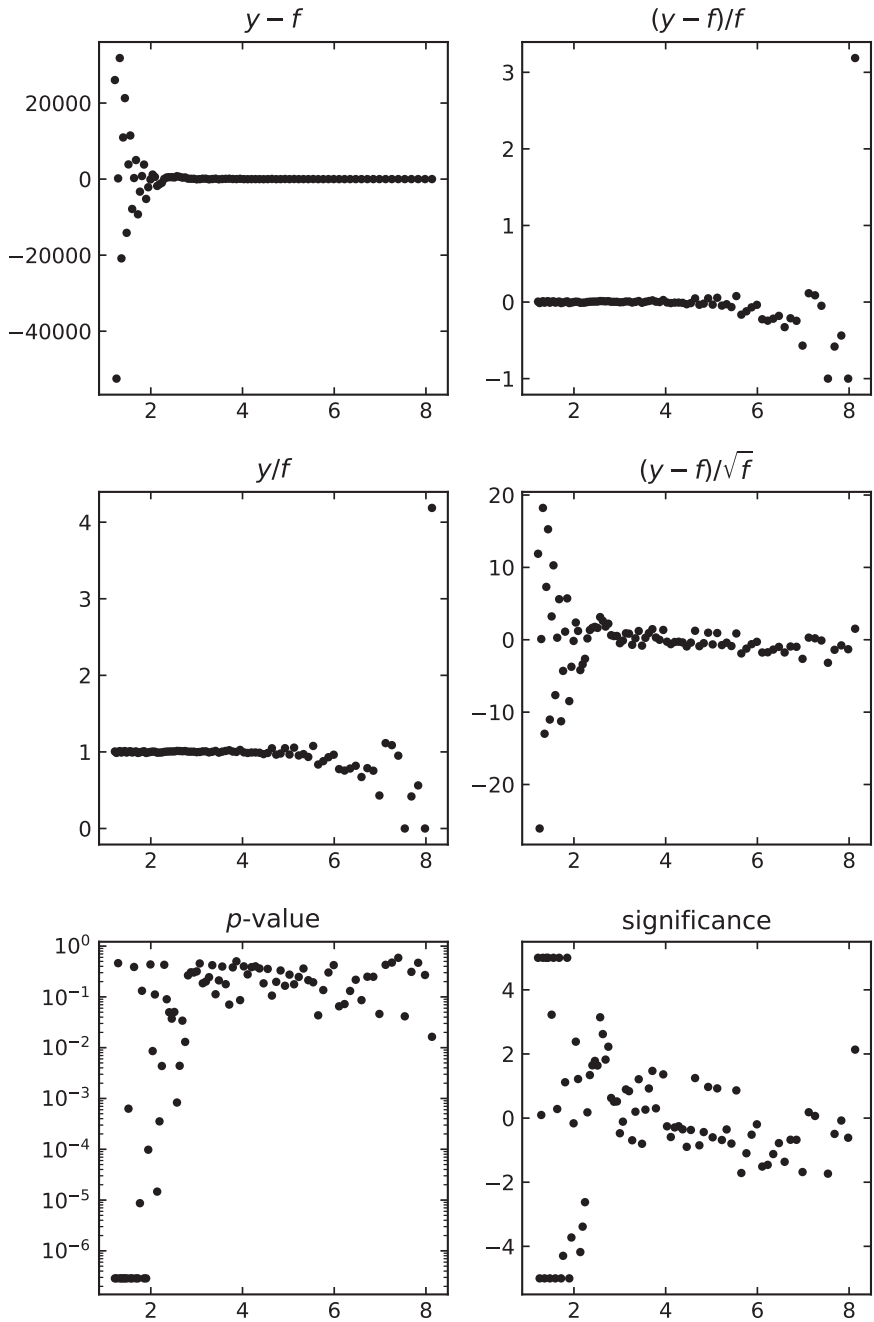
The p-value and significance are calculated as follows. Consider the Poisson model of n observed events in data and $\hat{\mu}$ expended events according to the model:

$$P(n,\hat{\mu}) = \frac{\hat{\mu}^n}{n!} e^{-\hat{\mu}} \,. \tag{6.39}$$

The p-value is

$$p = \begin{cases} 1 - Q(n,\hat{\mu}) & \text{for} \quad n > \hat{\mu} \\ Q(n+1,\hat{\mu}) & \text{for} \quad n \leq \hat{\mu} \,, \end{cases} \tag{6.40}$$

where $Q(s,x) = 1 = P(s,x)$ is the upper regularized (normalized) incomplete gamma function (upper integral) and $P(s,x)$ is the cumulative distribution function for Gamma random variables with shape parameter s and scale parameter one.


$y - f$
$(y - f)/f$
y/f
$(y - f)/\sqrt{f}$
p-value
significance

The statistical significance is given by the Z-value

$$p = \int_Z^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx\,. \tag{6.41}$$

A Z-value ≥ 0 corresponds to a p-value ≤ 0.5, and negative Z-values correspond to p-values greater than 0.5. Significant deviations are characterized by quite small p-values corresponding to Z-values ≥ 3. For example, $p = 2.87 \times 10^{-7}$ corresponds to $Z = 5$, which is called a 5σ effect.

6.4.6 Example: Log-likelihood fits

We now repeat the later part of the previous χ^2-fitting example but this time using the log-likelihood function. Estimation of the variance will be considerably more involved than the χ-squared method, which is returned to you by `scipy.optimize.curve_fit`.

```
"""Fitting of dijet data using log-likelihood."""

import numpy as np
from scipy.optimize import minimize, root
import numdifftools as nd
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')
np.set_printoptions(formatter={'float': '{: .2e}'.format})

# Model functions.
def bg3(m,p0,p1,p2):
    sqrts = 13
    x = m / sqrts
    return p0 * np.power(1-x,p1) * np.power(x,p2)
def bg4(m,p0,p1,p2,p3):
    sqrts = 13
    x = m / sqrts
    return p0 * np.power(1-x,p1) * np.power(x,p2+p3*np.log(x))
def bg5(m,p0,p1,p2,p3,p4):
    sqrts = 13
    x = m / sqrts
    return p0 * np.power(1-x,p1) \
           * np.power(x,p2+p3*np.log(x)+p4*np.log(x)*np.log(x))

# log-likelihood ratio function for minimization.
def llbg(parms):
    if mode == 3: yPred = bg3(xdata,*parms)
    if mode == 4: yPred = bg4(xdata,*parms)
    if mode == 5: yPred = bg5(xdata,*parms)
    LL = 0
    for i in range(0,len(xdata)):
        if ydatap[i] == 0:
            LL += yPred[i]
        else:
            LL += yPred[i] - ydatap[i] \
                  + ydatap[i]*np.log(ydatap[i]/yPred[i])
```

```
    return 2*LL

# log-likelihood ratio function for variance.
def llbg2(p0,p1,p2,Lmin,n):
    parms = [p0,p1,p2]
    if mode == 3: yPred = bg3(xdata,*parms)
    if mode == 4: yPred = bg4(xdata,*parms)
    if mode == 5: yPred = bg5(xdata,*parms)
    LL = 0
    for i in range(0,len(xdata)):
        if ydatap[i] == 0:
            LL += yPred[i]
        else:
            LL += yPred[i] - ydatap[i] \
                  + ydatap[i]*np.log(ydatap[i]/yPred[i])
    return 2*LL - Lmin - n

# Find confidence-level contour.
def variance(p,Lmin):
    n = 1 # First contour.
    Nl = 100+1
    xc = []
    yc = []
    zc = []
    xh = 1.025*p[0]; xl = 0.975*p[0]
    yh = 1.005*p[1]; yl = 0.995*p[1]
    zh = 1.002*p[2]; zl = 0.998*p[2]
    X = np.linspace(xl,xh,Nl)
    Y = np.linspace(yl,yh,Nl)
    Z = np.linspace(zl,zh,Nl)
    for y in Y:
        for z in Z:
          sol = root(llbg2,p[0],args=(y,z,Lmin,n))
          if sol.success:
              if sol.fun < 1e-6:
                  xc.append(sol.x)
                  yc.append(y)
                  zc.append(z)
    xc = np.array(xc)
    yc = np.array(yc)
    zc = np.array(zc)
    return [xc,yc,zc]

# Read in the histogram from the csv file.
data = np.genfromtxt("dijet.csv",delimiter=',',dtype=int)
data = data.flatten('F')
edges, counts = np.split(data,2)
a = [i for i, e in enumerate(counts) if e != 0]
imin = a[0]
imax = a[-1]
min = edges[imin]
max = edges[imax+1]
for i, e in enumerate(edges):
    if e >= 1200:
        istart = i
        break
# Convert from GeV to TeV.
```

```
GeV2TeV = 1.0e-3
nedges = [i*GeV2TeV for i in edges]
# Truncate histogram bin range.
tedges = np.array(nedges[istart:imax+2])
ydata  = np.array(counts[istart:imax+1])
# Calculate bin centres and widths.
xdata = (tedges[:-1] + tedges[1:]) / 2
bw    =  tedges[1:]  - tedges[:-1]
# Weight by bin width (change histogram to function).
weight = False
if weight:
    ydatap = ydata * bw
else:
    ydatap = ydata

VERB = True
mode = 3

# Guess initial parameter values.
if mode == 3: initParms = [1.50e2,7.38e0,-4.68e0]
if mode == 4: initParms = [1.64e2,7.48e0,-4.62e0,1.17e-2]
if mode == 5: initParms = [1.40e5,1.26e1,1.84e0,2.39e0,3.16e-1]

# Minimize log likelihood.
res = minimize(llbg,initParms,method='Nelder-Mead',
               options={'maxfev':1000})
if not res.success:
    print("Minimization results:",res)

# Estimate parameter variance using Hessian.
hessian_matrix = nd.Hessian(llbg)(res.x)
covariance_matrix = np.linalg.inv(hessian_matrix)
perr = np.sqrt(np.diag(covariance_matrix))
if VERB:
    print("Optimize estimate:",res.x)
    print("Standard errors:  ",perr)
    print("Covariance matrix:")
    print(covariance_matrix)

# Estimate parameter variance using likelihood search.
if mode == 3:
    con = variance(res.x,res.fun)
    perrp = np.empty(mode)
    perrm = np.empty(mode)
    for i in range(mode):
      perrp[i] = np.max(con[i]) - res.x[i]
      perrm[i] = res.x[i] - np.min(con[i])

# Print fit results.
print()
print("Hessian: fit parameters and uncertainties.")
for i in range(mode):
    print(f"p{i} = {res.x[i]: .2e} +-{perr[i]: .2e}")
if mode == 3:
    print()
    print("Search: fit parameters and uncertainties.")
    for i in range(mode):
```

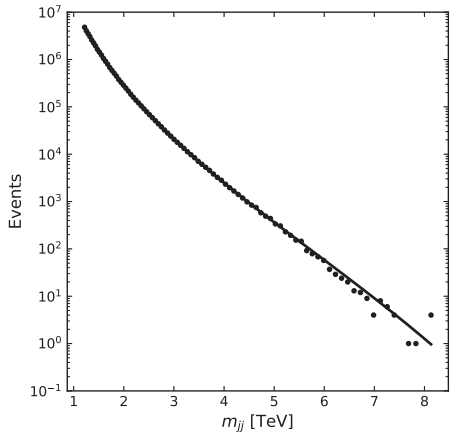
```
        print(f"p{i} = {res.x[i]: .2e} "
              f"+{perrp[i]: .2e} -{perrm[i]: .2e}")

# Calculate chi-squared information.
if mode == 3: f = bg3(xdata,*res.x)
if mode == 4: f = bg4(xdata,*res.x)
if mode == 5: f = bg5(xdata,*res.x)
chi2 = 0
for data,model,w in zip(ydatap,f,bw):
    if weight:
        chi2 += (data-model)*(data-model)/(w*model)
    else:
        chi2 += (data-model)*(data-model)/model
print()
print("chi-squared:",np.round(chi2,2))
print("degrees-of-freedom:",len(f),"-",mode,"=",len(f)-mode)
print("chi-squared/dof:",np.round(chi2/(len(f)-mode),1))
print()

# Calculate function values using fit parameters.
if mode == 3: yOut = bg3(xdata,*res.x)
if mode == 4: yOut = bg4(xdata,*res.x)
if mode == 5: yOut = bg5(xdata,*res.x)

# Superimpose the fit on the data.
fig, ax = plt.subplots()
plt.scatter(xdata,ydatap,marker='.')
plt.plot(xdata,yOut)
plt.xlabel(r"$m_{jj}$ [TeV]")
plt.ylabel("Events")
plt.ylim([1e-1,1e7])
plt.yscale('log')
plt.savefig('6_4_hessian.pdf')
plt.show()

# Plot confidence-level intervals.
if mode == 3:
  xh = 1.025*res.x[0]; xl = 0.975*res.x[0]
  yh = 1.005*res.x[1]; yl = 0.995*res.x[1]
  zh = 1.002*res.x[2]; zl = 0.998*res.x[2]
  fig = plt.figure(figsize=(12,4))

  plt.subplot(1,3,1)
  plt.scatter(con[0],con[1],marker='.')
  plt.xlabel(r"$p_0$")
  plt.ylabel(r"$p_1$")
  plt.xlim(xl,xh)
  plt.ylim(yl,yh)
  plt.axvline(x=res.x[0],ls='dashed')
  plt.axhline(y=res.x[1],ls='dashed')
  plt.axvline(x=res.x[0]-perr[0],ls='dotted')
  plt.axvline(x=res.x[0]+perr[0],ls='dotted')
  plt.axhline(y=res.x[1]-perr[1],ls='dotted')
  plt.axhline(y=res.x[1]+perr[1],ls='dotted')

  plt.subplot(1,3,2)
  plt.scatter(con[0],con[2],marker='.')
```

```
plt.xlabel(r"$p_0$")
plt.ylabel(r"$p_2$")
plt.xlim(xl,xh)
plt.ylim(zl,zh)
plt.axvline(x=res.x[0],ls='dashed')
plt.axhline(y=res.x[2],ls='dashed')
plt.axvline(x=res.x[0]-perr[0],ls='dotted')
plt.axvline(x=res.x[0]+perr[0],ls='dotted')
plt.axhline(y=res.x[2]-perr[2],ls='dotted')
plt.axhline(y=res.x[2]+perr[2],ls='dotted')

plt.subplot(1,3,3)
plt.scatter(con[1],con[2],marker='.')
plt.xlabel(r"$p_1$")
plt.ylabel(r"$p_2$")
plt.xlim(yl,yh)
plt.ylim(zl,zh)
plt.axvline(x=res.x[1],ls='dashed')
plt.axhline(y=res.x[2],ls='dashed')
plt.axvline(x=res.x[1]-perr[1],ls='dotted')
plt.axvline(x=res.x[1]+perr[1],ls='dotted')
plt.axhline(y=res.x[2]-perr[2],ls='dotted')
plt.axhline(y=res.x[2]+perr[2],ls='dotted')

plt.tight_layout()
plt.savefig('6_4_contour.pdf')
plt.show()
```

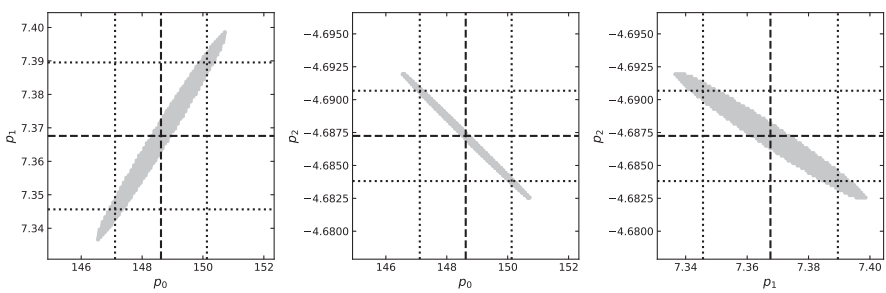
We first discus the fit results then the estimate of the variance on the parameters. Shown in the plot is the three-parameter model with maximum-likelihood estimators superimposed on the data. Since the likelihood fits do not require uncertainties on the data, I have not plotted them. The table below show best estimators for three-, four- and five-parameter models. The results are similar to those obtained in the χ-squared method.

Now let's discuss the variance determination. First we calculate the variance on the parameters using the Hessian matrix. The covariance matrix below is for the three-parameter model. The one standard deviation uncertainties on the parameters are shown for all three models in the table below. The χ^2 obtained is similar to the case of least-squares previously discussed.

We have also calculated the one standard deviation contour for the three-parameter model. Within about three significant digits, the uncertainties are symmetric. The ellipse contours for each pair of parameter uncertainties is shown in the three figures below. The dashed lines show the best estimator values in the centre of the figures. The dotted lines show the uncertainties from the Hessian matrix. The Hessian method gives small uncertainties. The uncertainties from the maximum-likelihood method are considerable smaller then the least-squares method. The least-squares approach assume Gaussian distributed errors and the maximum-likelihood results could be indicating that the uncertainties are non-Gaussian.




```
Optimize estimate: [ 1.49e+02  7.37e+00 -4.69e+00]
Standard errors:   [ 1.51e+00  2.20e-02  3.43e-03]
Covariance matrix:
[[ 2.27e+00  3.27e-02  5.17e-03]
 [ 3.27e-02  4.82e-04  7.38e-05]
 [ 5.17e-03  7.38e-05  1.18e-05]]

Hessian: fit parameters and uncertainties.
p0 =  1.49e+02 +- 1.51e+00
p1 =  7.37e+00 +- 2.20e-02
p2 = -4.69e+00 +- 3.43e-03

Search: fit parameters and uncertainties.
p0 =  1.49e+02 + 2.10e+00 - 2.07e+00
p1 =  7.37e+00 + 3.09e-02 - 3.09e-02
p2 = -4.69e+00 + 4.69e-03 - 4.69e-03

chi-squared: 2326.24
degrees-of-freedom: 88 - 3 = 85
chi-squared/dof: 27.4
```



Parameter	Value
p_0	$(1.49 \pm 0.0151) \times 10^2$
p_1	$(7.37 \pm 0.0220) \times 10^0$
p_2	$(-4.69 \pm 0.00343) \times 10^0$
χ^2/dof	$2326/85 = 27.7$
p_0	$(1.44 \pm 0.137) \times 10^2$
p_1	$(7.34 \pm 0.105) \times 10^0$
p_2	$(-4.71 \pm 0.0624) \times 10^0$
p_3	$(-3.45 \pm 0.113) \times 10^{-3}$
χ^2/dof	$2325/84 = 27.7$
p_0	$(1.42 \pm 0.632) \times 10^5$
p_1	$(1.26 \pm 0.0343) \times 10^1$
p_2	$(1.90 \pm 0.431) \times 10^0$
p_3	$(2.40 \pm 0.154) \times 10^0$
p_4	$(3.19 \pm 0.205) \times 10^{-1}$
χ^2/dof	$2293/83 = 27.6$

Parameter	Value
p_0	$(1.49 + 0.0210 - 0.0207) \times 10^2$
p_1	$(7.37 + 0.0309 - 0.0309) \times 10^0$
p_2	$(-4.69 + 0.00469 - 0.00469) \times 10^0$

6.5 GOODNESS OF FIT

A common way to judge the goodness of a fit is by eye. Plot the fit result on top of the data with error bars and look at how good it is. Sometimes it will be difficult to see, particularly on a log plot. Other metrics can be used.

Plot the data minus the fitted model, $[y_i - f(x_i)]$. These are called residuals. They should be centred on zero with fluctuations. More fluctuations at high x_i are expected. If the residuals are not centred on zero or fluctuating on one side of zero, there may be a bias. One can also plot the ratio of data to the fitted model, $[y_i/f(x_i)]$. Usually $f(x_i) > 0$ but $y_i = 0$ can occur for some i. The distribution should be centred on unity with low x_i having more fluctuations. Alternatively, plot the relative difference $[(y_i - f(x_i))/f(x_i)]$. This distribution should be centred on zero. It is just a shifted version of the previous ratio.

A common goodness-of-fit metric is the pull plot with points calculated using

$$\frac{y_i - f(x_i)}{\sigma_i}\,. \tag{6.42}$$

A good question then arise as to what to use for σ_i? There can be issues with using $\sqrt{y_i}$ for small y_i. So you could try using $\sqrt{f(x_i)}$. Different methods exist for replacing σ_i with other choices. More importantly, if y_i is distributed as a Gaussian, the pull will have a Gaussian distribution.

In the case of least squares, the minimum χ^2 value is a statistic that can be used to test the goodness of fit. The number of degrees of freedom (dof) is the number of measurements N minus the number of fitted parameters M: $\text{dof} = N - M$. Since the mean of the $\chi^2 \approx \text{dof}$, $\chi^2/\text{dof} \approx 1$. This is a useful merit of the goodness of fit.

If $\chi^2/\text{dof} \gg 1$, the errors are probably under estimated, or more likely, the model is wrong. If $\chi^2/\text{dof} \ll 1$, the errors are probably over estimated. If the errors are unknown $\chi^2/\text{dof} = 1$ can be used to estimate the errors. See Example 6.4.5 for the goodness of fit in action.

In the case of maximum likelihood, a smaller value of $-2\ln\lambda(\hat{\theta})$ corresponds to better agreement between data and the hypothesized form $\mu(\boldsymbol{\theta})$.

6.6 HYPOTHESIS TESTING

Statistical hypothesis testing is a systematic methodology for evaluating if a claim, or a hypothesis, is reasonable, or not, bases on data. The null hypothesis H_0 represents the currently accepted state of knowledge. An alternative hypothesis H_A represents a new claim that challenges the current state of knowledge. The two hypotheses H_0 and H_A should be mutually exclusive, so that one and only one of the hypotheses is likely.

The procedure for hypothesis testing is to first formulate the null hypothesis and the alternative hypothesis. Then select a test statistic such that its sampling distribution under the null hypothesis is known. Collect the data independent of the hypotheses. Compute the test statistic from the data and calculate its p-value under the null hypothesis. If the p-value is smaller than a predetermined significance level α, you can reject the null hypothesis. If the p-value is larger, you have fail to reject the null hypothesis.

Consider data as a single random variable x that follows a Gaussian distribution with mean μ and width σ. The pull $g = (x - \mu)/\sigma$ will be distributed as a standard Gaussian of mean 0 and width 1 as shown in Fig. 6.6. For the case of known σ, the probability that measurement x falls within $\pm\delta$ of the true value μ is

$$\begin{aligned} 1 - \alpha &= \frac{1}{\sqrt{2\pi}\sigma} \int_{\mu-\delta}^{\mu+\delta} dx \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] \\ &= \text{erf}\left[\frac{\delta}{\sqrt{2}\sigma}\right] = 2\Phi\left(\frac{\delta}{\sigma}\right) - 1\,, \end{aligned} \tag{6.43}$$

where Φ is the cumulative Gaussian distribution. For $\delta = \sigma, 1 - \alpha = 68.27\%$.

Some key values are

$\alpha = 0.3173$	$\delta = 1\sigma$
$\alpha = 5.7 \times 10^{-7}$	$\delta = 5\sigma$
$\alpha = 0.05$	$\delta = 1.96\sigma$

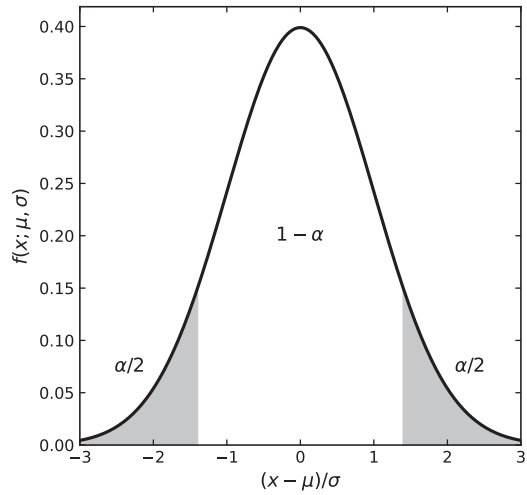


Figure 6.6 Gaussian distribution.

For a one-sided limit, α is one-half the above value. Also, $\alpha = 1 - F(\chi^2; n)$ is an equivalent distribution, where F is the cumulative distribution for a χ^2 distribution with $\chi^2 = (\delta/\sigma)^2$ and $n = 1$ degrees of freedom.

Now consider the maximum likelihood formalism. Define μ to be the hypothesized value of the model (not related to μ in Eq. (6.43)). Typically μ will be the parameter of interest (POI). Other parameters of the model $\boldsymbol{\theta} \equiv (\theta_1, \ldots, \theta_{M-1})$ are not of interest and are called nuisance parameters. $L(\hat{\mu}, \hat{\theta})$ is the maximum (unconditional) likelihood, i.e. $\hat{\mu}$ and $\hat{\theta}$ are their maximum likelihood estimators. $L(\mu, \hat{\hat{\theta}})$ is the conditional likelihood estimator of θ, i.e. the $\hat{\hat{\theta}}$ value that maximizes the maximum-likelihood for a given μ. The profile likelihood ratio is

$$\lambda(\mu) = \frac{L(\mu, \hat{\hat{\theta}})}{L(\hat{\mu}, \hat{\theta})} \,. \tag{6.44}$$

And the quantity

$$t_\mu = -2 \ln \lambda(\mu) \tag{6.45}$$

can be treated as a test statistic in a p-value calculation.

To quantify the level of disagreement, the p-value is

$$p_\mu = \int_{t_{\mu,\text{obs}}}^{\infty} f(t_\mu|\mu) dt_\mu \,, \tag{6.46}$$

where $t_{\mu,\text{obs}}$ is the value of test statistic from data. Here $f(t_\mu|\mu)$ is the pdf of t_μ under the assumption of hypothesis μ. For the log-likelihood ratio, assuming

the model is correct, and a bunch of other conditions [27], t_μ follows a χ^2-distribution. If the p-value is low we should reject our hypothesis (model or fit). Figure 6.7 shows the relationship between the observed test statistic and the p-value.

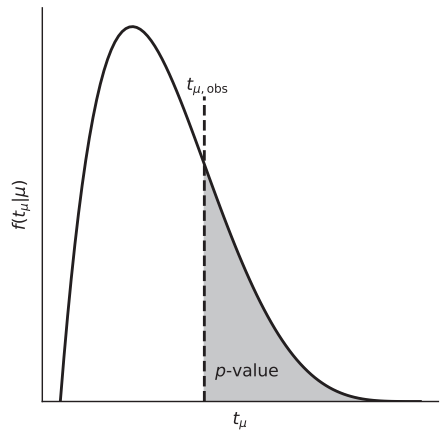


Figure 6.7 Illustration of the relation between the p-value obtained from an observed value of the test statistic t_μ.

Often we convert the p-value to an equivalent significance Z. A Z standard deviation upward fluctuation of a Gaussian random variable would have an upper tail area equal to p.

$$Z = \Phi^{-1}(1-p)\,, \tag{6.47}$$

where Φ is the cumulative Gaussian and Φ^{-1} is inverse (quantile) function. Figure 6.8 shows the relationship between the p-value and significance.

6.6.1 Example: Hypothesis testing

Consider a null hypothesis that claims that a random variable X has mean μ_0. Given a sample of X, we then wish to test if the sampled data is compatible with the null hypothesis.

We will consider the pull-type test statistic [3]

$$t = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}}\,, \tag{6.48}$$

where $\bar{x}$ is the sample mean, σ is the sample deviation and n is the sample size. Since the population variance is unknown in advance, the test statistic follows the Student's distribution $f(t, n-1)$.

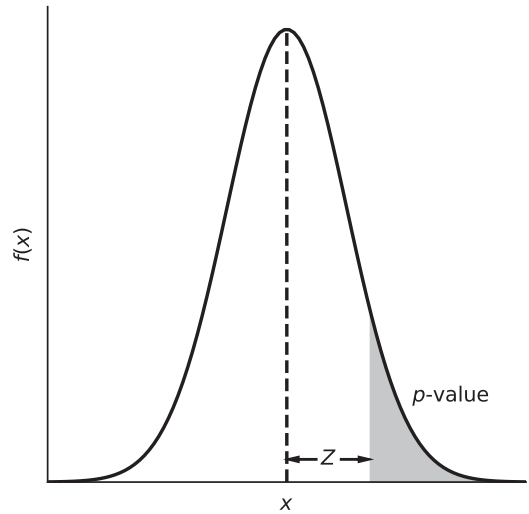


Figure 6.8 Standard normal distribution showing the relation between the significance Z and the p-value.

For this example, we consider a theory in which the null hypothesis says that a particular particle will decay with an exponentially distributed lifetime with mean of $\mu_0 = 1$. We simulate a sample of measured lifetimes by drawing 100 random variables from an exponential distribution slightly different from that claimed by the null hypothesis. I'll choose a lifetime of $\mu = 1.367$ because it is an interesting value for this example. You are welcome to experiment with different sample sizes and sample mean lifetimes.

To understand the code, I will derive the formula for the p-value in terms of the cumulative distribution of the test statistic. If the test statistic t follows the symmetric pdf $f(t)$,

$$\begin{aligned} 1 &= \int_{-\infty}^{-|t|} f(t)dt + \int_{-|t|}^{|t|} f(t)dt + \int_{|t|}^{+\infty} f(t)dt \\ 1 &= F(-|t|) + 1 - p + F(-|t|)\,, \end{aligned} \tag{6.49}$$

where $F(-|t|)$ is the cumulative distribution of $f(t)$;

$$p = 2F(-|t|)\,. \tag{6.50}$$

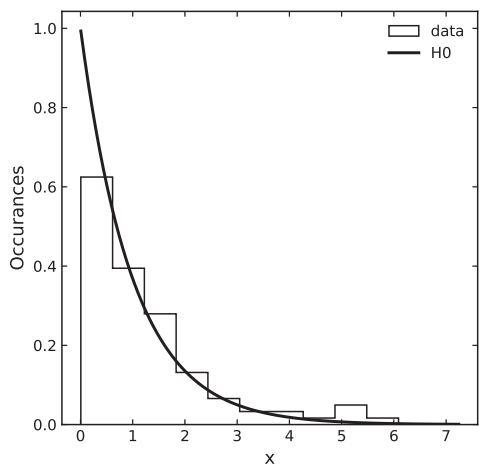
Consider the following code.

```
"""Hypothesis testing against an exponential distribution."""

import numpy as np
from scipy import stats
import matplotlib.pyplot as plt

plt.style.use('./mystyle.mplstyle')
```

```
# Null hypothesis lifetime.
tau0 = 1.0
print("null hypothesis",tau0)
tau  = 1.367
# Simulated data distribution lifetime
# (don't now this in real data).
print("generated sample lifetime",tau)

n = 100
print("sample size",n)
print()

# Generate the simulated data sample.
np.random.seed(0)
T = stats.expon(scale=tau)
T_samples = T.rvs(n)
mean  = T_samples.mean()
sigma = T_samples.std(ddof=1)
print("Data sample mean and deviation",
      np.round(mean,3),np.round(sigma,3))
print()

# Calculate the observed test statistic
# given the null hypothesis.
z = (mean - tau0) / (sigma/np.sqrt(n))
print("observed test statistic",np.round(z,2))
print()

# For fun, calculate test statistic required to
# give a p-value of 5%.
alpha = 0.05
delta = abs(stats.t(df=(n-1)).ppf(alpha/2))
# alpha/2 one-sided.
print("for alpha",alpha,"need test statistic",
      np.round(delta,2))
print()

# Caluclate p-value.
zpvalue = 2*stats.t(df=(n-1)).cdf(-abs(z))
print("observed p-value",np.round(zpvalue,4))
print("significance",
      np.round(stats.norm().ppf(1-zpvalue/2),2),
      "Gaussian standard deviations")

# Calculate test statistic and p-value in one go.
t, p = stats.ttest_1samp(T_samples,tau0)
print()
print("ttest_1samp method: test statistic",np.round(t,2),
      "p-value",np.round(p,4))

# Histogram data on top of null hypothesis.
fig, ax = plt.subplots()

plt.hist(T_samples,density=True,histtype='step',label="data")

x = np.linspace(*T.interval(0.99),num=100)
```

```
ax.plot(x,stats.expon(scale=tau0).pdf(x),lw=2,c='k',label="H0")

plt.xlabel("x")
plt.ylabel("Occurances")
plt.legend()
plt.savefig('6_6_hypothesis.pdf')
plt.show()
```

```
null hypothesis 1.0
generated sample lifetime 1.367
sample size 100

Data sample mean and deviation 1.256 1.29

observed test statistic 1.98

for alpha 0.05 need test statistic 1.98

observed p-value 0.0501
significance 1.96 Gaussian standard deviations

ttest_1samp method: test statistic 1.98 p-value 0.0501
```



We notice that the sample mean and deviation are not exactly as generated. If the sample size is increased, the sample mean and deviation will approach the generated values. For fun, I have calculated the value of the test statistic that would give a p-value of 0.05, and after the fact, have strategically picked the generated lifetime to match this value. I have converted the p-value to a Gaussian significance which often is more intuitive than the p-value; it

is as expected. Lastly, I have verified my code by calculating both the test statistic and p-value using a special library function.

Note that the p-value is small so one might consider the data to be incompatible with the null hypothesis. Nevertheless, the significance is not particularly high; it is nevertheless in the interesting region. A figure of merit for a significant difference would be about five standard deviations. You might also think the generated lifetime is significantly different than the null hypothesis. However, the sample size is only 100 events. Since the test statistic depends on sample size, increasing the sample size while keeping the generated lifetime fixed will increase the discrepancy of the data with null hypothesis. This demonstrates the need to accumulate a high-statistics data sample in order to discover something new.

6.7 PROBLEMS

1. **Function fitting**

   The Bessel function of the first kind of order zero $J_0(x)$ in the interval $0 \leq x \leq 3$ is accurately approximated by

   $$J_0(x) = 1 - 2.2499997(x/3)^2 + 1.2656208(x/3)^4 - 0.31163866(x/3)^6 + 0.0444479(x/3)^8 - 0.0039444(x/3)^{10} + 0.0002100(x/3)^{12}\,.$$

   Calculate 300 points in the interval $0 \leq x \leq 3$ using this equation. Treat these generated (x_i, y_i) pairs as data. Fit this data to polynomials $P(x_i)$ from quadratic up to 15th order (i.e. $n = 3$ to 15).

   (a) Plot the fit results for each polynomial on top of the data.

   (b) On a separate plot, show the estimated maximum residual $\max(|PdY(x_i) - y_i|)$ versus the number of fit parameters n on a log-y scale.

   (c) Estimate the χ^2 error by assuming the χ^2 per degree of freedom is unity and all the errors are equal. On a separate plot, show this chi-squared estimated error versus the number of fitting parameters n on a log-y scale.

   (d) Comment on the trend you see in the plots of part (b) and part (c), and also compare (b) and (c).

   You may use `scipy.optimize` in this problem.

2. **Stefan-Boltzmann law**

   Inspired by Alex Gezerlis [2]

   The Stefan-Boltzmann law describes the thermal radiation emitted by matter in terms of the matter's temperature. For a black-body (ideal absorber and emitter), the law can be written as $I = \sigma T^4$, where I is the

energy radiated per second per surface area at absolute temperature T and σ is the Stefan-Boltzmann constant. Lummer and Pringsheim [28] attempted to confirm the temperature dependence by performing an experiment and recording the data shown in the cell below. The x-axis measurements T are absolute temperature, as in the Stefan-Boltzmann law. However, the y-axis data Y are not in units that allow a determination of σ, and the data can have a y-axis offset: the measurements are called the reduced deflection.

```
import numpy as np
T = np.array([371.1,492.5,733,755,799,820,877,1106,1125,1403,1492,
              1522,1561])
Y = np.array([156.0,638,3320,3810,4440,5150,6910,16400,17700,44700,
              57400,60600,67800])
```

(a) Plot the data. Discuss your choice of linear or log-scale axes.

(b) Fit the data to the model $p(x) = c_0 + c_1 x^{c_2}$, where c_0, c_1, c_2 are parameters to be determined by the fit. Do not assume the form of the Stefan-Boltzmann law.

(c) Explain how you handled the data uncertainties, and why.

(d) Superimpose the resulting fit on the data.

(e) What is your temperature exponent and its uncertainty?

(f) What is the chi-squared of the fit? Is the chi-squared reasonable? Can you say anything about the data uncertainties?

You may use `scipy.optimize` in this problem.

3. **Gravitational ringdown**

(a) Read in the data from file ringdown.txt which contains two NumPy arrays: `(x,y) = numpy.loadtxt("ringdown.txt")`.

(b) Plot all the data using a log y-axis (linear x-axis). The x-axis is time.

Globally, the data decreases with increasing time. Locally, the data is quasiperiodic with a high-time tail which shows power-law behaviour.

(c) Fit the data to a four-parameter model $|\exp(i\omega t)|$, where $\omega = \omega_R + i\omega_I$ is a complex frequency, i.e. $A\exp(-\omega_I t)|\sin(\omega_R t + \phi)|$, with $A, \omega_R, \omega_I, \phi$ the four fit parameters.

Since the data has a variable oscillation period but the period looks constant in a central time region, fit a central time region by ignoring the low-time and high-time regions in your fit.

(d) Although you have only fit a central region in time, plot your model fit result over the entire time range on top of the data plot. Use the parameters you obtained from the fit. Indicate on the plot with vertical lines your fit range.

(e) Show that the fit converges. Examine the parameter values and their uncertainty to make sure the fit makes sense. We are really only interested in the real frequency ω_R. You don't need to demonstrate the goodness of fit, although you can if you like.

You may use `scipy.optimize`. Congratulations, you have calculated the fundamental frequency of a Schwarzschild black hole after it has been gravitationally perturbed.

4. **Confidence-level contour**

Imagine that you have a likelihood function in two parameters. Although I give you an analytic form for this function below, you are to solve the problem as if you did not know its analytic form.

```
import numpy as np
def likelihood(p0,p1,n=0):
    return (13*p0**2 + 6*np.sqrt(3)*p0*p1 + 7*p1**2)/2 - n
```

(a) Numerically minimize the likelihood function to find the best estimators for the two parameters.

(b) Numerically determine the one standard deviation likelihood contour, and plot it.

(c) Determine the estimators uncertainties. Are they symmetric?

5. **Pseudo-experiments**

Start with the function

$$f(x; p_0, p_1, p_2) = p_0(1-x)^{p_1} x^{p_2} ,$$

where $x = m_{\mathrm{jj}}/\sqrt{s}$ and $\sqrt{s} = 13\,\mathrm{TeV}$. Use the following parameters obtained by fitting the data at the end of the problem.

Parameter	Value
p_0	1.50e+02
p_1	7.38e+00
p_2	-4.68e+00

We will consider this our model for the data. How do we determine the uncertainty on the model? For simplistic purposes, we will only discuss here the statistical uncertainty.

Consider the idea that our experiment that measured and recorded the data was only one of many possible experiments. Each experiment would fit their data to the same function and obtain slightly different fit parameters. If each experiment had identical resolution and recorded infinite statistics they would all obtain exactly the same fit parameters.

Our experiment is one of an ensemble of possible experiments with finite statistics. If we consider our resulting model an unbiased estimator of the data, we can use it to generate pseudo-distributions that simulate the possible outcomes of other experiments. We will refer to our model function and fit parameters above as the nominal model.

(a) Assume each bin of data obeys Poisson statistics. For a given bin, use the value of the nominal function evaluated at a bin centre (given below in units of TeV) as the mean of a Poisson distribution. For a given bin, generate a random Poisson distributed variable. Use this random value as a data point in that bin for a pseudo-experiment. Repeat through all bins building up the entire pseudo-data distribution for one pseudo-experiment. This is one instance of our Poisson fluctuated model.

(b) The pseudo-distribution has fluctuations but that is not our main interest. We wish to know the resulting fluctuation in the model function. Fit the pseudo-data distribution to the model function but with free parameters to be determined by the fit.

(c) Evaluate the new function values at the bin centres. Store the results of each bin in an array or histogram.

(d) Repeat the entire procedure (a) through (c) to generate 1000 pseudo-experiments. Each will result in slightly different fit parameter values. Show the pseudo-data and resulting function on the same plot for each pseudo-experiment.

(e) From the distribution of function values in each bin i calculate the mean μ_i and standard deviation σ_i.

The distribution in each bin should have a Poisson shape that approached a Gaussian shape for a large number of pseudo-experiments. The distributions should be centred on a mean value of μ_i corresponding to the nominal function values from which the Poisson distributed variables are generated. Alternatively, the distributions could be fit by a Gaussian function to determine the means μ_i and widths σ_i, but we will not do that here.

(f) Plot the distribution of pull values.

We now have a standard deviation for each bin.

(g) Plot the ratio of the two functions given by nominal $\pm$ σ of the nominal function to the nominal function. The envelope of the two functions should represent the statistical uncertainty on our model fit.

(h) State approximately what the maximum percentage statistical uncertainty on the model is.

```
x = numpy.array([
    1.217 , 1.2515, 1.287 , 1.323 , 1.3595, 1.397 , 1.435 , 1.4735,
    1.513 , 1.553 , 1.5935, 1.635 , 1.677 , 1.7195, 1.763 , 1.8075,
    1.8525, 1.898 , 1.9445, 1.992 , 2.0405, 2.0895, 2.139 , 2.1895,
    2.241 , 2.2935, 2.347 , 2.4015, 2.457 , 2.5135, 2.571 , 2.6295,
    2.689 , 2.7495, 2.811 , 2.8735, 2.937 , 3.0015, 3.067 , 3.1335,
    3.201 , 3.27  , 3.3405, 3.412 , 3.4845, 3.5585, 3.634 , 3.7105,
    3.788 , 3.867 , 3.9475, 4.029 , 4.112 , 4.1965, 4.2825, 4.37  ,
    4.459 , 4.5495, 4.6415, 4.735 , 4.83  , 4.9265, 5.0245, 5.1245,
    5.226 , 5.329 , 5.434 , 5.541 , 5.65  , 5.761 , 5.874 , 5.989 ,
    6.106 , 6.225 , 6.346 , 6.469 , 6.5945, 6.7225, 6.8525, 6.985 ,
    7.12  , 7.257 , 7.3965, 7.5385, 7.683 , 7.83  , 7.9795, 8.1315
])
```

6. **Hypothesis testing: Gaussian distribution**

Consider a null hypothesis that claims that a random variable X follows a Gaussian distribution with mean $\mu_0 = 1$ and a standard deviation of $\sigma_0 = 0.5$. Given a sample of X, you will test if the sampled data is compatible with the null hypothesis.

Consider the pull-type test statistic

$$t = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}} ,$$

where n is the number of data values and $\bar{x}$ is the mean of the data sample X.

If the population standard deviation σ_0 is known, the test statistic is normally distributed with a mean of 0 and a width of 1. If the population variance is not known, usually the sample standard deviation and the test statistic follows the Student's t-distribution.

For an unbiased estimator of the population variance, use

$$\sigma^2 = \frac{1}{n-1} \sum_{i=0}^{n-1} (X_i - \bar{x})^2 ,$$

with $(n-1)$ degrees of freedom.

(a) Read in the data measurements from the file `data.txt` which contains one NumPy array: `X = numpy.loadtxt("data.txt")`.

(b) Histogram the data.

(c) Plot the null hypothesis on top of the data.

(d) Calculate the null hypothesis test statistic (normal distribution: population standard deviation known).

(e) Calculate the null hypothesis test statistic (Student's t-distribution: use sample standard deviation).

(f) Calculate the statistic threshold for a significance of 5% (2.5% on each side).

(g) Calculate the p-values and standard Z (Gaussian) deviations for both test statistics.

(h) Comment on your results.

You may use `scipy.stats`. Do not use `scipy.stats.ttest_1samp`. Although you may use it for validation. This problem was inspired by Ref. [3].

Matplotlib style sheet

The following Matplotlib custom style sheet was used in the Python code for this book.

```
lines.linewidth : 2
xtick.top       : True
ytick.right     : True
xtick.direction : in
ytick.direction : in
axes.linewidth  : 1
axes.prop_cycle : cycler('color',['k'])
axes.labelsize  : large
legend.frameon  : False
savefig.bbox    : tight
savefig.dpi     : 300
```

DOI: 10.1201/9781003660873-A

Data for problems

This code makes the data for problem 5.5.3.

```
import numpy
import scipy.stats as stats
x = numpy.linspace(0,10,100+1)
y = numpy.pi * stats.gamma.pdf(x,a=3,scale=1)
file = open("function.csv","w")
for i in range(len(x)):
    s = str(x[i]) + "," + str(y[i]) + "\n"
    file.write(s)
file.close()
```

The data for problem 4.5.4 and problem 6.7.6 is the plot made in problem 3.3.15.

This code makes the data for problem 6.7.6.

```
import numpy as np
from scipy import stats
mu = 0.85
n = 100
np.random.seed(0)
X = stats.norm(mu,sigma0)
X_samples = X.rvs(n)
np.savetxt("data.txt",X_samples)
```


DOI: 10.1201/9781003660873-B

Bibliography

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipies*. The Art of Scientific Computing. Cambridge University Press, third edition, 2007.

[2] Alex Gezerlis. *Numerical Methods in Physics with Python*. Cambridge University Press, 2020.

[3] Robert Johansson. *Numerical Python*. Scientific Computing and Data Science Application with Numpy, ScPy and Matplotlib. A press, second edition, 2019.

[4] Herbert D. Peckham. *Computers, BASIC, and Physics*. Addison-Wesley, 1971.

[5] Lewis Fry Richardson. IX. The approximate arithmetical solution by finite differences of physical problems involving differetial equations, with an application to the stresses in a masonry dam. *Proceedings of the Royal Society of London. Series A*, 210:307–357, 1911.

[6] Rubin H. Landaus, Manuel J. Páez, and Cristian C. Bordeianu. *Computatational Physics*. Problem Solving with Python. Wiley-VCH, 2024.

[7] Alejandro L. Garcia. *Numerical Methods for Physics (Python)*. Self-published, revised second edition, 2017.

[8] John M. Boyd. *Chebyshev and Fourier Spectal Methods*. Dover Publicatons, second (revised) edition, 2001.

[9] Hakan Ciftci, Richard L Hall, and Nasser Saad. Asymptotic iteration method for eigenvalue problems. *Journal of Physics A: Mathematical and General*, 36(47):11807–11816, November 2003.

[10] H. T. Cho, A. S. Cornell, Jason Doukas, T. R. Huang, and Wade Naylor. A New Approach to Black Hole Quasinormal Modes: A Review of the Asymptotic Iteration Method. *Advances in Mathematical Physics,* 2012:281705, 2012.

[11] E. W. Leaver. An Analytic representation for the quasi normal modes of Kerr black holes. *Proceedings of the Royal Society of London. Series A*, 402:285–298, 1985.

[12] Ian H. Hutchinson. *A Student's Guide to Numerical Methods*. Cambridge University Press, 2015.

[13] M.L. James, G.M. Smith, and J.C. Wolford. *Applied Numerical Methods for Digital Computation*. IEP–A Dun-Donnelley Publisher, 1977.

[14] William H. Press and Saul A. Teukolsky. Perturbations of a Rotating Black Hole. II. Dynamical Stability of the Kerr Metric. *The Astrophysical Journal*, 185:649–674, October 1973.

[15] E. Wasserstrom. A new method for solving eigenvalue problems. *Journal of Computational Physics*, 9(1):53–74, 1972.

[16] Sølve Selstø. *A Computational Introduction to Quantum Physics*. Problem Solving with Python. Cambridge Universtiy Press, first edition, 2024.

[17] James M. Cooley and John W. Tukery. An algorithm for the machine calculation of the xomplex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[18] G. O. Roberts, A. Gelman, and W. R Gilks. Weak convergence and optimal scaling of random walk Metropolis algorithms. *The Annals of Applied Proboability*, 7:110, 1997.

[19] Andi Klein and Alexander Godunov. *Introductory Computational Physics*. Cambridge University Press, 2006.

[20] Mark Newman. *Computational Physics*. Self-published, revised and expanded edition, 2013.

[21] G. P. Lepage. Lattice QCD for novices. In *13th Annual HUGS AT CEBAF*, pages 49–90, 5 1998.

[22] G Peter Lepage. A new algorithm for adaptive multidimensional integration. *Journal of Computational Physics*, 27(2):192–203, 1978.

[23] R. Aggarwal and A. Caldwell. Error Bars for Distributions of Numbers of Events. *Eur. Phys. J. Plus*, 127:24, 2012.

[24] Glen Cowan. *Statistical Data Analysis*. Oxford University Press, 1998.

[25] Frederick James. *Statistical Mehotds in Experimental Physics*. World Scientific Publishing, second edition, 2006.

[26] V. V. Gligorov, S. Hageboeck, T. Nanut, A. Sciandra, and D. Y. Tou. Avoiding biases in binned fits. *Journal of Instrumentation*, 16(08):T08004, 2021.

[27] Glen Cowan, Kyle Cranmer, Eilam Gross, and Ofer Vitells. Asymptotic formulae for likelihood-based tests of new physics. *European Physical Journal C*, 71:1554, 2011. [Erratum: Eur.Phys.J.C 73, 2501 (2013)].

[28] O. Lummer and E. Prinsheim. Die Strahlung eines "schwarzen" Körpers zwischen 200 and 1300 C. *Annalen deer Physik und Chemie*, 299:395, 1897.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Index