



PYTHON FOR PHYSICIST

Before You Say Goodbye to Python



HARADHAN ADHIKARY

Python for Physicist by HA

May 17, 2021

This book is dedicated to all my teachers

Copyright © 2021 Haradhan Adhikary.

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Introduction

Python is a free, open source, easy-to-use software tool that offers a significant alternative to proprietary packages such as Matlab and Mathematica. This book covers everything the working scientist needs to know to start using Python effectively to study many interesting mathematical and physical phenomena using Python3. I am assuming, you have prior knowledge about python coding. In this note, I will discuss several topics in different ways. Also I will explain in details about all topics as far my knowledge. I hope this note will be helpful to you, if you working to construct statistical model or data analysis or if you are a student.

The Python programming language is useful for all kinds of scientific and engineering tasks. You can use it to analyze and plot data. You can also use it to numerically solve science and engineering problems that are difficult or even impossible to solve analytically. While we want to marshal Python's powers to address scientific problems, you should know that Python is a general purpose computer language that is widely used to address all kinds of computing tasks, from web applications to processing financial data on Wall Street and various scripting tasks for computer system management. Over the past decade it has been increasingly used by scientists and engineers for numerical computations, graphics, and as a “wrapper” for numerical software originally written in other languages, like Fortran and C.

Python has many powerful but unfamiliar facts, and these need more explanation than the familiar ones. In particular, if you encounter in this text are reference to the “beginner” or the “unwary”, it signifies a point which is not made clear in the documentation, and has caught out this author at least once.

By “Physicist”, I mean anyone who uses quantitative models either to obtain conclusions by processing precollected experimental data or to model potentially observable results from a more abstract theory, and who asks “what if?”. What if I analyze the data in a different way? What if I change the model? Thus the term also includes economists, engineers, mathematicians among others, as well as the usual concept of scientists. Given the volume of potential data or the complexity (non-linearity) of many theoretical models, the use of computers to answer these questions is fast becoming mandatory.

The purpose of this intentionally short book is to show how easy it is for the working scientist to implement and test non-trivial mathematical algorithms using Python. We have quite deliberately preferred brevity and simplicity to encyclopaedic coverage in order to get the inquisitive reader up and running as soon as possible. We aim to leave the reader with a well-founded framework to handle many basic, and not so basic, tasks. Obviously, most readers will need to dig further into techniques for their particular research needs. But after reading this book, they should have a sound basis for this.

No prior knowledge of programming is needed to read this book. We start with some very simple examples to get started with programming and then move on to introduce fundamental programming concepts such as loops, functions, if-tests, lists, and classes. These generic concepts are supplemented by more specific and practical tools for scientific programming, primarily plotting and array based computations. The book's overall purpose is to introduce the reader to programming and, in particular, to demonstrate how programming can be an extremely useful and powerful tool in many branches of the natural sciences.

Haradhan Adhikary

Contents

1. Getting started in Python
 - a. Why Python language?
 - i. The design philosophy of Python
 - b. Download Python
 - c. Getting Up and Running With Jupyter Notebook
 - i. Installation
 - ii. Creating a Notebook
 - iii. A first program (Celsius to Fahrenheit converter)
 - d. Key-notes of Python programming
 - i. for loop and range function
 - ii. Random numbers
 - iii. The math function
 - iv. if else statement
 - v. elif statement
 - vi. Flag variable
 - vii. String
 - viii. List
 - ix. Function
2. Python for Mathematical concept
 - a. Fibonacci sequence
 - b. Prime numbers
 - c. Use of Numpy
 - d. Creation of Matrix using 'numpy.matrix() function'
 - e. Addition of Matrix in Python
 - i) Traditional method
 - ii) using '+' operator
 - f. Subtraction of matrix using Python
 - g. Transpose of a Python Matrix
 - h. Exponent of a Python Matrix
3. Graphical plotting and histogram

- a. Making plots
- b. Working With Pyplot: Plotting Routines
- c. Histograms
- 4. Mechanical Oscillator
 - a. Mechanical oscillator using Python
 - b. Damped Harmonic oscillator
- 5. Application of Python for electrical circuit
 - a. DC circuit using Python
 - b. Kirchoff current law using Python
- 6. Reading data from files
 - a. Read and write from/to text file
 - b. Applications
- 7. Python numerical integration
 - a. Simpson $\frac{1}{3}$ rule
 - b. Trapezoidal rule
 - c. Riemanns integral
- 8. Root finding in Python
 - a. Bisection method using Python
 - b. Newton-Raphson method using Python
 - c. Secant method using Python
- 9. Special function using Python
 - a. Besel function
 - b. Gamma function
 - c. Airy function
 - d. Legendre polynomials
 - e. Laguerre polynomials
 - f. Hermite polynomials
- 10. Solving one dimensional differential equations
 - a. Solve differential equations using Python
 - b. Solving systems of nonlinear equations using Python
- 11. Fourier Series using Python
 - a. Fourier series analysis for a sawtooth wave function

- b. Fourier series analysis for a square wave function
 - c. Fourier series analysis for a Triangular wave function
 - d. Fourier series analysis for an Arbitrary wave function
- 12. Discrete (Fast) Fourier Transform
 - a. Continuous Fourier Transformation
 - b. Discrete Fourier Transformation (FFT)
- 13. Logic GATE
 - a. NOT GATE
 - b. AND GATE
 - c. OR GATE
 - d. XOR, XNOR, NAND GATE
- 14. Python for Electromagnetism
 - a. Visualizing a vector field with Matplotlib
 - b. Electric field and potential due to a charge particle
 - i. Electric field for charge particle
 - ii. Electric potential for charge particle
 - iii. Electric field lines and potential for four charges on square
 - iv. Maxwell's plot
 - c. Electrostatic potential of an electric dipole
 - d. Magnetism using Python
 - i. Magnetic field of a straight wire
 - ii. Magnetic field produced by a dipole
 - e. Charged Particle Trajectories in Electric and Magnetic Fields
- 15. Basic Operations on Quantum Objects using Python
 - a. Particle in a box problem with application
 - b. Harmonic oscillator in quantum mechanics
 - c. Creating and inspecting quantum objects (matrix representation)
 - d. States and operators using Python
 - i. Density matrices
 - ii. Pauli Sigma matrix properties
 - iii. Harmonic oscillator operators
 - iv. Quantum annihilation and creation operators

- v. Quantum commutation relation using Python
 - e. Cat vs coherent states in a Kerr resonator, and the role of measurement
- 16. WKB approximation using Python
- 17. Monte Carlo Study of Ferro-magnetism using an Ising Model
 - a. Monte Carlo methods using Python
 - b. Random number generation using Python
 - i. Creating Random Numbers
 - ii. Testing for randomness
 - iii. Random Walks and the Markov process
 - iv. The Metropolis algorithm
 - c. Ferromagnetism using the Ising Model
 - d. Ising Model using Python
 - i. Road map of 1d Ising model
 - ii. Critical dynamics in a 1-D Ising model
 - iii. Exercise of 2D ising model
- 18. Python for Solid state physics
 - a. Visualizing the reciprocal lattice in 2D and unit cell
 - b. The reciprocal space lattice
 - c. Construction of the first Brillouin zone
 - d. Tight-binding package for Python
 - e. Square lattice using Pybinding package
 - f. Graphene using Python
 - i. Brillouin zone
 - ii. Switching lattices
- 19. The Chaos theory using Python
 - a. Chaos on a Strange Attractor
 - b. Lorenz equations
 - c. Butterfly effect using Python

Upcomig topics

1. Python for Quantum Field Theory (Q.F.T)
2. Python for Quantum Chromodynamics (Q.C.D)

1 Getting started in Python

In this chapter I will discuss an overview of the Python language and its core philosophy, how to set up the Python development environment, and the key concepts around Python programming to get you started with basics. This introductory chapter for those who are not familiar with Python programming language. If you are already comfortable with Python, I would recommend that you quickly run through the contents to ensure you are aware of all of the key concepts or just go to next chapter.

1.1 Why Python language ?

As the saying goes, **“The best things in life are free!”**. Python is an open source, high-level, object-oriented, interpreted, and general-purpose dynamic programming language. Python’s core design theory underline code readability, and its coding structure enables programmers to articulate computing concepts in fewer lines of code as compared to other high-level programming languages such as Java, C or C++.

1.1.1 The design philosophy of Python :

- Simple is better than complex – keep it simple and stupid (KISS).
- Beautiful is better than ugly – be consistent.
- Complex is better than complicated – use existing libraries.
- Although practicality beats purity - if required, break the rules.
- Unless explicitly silenced – error logging and traceability.
- In ambiguity, refuse the temptation to guess – Python syntax is simpler; however, many times we might take a longer time to decipher it.
- Although that way may not be obvious at first unless you’re Dutch – there is not only one of way of achieving something.
- There should be preferably only one obvious way to do it – use existing libraries.

- If the implementation is hard to explain, it's a bad idea – if you can't explain in simple terms then you don't understand it well enough.
- Now is better than never – there are quick/dirty ways to get the job done rather than trying too much to optimize.
- Although never is often better than *right* now – although there is a quick/dirty way, don't head in the path that will not allow a graceful way back.
- Namespaces are one honking great idea, so let's do more of those! – be specific.
- If the implementation is easy to explain, it may be a good idea – simplicity.
- Flat is better than nested – avoid nested ifs.
- Explicit is better than implicit – be clear.
- Sparse is better than dense – separate code into modules.
- Readability counts – indenting for easy readability.
- Special cases aren't special enough to break the rules – everything is an object.
- Errors should never pass silently – good exception handler.

Python was officially born on February 20, 1991, with version number 0.9.0 and has taken a tremendous growth path to become the most popular language for the last 10 years in a row (2010 to 2020). Its application cuts across various areas such as website development, mobile apps development, scientific and numeric computing, desktop GUI, and complex software development. Even though Python is a more general-purpose programming and scripting language, it has been gaining popularity over the past 10 years among data scientists and Machine Learning engineers.

Python has lot of powerful modules such as NumPy and Pandas exist for the efficient use of numeric data. Scientific computing is made easy with SciPy package. A number of primary machine learning algorithms have been efficiently implemented in scikit-learn (also known as sklearn). HadoopPy, PySpark provides seamless work experience with big data technology stacks. Cython and Numba modules allow executing Python code in par with the speed of C code. Modules such as nosetest emphasize high-quality, continuous integration tests, and automatic deployment.

Combining all of the above has made many Physicist to use Python as the choice of language to explore data, identify patterns, and build and deploy models for research purpose and to solve many Physics problems. Most importantly the business-friendly licenses for various key Python packages are encouraging the collaboration of many experiment like CMS, ATLAS, STAR , RHIC etc. and the open source community for the benefit of both worlds. Overall the Python programming ecosystem allows for quick results and happy programmers. We have been seeing the trend of developers being part of the open source community to contribute to the bug fixes and new algorithms for the use by the global community, at the same time protecting the core IP of the respective company they work for.

So I think I able to convince you to use Python for your research work or to use Python for your study or to build-up some Physics model.

1.2 Download Python :

You can go to Python's official website <https://www.python.org/downloads/> and browse to the appropriate OS section and download the installer. Note that OSX and most of the Linux come with preinstalled Python so there is no need of additional configuring.

All the code used in this book are available as IPython Notebook (now known as the Jupyter Notebook), it is an interactive computational environment, in which you can combine code execution, rich text, mathematics, plots and rich media. You can launch the Jupyter Notebook by clicking on the icon installed by Anaconda in the start menu (Windows) or by typing 'jupyter notebook' in a terminal (cmd on Windows). Then browse for the relevant IPython Notebook file that you would like to paly with. Note that the codes can break with change in package version, hence for reproducibility, I have shared my package version numbers, please refer Module_Versions IPython Notebook.

You can get more details information about **Jupyter Notebook** from this website <https://jupyter.readthedocs.io/en/latest/install.html>

1.3 Getting Up and Running With Jupyter Notebook:

The Jupyter Notebook is not included with Python, so if you want to try it out, you will need to install Jupyter. There are many distributions of the Python language. This section will focus on just two of them for the purposes of installing Jupyter Notebook. The most popular is CPython, which is the reference version of Python that you can get from their <https://www.python.org/>. It is also assumed that you are using Python 3.

1.3.1 Installation:

If so, then you can use a handy tool that comes with Python called pip to install Jupyter Notebook like this:

```
$ pip install jupyter
```

The next most popular distribution of Python is Anaconda. Anaconda has its own installer tool called conda that you could use for installing a third-party package. However, Anaconda comes with many scientific libraries preinstalled, including the Jupyter Notebook, so you don't actually need to do anything other than install Anaconda itself.

1.3.2 Starting the Jupyter Notebook Server:

Now that you have Jupyter installed, let's learn how to use it. To get started, all you need to do is open up your terminal application and go to a folder of your choice. I recommend using something like your Documents folder to start out with and create a subfolder there called Notebooks or something else that is easy to remember.

Then just go to that location in your terminal and run the following command:

```
$ jupyter notebook
```

Use command **Shift+Enter**

This will start up Jupyter and your default browser should start (or open a new tab) to the following URL: `http://localhost:8888/tree`

Note that right now you are not actually running a Notebook, but instead you are just running the Notebook server. Let's actually create a Notebook now!

1.3.3 Creating a Notebook:

Now that you know how to start a Notebook server, you should probably learn how to create an actual Notebook document.

All you need to do is click on the New button (upper right), and it will open up a list of choices. On my machine, I happen to have Python 2 and Python 3 installed, so I can create a Notebook that uses either of these. For simplicity's sake, let's choose Python 3.

Now you can have a look any tutorial <https://learn.onemonth.com/jupyter-notebook-a-beginners-tutorial/> for Jupyter Notebook then you will be master to use Jupyter Notebook. One important point about Jupyter Notebook, you can use Latex which is in build in Jupyter Notebook.

1.4 A first program (Celsius to Fahrenheit converter):

```
[1]: temp =eval(input("Enter a temperature in Celsius:"))
      print("In Fahrenheit, that is ", 9/5*temp+32)
```

```
Enter a temperature in Celsius:76
In Fahrenheit, that is  168.8
```

Let's examine how the program does what it does. The first line asks the user to enter a temperature. The `input` function's job is to ask the user to type something in and to capture what the user types. The part in quotes is the prompt that the user sees. It is called a string and it will appear to the program's user exactly as it appears in the code itself. The `eval` function is something we use here, but it won't be clear exactly why until later. So for now, just remember that we use it when we're getting numerical input.

We need to give a name to the value that the user enters so that the program can remember it and use it in the second line. The name we use is `temp` and we use the equals sign to assign the user's value to `temp`.

The second line uses the `print` function to print out the conversion. The part in quotes is another string and will appear to your program's user exactly as it appears in quotes here.

1.5 Key-notes of Python programming

Let's first discuss few important key points to do coding in Python. Basically if you know any programming language the basic concept of all this key points are same in any programming language.

1.5.1 For loop:

```
[2]: for i in range(3):  
      print('Hello')
```

```
Hello  
Hello  
Hello
```

The structure of a for loop is as follows:

for variable name in range(number of times to repeat):

statements to be repeated

The syntax is important here. The word for must be in lower case, the first line must end with a colon, and the statements to be repeated must be indented. Indentation is used to tell Python which statements will be repeated.

1.5.2 The range function:

The value we put in the range function determines how many times we will loop. The way range works is it produces a list of numbers from zero to the value minus one. For instance, range(5) produces five values: 0, 1, 2, 3, and 4.

If we want the list of values to start at a value other than 0, we can do that by specifying the starting value. The statement range(1,5) will produce the list 1, 2, 3, 4. This brings up one quirk of the range function it stops one short of where we think it should. If we wanted the list to contain the numbers 1 through 5 (including 5), then we would have to do range(1,6).

Another thing we can do is to get the list of values to go up by more than one at a time. To do this, we can specify an optional step as the third argument. The statement range(1,10,2) will step through the list by twos, producing 1, 3, 5, 7, 9.

To get the list of values to go backwards, we can use a step of -1. For instance, range(5,1,-1) will produce the values 5, 4, 3, 2, in that order. (Note that the range function stops one short of the ending value 1).

```
[3]: for i in range(5,0,-1):  
      print(i, end=' ')  
      print('Blast off!!!')
```

```
5 4 3 2 1 Blast off!!!
```

Homework:

1. Write a program that uses a for loop to print the numbers 8, 11, 14, 17, 20, . . . , 83, 86, 89.
2. Write a program that uses a for loop to print the numbers 100, 98, 96, . . . , 4, 2.

3. Write a program that asks the user for their name and how many times to print it. The program should print out the user's name the specified number of times.
4. Write a program that prints a giant letter A like the one below. Allow the user to specify how large the letter should be.

1.5.3 Random numbers:

To make an interesting computer game or Physics model, it's good to introduce some randomness into it. Python comes with a module, called `random`, that allows us to use random numbers in our programs. Before we get to random numbers, we should first explain what a module is. The core part of the Python language consists of things like for loops, if statements, math operators, and some functions, like `print` and `input`. Everything else is contained in modules, and if we want to use something from a module we have to first import it—that is, tell Python that we want to use it

```
[4]: from random import randint
x = randint(1,10)
print('A random number between 1 and 10:', x)
```

A random number between 1 and 10: 3

1.5.4 The Math function:

The math module Python has a module called `math` that contains familiar math functions, including `sin`, `cos`, `tan`, `exp`, `log`, `log10`, `factorial`, `sqrt`, `floor`, and `ceil`. There are also the inverse trig functions, hyperbolic functions, and the constants `pi` and `e`. Here is a short example:

```
[5]: from math import sin, pi
print('Pi is roughly', pi)
print('sin(0) =', sin(0))
print(abs(-4.3))
print(round(3.336, 2))
print(round(345.2, -1))
```

Pi is roughly 3.141592653589793
sin(0) = 0.0
4.3
3.34
350.0

The `round` function takes two arguments: the first is the number to be rounded and the second is the number of decimal places to round to. The second argument can be negative.

Homework:

1. Write a program that generates 50 random numbers such that the first number is between 1 and 2, the second is between 1 and 3, the third is between 1 and 4, . . . , and the last is between 1 and 51.

2. Write a program that asks the user to enter an angle between -180° and 180° . Using an expression with the modulo operator, convert the angle to its equivalent between 0° and 360° .
3. Write a program that asks the user for a number of seconds and prints out how many minutes and seconds that is. For instance, 200 seconds is 3 minutes and 20 seconds.
4. Write a program that asks the user for a number and prints out the factorial of that number.
5. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years unless they are also divisible by 400. Ask the user to enter a year, and, using the // operator, determine how many leap years there have been between 1600 and that year.

1.5.5 If else statements:

Quite often in programs we only want to do something provided something else is true but if that is false we want to do something. Python's if else statement is what we need. Let's try a guess-a-number program. The computer picks a random number, the player tries to guess, and the program tells them if they are correct or else ask to try again. To see if the player's guess is correct, we need something new, called an if statement.

```
[6]: from random import randint
num = randint(1,10)
guess =eval(input('Enter your guess:'))

if guess==num:
    print('You got it!')
else:
    print("Try again")
print("your random number was: ",num)
```

```
Enter your guess:3
Try again
your random number was:  10
```

Conditional operators: In Physics many times we use many conditions like : The **Curie temperature** of iron is 1043 K (The **Curie temperature** is an essential temperature for a ferromagnetic material. For example, if a ferromagnetic material has a temperature under its Curie temperature, then the material has a net spontaneous magnetization, which means that the material becomes ferromagnetic, or magnetic. If a ferromagnetic material has a temperature over its Curie temperature, then the material becomes paramagnetic, or does not become a magnet). When the temperature of iron is at the Curie temperature or higher, then the iron becomes paramagnetic and when the temperature of iron is below the Curie temperature, then it is ferromagnetic.

```
[7]: T =eval(input('Enter temparature of the element in Kelvin: '))
Tc = 1043 #Critical tempareture of Iron in K
if T > Tc:
    print("Iron becomes paramagnetic")
```

```
else:
    print("Iron is ferromagnetic")
```

Enter temperature of the element in Kelvin: 2000

Iron becomes paramagnetic

There are three additional operators used to construct more complicated conditions: **and**, **or**, and **not**. Here are some examples:

```
[8]: grade =eval(input('Enter your grade:'))
score =eval(input('Enter score:'))
time = eval(input('Enter time:'))

if grade >= 80 and grade < 90:
    print('You are Passed.')
else:
    print("Sorry! Better luck next time")

if score > 1000 or time > 20:
    print('Game over.')
if not (score > 1000 or time > 20):
    print('Game continues.')
```

Enter your grade:85

Enter score:800

Enter time:10

You are Passed.

Game continues.

elif statement: Let's try to understand **elif** statement by an example.

```
[9]: grade =eval(input('Enter your score:'))
if grade>=90:
    print('A')
if grade>=80 and grade<90:
    print('B')
if grade>=70 and grade<80:
    print('C')
if grade>=60 and grade<70:
    print('D')
if grade<60:
    print('F')
```

Enter your score:73

C

The above code can also simplify using **elif** statement.

```
[10]: grade =eval(input('Enter your score:'))
      if grade>=90:
          print('A')
      elif grade>=80:
          print('B')
      elif grade>=70:
          print('C')
      elif grade>=60:
          print('D')
      else:
          print('F')
```

Enter your score:73

C

Using elif, as soon as we find where the score matches, we stop checking conditions and skip all the way to the end of the whole block of statements. An added benefit of this is that the conditions we use in the elif statements are simpler than in their if counterparts. For instance, when using elif, the second part of the second if statement condition, $grade < 90$, becomes unnecessary because the corresponding elif does not have to worry about a score of 90 or above, as such a score would have already been caught by the first if statement.

Homework:

1. Write a program that asks the user how many credits they have taken. If they have taken 23 or less, print that the student is a freshman. If they have taken between 24 and 53, print that they are a sophomore. The range for juniors is 54 to 83, and for seniors it is 84 and over.
2. Write a program that asks the user for an hour between 1 and 12, asks them to enter am or pm, and asks them how many hours into the future they want to go. Print out what the hour will be that many hours into the future, printing am or pm as appropriate.

1.5.6 Flag variables:

A flag variable can be used to let one part of your program know when something happens in another part of the program. Here is an example that determines if a number is prime.

```
[11]: num =eval(input('Enter number:'))
      flag = 0
      for i in range(2,num):
          if num%i==0:
              flag = 1
      if flag==1:
          print('Not prime')
      else:
          print('Prime')
```



```
Enter number:45
Not prime
```

Recall that a number is prime if it has no divisors other than 1 and itself. The way the program above works is flag starts off at 0. We then loop from 2 to num-1. If one of those values turns out to be a divisor, then flag gets set to 1. Once the loop is finished, we check to see if the flag got set or not. If it did, we know there was a divisor, and num isn't prime. Otherwise, the number must be prime.

Homework:

1. Write a program that asks the user to enter a number and prints the sum of the divisors of that number. The sum of the divisors of a number is an important function in number theory.
2. An integer is called squarefree if it is not divisible by any perfect squares other than 1. For instance, 42 is squarefree because its divisors are 1, 2, 3, 6, 7, 21, and 42, and none of those numbers (except 1) is a perfect square. On the other hand, 45 is not squarefree because it is divisible by 9, which is a perfect square. Write a program that asks the user for an integer and tells them if it is squarefree or not.
3. This exercise is about the well-known Monty Hall problem. In the problem, you are a contestant on a game show. The host, Monty Hall, shows you three doors. Behind one of those doors is a prize, and behind the other two doors are goats. You pick a door. Monty Hall, who knows behind which door the prize lies, then opens up one of the doors that doesn't contain the prize. There are now two doors left, and Monty gives you the opportunity to change your choice. Should you keep the same door, change doors, or does it not matter?
 - a) Write a program that simulates playing this game 10000 times and calculates what percentage of the time you would win if you switch and what percentage of the time you would win by not switching.
 - b) Try the above but with four doors instead of three. There is still only one prize, and Monty still opens up one door and then gives you the opportunity to switch.

1.5.7 Strings:

Strings are a data type in Python for dealing with text. Python has a number of powerful features for manipulating strings.

```
[12]: string =input('Enter a string:')
      print(len(string))
```

```
Enter a string:Pythonian
9
```

```
[13]: s = ''
      for i in range(3):
          t =input('Enter a letter:')
```

```

    if t=='a' or t=='e' or t=='i' or t=='o' or t=='u':
        s = s + t
print(s)

```

Enter a letter:3

Enter a letter:r

Enter a letter:y

```

[14]: s =input('Enter a string ')

if s[0].isalpha(): # returns True if every character of the string is a letter
    print('Your string starts with a letter')
if not s.isalpha():
    print('Your string contains a non-letter.')

```

Enter a string Pythonia3

Your string starts with a letter

Your string contains a non-letter.

Excercise: Write a program that, given a string that contains a decimal number, prints out the decimal part of the number. For instance, if given 3.14159, the program should print out .14159.

```

[15]: from math import floor
num =eval(input('Enter number:'))
print(num- floor(num))

```

Enter number:3.14159

0.141589999999999988

Homework:

1. A simple way to estimate the number of words in a string is to count the number of spaces in the string. Write a program that asks the user for a string and returns an estimate of how many words are in the string.
2. Write a program that asks the user to enter their name in lowercase and then capitalizes the first letter of each word of their name.

Hints:

lower() returns a string with every letter of the original in lowercase

upper() returns a string with every letter of the original in uppercase

replace(x,y) returns a string with every occurrence of x replaced by y

count(x) counts the number of occurrences of x in the string

index(x) returns the location of the first occurrence of x

1.5.8 Lists

In list use square brackets to indicate the start and end of the list, and separate the items by commas.

```
[16]: L =eval(input('Enter a list:'))
print('The first element is', L[0])
print(L)
```

```
Enter a list:6,7,8
The first element is 6
(6, 7, 8)
```

Example: Write a program that generates a list L of 50 random numbers between 1 and 100.

```
[17]: from random import randint
L = []
for i in range(5):
    L.append(randint(1,100))
print(L)
```

```
[13, 25, 81, 26, 12]
```

Game for a Quiz:

```
[18]: questions = ['Who had given the concept of 'inertia?','Why is clear sky night_
    ↳cooler than the cloudy sky night? ']
answers = [' Galilee','Radiation']
num_right = 0
for i in range(len(questions)):
    guess =input(questions[i])
    if guess.lower()==answers[i].lower():
        print('Correct')
        num_right=num_right+1
    else:
        print('Wrong. The answer is', answers[i])
print('You have', num_right,'out of', i+1,'right.')
```

```
Who had given the concept of 'inertia?'Galilee
Wrong. The answer is Galilee
You have 0 out of 1 right.
Why is clear sky night cooler than the cloudy sky night? Radiation
Correct
You have 1 out of 2 right.
```

Homework:

1. Write a program that asks the user for an integer and creates a list that consists of the factors of that integer.
2. Write a program that rotates the elements of a list so that the element at the first index moves to the second index, the element in the second index moves to the third index, etc., and the element in the last index moves to the first index.

1.5.9 Functions:

Functions are useful for breaking up a large program to make it easier to read and maintain. They are also useful if you find yourself writing the same code at several different points in your program. You can put that code in a function and call the function whenever you want to execute that code. You can also use functions to create your own utilities, math functions, etc.

```
[19]: def convert(t):
      return t*9/5+32
      print(convert(20))
```

68.0

Example: As another example, the Python math module contains trigonometry functions, but they only work in radians. Let us write our own sine function that works in degrees.

```
[20]: from math import pi, sin
      def deg_sin(x):
          return sin(pi*x/180)
      print("sin(90) =", deg_sin(90))
```

sin(90) = 1.0

Homework:

1. Write a function that takes an integer and returns a random integer with exactly n digits. For instance, if n is 3, then 125 and 593 would be valid return values, but 093 would not because that is really 93, which is a two-digit number.
2. Write a function called `factors` that takes an integer and returns a list of its factors.

2 Python for Mathematical concept

2.1 Fibonacci sequence

A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8.... The first two terms are 0 and 1. All other terms are obtained by adding the preceding two terms. This means to say the n th term is the sum of $(n-1)$ th and $(n-2)$ th term.

Fibonacci numbers are named after the Italian mathematician Leonardo of Pisa, later known as Fibonacci. In his 1202 book Liber Abaci, Fibonacci introduced the sequence to Western European mathematics, although the sequence had been described earlier in Indian mathematics, as early as 200 BC in work by Pingala on enumerating possible patterns of Sanskrit poetry formed from syllables of two lengths.

Fibonacci numbers appear unexpectedly often in mathematics, so much so that there is an entire journal dedicated to their study, the Fibonacci Quarterly. Applications of Fibonacci numbers include computer algorithms such as the Fibonacci search technique and the Fibonacci heap data structure, and graphs called Fibonacci cubes used for interconnecting parallel and distributed systems.

```
[21]: # Program to display the Fibonacci sequence up to n-th term
nterms = int(input("How many terms? "))
from IPython.display import display, Image

# first two terms
n1, n2 = 0, 1
count = 0
# check if the number of terms is valid
if nterms <= 0:
    print("Please enter a positive integer")
elif nterms == 1:
    print("Fibonacci sequence upto",nterms,":")
    print(n1, end = " ")
else:
    #print("Fibonacci sequence:")
    while count < nterms:
        print(n1, end = " ")
        nth = n1 + n2
        # update values
        n1 = n2
        n2 = nth
        count += 1
```

How many terms? 8

0 1 1 2 3 5 8 13

Here, we store the number of terms in nterms. We initialize the first term to 0 and the second term to 1. If the number of terms is more than 2, we use a while loop to find the next term in the sequence by adding the preceding two terms. We then interchange the variables (update it) and continue on with the process.

```
[22]: # Python program to display the Fibonacci sequence using recursion function

def recur_fibo(n):
    if n <= 1:
        return n
    else:
```

```

        return(recur_fibo(n-1) + recur_fibo(n-2))

nterms = 7

# check if the number of terms is valid
if nterms <= 0:
    print("Plese enter a positive integer")
else:
    print("Fibonacci sequence:")
    for i in range(nterms):
        print(recur_fibo(i), end = " ")

```

Fibonacci sequence:
0 1 1 2 3 5 8

In this program, we store the number of terms to be displayed in nterms. A recursive function recur_fibo() is used to calculate the nth term of the sequence. We use a for loop to iterate and calculate each term recursively.

```

[23]: def fib(n):
        """Returns nth Fibonacci numbers """
        a,b=0,1
        for i in range(n):
            a,b=b,a+b
        return a
    if __name__ == "__main__":
        for i in range(7):
            print(fib(i), end = " ")

```

0 1 1 2 3 5 8

2.2 Prime number

Given a positive integer N, The task is to write a Python program to check if the number is prime or not. Definition: A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. The first few prime numbers are {2, 3, 5, 7, 11,}.

There are infinitely many primes, as demonstrated by Euclid around 300 BC. No known simple formula separates prime numbers from composite numbers. However, the distribution of primes within the natural numbers in the large can be statistically modelled. The first result in that direction is the prime number theorem, proven at the end of the 19th century, which says that the probability of a randomly chosen number being prime is inversely proportional to its number of digits, that is, to its logarithm.

```

[24]: # Python program to check if
        # given number is prime or not
        num = 11
        # If given number is greater than 1

```

```

if num > 1:
    for i in range(2, int(num/2)+1):
        if (num % i) == 0:
            print(num, "is not a prime number")
            break
        else:
            print(num, "is a prime number")
else:
    print(num, "is not a prime number")

```

11 is a prime number

[25]: *# Python program to display all the prime numbers within an interval*

```

lower = 0
upper = 20

print("Prime numbers between", lower, "and", upper, "are:")

for num in range(lower, upper + 1):
    # all prime numbers are greater than 1
    if num > 1:
        for i in range(2, num):
            if (num % i) == 0:
                break
        else:
            print(num, end = " ")

```

Prime numbers between 0 and 20 are:

2 3 5 7 11 13 17 19

2.2.1 Use of Numpy

This section is an introduction to numpy. While the core is pretty stable, extensions near the edges are an ongoing process. The definitive documentation is a recent “user guide”, Numpy Community (2013b) at 103 pages and the “reference manual”, Numpy Community (2013a) at 1409 pages. Earlier, more discursive accounts, including a wealth of examples, can be found in Langtangen (2008) and/or Langtangen (2009). Before we start, we must import the numpy module. The preferred approach is to preface the code with:

[26]: `import numpy as np`

Perhaps the most useful constructor is `np.linspace`, which builds an equally spaced array of floats. Also function `np.logspace`, is similar, but the numbers are equally spaced on a logarithmic scale. Somewhat closer to the range function of Python is the function, `np.arange`, which returns an array rather than a list.

```
[27]: a=np.linspace(0,1,5)
      c=np.linspace(1,3,5)
      a+=c
```

```
[28]: a
```

```
[28]: array([1. , 1.75, 2.5 , 3.25, 4.  ])
```

2.3 Creation of Matrix using 'numpy.matrix() function' :

```
[29]: import numpy as p
      matA = p.matrix([[5, 10], [15, 20]])
      print('MatrixA:\n', matA)
      matB = p.matrix(' [5,10;15,20]', dtype=p.int32)
      print('\nMatrixB:\n', matB)
```

```
MatrixA:
[[ 5 10]
 [15 20]]
```

```
MatrixB:
[[ 5 10]
 [15 20]]
```

2.4 Addition of Matrix in Python:

For more understanding to can add two matrix in two way. First I will distuss traditional way to add two matrix using for loop and then I will add two same matrix using '+' operator.

2.4.1 Traditional methods:

```
[30]: import numpy as np
      A = np.matrix([[5, 7], [6, 11]])
      B = np.matrix([[9, 12], [22, 10]])
      result = np.matrix(np.zeros((2,2)))
      print('Matrix A :\n', A)
      print('\nMatrix B :\n', B)

      for x in range(A.shape[1]):
          for y in range(B.shape[0]):
              result[x, y] = A[x, y] + B[x, y]

      print('\nResult :\n', result)
```



```
Matrix A :  
[[ 5  7]  
[ 6 11]]
```

```
Matrix B :  
[[ 9 12]  
[22 10]]
```

```
Result :  
[[14. 19.]  
[28. 21.]]
```

2.4.2 Using '+' operator:

This method provide better efficiency of your code, which is the main motivation of Python language, by decrease line of code to calculate addition of two matrix,

```
[31]: import numpy as np  
A = np.matrix([[5, 7], [6, 11]])  
B = np.matrix([[9, 12], [22, 10]])  
#result = np.matrix(np.zeros((2,2)))  
print('Matrix A :\n', A)  
print('\nMatrix B :\n', B)  
  
result = A+B  
  
print('\nResult :\n', result)
```

```
Matrix A :  
[[ 5  7]  
[ 6 11]]
```

```
Matrix B :  
[[ 9 12]  
[22 10]]
```

```
Result :  
[[14 19]  
[28 21]]
```

2.5 Matrix multiplication in Python

There are two kind of matrix multiplication, scaler and matrix multiplication.

2.5.1 Scalar multiplication:

In the scalar product, a scalar/constant value is multiplied by each element of the matrix.

```
[32]: import numpy as np
A = np.matrix([[11, 22], [33, 44]])
print("Matrix A:\n", A)
print("\n")
print("Scalar Product of Matrix A:\n", A * 10)
```

Matrix A:

```
[[11 22]
 [33 44]]
```

Scalar Product of Matrix A:

```
[[110 220]
 [330 440]]
```

2.5.2 Matrix multiplication:

```
[33]: import numpy as np
A = np.matrix([[5, 7], [6, 11]])
B = np.matrix([[9, 12], [22, 10]])
#result = np.matrix(np.zeros((2,2)))
print('Matrix A :\n', A)
print('\nMatrix B :\n', B)

print("\nDot Product of Matrix A and Matrix B:\n", np.dot(A,B))

print("\nMatrix multiplication using numpy.matmul() method")
res = np.matmul(A,B)
print(res)
```

Matrix A :

```
[[ 5  7]
 [ 6 11]]
```

Matrix B :

```
[[ 9 12]
 [22 10]]
```

Dot Product of Matrix A and Matrix B:

```
[[199 130]
 [296 182]]
```

Matrix multiplication using numpy.matmul() method

```
[[199 130]
 [296 182]]
```

2.6 Subtraction of matrix using Python

```
[34]: import numpy as np
A = np.matrix([[5, 7], [6, 11]])
B = np.matrix([[9, 12], [22, 10]])
#result = np.matrix(np.zeros((2,2)))
print('Matrix A :\n', A)
print('\nMatrix B :\n', B)

print("\nSubtraction of Matrix A and Matrix B:\n", (A - B))
```

```
Matrix A :
[[ 5  7]
 [ 6 11]]
```

```
Matrix B :
[[ 9 12]
 [22 10]]
```

```
Subtraction of Matrix A and Matrix B:
[[ -4  -5]
 [-16   1]]
```

2.7 Transpose of a Python Matrix

Transpose of a matrix basically involves the flipping of matrix over the corresponding diagonals i.e. it exchanges the rows and the columns of the input matrix. The rows become the columns and vice-versa.

```
[35]: import numpy
A = numpy.array([numpy.arange(10,15), numpy.arange(15,20)])
print("Original Matrix A:\n")
print(A)
print('\nDimensions of the original MatrixA: ',A.shape)
print("\nTranspose of Matrix A:\n ")
result = A.T
print(result)
print('\nDimensions of the Matrix A after performing the Transpose Operation:  ↳
↳ ',result.shape)
```

```
Original Matrix A:
```

```
[[10 11 12 13 14]
```

```
[15 16 17 18 19]]
```

Dimensions of the original MatrixA: (2, 5)

Transpose of Matrix A:

```
[[10 15]
 [11 16]
 [12 17]
 [13 18]
 [14 19]]
```

Dimensions of the Matrix A after performing the Transpose Operation: (5, 2)

2.8 Exponent of a Python Matrix

The exponent on a Matrix is calculated element-wise i.e. exponent of every element is calculated by raising the element to the power of an input scalar/constant value.

```
[36]: import numpy
A = numpy.array([numpy.arange(0,2), numpy.arange(2,4)])
print("Original Matrix A:\n")
print(A)
print("\nExponent of the input matrix:\n")
print(A ** 2)
```

Original Matrix A:

```
[[0 1]
 [2 3]]
```

Exponent of the input matrix:

```
[[0 1]
 [4 9]]
```

2.9 Problem :

You are given a system of linear equations as follows, and need to find the values of w,x,y,z:

$$w + 3x - 5y + 2z = 0$$

$$4x - 2y + z = 6$$

$$2w - x + 3y - z = 5$$

$$w + x + y + z = 10$$

2.10 Solution:

The system of equation can be written as a matrix multiplication:

$$\begin{bmatrix} 1 & 3 & -5 & 2 \\ 0 & 4 & -2 & 1 \\ 2 & -1 & 3 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 6 \\ 5 \\ 10 \end{bmatrix}$$

or

$$Mx = b$$

```
[37]: import numpy as np
M = np.matrix('1 3 -5 2; 0 4 -2 1; 2 -1 3 -1; 1 1 1 1')
b = np.matrix('0; 6; 5; 10')
x = np.linalg.inv(M) * b
print(x)
```

```
[[1.]
 [2.]
 [3.]
 [4.]]
```

2.11 Problem

Python Matrix Multiplication, Inverse Matrix, Matrix Transpose of two matrix. You can chose your own matrix. Write your answer which can be readable.

2.12 Solution

```
[38]: import numpy as np

# initialize a 3x2 matrix of random values
matA = np.matrix(np.random.rand(3, 2))
# print the first matrix
print('The first matrix is :\n', matA)

# initialize a 2x3 matrix of random values
matB = np.matrix(np.random.rand(2, 3))
# print the second matrix
print('\nThe second matrix is :\n', matB)

# multiply two matrix using * operator
result = matA * matB
# print the resultant matrix
print('\nMatrix multiplication result :\n', result)
```

```
# get the inverse of the first matrix
inverseMatA = matA.getI()
print('\nThe inverse of the first matrix is :\n', inverseMatA)

# get the transpose matrix of the second matrix
transposeMatB = matB.getT()
print('\nThe transpose of the second matrix is :\n', transposeMatB)
```

The first matrix is :

```
[[0.66521422 0.19364653]
 [0.37885605 0.72947794]
 [0.36674489 0.06013573]]
```

The second matrix is :

```
[[0.67522629 0.81792215 0.55733008]
 [0.58658754 0.16439356 0.04910689]]
```

Matrix multiplication result :

```
[[0.56276077 0.57592769 0.38025327]
 [0.68371624 0.42979623 0.24697027]
 [0.28291066 0.3098547 0.20735104]]
```

The inverse of the first matrix is :

```
[[ 1.295336   -0.4098538   0.80055558]
 [-0.62759384  1.57798131 -0.4917408  ]]
```

The transpose of the second matrix is :

```
[[0.67522629 0.58658754]
 [0.81792215 0.16439356]
 [0.55733008 0.04910689]]
```

2.13 Graphical plotting and histogram

2.13.1 Making graphs

At first sight, it will seem that there are quite some components to consider when you start plotting with this Python data visualization library. You'll probably agree with me that it's confusing and sometimes even discouraging seeing the amount of code that is necessary for some plots, not knowing where to start yourself and which components you should use.

Luckily, this library is very flexible and has a lot of handy, built-in defaults that will help you out tremendously. As such, you don't need much to get started: you need to make the necessary imports, prepare some data, and you can start plotting with the help of the `plot()` function! When you're ready, don't forget to show your plot using the `show()` function.

```
[39]: # Import the necessary packages and modules
import matplotlib.pyplot as plt
```

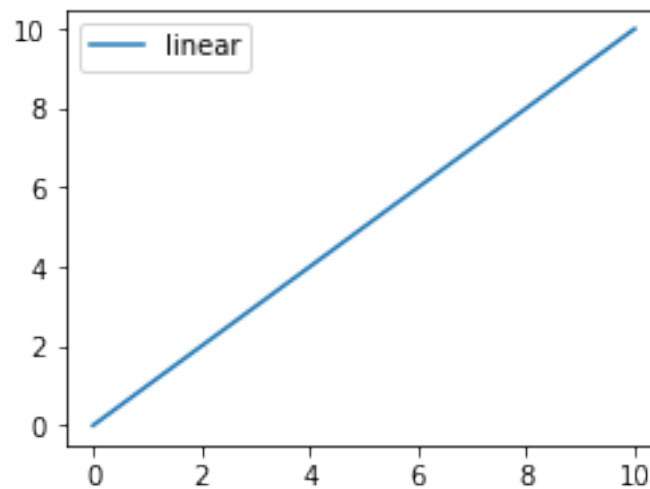
```
import numpy as np
plt.figure(figsize=(4, 3))

# Prepare the data
x = np.linspace(0, 10, 100)

# Plot the data
plt.plot(x, x, label='linear')

# Add a legend
plt.legend()

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



First off, you'll already know Matplotlib by now. When you talk about “Matplotlib”, you talk about the whole Python data visualization package. This should not come to you as a big surprise :)

Secondly, pyplot is a module in the matplotlib package. That's why you often see `matplotlib.pyplot` in code. The module provides an interface that allows you to implicitly and automatically create figures and axes to achieve the desired plot.

This is especially handy when you want to quickly plot something without instantiating any Figures or Axes, as you saw in the example in the first section of this tutorial. You see, you haven't explicitly specified these components, yet you manage to output a plot that you have even customized! The defaults are initialized and any customizations that you do, will be done with the current Figure and Axes in mind.

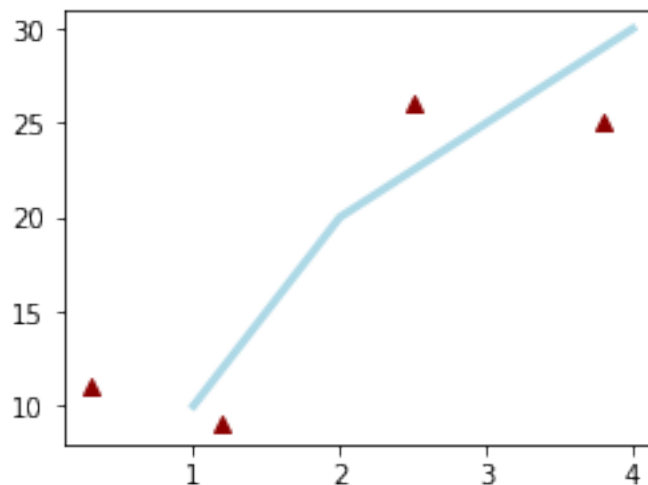
Lastly, pylab is another module, but it gets installed alongside the matplotlib package. It bulk imports pyplot and the numpy library and was generally recommended when you were working

with arrays, doing mathematics interactively and wanted access to plotting features.

You might still see this popping up in older tutorials and examples of matplotlib, but its use is no longer recommended, especially not when you're using the IPython kernel in your Jupyter notebook. You can read more about this [here](#).

As a solution, you can best use `%matplotlib` magic in combination with the right backend, such as `inline`, `qt`, etc. Most of the times, you will want to use `inline`, as this will make sure that the plots are embedded inside the notebook. Read more about that in DataCamp's [Definitive Guide to Jupyter Notebook](#).

```
[40]: import matplotlib.pyplot as plt
plt.figure(figsize=(4, 3))
plt.plot([1, 2, 3, 4], [10, 20, 25, 30], color='lightblue', linewidth=3)
plt.scatter([0.3, 3.8, 1.2, 2.5], [11, 25, 9, 26], color='darkred', marker='^')
#plt.set_xlim(0.5, 4.5)
plt.show()
```



You see that the `add_subplot()` function in itself also poses you with a challenge, because you see `add_subplots(111)` in the above code chunk.

What does 111 mean?

Well, 111 is equal to 1,1,1, which means that you actually give three arguments to `add_subplot()`. The three arguments designate the number of rows (1), the number of columns (1) and the plot number (1). So you actually make one subplot.

Note that you can really go bananas with this function when you are using this function, especially when you're just starting out with this library and you keep on forgetting for what the three numbers stand.

Consider the following commands and try to envision what the plot will look like and how many Axes your Figure will have: `ax = fig.add_subplot(2,2,1)`.

2.13.2 Working With Pyplot: Plotting Routines

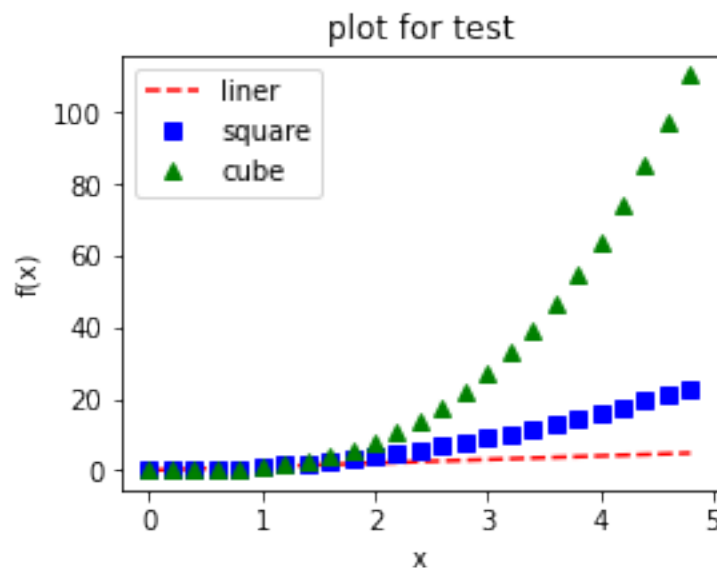
Now that all is set for you to start plotting your data, it's time to take a closer look at some plotting routines. You'll often come across functions like `plot()` and `scatter()`, which either draw points with lines or markers connecting them, or draw unconnected points, which are scaled or colored.

But, as you have already seen in the example of the first section, you shouldn't forget to pass the data that you want these functions to use!

These functions are only the bare basics. You will need some other functions to make sure your plots look awesome:

```
[41]: import numpy as np
import matplotlib.pyplot as plt
plt.figure(figsize=(4, 3))

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)
# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--',label='liner')
plt.plot(t, t**2, 'bs',label='square')
plt.plot(t, t**3, 'g^',label='cube')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('plot for test')
#plt.yscale('log')
plt.legend()
plt.show()
```



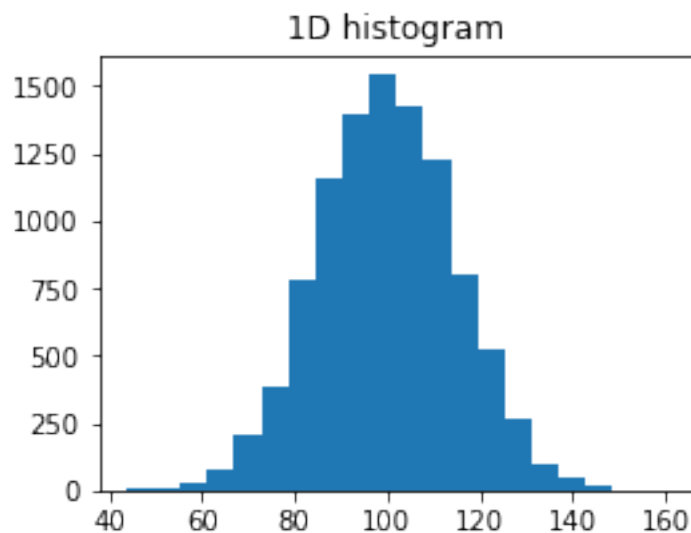
This is a perfect example of standard publishable plot. You can label axis, name of plot and legend also.

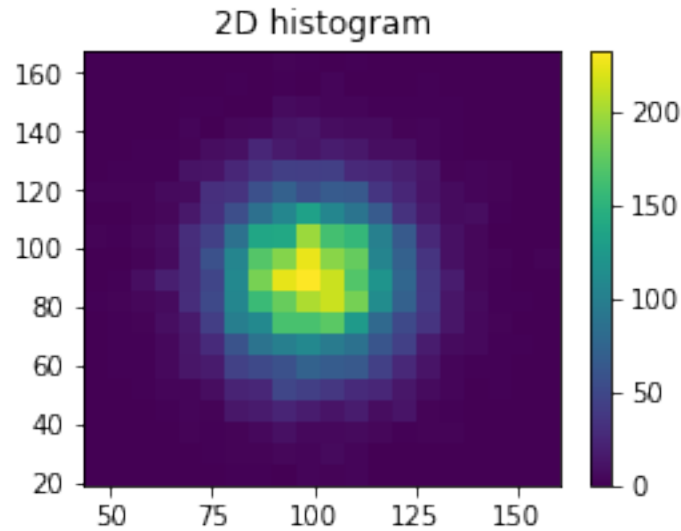
2.14 Histogram:

Now I will show you how to create 1D and 2D histograms and how to save them as a publishable histogram.

```
[42]: import numpy as np
import matplotlib.pyplot as plt
plt.figure(figsize=(4, 3))

# Fixing random state for reproducibility
np.random.seed(19680801)
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)
nu, gamma = 90, 20
y = nu + gamma * np.random.randn(10000)
plt.hist(x, bins=20)
plt.title('1D histogram')
plt.show()
plt.figure(figsize=(4, 3))
plt.hist2d(x, y, bins=20)
plt.title('2D histogram')
plt.colorbar()
plt.show()
```





Now we will write all possible settings for 2d histograms:

2.15 Problem 1 :

You can try to solve this first yourself.

1) Plot both of the following functions on a single figure, with a usefully sized scale:

a)

$$f_1(x) = x^4 e^{-2x}$$

b)

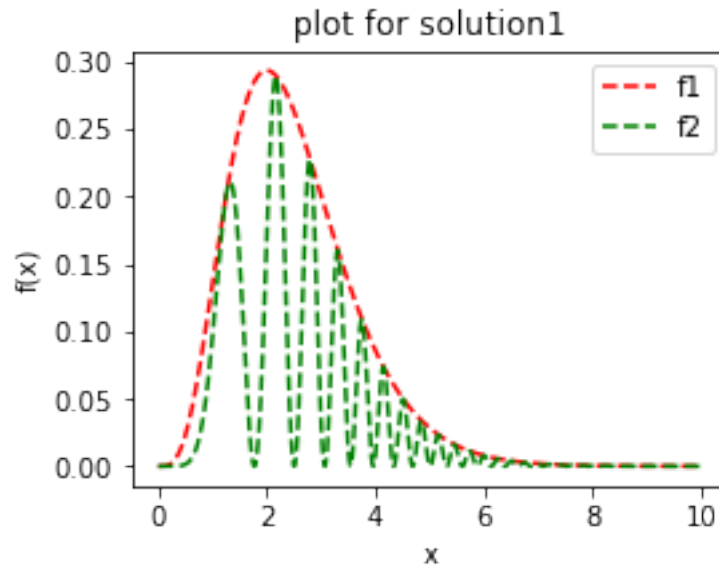
$$f_2(x) = [x^2 e^{-x} \sin(x^2)]^2$$

Make sure your figure has legend, range, title, axis labels and publishable.

2.16 Solution 1:

```
[43]: import numpy as np
import matplotlib.pyplot as plt
import math
x = np.arange(0., 10., 0.01)
b= np.exp(-2*x)
plt.figure(figsize=(4, 3))
plt.plot(x,x**4*np.exp(-2*x), 'r--',label='f1')
plt.plot(x,(x**2*np.exp(-x)*np.sin(x**2))**2,'g--',label='f2')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('plot for solution1')
```

```
#plt.yscale('log')
plt.legend()
plt.show()
```



3 Mechanical oscillator

The case of the one dimensional mechanical oscillator leads to the following equation:

$$m\ddot{x} + \mu\dot{x} + kx = m\ddot{x}_d$$

Where:

- x is the position,
- \dot{x} and \ddot{x} are respectively the speed and acceleration,
- m is the mass,
- μ the
- k the stiffness,
- and \ddot{x}_d the driving acceleration which is null if the oscillator is free.

3.1 Canonical equation

Most 1D oscillators follow the same canonical equation:

$$\ddot{x} + 2\zeta\omega_0\dot{x} + \omega_0^2x = \ddot{x}_d$$

Where:

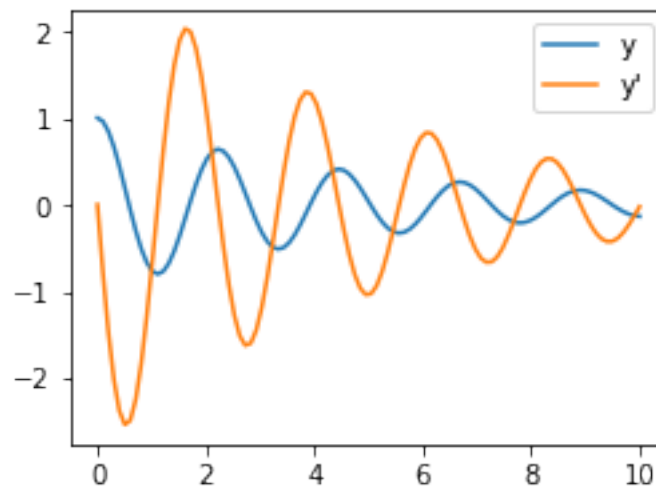
- ω_0 is the undamped pulsation,
- ζ is damping ratio,
- $\ddot{x}_d = a_d \sin(\omega_d t)$ is the imposed acceleration.

In the case of the mechanical oscillator:

$$\omega_0 = \sqrt{\frac{k}{m}}$$

$$\zeta = \frac{\mu}{2\sqrt{mk}}$$

```
[44]: import numpy as np
from scipy.integrate import odeint
from matplotlib import pyplot as plt
mass = 0.5 # kg
kspring = 4 # N/m
cviscous = 0.4 # N s/m
eps = cviscous / (2 * mass * np.sqrt(kspring/mass))
omega = np.sqrt(kspring / mass)
def calc_der(yvec, time, eps, omega):
    return (yvec[1], -eps * omega * yvec[1] - omega **2 * yvec[0])
time_vec = np.linspace(0, 10, 100)
yinit = (1, 0)
yarr = odeint(calc_der, yinit, time_vec, args=(eps, omega))
plt.figure(figsize=(4, 3))
plt.plot(time_vec, yarr[:, 0], label='y')
plt.plot(time_vec, yarr[:, 1], label="y'")
plt.legend(loc='best')
plt.show()
```



3.2 Damped harmonic oscillator:

```
[45]: from matplotlib import pyplot as plt
from math import sin, fabs, pi
plt.figure(figsize=(4, 3))

#The restoring force due to the spring
def f(k,x):
    restoring_force = -k*x
    return restoring_force
#The drag force. b is the drag coefficient and v is the speed
def d(b,v):
    drag_force = -b*v
    return drag_force
#The driving force. This is a function of time. f is the frequency of the
→driving force and a is its maximum value
def driving(a, f, t):
    driving_force = a*sin(2*pi*f*t)
    return driving_force
#Takes a three item list, moves the elements down one place and adds a new
→element into position [2]
def cycle(my_list, new):
    my_list[0] = my_list[1]
    my_list[1] = my_list[2]
    my_list[2] = new
    return my_list
def simulate_oscillations(driving_force_amplitude, driving_frequency,
→drag_coefficient, k, m, dt, option): #This does all the work.

    x = 0 # Initial displacement
    t,v = 0,0 #Initial time and speed

    x_list = [0,0,0] #This list keeps track of the three previous x
→values. If the number in the middle is the largest then
                                #we have found a local maximum (i.e. the
→amplitude).
    displacement = []
    time = [] #
                                #These are lists that will hold the data used
→to produce a pretty graph at the end
    amplitude_list = [] #
    time1 = []
```

```

flag = 0

while flag == 0:    #This loop keeps going until we are satisfied that the
    →amplitude has reached a stable, maximum value

    r = f(k,x) + d(drag_coefficient,v) + driving(driving_force_amplitude,
    →driving_frequency, t) #Calculates the resultant force on the mass
    a = r / m                #Calculates the acceleration of the mass
    v = v + (a*dt)           #Updates the speed
    x = x + (v*dt)           #Updates the position
    x_list = cycle(x_list, x) #Updates x_list with the latest x value

    if (x_list[1]) > (x_list[0]) and (x_list[1]) > (x_list[2]): #Checks to
    →see if x_list[1] is larger than x_list[0] and x_list[2]
        amplitude_list.append(x_list[1])                #If it is,
    →we add this to our list of amplitudes
        time1.append(t-dt)                                #Records
    →the time at which the mass reached the final amplitude

    l = len(amplitude_list)
    if l > 3: #Wait until we have 3 amplitudes to compare
        if fabs(amplitude_list[l-1] - amplitude_list[l-2]) < 1e-5 and
    →fabs(amplitude_list[l-2] - amplitude_list[l-3]) < 1e-5: #If the amplitude is
    →changing by less than 0.0001 each cycle then it is constant
            #print("The amplitude of this system is ", amplitude_list[l-1],
    →'when the driving frequency is ', driving_frequency) #Outputs the final
    →amplitude of the system. This line can be removed for long runs
            flag = 1 #Breaks out of the loop

    time.append(t)                #This data can be used to plot a pretty graph
    →of displacement
    displacement.append(x)        #against time to check individual simulations. 
    →not used in final version

    t = t + dt                    #Advances the time

    if option == 'show_graph':
        plt.plot(time1, amplitude_list)
        plt.plot(time, displacement)
        plt.suptitle('A forced, damped oscillator.')
        plt.xlabel('time')
        plt.ylabel('displacement')
        plt.grid('on')
        plt.show()

```

```

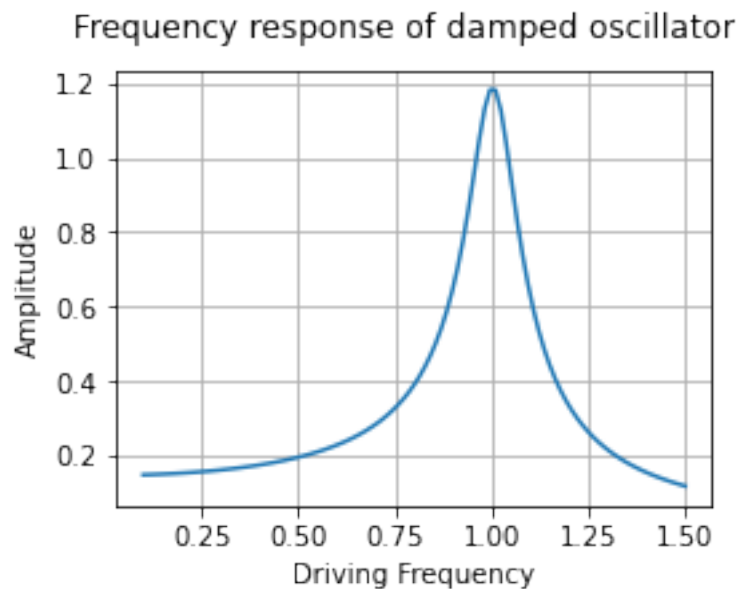
    return amplitude_list[l-1]

def run_the_experiment(a,b,c):
    step_size = (b-a)/c
    run = 0
    f_list = []
    results = []
    while run <=c:
        results.append(simulate_oscillations(3, a, 0.4, 20, 0.5, 0.001,
↪'no_graph')) #(driving_force_amplitude, driving_frequency,
↪drag_coefficient, k, m, dt, option)
        f_list.append(a)
        a = a + step_size
        run = run + 1

    plt.plot(f_list, results)
    plt.suptitle('Frequency response of damped oscillator')
    plt.xlabel('Driving Frequency')
    plt.ylabel('Amplitude')
    plt.grid('on')
    plt.draw
    plt.show()

run_the_experiment(0.1, 1.5, 100) #run_the_experiment(begin frequency range, end
↪of frequency range, number of points between a and b)

```



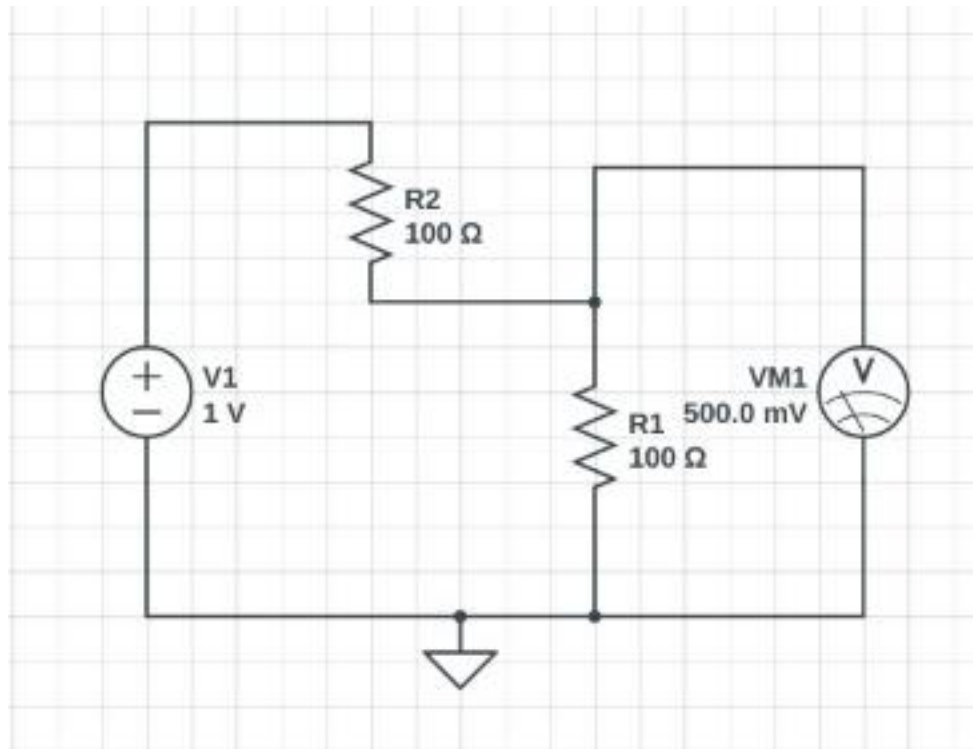
4 Application of Python for electrical circuit

This part I will explain how to solve electrical circuits using the **sympy** symbolic math module.

4.1 Example DC circuit

The following circuit includes one voltage source, one current source and two resistors.

The objective is to obtain the output voltage **V_o** as function of the components.



The circuit will be solved using the **nodal** method.

First we need to locate the circuit **nodes**, assign one as **ground** and assign a number for the rest of them. As the circuit has three nodes and one is ground, we have two nodes left: 1 and 2.

We will first generate a set of sympy symbols. There will be:

- One symbol for each component: V1, R1, R2, Is
- One current symbol for each power supply: iVs
- One symbol for each measurement we want to obtain: V_o
- One symbol for each node voltage that is not ground: V1, V2

```
[46]: # Import the sympy module
import sympy
# Create the circuit symbols
```

```
Vs,iVs,R1,R2,Is,Vo, V1, V2 = sympy.symbols('Vs,iVs,R1,R2,Is,Vo,V1,V2')
```

Then we can define the current equations on each node except ground.

The current equations add all the currents in the node from each component.

All equations we add, are supposed to have a result of **zero**.

```
[47]: # Create an empty list of equations
equations = []

# Nodal equations
equations.append(iVs-(V1-V2)/R1)      # Node 1
equations.append(Is-(V2-V1)/R1-V2/R2) # Node 2
```

```
[48]: # Voltage source equations
equations.append(sympy.Eq(Vs,V1))
# Measurement equations
equations.append(sympy.Eq(Vo,V2))
```

Now we can define the unknowns for the circuit.

The number of unknowns shall be equal to the number of equations.

The list includes:

- The node voltages: V1, V2
- The current on voltage sources: iVs
- The measurement values: Vo

```
[49]: unknowns = [V1,V2,iVs,Vo]
```

We can see the equations and unknowns before solving the circuit.

To ease reusing the code, we will define a **showCircuit** function that shows equations and unknowns

```
[50]: # Define the function
def showCircuit():
    print('Equations')
    for eq in equations:
        print('    ',eq)
    print()
    print('Unknowns:',unknowns)
    print()

# Use the function
showCircuit()
```

Equations

$$iVs - (V1 - V2)/R1$$

```

Is - V2/R2 - (-V1 + V2)/R1
Eq(Vs, V1)
Eq(Vo, V2)

```

Unknowns: [V1, V2, iVs, Vo]

```

[51]: # Solve the circuit
      solution = sympy.solve(equations,unknowns)

      # List the solutions
      print('Solutions')
      for sol in solution:
          print(' ',sol,'=',solution[sol])

```

Solutions

```

V1 = Vs
V2 = R2*(Is*R1 + Vs)/(R1 + R2)
iVs = (-Is*R2 + Vs)/(R1 + R2)
Vo = R2*(Is*R1 + Vs)/(R1 + R2)

```

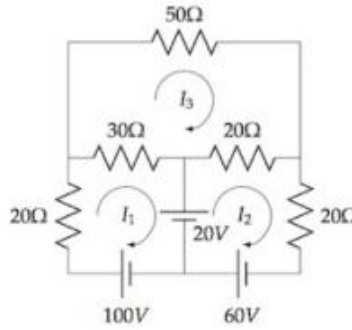
```

[52]: solution = sympy.solve(Is-(Vo-Vs)/R1-Vo/R2,Vo)
      print('Vo =',solution[0])

```

```
Vo = R2*(Is*R1 + Vs)/(R1 + R2)
```

Another problem to calculate current in electric circuit using Kirchoff's law.



For each closed loop, we can apply Kirchoff's Voltage Law, $\sum_k V_k = 0$, in conjunction with Ohm's Law, $V = IR$, to give three simultaneous equations:

$$\begin{aligned} 50I_1 - 30I_3 &= 80, \\ 40I_2 - 20I_3 &= 80, \\ -30I_1 - 20I_2 + 100I_3 &= 0. \end{aligned}$$

These can be expressed in matrix form as $\mathbf{RI} = \mathbf{V}$:

$$\begin{pmatrix} 50 & 0 & -30 \\ 0 & 40 & -20 \\ -30 & -20 & 100 \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} 80 \\ 80 \\ 0 \end{pmatrix},$$

We could use the numerically stable `np.linalg.solve` method (Section 6.5.3) to find the loop currents, \mathbf{I} here, but in this well-behaved system, let's find them by left multiplication by the matrix inverse, \mathbf{R}^{-1} :

$$\mathbf{R}^{-1}\mathbf{RI} = \mathbf{I} = \mathbf{R}^{-1}\mathbf{V}.$$

```
[53]: R = np.matrix('50 0 -30; 0 40 -20; -30 -20 100')
      V = np.matrix('80; 80; 0')
      I = np.linalg.inv(R) * V
      print(I)
```

```
[[2.33333333]
 [2.61111111]
 [1.22222222]]
```

$$I_1 = 2.33A, I_2 = 2.611A, I_3 = 1.222A$$

LINPACK can also provide eigenvalues and eigenvectors of matrices as well, using `linalg.eig()`. It should be noted that the size of the matrix that LINPACK can handle is limited by memory available on your computer.

5 Reading data from files

When we performed data analysis from any experimental data, we used to store required information in some text file. Python can also read data from text files quite well. I will show a

demonstration how to read text files using `loadtxt()` function.

In this chapter I will introduce you to the task of text analysis in Python. You will learn how to read an entire corpus into Python, clean it and how to perform certain data analyses on those texts. We will also briefly introduce you to using Python’s plotting library *matplotlib*, with which you can visualize your data.

Before we delve into the main subject of this chapter, text analysis, we will first write a couple of utility functions that build upon the things you learnt in the previous chapter. Often we don’t work with a single text file stored at our computer, but with multiple text files or entire corpora. We would like to have a way to load a corpus into Python.

Remember how to read files? Each time we had to open a file, read the contents and then close the file. Since this is a series of steps we will often need to do, we can write a single function that does all that for us. We write a small utility function `read_file(filename)` that reads the specified file and simply returns all contents as a single string.

```
[54]: def read_file(filename):  
    "Read the contents of FILENAME and return as a string."  
    infile = open(filename) # windows users should use codecs.open after_  
    ↪importing codecs  
    contents = infile.read()  
    infile.close()  
    return contents
```

```
[55]: text = read_file("data.txt")  
print(text)
```

#Frequency	Mic1	Mic2
10.0	0.654	0.192
11.0	0.127	0.032
12.0	0.120	0.030
13.0	0.146	0.031
14.0	0.155	0.033
15.0	0.175	0.036

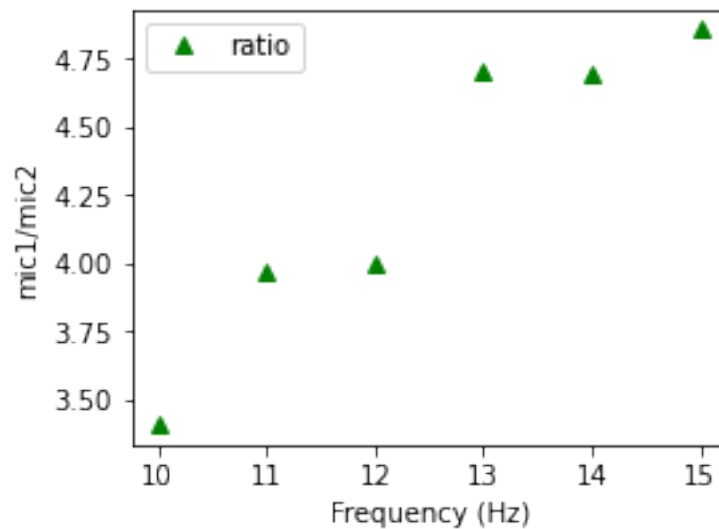
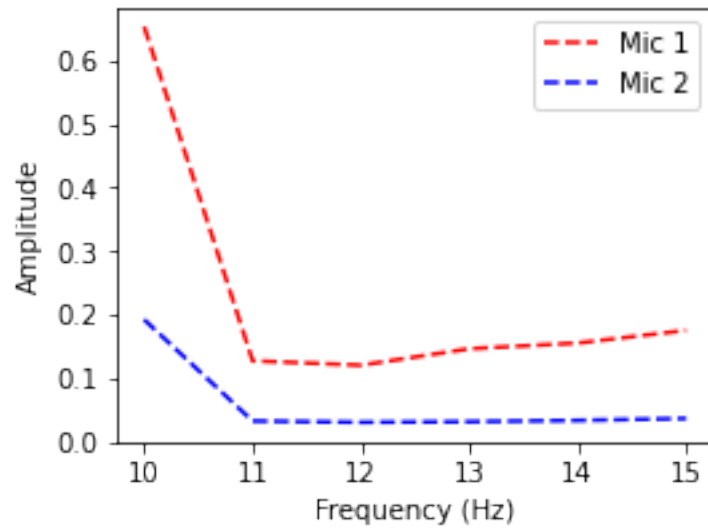
The `loadtxt()` function takes one required argument: the file name (you can also write file name with actual path on your computer). There are a number of optioanal arguments: one we’re going to use here is “unpack”, which tells `loadtxt()` that the file contains columns of data that should be returned in separate arrays. In this case, we have told Python to call those arrays “frequency”, “mic1”, “mic2”. The `loadtxt()` function is very handy and resonably intelligent.

```
[56]: import numpy as np  
import matplotlib.pyplot as plt  
plt.figure(figsize=(4, 3))  
  
#data = np.loadtxt("./weight_height_1.txt")  
frequency,mic1,mic2 = np.loadtxt("data.txt",unpack=True)  
plt.plot(frequency,mic1,'r--',frequency,mic2,'b--')
```

```
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude")
plt.legend(["Mic 1", "Mic 2"])
plt.show()

plt.figure(figsize=(4, 3))

plt.plot(frequency, mic1/mic2, 'g^')
plt.xlabel("Frequency (Hz)")
plt.ylabel("mic1/mic2")
plt.legend(["ratio"])
plt.show()
```



Here is an example how you can search text file in your computer directory. Also you can write names of text files in a certain directory of your computer.

```
[57]: from os import listdir
def list_textfiles(directory):
    "Return a list of filenames ending in '.txt' in DIRECTORY."
    textfiles = []
    for filename in listdir(directory):
        if filename.endswith(".txt"):
            textfiles.append(directory + "/" + filename)
    return textfiles
```

The function `listdir` takes as argument the name of a directory and lists all filenames in that directory. I iterate over this list and append each filename that ends with the extension, `.txt` to a new list of `textfiles`. Using the `list_textfiles` function, the following code will read all text files in the directory `/home/haradhan` and outputs the length (in characters) of each:

```
[58]: for filepath in list_textfiles("/home/haradhan"):
    text = read_file(filepath)
    print(filepath + " has " + str(len(text)) + " characters.")
```

```
/home/haradhan/data.txt has 188 characters.
/home/haradhan/Problem2.txt has 0 characters.
/home/haradhan/Output.txt has 21 characters.
```

Let's see, how can you read text file from some website or from some particular locaion.

```
[59]: import pandas as pd
import matplotlib.pyplot as plt

# stock ticker symbol
url = 'https://apmonitor.com/che263/uploads/Main/goog.csv'

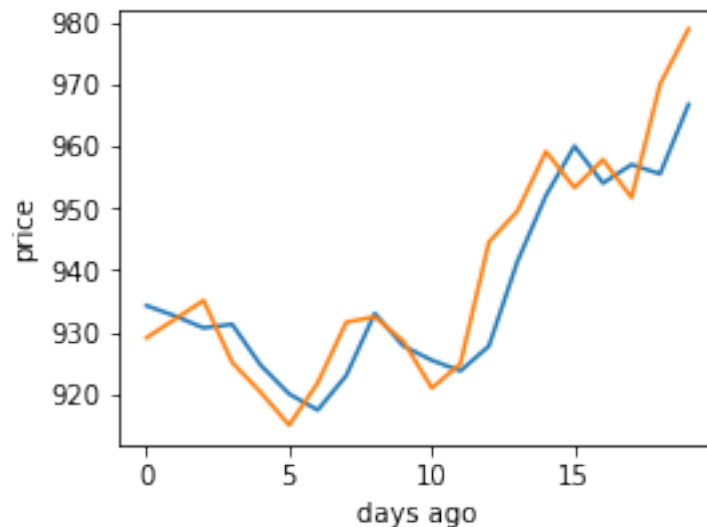
# import data with pandas
data = pd.read_csv(url)
print(data['Close'][0:5])
print('min: '+str(min(data['Close'][0:20])))
print('max: '+str(max(data['Close'][0:20])))

# plot data with pyplot
plt.figure(figsize=(4, 3))
plt.plot(data['Open'][0:20])
plt.plot(data['Close'][0:20])
plt.xlabel('days ago')
plt.ylabel('price')
plt.show()
```

```

0    929.080017
1    932.070007
2    935.090027
3    925.109985
4    920.289978
Name: Close, dtype: float64
min: 915.0
max: 978.8900150000001

```



5.1 Data files:

In computational physics, the inputs and outputs of any experimental result are large set of data. Rather than re-enter these large sets each time we run the program, we load and save the data in the form of text files.

When working with files, we start by opening the file. The open function tell the operating system what file we will be working on, and what we want to do with the file.

```
FileHandle = open("FileName",Mode)
```

FileName should be a string describing the location and name of the file. The mood can be one of these:

- 1) "r": read mode only, you can't change it only you can read.
- 2) "w": write mode will create the file if it doesn't exist. If the file already exist, opening it using "w" means it will re-write it and will destroy the current file
- 3) "a": append mode allows you to write onto the end of a previously-existing file without destroying what was already exist.

5.2 Problem 2:

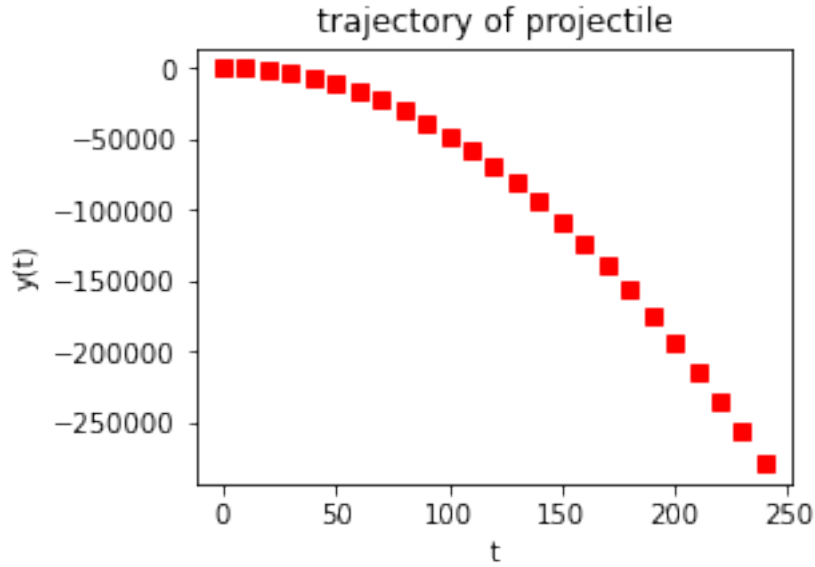
For the model used in introductory physics courses, a projectile thrown vertically at some initial velocity v_i has position $y(t) = y_i + v_i t - \frac{1}{2}gt^2$, where $g = 9.8 \text{ m/s}^2$. Write a Python program that creates two lists, one containing time data (50 datas over 5 seconds) and the other containing the corresponding vertical position data for this projectile. The program should ask the user for the initial height and initial velocity v_i , and should print a nicely formatted table of the list values after it has calculated them.

```
[60]: import numpy as np
import matplotlib.pyplot as plt
import math
t = np.arange(0.,250., 10)
yi = int(input("What is the initial height? "))
vi = int(input("What is the initial velocity? "))
g=9.8
#for t in range
yt= yi+vi*t-0.5*g*t**2
file1 = open("Problem2.txt","w")
L = ["t  y(t) \n"]
Q = [t,yt]
#file1.write(yt)
#file1.writelines(Q)
file1.close() #to change file access modes
plt.figure(figsize=(4, 3))

plt.plot(t,yt,'rs')
plt.xlabel("t")
plt.ylabel("y(t)")
plt.title("trajectory of projectile")
plt.show()
```

What is the initial height? 13

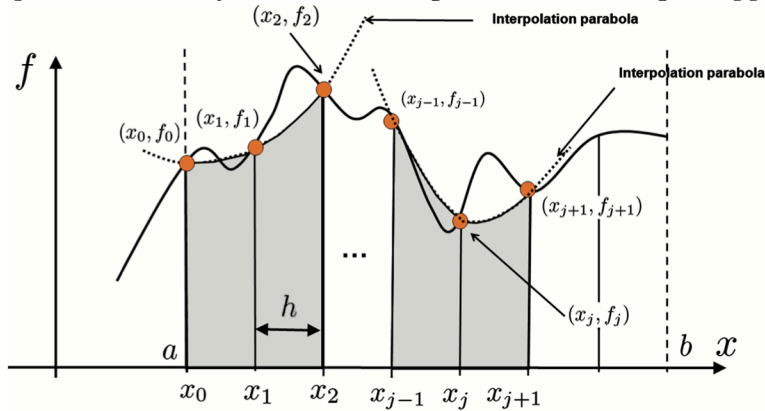
What is the initial velocity? 10



6 Python numerical integration

6.1 Simpson $\frac{1}{3}$ rule

Consider two consecutive subintervals, $[-1, 1]$ and $[1, 2]$. Simpson's Rule approximates the area under $f(x)$ over these two subintervals by fitting a quadratic polynomial through the points $(-1, f(-1))$, $(1, f(1))$, and $(2, f(2))$, which is a unique polynomial, and then integrating the quadratic exactly. The following shows this integral approximation for an arbitrary function.



Simpson $\frac{1}{3}$ formula:

$$\int_a^b f(x)dx \approx \frac{h}{3} [f(x_0) + 4 \left(\sum_{i=1, i: \text{odd}}^{n-1} f(x_i) \right) + 2 \left(\sum_{i=2, i: \text{even}}^{n-2} f(x_i) \right) + f(x_n)]$$

6.2 Problem 3:

Use Simpson's Rule to approximate $\int_0^\pi \sin()$ with 11 evenly spaced grid points over the whole interval. Compare this value to the exact value of 2

6.3 Solution 3:

```
[61]: import numpy as np

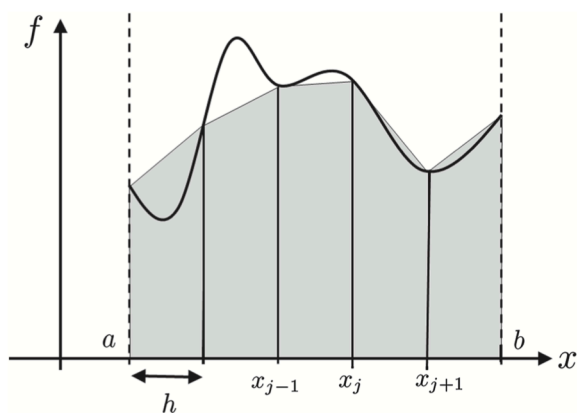
a = 0
b = np.pi
n = 11
h = (b - a) / (n - 1)
x = np.linspace(a, b, n)
f = np.sin(x)

I_simp = (h/3) * (f[0] + 4*sum(f[1:n-1:2]) + 2*sum(f[2:n-2:2]) + f[n-1])
err_simp = 2 - I_simp

print("Integral by Simpson 1/3 rule = ", I_simp)
print("Error in simpson = ", err_simp)
```

```
Integral by Simpson 1/3 rule = 2.0001095173150043
Error in simpson = -0.00010951731500430384
```

6.4 Trapezoid Rule:



The Trapezoid Rule fits a trapezoid into each subinterval and sums the areas of the trapezoid to approximate the total integral. This approximation for the integral to an arbitrary function is shown in the following figure. For each subinterval, the Trapezoid Rule computes the area of a trapezoid with corners at $(x_{j-1}, f(x_{j-1}))$, $(x_j, f(x_j))$, $(x_{j+1}, f(x_{j+1}))$, and $(x_{j-1}, f(x_{j-1}))$, which is $h \frac{f(x_{j-1}) + f(x_{j+1}))}{2}$. Thus, the Trapezoid Rule approximates integrals according to the expression:

$$\int_a^b f(x) dx \approx \sum_{i=1}^{n-1} h \frac{f(x_i) + f(x_{i+1}))}{2}$$

6.5 Problem 4:

Use Trapezoidal Rule to approximate $\int_0^\pi \sin(x) dx$ with 11 evenly spaced grid points over the whole interval. Compare this value to the exact value of 2. Also calculate the difference between Simpson 1/3 rule and Trapezoidal rule calculation.

6.6 Solution 4:

```
[62]: import numpy as np

a = 0
b = np.pi
n = 11
h = (b - a) / (n - 1)
x = np.linspace(a, b, n)
f = np.sin(x)

I_trap = (h/2)*(f[0] + \
                2 * sum(f[1:n-1]) + f[n-1])
err_trap = 2 - I_trap

print("Integral by Trapezoid's rule = ", I_trap)
print("Error in Trapezoid = ", err_trap)
print("Difference Simpson and Trapezoidal result = ", I_simp - I_trap)
```

```
Integral by Trapezoid's rule =  1.9835235375094546
Error in Trapezoid =  0.01647646249054535
Difference Simpson and Trapezoidal result =  0.016585979805549655
```

6.7 Riemanns Integral:

The simplest method for approximating integrals is by summing the area of rectangles that are defined for each subinterval. The width of the rectangle is $\Delta x = \frac{b-a}{n}$, and the height is defined by a function value $f(x_i)$ for some x_i in the subinterval. An obvious choice for the height is the function value at the left endpoint, $f(x_{i-1})$, or the right endpoint, $f(x_i)$, because these values can be used even if the function itself is not known. This method gives the Riemann Integral approximation, which is:

$$\int_a^b f(x) dx \approx \sum_{i=1}^{n-1} h f(x_{i-1})$$

or

$$\int_a^b f(x) dx \approx \sum_{i=1}^n h f(x_i)$$

depending on whether the left or right endpoint is chosen.

6.8 Problem 5:

Use Riemanns integral to approximate $\int_0^\pi \sin()$ with 11 evenly spaced grid points over the whole interval. Compare this value to the exact value of 2.

```
[63]: import numpy as np

a = 0
b = np.pi
n = 11
h = (b - a) / (n - 1)
x = np.linspace(a, b, n)
f = np.sin(x)

I_riemannL = h * sum(f[1:n-1])
err_riemannL = 2 - I_riemannL

I_riemannR = h * sum(f[1:n])
err_riemannR = 2 - I_riemannR

I_mid = h * sum(np.sin((x[:n-1] \
    + x[1:])/2))
err_mid = 2 - I_mid

print("Riemann left = ",I_riemannL)
print("Riemann left error = ",err_riemannL)

print("Riemann right = ",I_riemannR)
print("Riemann right error = ",err_riemannR)

print("Riemann middle = ",I_mid)
print("Riemann middle error = ",err_mid)
```

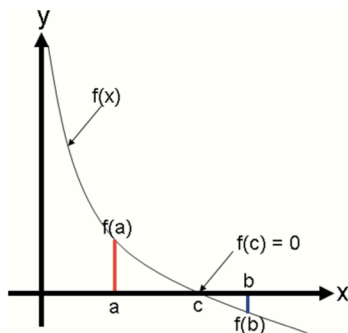
```
Riemann left = 1.9835235375094546
Riemann left error = 0.01647646249054535
Riemann right = 1.9835235375094546
Riemann right error = 0.01647646249054535
Riemann middle = 2.0082484079079745
Riemann middle error = -0.008248407907974542
```

7 Root finding in Python

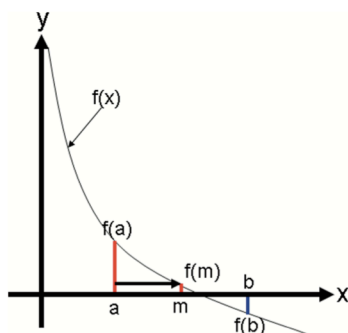
7.1 Bisection method:

The simplest root finding algorithm is the bisection method. The algorithm applies to any continuous function $f(x)$ on an interval $[a, b]$ where the value of the function changes sign from a to b .

The idea is simple: divide the interval in two, a solution must exist within one subinterval, select the subinterval where the sign of changes and repeat.



The bisection method uses the intermediate value theorem iteratively to find roots. Let $f(x)$ be a continuous function, and a and b be real scalar values such that $a < b$. Assume, without loss of generality, that $f(a) > 0$ and $f(b) < 0$. Then by the intermediate value theorem, there must be a root on the open interval (a, b) . Now let $m = \frac{a+b}{2}$, the midpoint between a and b . If $f(m) = 0$ or is close enough, then m is a root. If $f(m) > 0$, then m is an improvement on the left bound, a , and there is guaranteed to be a root on the open interval (m, b) . If $f(m) < 0$, then m is an improvement on the right bound, b , and there is guaranteed to be a root on the open interval (a, m) .



The process of updating a and b can be repeated until the error is acceptably low.

7.2 Problem 6.a:

Program a function `my_bisection(f, a, b, tol)` that approximates a root of f , bounded by a and b to within $|\frac{b-a}{2}| < tol$.

7.3 Solution 6.a:

```
[64]: import numpy as np

def my_bisection(f, a, b, tol):
    if np.sign(f(a)) == np.sign(f(b)):
        raise Exception("The scalars a and b do not bound a root")
```

```

# get midpoint
m = (a + b)/2

if np.abs(f(m)) < tol:
    # stopping condition, report m as root
    return m
elif np.sign(f(a)) == np.sign(f(m)):
    # case where m is an improvement on a.
    # Make recursive call with a = m
    return my_bisection(f, m, b, tol)
elif np.sign(f(b)) == np.sign(f(m)):
    # case where m is an improvement on b.
    # Make recursive call with b = m
    return my_bisection(f, a, m, tol)

```

7.4 Problem 6.b:

The $\sqrt{2}$ can be computed as the root of the function $f(x) = x^2 - 2$. Starting at $a=0$ and $b=2$, use `my_bisection` to approximate the $\sqrt{2}$ to a tolerance of $|f(a)| < 0.1$ and $|f(b)| < 0.01$. Verify that the results are close to a root by plugging the root back into the function. Plot the function in the range $(0,2)$.

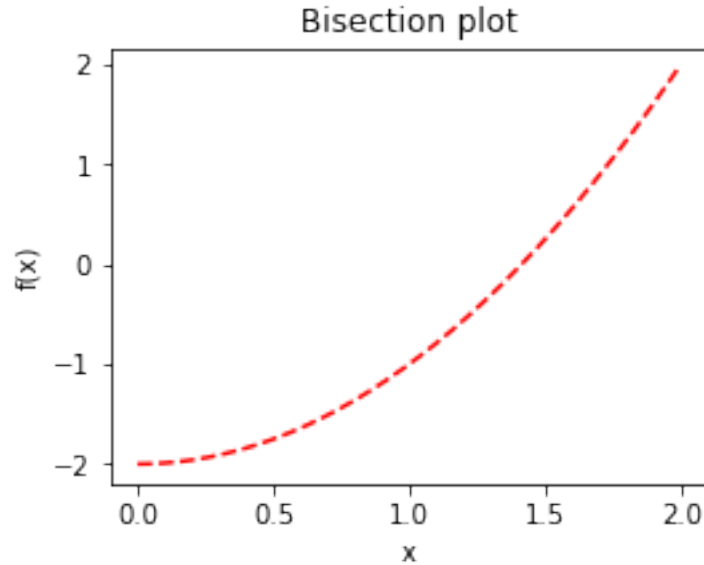
7.5 Solution 6.b:

```

[65]: import numpy as np
import matplotlib.pyplot as plt
import math
x = np.arange(0.0,2.,0.01)
fx = x**2 - 2
plt.figure(figsize=(4, 3))

plt.plot(x,fx,'r--')
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Bisection plot")
plt.show()

```



```
[46]: f = lambda x: x**2 - 2
      #total are here 0.1 and 0.01, a=0,b=2
      r1 = my_bisection(f, 0, 2, 0.1)
      print("r1 =", r1)
      r01 = my_bisection(f, 0, 2, 0.01)
      print("r01 =", r01)

      print("f(r1) =", f(r1))
      print("f(r01) =", f(r01))
```

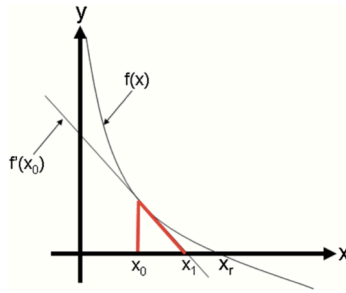
```
r1 = 1.4375
r01 = 1.4140625
f(r1) = 0.06640625
f(r01) = -0.00042724609375
```

7.6 Newton-Raphson Method

Let $f(x)$ be a smooth and continuous function and x_r be an unknown root of $f(x)$. Now assume that x_0 is a guess for x_r . Unless x_0 is a very lucky guess, $f(x_0)$ will not be a root. Given this scenario, we want to find an x_1 that is an improvement on x_0 (i.e., closer to x_r than x_0). If we assume that x_0 is “close enough” to x_r , then we can improve upon it by taking the linear approximation of $f(x)$ around x_0 , which is a line, and finding the intersection of this line with the x-axis. Written out, the linear approximation of $f(x)$ around x_0 is $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$. Using this approximation, we find x_1 such that $f(x_1) = 0$.

$$0 = f(x_0) + f'(x_0)(x_1 - x_0)$$

which when solve for x_1 is, $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$



7.7 Problem 7.a:

The $\sqrt{2}$ can be computed as the root of the function $f(x) = x^2 - 2$. Using $x_0 = 1.4$ as a starting point, use the previous equation to estimate $\sqrt{2}$. Compare this approximation with the value computed by Python's sqrt function.

```
[66]: import numpy as np

f = lambda x: x**2 - 2
f_prime = lambda x: 2*x
newton_raphson = 1.4 - (f(1.4))/(f_prime(1.4))

print("newton_raphson =", newton_raphson)
print("sqrt(2) =", np.sqrt(2))
```

```
newton_raphson = 1.4142857142857144
sqrt(2) = 1.4142135623730951
```

```
[67]: def my_newton(f, df, x0, tol):
    if abs(f(x0)) < tol:
        return x0
    else:
        return my_newton(f, df, x0 - f(x0)/df(x0), tol)
estimate = my_newton(f, f_prime, 1.5, 1e-6)
print("estimate =", estimate)
print("sqrt(2) =", np.sqrt(2))
```

```
estimate = 1.4142135623746899
sqrt(2) = 1.4142135623730951
```

Python has the existing root-finding functions for us to use to make things easy. The function we will use to find the root is `f_solve` from the `scipy.optimize`.

7.8 Problem 7.b:

Compute the root of the function $f(x) = x^3 - 100x^2 - x + 100$ using `f_solve`.

```
[68]: from scipy.optimize import fsolve
f = lambda x: x**3-100*x**2-x+100
print(fsolve(f, [2, 80]))
```

```
[ 1. 100.]
```

7.9 Secant Method

The secant method is a modification of Newton Raphson's method which has the advantage of not needing the derivative of the function.

Start with two guesses a and b , these should be near the desired solution, as with Newton's method, but they don't have to bracket the solution like they do with the bisection method. Use the value of $f(a)$ and $f(b)$ to approximate the slope of the curve, instead of using the function $f'(x)$ to find the slope exactly.

Algorithm for the secant method:

1. Start
2. Define function as $f(x)$
3. Input initial guesses (x_0 and x_1), tolerable error (e) and maximum iteration (N)
4. Initialize iteration counter $i = 1$
5. If $f(x_0) = f(x_1)$ then print "Mathematical Error" and goto (11) otherwise goto (6)
6. Calculate $x_2 = x_1 - (x_1 - x_0) * \frac{f(x_1)}{f(x_1) - f(x_0)}$
7. Increment iteration counter $i = i + 1$
8. If $i \geq N$ then print "Not Convergent" and goto (11) otherwise goto (9)
9. If $|f(x_2)| > e$ then set $x_0 = x_1$, $x_1 = x_2$ and goto (5) otherwise goto (10)
10. Print root as x_2
11. Stop

```
[69]: def secant(f,a,b,N):

    if f(a)*f(b) >= 0:
        print("Secant method fails.")
        return None
    a_n = a
    b_n = b
    for n in range(1,N+1):
        m_n = a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))
        f_m_n = f(m_n)
        if f(a_n)*f_m_n < 0:
            a_n = a_n
            b_n = m_n
```

```

elif f(b_n)*f_m_n < 0:
    a_n = m_n
    b_n = b_n
elif f_m_n == 0:
    print("Found exact solution.")
    return m_n
else:
    print("Secant method fails.")
    return None
return a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))

```

7.10 Problem 8:

Find the real root of the polynomial: $p(x) = x^3 - x^2 - 1$ using secant method.

7.11 Solution 8:

```

[70]: from scipy.optimize import fsolve
p = lambda x: x**3 - x**2 - 1
print(fsolve(p, [1, 80]))

```

```
[1.46557123 1.46557123]
```

Since the polynomial changes sign in the interval $[1,2]$, we can apply the secant method with this as the starting interval:

```

[71]: approx = secant(p,1,2,20)
print("real root by secant method = ",approx)

```

```
real root by secant method = 1.4655712311394433
```

7.11.1 To do by yourself:

The Fermi-Dirac distribution describes the probability of finding a quantum particle with half-integer spin ($\frac{1}{2}, \frac{3}{2}, \dots$) in energy state E :

$$f_{FD} = \frac{1}{e^{\frac{E-\mu}{kT}} + 1}$$

The μ in the Fermi-Dirac distribution is called the Fermi Energy, and in this case we want to adjust μ so that the probability of finding the particle somewhere is exactly one.

$$\int_{E_{min}}^{E_{max}} f_{FD} dE = 1$$

Imagine a room-temperature quantum system where for some reason the energy E is constrained to be in between 0 and 2 eV. What is the μ in this case? At room temperature, $kT \approx \frac{1}{40}$ eV. Feel free to use any of the above integration or root finding method.

7.11.2 Note:

One can use “Scipy” stands for “Scientific Python” and it is a package which provides numerous scientific tools. Like one can use : from scipy import integrate

```
[72]: # import scipy.integrate.quad
from scipy import integrate
x = np.arange(0, 10)
y = np.arange(0, 10)
y = lambda x: x**2
dy = lambda x: 2*x
# using scipy.integrate.quad() method
I1 = integrate.quad(y, 0, 3)
print(I1)
```

(9.0000000000000002, 9.992007221626411e-14)

8 Special function using Python

SciPy provides a plethora of special functions, including Bessel functions (and routines for finding their zeros, derivatives, and integrals), error functions, the gamma function, Legendre, Laguerre, and Hermite polynomials (and other polynomial functions), Mathieu functions, many statistical functions, and a number of other functions. Most are contained in the `scipy.special` library, and each has its own special arguments and syntax, depending on the vagaries of the particular function. We demonstrate a number of them in the code below that produces a plot of the different functions called. For more information, you should consult the SciPy web site on the `scipy.special` library.

8.1 Bessel function:

Bessel functions, first defined by the mathematician Daniel Bernoulli and then generalized by Friedrich Bessel, are canonical solutions $y(x)$ of Bessel's differential equation:

$$x^2 \frac{d^2 y(x)}{dx^2} + x \frac{dy(x)}{dx} + (x^2 - \alpha^2) y(x) = 0$$

8.1.1 Applications of Bessel functions:

- Electromagnetic waves in a cylindrical waveguide
- Pressure amplitudes of inviscid rotational flows
- Heat conduction in a cylindrical object
- Modes of vibration of a thin circular (or annular) acoustic membrane
- Diffusion problems on a lattice

f. Solutions to the radial Schrödinger equation (in spherical and cylindrical coordinates) for

8.1.2 Bessel function of 1st kind:

Bessel functions of the first kind, denoted as $J_\alpha(x)$, are solutions of Bessel's differential equation. For integer or positive α , Bessel functions of the first kind are finite at the origin ($x = 0$); while for negative non-integer α , Bessel functions of the first kind diverge as x approaches zero. It is possible to define the function by its series expansion around $x = 0$, which can be found by applying the Frobenius method to Bessel's equation:

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m! \Gamma(m + \alpha + 1)} \left(\frac{x}{2}\right)^{2m+\alpha}$$

For more details read Mathematical Methods for Physicists Arfken.

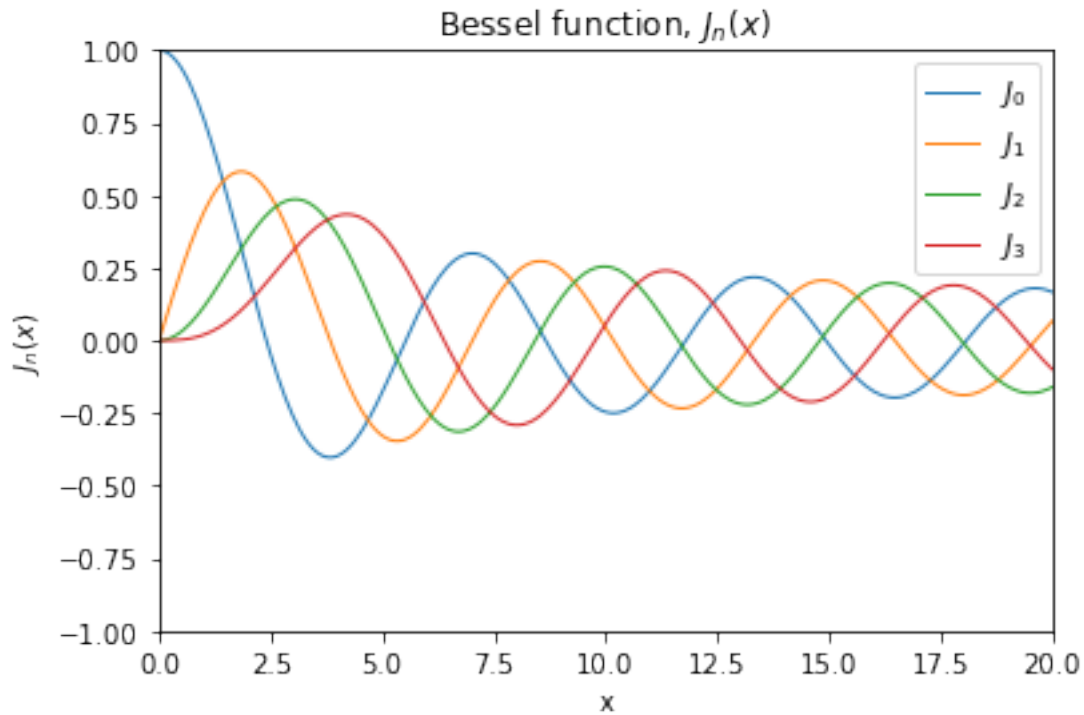
```
[73]: import numpy as np
      from scipy import special
      import matplotlib.pyplot as plt

      # create a figure window
      #fig = plt.figure(1, figsize=(10,12))
      x = np.linspace(0, 20, 256)

      plt.figure()
      plt.plot(x, special.jn(0, x), linewidth=1, label=r"$J_0$")
      plt.plot(x, special.jn(1, x), linewidth=1, label=r"$J_1$")
      plt.plot(x, special.jn(2, x), linewidth=1, label=r"$J_2$")
      plt.plot(x, special.jn(3, x), linewidth=1, label=r"$J_3$")

      #Set limits for axes
      plt.xlim([0,20])
      plt.ylim([-1,1])

      #Set axes labels
      plt.xlabel("x")
      plt.ylabel(r"$J_n(x)$")
      plt.title(r"Bessel function, $J_n(x)$")
      plt.legend()
      plt.show();
```

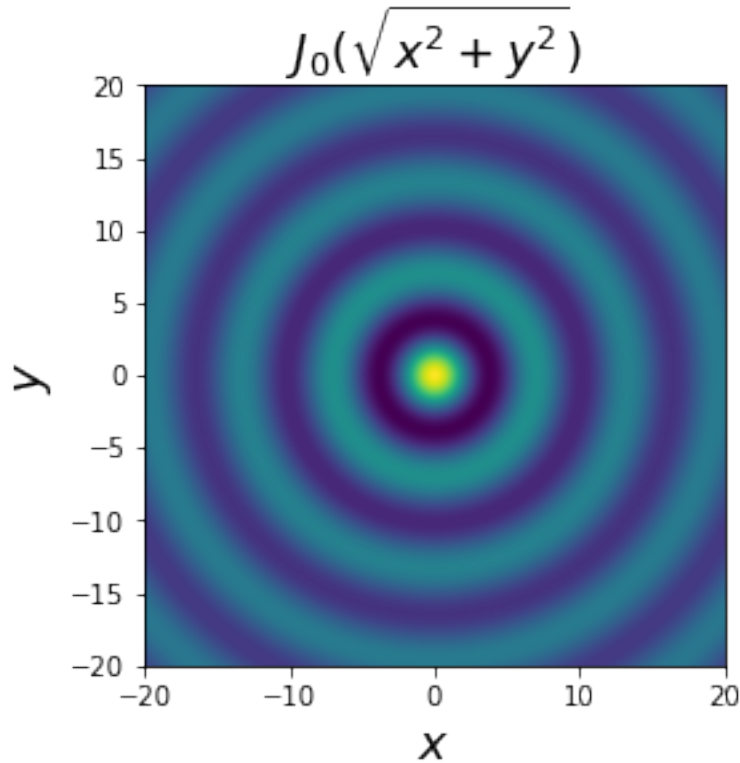


8.2 2d plot of Bessel function:

```
[74]: import numpy as np
from scipy import special
import matplotlib.pyplot as plt
# Suppose we have a function of two variables x and y (in this case a
# ↪ cylindrical wave)
def f(x,y):
    r=np.sqrt(x**2 + y**2)
    return special.j0(r)
xp=np.linspace(-20,20,500)
yp=np.linspace(-20,20,500)
X,Y = np.meshgrid(xp, yp)
Z = f(X, Y)
plt.imshow(Z,origin='lower',extent=(-20,20,-20,20))

plt.xlabel("$x$",fontsize=18)
plt.ylabel("$y$",fontsize=18)

plt.title("$J_0(\sqrt{x^2 + y^2})$",fontsize=18);
```



8.3 Gamma function:

In mathematics, the gamma function Γ is one commonly used extension of the factorial function to complex numbers. The gamma function is defined for all complex numbers except the non-positive integers. For any positive integer n ,

$$\Gamma(n) = (n - 1)!$$

Derived by Daniel Bernoulli, for complex numbers with a positive real part, the gamma function is defined via a convergent improper integral:

$$\Gamma(z) = \int_0^{\infty} x^{z-1} e^{-x} dx$$

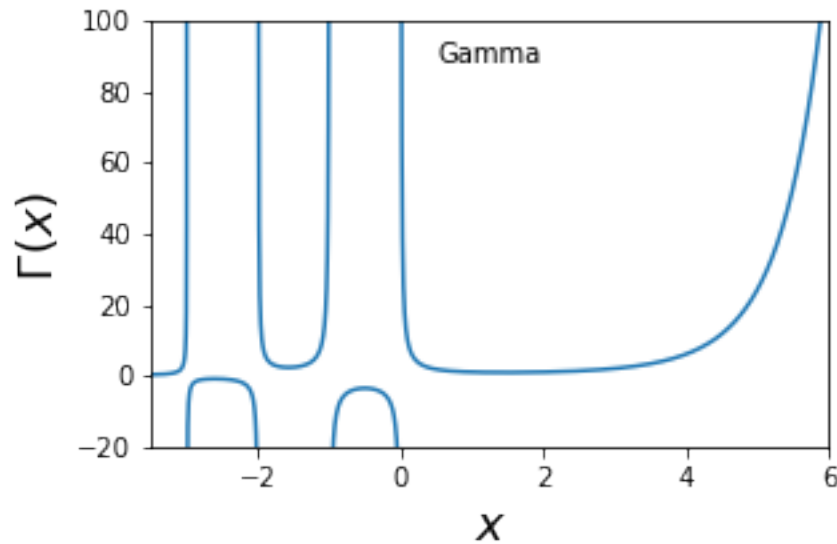
The gamma function then is defined as the analytic continuation of this integral function to a meromorphic function that is holomorphic in the whole complex plane except zero and the negative integers, where the function has simple poles.

```
[75]: fig = plt.figure(1, figsize=(10,10))
      x = np.linspace(-3.5, 6., 3601)
```

```

g = special.gamma(x)
g = np.ma.masked_outside(g, -100, 400)
fig = plt.figure(1, figsize=(10,12))
ax2 = fig.add_subplot(322)
ax2.plot(x,g)
ax2.set_xlim(-3.5, 6)
ax2.set_ylim(-20, 100)
ax2.text(0.5, 0.95,"Gamma", ha="center", va="top",transform = ax2.transAxes);
plt.xlabel("$x$",fontsize=18)
plt.ylabel("$\Gamma(x)$",fontsize=18);

```



8.4 Error function:

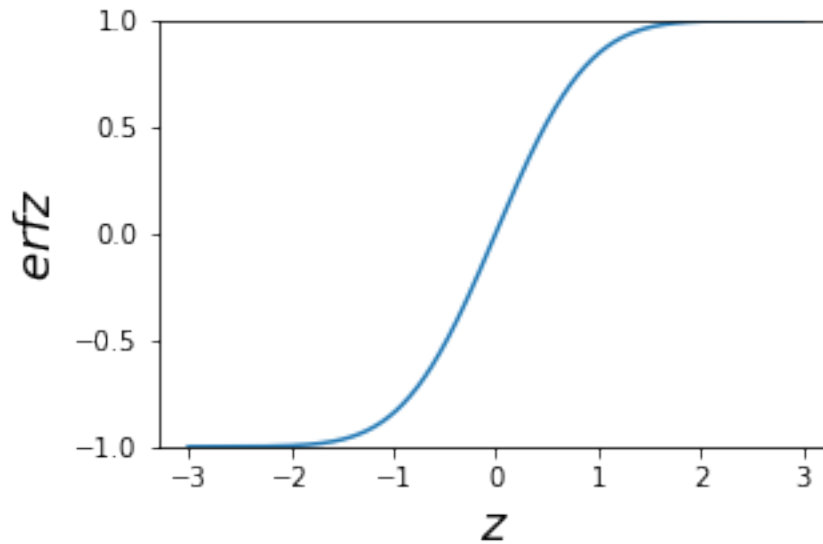
In mathematics, the error function (also called the Gauss error function), often denoted by erf , is a complex function of a complex variable defined as:

$$\text{erf}z = \frac{2}{\sqrt{\pi}} \int_0^{\infty} e^{-t^2} dt$$

This integral is a special (non-elementary) sigmoid function that occurs often in probability, statistics, and partial differential equations. In many of these applications, the function argument is a real number. If the function argument is real, then the function value is also real.

In statistics, for non-negative values of x , the error function has the following interpretation: for a random variable Y that is normally distributed with mean 0 and variance 1/2, $\text{erf } x$ is the probability that Y falls in the range $(-x, x)$


```
[76]: fig = plt.figure(1, figsize=(10,10))
z = np.linspace(-3, 3, 500)
ef = special.erf(z)
fig = plt.figure(1, figsize=(10,12))
ax3 = fig.add_subplot(322)
ax3.plot(z,ef)
ax3.set_ylim(-1,1)
ax3.text(0.5, 0.95,"Error", ha="center", va="top",transform = ax2.transAxes);
plt.xlabel("$z$",fontsize=18)
plt.ylabel("$\text{erf } z$",fontsize=18);
```



8.5 Airy function:

In the physical sciences, the Airy function (or Airy function of the first kind) $\text{Ai}(x)$ is a special function named after the British astronomer George Biddell Airy (1801–1892). The function $\text{Ai}(x)$ and the related function $\text{Bi}(x)$, are linearly independent solutions to the differential equation:

$$\frac{d^2 y(x)}{dx^2} - xy = 0$$

known as the Airy equation or the Stokes equation. This is the simplest second-order linear differential equation with a turning point (a point where the character of the solutions changes from oscillatory to exponential).

The Airy function is the solution to time-independent Schrödinger equation for a particle confined within a triangular potential well and for a particle in a one-dimensional constant force field. For the same reason, it also serves to provide uniform semiclassical approximations near a turning point in the WKB approximation, when the potential may be locally approximated by a linear function

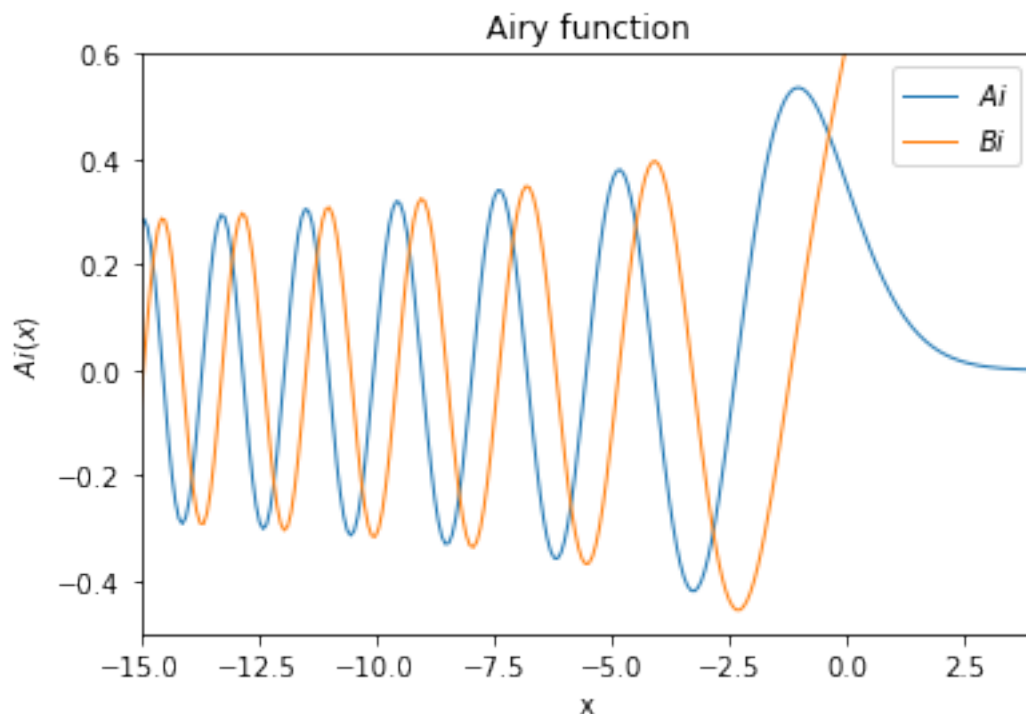
of position. The triangular potential well solution is directly relevant for the understanding of electrons trapped in semiconductor heterojunctions.

```
[77]: #fig = plt.figure(1, figsize=(10,10))
x = np.linspace(-15, 4, 256)
ai, aip, bi, bip = special.airy(x)
#fig = plt.figure(1, figsize=(10,12))

plt.figure()
plt.plot(x, ai, linewidth=1, label=r"$Ai$")
plt.plot(x, bi, linewidth=1, label=r"$Bi$")

#Set limits for axes
plt.xlim([-15,4])
plt.ylim([-0.5,0.6])

#Set axes labels
plt.xlabel("x")
plt.ylabel(r"$Ai(x)$")
plt.title(r"Airy function")
plt.legend();
plt.show();
```



8.6 Legendre polynomials:

In physical science and mathematics, Legendre polynomials (named after Adrien-Marie Legendre, who discovered them in 1782) are a system of complete and orthogonal polynomials, with a vast number of mathematical properties, and numerous applications. They can be defined in many ways, and the various definitions highlight different aspects as well as suggest generalizations and connections to different mathematical structures and physical and numerical applications.

Legendre's differential equation:

$$\frac{d}{dx}[(1-x^2)\frac{dP_n(x)}{dx}] + n(n+1)P_n(x) = 0$$

Solution of Legendre differential equation can be written using Rodrigues formula:

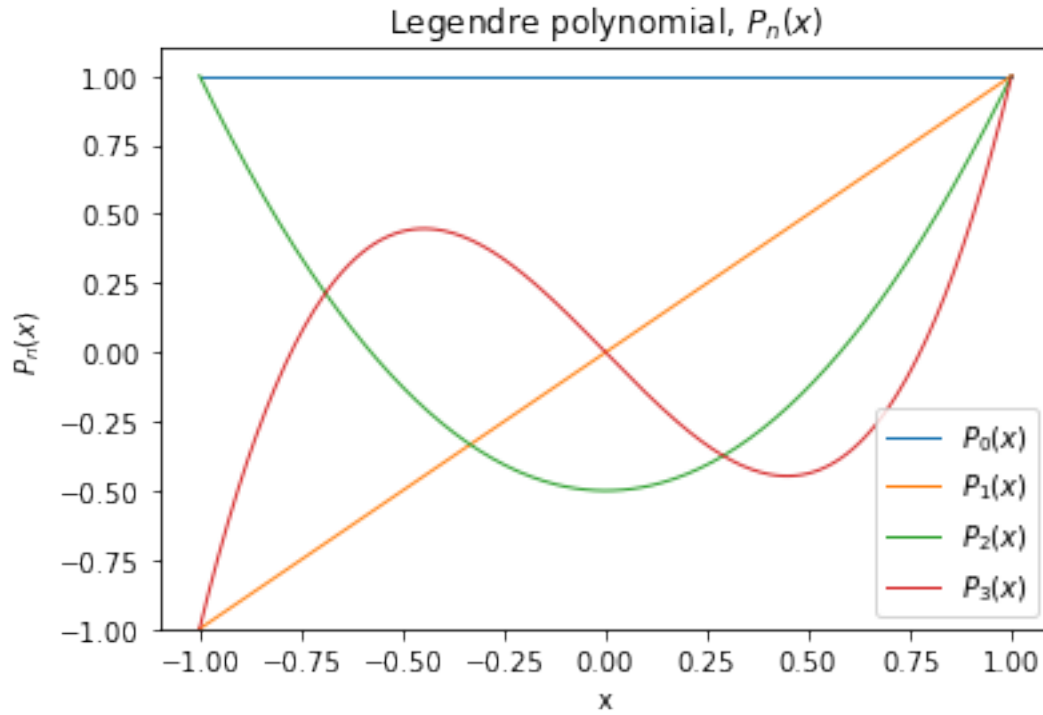
$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n$$

```
[78]: #fig = plt.figure(1, figsize=(10,10))
x = np.linspace(-1, 1, 256)

plt.figure()
plt.plot(x, np.polyval(special.legendre(0),x), linewidth=1,label=r"$P_0(x)$")
plt.plot(x, np.polyval(special.legendre(1),x), linewidth=1,label=r"$P_1(x)$")
plt.plot(x, np.polyval(special.legendre(2),x), linewidth=1,label=r"$P_2(x)$")
plt.plot(x, np.polyval(special.legendre(3),x), linewidth=1,label=r"$P_3(x)$")

#Set limits for axes
#plt.xlim([0,20])
plt.ylim([-1,1.1])

#Set axes labels
plt.xlabel("x")
plt.ylabel(r"$P_n(x)$")
plt.title(r"Legendre polynomial, $P_n(x)$")
plt.legend()
plt.show();
```



8.7 Laguerre polynomials:

In mathematics, the Laguerre polynomials, named after Edmond Laguerre (1834–1886), are solutions of Laguerre's equation:

$$x \frac{d^2 y(x)}{dx^2} + (1-x) \frac{dy(x)}{dx} + ny = 0$$

Laguerre polynomials can be defined by the Rodrigues formula,

$$L_n(x) = \frac{e^x}{n!} \frac{d^n (e^{-x} x^n)}{dx^n}$$

```
[79]: #fig = plt.figure(1, figsize=(10,10))
x = np.linspace(-5, 8, 256)

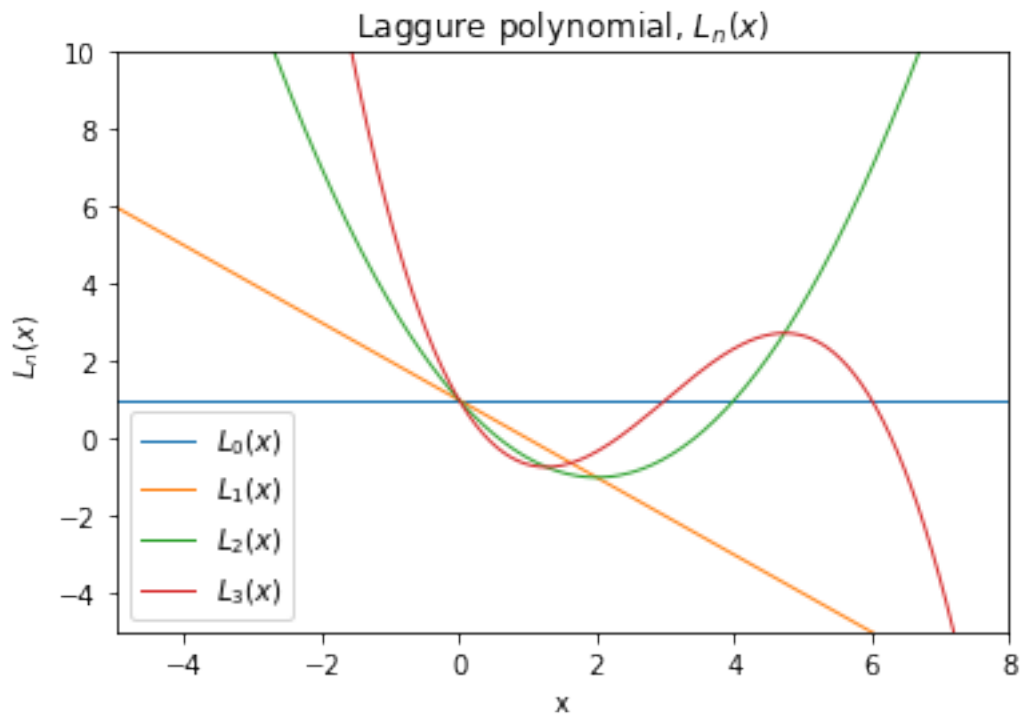
plt.figure()
plt.plot(x, np.polyval(special.laguerre(0),x), linewidth=1,label=r"$L_0(x)$")
plt.plot(x, np.polyval(special.laguerre(1),x), linewidth=1,label=r"$L_1(x)$")
plt.plot(x, np.polyval(special.laguerre(2),x), linewidth=1,label=r"$L_2(x)$")
plt.plot(x, np.polyval(special.laguerre(3),x), linewidth=1,label=r"$L_3(x)$")
```

```

#Set limits for axes
plt.xlim([-5,8])
plt.ylim([-5,10])

#Set axes labels
plt.xlabel("x")
plt.ylabel(r"$L_n(x)$")
plt.title(r"Laggure polynomial, $L_n(x)$")
plt.legend()
plt.show();

```



8.8 Hermite Polynomials:

Hermite polynomials were defined by Pierre-Simon Laplace in 1810 though in scarcely recognizable form, and studied in detail by Pafnuty Chebyshev in 1859. Chebyshev's work was overlooked, and they were named later after Charles Hermite, who wrote on the polynomials in 1864, describing them as new. They were consequently not new, although Hermite was the first to define the multidimensional polynomials in his later 1865 publications.

Hermite polynomials are solutions of Hermite differential equation:

$$\frac{d^2y(x)}{dx^2} - 2x\frac{dy(x)}{dx} + 2\lambda y = 0$$

Where Rodrigues formula for Hermite polynomial:

$$H_n(x) = (-1)^n e^{-x^2} \frac{d^n e^{-x^2}}{dx^n}$$

```
[80]: import matplotlib
import matplotlib.pyplot as plt
import numpy
import numpy.polynomial.hermite as Herm

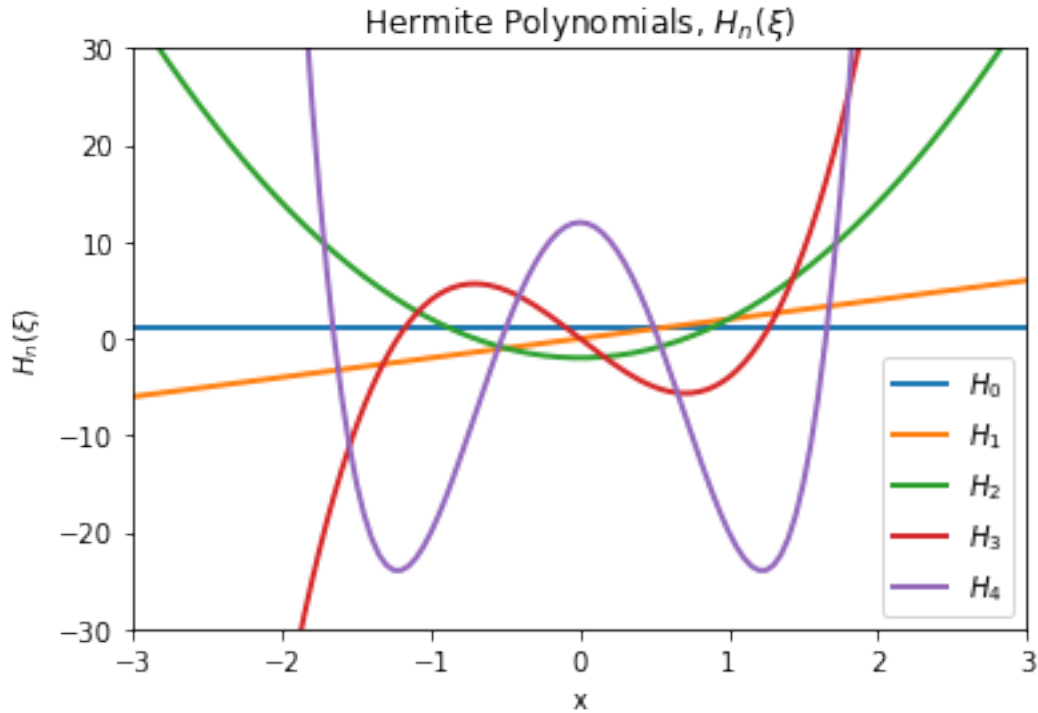
#Discretized space
dx = 0.05
x_lim = 12
x = numpy.arange(-x_lim,x_lim,dx)

def hermite(x, n):
    xi = x
    herm_coeffs = numpy.zeros(n+1)
    herm_coeffs[n] = 1
    return Herm.hermval(xi, herm_coeffs)

plt.figure()
plt.plot(x, hermite(x,0), linewidth=2,label=r"$H_0$")
plt.plot(x, hermite(x,1), linewidth=2,label=r"$H_1$")
plt.plot(x, hermite(x,2), linewidth=2,label=r"$H_2$")
plt.plot(x, hermite(x,3), linewidth=2,label=r"$H_3$")
plt.plot(x, hermite(x,4), linewidth=2,label=r"$H_4$")

#Set limits for axes
plt.xlim([-3,3])
plt.ylim([-30,30])

#Set axes labels
plt.xlabel("x")
plt.ylabel(r"$H_n(\xi)$")
plt.title(r"Hermite Polynomials, $H_n(\xi)$")
plt.legend()
plt.show()
```



We will use all this special function when we will discuss Quantum mechanics using Python section. We are very much useful for Harmonic oscillator problem and H-atom problem.

9 Solving ODEs using Python

Let's start this topic by solving a problem using a well known Brent method. Then we will solve ODEs using odeint (ODE integrator). A typical problem is to solve a second or higher order ODE for a given set of initial conditions. Here we illustrate using odeint to solve the equation for a driven damped pendulum.

9.1 Problem:

Find the solution of $\tan(x) - \sqrt{(8x)^2 - 1} = 0$ using Brent method.

9.2 Solution:

```
[81]: import numpy as np
      from scipy import optimize
      import matplotlib.pyplot as plt

      def tdl(x):
```

```

    y = 8./x
    return np.tan(x) - np.sqrt(y*y-1.0)
rx1 = optimize.brentq(tdl, 0.5, 0.49*np.pi)
rx2 = optimize.brentq(tdl, 0.51*np.pi, 1.49*np.pi)
rx3 = optimize.brentq(tdl, 1.51*np.pi, 2.49*np.pi)
rx = np.array([rx1, rx2, rx3])
ry = np.zeros(3)
print("\nTrue roots:")
print("\n".join("f({0:0.5f}) = {1:0.2e}".format(x, tdl(x)) for x in rx))

# Find false roots
rx1f = optimize.brentq(tdl, 0.49*np.pi, 0.51*np.pi)
rx2f = optimize.brentq(tdl, 1.49*np.pi, 1.51*np.pi)
rx3f = optimize.brentq(tdl, 2.49*np.pi, 2.51*np.pi)
rxf = np.array([rx1f, rx2f, rx3f])
print("\nfalse roots:")
print("\n".join("f({0:0.5f}) = {1:0.2e}".format(x, tdl(x)) for x in rxf))

x = np.linspace(0.7, 8, 128)
y = tdl(x)

ymask = np.ma.masked_where(np.abs(y)>20., y)
plt.figure(figsize=(4, 3))
plt.plot(x, ymask)
plt.axhline(color='black')
plt.axvline(x=np.pi/2., color="gray", linestyle="--", zorder=-1)
plt.axvline(x=3.*np.pi/2., color="gray", linestyle="--", zorder=-1)
plt.axvline(x=5.*np.pi/2., color="gray", linestyle="--", zorder=-1)
plt.xlabel(r"$x$")
plt.ylabel(r"$\tan x - \sqrt{(8/x)^2-1}$")
plt.ylim(-8, 8)
plt.plot(rx, ry, 'og', ms=5, label="true roots")
plt.plot(rxf, ry, 'xr', ms=5, label="false roots")
plt.legend(numpoints=1, fontsize="small", loc = "upper right", bbox_to_anchor =
    ↪(0.92, 0.97))
plt.tight_layout()
plt.show()

```

True roots:

f(1.39547) = -6.39e-14

f(4.16483) = -7.95e-14

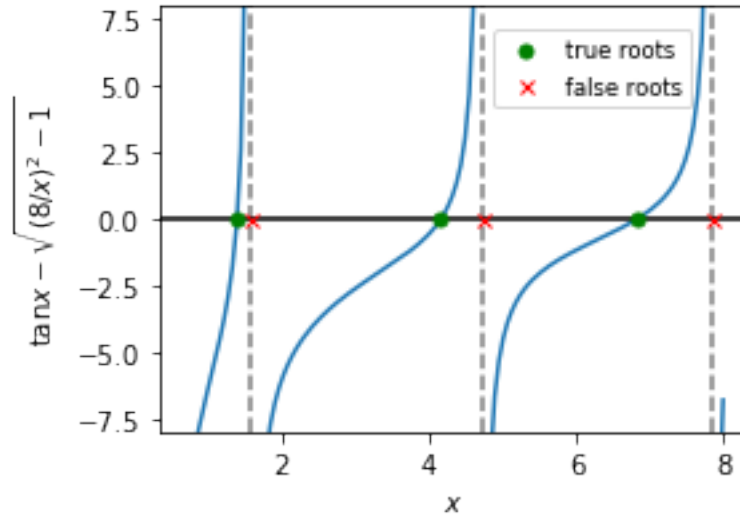
f(6.83067) = -1.22e-15

false roots:

f(1.57080) = -1.61e+12

f(4.71239) = -1.56e+12

$f(7.85398) = 1.17\text{e}+12$



9.3 Solve differential equations:

An example of using ODEINT is with the following differential equation with parameter $k=0.3$, the initial condition $y_0=5$ and the following differential equation.

$$\frac{dy(t)}{dt} + ky(t) = 0$$

```
[82]: import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
plt.figure(figsize=(4, 3))

# function that returns dy/dt
def func(y,t):
    k = 0.3
    dydt = - k * y
    return dydt

# initial condition
y0 = 5

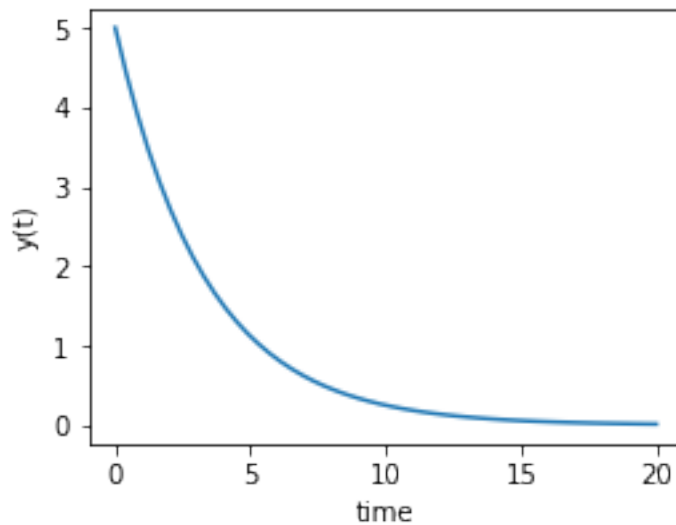
# time points
t = np.linspace(0,20)

# solve ODE
y = odeint(func,y0,t)
```

```

# plot results
plt.plot(t,y)
plt.xlabel('time')
plt.ylabel('y(t)')
plt.show()

```



```

[118]: import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
plt.figure(figsize=(4, 3))

# function that returns dy/dt
def model(y,t,k):
    dydt = -k * y
    return dydt

# initial condition
y0 = 5

# time points
t = np.linspace(0,20)

# solve ODEs
k = 0.1
y1 = odeint(model,y0,t,args=(k,))
k = 0.2
y2 = odeint(model,y0,t,args=(k,))

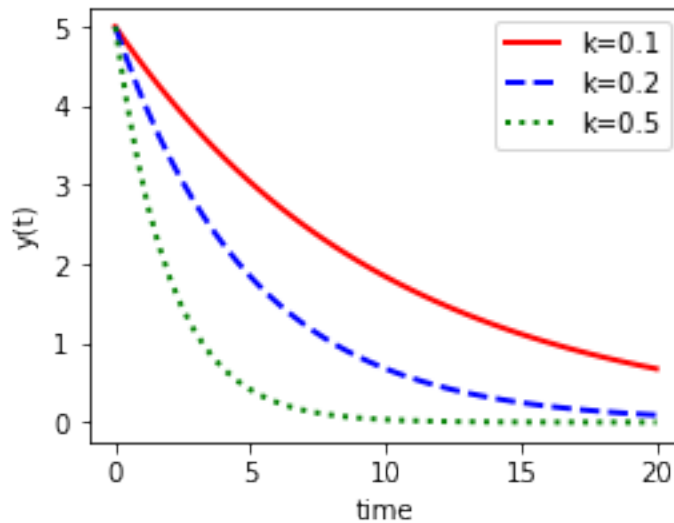
```

```

k = 0.5
y3 = odeint(model,y0,t,args=(k,))

# plot results
plt.plot(t,y1,'r-',linewidth=2,label='k=0.1')
plt.plot(t,y2,'b--',linewidth=2,label='k=0.2')
plt.plot(t,y3,'g:',linewidth=2,label='k=0.5')
plt.xlabel('time')
plt.ylabel('y(t)')
plt.legend()
plt.show()

```



9.4 Problem:

Find a numerical solution to the following differential equations with the associated initial conditions. Expand the requested time horizon until the solution reaches a steady state. Show a plot of the states ($x(t)$ and/or $y(t)$). Report the final value of each state as $t \rightarrow \infty$.

1.

$$\frac{dy(t)}{dt} + ky(t) - 1 = 0$$

$$y(0) = 0$$

```

[83]: import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
plt.figure(figsize=(4, 3))

```

```

# function that returns dy/dt
def model(y,t):
    dydt = -y + 1.0
    return dydt

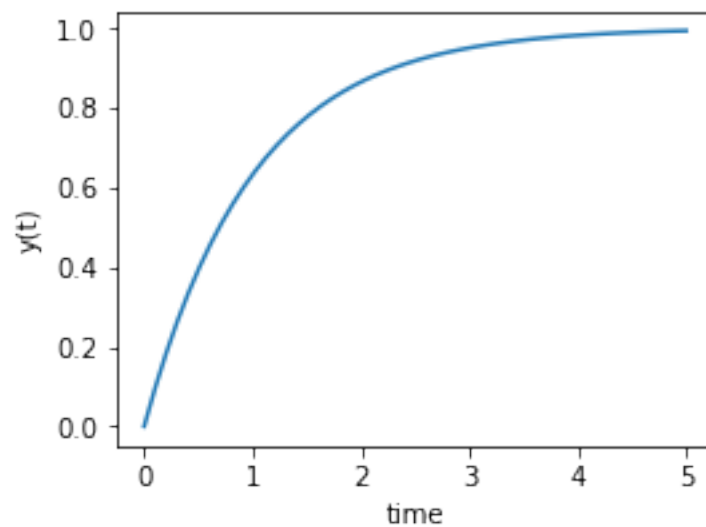
# initial condition
y0 = 0

# time points
t = np.linspace(0,5)

# solve ODE
y = odeint(model,y0,t)

# plot results
plt.plot(t,y)
plt.xlabel('time')
plt.ylabel('y(t)')
plt.show()

```



2.

$$5 \frac{dy(t)}{dt} + y(t) - u(t) = 0$$

$$y(0) = 1$$

u steps from 0 to 2 at t=10

```

[84]: import numpy as np
      from scipy.integrate import odeint
      import matplotlib.pyplot as plt

```

```

plt.figure(figsize=(4, 3))

# function that returns dy/dt
def model(y,t):
    # u steps from 0 to 2 at t=10
    if t<10.0:
        u = 0
    else:
        u = 2
    dydt = (-y + u)/5.0
    return dydt

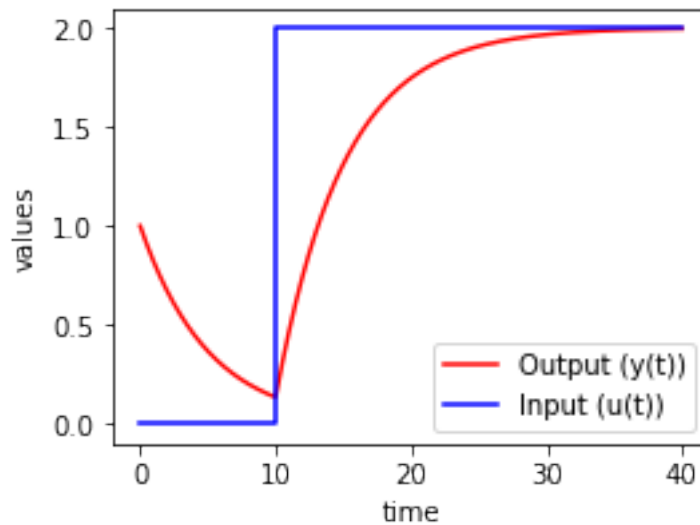
# initial condition
y0 = 1

# time points
t = np.linspace(0,40,1000)

# solve ODE
y = odeint(model,y0,t)

# plot results
plt.plot(t,y, 'r-',label='Output (y(t))')
plt.plot([0,10,10,40],[0,0,2,2], 'b-',label='Input (u(t))')
plt.ylabel('values')
plt.xlabel('time')
plt.legend(loc='best')
plt.show()

```



3. Solve for $x(t)$ and $y(t)$ and show that the solutions are equivalent.

$$\frac{dx(t)}{dt} = 3e^{-t}$$

$$\frac{dy(t)}{dt} = 3 - y(t)$$

$$y(0) = 0$$

$$x(0) = 0$$

```
[85]: import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
plt.figure(figsize=(4, 3))

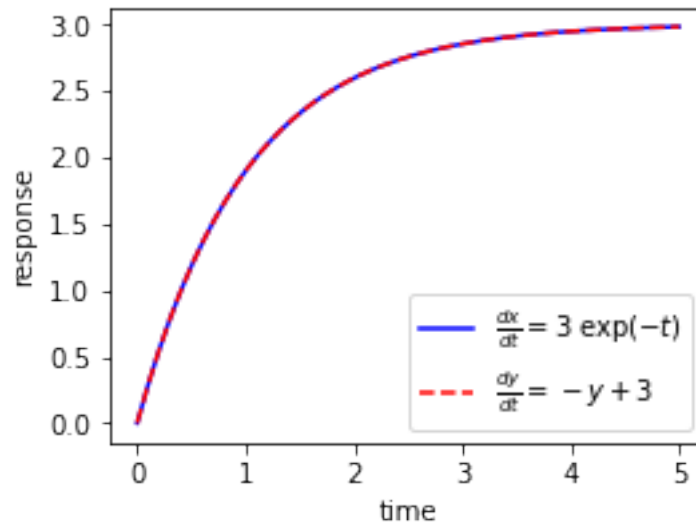
# function that returns dz/dt
def model(z,t):
    dxdt = 3.0 * np.exp(-t)
    dydt = -z[1] + 3
    dzdt = [dxdt,dydt]
    return dzdt

# initial condition
z0 = [0,0]

# time points
t = np.linspace(0,5)

# solve ODE
z = odeint(model,z0,t)

# plot results
plt.plot(t,z[:,0], 'b-', label=r'$\frac{dx}{dt}=3 \exp(-t)$')
plt.plot(t,z[:,1], 'r--', label=r'$\frac{dy}{dt}=-y+3$')
plt.ylabel('response')
plt.xlabel('time')
plt.legend(loc='best')
plt.show()
```



4.

$$2 \frac{dx(t)}{dt} = -x(t) + u(t)$$

$$5 \frac{dy(t)}{dt} = -y(t) + x(t)$$

$$u = 2S(t-5)$$

$$y(0) = 0$$

$$x(0) = 0$$

where $S(t-5)$ is a step function that changes from zero to one at $t=5$. When it is multiplied by two, it changes from zero to two at that same time, $t=5$.

```
[86]: import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
plt.figure(figsize=(4, 3))

# function that returns dz/dt
def model(z,t,u):
    x = z[0]
    y = z[1]
    dxdt = (-x + u)/2.0
    dydt = (-y + x)/5.0
    dzdt = [dxdt,dydt]
    return dzdt

# initial condition
z0 = [0,0]
```

```

# number of time points
n = 401

# time points
t = np.linspace(0,40,n)

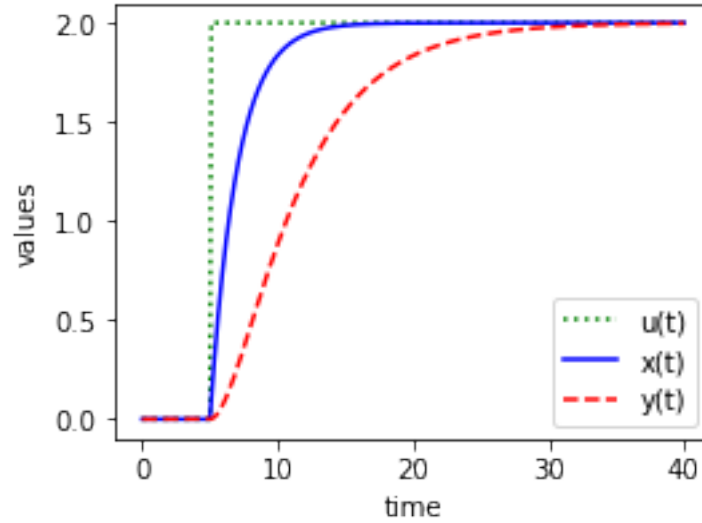
# step input
u = np.zeros(n)
# change to 2.0 at time = 5.0
u[51:] = 2.0

# store solution
x = np.empty_like(t)
y = np.empty_like(t)
# record initial conditions
x[0] = z0[0]
y[0] = z0[1]

# solve ODE
for i in range(1,n):
    # span for next time step
    tspan = [t[i-1],t[i]]
    # solve for next step
    z = odeint(model,z0,tspan,args=(u[i],))
    # store solution for plotting
    x[i] = z[1][0]
    y[i] = z[1][1]
    # next initial condition
    z0 = z[1]

# plot results
plt.plot(t,u,'g:',label='u(t)')
plt.plot(t,x,'b-',label='x(t)')
plt.plot(t,y,'r--',label='y(t)')
plt.ylabel('values')
plt.xlabel('time')
plt.legend(loc='best')
plt.show()

```

9.5 Solving systems of nonlinear equations:

Solving systems of nonlinear equations is not for the faint of heart. It is a difficult problem that lacks any general purpose solutions. Nevertheless, SciPy provides quite an assortment of numerical solvers for nonlinear systems of equations. However, because of the complexity and subtleties of this class of problems, we do not discuss their use here.

A typical problem is to solve a second or higher order ODE for a given set of initial conditions. Here we illustrate using `odeint` to solve the equation for a driven damped pendulum. The equation of motion for the angle θ that the pendulum makes with the vertical is given by:

$$\frac{d^2\theta(t)}{dt^2} = -\frac{1}{Q} \frac{d\theta(t)}{dt} + \sin\theta(t) + d\cos\Omega t$$

where t is time, Q is the quality factor, d is the forcing amplitude, and Ω is the driving frequency of the forcing. Reduced variables have been used such that the natural (angular) frequency of oscillation is 1. The ODE is nonlinear owing to the $\sin\theta$ term. Of course, it's precisely because there are no general methods for solving nonlinear ODEs that one employs numerical techniques, so it seems appropriate that we illustrate the method with a nonlinear ODE.

we can rewrite our second order ODE as two coupled first order ODEs:

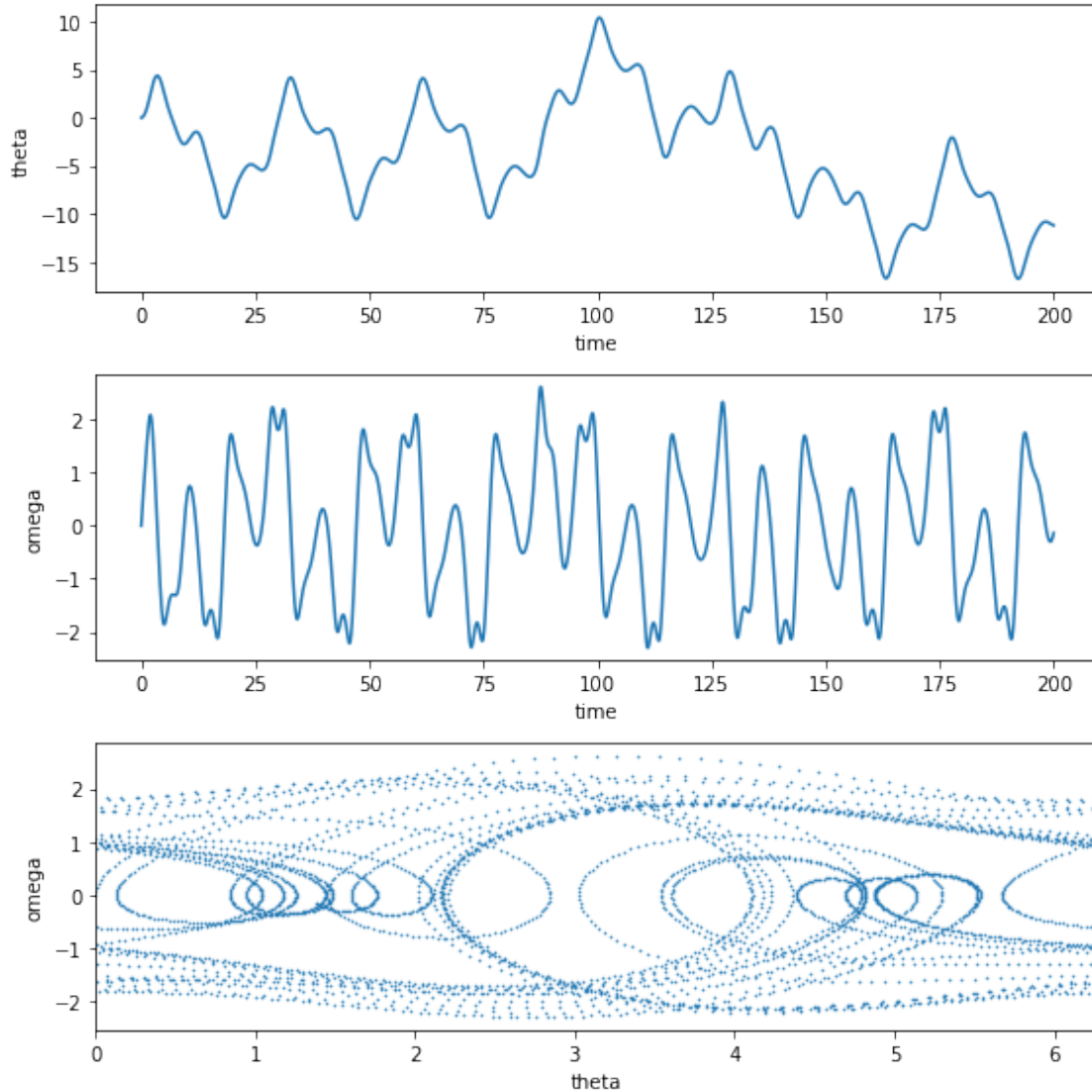
$$\frac{d\theta(t)}{dt} = w$$

$$\frac{dw(t)}{dt} = -\frac{w}{Q} + \sin\theta(t) + d\cos\Omega t$$

```

[87]: import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
def f(y, t, params):
    theta, omega = y # unpack current values of y
    Q, d, Omega = params
    derivs = [omega, -omega/Q + np.sin(theta) + d*np.cos(Omega*t)]
    return derivs
Q = 2.0 # quality factor (inverse damping)
d = 1.5
Omega = 0.65
theta0 = 0.0
omega0 = 0.0
params = [Q, d, Omega]
y0 = [theta0, omega0]
tStop = 200.
tInc = 0.05
t = np.arange(0., tStop, tInc)
psoln = odeint(f, y0, t, args=(params,))
fig = plt.figure(1, figsize=(8,8))
ax1 = fig.add_subplot(311)
ax1.plot(t, psoln[:,0])
ax1.set_xlabel("time")
ax1.set_ylabel("theta")
ax2 = fig.add_subplot(312)
ax2.plot(t, psoln[:,1])
ax2.set_xlabel("time")
ax2.set_ylabel("omega")
ax3 = fig.add_subplot(313)
twopi = 2.0*np.pi
ax3.plot(psoln[:,0]%twopi, psoln[:,1], ".", ms=1)
ax3.set_xlabel("theta")
ax3.set_ylabel("omega")
ax3.set_xlim(0., twopi)
plt.tight_layout()
plt.show()

```



The plots above reveal that for the particular set of input parameters chosen $Q = 2.0$, $d = 1.5$, and $\Omega = 0.65$, the pendulum trajectories are chaotic. Weaker forcing (smaller d) leads to what is perhaps the more familiar behavior of sinusoidal oscillations with a fixed frequency which, at long times, is equal to the driving frequency.

10 Fourier Series using Python

We know that there are many ways by which any complicated function may be expressed as power series. This is not the only way in which a function may be expressed as a series but there is a method of expressing a periodic function as an infinite sum of sine and cosine functions. This representation is known as Fourier series. The computation and study of Fourier series is known as harmonic analysis and is useful as a way to break up an arbitrary periodic function into a set of simple harmonic terms.

that can be plugged in, solved individually, and then recombined to obtain the solution to the original problem or an approximation to it to whatever accuracy is desired. Unlike Taylor series, a Fourier series can describe functions that are not everywhere continuous and/or differentiable. There are other advantages of using trigonometric terms. They are easy to differentiate and integrate and each term contains only one characteristic frequency. Analysis of Fourier series becomes important because this method is used to represent the response of a system to a periodic input and the response depends on the frequency content of the input.

So the Fourier series of the function $f(x)$ over the periodic interval $(0, L)$ written as:

$$f(x') = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{2\pi n x'}{L}\right) + b_n \sin\left(\frac{2\pi n x'}{L}\right) \right]$$

where Fourier coefficients are:

$$\begin{aligned} a_0 &= \frac{2}{L} \int_0^L f(x') dx' \\ a_n &= \frac{2}{L} \int_0^L f(x') \cos\left(\frac{2\pi n x'}{L}\right) dx' \\ b_n &= \frac{2}{L} \int_0^L f(x') \sin\left(\frac{2\pi n x'}{L}\right) dx' \end{aligned}$$

So the Fourier series of the function $f(x)$ over the periodic interval $(-L, L)$ written as:

$$f(x') = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{\pi n x'}{L}\right) + b_n \sin\left(\frac{\pi n x'}{L}\right) \right]$$

where Fourier coefficients are:

$$\begin{aligned} a_0 &= \frac{2}{L} \int_{-L}^L f(x') dx' \\ a_n &= \frac{2}{L} \int_{-L}^L f(x') \cos\left(\frac{\pi n x'}{L}\right) dx' \\ b_n &= \frac{2}{L} \int_{-L}^L f(x') \sin\left(\frac{\pi n x'}{L}\right) dx' \end{aligned}$$

So the Fourier series of the function $f(x)$ over the periodic interval $(-\pi, \pi)$ written as:

$$f(x') = \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos(nx') + b_n \sin(nx')]$$

where Fourier coefficients are:

$$a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x') dx'$$

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x') \cos(nx') dx'$$

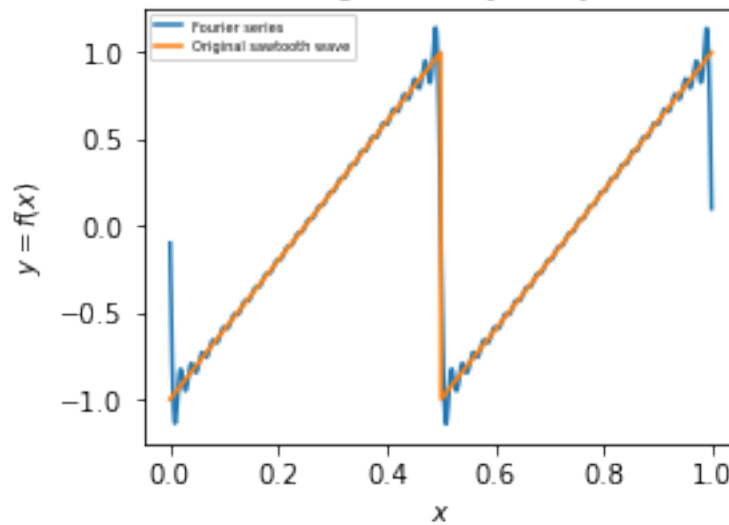
$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x') \sin(nx') dx'$$

10.1 a) Fourier series analysis for a sawtooth wave function

```
[88]: import numpy as np
from scipy.signal import square,sawtooth
import matplotlib.pyplot as plt
from scipy.integrate import.simps
plt.figure(figsize=(4, 3))

# Periodicity of the periodic function f(x)
L=1
# No of waves in time period L
freq=2
width_range=1
samples=1000
terms=50
# Generation of Sawtooth function
x=np.linspace(0,L,samples,endpoint=False)
y=sawtooth(2.0*np.pi*x*freq/L,width=width_range)
# Calculation of Co-efficients
a0=2./L*simps(y,x)
an=lambda n:2.0/L*simps(y*np.cos(2.*np.pi*n*x/L),x)
bn=lambda n:2.0/L*simps(y*np.sin(2.*np.pi*n*x/L),x)
# Sum of the series
s=a0/2.+sum([an(k)*np.cos(2.*np.pi*k*x/L)+bn(k)*np.sin(2.*np.pi*k*x/L) for k in
↪range(1,terms+1)])
plt.plot(x,s,label="Fourier series")
plt.plot(x,y,label="Original sawtooth wave")
plt.xlabel("$x$")
plt.ylabel("$y=f(x)$")
plt.legend(loc='best',prop={'size':5})
plt.title("Sawtooth wave signal analysis by Fouries series")
#plt.savefig("fs_sawtooth.png")
plt.show()
```

Sawtooth wave signal analysis by Fourier series



10.2 b) Fourier series analysis for a square wave function

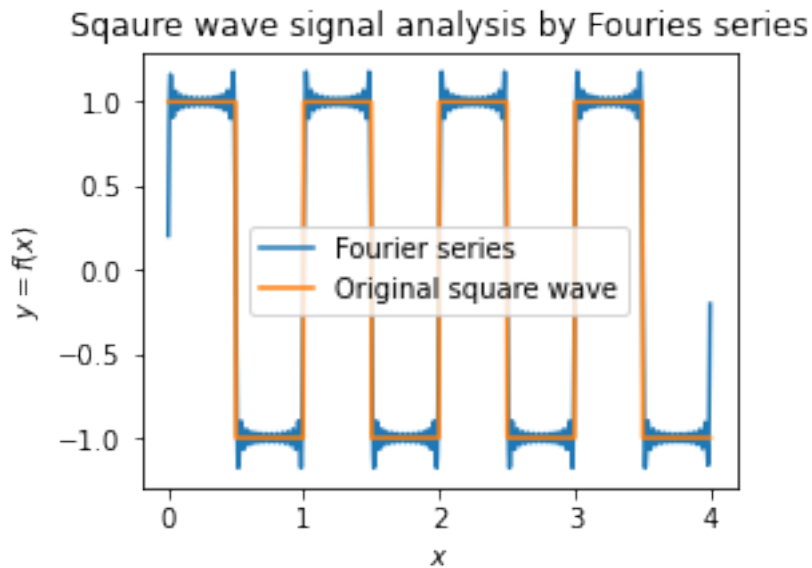
```
[89]: import numpy as np
from scipy.signal import square
import matplotlib.pyplot as plt
from scipy.integrate import.simps
plt.figure(figsize=(4, 3))

# Periodicity of the periodic function f(x)
L=4
# No of waves in time period L
freq=4
dutycycle=0.5
samples=1000
terms=100
# Generation of square wave
x=np.linspace(0,L,samples,endpoint=False)
y=square(2.0*np.pi*x*freq/L,duty=dutycycle)
# Calculation of Fourier coefficients
a0=2./L*simps(y,x)
an=lambda n:2.0/L*simps(y*np.cos(2.*np.pi*n*x/L),x)
bn=lambda n:2.0/L*simps(y*np.sin(2.*np.pi*n*x/L),x)
# sum of the series
s=a0/2.+sum([an(k)*np.cos(2.*np.pi*k*x/L)+bn(k)*np.sin(2.*np.pi*k*x/L) for k in
    range(1,terms+1)])
# Plotting
```

```

plt.plot(x,s,label="Fourier series")
plt.plot(x,y,label="Original square wave")
plt.xlabel("$x$")
plt.ylabel("$y=f(x)$")
plt.legend(loc='best',prop={'size':10})
plt.title("Sqaure wave signal analysis by Fouries series")
#plt.savefig("fs_square.png")
plt.show()

```



10.3 c) Fourier series analysis for a Triangular wave function:

```

[90]: import numpy as np
from scipy.signal import square,sawtooth,triang
import matplotlib.pyplot as plt
from scipy.integrate import.simps
plt.figure(figsize=(4, 3))

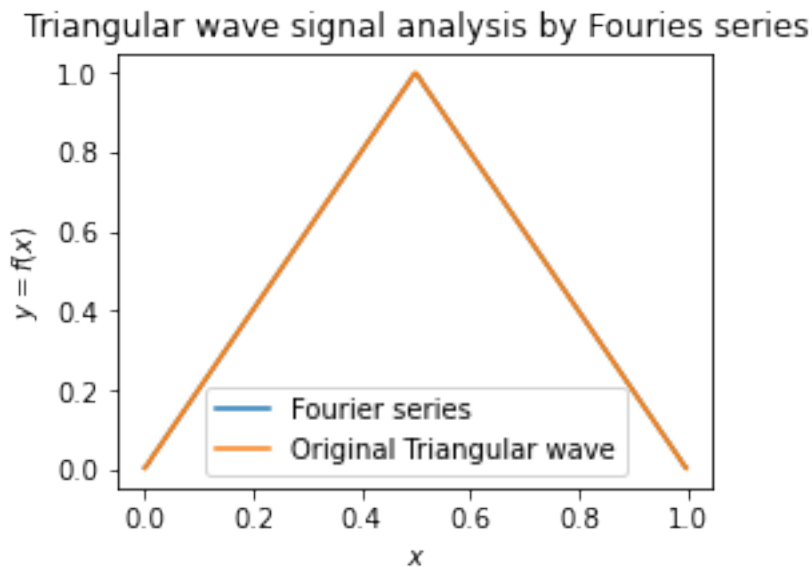
# Periodicity of the periodic function f(x)
L=1
samples=501
terms=50
# Generation of Triangular wave
x=np.linspace(0,L,samples,endpoint=False)
y=triang(samples)
# Fourier Coefficients
a0=2./L*simps(y,x)

```

```

an=lambda n:2.0/L*simps(y*np.cos(2.*np.pi*n*x/L),x)
bn=lambda n:2.0/L*simps(y*np.sin(2.*np.pi*n*x/L),x)
# Series sum
s=a0/2.+sum([an(k)*np.cos(2.*np.pi*k*x/L)+bn(k)*np.sin(2.*np.pi*k*x/L) for k in
    range(1,terms+1)])
# Plotting
plt.plot(x,s,label="Fourier series")
plt.plot(x,y,label="Original Triangular wave")
plt.xlabel("$x$")
plt.ylabel("$y=f(x)$")
plt.legend(loc='best',prop={'size':10})
plt.title("Triangular wave signal analysis by Fouries series")
plt.savefig("fs_triangular.png")
plt.show()

```



10.4 d) Fourier series analysis for a Arbitrary waves function:

```

[91]: import matplotlib.pyplot as plt
from scipy.integrate import simps
import numpy as np
plt.figure(figsize=(4, 3))

L=1.0 # half wavelength, Wavelength=2L
freq=2 # frequency
samples=1001
terms=300

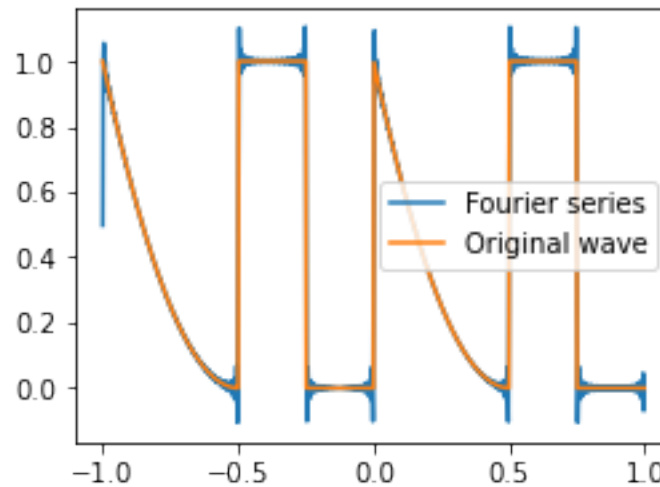
```



```

# Generating wave
x=np.linspace(-L,L,samples,endpoint=False)
F=lambda x: np.array([u**2 if -L<=u<0 else 1 if 0<u<0.5 else 0 for u in x])
#F=lambda x: abs(np.sin(2*np.pi*x))
f=lambda x: F(freq*x%(2*L)-L)
# Fourier Coefficients
a0=1./L*simps(f(x),x)
an=lambda n:1.0/L*simps(f(x)*np.cos(1.*np.pi*n*x/L),x)
bn=lambda n:1.0/L*simps(f(x)*np.sin(1.*np.pi*n*x/L),x)
# Series sum
xp=x
s=a0/2.+sum([an(k)*np.cos(1.*np.pi*k*xp/L)+bn(k)*np.sin(1.*np.pi*k*xp/L) for k_
    ↪in range(1,terms+1)])
#Plotting
plt.plot(xp,s,label="Fourier series")
plt.plot(xp,f(xp),label="Original wave")
plt.legend(loc='best',prop={'size':10})
#plt.savefig("arb_ud.png")
plt.show()

```



11 Discrete (Fast) Fourier Transform using Python

The SciPy library has a number of routines for performing discrete Fourier transforms. Before delving into them, we provide a brief review of Fourier transforms and discrete Fourier transforms.

11.1 Continuous Fourier Transformation:

The Fourier transform of a function $g(t)$ is given by:

$$G(f) = \int_{-\infty}^{\infty} g(t) e^{-2\pi i f t} dt$$

where f is the Fourier transform variable; if t is time, then f is frequency. The inverse transform is given by:

$$g(t) = \int_{-\infty}^{\infty} G(f) e^{2\pi i f t} dt$$

The conventional Fourier transform is defined for continuous functions, or at least for functions that are dense and thus have an infinite number of data points.

11.2 Discrete Fourier Transformation:

When we are doing numerical analysis, however, you work with discrete data sets, that is, data sets defined for a finite number of points. The discrete Fourier transform (DFT) is defined for a function g_n consisting of a set of N discrete data points. Those N data points must be defined at equally-spaced times $t_n = n\Delta t$ where Δt is the time between successive data points and n runs from 0 to $N-1$. The discrete Fourier transform (DFT) of g_n is defined as:

$$G_l = \sum_{n=0}^{N-1} g_n e^{-i(\frac{2\pi}{N})ln}$$

Inverse discrete Fourier transform is defined as:

$$g_n = \frac{1}{N} \sum_{l=0}^{N-1} G_l e^{i(\frac{2\pi}{N})ln}$$

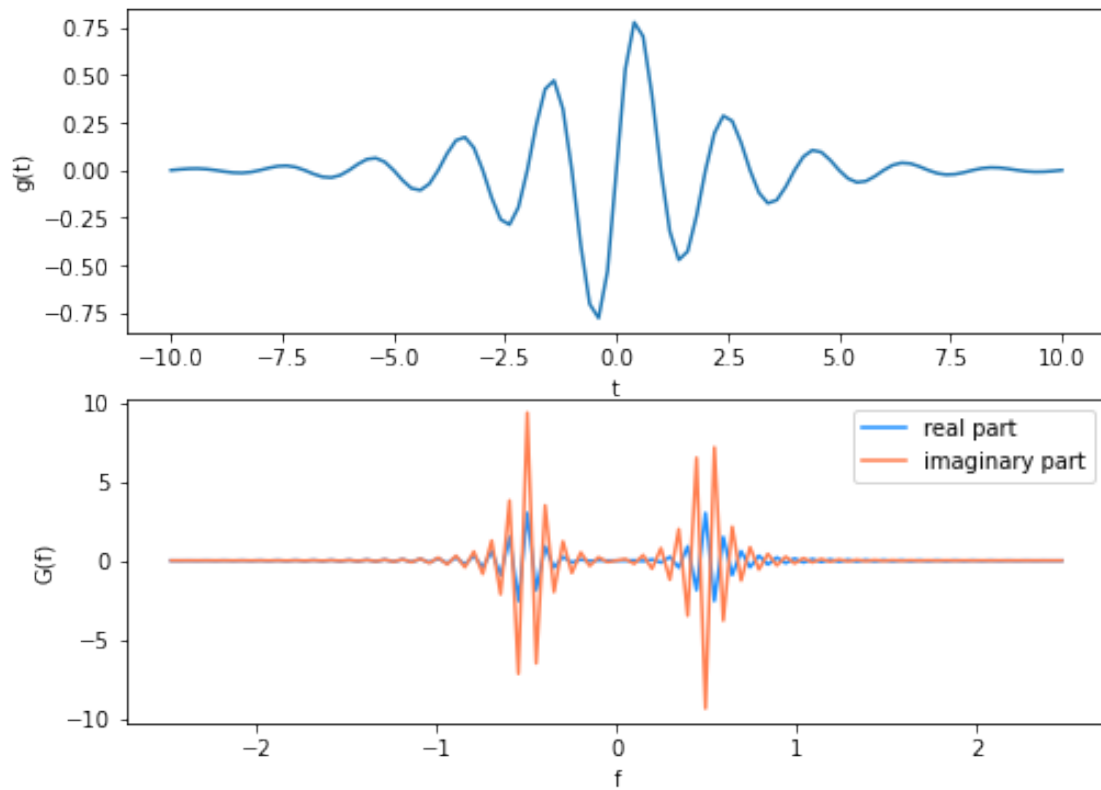
The DFT is usually implemented on computers using the well-known Fast Fourier Transform (FFT) algorithm, generally credited to Cooley and Tukey who developed it at AT&T Bell Laboratories during the 1960s. But their algorithm is essentially one of many independent rediscoveries of the basic algorithm dating back to Gauss who described it as early as 1805.

```
[92]: import numpy as np
      from scipy import fftpack
      import matplotlib.pyplot as plt
      width = 2.0
      freq = 0.5
      t = np.linspace(-10, 10, 101) # linearly space time array
      g = np.exp(-np.abs(t)/width)*np.sin(2.0*np.pi*freq*t)
      dt = t[1]-t[0]
```

```

G = fftpack.fft(g)
f = fftpack.fftfreq(g.size, d=dt)
f = fftpack.fftshift(f)
G = fftpack.fftshift(G)
fig = plt.figure(1, figsize=(8,6), frameon=False)
ax1 = fig.add_subplot(211)
ax1.plot(t, g)
ax1.set_xlabel("t")
ax1.set_ylabel("g(t)")
ax2 = fig.add_subplot(212)
ax2.plot(f, np.real(G), color="dodgerblue", label="real part")
ax2.plot(f, np.imag(G), color="coral", label="imaginary part")
ax2.legend()
ax2.set_xlabel("f")
ax2.set_ylabel("G(f)")
plt.show()

```

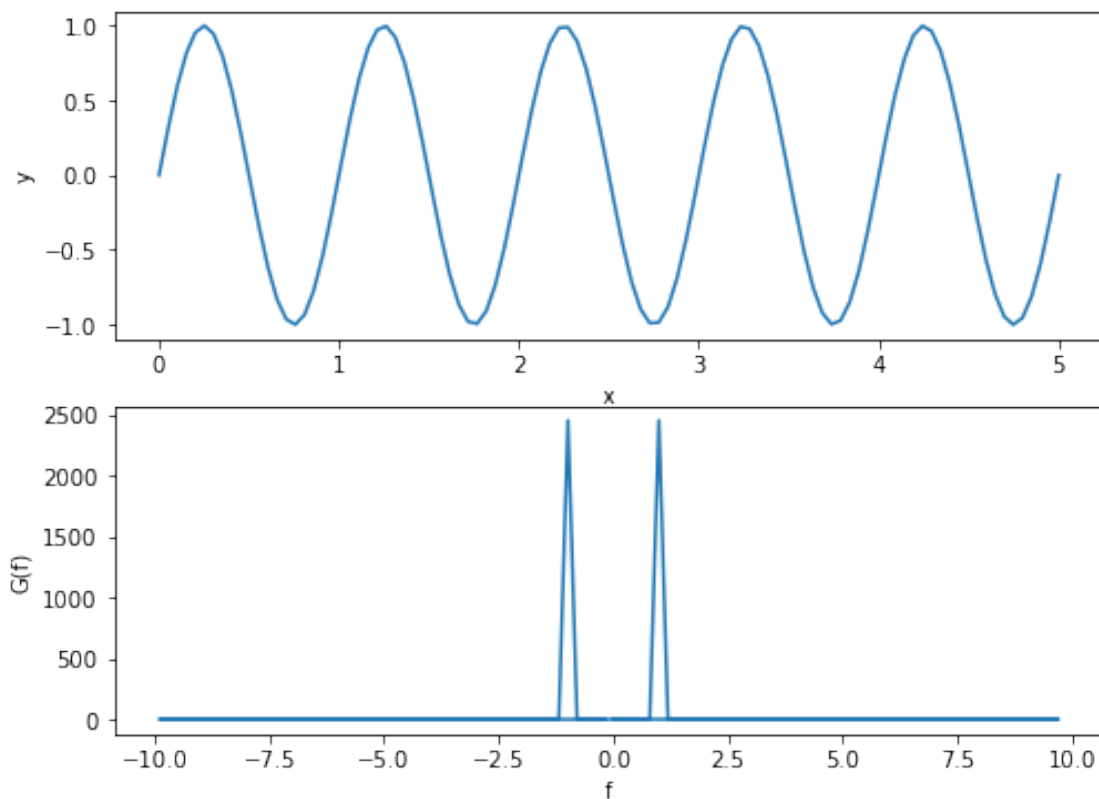


11.3 Problem:

Calculate Fast Fourier Transform (FFT) of $\sin(2\pi x)$:

11.4 Solution:

```
[93]: x = np.linspace(0,5,100)
y = np.sin(2*np.pi*x)
fig = plt.figure(1, figsize=(8,6), frameon=False)
ax1 = fig.add_subplot(211)
ax1.plot(x, y)
ax1.set_xlabel("x")
ax1.set_ylabel("y")
## fourier transform
f = np.fft.fft(y)
## sample frequencies
freq = np.fft.fftfreq(len(y), d=x[1]-x[0])
ax2 = fig.add_subplot(212)
ax2.plot(freq, abs(f)**2)
ax2.set_xlabel("f")
ax2.set_ylabel("G(f)")
plt.show()
```



```
[94]: # app.py
```

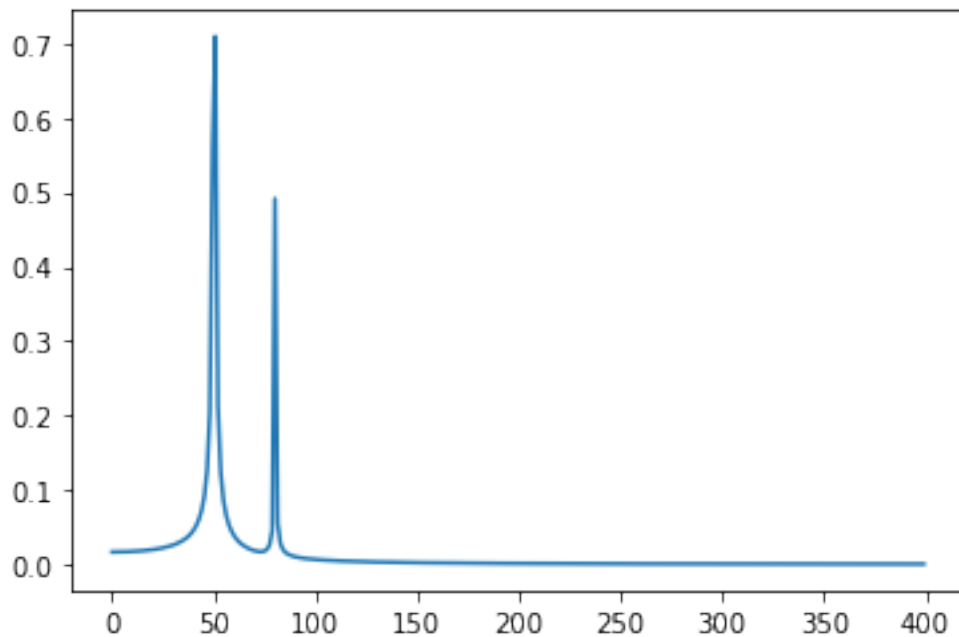
```

import matplotlib.pyplot as plt
import numpy as np
import scipy.fftpack

# Number of sample points
N = 600
# sample spacing
T = 1.0 / 800.0
x = np.linspace(0.0, N*T, N)
y = np.sin(50.0 * 2.0*np.pi*x) + 0.5*np.sin(80.0 * 2.0*np.pi*x)
yf = scipy.fftpack.fft(y)
xf = np.linspace(0.0, 1.0/(2.0*T), N//2)

fig, ax = plt.subplots()
ax.plot(xf, 2.0/N * np.abs(yf[:N//2]))
plt.show()

```



```

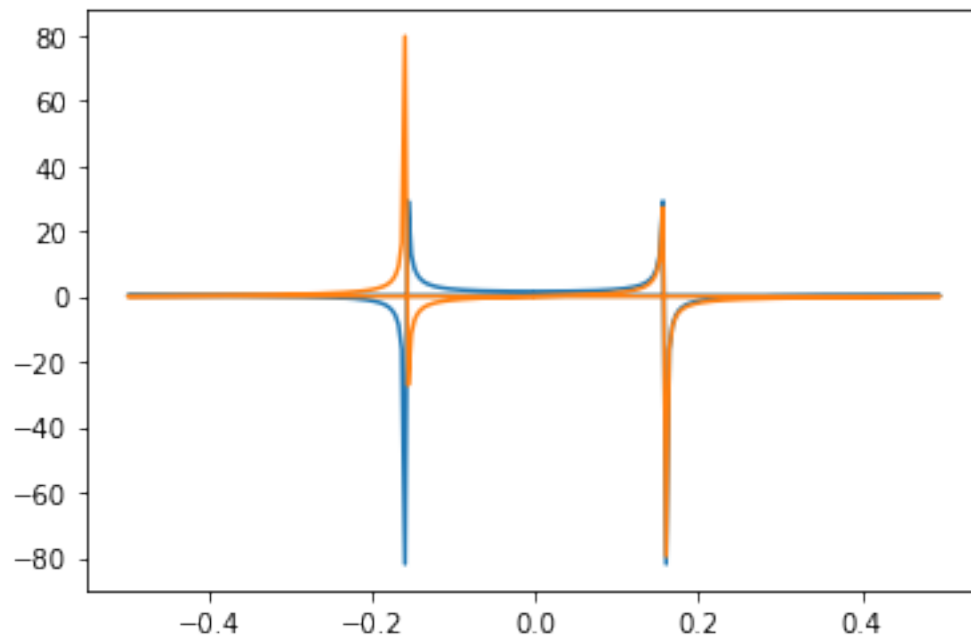
[95]: # app.py

import matplotlib.pyplot as plt
import numpy as np

t = np.arange(256)
sp = np.fft.fft(np.sin(t))
freq = np.fft.fftfreq(t.shape[-1])
plt.plot(freq, sp.real, freq, sp.imag)

```

```
plt.show()
```

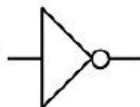
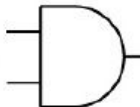


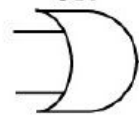



12 Logic gates using Python

```
[98]: Image(filename="pasted-image-0-3.png",width=800)
```

[98]:

Basic Logic Gates

<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Y	0	1	1	0		Y									
A	Y																
0	1																
1	0																
<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1		Y
A	B	Y															
0	0	0															
0	1	0															
1	0	0															
1	1	1															
<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0		Y
A	B	Y															
0	0	1															
0	1	1															
1	0	1															
1	1	0															
<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0		Y
A	B	Y															
0	0	0															
0	1	1															
1	0	1															
1	1	0															
<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1		Y
A	B	Y															
0	0	0															
0	1	1															
1	0	1															
1	1	1															
<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0		Y
A	B	Y															
0	0	1															
0	1	0															
1	0	0															
1	1	0															

12.1 NOT GATE:

```
[99]: # Function to simulate NOT Gate
def NOT(A):
    return ~A+2
print("Output of NOT GATE:")
print("A = 0, Y =",NOT(0))
print("A = 1, Y =",NOT(1))
```

Output of NOT GATE:

A = 0, Y = 1

A = 1, Y = 0

12.2 AND GATE:

```
[100]: # Function to simulate AND Gate
def AND(A, B):
    return A & B
print("Output of AND GATE:")
print("A = 0, B = 0 | Y =", AND(0, 0))
```

```
print("A = 0, B = 1 | Y =", AND(0, 1))
print("A = 1, B = 0 | Y =", AND(1, 0))
print("A = 1, B = 1 | Y =", AND(1, 1))
```

Output of AND GATE:

```
A = 0, B = 0 | Y = 0
A = 0, B = 1 | Y = 0
A = 1, B = 0 | Y = 0
A = 1, B = 1 | Y = 1
```

12.3 OR GATE:

```
[101]: # Function to simulate OR Gate
def OR(A, B):
    return A | B

print("Output of OR GATE:")

print("A = 0, B = 0 | Y =", OR(0, 0))
print("A = 0, B = 1 | Y =", OR(0, 1))
print("A = 1, B = 0 | Y =", OR(1, 0))
print("A = 1, B = 1 | Y =", OR(1, 1))
```

Output of OR GATE:

```
A = 0, B = 0 | Y = 0
A = 0, B = 1 | Y = 1
A = 1, B = 0 | Y = 1
A = 1, B = 1 | Y = 1
```

12.4 NAND GATE:

```
[102]: # Function to simulate AND Gate
def AND(A, B):
    return A & B;

# Function to simulate NOT Gate
def NOT(A):
    return ~A+2

# Function to simulate NAND Gate
def NAND(A, B):
    return NOT(AND(A, B))

print("Output of NAND GATE:")
print("A = 0, B = 0 | Y =", NAND(0, 0))
```



```
print("A = 0, B = 1 | Y =", NAND(0, 1))
print("A = 1, B = 0 | Y =", NAND(1, 0))
print("A = 1, B = 1 | Y =", NAND(1, 1))
```

Output of NAND GATE:

```
A = 0, B = 0 | Y = 1
A = 0, B = 1 | Y = 1
A = 1, B = 0 | Y = 1
A = 1, B = 1 | Y = 0
```

12.5 NOR GATE:

```
[103]: # Function to calculate OR Gate
def OR(A, B):
    return A | B;

# Function to simulate NOT Gate
def NOT(A):
    return ~A+2

# Function to simulate NOR Gate
def NOR(A, B):
    return NOT(OR(A, B))

print("Output of NOR GATE:")
print("A = 0, B = 0 | Y =", NOR(0, 0))
print("A = 0, B = 1 | Y =", NOR(0, 1))
print("A = 1, B = 0 | Y =", NOR(1, 0))
print("A = 1, B = 1 | Y =", NOR(1, 1))
```

Output of NOR GATE:

```
A = 0, B = 0 | Y = 1
A = 0, B = 1 | Y = 0
A = 1, B = 0 | Y = 0
A = 1, B = 1 | Y = 0
```

12.6 XOR GATE:

```
[104]: # Function to simulate XOR Gate
def XOR(A, B):
    return A ^ B

print("Output of XOR GATE:")
print("A = 0, B = 0 | Y =", XOR(0, 0))
print("A = 0, B = 1 | Y =", XOR(0, 1))
```

```
print("A = 1, B = 0 | Y =", XOR(1, 0))
print("A = 1, B = 1 | Y =", XOR(1, 1))
```

Output of XOR GATE:

A = 0, B = 0 | Y = 0

A = 0, B = 1 | Y = 1

A = 1, B = 0 | Y = 1

A = 1, B = 1 | Y = 0

12.7 Problem:

Calculate XNOR GATE using Python.

12.8 Solution:

```
[105]: # Function to simulate XOR Gate
def XOR(A, B):
    return A ^ B

# Function to simulate NOT Gate
def NOT(A):
    return ~A+2

# Function to simulate XNOR Gate
def XNOR(A, B):
    return NOT(XOR(A, B))

print("Output of XNOR GATE:")
print("A = 0, B = 0 | Y =", XNOR(0, 0))
print("A = 0, B = 1 | Y =", XNOR(0, 1))
print("A = 1, B = 0 | Y =", XNOR(1, 0))
print("A = 1, B = 1 | Y =", XNOR(1, 1))
```

Output of XNOR GATE:

A = 0, B = 0 | Y = 1

A = 0, B = 1 | Y = 0

A = 1, B = 0 | Y = 0

A = 1, B = 1 | Y = 1

12.9 Homework:

You can try to simulate all above logic gate from scratch.

12.9.1 Example 1: NOR GATE

```
[106]: def NOR(A, B):  
        if(A == 0) and (B == 0):  
            return 1  
        elif(A == 0) and (B == 1):  
            return 0  
        elif(A == 1) and (B == 0):  
            return 0  
        elif(A == 1) and (B == 1):  
            return 1  
    # main function  
    if __name__=='__main__':  
  
        print("Output of NOR GATE:")  
        print("A = 0, B = 0 | Y =", NOR(0, 0))  
        print("A = 0, B = 1 | Y =", NOR(0, 1))  
        print("A = 1, B = 0 | Y =", NOR(1, 0))  
        print("A = 1, B = 1 | Y =", NOR(1, 1))
```

Output of NOR GATE:

```
A = 0, B = 0 | Y = 1  
A = 0, B = 1 | Y = 0  
A = 1, B = 0 | Y = 0  
A = 1, B = 1 | Y = 1
```

12.9.2 Example 2: XOR GATE

```
[107]: def XOR (a, b):  
        if a != b:  
            return 1  
        else:  
            return 0  
    # main function  
    if __name__=='__main__':  
        print("Output of XOR GATE:")  
        print("A = 0, B = 0 | Y =", XOR(0, 0))  
        print("A = 0, B = 1 | Y =", XOR(0, 1))  
        print("A = 1, B = 0 | Y =", XOR(1, 0))  
        print("A = 1, B = 1 | Y =", XOR(1, 1))
```

Output of XOR GATE:

```
A = 0, B = 0 | Y = 0  
A = 0, B = 1 | Y = 1  
A = 1, B = 0 | Y = 1  
A = 1, B = 1 | Y = 0
```

You can now try remaining GATE to construct from basic.

13 Python for electrodynamics

I will try to play with Python in electrodynamics domain. This will be very interesting to play in this domain. We will try to explain few interesting problem in electrodynamics. There are many topics we can discuss electrodynamics using Python but we will try to keep our discussion as simple as possible.

13.1 Visualizing a vector field with Matplotlib :

Matplotlib provides a function, streamplot, to create a plot of streamlines representing a vector field. The following program displays a representation of the electric field vector resulting from a multipole arrangement of charges. The multipole is selected as a power of 2 on the command line (1=dipole, 2=quadrupole, etc.)

It requires Matplotlib 1.5+ because of the choice of colormap (plt.cm.inferno): this can be replaced with another (for example plt.cm.hot) if using an older version of Matplotlib.

```
[108]: import sys
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Circle

def E(q, r0, x, y):
    """Return the electric field vector E=(Ex,Ey) due to charge q at r0."""
    den = np.hypot(x-r0[0], y-r0[1])**3
    return q * (x - r0[0]) / den, q * (y - r0[1]) / den

# Grid of x, y points
nx, ny = 64, 64
x = np.linspace(-2, 2, nx)
y = np.linspace(-2, 2, ny)
X, Y = np.meshgrid(x, y)

# Create a multipole with nq charges of alternating sign, equally spaced
# on the unit circle.
#nq = 2**int(sys.argv[1])
nq = 4
charges = []
for i in range(nq):
    q = i%2 * 2 - 1
    charges.append((q, (np.cos(2*np.pi*i/nq), np.sin(2*np.pi*i/nq))))

# Electric field vector, E=(Ex, Ey), as separate components
Ex, Ey = np.zeros((ny, nx)), np.zeros((ny, nx))
```

```

for charge in charges:
    ex, ey = E(*charge, x=X, y=Y)
    Ex += ex
    Ey += ey

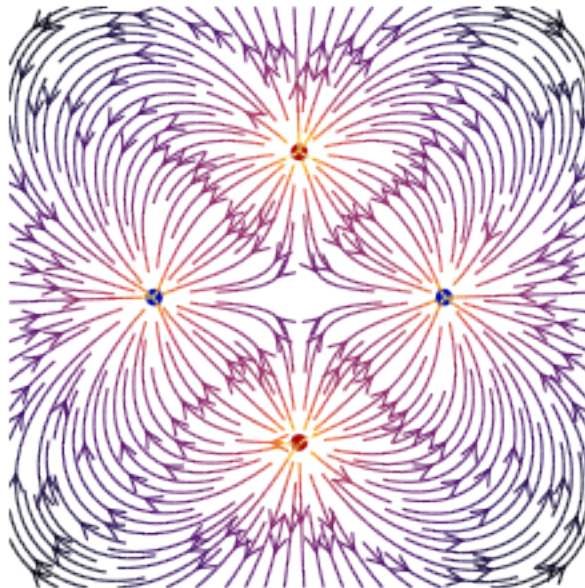
fig = plt.figure()
ax = fig.add_subplot(111)

# Plot the streamlines with an appropriate colormap and arrow style
color = 2 * np.log(np.hypot(Ex, Ey))
ax.streamplot(x, y, Ex, Ey, color=color, linewidth=1, cmap=plt.cm.inferno,
              density=2, arrowstyle='->', arrowsize=1.5)

# Add filled circles for the charges themselves
charge_colors = {True: '#aa0000', False: '#0000aa'}
for q, pos in charges:
    ax.add_artist(Circle(pos, 0.05, color=charge_colors[q>0]))

ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_xlim(-2,2)
ax.set_ylim(-2,2)
ax.set_aspect('equal')
plt.axis('off');
plt.show()

```



13.2 Electric field and potential due to a charge particle:

Let's first define a charged particle class that allows us to compute the field and the potential.

13.2.1 Electric field for charge particle:

```
[109]: import numpy as np
import matplotlib.pyplot as plt
plt.style.use('bmh')
# Let's define a class for electric field and potential

class ChargedParticle:
    def __init__(self, pos, charge):
        self.pos = np.asarray(pos)
        self.charge = charge

    def compute_field(self, x, y):
        X, Y = np.meshgrid(x, y)
        u_i = np.hstack((X.ravel()[:, np.newaxis], Y.ravel()[:, np.newaxis]))
        ↪ self.pos
        r = np.sqrt((X - self.pos[0])**2 + (Y - self.pos[1])**2)
        field = ((self.charge / r**2).ravel()[:, np.newaxis] * u_i).reshape(X.
        ↪ shape + (2,))
        return field

    def compute_potential(self, x, y):
        X, Y = np.meshgrid(x, y)
        r = np.sqrt((X - self.pos[0])**2 + (Y - self.pos[1])**2)
        potential = self.charge / r
        return potential

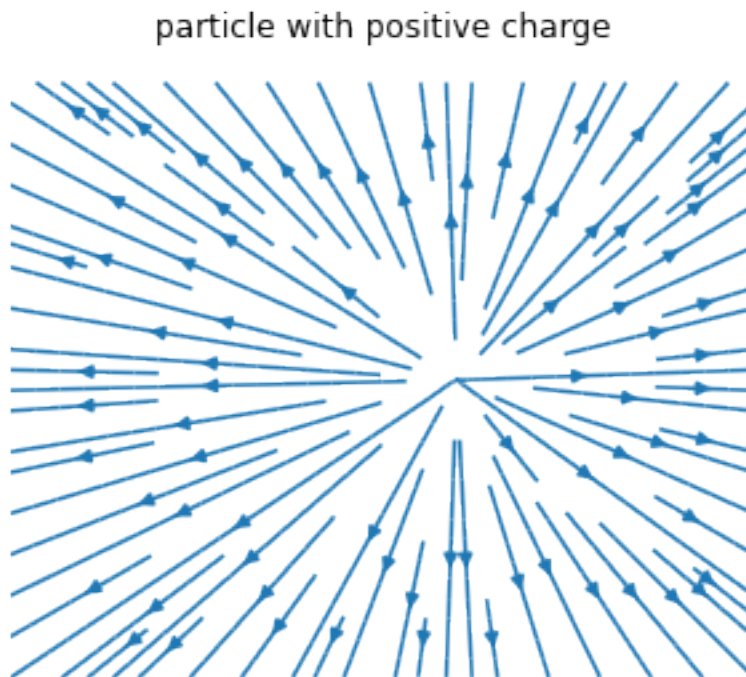
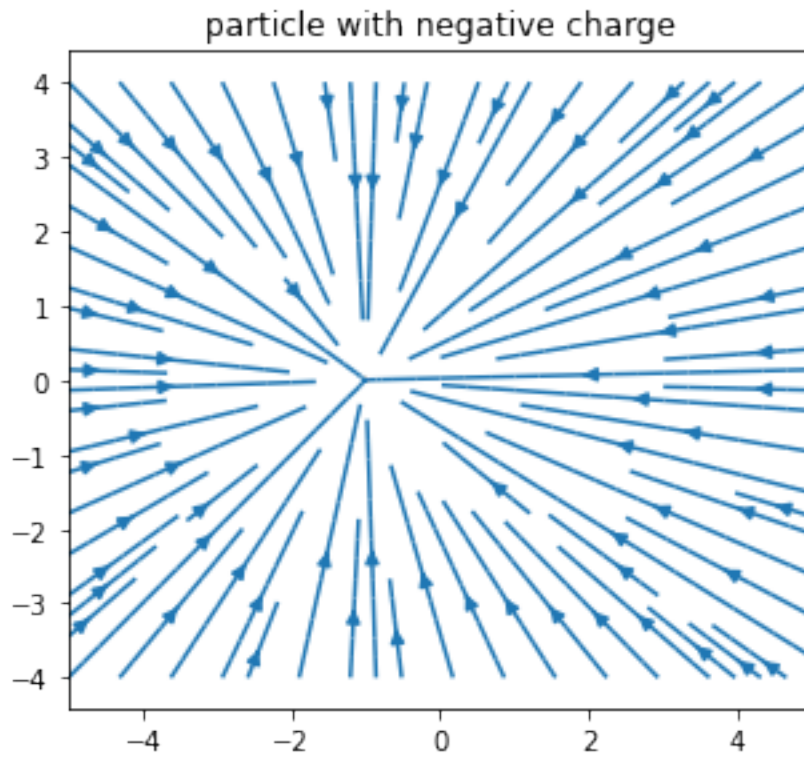
x = np.linspace(-5, 5, 100)
y = np.linspace(-4, 4, 80)

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

q1 = ChargedParticle((-1, 0), -1)
q2 = ChargedParticle((1, 0), 1)

field1 = q1.compute_field(x, y)
field2 = q2.compute_field(x, y)
fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(5, 10))
ax1.streamplot(x, y, u=field1[:, :, 0], v=field1[:, :, 1])
ax1.set_title("particle with negative charge");
ax1.axis('equal')
plt.axis('off')
```

```
ax2.streamplot(x, y, u=field2[:, :, 0], v=field2[:, :, 1])
ax2.set_title("particle with positive charge");
ax2.axis('equal');
plt.axis('off');
```



13.2.2 Electric potential for charge particle:

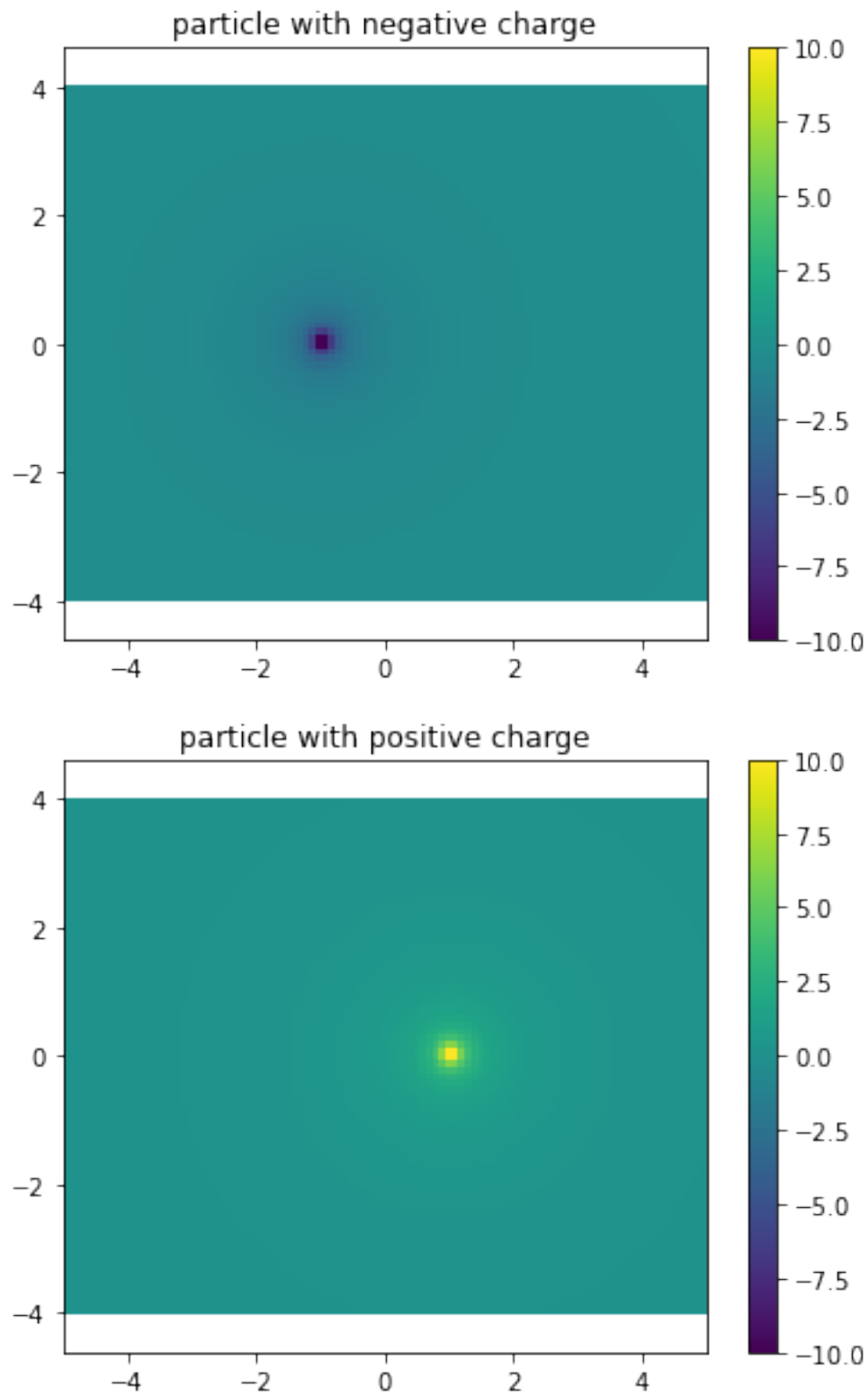
```
[110]: pot1 = q1.compute_potential(x, y)
      pot2 = q2.compute_potential(x, y)

      fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(6, 10))
      map1 = ax1.pcolormesh(x, y, pot1, vmin=-10, vmax=10)
      ax1.set_title("particle with negative charge");
      ax1.axis('equal')
      plt.colorbar(map1, ax=ax1)
      map2 = ax2.pcolormesh(x, y, pot2, vmin=-10, vmax=10)
      ax2.set_title("particle with positive charge");
      ax2.axis('equal');
      plt.colorbar(map2, ax=ax2)
      plt.show()
```

<ipython-input-110-bd3c59012fb4>:5: MatplotlibDeprecationWarning: shading='flat' when X and Y have the same dimensions as C is deprecated since 3.3. Either specify the corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading']. This will become an error two minor releases later.

```
map1 = ax1.pcolormesh(x, y, pot1, vmin=-10, vmax=10)
<ipython-input-110-bd3c59012fb4>:9: MatplotlibDeprecationWarning: shading='flat' when X and Y have the same dimensions as C is deprecated since 3.3. Either specify the corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading']. This will become an error two minor releases later.
```

```
map2 = ax2.pcolormesh(x, y, pot2, vmin=-10, vmax=10)
```

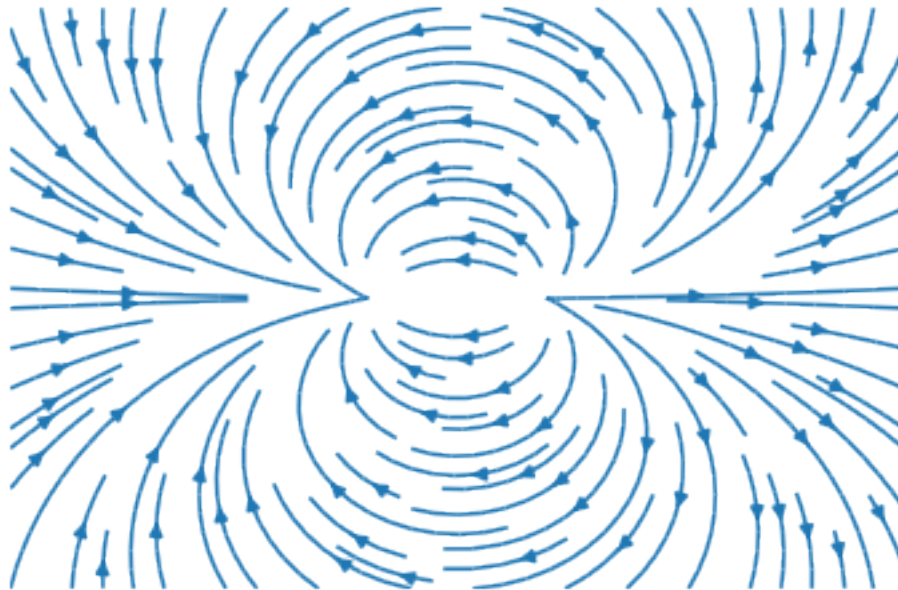



We can now compute the whole field by summing over the individual electric fields.

```
[111]: def compute_resulting_field(particles, x, y):
        fields = [p.compute_field(x, y) for p in particles]
        total_field = np.zeros_like(fields[0])
        for field in fields:
            total_field += field
        return total_field

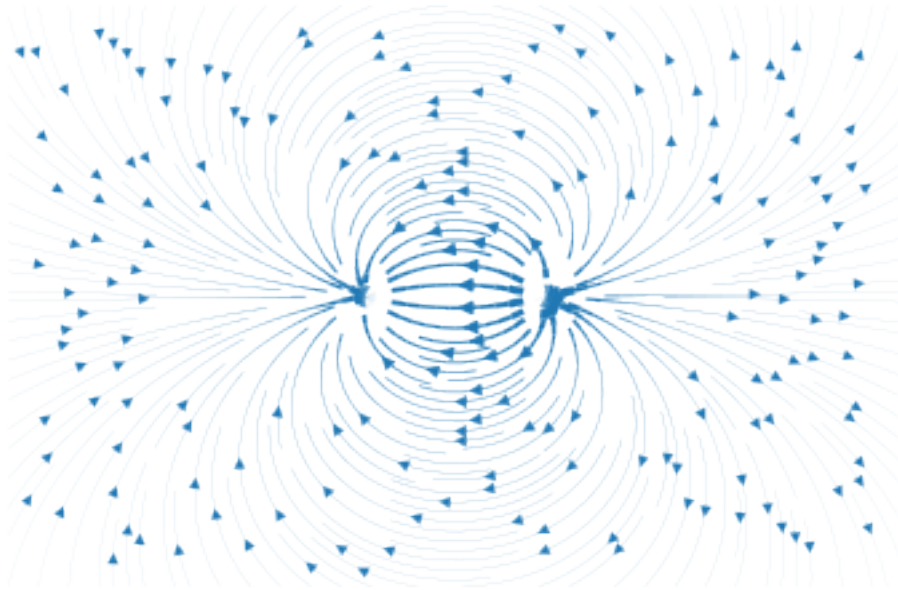
total_field = compute_resulting_field([q1, q2], x, y)

plt.streamplot(x, y, total_field[:, :, 0], total_field[:, :, 1])
plt.xlim(x.min(), x.max())
plt.ylim(y.min(), y.max());
plt.axis('off');
```



We can even explore some options regarding the streamplot.

```
[112]: lw = np.linalg.norm(total_field, axis=2)
        lw /= lw.max()
        plt.streamplot(x, y, total_field[:, :, 0], total_field[:, :, 1],
            ↳ linewidth=10*lw, density=2)
        plt.xlim(x.min(), x.max())
        plt.ylim(y.min(), y.max());
        plt.axis('off');
```



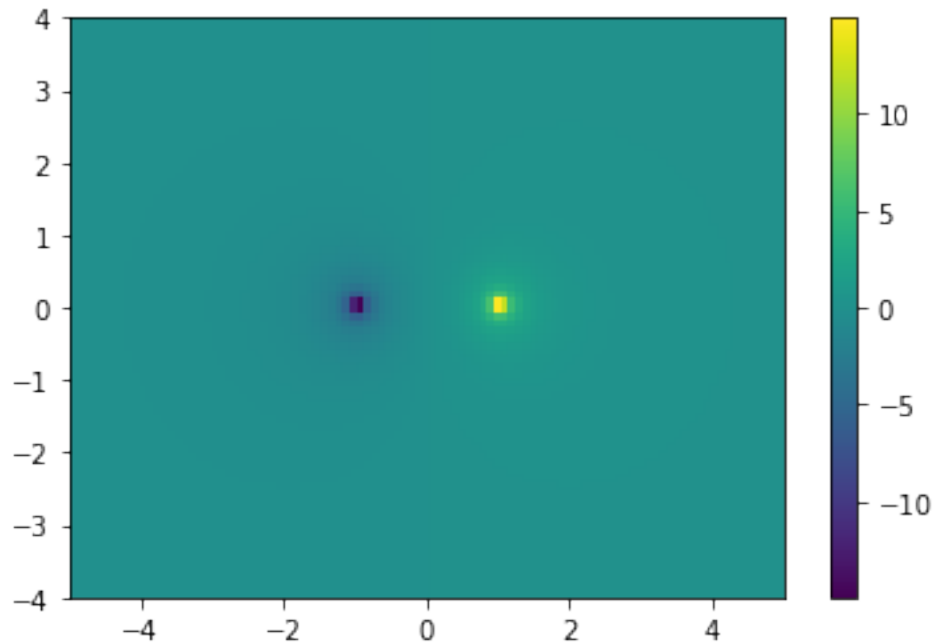
We can also compute the whole potential by summing over the individual electric fields.

```
[113]: def compute_resulting_potential(particles, x, y):
    potentials = [p.compute_potential(x, y) for p in particles]
    total_potential = np.zeros_like(potentials[0])
    for pot in potentials:
        total_potential += pot
    return total_potential

total_potential = compute_resulting_potential([q1, q2], x, y)
plt.pcolormesh(x, y, total_potential)
plt.colorbar();
```

<ipython-input-113-2d6d1c14e0f5>:9: MatplotlibDeprecationWarning: shading='flat' when X and Y have the same dimensions as C is deprecated since 3.3. Either specify the corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading']. This will become an error two minor releases later.

```
plt.pcolormesh(x, y, total_potential)
```



13.2.3 Four charges on square:

```
[114]: q1 = ChargedParticle((1, 1), -1)
q2 = ChargedParticle((-1, 1), 1)
q3 = ChargedParticle((-1, -1), -1)
q4 = ChargedParticle((1, -1), 1)
total_field = compute_resulting_field([q1, q2, q3, q4], x, y)
total_potential = compute_resulting_potential([q1, q2, q3, q4], x, y)

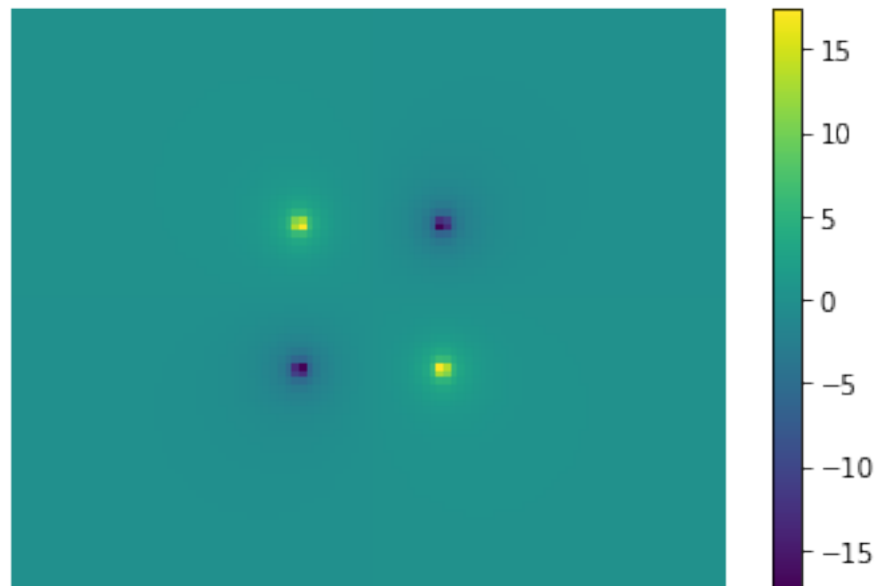
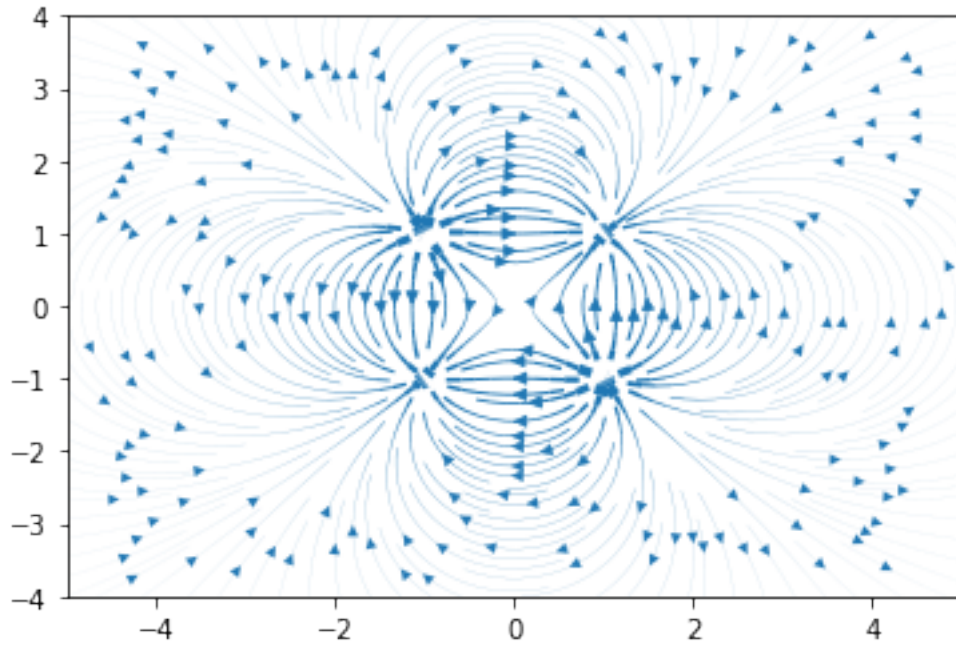
lw = np.linalg.norm(total_field, axis=2)
lw /= lw.max()
fig, ax = plt.subplots()
ax.streamplot(x, y, total_field[:, :, 0], total_field[:, :, 1],
    ↳linewidth=10*lw, density=2)
ax.set_xlim(x.min(), x.max())
ax.set_ylim(y.min(), y.max())

fig, ax = plt.subplots()
mappable = ax.pcolormesh(x, y, total_potential)
plt.colorbar(mappable);
plt.axis('off');
```

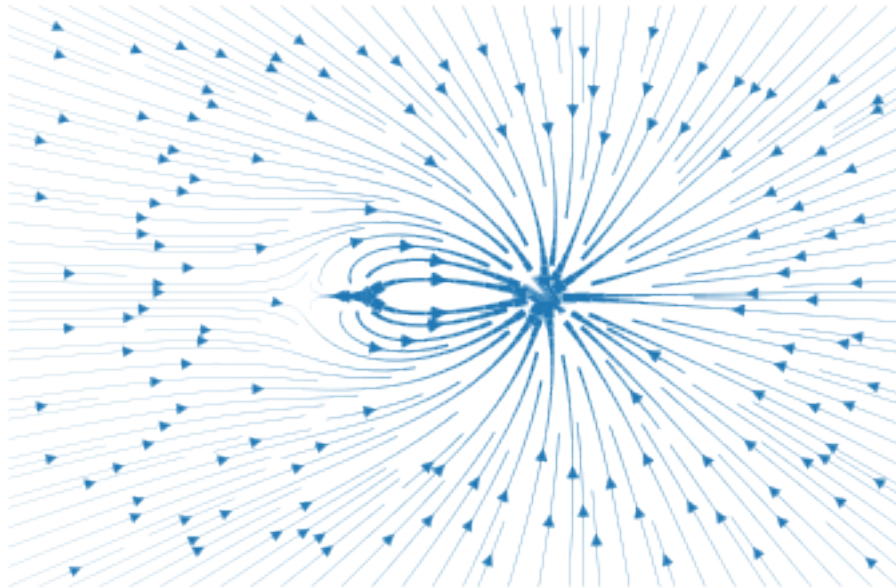
<ipython-input-114-f2aea8e838b7>:16: MatplotlibDeprecationWarning:
shading='flat' when X and Y have the same dimensions as C is deprecated since

3.3. Either specify the corners of the quadrilaterals with X and Y, or pass `shading='auto'`, `'nearest'` or `'gouraud'`, or set `rcParams['pcolor.shading']`. This will become an error two minor releases later.

```
mappable = ax.pcolormesh(x, y, total_potential)
```



```
[115]: q1 = ChargedParticle((1, 0), -4)
q2 = ChargedParticle((-1, 0), 1)
total_field = compute_resulting_field([q1, q2], x, y)
lw = np.linalg.norm(total_field, axis=2)
lw /= lw.max()
plt.streamplot(x, y, total_field[:, :, 0], total_field[:, :, 1],
    ↳linewidth=20*lw, density=2)
plt.xlim(x.min(), x.max())
plt.ylim(y.min(), y.max());
plt.axis('off');
```

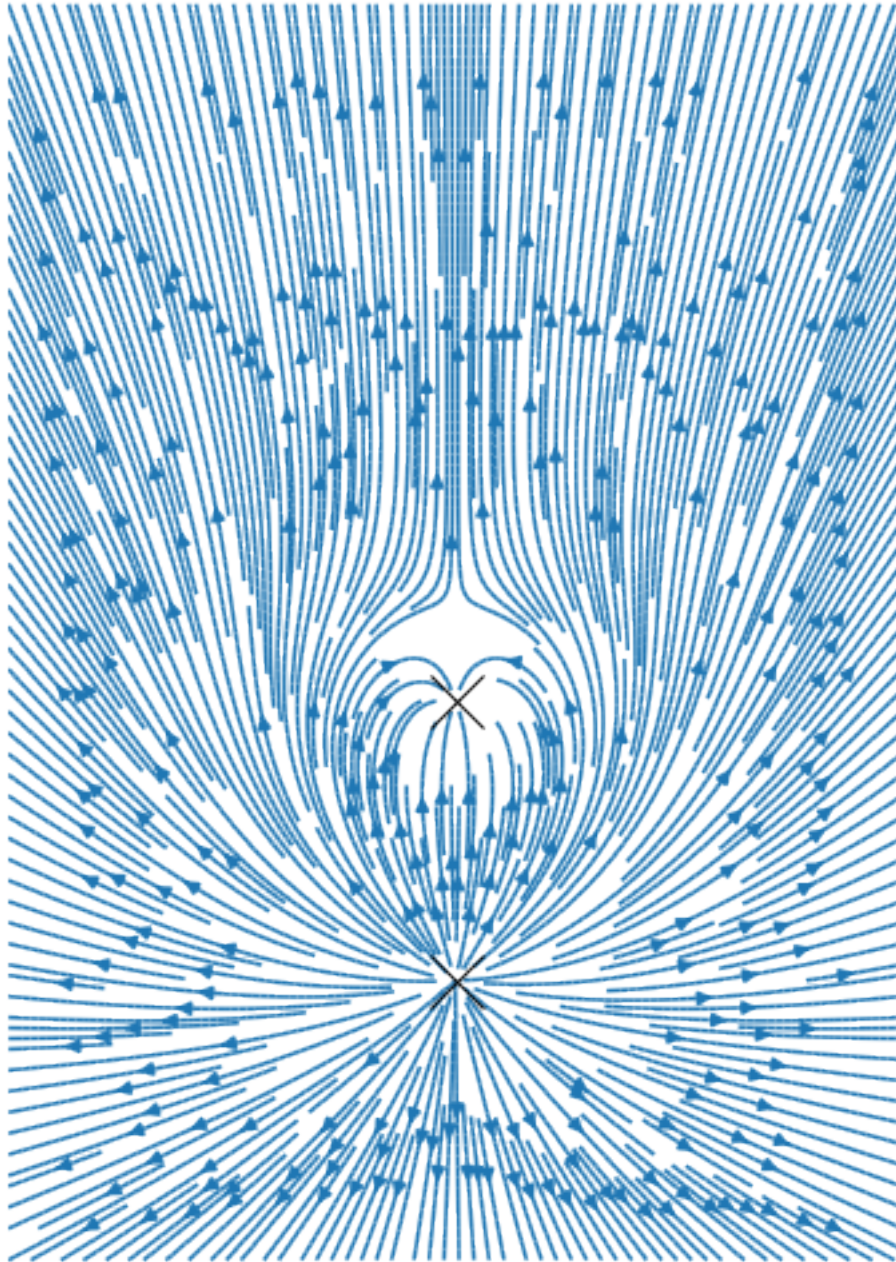


13.2.4 Maxwell's plot:

```
[116]: x = np.linspace(-1.6, 1.6, 100)
y = np.linspace(-1, 3.5, 200)
q1 = ChargedParticle((0, 0), 20)
q2 = ChargedParticle((0, 1), -5)
total_field = compute_resulting_field([q1, q2], x, y)
lw = np.linalg.norm(total_field, axis=2)
lw /= lw.max()
fig = plt.figure(figsize=(6, 9))
plt.plot(*q1.pos, 'kx', ms=20)
plt.plot(*q2.pos, 'kx', ms=20)
plt.streamplot(x, y, total_field[:, :, 0], total_field[:, :, 1], density=4,
    ↳cmap='magma', integration_direction='backward')
```



```
plt.xlim(x.min(), x.max())  
plt.ylim(y.min(), y.max())  
plt.axis('equal')  
plt.axis('off');
```

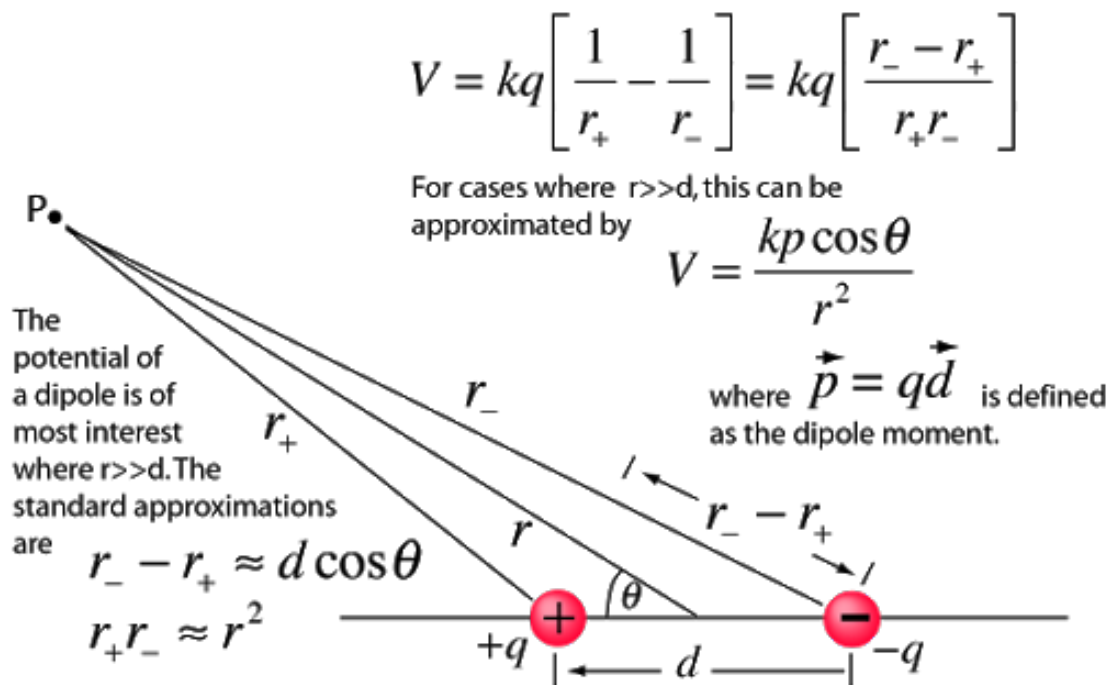


13.3 Electrostatic potential of an electric dipole:

The following code produces a plot of the electrostatic potential of an electric dipole $\vec{p}=(qd,0,0)$ in the (x,y) plane for $q = 1.602 \times 10^{-19} \text{C}$, $d=1 \text{ pm}$ using the point dipole approximation.

```
[117]: from IPython.display import display, Image
        Image(filename="potential.png",width=800)
```

[117]:



```
[118]: import numpy as np
import matplotlib.pyplot as plt

# Dipole charge (C), Permittivity of free space (F.m-1)
q, eps0 = 1.602e-19, 8.854e-12
# Dipole +q, -q distance (m) and a convenient combination of parameters
d = 1.e-12
k = 1/4/np.pi/eps0 * q * d

# Cartesian axis system with origin at the dipole (m)
X = np.linspace(-5e-11, 5e-11, 1000)
Y = X.copy()
X, Y = np.meshgrid(X, Y)

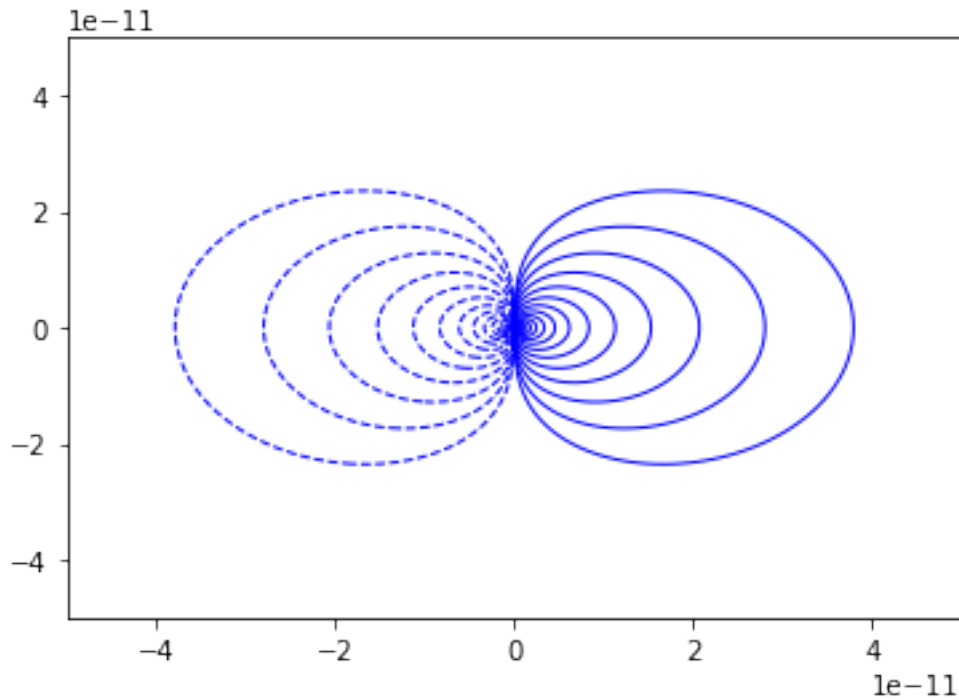
# Dipole electrostatic potential (V), using point dipole approximation
Phi = k * X / np.hypot(X, Y)**3
```



```

fig = plt.figure()
ax = fig.add_subplot(111)
# Draw contours at values of Phi given by levels
levels = np.array([10**pw for pw in np.linspace(0,5,20)])
levels = sorted(list(-levels) + list(levels))
# Monochrome plot of potential
ax.contour(X, Y, Phi, levels=levels, colors='blue', linewidths=1)
plt.show()

```



13.4 Magnetic field of a straight wire:

According to the Biot-Savart law, magnetic field is defined as:

$$\vec{B} = \frac{\mu_0}{4\pi} \frac{I \vec{dl} \times \vec{r}}{r^3}$$

Using Biot-Savart law, we can calculate magnetic field at a distance r:

$$B = \frac{\mu_0 I}{2\pi r}$$

```

[119]: from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-6,6,6)
y = np.linspace(-6,6,6)
z = np.linspace(-6,6,6)

x,y,z = np.meshgrid(x,y,z)

# 3d figure
fig = plt.figure()
ax = fig.gca(projection='3d')

def B(x,y):
    i = 0.5
    mu = 1.26 * 10**(-6)
    mag = (mu/(2*np.pi))*(i/np.sqrt((x)**2+(y)**2))
    by = mag * (np.cos(np.arctan2(y,x)))
    bx = mag * (-np.sin(np.arctan2(y,x)))
    bz = z*0
    return bx,by,bz

def cylinder(r):
    phi = np.linspace(-2*np.pi,2*np.pi,100)
    x = r*np.cos(phi)
    y = r*np.sin(phi)
    return x,y

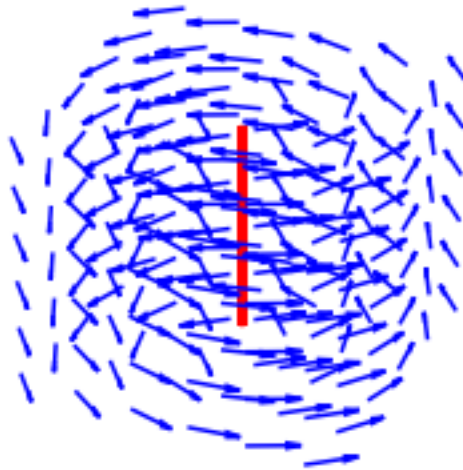
# Plot of the fields
bx,by,bz = B(x,y)
cx,cy = cylinder(0.1)

# Plot of the 3d vector field
ax.quiver(x,y,z,bx,by,bz,color='b',length=2,normalize=True)

for i in np.linspace(-5,5,1000):
    ax.plot(cx,cy,i,label='Cylinder',color='r')

plt.xlabel('x')
plt.ylabel('y')
plt.axis('off');
plt.show()

```



13.5 Problem:

Show how magnetic field change with distance.

13.6 Solution:

```
[120]: import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
z = np.linspace(-10,10,100)

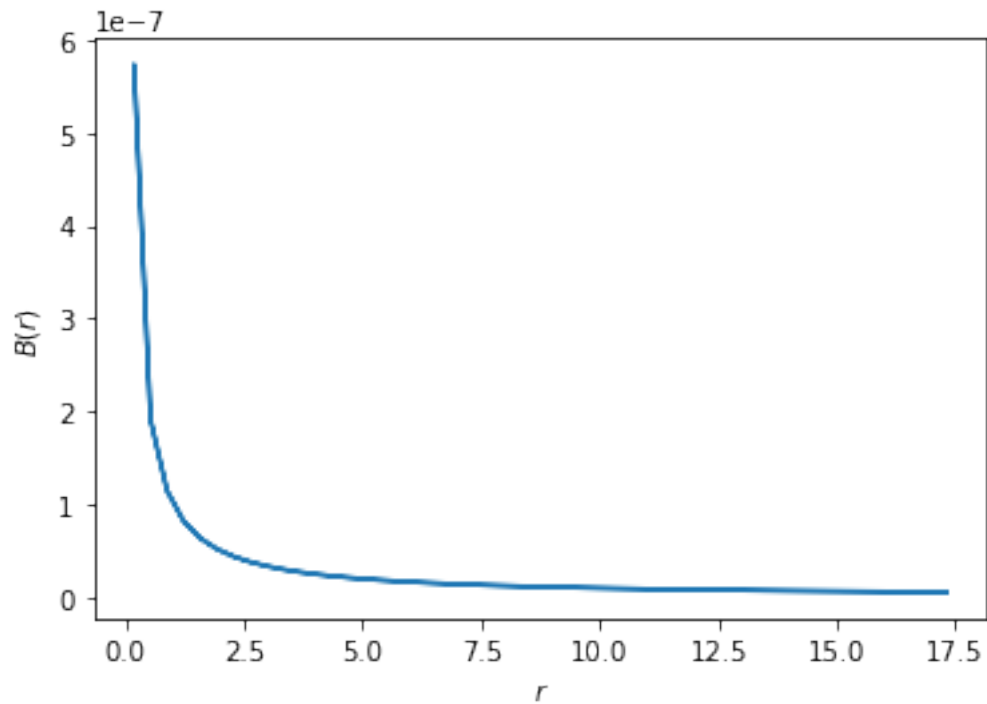
def B(x,y,z):
    i = 0.5 #Amps in the wire
    mu = 1.26 * 10**(-6)
    return (mu/(2*np.pi))*(i/np.sqrt((x)**2+(y)**2+(z)**2))

def r(x,y,z):
    return np.sqrt(x*x+y*y+z*z)

plt.plot(r(x,y,z), B(x,y,z))

plt.xlabel(r"$r$")
```

```
plt.ylabel(r"$B(r)$")
plt.show()
```



13.6.1 Magnetic field produced by a dipole :

As we know , Magnetic field produced by a dipole is:

$$\mathbf{B}(\mathbf{r}) = \frac{\mu_0}{4\pi} \left(\frac{3\mathbf{r}(\mathbf{m} \cdot \mathbf{r})}{r^5} - \frac{\mathbf{m}}{r^3} \right)$$

Where,

$$\mu_0 = \frac{\pi}{2.5 \cdot 10^6} \frac{\text{N}}{\text{A}^2}$$

$$\frac{\mu_0}{4\pi} = \frac{\pi}{4\pi \cdot 2.5 \cdot 10^6} = 10^{-7} \frac{\text{H}}{\text{m}}$$

```
[121]: def dipole(m, r, r0):
        # we use np.subtract to allow r and r0 to be a python lists, not only np.
        ↪ array
        R = np.subtract(np.transpose(r), r0).T
```

```

# assume that the spatial components of r are the outermost axis
norm_R = np.sqrt(np.einsum("i...,i...", R, R))

# calculate the dot product only for the outermost axis,
# that is the spatial components
m_dot_R = np.tensordot(m, R, axes=1)

# tensordot with axes=0 does a general outer product - we want no sum
B = 3 * m_dot_R * R / norm_R**5 - np.tensordot(m, 1 / norm_R**3, axes=0)

# include the physical constant
B *= 1e-7

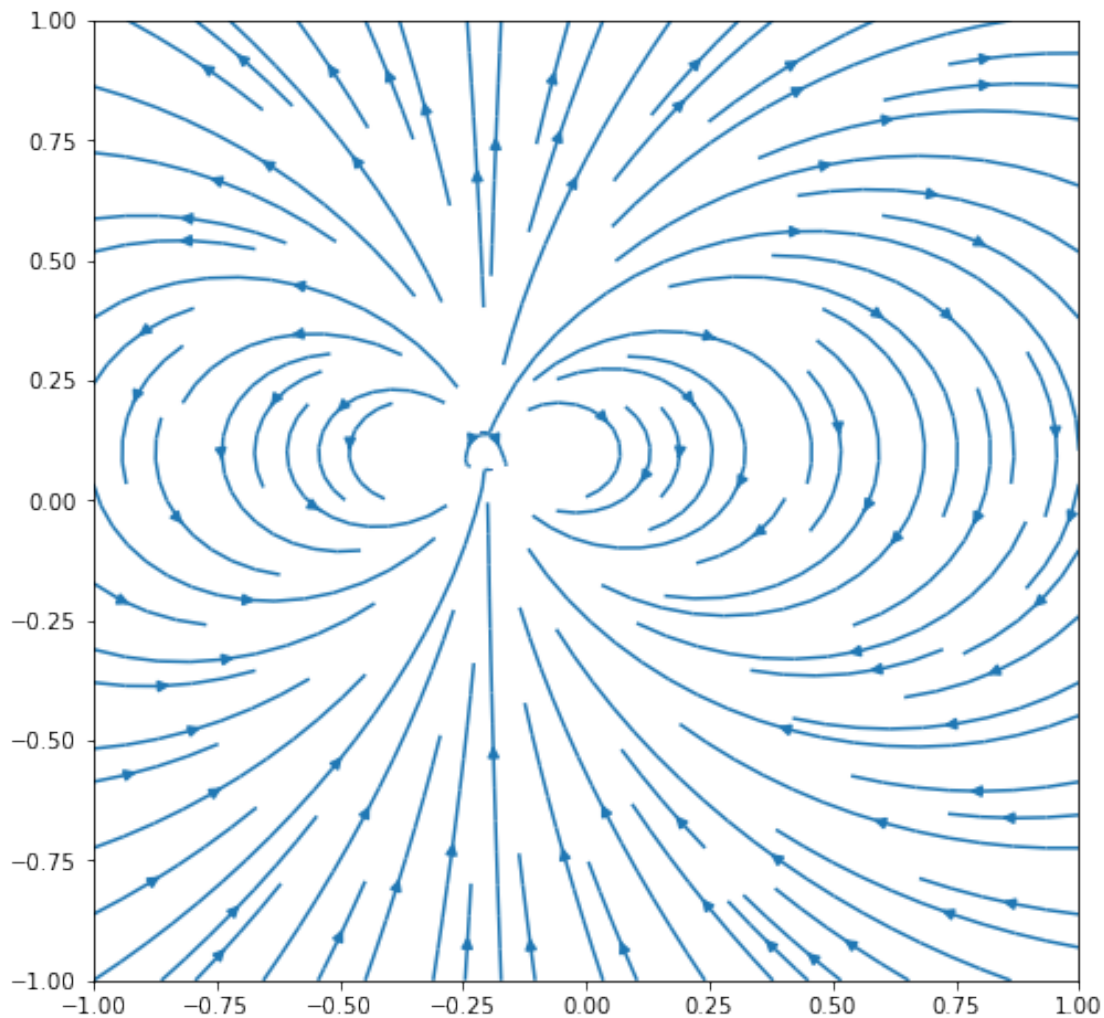
return B

X = np.linspace(-1, 1)
Y = np.linspace(-1, 1)

Bx, By = dipole(m=[0, 0.1], r=np.meshgrid(X, Y), r0=[-0.2,0.1])

plt.figure(figsize=(8, 8))
plt.streamplot(X, Y, Bx, By)
plt.margins(0, 0)

```



13.7 Charged Particle Trajectories in Electric and Magnetic Fields:

First we will discuss motion of charge particle in a constant magnetic field. The equation of motion for a charged particle in a magnetic field is as follows:

$$\frac{d\vec{v}}{dt} = \frac{q}{m}(\vec{v} \times \vec{B})$$

We choose to put the particle in a field that is written:

$$\vec{B} = B\hat{x}$$

We thus expect the particle to rotate in the (y,z) plane while moving along the x axis. Let's check the integration results. We expect a circle in the (y,z) plane.

```

[122]: import numpy as np
from scipy.integrate import ode
%matplotlib inline
import matplotlib.pyplot as plt
def newton(t, Y, q, m, B):

    x, y, z = Y[0], Y[1], Y[2]
    u, v, w = Y[3], Y[4], Y[5]

    alpha = q / m * B
    return np.array([u, v, w, 0, alpha * w, -alpha * v])

r = ode(newton).set_integrator('dopri5')

# Initial conditions
t0 = 0
x0 = np.array([0, 0, 0])
v0 = np.array([1, 1, 0])
initial_conditions = np.concatenate((x0, v0))

#Let's now set the conditions on our integrators and
#solve the problem using time stepping. We assume the charged particle has unit
↪mass and unit charge.

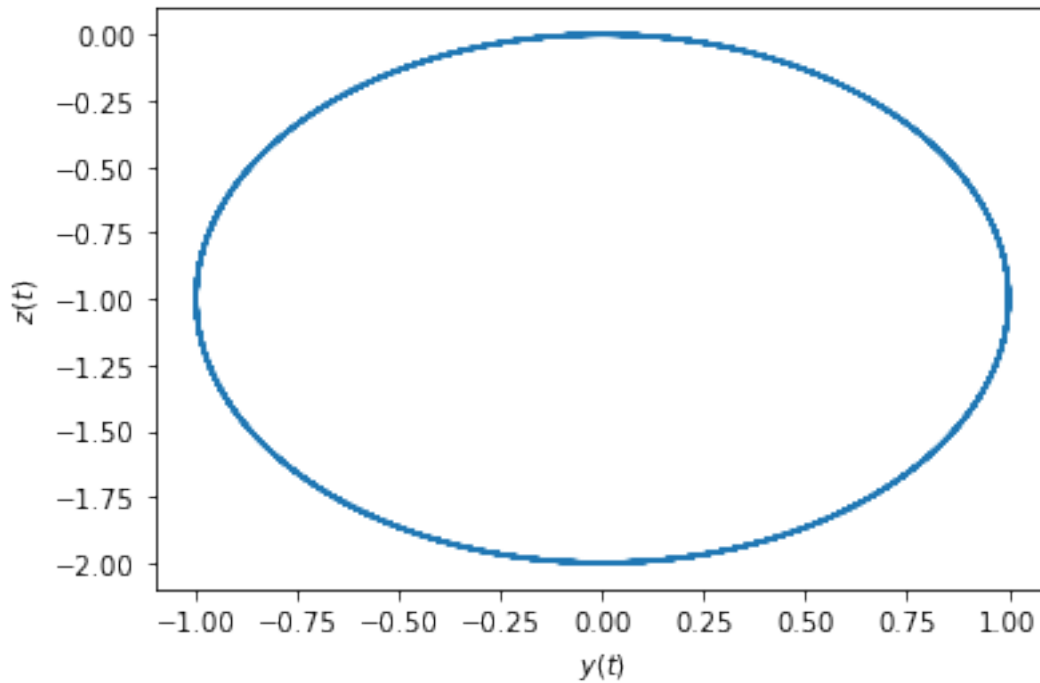
r.set_initial_value(initial_conditions, t0).set_f_params(1.0, 1.0, 1.0)

positions = []
t1 = 50
dt = 0.05
while r.successful() and r.t < t1:
    r.integrate(r.t+dt)
    positions.append(r.y[:3]) # keeping only position, not velocity

positions = np.array(positions)

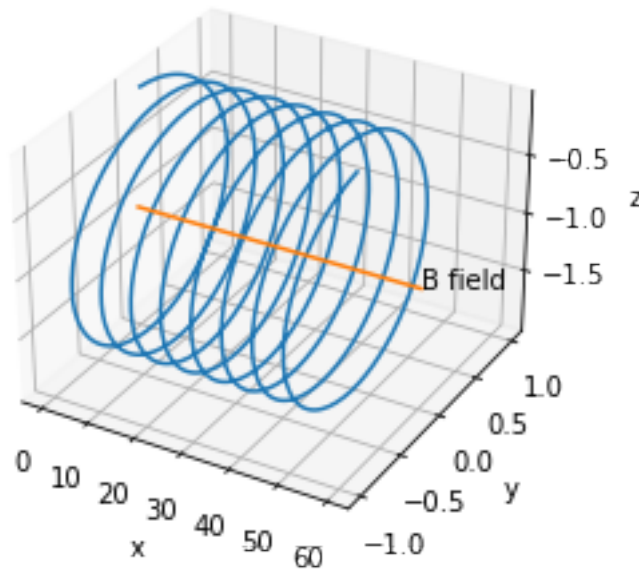
plt.plot(positions[:, 1], positions[:, 2])
plt.xlabel("$y(t)$")
plt.ylabel("$z(t)$")
plt.show()

```



```
[11]: import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot3D(positions[:, 0], positions[:, 1], positions[:, 2])

B1 = np.array([x0[0], x0[1], -1])
B2 = np.array([60, 0, 0])
B_axis = np.vstack((B1, B1 + B2))
ax.plot3D(B_axis[:, 0],
          B_axis[:, 1],
          B_axis[:, 2])
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlabel('z')
ax.text3D((B1 + B2)[0], (B1 + B2)[1], (B1 + B2)[2], "B field");
```

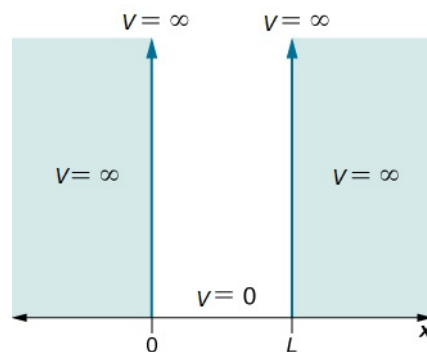
14 Basic Operations on Quantum Objects

14.1 Particle in a box problem:

The particle-in-a-box problem is usefulness in our context is that it illustrates several quantum mechanical features. The potential energy at the barrier is set to infinity (i.e. the particle cannot escape) and the potential energy inside the barrier is set to 0. Under these conditions, classical mechanics predicts that the particle has an equal probability of being in any part of the box and the kinetic energy of the particle is allowed to have any value. Taking this assumption into consideration, we get different equations for the particle's energy at the barrier and inside the box.

[123]: `Image(filename="CNX_UPhysics_40_04_box.jpg",width=700)`

[123]:



If we solve Schrodinger equation the range from (0,L):

$$\frac{\hbar^2}{2m} \frac{d^2(\psi(x))}{dx^2} + (E - 0)\psi(x) = 0$$

Then Energy of nth state:

$$E_n = \frac{n^2 h^2}{8mL^2}$$

and wavefunction of nth state:

$$\psi_n = \sqrt{\frac{2}{L}} \sin\left(\frac{n\pi x}{L}\right)$$

Let's write a code in such a way that, as a input we consider n and L and we will plot wavefunction and corresponding probability density for free particle of mass m inside the box (0, L).

```
[124]: import matplotlib.pyplot as plt
import numpy as np

# Defining the wavefunction
def psi(x,n,L): return np.sqrt(2.0/L)*np.sin(float(n)*np.pi*x/L)

# Reading the input variables from the user
n = int(input("Enter the value for the quantum number n = "))
L = float(input("Enter the size of the box in Angstroms = "))

# Generating the wavefunction graph
plt.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral', 'mathtext.
    ↳fontset': 'stix'})
x = np.linspace(0, L, 900)
fig, ax = plt.subplots()
lim1=np.sqrt(2.0/L) # Maximum value of the wavefunction
ax.axis([0.0,L,-1.1*lim1,1.1*lim1]) # Defining the limits to be plot in the
    ↳graph
str1=r"$n = "+str(n)+r"$"
ax.plot(x, psi(x,n,L), linestyle='--', label=str1, color="orange", linewidth=2.
    ↳8) # Plotting the wavefunction
ax.hlines(0.0, 0.0, L, linewidth=1.8, linestyle='--', color="black") # Adding a
    ↳horizontal line at 0
# Now we define labels, legend, etc
ax.legend(loc=2);
ax.set_xlabel(r'$L$')
ax.set_ylabel(r'$\psi_n(x)$')
plt.title('Wavefunction')
plt.legend(bbox_to_anchor=(1.1, 1), loc=2, borderaxespad=0.0)

# Generating the probability density graph
fig, ax = plt.subplots()
ax.axis([0.0,L,0.0,lim1*lim1*1.1])
str1=r"$n = "+str(n)+r"$"
```

```

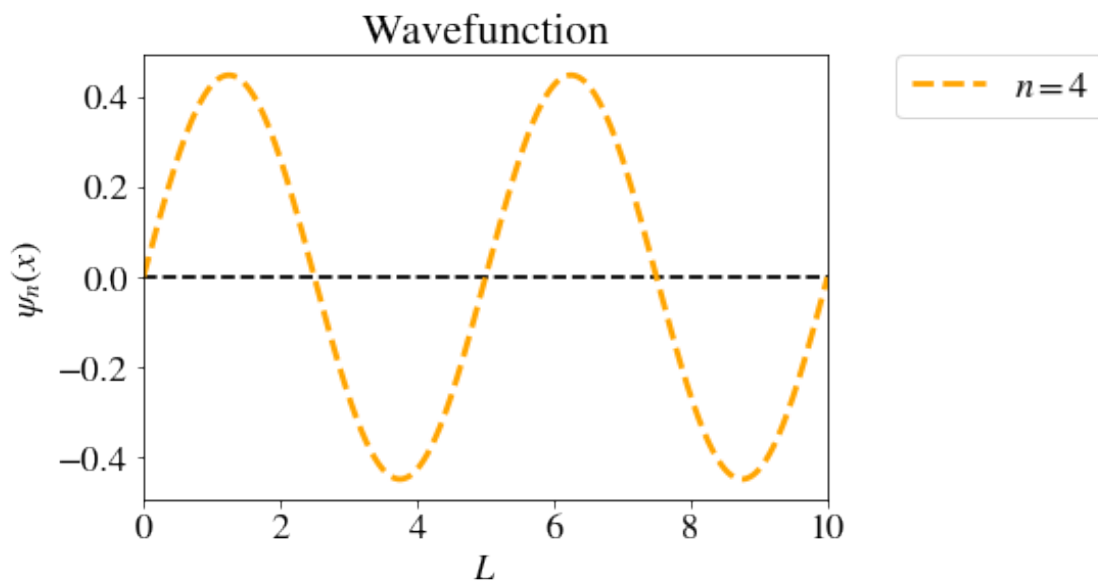
ax.plot(x, psi(x,n,L)*psi(x,n,L), label=str1, linewidth=2.8)
ax.legend(loc=2);
ax.set_xlabel(r'$L$')
ax.set_ylabel(r'$|\psi_n|^2(x)$')
plt.title('Probability Density')
plt.legend(bbox_to_anchor=(1.1, 1), loc=2, borderaxespad=0.0)

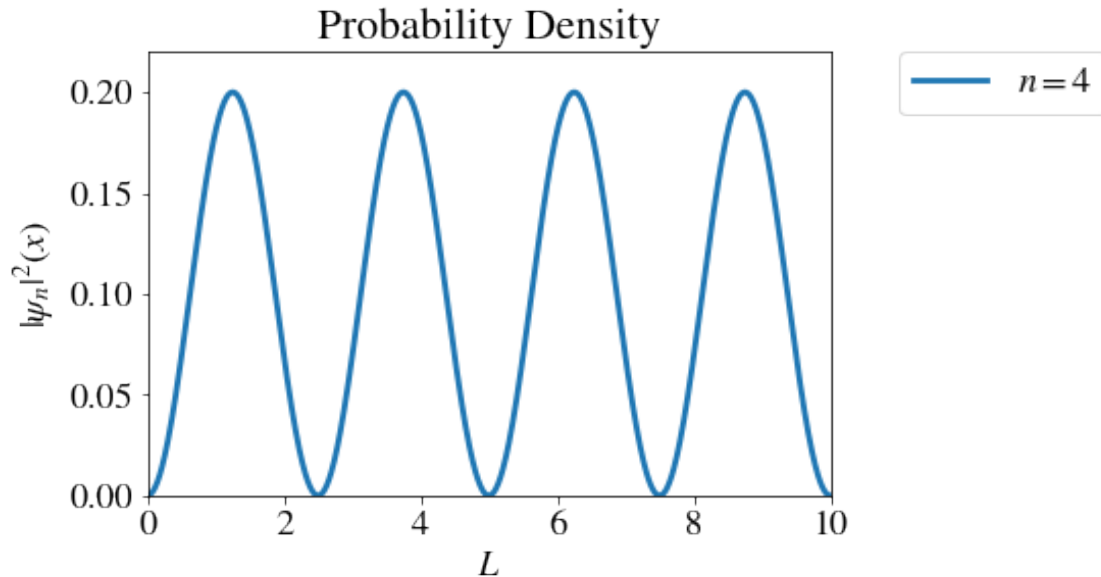
# Show the plots on the screen once the code reaches this point
plt.show()

```

Enter the value for the quantum number $n = 4$

Enter the size of the box in Angstroms = 10





14.2 Problem:

Write a python script for particle in box problem to show the changes in the Wavefunction and Probability Density for a given state n in boxes of different length L . Also consider length of the box should not larger than 20 Å.

14.3 Solution:

```
[125]: import matplotlib.pyplot as plt
import numpy as np

# Reading the input boxes sizes from the user, and making sure the values are
↳not larger than 20 Å
L = 100.0
while(L>20.0):
    L1 = float(input("Enter the value of L for the first box (in Angstroms and
↳not larger then 20 Å) = "))
    L2 = float(input("Enter the value of L for the second box (in Angstroms and
↳not larger then 20) = "))
    L = max(L1,L2)
    if(L>20.0):
        print ("The sizes of the boxes cannot be larger than 20 Å. Please enter
↳the values again.\n")

# Generating the wavefunction and probability density graphs
```

```

plt.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral', 'mathtext.
    ↳fontset': 'stix'})
fig, ax = plt.subplots(figsize=(12,6))
ax.spines['right'].set_color('none')
ax.xaxis.tick_bottom()
ax.spines['left'].set_color('none')
ax.axes.get_yaxis().set_visible(False)
ax.spines['top'].set_color('none')

val = 1.1*max(L1,L2)
X1 = np.linspace(0.0, L1, 900,endpoint=True)
X2 = np.linspace(0.0, L2, 900,endpoint=True)
ax.axis([-0.5*val,1.5*val,-np.sqrt(2.0/L),3*np.sqrt(2.0/L)])
ax.set_xlabel(r'$X$ (Angstroms)')
strA="$\psi_n$"
strB="$|\psi_n|^2$"
ax.text(-0.12*val, 0.0, strA, rotation='vertical', fontsize=30, color="black")
ax.text(-0.12*val, np.sqrt(4.0/L), strB, rotation='vertical', fontsize=30,
    ↳color="black")
str1=r"$L = "+str(L1)+r"$ A"
str2=r"$L = "+str(L2)+r"$ A"
ax.plot(X1,psi(X1,n,L1)*np.sqrt(L1/L), color="red", label=str1, linewidth=2.8)
ax.plot(X2,psi(X2,n,L2)*np.sqrt(L2/L), color="blue", label=str2, linewidth=2.8)
ax.plot(X1,psi(X1,n,L1)*psi(X1,n,L1)*(L1/L) + np.sqrt(4.0/L), color="red",
    ↳linewidth=2.8)
ax.plot(X2,psi(X2,n,L2)*psi(X2,n,L2)*(L2/L) + np.sqrt(4.0/L), color="blue",
    ↳linewidth=2.8)
ax.margins(0.00)
ax.legend(loc=9)
str2="$V = +\infty$"
ax.text(-0.3*val, 0.5*np.sqrt(2.0/L), str2, rotation='vertical', fontsize=40,
    ↳color="black")
ax.vlines(0.0, -np.sqrt(2.0/L), 2.5*np.sqrt(2.0/L), linewidth=4.8, color="red")
ax.vlines(L1, -np.sqrt(2.0/L), 2.5*np.sqrt(2.0/L), linewidth=4.8, color="red")
ax.vlines(0.0, -np.sqrt(2.0/L), 2.5*np.sqrt(2.0/L), linewidth=4.8, color="blue")
ax.vlines(L2, -np.sqrt(2.0/L), 2.5*np.sqrt(2.0/L), linewidth=4.8, color="blue")
ax.hlines(0.0, 0.0, L, linewidth=1.8, linestyle='--', color="black")
ax.hlines(np.sqrt(4.0/L), 0.0, L, linewidth=1.8, linestyle='--', color="black")
plt.title('Wavefunction and Probability Density', fontsize=30)
str3=r"$n = "+str(n)+r"$"
ax.text(1.1*L,np.sqrt(4.0/L), r"$n = "+str(n)+r"$", fontsize=25, color="black")
plt.legend(bbox_to_anchor=(0.73, 0.95), loc=2, borderaxespad=0.)

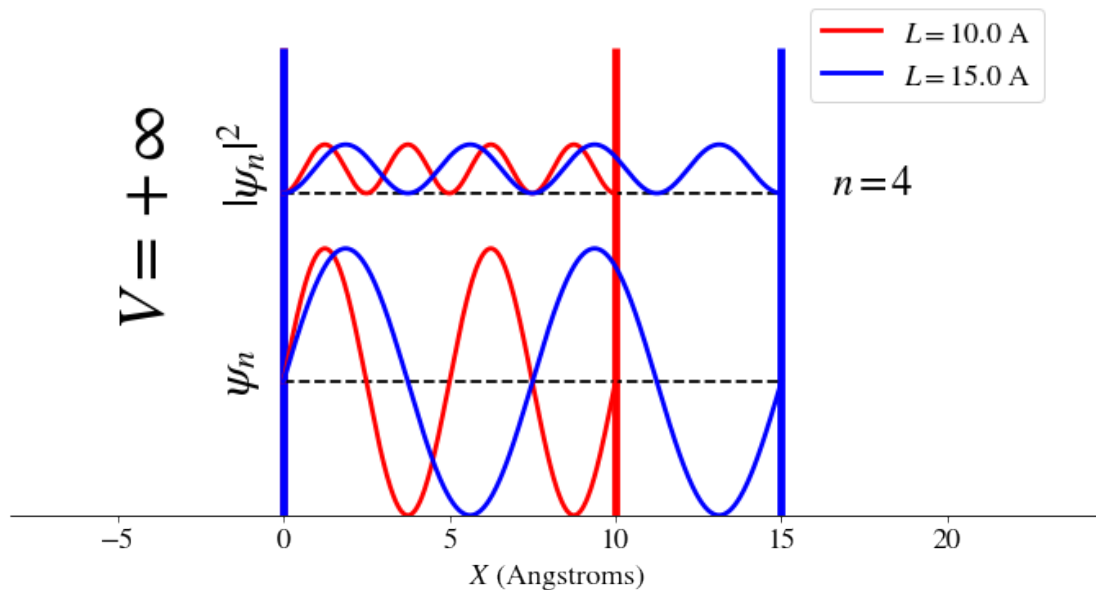
# Show the plots on the screen once the code reaches this point
plt.show()

```

Enter the value of L for the first box (in Angstroms and not larger than 20 A)
= 10

Enter the value of L for the second box (in Angstroms and not larger than 20) =
15

Wavefunction and Probability Density



14.4 Problem:

Another very interesting problem to check how the Energy Levels E_n for an electron change as a function of the size of the box.

(Hints: You can calculate quantize energy of an electron, considering electron is inside two different box of length L_1 and L_2)

14.5 Solution:

```
[126]: import matplotlib.pyplot as plt
h=6.62607e-34
me=9.1093837e-31

def En(n,L,m): return (h**2 / (m*8))* (1e10)**2 *6.242e+18*((float(n)/L)**2)

L1 = float(input("Enter the value for L for the first box (in Angstroms) = "))
nmax1 = int(input("Enter the number of levels you want to plot for the first_
↪box = "))
```

```

L2 = float(input("Enter the value for L for the second box (in Angstroms) = "))
nmax2 = int(input("Enter the number of levels you want to plot for the second_
↳box = "))

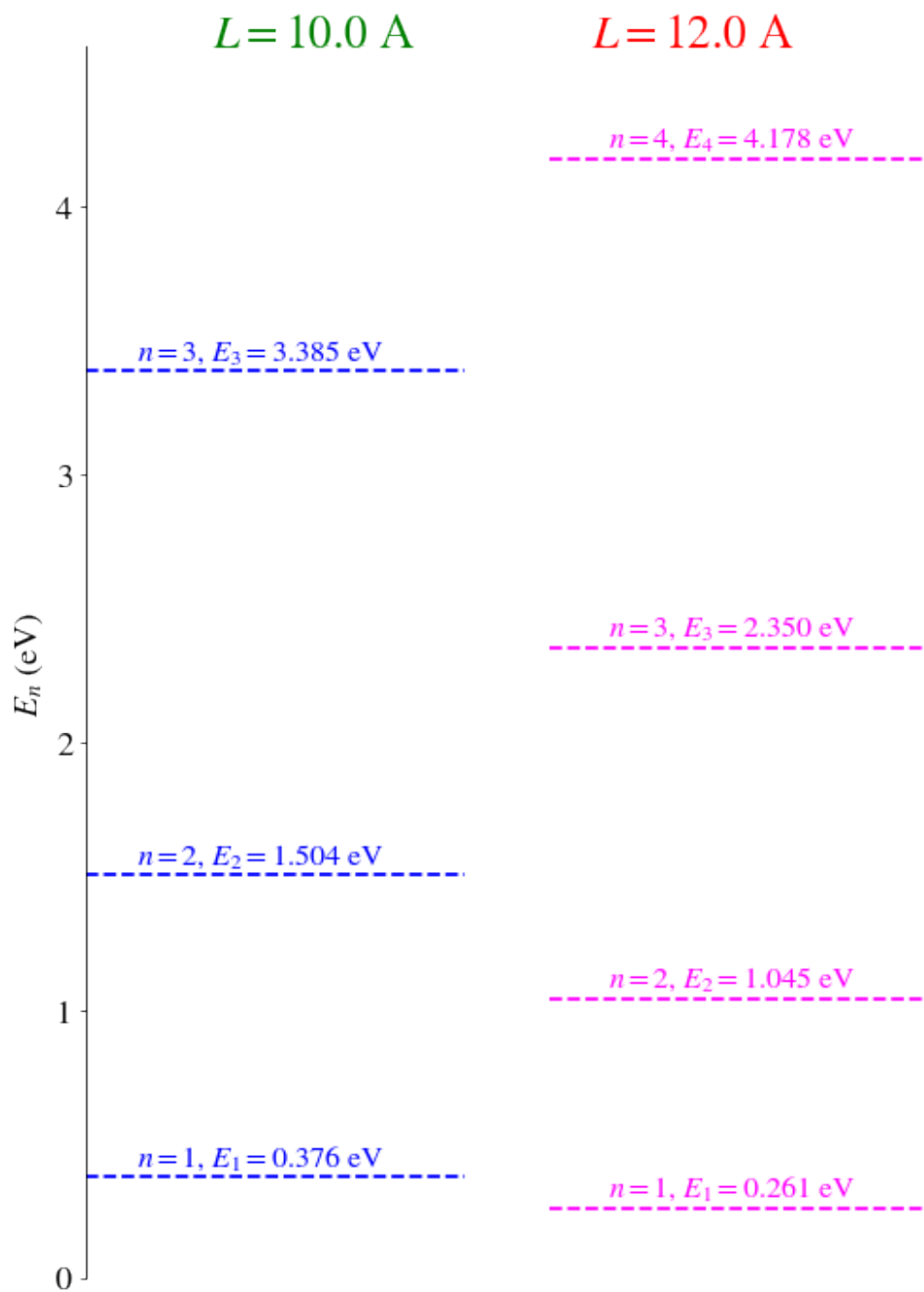
# Beautiful way to represnt energy eigen values
plt.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral', 'mathtext.
↳fontset': 'stix'})
fig, ax = plt.subplots(figsize=(8,12))
ax.spines['right'].set_color('none')
ax.yaxis.tick_left()
ax.spines['bottom'].set_color('none')
ax.axes.get_xaxis().set_visible(False)
ax.spines['top'].set_color('none')
val = 1.1*max(En(nmax1,L1,me),En(nmax2,L2,me))
val2= 1.1*max(L1,L2)
ax.axis([0.0,10.0,0.0,val])
ax.set_ylabel(r'$E_n$ (eV)')
for n in range(1,nmax1+1):
    str1="$n = "+str(n)+r"$, $E_{"+str(n)+r"} = %.3f$ eV"%(En(n,L1,me))
    ax.text(0.6, En(n,L1,me)+0.01*val, str1, fontsize=16, color="blue")
    ax.hlines(En(n,L1,me), 0.0, 4.5, linewidth=1.8, linestyle='--',
↳color="blue")
for n in range(1,nmax2+1):
    str1="$n = "+str(n)+r"$, $E_{"+str(n)+r"} = %.3f$ eV"%(En(n,L2,me))
    ax.text(6.2, En(n,L2,me)+0.01*val, str1, fontsize=16, color="magenta")
    ax.hlines(En(n,L2,me), 5.5, 10.0, linewidth=1.8, linestyle='--',
↳color="magenta")
str1=r"$L = "+str(L1)+r"$ A"
plt.title("Energy Levels for a particle of mass = $m_{electron}$ \n ",
↳fontsize=30)
str1=r"$L = "+str(L1)+r"$ A"
str2=r"$L = "+str(L2)+r"$ A"
ax.text(1.5,val, str1, fontsize=25, color="green")
ax.text(6,val, str2, fontsize=25, color="red")

plt.show()

```

Enter the value for L for the first box (in Angstroms) = 10
 Enter the number of levels you want to plot for the first box = 3
 Enter the value for L for the second box (in Angstroms) = 12
 Enter the number of levels you want to plot for the second box = 4

Energy Levels for a particle of mass = $m_{electron}$



Isn't it looks very cool! Now you can play with it more. You can also see the energy eigen value formula, where E_n is function of m . Let's do another interesting problem.

14.6 Problem:

Show how the Energy Levels, E_n change as a function of the mass of the particle.

14.7 Solution:

```
[127]: import matplotlib.pyplot as plt

L = float(input("Enter the value for L for both boxes (in Angstroms) = "))
m1 = float(input("Enter mass of first particle (the mass of 1 electron) = "))
nmax1 = int(input("Enter n_{max1} = "))
m2 = float(input("Enter mass of second particle (the mass of 1 electron) = "))
nmax2 = int(input("Enter n_{max2} = "))

# Beautiful way to represent Energy eigen valuse
plt.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral', 'mathtext.
    ↪fontset': 'stix'})
fig, ax = plt.subplots(figsize=(8,12))
ax.spines['right'].set_color('none')
ax.yaxis.tick_left()
ax.spines['bottom'].set_color('none')
ax.axes.get_xaxis().set_visible(False)
ax.spines['top'].set_color('none')
val = 1.1*max(En(nmax1,L,m1*me),En(nmax2,L,m2*me))
val2= 1.1*max(m1,m2)
ax.axis([0.0,10.0,0.0,val])
ax.set_ylabel(r'$E_n$ (eV)')
for n in range(1,nmax1+1):
    str1="$n = "+str(n)+r"$, $E_{"+str(n)+r"} = %.3f$ eV"%(En(n,L,m1*me))
    ax.text(0.6, En(n,L,m1*me)+0.01*val, str1, fontsize=16, color="blue")
    ax.hlines(En(n,L,m1*me), 0.0, 4.5, linewidth=1.8, linestyle='--',
    ↪color="blue")
for n in range(1,nmax2+1):
    str1="$n = "+str(n)+r"$, $E_{"+str(n)+r"} = %.3f$ eV"%(En(n,L,m2*me))
    ax.text(6.2, En(n,L,m2*me)+0.01*val, str1, fontsize=16, color="green")
    ax.hlines(En(n,L,m2*me), 5.5, 10.0, linewidth=1.8, linestyle='--',
    ↪color="green")
str1=r"$m = "+str(m1)+r"$ A"
plt.title("Energy Levels for two particles with different masses\n ",
    ↪fontsize=30)
str1=r"$m_1 = "+str(m1)+r"$ $m_e$ "
str2=r"$m_2 = "+str(m2)+r"$ $m_e$ "
ax.text(1.1,val, str1, fontsize=25, color="red")
ax.text(6.5,val, str2, fontsize=25, color="magenta")

# Show the plots on the screen once the code reaches this point
```

```
plt.show()
```

Enter the value for L for both boxes (in Angstroms) = 10

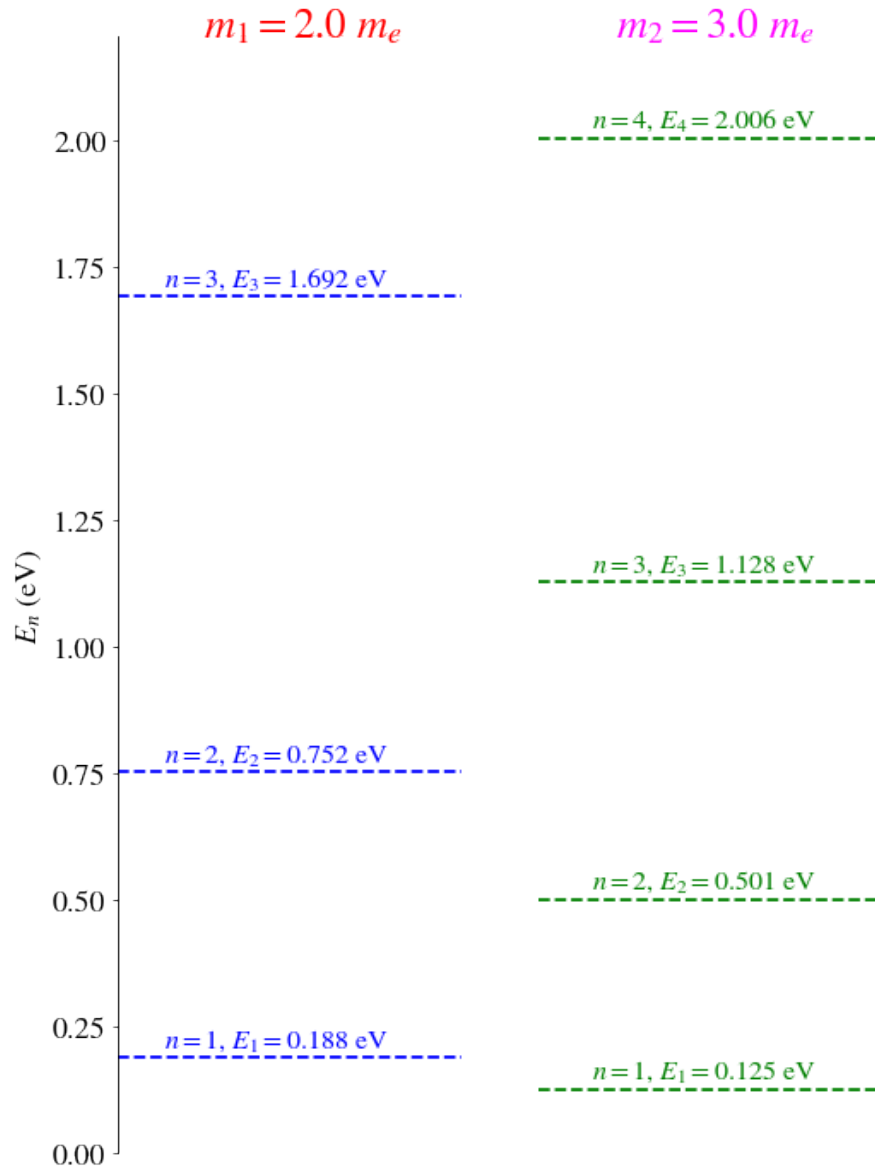
Enter mass of first particle (the mass of 1 electron) = 2

Enter n_{max1} = 3

Enter mass of second particle (the mass of 1 electron) = 3

Enter n_{max2} = 4

Energy Levels for two particles with different masses



14.8 Combined presentation of Energy Levels, Wavefunctions and Probability Densities:

We can combine the information from the wavefunctions, probability density, and energies into a single plot that compares the wavefunctions and the probability densities for different states, each one represented at its energy value. These plots are made using the electron mass.

```
[128]: import matplotlib.pyplot as plt
import numpy as np

# Here the users inputs the value of L
L = float(input("Enter the value of L (in Angstroms) = "))
nmax = int(input("Enter the maximum value of n you want to plot = "))

# Generating the wavefunction graph
fig, ax = plt.subplots(figsize=(12,9))
ax.spines['right'].set_color('none')
ax.xaxis.tick_bottom()
ax.spines['left'].set_color('none')
ax.axes.get_yaxis().set_visible(False)
ax.spines['top'].set_color('none')
X3 = np.linspace(0.0, L, 900, endpoint=True)
Emax = En(nmax, L, me)
amp = (En(2, L, me) - En(1, L, me)) * 0.9
Etop = (Emax + amp) * 1.1
ax.axis([-0.5*L, 1.5*L, 0.0, Etop])
ax.set_xlabel(r'$X$ (Angstroms)')

for n in range(1, nmax+1):
    ax.hlines(En(n, L, me), 0.0, L, linewidth=1.8, linestyle='--', color="black")
    str1 = "$n = " + str(n) + r"$, $E_{" + str(n) + r"} = %.3f$ eV"%(En(n, L, me))
    ax.text(1.03*L, En(n, L, me), str1, fontsize=16, color="black")
    ax.plot(X3, En(n, L, me) + amp*np.sqrt(L/2.0)*psi(X3, n, L), color="red",
    ↪label="", linewidth=2.8)
ax.margins(0.00)
ax.vlines(0.0, 0.0, Etop, linewidth=4.8, color="blue")
ax.vlines(L, 0.0, Etop, linewidth=4.8, color="blue")
ax.hlines(0.0, 0.0, L, linewidth=4.8, color="blue")
plt.title('Wavefunctions', fontsize=30)
plt.legend(bbox_to_anchor=(0.8, 1), loc=2, borderaxespad=0.)
str2 = "$V = +\infty$"
ax.text(-0.15*L, 0.6*Emax, str2, rotation='vertical', fontsize=40,
    ↪color="black")

# Generating the probability density graph
fig, ax = plt.subplots(figsize=(12,9))
ax.spines['right'].set_color('none')
```

```

ax.xaxis.tick_bottom()
ax.spines['left'].set_color('none')
ax.axes.get_yaxis().set_visible(False)
ax.spines['top'].set_color('none')
X3 = np.linspace(0.0, L, 900, endpoint=True)
Emax = En(nmax,L,me)
ax.axis([-0.5*L,1.5*L,0.0,Etop])
ax.set_xlabel(r'$X$ (Angstroms)')
for n in range(1,nmax+1):
    ax.hlines(En(n,L,me), 0.0, L, linewidth=1.8, linestyle='--', color="black")
    str1="$n = "+str(n)+r"$, $E_{"+str(n)+r"} = %.3f$ eV"%(En(n,L,me))
    ax.text(1.03*L, En(n,L,me), str1, fontsize=16, color="black")
    ax.plot(X3,En(n,L,me)+ amp*(np.sqrt(L/2.0)*psi(X3,n,L))**2, color="red",
    ↪label="", linewidth=2.8)
ax.margins(0.00)
ax.vlines(0.0, 0.0, Etop, linewidth=4.8, color="blue")
ax.vlines(L, 0.0, Etop, linewidth=4.8, color="blue")
ax.hlines(0.0, 0.0, L, linewidth=4.8, color="blue")
plt.title('Probability Density', fontsize=30)
plt.legend(bbox_to_anchor=(0.8, 1), loc=2, borderaxespad=0.)
str2="$V = +\infty$"
ax.text(-0.15*L, 0.6*Emax, str2, rotation='vertical', fontsize=40,
    ↪color="black")

# Show the plots on the screen once the code reaches this point
plt.show()

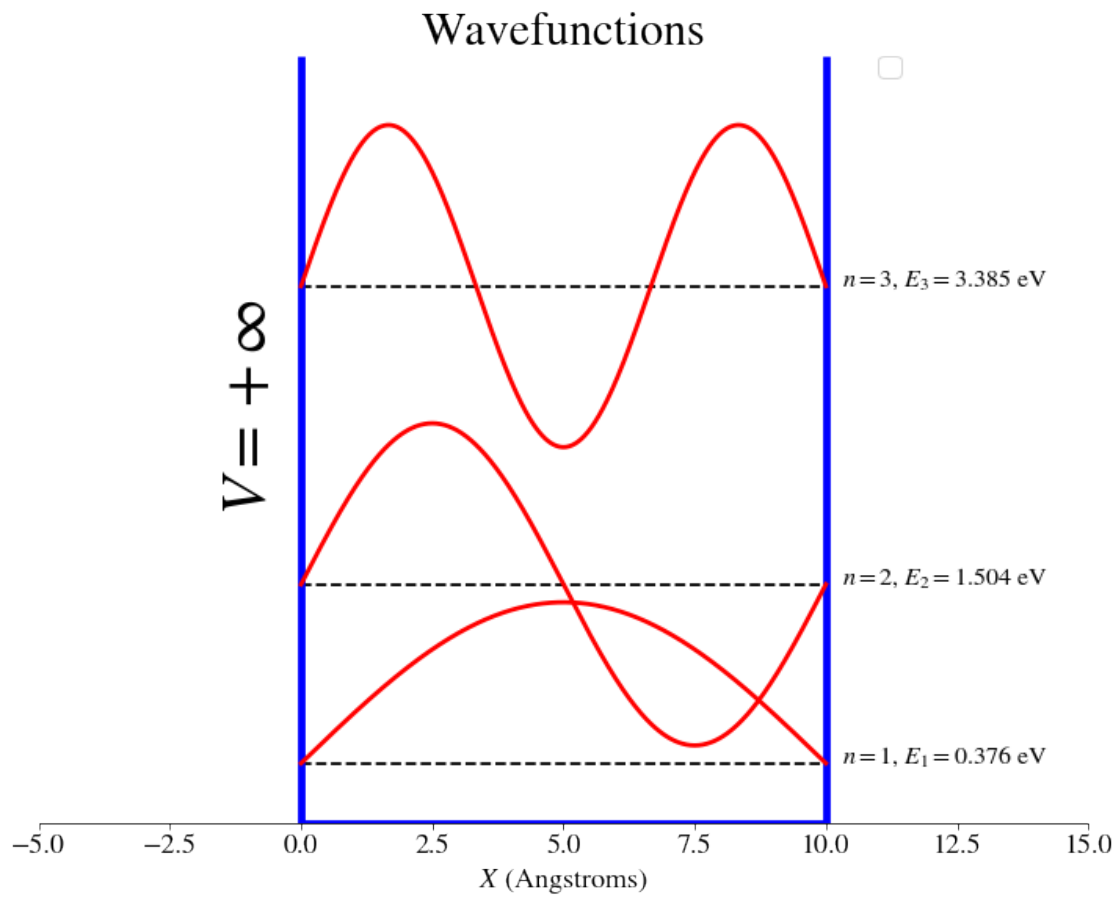
```

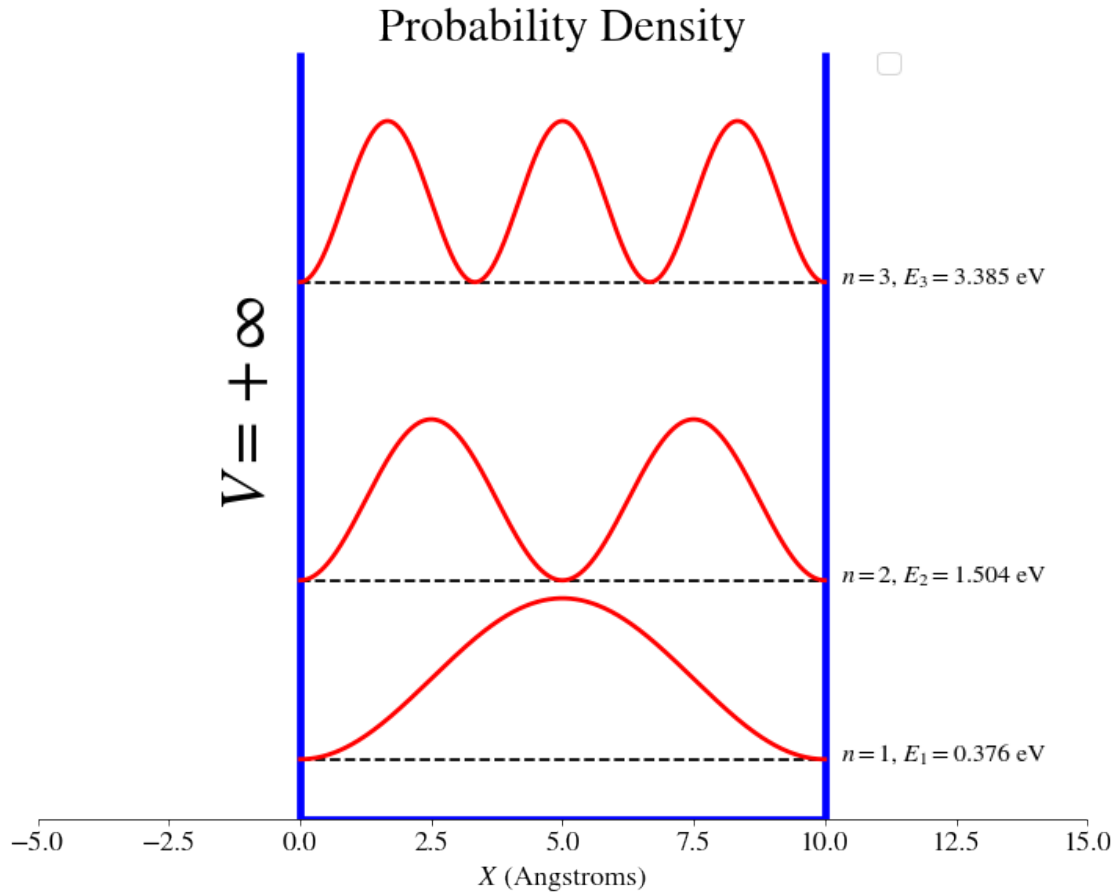
Enter the value of L (in Angstroms) = 10

Enter the maximum value of n you want to plot = 3

No handles with labels found to put in legend.

No handles with labels found to put in legend.





14.9 Particle in 2D box:

If we consider a particle of mass m is inside a 2D box, depending only on the variable x and one depending only on the variable y , the solution to the 2D Schrödinger equation will be a wavefunction that is the product of the 1D solutions in the x and y directions with independent quantum numbers n and m :

$$\psi_{n,m}(x, y) = \psi_n(x)\psi_m(y)$$

or

$$\psi_{n,m}(x, y) = \sqrt{\frac{2}{L_x L_y}} \sin\left(\frac{n\pi x}{L_x}\right) \sin\left(\frac{m\pi y}{L_y}\right)$$

```
[129]: import matplotlib.pyplot as plt
import numpy as np

# Defining the wavefunction
```

```

def psi2D(x,y): return 2.0*np.sin(n*np.pi*x)*np.sin(m*np.pi*y)

# Here the users inputs the values of n and m
n = int(input("Let's look at the Wavefunction for a 2D box \nEnter the value_
↳for n = "))
m = int(input("Enter the value for m = "))

# Generating the wavefunction graph
x = np.linspace(0, 1, 100)
y = np.linspace(0, 1, 100)
X, Y = np.meshgrid(x, y)
fig, axes = plt.subplots(1, 1, figsize=(8,8))
axes.imshow(psi2D(X,Y), origin='lower', extent=[0.0, 1.0, 0.0, 1.0])
axes.set_title(r'plot of  $\sqrt{L_x L_y} \Psi_{n,m}(x,y)$  for  $n=$ ' + str(n) + r'$ and_
↳ $m=$ ' + str(m) + r'$')
axes.set_ylabel(r'$y/L_y$')
axes.set_xlabel(r'$x/L_x$')

# Plotting the colorbar for the density plots
fig = plt.figure(figsize=(10,3))

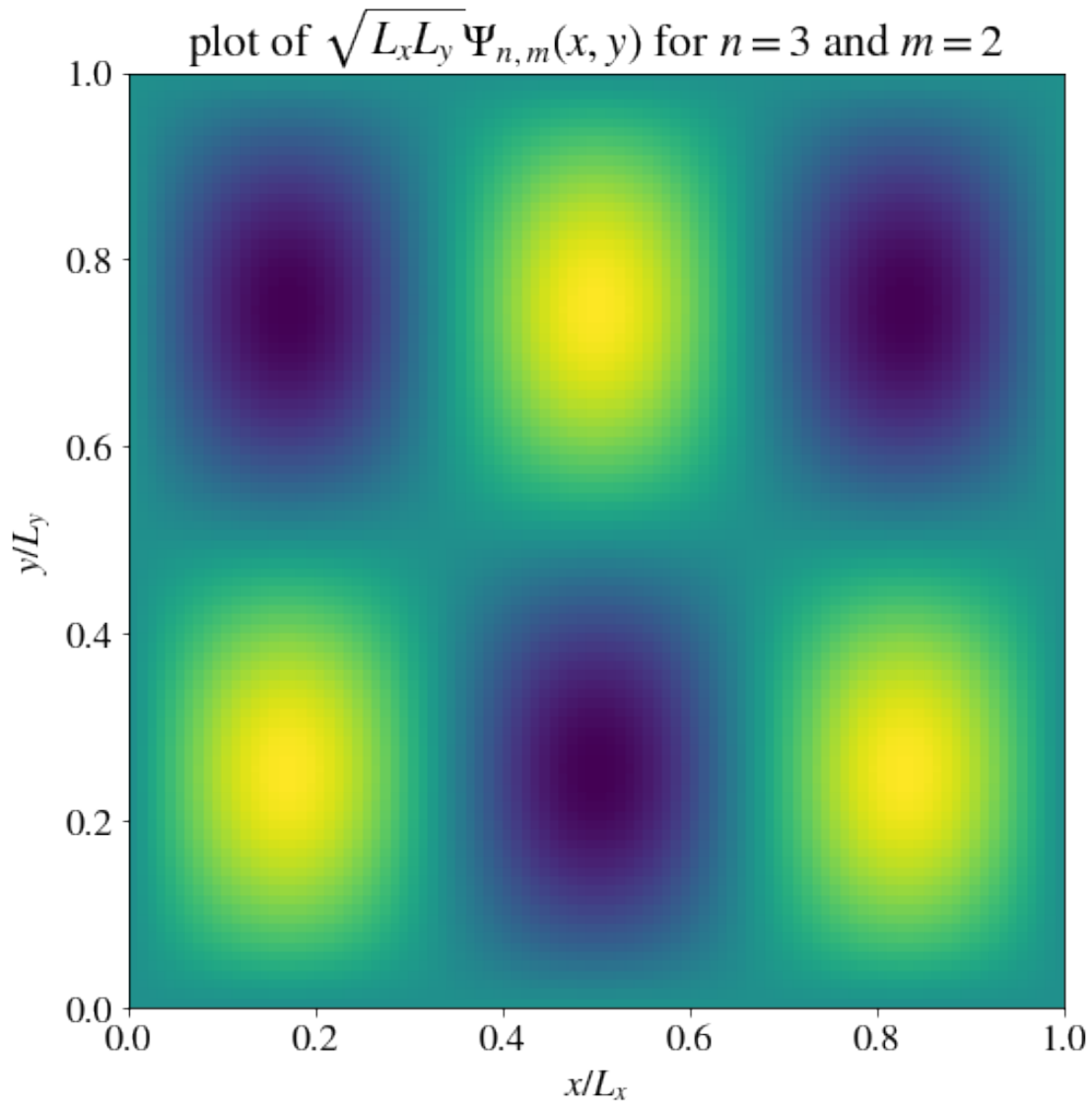
plt.show()

```

Let's look at the Wavefunction for a 2D box

Enter the value for n = 3

Enter the value for m = 2



<Figure size 720x216 with 0 Axes>

The energy will be given by the sum of the 1D energies:

$$E_{n,m} = E_n + E_m = \frac{h^2}{8m} \left(\frac{n^2}{L_x^2} + \frac{m^2}{L_y^2} \right)$$

```
[130]: import matplotlib.pyplot as plt

# Defining the energy as a function
def En2D(n,m,L1,L2): return 37.60597*((float(n)/L1)**2+ (float(m)/L2)**2)

# Reading data from the user
```



```

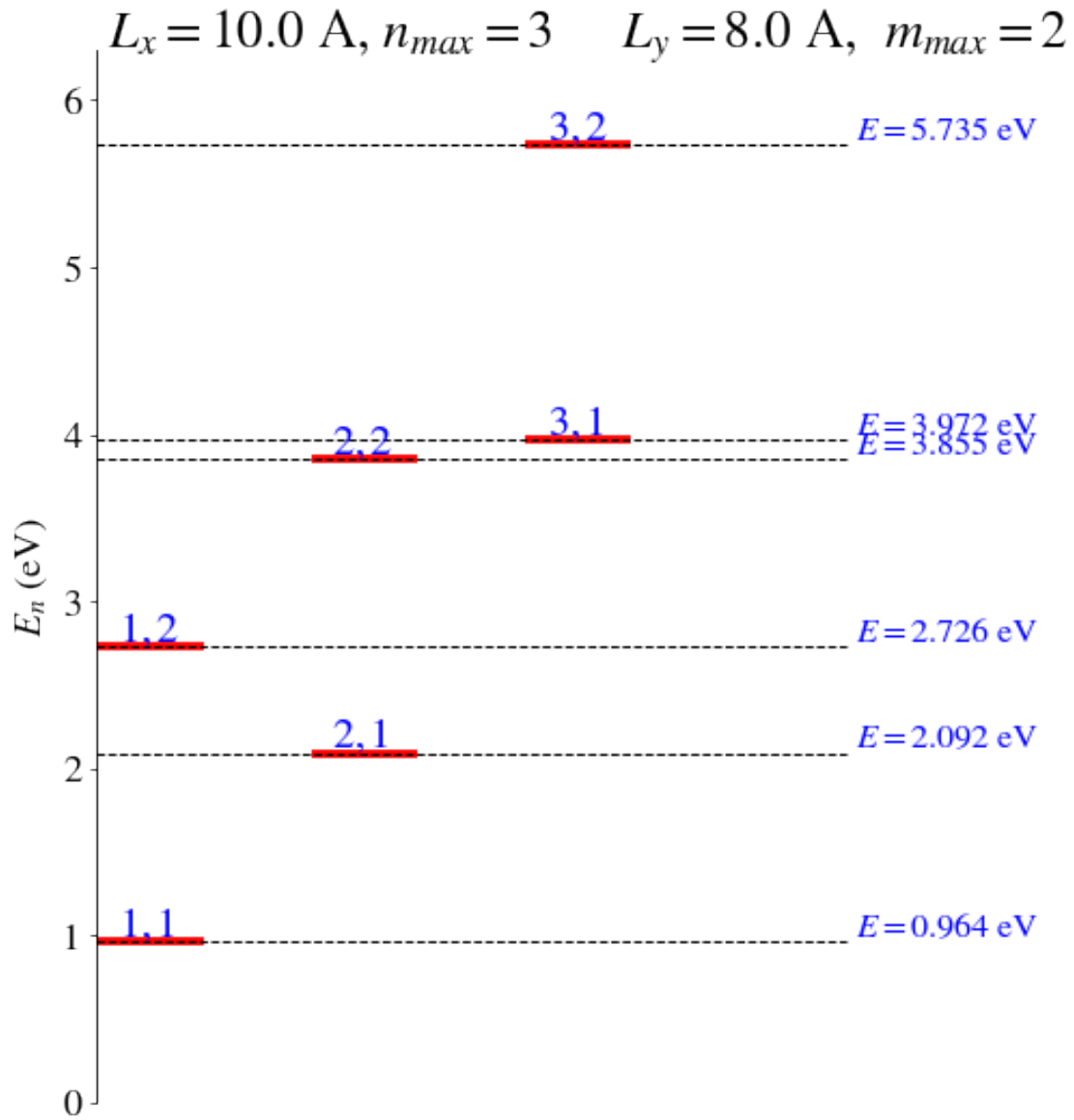
L1 = float(input("Can we count DEGENERATE states?\nEnter the value for Lx (in
↳Angstroms) = "))
nmax1 = int(input("Enter the maximum value of n to consider = "))
L2 = float(input("Enter the value for Ly (in Angstroms) = "))
mmax2 = int(input("Enter the maximum value of m to consider = "))

# Plotting the energy levels
plt.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral', 'mathtext.
↳fontset': 'stix'})
fig, ax = plt.subplots(figsize=(nmax1*2+2,nmax1*3))
ax.spines['right'].set_color('none')
ax.yaxis.tick_left()
ax.spines['bottom'].set_color('none')
ax.axes.get_xaxis().set_visible(False)
ax.spines['top'].set_color('none')
val = 1.1*(En2D(nmax1,mmax2,L1,L2))
val2= 1.1*max(L1,L2)
ax.axis([0.0,3*nmax1,0.0,val])
ax.set_ylabel(r'$E_n$ (eV)')
for n in range(1,nmax1+1):
    for m in range(1, mmax2+1):
        str1="$"+str(n)+r", "+str(m)+r"$"
        str2=" $E = %.3f$ eV"%(En2D(n,m,L1,L2))
        ax.text(n*2-1.8, En2D(n,m,L1,L2)+ 0.005*val, str1, fontsize=20,
↳color="blue")
        ax.hlines(En2D(n,m,L1,L2), n*2-2, n*2-1, linewidth=3.8, color="red")
        ax.hlines(En2D(n,m,L1,L2), 0.0, nmax1*2+1, linewidth=1.,
↳linestyle='--', color="black")
        ax.text(nmax1*2+1, En2D(n,m,L1,L2)+ 0.005*val, str2, fontsize=16,
↳color="blue")
plt.title("Energy Levels for \n ", fontsize=30)
str1=r"$L_x = "+str(L1)+r"$ A, $n_{max} = "+str(nmax1)+r"$      $L_y =
↳"+str(L2)+r"$ A,  $m_{max}="+str(mmax2)+r"$"
ax.text(0.1,val, str1, fontsize=25, color="black")
# Show the plots on the screen once the code reaches this point
plt.show()

```

Can we count DEGENERATE states?
Enter the value for Lx (in Angstroms) = 10
Enter the maximum value of n to consider = 3
Enter the value for Ly (in Angstroms) = 8
Enter the maximum value of m to consider = 2

Energy Levels for



This study is very important for atomic and molecular physics study also.

14.10 Harmonic Oscillator problem:

The equation of motion of quantum mechanics for a particle is given by the Schrödinger equation,

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

Here, Ψ is the wave function of the particle, which is a function of both time, t , and position, x , moving in a potential field described by V . Solution of the Schrödinger equation typically uses the method of separating the time- from the space-dependent part of the equation. The spatial portion is the so-called *stationary* (or *time-independent*) Schrödinger equation, an eigenvalue equation which, in the coordinate representation, takes the form of a linear differential equation. The solutions of of this equation are wave functions $\psi(r)$, which assign a complex number to every point r . More specifically, they describe those states of the physical system for which the probability $|\psi(\mathbf{r})|^2$ does not change with time. To obtain a numerical solution of the Schrödinger equation, one can either approximately discretize the linear differential equation and put it in matrix form, or expand $\psi(\mathbf{r})$ in terms of a complete set of wave functions $\psi_n(\mathbf{r})$ and consider only a finite number of them. In both cases, the stationary Schrödinger equation leads to an eigenvalue equation of a finite matrix.

a one-dimensional problem where a mass m moves in the quadratic potential $V(x) = m\omega^2 x^2/2$. Here, x is the spatial coordinate and ω is the angular frequency of the harmonic oscillator. The Hamiltonian for such a system is:

$$H = \frac{1}{2} \left(\frac{p^2}{m} + m\omega^2 x^2 \right)$$

or in operator format,

$$\hat{H} = \frac{1}{2m} \left(\hbar^2 m \frac{\partial^2}{\partial x^2} + m^2 \omega^2 x^2 \right)$$

The eigenfunctions (i.e wave functions) of H_0 are:

$$\psi_n(\xi) = \left(\frac{m\omega}{\pi\hbar} \right)^{1/4} \frac{1}{\sqrt{2^n n!}} H_n(\xi) e^{-\xi^2/2}, n = 0, 1, 2, \dots$$

where the $H_n(\xi)$ here represents the Hermite polynomial, set of functions (and *not* the Hamiltonian H) and ξ is related to the position coordinate via $\xi = \sqrt{(m\omega/\hbar)} x$. From these eigenfuctions, we can set up the following eigenvalue equation to represent the time-independent Schrödinger equation:

$$H|\psi_n\rangle = E_n|\psi_n\rangle,$$

where the eigenvalues (i.e energies) of H are

$$E_n = \left(n + \frac{1}{2} \right) \hbar\omega,$$

```
[131]: import matplotlib
import matplotlib.pyplot as plt
import numpy
import numpy.polynomial.hermite as Herm
import math
#plt.figure(figsize=(5, 5))

#Choose simple units
m=1.
w=1.
```

```

hbar=1.
#Discretized space
dx = 0.05
x_lim = 12
x = numpy.arange(-x_lim,x_lim,dx)

def hermite(x, n):
    xi = numpy.sqrt(m*w/hbar)*x
    herm_coeffs = numpy.zeros(n+1)
    herm_coeffs[n] = 1
    return Herm.hermval(xi, herm_coeffs)

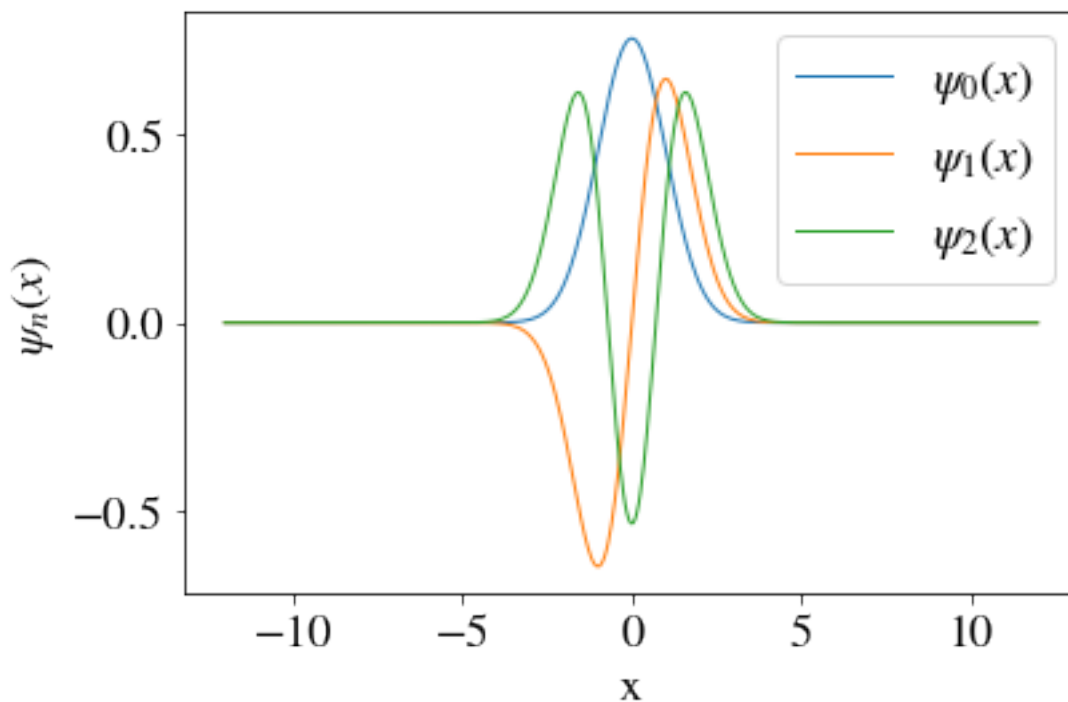
def stationary_state(x,n):
    xi = numpy.sqrt(m*w/hbar)*x
    prefactor = 1./math.sqrt(2.**n * math.factorial(n)) * (m*w/(numpy.
→pi*hbar))**(0.25)
    psi = prefactor * numpy.exp(- xi**2 / 2) * hermite(x,n)
    return psi

plt.figure()
plt.plot(x, stationary_state(x,0), linewidth=1,label=r"$\psi_0(x)$")
plt.plot(x, stationary_state(x,1), linewidth=1,label=r"$\psi_1(x)$")
plt.plot(x, stationary_state(x,2), linewidth=1,label=r"$\psi_2(x)$")

#Set limits for axes
#plt.xlim([-3,3])
#plt.ylim([-30,30])

#Set axes labels
plt.xlabel("x")
plt.ylabel(r"$\psi_n(x)$")
plt.legend()
plt.show()

```



14.11 Comparing Classical vs. Quantum Harmonic Results:

Compare the behavior of a quantum harmonic oscillator to a classical harmonic oscillator. Connect what happens as you increase the quantum number to the transition from quantum to classical behavior. There it is shown that for a classical harmonic oscillator with energy E , the classical probability of finding the particle at x is given by:

$$x_{max} = \sqrt{\frac{2E}{m\omega^2}}$$

where x_{max} is the classical turning point, and $P_{classical}(x)$ is understood to be zero for $|x| > |x_{max}|$.

```
[132]: import matplotlib
import matplotlib.pyplot as plt
import numpy
import numpy.polynomial.hermite as Herm
import math

#Choose simple units
m=1.
w=1.
hbar=1.
#Discretized space
dx = 0.05
```

```

x_lim = 12
x = numpy.arange(-x_lim,x_lim,dx)

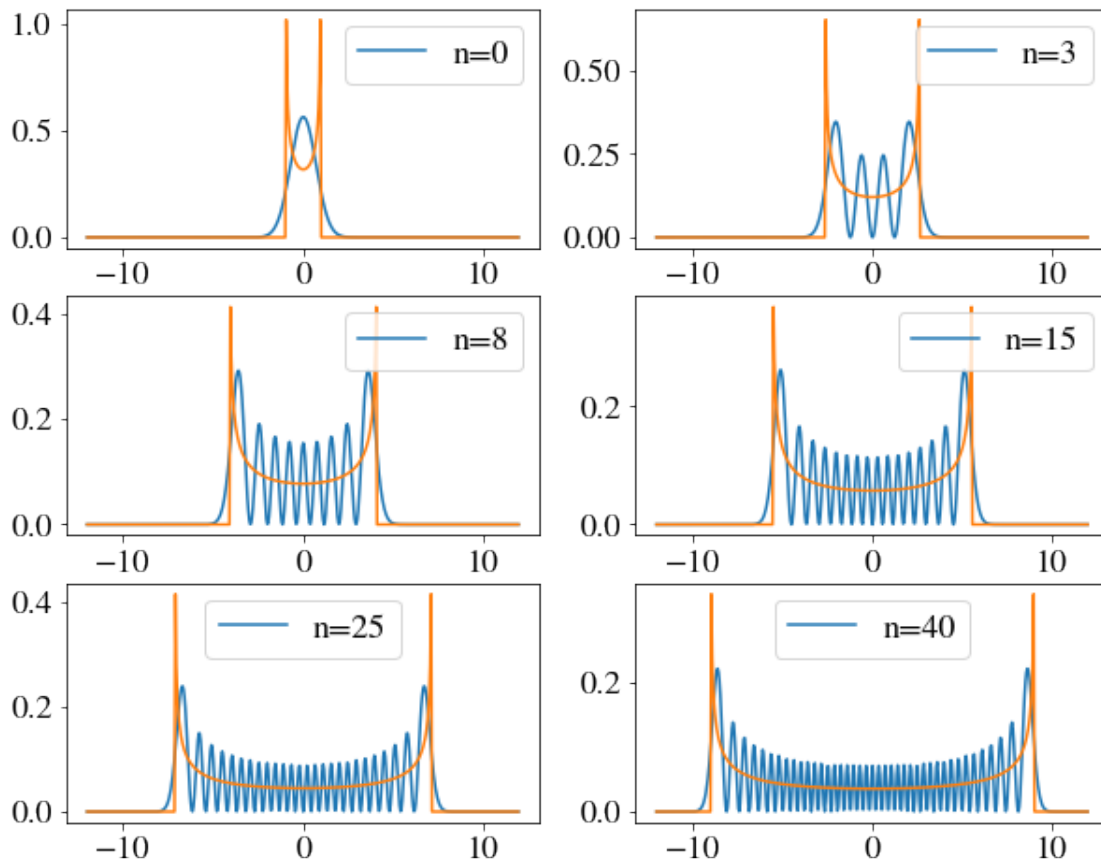
def hermite(x, n):
    xi = numpy.sqrt(m*w/hbar)*x
    herm_coeffs = numpy.zeros(n+1)
    herm_coeffs[n] = 1
    return Herm.hermval(xi, herm_coeffs)
def stationary_state(x,n):
    xi = numpy.sqrt(m*w/hbar)*x
    prefactor = 1./math.sqrt(2.**n * math.factorial(n)) * (m*w/(numpy.
    ↪pi*hbar)).**(0.25)
    psi = prefactor * numpy.exp(- xi**2 / 2) * hermite(x,n)
    return psi

def classical_P(x,n):
    E = hbar*w*(n+0.5)
    x_max = numpy.sqrt(2*E/(m*w**2))
    classical_prob = numpy.zeros(x.shape[0])
    x_inside = abs(x) < (x_max - 0.025)
    classical_prob[x_inside] = 1./numpy.pi/numpy.
    ↪sqrt(x_max**2-x[x_inside]*x[x_inside])
    return classical_prob

plt.figure(figsize=(10, 8))
plt.subplot(3,2,1)
plt.plot(x, numpy.conjugate(stationary_state(x,0))*stationary_state(x,0),
    ↪label="n=0")
plt.plot(x, classical_P(x,0))
plt.legend()
plt.subplot(3,2,2)
plt.plot(x, numpy.conjugate(stationary_state(x,3))*stationary_state(x,3),
    ↪label="n=3")
plt.plot(x, classical_P(x,3))
plt.legend()
plt.subplot(3,2,3)
plt.plot(x, numpy.conjugate(stationary_state(x,8))*stationary_state(x,8),
    ↪label="n=8")
plt.plot(x, classical_P(x,8))
plt.legend()
plt.subplot(3,2,4)
plt.plot(x, numpy.conjugate(stationary_state(x,15))*stationary_state(x,15),
    ↪label="n=15")
plt.plot(x, classical_P(x,15))
plt.legend()
plt.subplot(3,2,5)

```

```
plt.plot(x, numpy.conjugate(stationary_state(x,25))*stationary_state(x,25),
↪label="n=25")
plt.plot(x, classical_P(x,25))
plt.legend()
plt.subplot(3,2,6)
plt.plot(x, numpy.conjugate(stationary_state(x,40))*stationary_state(x,40),
↪label="n=40")
plt.plot(x, classical_P(x,40))
plt.legend()
plt.show()
```



14.12 Visualizing the spherical harmonics:

Visualising the spherical harmonics is a little tricky because they are complex and defined in terms of angular co-ordinates, (θ, ϕ) .

```
[133]: import matplotlib.pyplot as plt
from matplotlib import cm, colors
```

```

from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from scipy.special import sph_harm

phi = np.linspace(0, np.pi, 100)
theta = np.linspace(0, 2*np.pi, 100)
phi, theta = np.meshgrid(phi, theta)

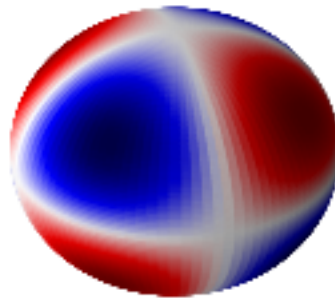
# The Cartesian coordinates of the unit sphere
x = np.sin(phi) * np.cos(theta)
y = np.sin(phi) * np.sin(theta)
z = np.cos(phi)

m, l = 2, 3

# Calculate the spherical harmonic Y(l,m) and normalize to [0,1]
fcolors = sph_harm(m, l, theta, phi).real
fmax, fmin = fcolors.max(), fcolors.min()
fcolors = (fcolors - fmin)/(fmax - fmin)

# Set the aspect ratio to 1 so our sphere looks spherical
fig = plt.figure(figsize=plt.figaspect(1.))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z, rstride=1, cstride=1, facecolors=cm.seismic(fcolors))
# Turn off the axis planes
ax.set_axis_off()
plt.show()

```



15 WKB Approximation using python

There is an approximate solution to the one dimensional Helmholtz equation that can be used to describe propagation through any slowly varying index profile. This is called the WKB approximation and the approximate solution is given by:

$$\phi(x) = \frac{1}{\sqrt{n(x)}} e^{\pm i k_0 \int^x n(x') dx'}$$

with the sign being determined by whether the wave is travelling to the right (positive) or left (negative). In this case I choose rightwards propagation.

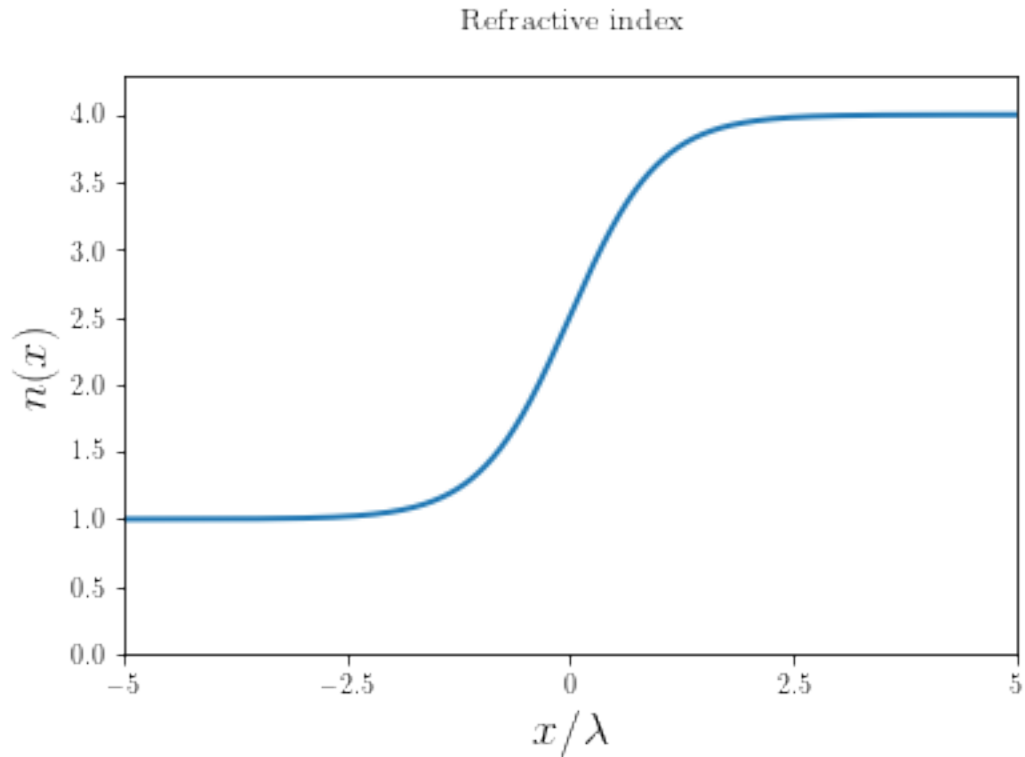
```
[134]: import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as od
from matplotlib import rc
%matplotlib inline
rc('text',usetex=True)
```

15.1 The refractive index profile:

Here I choose a refractive index profile that gradually increases from one constant to another, over a length scale of approximately two wavelengths.

```
[135]: k0=1.0 # Free space wavevector
lm=2.0*np.pi/k0 # Free space wavelength
a=1.0*lm # Length scale of profile
h=3.0 # Difference between index on the far left and the far right
def n(x):
    return 1.0+0.5*h*(1.0+np.tanh(x/a))
```

```
[136]: Xv=np.linspace(-5*lm,5*lm,2000)
nv=n(Xv)
plt.plot(Xv,nv,lw=2)
plt.xlim(-5*lm,5*lm)
plt.ylim(0.0,1.1*h+1.0)
plt.xticks([-5*lm,-2.5*lm,0,2.5*lm,5*lm],["$-5$", "$-2.5$", "$0$", "$2.5$", "$5$"])
plt.xlabel("$x/\\lambda$", fontsize=18)
plt.ylabel("$n(x)$", fontsize=18)
plt.title("Refractive index", y=1.05);
```



15.2 The WKB approximation:

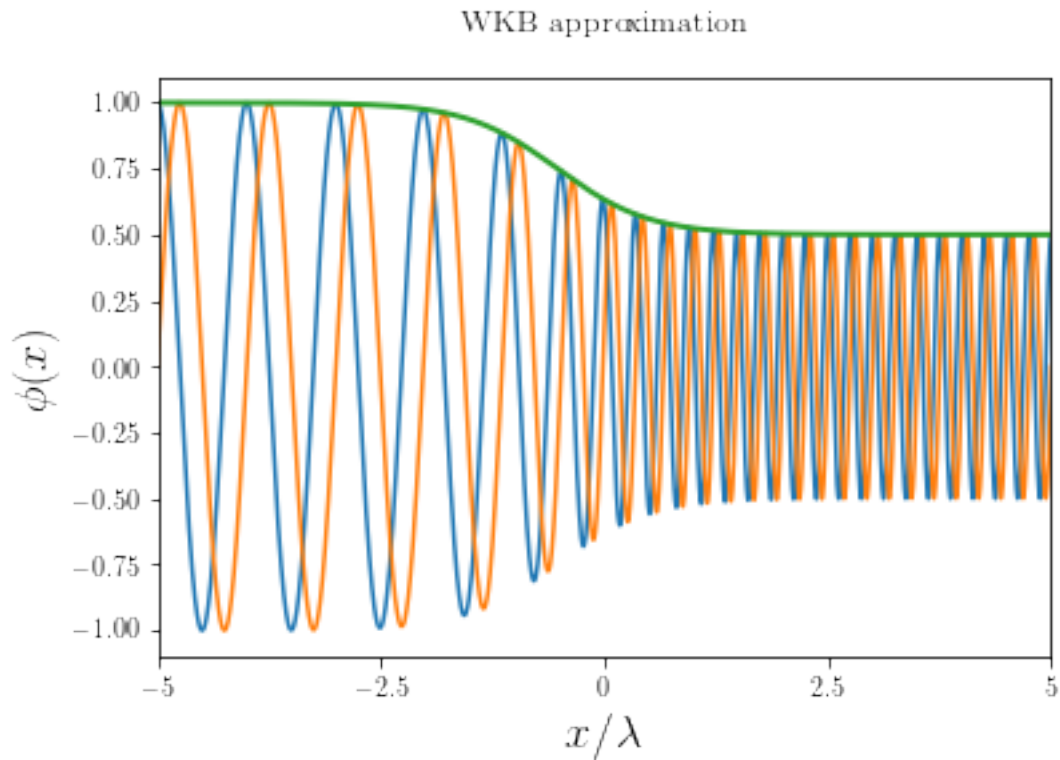
```
[137]: # WKB phase
def ph(x):
    res=od.quad(n,0,x)[0]
    return res

def phi_wkb(x):
    pf=1.0/np.sqrt(n(x))
    return pf*np.exp(1j*k0*ph(x))
```

```
[138]: phv=np.array([phi_wkb(i) for i in Xv])
phv=phv/phv[0]
```

```
[139]: plt.plot(Xv,np.real(phv))
plt.plot(Xv,np.imag(phv))
plt.plot(Xv,np.abs(phv),lw=2)
plt.xlim(-5*lm,5*lm)
plt.ylim(-1.1,1.1)
plt.xticks([-5*lm,-2.5*lm,0,2.5*lm,5*lm],["$-5$", "$-2.5$", "$0$", "$2.5$", "$5$"])
plt.xlabel("$x/\backslash\lambda$", fontsize=18)
```

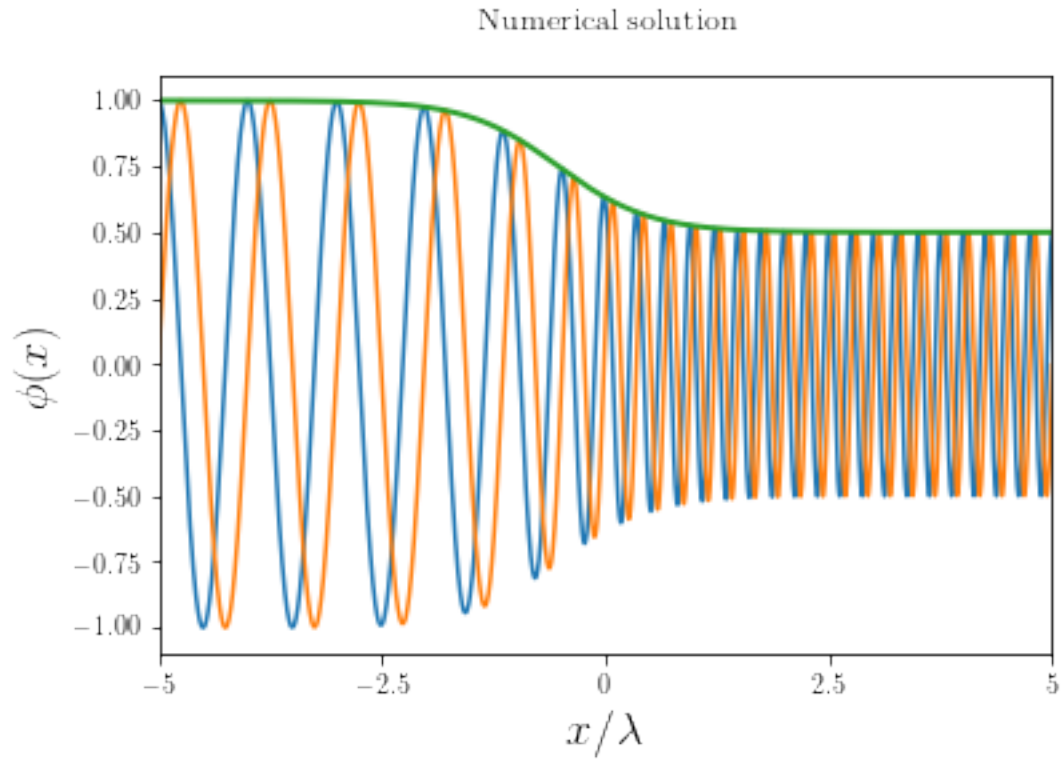
```
plt.ylabel("$\phi(x)$", fontsize=18)
plt.title("WKB approximation", y=1.05);
```



15.3 Numerical solution:

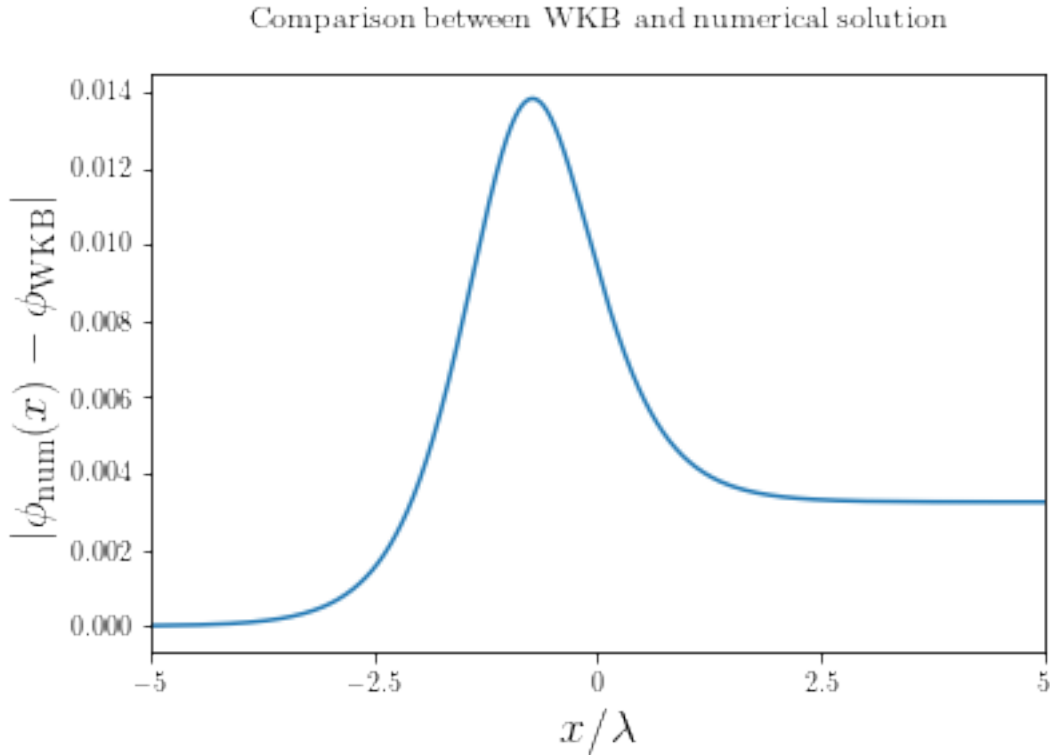
```
[140]: def dphi(phi,x):
        return [phi[2],phi[3],-phi[0]*(n(x)*k0)**2.0,-phi[1]*(n(x)*k0)**2.0]

phnv_all=od.odeint(dphi,[1.0,0.0,0.0,n(5*lm)*k0],Xv[:, :-1])
phnv=np.array([i[0]+1j*i[1] for i in phnv_all])
phnv=phnv[:, :-1]/phnv[len(phnv)-1]
plt.plot(Xv,np.real(phnv))
plt.plot(Xv,np.imag(phnv))
plt.plot(Xv,np.abs(phnv),lw=2)
plt.xlim(-5*lm,5*lm)
plt.ylim(-1.1,1.1)
plt.xticks([-5*lm,-2.5*lm,0,2.5*lm,5*lm],["$-5$","$-2.5$","$0$","$2.5$","$5$"])
plt.xlabel("$x/\lambda$", fontsize=18)
plt.ylabel("$\phi(x)$", fontsize=18)
plt.title("Numerical solution", y=1.05);
```



To compare the two solutions I take the absolute value between the numerical result and the WKB solution.

```
[141]: plt.plot(Xv,np.abs(phnv-phv))
plt.xlim(-5*lm,5*lm)
plt.xticks([-5*lm,-2.5*lm,0,2.5*lm,5*lm],["$-5$", "$-2.5$", "$0$", "$2.5$", "$5$"])
plt.xlabel("$x/\\lambda$", fontsize=18)
plt.ylabel("$|\\phi_{\\rm num}(x)-\\phi_{\\rm WKB}|$", fontsize=18)
plt.title("Comparison between WKB and numerical solution", y=1.05);
```



16 Matrix representaion of quantum mechanics:

In this chapter we will discuss about QuTiP is a python package for calculations and numerical simulations of quantum systems.

It includes facilities for representing and doing calculations with quantum objects such state vectors (wavefunctions), as bras/kets/density matrices, quantum operators of single and composite systems, and superoperators (useful for defining master equations).

It also includes solvers for a time-evolution of quantum systems, according to: Schrodinger equation, von Neuman equation, master equations, Floquet formalism, Monte-Carlo quantum trajectories, experimental implementations of the stochastic Schrodinger/master equations.

```
[143]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import Image
from qutip import *
```

At the heart of the QuTiP package is the Qobj class, which is used for representing quantum object such as states and operator.

The Qobj class contains all the information required to describe a quantum system, such as its matrix representation, composite structure and dimensionality.

16.1 Creating and inspecting quantum objects

```
[144]: q = Qobj([[1], [0]])
```

```
q
```

```
[144]: Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
```

$$\begin{pmatrix} 1.0 \\ 0.0 \end{pmatrix}$$

```
[145]: print("\nthe dimension, or composite Hilbert state space structure:\n",q.dims)
```

```
the dimension, or composite Hilbert state space structure:
[[2], [1]]
```

```
[146]: print("\nthe shape of the matrix data representation:\n",q.shape)
# get the dense matrix representation
print("\nget the dense matrix representation:\n",q.full())
```

```
the shape of the matrix data representation:
(2, 1)
```

```
get the dense matrix representation:
[[1.+0.j]
 [0.+0.j]]
```

```
[147]: print("\nIs q is Hermitian?:",q.isherm)
print("\nWhat is type of Quantum object?: ",q.type)
```

```
Is q is Hermitian?: False
```

```
What is type of Quantum object?: ket
```

```
[148]: # the sigma-y Pauli operator
sy = Qobj([[0,-1j], [1j,0]])
sy
```

```
[148]: Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
```

$$\begin{pmatrix} 0.0 & -1.0j \\ 1.0j & 0.0 \end{pmatrix}$$

```
[149]: # the sigma-z Pauli operator
sz = Qobj([[1,0], [0,-1]])
sz
```

[149]: Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True

$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & -1.0 \end{pmatrix}$$

```
[150]: # some arithmetic with quantum objects
H = 1.0 * sz + 0.1 * sy
print("Qubit Hamiltonian = \n")
H
```

Qubit Hamiltonian =

[150]: Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True

$$\begin{pmatrix} 1.0 & -0.100j \\ 0.100j & -1.0 \end{pmatrix}$$

```
[151]: # The hermitian conjugate
sy.dag()
```

[151]: Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True

$$\begin{pmatrix} 0.0 & -1.0j \\ 1.0j & 0.0 \end{pmatrix}$$

```
[152]: # The trace
H.tr()
```

[152]: 0.0

```
[153]: print("\nEigen energies:\n",H.eigenenergies())
```

Eigen energies:
[-1.00498756 1.00498756]

16.2 States and operators:

Normally we do not need to create Qobj instances from scratch, using its constructor and passing its matrix representation as argument. Instead we can use functions in QuTiP that generate common states and operators for us. Here are some examples of built-in state functions:

16.2.1 State vectors:

```
[154]: # Fundamental basis states (Fock states of oscillator modes)
```

```
N = 3 # number of states in the Hilbert space
```

```
n = 2 # the state that will be occupied
```

```
basis(N, n) # equivalent to fock(N, n)
```

[154]: Quantum object: dims = [[3], [1]], shape = (3, 1), type = ket

$$\begin{pmatrix} 0.0 \\ 0.0 \\ 1.0 \end{pmatrix}$$

```
[155]: # a coherent state
```

```
print(coherent(N=5, alpha=1))
```

Quantum object: dims = [[5], [1]], shape = (5, 1), type = ket

Qobj data =

[0.60655682]

[0.60628133]

[0.4303874]

[0.24104351]

[0.14552147]]

16.2.2 Density matrices:

```
[156]: # a fock state as density matrix
```

```
fock_dm(5, 1) # 5 = hilbert space size, 1 = state that is occupied
```

[156]: Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = True

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

```
[157]: # coherent state as density matrix
```

```
coherent_dm(N=4, alpha=1.0)
```

[157]: Quantum object: dims = [[4], [4]], shape = (4, 4), type = oper, isherm = True

$$\begin{pmatrix} 0.367 & 0.370 & 0.250 & 0.182 \\ 0.370 & 0.372 & 0.252 & 0.183 \\ 0.250 & 0.252 & 0.170 & 0.124 \\ 0.182 & 0.183 & 0.124 & 0.090 \end{pmatrix}$$

16.2.3 Operators:

σ_x operator:

```
[158]: # Pauli sigma x  
sigmax()
```

[158]: Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True

$$\begin{pmatrix} 0.0 & 1.0 \\ 1.0 & 0.0 \end{pmatrix}$$

σ_y operator:

```
[159]: # Pauli sigma y  
sigmay()
```

[159]: Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True

$$\begin{pmatrix} 0.0 & -1.0j \\ 1.0j & 0.0 \end{pmatrix}$$

σ_z operator:

```
[160]: # Pauli sigma z  
sigmaz()
```

[160]: Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True

$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & -1.0 \end{pmatrix}$$

16.3 Harmonic oscillator operators:

```
[161]: # annihilation operator
```

```
destroy(N=4) # N = number of fock states included in the Hilbert space
```

[161]: Quantum object: dims = [[4], [4]], shape = (4, 4), type = oper, isherm = False

$$\begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.414 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.732 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

```
[162]: # creation operator
```

```
create(N=4) # equivalent to destroy(5).dag()
```

[162]: Quantum object: dims = [[4], [4]], shape = (4, 4), type = oper, isherm = False

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.414 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.732 & 0.0 \end{pmatrix}$$

```
[163]: # the position operator is easily constructed from the annihilation operator
a = destroy(4)

x = a + a.dag()

x
```

[163]: Quantum object: dims = [[4], [4]], shape = (4, 4), type = oper, isherm = True

$$\begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 1.414 & 0.0 \\ 0.0 & 1.414 & 0.0 & 1.732 \\ 0.0 & 0.0 & 1.732 & 0.0 \end{pmatrix}$$

16.3.1 Commutator relation:

```
[164]: def commutator(op1, op2):
        return op1 * op2 - op2 * op1
```

```
[165]: a = destroy(4)

commutator(a, a.dag())
```

[165]: Quantum object: dims = [[4], [4]], shape = (4, 4), type = oper, isherm = True

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.000 & 0.0 \\ 0.0 & 0.0 & 0.0 & -3.000 \end{pmatrix}$$

Let's check well known commutator relation between x and p. Checked matrix representation of operator chapter from Zetli or any other quantum mechanics book, where

$$x = \frac{a + a^\dagger}{\sqrt{2}}$$

$$p = -j \frac{a - a^\dagger}{\sqrt{2}}$$

```
[166]: x = (a + a.dag())/np.sqrt(2)
p = -1j * (a - a.dag())/np.sqrt(2)
commutator(x, p)
```

[166]: Quantum object: dims = [[4], [4]], shape = (4, 4), type = oper, isherm = False

$$\begin{pmatrix} 1.000j & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0j & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.000j & 0.0 \\ 0.0 & 0.0 & 0.0 & -3.000j \end{pmatrix}$$

16.3.2 Pauli Spin identities:

Let's try to check this identity:

$$[\sigma_x, \sigma_y] = 2i\sigma_z$$

```
[167]: commutator(sigmax(), sigmay()) - 2j * sigmaz()
```

[167]: Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True

$$\begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{pmatrix}$$

Also we can check:

$$-i\sigma_x\sigma_y\sigma_z = 1$$

```
[168]: -1j * sigmax() * sigmay() * sigmaz()
```

[168]: Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True

$$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$$

16.4 Cat vs coherent states in a Kerr resonator, and the role of measurement:

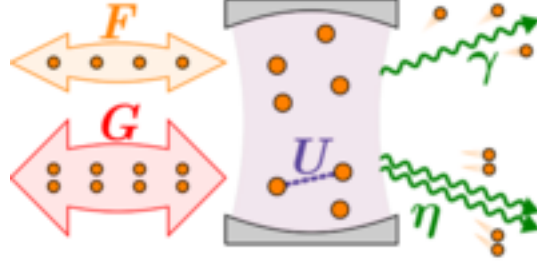
we show how the same system can produce extremely different results according to the way an observer collects the emitted field of a resonator.

```
[169]: import matplotlib.pyplot as plt
import numpy as np
from qutip import *
from IPython.display import display, Math, Latex
```

16.4.1 The two-photon Kerr Resonator:

[170]: `Image(filename="PhysRevA.94.033841.png",width=650)`

[170]:



Let us consider a single nonlinear Kerr resonator subject to a parametric two-photon driving. In a frame rotating at the pump frequency, the Hamiltonian reads:

$$H = \frac{U}{2} a^\dagger a^\dagger a a + \frac{G}{2} (a^\dagger a^\dagger + a a)$$

where U is the Kerr photon-photon interaction strength, G is the two-photon driving amplitude, and $a^\dagger(a)$ is the bosonic creation (annihilation) operator. where γ and η are, respectively, the one- and two-photon dissipation rates.

This model can be solved exactly for its steady state. The corresponding density matrix ρ_{ss} is well approximated by the statistical mixture of two orthogonal states:

$$\rho_{ss} = p^+ |C_\alpha^+ \rangle \langle C_\alpha^+| + p^- |C_\alpha^- \rangle \langle C_\alpha^-|$$

Where $|C_\alpha^\pm\rangle = |\alpha \pm i| - \alpha\rangle$ are photonic Schrödinger cat states whose complex amplitude α is determined by the system parameters. The state $|C_\alpha^+\rangle$ is called the even cat, since it can be written as a superposition of solely even Fock states, while $|C_\alpha^-\rangle$ is the odd cat. In the previous equation, the coefficients p^\pm can be interpreted as the probabilities of the system of being found in the corresponding cat state.

Below, we demonstrate this feature by diagonalising the steady-state density matrix, and by plotting the photon-number probability for the two most probable states.

[176]: `font_size=20
label_size=30
title_font=35`

```

a=destroy(20)
U=1
G=4
gamma=1
eta=1
H=U*a.dag()*a.dag()*a*a + G*(a*a + a.dag()*a.dag())
c_ops=[np.sqrt(gamma)*a,np.sqrt(eta)*a*a]

parity=1.j*np.pi*a.dag()*a
parity=parity.expm()

rho_ss=steadystate(H, c_ops)

```

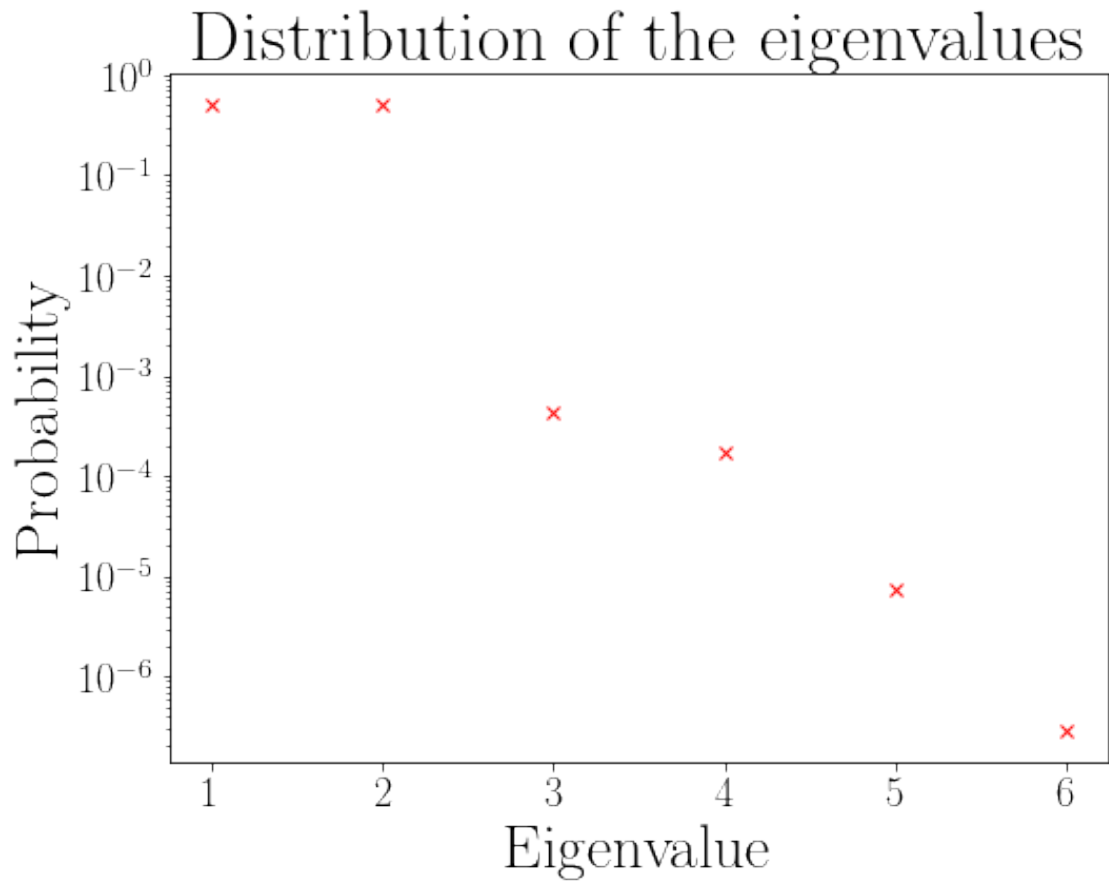
```

[177]: vals, vecs = rho_ss.eigenstates(sort='high')
print("The mean number of photon is " + str(expect(a.dag()*a, rho_ss)))

plt.figure(figsize=(8, 6))
plt.rc('text', usetex=True)
plt.rc('font', family='serif', size=font_size)
plt.semilogy(range(1,7),vals[0:6], 'rx')
plt.xlabel('Eigenvalue', fontsize=label_size)
plt.ylabel('Probability', fontsize=label_size)
plt.title('Distribution of the eigenvalues',fontsize=title_font)
plt.show()

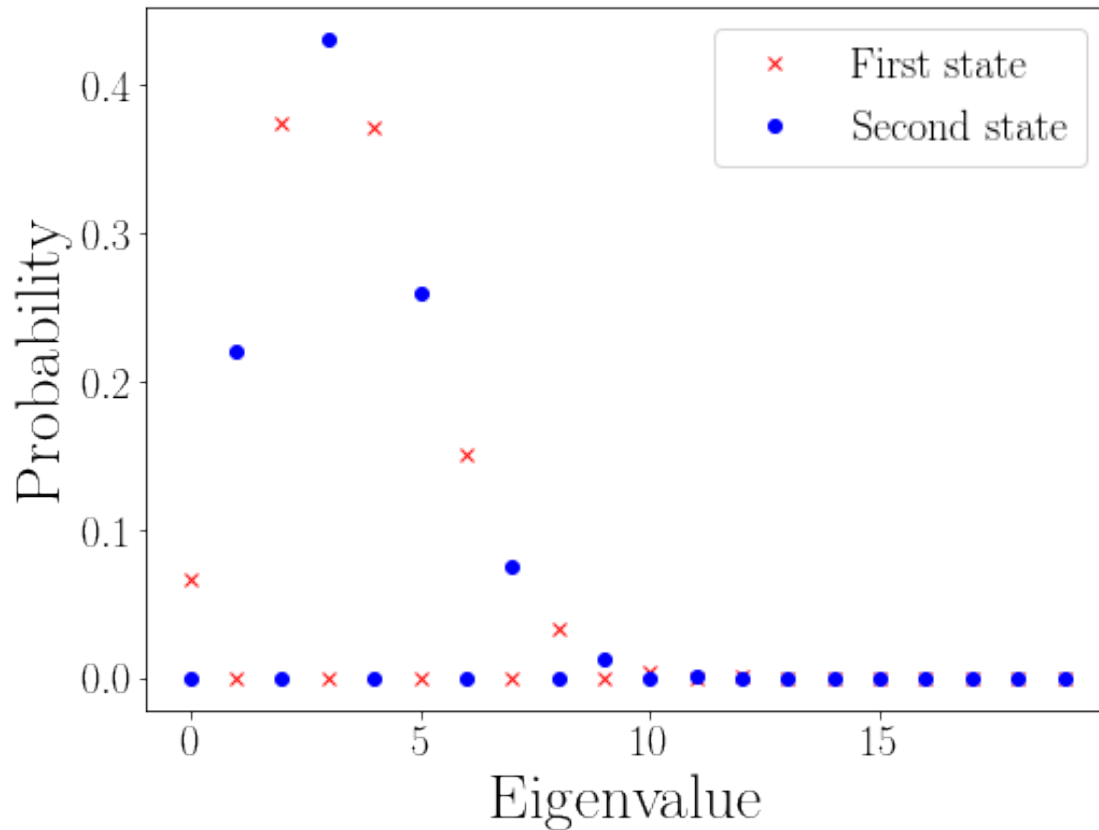
```

The mean number of photon is 3.4606002041553965



```
[178]: state_zero=vecs[0].full()
state_one=vecs[1].full()

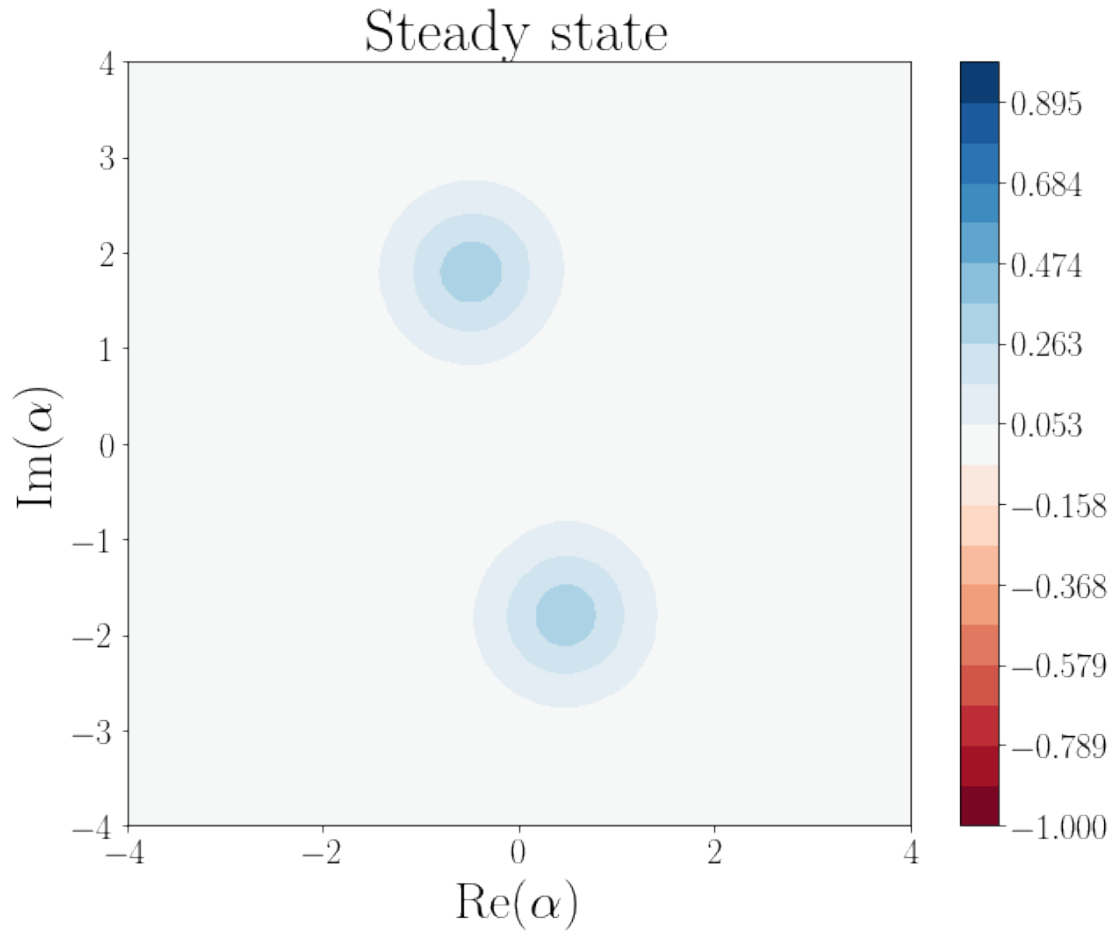
plt.figure(figsize=(8, 6))
plt.rc('text', usetex=True)
plt.rc('font', family='serif', size=font_size)
plt.plot(range(0,20), [abs(i)**2 for i in state_zero[0:20]], 'rx', label='First_
→state')
plt.plot(range(0,20), [abs(i)**2 for i in state_one[0:20]], 'bo', label='Second_
→state')
plt.legend()
plt.xlabel('Eigenvalue', fontsize=label_size)
plt.ylabel('Probability', fontsize=label_size)
plt.show()
```



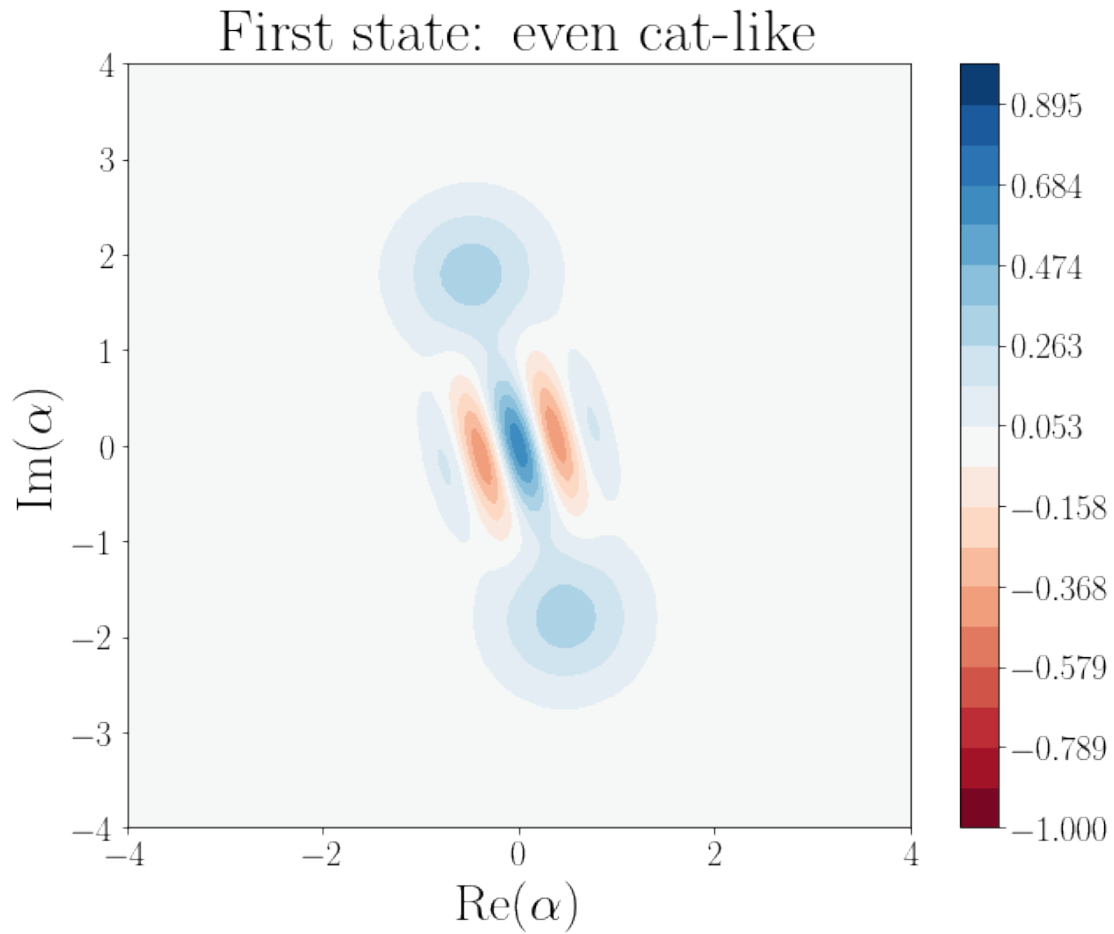
```
[179]: xvec=np.linspace(-4,4, 500)
W_even=wigner(vecs[0], xvec, xvec, g=2)
W_odd=wigner(vecs[1], xvec, xvec, g=2)

W_ss=wigner(rho_ss, xvec, xvec, g=2)
W_ss=np.around(W_ss, decimals=2)
plt.figure(figsize=(10, 8))

plt.contourf(xvec,xvec, W_ss, cmap='RdBu', levels=np.linspace(-1, 1, 20))
plt.colorbar()
plt.xlabel(r'Re$(\alpha)$', fontsize=label_size)
plt.ylabel(r'Im$(\alpha)$', fontsize=label_size)
plt.title("Steady state", fontsize=title_font)
plt.show()
```

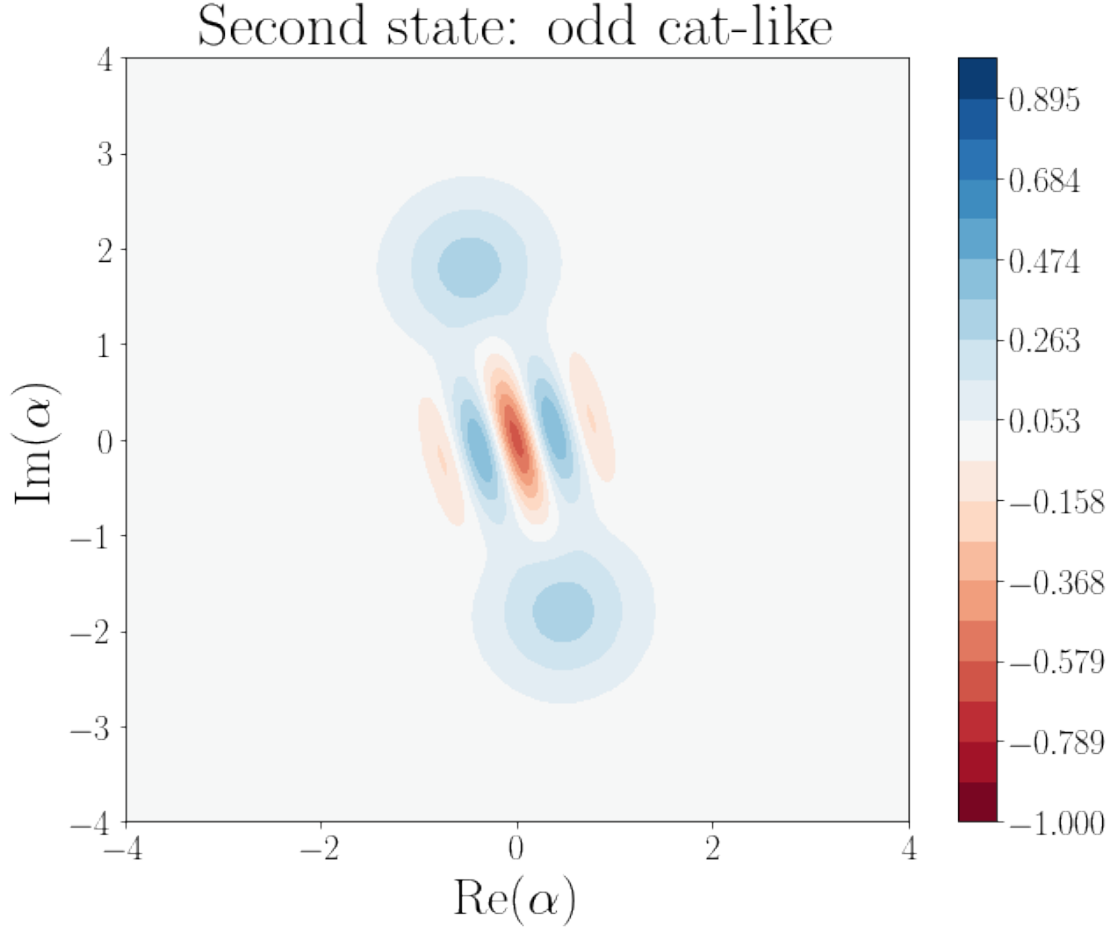


```
[180]: xvec=np.linspace(-4,4, 500)
W_even=wigner(vecs[0], xvec, xvec, g=2)
W_odd=wigner(vecs[1], xvec, xvec, g=2)
font_size=20
label_size=30
title_font=35
W_even=np.around(W_even, decimals=2)
plt.figure(figsize=(10, 8))
plt.contourf(xvec,xvec, W_even, cmap='RdBu', levels=np.linspace(-1, 1, 20))
plt.colorbar()
plt.xlabel(r"Re$(\alpha)$", fontsize=label_size)
plt.ylabel(r"Im$(\alpha)$", fontsize=label_size)
plt.title("First state: even cat-like", fontsize=title_font)
plt.show()
```

```
[181]: W_odd=np.around(W_odd, decimals=2)
plt.figure(figsize=(10, 8))

plt.contourf(xvec,xvec, W_odd, cmap='RdBu', levels=np.linspace(-1, 1, 20))
plt.colorbar()
plt.xlabel(r'Re$(\alpha)$', fontsize=label_size)
plt.ylabel(r'Im$(\alpha)$', fontsize=label_size)
plt.title("Second state: odd cat-like", fontsize=title_font)
plt.show()
```



```
[182]: tlist=np.linspace(0,8000,800)
sol_hom=ssesolve(H, fock(20,0), tlist, c_ops, [a.dag()*a, (a+a.dag())/2, -1.
↪j*(a-a.dag())/2, parity], ntraj=1, nsubsteps=9500, store_measurement=False,
↪method='homodyne')
```

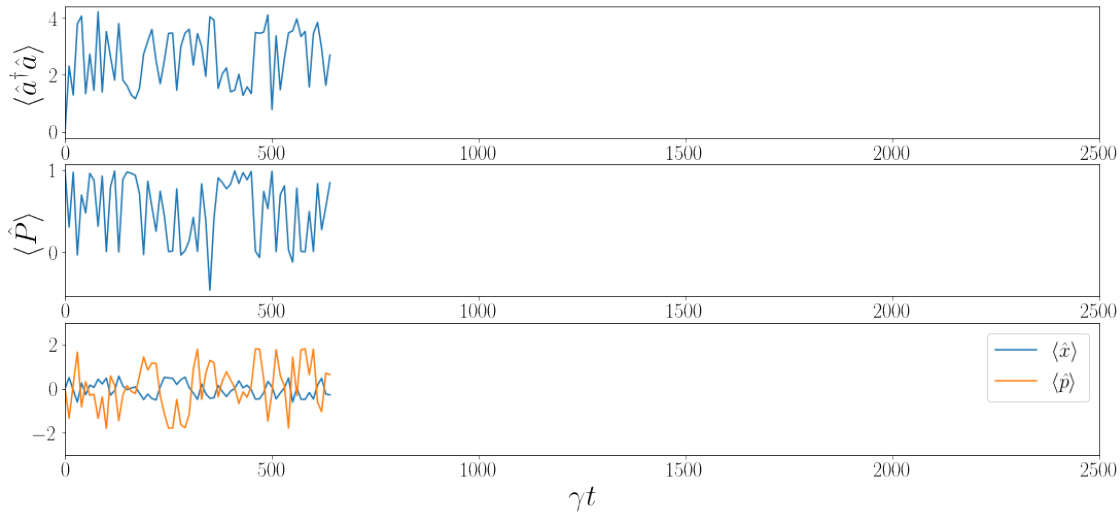
Total run time: 331.46s

16.4.2 Homodyne:

Another possible way to monitor a quantum-optical system is through homodyne detection, a widely-used experimental technique which allows to access the field quadratures. To implement this kind of measurement, the cavity output field is mixed to the coherent field of a reference laser through a beam splitter (here assumed of perfect transmittance). Then, the mixed fields are probed via (perfect) photodetectors, whose measures are described by new jump operators. We stress that both the coherent and the cavity fields are measured simultaneously. we want to probe independently the two dissipation channels. To distinguish between one- and two-photon losses, one can exploit a nonlinear element acting on the cavity output field. Indeed, in experimental

realisations such as, a nonlinear element is already part of the system and is the key ingredient to realise two-photon processes.

```
[183]: plt.figure(figsize=(18, 8))
plt.subplot(311)
plt.plot(tlist, sol_hom.expect[0])
plt.ylabel(r'$\langle \hat{a}^\dagger \hat{a} \rangle$', fontsize=label_size)
plt.xlim([0,2500])
plt.subplot(312)
plt.plot(tlist, sol_hom.expect[3])
plt.ylabel(r'$\langle \hat{P} \rangle$', fontsize=label_size)
plt.xlim([0,2500])
plt.subplot(313)
plt.plot(tlist, sol_hom.expect[1], label=r'$\langle \hat{x} \rangle$')
plt.plot(tlist, sol_hom.expect[2], label=r'$\langle \hat{p} \rangle$')
plt.xlabel(r'$\gamma t$', fontsize=label_size)
#plt.ylabel(r'$\langle \hat{p} \rangle$', fontsize=label_size)
plt.xlim([0,2500])
plt.ylim([-3,3])
plt.legend()
plt.show()
```



17 Monte Carlo Study of Ferro-magnetism using an Ising Model

The goal of this chapter is to create a statistical model simulating the evolution of magnetism as a function of material temperature.

Since the emergence of magnetism is attributed to the contribution of a great many small atomic magnetic dipoles a statistical method is to be utilised: - Monte Carlo methods - Random number

generation - Ferromagnetism - Ising Model

The subject of this project will be statistical in nature, and hence a basic understanding of Monte Carlo methods and random number algorithms will be necessary.

17.1 Monte Carlo Methods

Numerical computations which utilise random numbers are called Monte Carlo methods after the famous casino. The obvious applications of such methods are in [stochastic physics](#): e.g., statistical thermodynamics. However, there are other, less obvious, applications including the evaluation of multi-dimensional integrals.

This method was popularised by physicists such as Stanislaw Ulam, Enrico Fermi, John von Neumann, and Nicholas Metropolis, among others. A famous early use was employed by Enrico Fermi who in 1930 used a random method to calculate the properties of the recently discovered neutron. Of course, these early simulations were greatly restricted by the limited computational power available at that time.

Uses of Monte Carlo methods require large amounts of random numbers, and it was their use that spurred the development of pseudorandom number generators, which were far quicker to use than the tables of random numbers which had been previously used for statistical sampling.

17.2 Creating Random Numbers

No numerical algorithm can generate a truly random sequence of numbers. However, there exist algorithms which generate repeating sequences of N_{max} (say) integers which are, to a fairly good approximation, randomly distributed in the range 0 to $N_{max}-1$. Here, N_{max} is a (hopefully) large integer. This type of sequence is termed **pseudo-random**.

The most well-known algorithm for generating pseudo-random sequences of integers is the so-called **linear congruential** method. The formula linking the n^{th} and $(n+1)^{th}$ integers in the sequence is

$$X_{n+1} = (AX_n + C) \text{ MOD } N_{max} \quad (1)$$

where A , C , and N_{max} are positive integer constants. The first number in the sequence, the so-called “seed” value, is selected by the user.

As an example, calculate a list of numbers using $A = 7$, $C = 0$, and $N_{max} = 10$.

```
[184]: # Generate pseudo-random numbers
I = 1
A = 7; C=0; M=10;
for i in range(8):
    In = (A * I + C) % M
    print(In, end=" ")
    I = In
```

9 5 7 1 9 5 7 1

A typical sequence of numbers generated by this formula is

$$X = 9, 5, 7, 1, 9, 5, 7, \dots \quad (2)$$

Evidently, the above choice of values for A, C, and N_{max} is not a particularly good one, since the sequence repeats after only four iterations. However, if A, C, and N_{max} are properly chosen then the sequence is of maximal length (i.e., of length N_{max}), and approximately randomly distributed in the range 0 to $N_{max} - 1$.

17.2.1 Testing for randomness

As a general rule, before implementing a random-number generator in your programs, you should check its range and that it produces numbers that “appear” random. This can be attempted wither using graphical display of your random numbers or a more robustly, performing a mathematical analysis.

With the visual method, since your brain is quite refined at recognising patterns it can imitate if there is one in your random numbers. For instance, separate your random numbers into pairs $(x, y) = (X_i, X_{i+1})$ and analyse visually using a plot(x,y).

```
[185]: # Generate psuedo-random numbers
# in to pairs and compare visually with a plot
# Run a loop to calculate

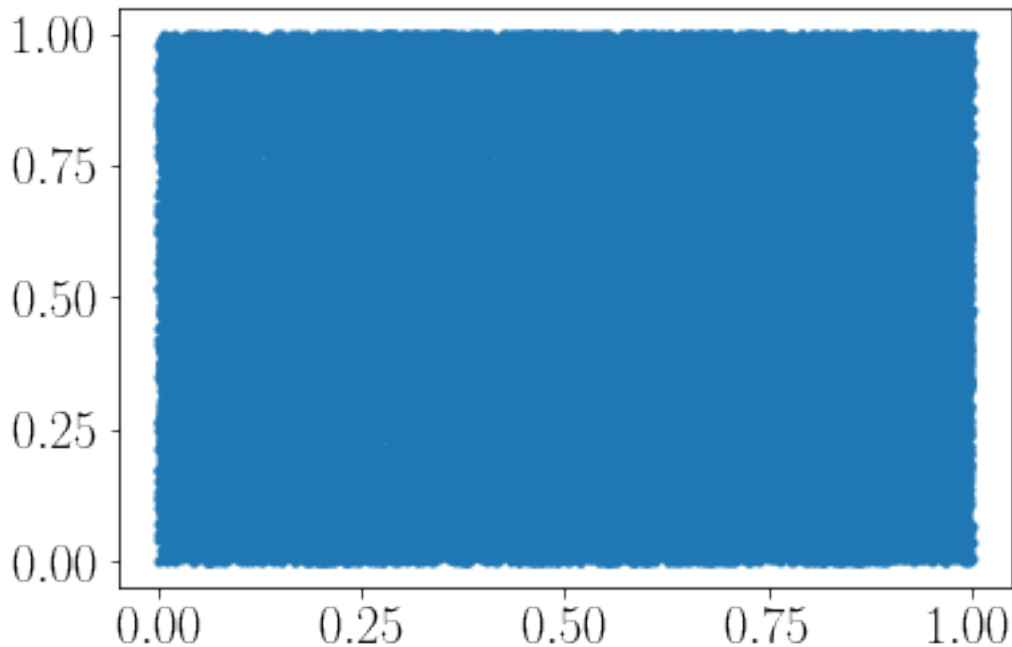
import matplotlib.pyplot as plt

Xn = 1
x = []
y = []

A = 16807; C=0; Nmax= 2147483647; # 231-1
#A = 7; C=2; Nmax=10;
for i in range(100000):

    N = (1.0*A*Xn+C)%Nmax
    if i%2 ==0:
        x.append(N/Nmax)
    else:
        y.append(N/Nmax)
    Xn = N
    #print X/Nmax

plt.plot(x,y, '.');
```

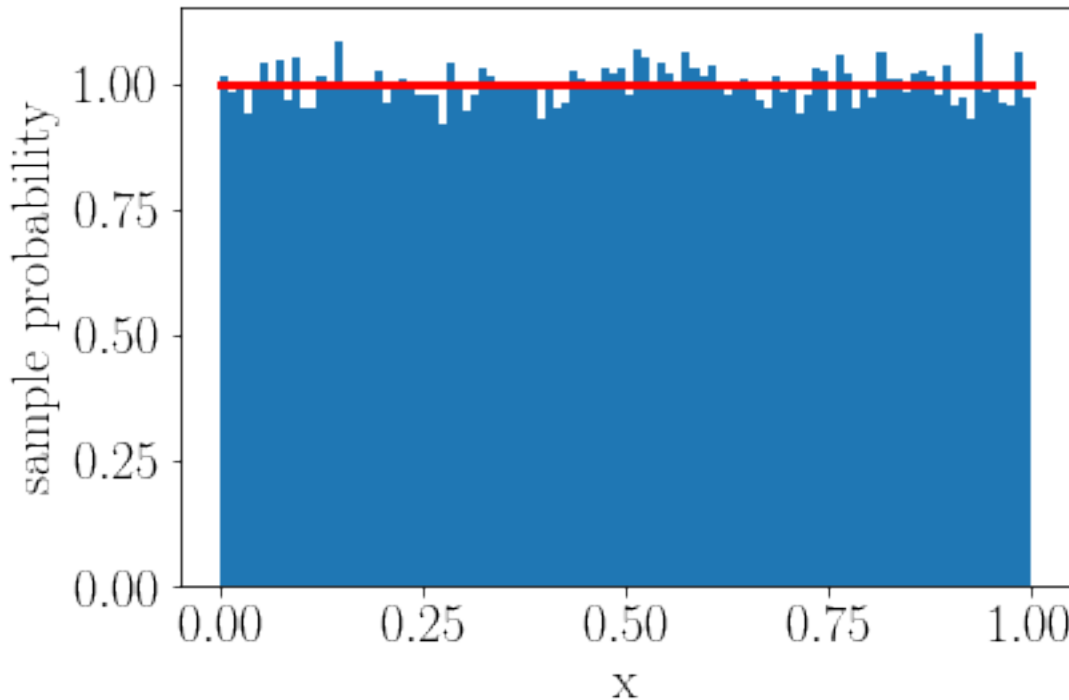


Another visual method is to plot using a histogram. If we observe a flat line at 1.0, subject to random fluctuations we can confirm there is no bias in the random distribution.

Look up the format of the hist function and plot: - a normalized probability your list of random numbers from 0-1 - in 100 bins - draw a red line to show the probability = 1

The more bins/samples we take the smaller the fluctuations about the average value.

```
[186]: import numpy as np
import matplotlib.pyplot as plt
N = x+y
n, bins, rectangles = plt.hist(N, 100, density=True)
plt.plot([0,1.],[1.,1.], 'r-', lw=3)
plt.xlabel("x", fontsize=20)
plt.ylabel("sample probability", fontsize=20)
plt.show()
```



If your list is truly random you should observe that every value of x is (roughly) equally as likely to be chosen as every other value.

Now import the function ‘random’ via Numpy which produces a **Uniformly distributed values**:

As before, when you want to reproduce a particular set of random numbers, you can set the “seed” value of the generator to a particular value. Every time you set the seed to this value, it returns the generator to its initial state and will always give you the same sequence of numbers.

The default seed value in python is **None**. The python kernel interprets this to mean that we want it to try to initialise itself using a pseudo-random number from the computer’s random number cache or it will use the current computer clock time to set the seed.

Let’s see what setting and resetting the seed does:

```
[187]: import numpy as np
        #Using the random function
        print("One set of random numbers:\n")
        # print 5 uniformly distributed numbers between 0 and 1
        print( np.random.random(5) )
        print("\nAnother set of random numbers:\n")
        # now print another 5 - should be different
        print( np.random.random(5) )
```

One set of random numbers:

```
[0.01118994 0.12333656 0.05090431 0.61643204 0.4395061 ]
```

Another set of random numbers:

```
[0.41331751 0.23825858 0.36150417 0.50209247 0.58702097]
```

Set the seed to any value using the **seed** function, and print 5 random numbers:

```
[188]: #now set the seed to something:
np.random.seed(4242)

# print 5 random numbers from the generator with this seed
print("Using seed = 4242:")
print(np.random.random(5))
```

Using seed = 4242:

```
[0.32494949 0.94041458 0.91400794 0.28650938 0.78180262]
```

Run this cell of commands with the same initial seed a few times. You should see the it produces the same result.

17.3 Random Walks and the Markov process

A classic visualisation of random behaviour is the random walk. Consider a completely drunk person who walks along a street and being drunk has no sense of direction. So this drunkard may move forwards with equal probability that he moves backwards.

A **Markov process** is a random walk with a selected probability for making a move. The new move is independent of the previous history of the system. The Markov chain is used repeatedly in Monte Carlo simulations in order to generate new random states.

In the context of a physical system containing many atoms, molecules etc, the different energy states are practically infinite. Hence, statistical approaches utilise algorithms to sample this large state-space and calculate average measurements such as energy and magnetisation. With Monte Carlo methods, we can explore and sample this state space using a random walk. The role of the Markov chain is to sample those states that make the most significant contributions.

The reason for choosing a Markov process is that when it is run for a long enough time starting with a random state, we will eventually reach the most likely state of the system. In thermodynamics, this means that after a certain number of Markov steps we reach an equilibrium distribution. This mimicks the way a real system reaches its most likely state at a given temperature of the surroundings.

To reach this distribution, the Markov process needs to obey two important conditions, that of ergodicity and detailed balance. These conditions impose then constraints on our algorithms for accepting or rejecting new random states.

The Metropolis algorithm discussed next abides to both these constraints. The Metropolis algorithm is widely used in Monte Carlo simulations and the understanding of it rests within the interpretation of random walks and Markov processes.

17.4 The Metropolis algorithm

In order to follow the predictions of a given statistical probability function such as a Boltzmann distribution, the samples of state-space need to be considered accordingly. Instead of sampling a lot of states and then weighting them by their Boltzmann probability factors, it makes more sense to choose states based on their Boltzmann probability and to then weight them equally. This is known as the Metropolis algorithm which has a characteristic cycle: 1. A trial configuration is made by randomly choosing one state 2. The energy difference, ΔE , of adopting this trial state relative to the present state is calculated. 3. If this reduces the total energy of the system, i.e. if $\Delta E \leq 0$, then the trial state is energetically favorable and thus accepted. 4. Otherwise, it will only be accepted if its probability is greater than some random number $\exp(-\Delta E/k_B T) > \eta$ where $0 \leq \eta \leq 1$.

Each cycle accepts or rejects a potential state and repeats testing many other states in a Markov process. The total number of cycles is typically the number of atoms, or bodies in the system. Obviously, the system must be allowed to reach thermal equilibrium before sampling the Boltzmann distribution in this way.

17.5 Ferromagnetism using the Ising Model

A ferromagnetic material is one that produces a magnetic field of its own, even without the presence of an external magnetic field. A ferromagnet can be any material that forms itself a permanent magnet, the magnitude of which is not reduced in the presence of any other magnetic fields.

A **paramagnet** is a material in which, with the presence of an external magnetic field, interatomic induced magnetic fields are formed, and therefore a magnetic field through the material is produced. However, once the external field is removed, the induced magnetic fields between atoms are lost and therefore the material can only have an induced magnetic field.

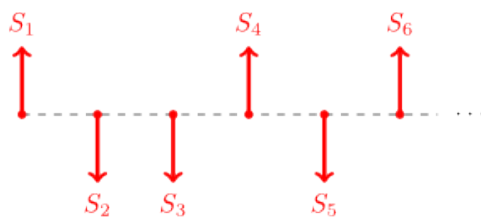
Ferromagnets contain finite-size domains in which the spins of all the atoms point in the same direction. When an external magnetic field is applied to these materials, the different domains align and the materials become “magnetised.” Yet as the temperature is raised, the total magnetism decreases, and at the Curie temperature the system goes through a phase transition beyond which all magnetisation vanishes.

Ising model can explain the thermal behaviour of ferromagnets.

The **Ising model** is the simplest model of a ferromagnet. The basic phenomenology of the Ising model is simple: there is a certain temperature T_c below which the system will spontaneously magnetise. This is what we will study with Monte Carlo.

```
[189]: Image(filename="ising.png",width=450)
```

[189]:



17.6 Ising Model

The model consists of an array of particles, with a spin value of either +1 or -1, corresponding to an up or down spin configuration respectively. Inside the lattice, spins interact with their ‘neighbours’, and each particle on the lattice has an associated interaction energy. The value of this interaction energy is dependent on whether neighbouring particles have parallel or anti-parallel spins. The interaction energy between two parallel spins is $-J$, and for anti-parallel spins; $+J$; where J is an energy coupling constant that is dependent on the material being simulated: - $J > 0$ corresponds to a ferromagnetic state in which spins tend to align with each other in order to minimize the energy. - $J < 0$, they prefer to be antiparallel and for a simple lattice that leads to a chessboard-like alignment, a feature of the anti-ferromagnetic state - $J = 0$, the spin alignment is arbitrary.

By summing the interaction energies of every particle on the lattice, the total energy, E , of the configuration can be obtained, and is given by equation:

$$E = -J \sum_{\langle i,j \rangle} S_i S_j \quad (3)$$

Where $\langle i, j \rangle$ represents nearest neighbours and $S_i = +1$ for an up spin and -1 for a down spin on site i . For a give spin, it’s local energy is calculated by summing over all the energies of each spin of it’s neighbours as given by:

$$E_i = -J \times S_i \times (S_{j1} + S_{j2} + S_{j3} + S_{j4}) \quad (4)$$

The change in energy of the system is dictated by the interaction of a dipole with its neighbours, so that flipping S_i to S'_i changes the energy by:

$$\Delta E = E_i - E'_i = 2J \sum_j S_i S_j \quad (5)$$

The two-dimensional square lattice Ising model is very difficult to solve analytically, the first such description was achieved by Lars Onsager in 1944, who solved the dependance:

$$\frac{K_b T_c}{J} = \frac{2}{\ln(1 + \sqrt{2})} \approx 2.269$$

His solution, although elegant, is rather complicated. We’re going to use the Monte Carlo method to see the effects that his solution describes.

We expect there is some temperature at which this phase transition happens - where the systems goes from being a Ferromagnet to a Paramagnet. This temperature was solved for exactly by Lars Onsager in 1944

The program starts at a certain given temperature and calculates whether the considered spin flips or not for a certain number of iterations. For each step we first performed 1 iterations to reach thermal equilibrium and then performed another 1/2 iterations to determine the physical quantities Energy per site, Magnetization per site, Magnetic Susceptibility, Specific Heat, Correlation Function and the Correlation Length.

17.6.1 Road map of Ising model:

We will consider a square two-dimensional lattice with periodic boundary conditions. Here, the first spin in a row ‘sees’ the last spin in the row and vice versa. The same applies for spins at the top and bottom of a column.

We will define individual functions for all the components of our model, such as: - creating a 2D grid of lattice of spins - randomly choose a spin - flip the spin - calculate nearest neighbour values - calculate energy and magnetisation of lattice - metropolis algorithm

The ising model will then simply start at some temperature, T , evolve to equilibrium and then evolve further to a steady state.

```
[190]: %matplotlib inline
import numpy as np
from numpy.random import rand
import matplotlib.pyplot as plt

[191]: def initialstate(N):
    ''' generates a random spin configuration for initial condition'''
    state = 2*np.random.randint(2, size=(N,N))-1
    return state

def mcmove(config, beta):
    '''Monte Carlo move using Metropolis algorithm'''
    for i in range(N):
        for j in range(N):
            a = np.random.randint(0, N)
            b = np.random.randint(0, N)
            s = config[a, b]
            nb = config[(a+1)%N,b] + config[a,(b+1)%N] + config[(a-1)%N,b]
            + config[a,(b-1)%N]
            cost = 2*s*nb
            if cost < 0:
                s *= -1
            elif rand() < np.exp(-cost*beta):
                s *= -1
            config[a, b] = s
    return config

def calcEnergy(config):
    '''Energy of a given configuration'''
    energy = 0
    for i in range(len(config)):
        for j in range(len(config)):
            S = config[i,j]
```

```

        nb = config[(i+1)%N, j] + config[i,(j+1)%N] + config[(i-1)%N, j]
        + config[i,(j-1)%N]
        energy += -nb*S
    return energy/4.

```

```

def calcMag(config):
    '''Magnetization of a given configuration'''
    mag = np.sum(config)
    return mag

```

```

[192]: ## change these parameters for a smaller (faster) simulation
nt      = 32          # number of temperature points
N       = 50          # size of the lattice, N x N
eqSteps = 1024        # number of MC sweeps for equilibration
mcSteps = 1024        # number of MC sweeps for calculation

T       = np.linspace(1.53, 3.28, nt);
E,M,C,X = np.zeros(nt), np.zeros(nt), np.zeros(nt), np.zeros(nt)
n1, n2  = 1.0/(mcSteps*N*N), 1.0/(mcSteps*mcSteps*N*N)
# divide by number of samples, and by system size to get intensive values

```

```

[193]: #-----
#  MAIN PART OF THE CODE
#-----
for tt in range(nt):
    E1 = M1 = E2 = M2 = 0
    config = initialstate(N)
    iT=1.0/T[tt]; iT2=iT*iT;

    for i in range(eqSteps):          # equilibrate
        mcmove(config, iT)           # Monte Carlo moves

    for i in range(mcSteps):
        mcmove(config, iT)
        Ene = calcEnergy(config)      # calculate the energy
        Mag = calcMag(config)         # calculate the magnetisation

        E1 = E1 + Ene
        M1 = M1 + Mag
        M2 = M2 + Mag*Mag
        E2 = E2 + Ene*Ene

    E[tt] = n1*E1
    M[tt] = n1*M1
    C[tt] = (n1*E2 - n2*E1*E1)*iT2
    X[tt] = (n1*M2 - n2*M1*M1)*iT

```

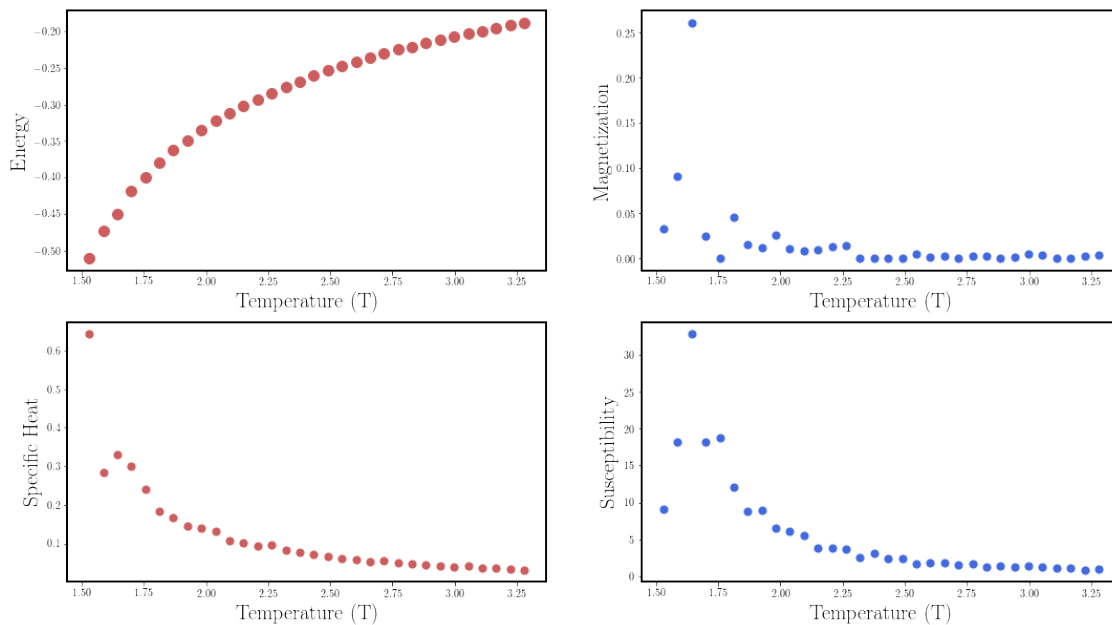
```
[234]: f = plt.figure(figsize=(18, 10)); # plot the calculated values

sp = f.add_subplot(2, 2, 1 );
plt.scatter(T, E, s=100, color='IndianRed')
plt.xlabel("Temperature (T)", fontsize=20);
plt.ylabel("Energy ", fontsize=20);          plt.axis('tight');

sp = f.add_subplot(2, 2, 2 );
plt.scatter(T, abs(M), s=50, marker='o', color='RoyalBlue')
plt.xlabel("Temperature (T)", fontsize=20);
plt.ylabel("Magnetization ", fontsize=20);    plt.axis('tight');

sp = f.add_subplot(2, 2, 3 );
plt.scatter(T, C, s=50, marker='o', color='IndianRed')
plt.xlabel("Temperature (T)", fontsize=20);
plt.ylabel("Specific Heat ", fontsize=20);    plt.axis('tight');

sp = f.add_subplot(2, 2, 4 );
plt.scatter(T, X, s=50, marker='o', color='RoyalBlue')
plt.xlabel("Temperature (T)", fontsize=20);
plt.ylabel("Susceptibility", fontsize=20);    plt.axis('tight');
```



```
[194]: %matplotlib inline
# Simulating the Ising model
from __future__ import division
import numpy as np
from numpy.random import rand
```

```

import matplotlib.pyplot as plt

class Ising():
    ''' Simulating the Ising model '''
    ## monte carlo moves
    def mcmove(self, config, N, beta):
        ''' This is to execute the monte carlo moves using
        Metropolis algorithm such that detailed
        balance condition is satisfied'''
        for i in range(N):
            for j in range(N):
                a = np.random.randint(0, N)
                b = np.random.randint(0, N)
                s = config[a, b]
                nb = config[(a+1)%N,b] + config[a,(b+1)%N] +
→config[(a-1)%N,b]
                + config[a,(b-1)%N]
                cost = 2*s*nb
                if cost < 0:
                    s *= -1
                elif rand() < np.exp(-cost*beta):
                    s *= -1
                config[a, b] = s
            return config

    def simulate(self):
        ''' This module simulates the Ising model'''
        N, temp = 64, .4 # Initiaalse the lattice
        config = 2*np.random.randint(2, size=(N,N))-1
        f = plt.figure(figsize=(15, 15), dpi=80);
        self.configPlot(f, config, 0, N, 1);

        msrmnt = 1001
        for i in range(msrmnt):
            self.mcmove(config, N, 1.0/temp)
            if i == 1: self.configPlot(f, config, i, N, 2);
            if i == 4: self.configPlot(f, config, i, N, 3);
            if i == 32: self.configPlot(f, config, i, N, 4);
            if i == 100: self.configPlot(f, config, i, N, 5);
            if i == 1000: self.configPlot(f, config, i, N, 6);

    def configPlot(self, f, config, i, N, n_):
        ''' This modules plts the configuration once passed to it along with
        →time etc '''
        X, Y = np.meshgrid(range(N), range(N))
        sp = f.add_subplot(3, 3, n_ )

```

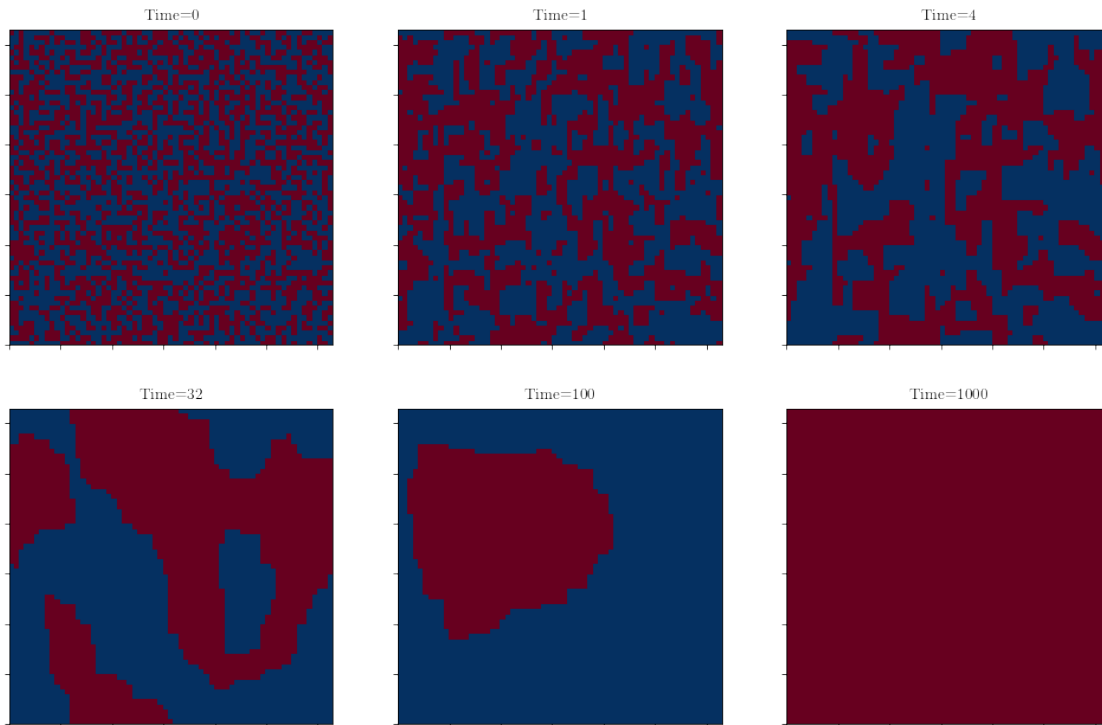
```
plt.setp(sp.get_yticklabels(), visible=False)
plt.setp(sp.get_xticklabels(), visible=False)
plt.pcolormesh(X, Y, config, cmap=plt.cm.RdBu);
plt.title('Time=%d'%i); plt.axis('tight')
plt.show()
```

```
[195]: rm = Ising()
```

```
[196]: rm.simulate()
```

<ipython-input-194-6dd4ce2880d5>:53: MatplotlibDeprecationWarning:
shading='flat' when X and Y have the same dimensions as C is deprecated since
3.3. Either specify the corners of the quadrilaterals with X and Y, or pass
shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading']. This
will become an error two minor releases later.

```
plt.pcolormesh(X, Y, config, cmap=plt.cm.RdBu);
```



17.6.2 Critical dynamics in a 1-D Ising model:

```
[197]: #Coding attempt MCMC 1-Dimensional Ising Model
import numpy as np
import matplotlib.pyplot as plt
```

```

#Using Probability Distribution given
def get_probability(delta_energy, Temperature):
    return np.exp(-delta_energy / Temperature)

def get_energy(spins):
    energy=0
    for i in range(len(spins)):
        energy=energy+interaction*spins[i-1]*spins[i]
    energy= energy-field*sum(spins)
    return energy

def delta_energy(spins,random_spin):
    #If you do flip one random spin, the change in energy is:
    #(By using a reduced formula that only involves the spin
    # and its neighbours)
    if random_spin==L-1:
        PBC=0
    else:
        PBC=random_spin+1

    old = -interaction*(spins[random_spin-1]*spins[random_spin]
        + spins[random_spin]*spins[PBC]) - ↵
    ↪field*spins[random_spin]
    new = interaction*(spins[random_spin-1]*spins[random_spin]
        + spins[random_spin]*spins[PBC]) + ↵
    ↪field*spins[random_spin]

    return new-old

def metropolis(L = 100, MC_samples=1000, Temperature = 1, interaction = 1, ↵
    ↪field = 0):

    # intializing
    #Spin Configuration
    spins = np.random.choice([-1,1],L)

    Beta = Temperature**(-1)

    #Introducing Metropolis Hastings Algorithm
    data = []
    magnetization=[]
    energy=[]
    for i in range(MC_samples):
        #Each Monte Carlo step consists in L random spin moves
        for j in range(L):
            #Choosing a random spin

```



```

random_spin=np.random.randint(0,L,size=(1))
#Computing the change in energy of this spin flip
delta=delta_energy(spins,random_spin)

#Metropolis accept-rejection:
if delta<0:
    #Accept the move if its negative
    spins[random_spin]=-spins[random_spin]
    #print('change')
else:
    #If its positive, we compute the probability
    probability=get_probability(delta,Temperature)
    random=np.random.rand()
    if random<=probability:
        #Accept de move
        spins[random_spin]=-spins[random_spin]

data.append(list(spins))

#Afer the MC step, we measure the system
magnetization.append(sum(spins)/L)
energy.append(get_energy(spins))

return data,magnetization,energy

def record_state_statistics(data,n=4):
    ix = tuple()

    sub_sample = [[d[i] for i in range(n)] for d in data]

    # get state number
    state_nums = [int(sum([(j+1)/2)*2**i for j,i in zip(
    ↪reversed(d),range(len(d)))))
        for d in sub_sample]

    return state_nums

# setting up problem
L = 200 # size of system
MC_samples = 1000 # number of samples
Temperature = 1 # "temperature" parameter
interaction = 1 # Strength of interaction between nearest neighbours
field = 0 # external field

```

```

# running MCMC
data = metropolis(L = L, MC_samples = MC_samples, Temperature = Temperature,
                 interaction = interaction, field = field)
results = record_state_statistics(data[0],n=4) # I was also interested in the
→probability
#of each micro-state in a sub-section of the system

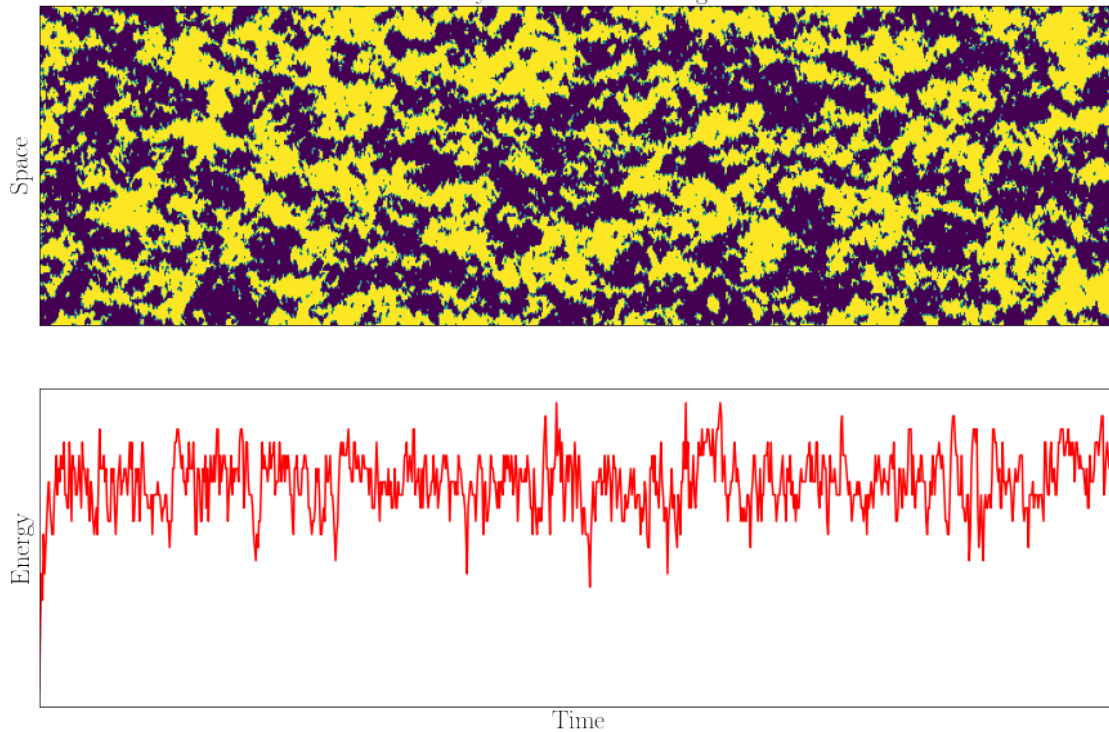
# Plotting
plt.figure(figsize=(15,10))

plt.subplot(2,1,1)
plt.imshow(np.transpose(data[0]))
plt.xticks([])
plt.yticks([])
plt.axis('tight')
plt.ylabel('Space',fontdict={'size':20})
plt.title('Critical dynamics in a 1-D Ising model',fontdict={'size':20})

plt.subplot(2,1,2)
plt.plot(data[2], 'r')
plt.xlim((0,MC_samples))
plt.xticks([])
plt.yticks([])
plt.ylabel('Energy',fontdict={'size':20})
plt.xlabel('Time',fontdict={'size':20});

```

Critical dynamics in a 1-D Ising model



17.7 Exercise 1: Setting up a 2D grid

We are to simulate a simplified 2D surface consisting of magnetic dipoles using the Ising approach. The Ising model represents a regular grid of points where each point has two possible states, spin up or spin down. States like to have the same spin as their immediate neighbors so when a spin-down state is surrounded by more spin-up states it will switch to spin-up and vice versa. Also, due to random fluctuations, points might switch spins, even if this switch is not favourable.

So to begin, we will define a function to create an initial $N \times M$ grid with magnetic spin values of ± 1 .

For this exercise create 3 functions: - `normallattice`: create $N \times M$ lattice with uniform spin values
- `randomlattice`: create $N \times M$ lattice with random spin values - `plotlattice`: plot an image of the lattice with a colour code for the spin

During your later investigations, you can use these functions to test if an initial random or uniform state is significant.

17.8 Exercise 2: Randomly choose & flip a lattice point

Implement a function to randomly select one of the particles in the lattice and return its coordinates (i, j) . Next create a function to flip the spin of the particle pointed by the (i, j) indices and return the new lattice state.

17.9 Exercise 3: Nearest Neighbour algorithm

A key element of this model is calculating the combined spin state of the 4 nearest neighbours around a given lattice point (i, j) .

Write a function to return the combined spin state and which respects periodic boundary conditions.

Perform some tests with a simple 5x5 lattice. Once you have convinced yourself that this functions correctly, add the next component to your model.

17.10 Exercise 4: Calculating the energy of the lattice

The local energy is defined as the total interaction energy between the selected particle and its immediate neighbours.

- (a) Write a function to calculate this.
- (b) Also write a function to calculate the total energy of the lattice.

Perform some tests with a simple 5x5 lattice, for example: - Compare the energy of the lattice for different configurations of spins. - What is the total energy of the system when all spins point up or down or randomly?

Once you have convinced yourself that this functions correctly, add the next component to your model.

17.11 Exercise 5: Calculate the magnetisation of the lattice

The total magnetisation of the lattice (M) is defined as:

$$M = \sum_i S_i \quad (6)$$

and the magnetisation per spin is:

$$m = \frac{1}{N_S} \sum_i S_i \quad (7)$$

where, N_S is the total number of spins on the lattice.

Implement functions to calculate these values.

Perform some tests with a simple 5x5 lattice, for example: - What will be the value of m if all spins are aligned up? what if all the spins are aligned down? What if half of the spins are up and half are down? - At the very beginning of your program you should have $m = 1.0$ as all the spins point up. Try to perturb a few particles and recalculate the magnetisation

Once you have convinced yourself that this functions correctly, add the next component to your model.

17.12 Exercise 6: Implement the Metropolis Algorithm

At this point in your code you should have all the necessary functions properly implemented and the thermodynamic simulation of the system can take place.

1. Set up the system in an initial configuration;
2. Choose one of the particles at random using a Markov Monte Carlo approach
3. Calculate the energy change ΔE of the system if the spin of the chosen particle is flipped
4. If ΔE is negative, then select to flip the spin and go to step 7, otherwise
5. Generate a random number r such that $0 < r < 1$
6. If this number is less than the probability of ΔE i.e. $r < \exp(-\Delta E/k_B T)$, then flip the spin.
7. Choose another spin of the lattice at random and repeat steps 2 to 6 a chosen number of times (N_{MCS})

Note, the Metropolis algorithm only contains $\Delta E/k_B T$, where k_B is the Boltzmann constant. Therefore, by defining $T' = k_B T/J$ the values of J and k_B are not required and you can work with T' as a dimensionless parameter independent on the material chosen. Hence the expression in step (6) reduces to $r < \exp(-\Delta E/T')$, with ΔE an integer number with values between -4 and 4.

It is usual to reject the first $N_{MCS}/2$ configurations in each Monte Carlo run in order to first establish thermalisation, and to consider only one configuration every N_S to avoid correlations.

For your final production run choose $N_{MCS} = 100000$ or a larger number, while you should use a smaller value of N_{MCS} for debugging. When using lattices of different size, comparable quality of results can be obtained using the same value of N_{MCS}/N_S , where N_S is the number of spin. This ratio is referred to as ‘number of **Monte Carlo configurations per spin**’ and indicates that an equal number of random choices is taken for each spin in the system.

17.13 Exercise 7: Perform measurements

This is the key section of your project where you get to perform statistical measurements of the 2D system. These measurements are magnetisation, magnetic susceptibility, energy and specific heat. If you perform many simulations at different temperatures, you should be in a position to observe phase transitions and measure the transition or Curie temperature, kT_c .

For a given temperature, we will wish to calculate the **average magnetisation**.

The average magnetisation per spin of the lattice is given by:

$$\langle m \rangle = \frac{1}{N_c} \sum_i^{N_S} m_i \quad (8)$$

where, N_C is the number of configurations included in the statistical averaging, and m_i is the value of the magnetisation for a given configuration.

Investigate the magnetisation over a range of temperature where the ferromagnetic to paramagnetic phase transition occurs. It is best to start at a low temperature and work upwards. Analyse the magnetization as a function of temperature and visualise the results. Also consider providing representative lattice images, below, at and above the transition temperature, kT_c (or even a nice animated gif of the full temperature scan!).

Identify and discuss phase transitions in the evolution of the system.

Consider benchmarking your numerical model for magnetisation. Can you find analytic solutions from research literature to compare with your numerical model predictions?

The **magnetic susceptibility** is another useful material parameter. This tells us how much the magnetisation changes by increasing the temperature. From the results of Exercise 8 it should be possible to calculate the magnetic susceptibility as a function of temperature. The magnetic susceptibility is calculated using:

$$\chi = \frac{1}{T}[\langle m^2 \rangle - \langle m \rangle^2] \quad (9)$$

Interprete and discuss your results.

According to the fluctuation dissipation theorem in statistical physics, the **specific heat** per spin of the lattice at temperature T is given by

$$C_V = \frac{\langle E^2 \rangle - \langle E \rangle^2}{N_S T^2} \quad (10)$$

where E is the energy of the lattice. The thermal averages $\langle E \rangle$ and $\langle E^2 \rangle$ can again be calculated by the Monte Carlo method using:

$$\langle E \rangle = \frac{1}{N_C} \sum E \quad (11)$$

Using this method, investigate the specific heat of the spin system in the vicinity of the phase transition.

17.14 Exercise 8: Calculate statistical errors and estimate finite size effects

Estimating the statistical errors is very important when performing Monte Carlo simulations i.e a single simulation may produce a fluke result! Also, the finite size of the 2D space can effect the measurements such as T_C .

Repeat exercises 8 and 9 by varying the size of the lattice to 32x32, 64x64 etc to estimate the finite size effects of 2D grid.

Run each lattice simulation a number of times in order to estimate the statistical errors of the measurements. Save your data in a text files for future analysis.

18 Python for Solid state physics

latticepy is a python package for modeling bravais lattices and constructing (finite) lattice structures.

A new instance of a lattice model is initialized using the unit-vectors of the Bravais lattice. After the initialization the atoms of the unit-cell need to be added. To finish the configuration the number of distances in the lattice need to be set. This computes the nearest distances between all atoms of the unit-cells. If only the nearest distance is computed the lattice will be set to nearest neighbors.

```
[198]: import numpy as np
from lattpy import Lattice

latt = Lattice(np.eye(2))      # Construct a Bravais lattice with square
    ↪ unit-vectors
latt.add_atom(pos=[0.0, 0.0]) # Add an Atom to the unit cell of the lattice
latt.set_num_neighbors(1)     # Set the maximum number of distances in the
    ↪ configuration.
```

```
[199]: from lattpy import simple_square

latt = simple_square(a=1.0, neighbors=1) # Initializes a square lattice with
    ↪ one atom in the unit-cel
```

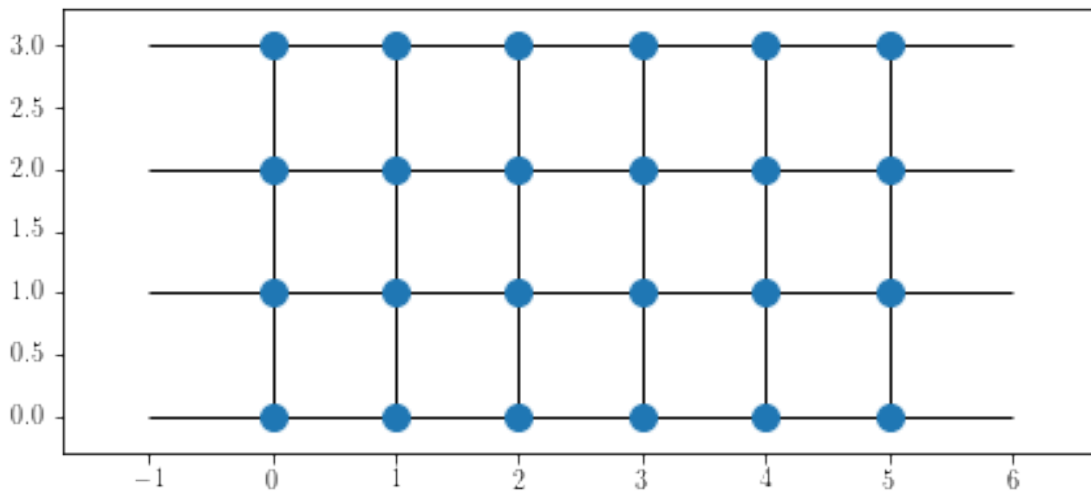
So far only the lattice structure has been configured. To actually construct a (finite) model of the lattice the model has to be built:

```
[200]: latt.build(shape=(5, 3));
#This will compute the indices and neighbors of all sites in the given shape
    ↪ and store the data.
```

```
[201]: #After building the lattice periodic boundary conditions can be set along one
    ↪ or multiple axes:
latt.set_periodic(axis=0)
```

```
[202]: from lattpy import simple_square

latt = simple_square(a=1.0, neighbors=1)
latt.build((5, 3), periodic=0)
latt.plot();
```



After configuring the lattice the attributes are available. Even without building a (finite) lattice structure all attributes can be computed on the fly for a given lattice vector, consisting of the translation vector \mathbf{n} and the atom index α . For computing the (translated) atom positions the `get_position` method is used. Also, the neighbors and the vectors to these neighbors can be calculated. The `dist_idx`-parameter specifies the distance of the neighbors (0 for nearest neighbors, 1 for next nearest neighbors, ...):

```
[203]: from lattpy import simple_square

latt = simple_square()

rvecs = latt.reciprocal_vectors()

# Get position of atom alpha=0 in the translated unit-cell
#positions = latt.get_position(n=[0, 0], alpha=0)

# Get lattice-indices of the nearest neighbors of atom alpha=0 in the
#→translated unit-cell
#neighbor_indices = latt.get_neighbors(n=[0, 0], alpha=0, distidx=0)

# Get vectors to the nearest neighbors of atom alpha=0 in the translated
#→unit-cell
#neighbor_vectors = latt.get_neighbor_vectors(alpha=0, distidx=0)
```

The 1st Brillouin zone is the Wigner-Seitz cell of the reciprocal lattice:

```
[204]: rlatt = latt.reciprocal_lattice()
bz = rlatt.wigner_seitz_cell()
```

The 1.BZ can also be obtained by calling the explicit method of the direct lattice:

```
[205]: bz = latt.brillouin_zone()
```

```
[206]: from lattpy import simple_square

latt = simple_square()
latt.build((5, 2))
idx = 2

# Get position of the atom with index i=2
positions = latt.position(idx)

# Get the atom indices of the nearest neighbors of the atom with index i=2
neighbor_indices = latt.neighbors(idx, distidx=0)

# the nearest neighbors can also be found by calling (equivalent to dist_idx=0)
neighbor_indices = latt.nearest_neighbors(idx)
```


18.1 Visualizing the reciprocal lattice in 2D:

```
[207]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import rc
rc('text', usetex=True)
```

Here we give the two real space lattice vectors:

```
[208]: # Pick these as you like
a1=np.array([1.0,0.0])
a2=np.array([np.cos(2.0*np.pi/3.0),np.sin(2.0*np.pi/3.0)])

# Vectors orthogonal to a1 and a2 (for plotting)
a1n=np.array([a1[1],-a1[0]])/np.sqrt(np.dot(a1,a1))
a2n=np.array([a2[1],-a2[0]])/np.sqrt(np.dot(a2,a2))
```

```
[209]: # 2N x 2N lattice points
N=4
nv=np.arange(-N,N)
mv=np.arange(-N,N)
# x-co-ordinates of the lattice points
xp=np.array([[i*a1[0]+j*a2[0] for i in nv for j in mv]])
yp=np.array([[i*a1[1]+j*a2[1] for i in nv for j in mv]])
```

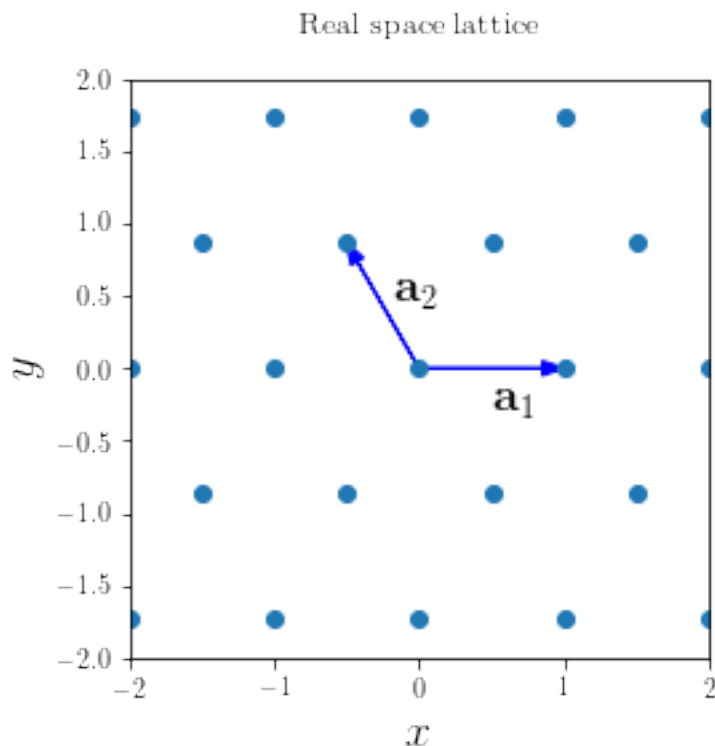
```
[210]: # Plot range
xmax=0.5*N*min(np.sqrt(np.dot(a1,a1)),np.sqrt(np.dot(a2,a2)))
# Plot things
plt.plot(xp.flatten(),yp.flatten(),'o')
plt.arrow(0,0,a1[0],a1[1],color='b',head_width=0.1,length_includes_head=True)
plt.arrow(0,0,a2[0],a2[1],color='b',head_width=0.1,length_includes_head=True)
plt.text(0.5*a1[0]+0.3*a1n[0],0.5*a1[1]+0.
→3*a1n[1],"$\mathbf{a}_{1}$",fontsize=18)
plt.text(0.5*a2[0]+0.1*a2n[0],0.5*a2[1]+0.
→1*a2n[1],"$\mathbf{a}_{2}$",fontsize=18)
plt.xlabel("$x$",fontsize=18)
plt.ylabel("$y$",fontsize=18)
plt.xlim(-xmax,xmax)
plt.ylim(-xmax,xmax)
plt.title("Real space lattice",y=1.05)
# The angles of the lines will look wrong if the aspect ratio is not equal
plt.axes().set_aspect('equal')
```

WARNING:py.warnings:<ipython-input-210-21b394a8c5d2>:15:

MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance. In a future version, a new

instance will always be created and returned. Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.

```
plt.axes().set_aspect('equal')
```



Now construct the unit cell

```
[211]: # Equations for the lines enclosing the unit cell

# Lines through the mid-point of a1
Xp1=[0.5*a1[0]+0.2*i*xmax*a2[0] for i in np.arange(-10,10)]
Yp1=[0.5*a1[1]+0.2*i*xmax*a2[1] for i in np.arange(-10,10)]
Xp2=[-0.5*a1[0]+0.2*i*xmax*a2[0] for i in np.arange(-10,10)]
Yp2=[-0.5*a1[1]+0.2*i*xmax*a2[1] for i in np.arange(-10,10)]

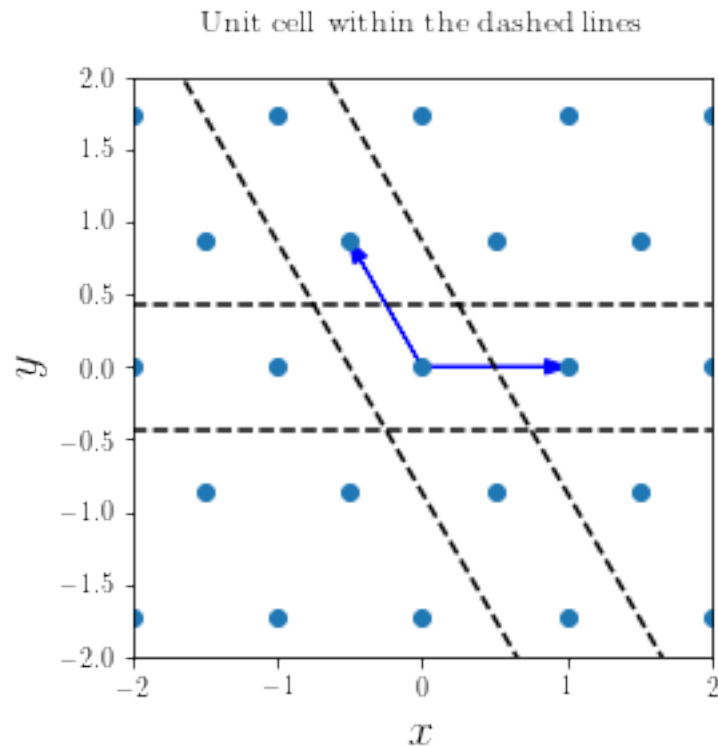
# Lines through the mid-point of a2
Xp3=[0.5*a2[0]+0.2*i*xmax*a1[0] for i in np.arange(-10,10)]
Yp3=[0.5*a2[1]+0.2*i*xmax*a1[1] for i in np.arange(-10,10)]
Xp4=[-0.5*a2[0]+0.2*i*xmax*a1[0] for i in np.arange(-10,10)]
Yp4=[-0.5*a2[1]+0.2*i*xmax*a1[1] for i in np.arange(-10,10)]
```

```
[212]: plt.plot(xp.flatten(),yp.flatten(),'o')
plt.plot(Xp1,Yp1,'k--')
plt.plot(Xp2,Yp2,'k--')
plt.plot(Xp3,Yp3,'k--')
plt.plot(Xp4,Yp4,'k--')
plt.arrow(0,0,a1[0],a1[1],color='b',head_width=0.1,length_includes_head=True)
plt.arrow(0,0,a2[0],a2[1],color='b',head_width=0.1,length_includes_head=True)
plt.xlabel("$x$",fontsize=18)
plt.ylabel("$y$",fontsize=18)
plt.xlim(-xmax,xmax)
plt.ylim(-xmax,xmax)
plt.title("Unit cell within the dashed lines",y=1.05)
# The angles of the lines will look wrong if the aspect ratio is not equal
plt.axes().set_aspect('equal')
```

WARNING:py.warnings:<ipython-input-212-cae24a4564b6>:14:

MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance. In a future version, a new instance will always be created and returned. Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.

```
plt.axes().set_aspect('equal')
```



18.1.1 The reciprocal space lattice:

```
[213]: b1=(2.0*np.pi/(a1[0]*a2[1]-a2[0]*a1[1]))*np.array([a2[1],-a2[0]])
      b2=(2.0*np.pi/(a1[0]*a2[1]-a2[0]*a1[1]))*np.array([-a1[1],a1[0]])

      # Vectors orthogonal to b1 and b2 (for plotting)
      b1n=np.array([b1[1],-b1[0]])/np.sqrt(np.dot(b1,b1))
      b2n=np.array([b2[1],-b2[0]])/np.sqrt(np.dot(b2,b2))

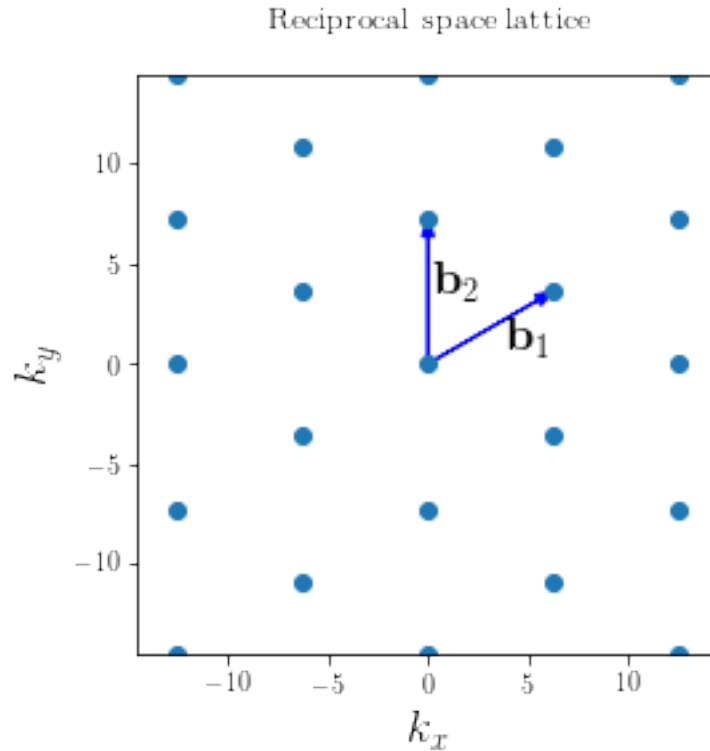
[214]: # 2N x 2N lattice points
      N=4
      nv=np.arange(-N,N)
      mv=np.arange(-N,N)
      # x-co-ordinates of the lattice points
      kxp=np.array([[i*b1[0]+j*b2[0] for i in nv] for j in mv])
      kyp=np.array([[i*b1[1]+j*b2[1] for i in nv] for j in mv])

[215]: # Plot range
      kxmax=0.5*N*min(np.sqrt(np.dot(b1,b1)),np.sqrt(np.dot(b2,b2)))
      # Plot things
      plt.plot(kxp.flatten(),kyp.flatten(),'o')
      plt.arrow(0,0,b1[0],b1[1],color='b',head_width=0.6,length_includes_head=True)
      plt.arrow(0,0,b2[0],b2[1],color='b',head_width=0.6,length_includes_head=True)
      plt.text(0.5*b1[0]+1.4*b1n[0],0.5*b1[1]+1.
        ↪4*b1n[1],"$\mathbf{b}_{1}$",fontsize=18)
      plt.text(0.5*b2[0]+0.3*b2n[0],0.5*b2[1]+0.
        ↪3*b2n[1],"$\mathbf{b}_{2}$",fontsize=18)
      plt.xlabel("$k_x$",fontsize=18)
      plt.ylabel("$k_y$",fontsize=18)
      plt.xlim(-kxmax,kxmax)
      plt.ylim(-kxmax,kxmax)
      plt.title("Reciprocal space lattice",y=1.05)
      # The angles of the lines will look wrong if the aspect ratio is not equal
      plt.axes().set_aspect('equal')
```

WARNING:py.warnings:<ipython-input-215-2e08d06ab26c>:15:

MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance. In a future version, a new instance will always be created and returned. Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.

```
plt.axes().set_aspect('equal')
```



18.1.2 Construction of the first Brillouin zone:

We don't do this in the same way as the unit cell, but construct the first Brillouin zone so that it has the same rotational symmetry as the lattice (i.e. for a hexagonal lattice this is 6-fold symmetry). To do this we take points half way between the origin (we call this the Γ point) and all the nearest neighbours. We draw lines orthogonal to these and the region within these lines is the first Brillouin zone.

```
[216]: # Vectors to nearest neighbours
v1=b1
v2=b2
v3=-b1
v4=-b2
v5=b2-b1
v6=b1-b2

# Vectors orthogonal to v1,v2,v3,v4,v5 and v6
v1n=np.array([v1[1],-v1[0]])
v2n=np.array([v2[1],-v2[0]])
v3n=np.array([v3[1],-v3[0]])
v4n=np.array([v4[1],-v4[0]])
v5n=np.array([v5[1],-v5[0]])
```

```

v6n=np.array([v6[1],-v6[0]])

# Lines along v1,v2,v3,v4,v5 and v6
l1=[0.5*v1+i*v1n for i in np.linspace(-10.0,10.0,10)]
l2=[0.5*v2+i*v2n for i in np.linspace(-10.0,10.0,10)]
l3=[0.5*v3+i*v3n for i in np.linspace(-10.0,10.0,10)]
l4=[0.5*v4+i*v4n for i in np.linspace(-10.0,10.0,10)]
l5=[0.5*v5+i*v5n for i in np.linspace(-10.0,10.0,10)]
l6=[0.5*v6+i*v6n for i in np.linspace(-10.0,10.0,10)]

plt.plot(kxp.flatten(),kyp.flatten(),'o')
plt.arrow(0,0,v1[0],v1[1],color='b',head_width=0.6,length_includes_head=True)
plt.arrow(0,0,v2[0],v2[1],color='b',head_width=0.6,length_includes_head=True)
plt.arrow(0,0,v3[0],v3[1],color='b',head_width=0.6,length_includes_head=True)
plt.arrow(0,0,v4[0],v4[1],color='b',head_width=0.6,length_includes_head=True)
plt.arrow(0,0,v5[0],v5[1],color='b',head_width=0.6,length_includes_head=True)
plt.arrow(0,0,v6[0],v6[1],color='b',head_width=0.6,length_includes_head=True)
plt.xlabel("$k_x$",fontsize=18)
plt.ylabel("$k_y$",fontsize=18)
plt.xlim(-kxmax,kxmax)
plt.ylim(-kxmax,kxmax)
plt.title("Nearest neighbours in reciprocal space",y=1.05)
# The angles of the lines will look wrong if the aspect ratio is not equal
plt.axes().set_aspect('equal')

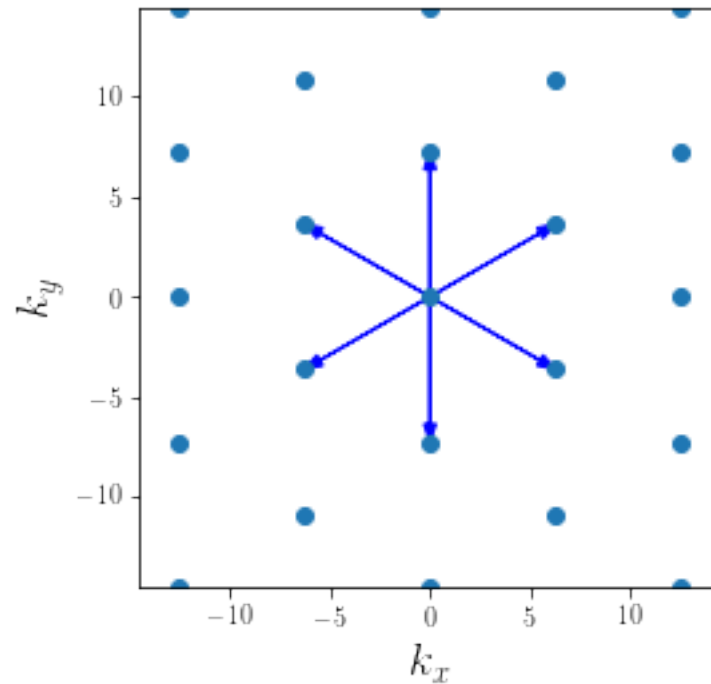
```

WARNING:py.warnings:<ipython-input-216-230363333a98>:38:

MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance. In a future version, a new instance will always be created and returned. Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.

```
plt.axes().set_aspect('equal')
```

Nearest neighbours in reciprocal space



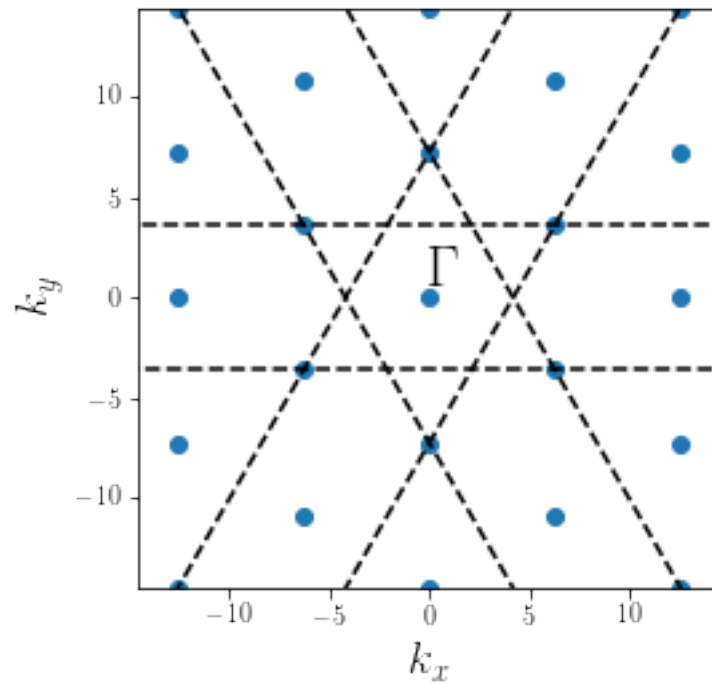
```
[220]: plt.plot(kxp.flatten(),kyp.flatten(),'o')
plt.xlabel("$k_x$",fontsize=18)
plt.ylabel("$k_y$",fontsize=18)
plt.xlim(-kxmax,kxmax)
plt.ylim(-kxmax,kxmax)
plt.plot([i[0] for i in l1],[i[1] for i in l1],'k--')
plt.plot([i[0] for i in l2],[i[1] for i in l2],'k--')
plt.plot([i[0] for i in l3],[i[1] for i in l3],'k--')
plt.plot([i[0] for i in l4],[i[1] for i in l4],'k--')
plt.plot([i[0] for i in l5],[i[1] for i in l5],'k--')
plt.plot([i[0] for i in l6],[i[1] for i in l6],'k--')
plt.text(0.0,.7,"$\Gamma$",fontsize=22)
plt.title("First Brillouin zone (central enclosed region)",y=1.05)
# The angles of the lines will look wrong if the aspect ratio is not equal
plt.axes().set_aspect('equal')
```

WARNING:py.warnings:<ipython-input-220-65eb105bfce7>:15:

MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance. In a future version, a new instance will always be created and returned. Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.

```
plt.axes().set_aspect('equal')
```

First Brillouin zone (central enclosed region)



```
[221]: import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
from matplotlib import animation, rc
from IPython.display import HTML
rc('text', usetex=True)
%matplotlib inline
```

```
[224]: # Area of unit cell
A=np.abs(a1[0]*a2[1]-a1[1]*a2[0])
```

```
[225]: b1=2.0*np.pi*np.array([a2[1],-a2[0]])/A
b2=2.0*np.pi*np.array([-a1[1],a1[0]])/A
```

```
[226]: # For a lattice of points [-Nmax,Nmax]x[-Nmax,Nmax]
Nmax=10

# Points of real space lattice
nv=np.arange(-Nmax,Nmax+1)
mv=np.arange(-Nmax,Nmax+1)
Xv=np.array([[i*a1[0]+j*a2[0] for i in nv] for j in mv])
```



```

Yv=np.array([[i*a1[1]+j*a2[1] for i in nv] for j in mv])

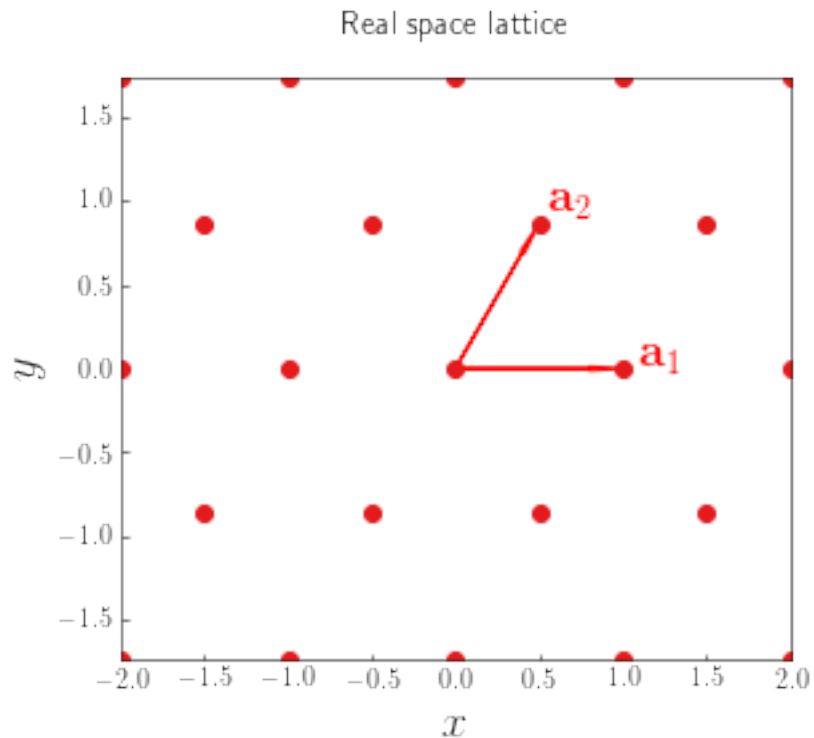
# Points of reciprocal space lattice
nv=np.arange(-Nmax,Nmax+1)
mv=np.arange(-Nmax,Nmax+1)
Kxv=np.array([[i*b1[0]+j*b2[0] for i in nv] for j in mv])
Kyv=np.array([[i*b1[1]+j*b2[1] for i in nv] for j in mv])

```

```

[73]: xd=max(a1[0],a2[0])
yd=max(a1[1],a2[1])
plt.plot(Xv.flatten(),Yv.flatten(),'o')
plt.xlim(-0.2*Nmax*xd,0.2*Nmax*xd)
plt.ylim(-0.2*Nmax*yd,0.2*Nmax*yd)
plt.xlabel("$x$",fontsize=18)
plt.ylabel("$y$",fontsize=18)
plt.title("Real space lattice",y=1.05)
plt.arrow(0,0,a1[0],a1[1],head_width=.025,head_length=.
↪2,length_includes_head=True,color='red')
plt.arrow(0,0,a2[0],a2[1],head_width=.025,head_length=.
↪2,length_includes_head=True,color='red')
plt.text(1.1*a1[0],1.1*a1[1],"${\\bf a}_1$",color='red',fontsize=18)
plt.text(1.1*a2[0],1.1*a2[1],"${\\bf a}_2$",color='red',fontsize=18)
plt.axes().set_aspect('equal')

```

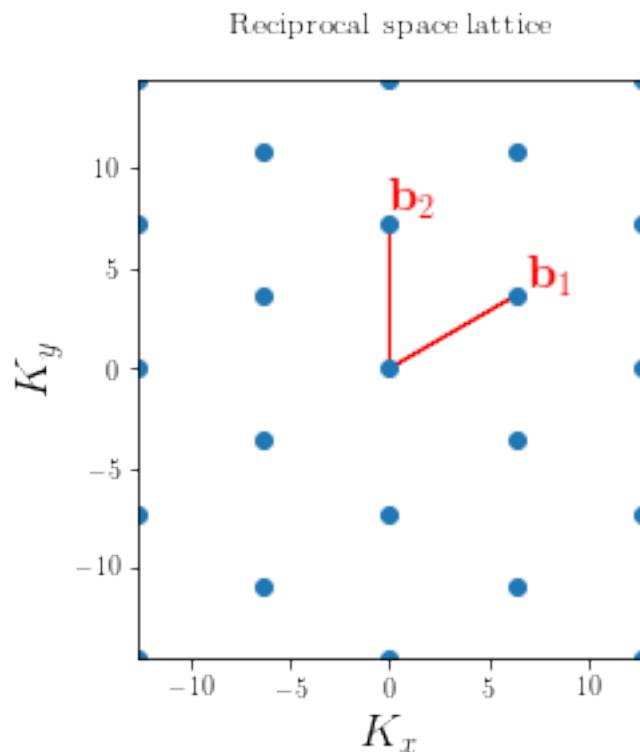


```
[227]: kxd=max(b1[0],b2[0])
kyd=max(b1[1],b2[1])
plt.plot(Kxv.flatten(),Kyv.flatten(),'o')
plt.xlim(-0.2*Nmax*kxd,0.2*Nmax*kxd)
plt.ylim(-0.2*Nmax*kyd,0.2*Nmax*kyd)
plt.xlabel("$K_x$",fontsize=18)
plt.ylabel("$K_y$",fontsize=18)
plt.title("Reciprocal space lattice",y=1.05)
plt.arrow(0,0,b1[0],b1[1],head_width=.025,head_length=.
↪2,length_includes_head=True,color='red')
plt.arrow(0,0,b2[0],b2[1],head_width=.025,head_length=.
↪2,length_includes_head=True,color='red')
plt.text(1.1*b1[0],1.1*b1[1],"${\\bf b}_1$",color='red',fontsize=18)
plt.text(1.1*b2[0],1.1*b2[1],"${\\bf b}_2$",color='red',fontsize=18)
plt.axes().set_aspect('equal')
```

WARNING:py.warnings:<ipython-input-227-4aa474c57bba>:13:

MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance. In a future version, a new instance will always be created and returned. Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.

```
plt.axes().set_aspect('equal')
```



The dispersion relation in the empty unit cell:

```
[228]: ns=[[0,0],[0,1],[1,0],[0,-1],[-1,0],[1,1],[-1,-1],[1,-1],[-1,1]]
# Consider a closed loop in the first Brillouin zone consisting of three points
P1=[0,0]
P2=0.5*(b1+b2)
P3=0.5*b2[1]*(b1+2*b2)/(b1[1]+2*b2[1])
```

```
[229]: for i in ns:
        v=i[0]*b1+i[1]*b2
        Xv=0.5*v[0]+np.linspace(-2,2,10)*v[1]
        Yv=0.5*v[1]-np.linspace(-2,2,10)*v[0]
        plt.plot(Xv,Yv,'k-')

plt.xlabel("$K_x$",fontsize=18)
plt.ylabel("$K_y$",fontsize=18)

# Draw the closed path
p1x=P1[0]+(P2[0]-P1[0])*np.linspace(0.0,1.0,5)
p1y=P1[1]+(P2[1]-P1[1])*np.linspace(0.0,1.0,5)
p2x=P2[0]+(P3[0]-P2[0])*np.linspace(0.0,1.0,5)
p2y=P2[1]+(P3[1]-P2[1])*np.linspace(0.0,1.0,5)
p3x=P3[0]+(P1[0]-P3[0])*np.linspace(0.0,1.0,5)
p3y=P3[1]+(P1[1]-P3[1])*np.linspace(0.0,1.0,5)
plt.plot(p1x,p1y,'r--')
plt.plot(p2x,p2y,'r--')
plt.plot(p3x,p3y,'r--')

plt.plot(P1[0],P1[1],'bo')
plt.plot(P2[0],P2[1],'bo')
plt.plot(P3[0],P3[1],'ro')

plt.text(P1[0],P1[1]+0.5,"$\Gamma$",fontsize=18)
plt.text(P2[0],P2[1]+0.5,"$M$",fontsize=18)
plt.text(P3[0],P3[1]+0.5,"$K$",fontsize=18)

plt.title("Our trip around the Brillouin zone",y=1.05)

plt.xlim(-6,6)
plt.ylim(-6,6)

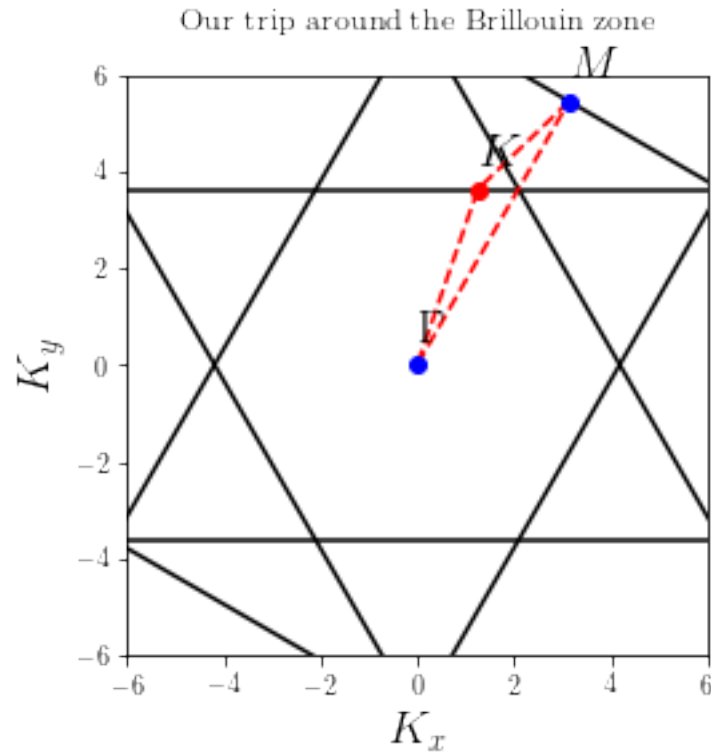
plt.axes().set_aspect('equal')
```

WARNING:py.warnings:<ipython-input-229-87e8b1b35b99>:34:

MatplotlibDeprecationWarning: Adding an axes using the same arguments as a

previous axes currently reuses the earlier instance. In a future version, a new instance will always be created and returned. Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.

```
plt.axes().set_aspect('equal')
```



```
[230]: # The dispersion relation
```

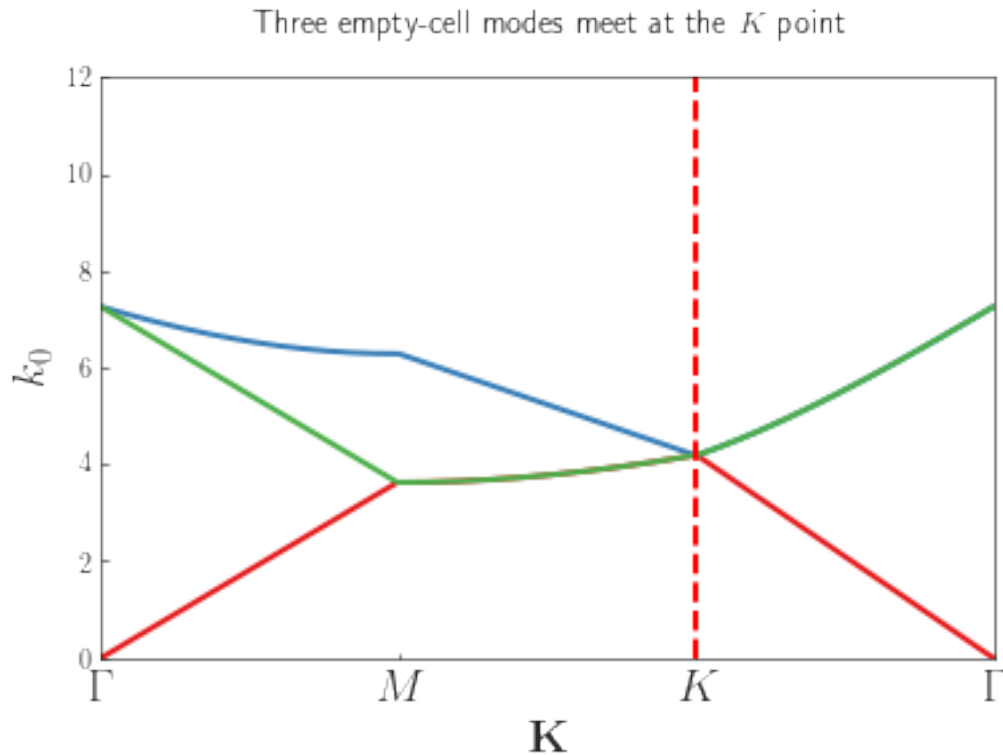
```
def k0(Kx,Ky,n,m):
    K=np.array([Kx,Ky])
    Knm=K+n*b1+m*b2
    res=np.sqrt(np.dot(Knm,Knm))
    return res
```

```
[231]: # The path in K space
```

```
K1=[P1+(P2-P1)*i for i in np.linspace(0.0,1.0,200)]
K2=[P2+(P3-P2)*i for i in np.linspace(0.0,1.0,200)]
K3=[P3+(P1-P3)*i for i in np.linspace(0.0,1.0,200)]
K=K1+K2+K3
# The three states degenerate at the K point
k0v0=[k0(i[0],i[1],0,0) for i in K]
k0v1=[k0(i[0],i[1],0,-1) for i in K]
```

```
k0v2=[k0(i[0],i[1],-1,-1) for i in K]
```

```
[83]: plt.plot(k0v0,lw=2)
plt.plot(k0v1,lw=2)
plt.plot(k0v2,lw=2)
plt.plot([400,400],[0,12],'r--',lw=2)
plt.xlabel("$\\bf{K}$",fontsize=18)
plt.ylabel("$k_0$",fontsize=18)
plt.title("Three empty-cell modes meet at the $K$ point",y=1.05)
plt.xticks([0,200,400,600],["$\\Gamma$", "$M$", "$K$", "$\\Gamma$"], fontsize=18);
```



```
[235]: # First define the three waves  $\phi = \phi_K \exp(i K x)$ 

def phi0(x,y):
    X=np.array([x,y])
    c=v[:,0]
    return (c[0]*psi0(x,y)+c[1]*psi1(x,y)+c[2]*psi2(x,y))*np.exp(1j*np.dot(K,X))
def phi1(x,y):
    X=np.array([x,y])
    c=v[:,1]
    return (c[0]*psi0(x,y)+c[1]*psi1(x,y)+c[2]*psi2(x,y))*np.exp(1j*np.dot(K,X))
def phi2(x,y):
    X=np.array([x,y])
```

```

c=v[:,2]
return (c[0]*psi0(x,y)+c[1]*psi1(x,y)+c[2]*psi2(x,y))*np.exp(1j*np.dot(K,X))

phi0=np.vectorize(phi0)
phi1=np.vectorize(phi1)
phi2=np.vectorize(phi2)

```

18.2 Tight-binding package for Python

Pybinding is a scientific Python package for numerical tight-binding calculations in solid state physics.

Install Pybinding using following command:

```
pip install pybinding
```

Let's create a triangular quantum dot of bilayer graphene and then apply a custom asymmetric strain function:

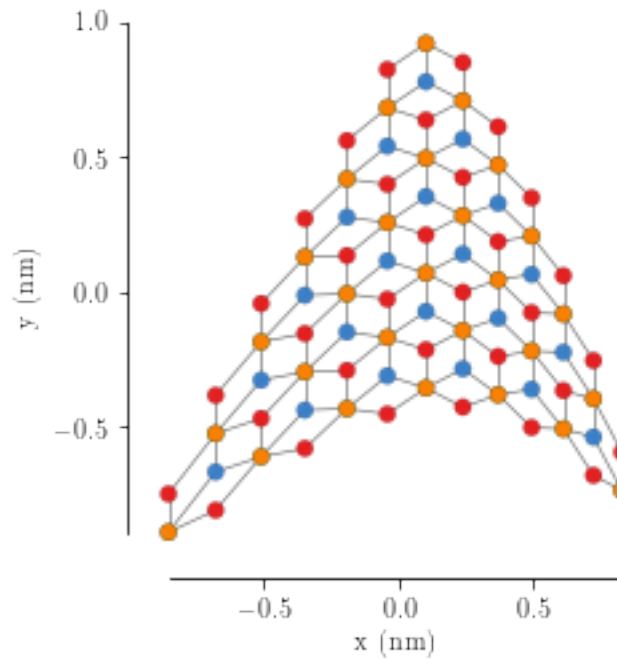
```

[237]: import pybinding as pb
from pybinding.repository import graphene

def asymmetric_strain(c):
    @pb.site_position_modifier
    def displacement(x, y, z):
        ux = -c/2 * x**2 + c/3 * x + 0.1
        uy = -c*2 * x**2 + c/4 * x
        return x + ux, y + uy, z
    return displacement

model = pb.Model(
    graphene.bilayer(),
    pb.regular_polygon(num_sides=3, radius=1.1),
    asymmetric_strain(c=0.42)
)
model.plot()

```

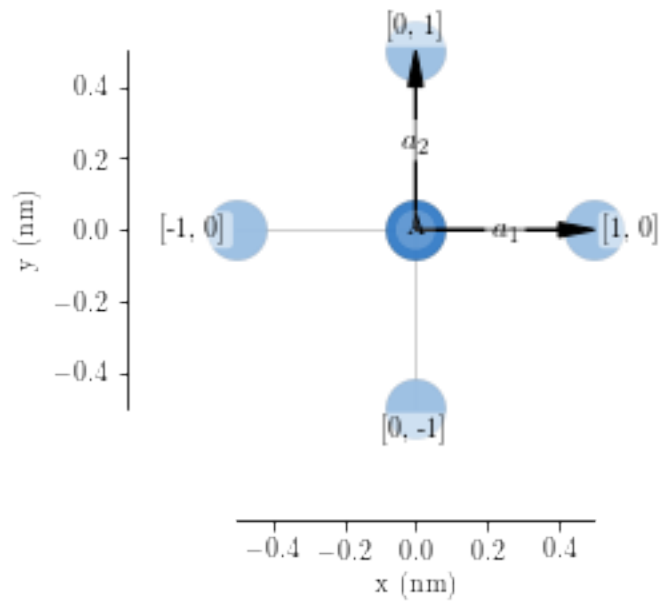


18.3 Square lattice using Pybinding package:

```
[239]: import pybinding as pb
import matplotlib.pyplot as plt

d = 0.5 # [nm] unit cell length
t = 2   # [eV] hopping energy

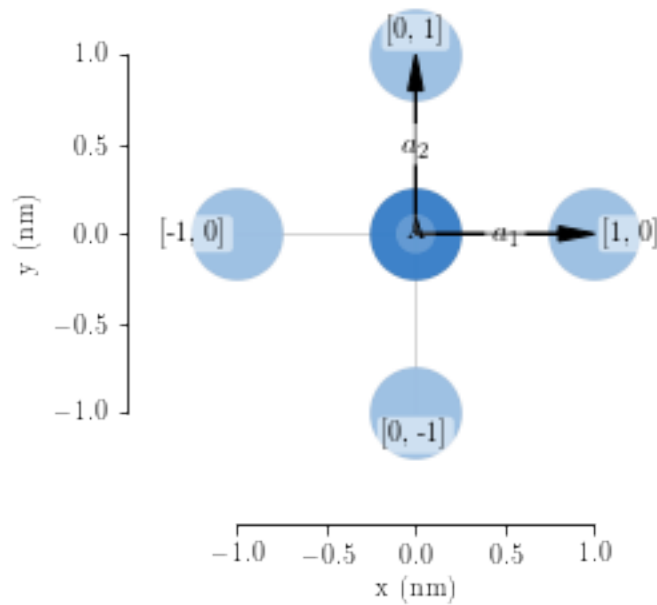
# create a simple 2D lattice with vectors a1 and a2
lattice = pb.Lattice(a1=[d, 0], a2=[0, d])
lattice.add_sublattices(
    ('A', [0, 0]) # add an atom called 'A' at position [0, 0]
)
lattice.add_hoppings(
    # (relative_index, from_sublattice, to_sublattice, energy)
    ([0, 1], 'A', 'A', t),
    ([1, 0], 'A', 'A', t)
)
lattice.plot() # plot the lattice that was just constructed
plt.show()    # standard matplotlib show() function
```



It's good practice to build the lattice inside a function to make it easily reusable. Here we define the same lattice as before, but note that the unit cell length and hopping energy are function arguments, which makes the lattice easily configurable.

```
[240]: def square_lattice(d, t):
    lat = pb.Lattice(a1=[d, 0], a2=[0, d])
    lat.add_sublattices(('A', [0, 0]))
    lat.add_hoppings(([0, 1], 'A', 'A', t),
                     ([1, 0], 'A', 'A', t))
    return lat

# we can quickly set a shorter unit length `d`
lattice = square_lattice(d=1, t=3)
lattice.plot()
plt.show()
```

18.4 Graphene using Python:

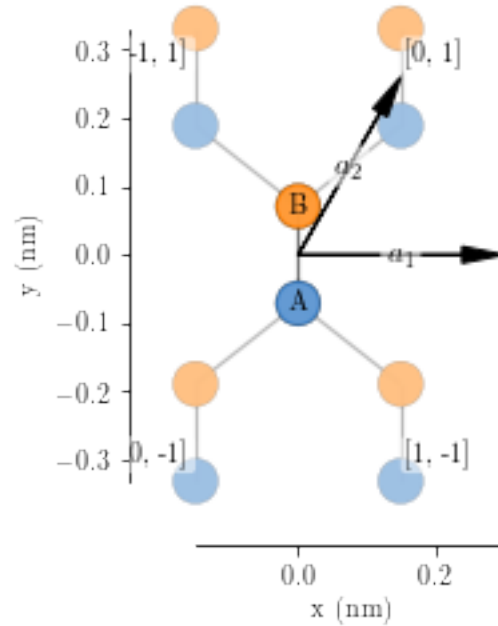
The next example shows a slightly more complicated two-atom lattice of graphene.

```
[242]: from math import sqrt

def monolayer_graphene():
    a = 0.3      # [nm] unit cell length
    a_cc = 0.142 # [nm] carbon-carbon distance
    t = -2.8     # [eV] nearest neighbour hopping

    lat = pb.Lattice(a1=[a, 0],
                    a2=[a/2, a/2 * sqrt(3)])
    lat.add_sublattices(('A', [0, -a_cc/2]),
                       ('B', [0, a_cc/2]))
    lat.add_hoppings(
        # inside the main cell
        ([0, 0], 'A', 'B', t),
        # between neighboring cells
        ([1, -1], 'A', 'B', t),
        ([0, -1], 'A', 'B', t)
    )
    return lat
```

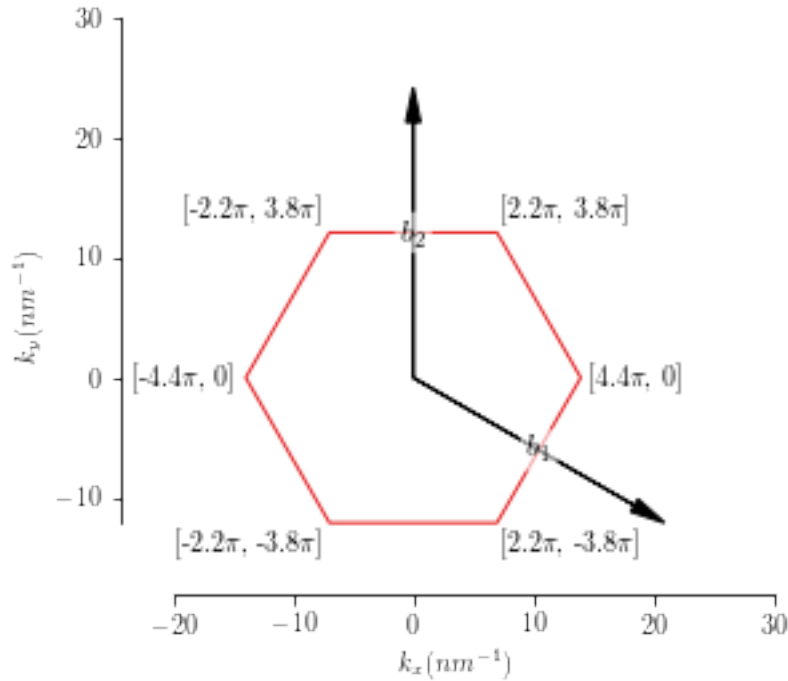
```
lattice = monolayer_graphene()
lattice.plot()
plt.show()
```



The `Lattice.add_sublattices()` method creates atoms A and B (blue and orange) at different offsets: $[0, -\text{acc}/2]$ and $[0, \text{acc}/2]$. Once again, the translated cells are given at positions $\mathbf{R} = n_1 \mathbf{a}_1 + n_2 \mathbf{a}_2$, however, this time the lattice vectors are not perpendicular which makes the integer indices $[n_1, n_2]$ slightly more complicated (see the labels in the figure).

18.5 Brillouin zone

```
[243]: lattice = monolayer_graphene()
lattice.plot_brillouin_zone()
```



18.6 Band structure:

```
[244]: import pybinding as pb
import numpy as np
import matplotlib.pyplot as plt

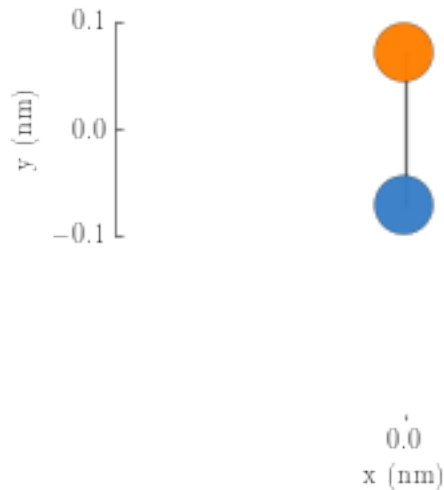
pb.pltutils.use_style()
%matplotlib inline
```

18.6.1 Model

A `Model` contains the full tight-binding description of the physical system that we wish to solve. We'll start by assigning a lattice to the model, and we'll use a pre-made one from the material repository.

```
[245]: from pybinding.repository import graphene

model = pb.Model(graphene.monolayer())
model.plot()
```



The result is not very exciting: just a single graphene unit cell with 2 atoms and a single hopping between them. The model does not assume translational symmetry or any other physical property. Given a lattice, it will just create a single unit cell. The model has a `System` attribute which keeps track of structural properties like the positions of lattice sites and the way they are connected, as seen in the figure above. The raw data can be accessed directly:

```
[246]: model.system.x
       model.system.y
       model.system.sublattices;
```

```
[247]: model.hamiltonian
```

```
[247]: <2x2 sparse matrix of type '<class 'numpy.float32'>'
       with 2 stored elements in Compressed Sparse Row format>
```

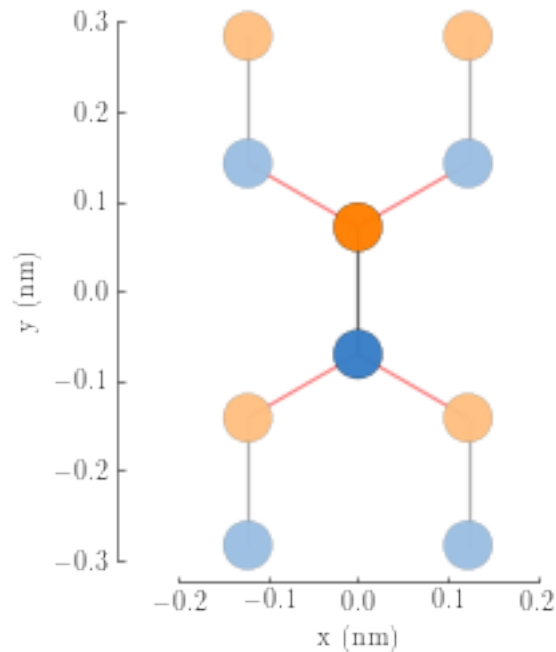
It's a sparse matrix (see `scipy.sparse.csr_matrix`) which corresponds to the tight-binding Hamiltonian of our model. The output above shows the default sparse representation of the data where each line corresponds to (row, col) value. Alternatively, we can see the dense matrix output:

```
[248]: model.hamiltonian.todense()
```

```
[248]: matrix([[ 0. , -2.8],
               [-2.8,  0. ]], dtype=float32)
```

Next, we include `translational_symmetry()` to create an infinite graphene sheet.

```
[249]: model = pb.Model(
        graphene.monolayer(),
        pb.translational_symmetry()
    )
    model.plot()
```



The red lines indicate hoppings on periodic boundaries. The lighter colored circles represent the translations of the unit cell. The number of translations is infinite, but the plot only presents the first one in each lattice vector direction.

18.7 Solver

A **Solver** can exactly calculate the eigenvalues and eigenvectors of a Hamiltonian matrix. We'll take a look at various **Eigenvalue solvers** and their capabilities in a later section, but right now we'll just grab the **lapack()** solver which is the simplest and most appropriate for small systems.

```
[250]: model = pb.Model(graphene.monolayer())
        solver = pb.solver.lapack(model)
        solver.eigenvalues
```

```
[250]: array([-2.8,  2.8], dtype=float32)
```

```
[251]: solver.eigenvectors
```

```
[251]: array([[ -0.70710677, -0.70710677],
              [ -0.70710677,  0.70710677]], dtype=float32)
```

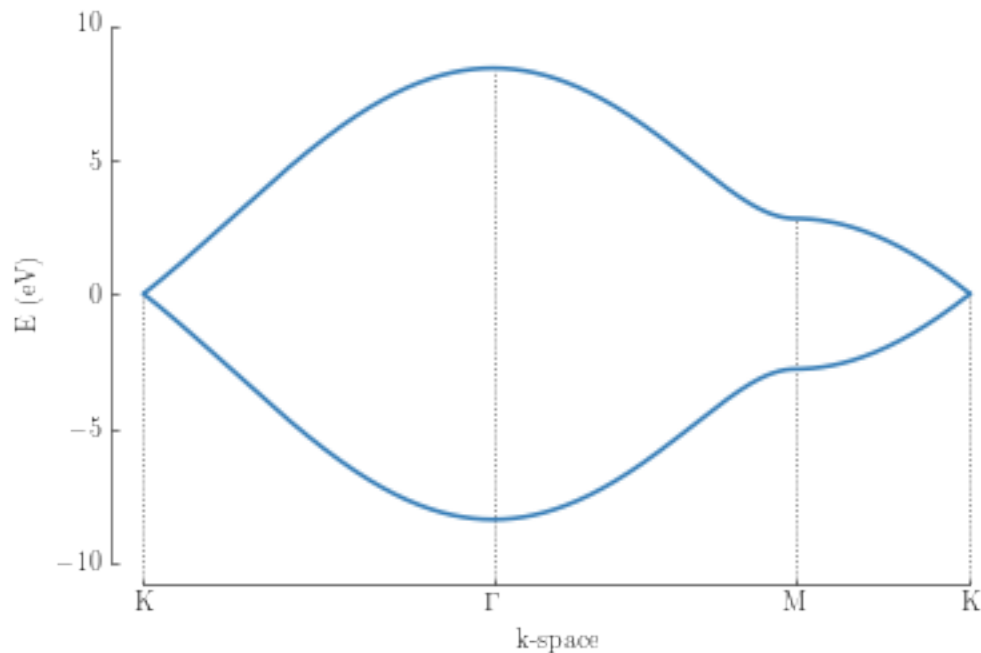
Beyond just the `eigenvalues` and `eigenvectors` properties, `Solver` has a convenient `calc_bands()` method which can be used to calculate the band structure of our model.

```
[252]: from math import sqrt, pi

model = pb.Model(graphene.monolayer(), pb.translational_symmetry())
solver = pb.solver.lapack(model)

a_cc = graphene.a_cc
Gamma = [0, 0]
K1 = [-4*pi / (3*sqrt(3)*a_cc), 0]
M = [0, 2*pi / (3*a_cc)]
K2 = [2*pi / (3*sqrt(3)*a_cc), 2*pi / (3*a_cc)]

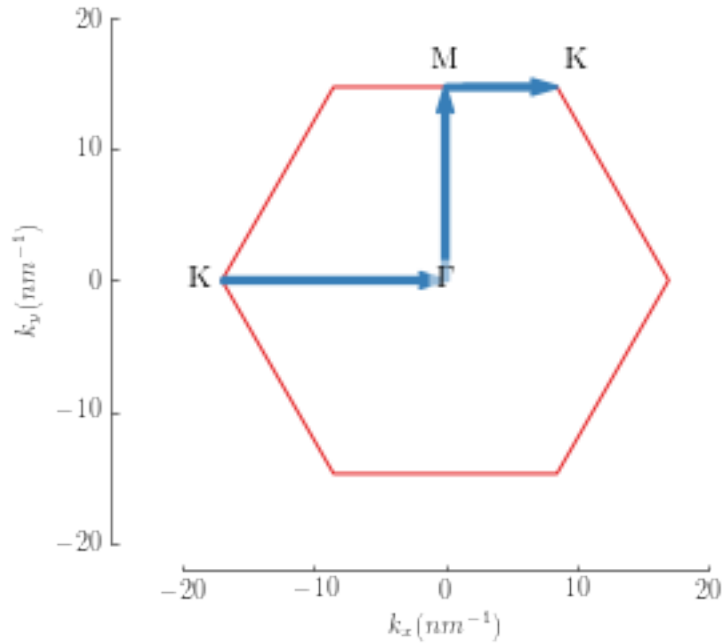
bands = solver.calc_bands(K1, Gamma, M, K2)
bands.plot(point_labels=['K', r'$\Gamma$', 'M', 'K'])
```



The points Γ , K and M are used to draw a path in the reciprocal space of graphene's Brillouin zone and `Solver.calc_bands()` calculates the band energy along that path. The return value of the method is a `Bands` result object.

All result objects have built-in plotting methods. Aside from the basic `plot()` seen above, `Bands` also has `plot_kpath()` which presents the path in reciprocal space. Plots can easily be composed, so to see the path in the context of the Brillouin zone, we can simply plot both:

```
[253]: model.lattice.plot_brillouin_zone(decorate=False)
bands.plot_kpath(point_labels=['K', r'$\Gamma$', 'M', 'K'])
```



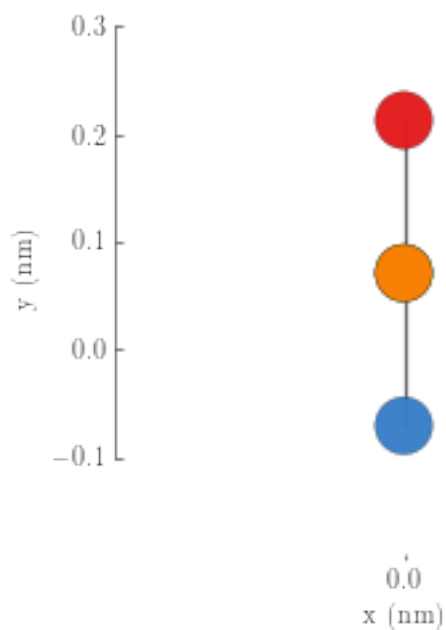
The extra argument for `Lattice.plot_brillouin_zone()` turns off the reciprocal lattice vectors and vertex coordinate labels (as seen in the previous section).

The band structure along a path in k-space can also be calculated manually by saving an array of `Solver.eigenvalues` at different k-points. This process is shown on the [Eigensolver](#) page.

18.8 Switching lattices

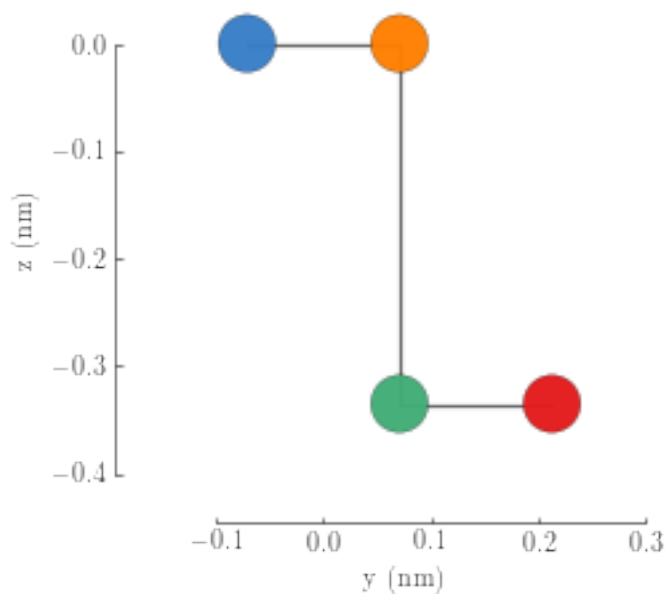
We can easily switch to a different material, just by passing a different lattice to the model. For this example, we'll use our pre-made `graphene.bilayer()` from the [Material Repository](#). But you can create any lattice as described in the previous section: [Lattice](#).

```
[254]: model = pb.Model(graphene.bilayer())
model.plot()
```



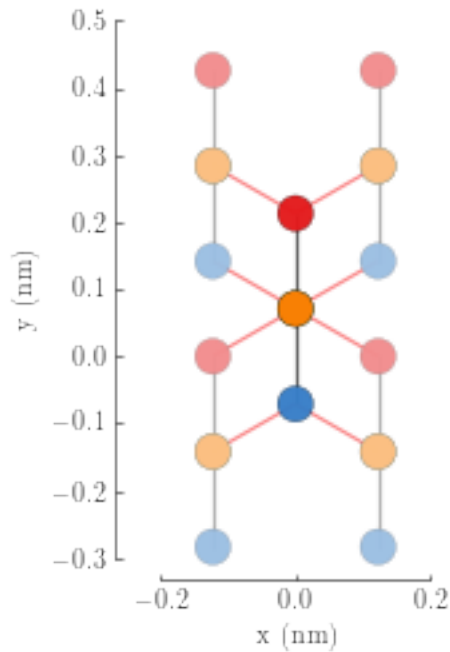
Without `translational_symmetry()`, the model is just a single unit cell with 4 atoms. Our bilayer lattice uses AB-stacking where a pair of atoms are positioned one on top of the another. By default, the `Model.plot()` method shows the xy-plane, so one of the bottom atoms isn't visible. We can pass an additional plot argument to see the yz-plane:

```
[255]: model = pb.Model(graphene.bilayer())
       model.plot(axes='yz')
```



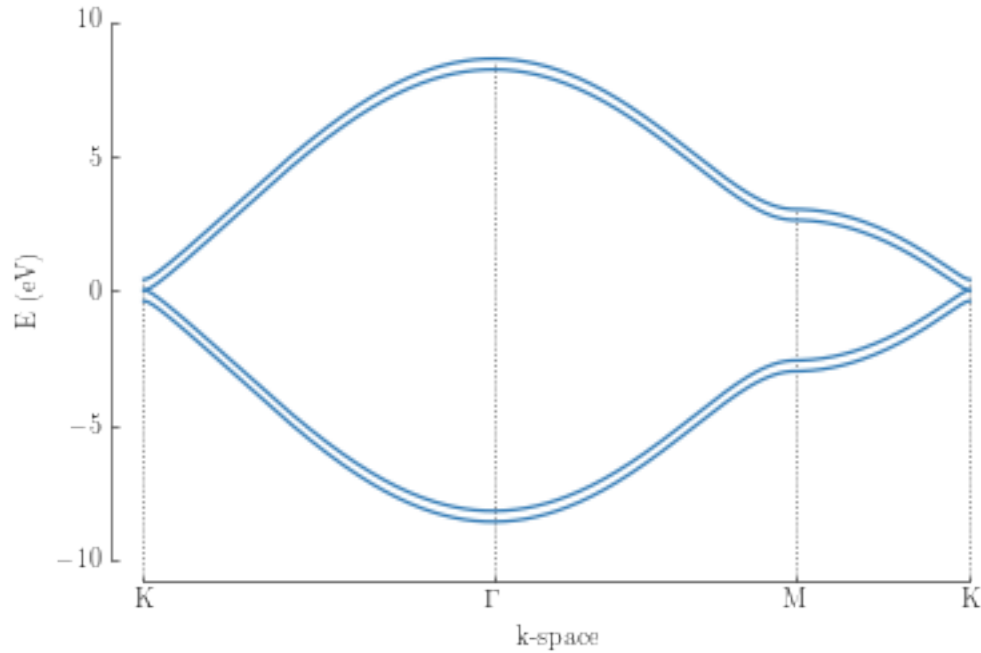
To compute the band structure, we'll need to include `translational_symmetry()`.

```
[256]: model = pb.Model(graphene.bilayer(), pb.translational_symmetry())
model.plot()
```



As before, the red hoppings indicate periodic boundaries and the lighter colored circles represent the first of an infinite number of translation units. We'll compute the band structure for the same Γ , K and M points as monolayer graphene:

```
[257]: solver = pb.solver.lapack(model)
bands = solver.calc_bands(K1, Gamma, M, K2)
bands.plot(point_labels=['K', r'$\Gamma$', 'M', 'K'])
```



19 Chaos Theory

Chaos is **aperiodic long-term** behavior in a **deterministic system** that exhibits **sensitive dependence on initial conditions**.

- aperiodic long-term**: there are trajectories which do not settle down to fixed points, periodic orbits, or quasiperiodic orbits as $t \rightarrow \infty$.
- deterministic system**: the system has no random or noisy inputs or parameters. The irregular behavior arises from the system's nonlinearity, rather than from noise.
- sensitive dependence on initial conditions**: nearby trajectories separate exponentially fast.

19.1 Chaos on a Strange Attractor:

Lorenz used numerical integration to see what the trajectories would do in the long run. He studied the particular case $\sigma = 10$, $\beta = 8/3$, $\rho = 28$. so he knew that something strange had to occur. Of course, strange things could occur for another reason the electromechanical computers of those days were unreliable and difficult to use, so Lorenz had to interpret his numerical results with cautions. He began integrating from the initial condition $(0, 1, 0)$, close to the saddle point at the origin.

19.2 Lorenz equations :

The Lorenz system is a system of ordinary differential equations first time studied by E.N. Lorenz. Lorenz who tried to predict the weather with computers by solving a system of ordinary differential equations for certain parameter values and initial conditions but instead gave rise to the modern field of chaos theory. In particular, the Lorenz attractor is a set of chaotic solutions of the Lorenz system.

$$\begin{aligned}x' &= \sigma(y - x) \\y' &= x(\rho - z) - y \\z' &= xy - \beta z\end{aligned}$$

The Lorenz equations are made up of three populations: x, y and z and three fixed coefficient: σ, ρ and β . This system of equations has the properties common to most other complex systems, such as lasers, dynamos, DC motors, electric circuits etc. Where σ is Prandtl number, ρ is the Rayleigh number and β has no name.

Nonlinearity: The system of equations has only two nonlinearities, the quadratic terms xy and xz . It is made up of very few simple components. The system is three-dimensional and deterministic.

Symmetry: There is an important symmetry in the Lorenz equations. If we replace $\{x, y\} \rightarrow \{-x, -y\}$ the equations remain the same. Hence, if $[x\{t\}, y\{t\}, z\{t\}]$ is a solution, so is $[-x\{t\}, -y\{t\}, -z\{t\}]$. In other words, all solutions are either symmetric themselves, or have a symmetric partner.

19.3 Butterfly effect:

In chaos theory, the butterfly effect is the sensitive dependence on initial conditions in which a small change in one state of a deterministic nonlinear system can result in large differences in a later state.

The term is closely associated with the work of mathematician and meteorologist Edward Lorenz. He noted that butterfly effect is derived from the metaphorical example of the details of a tornado (the exact time of formation, the exact path taken) being influenced by minor perturbations such as a distant butterfly flapping its wings several weeks earlier. Lorenz discovered the effect when he observed runs of his weather model with initial condition data that were rounded in a seemingly inconsequential manner. He noted that the weather model would fail to reproduce the results of runs with the unrounded initial condition data. A very small change in initial conditions had created a significantly different outcome.

```
[258]: import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from mpl_toolkits.mplot3d import Axes3D

rho = 28.0
sigma = 10.0
beta = 8.0 / 3.0
```

```

def f(state, t):
    x, y, z = state # Unpack the state vector
    return sigma * (y - x), x * (rho - z) - y, x * y - beta * z # Derivatives

state0 = [1.0, 1.0, 1.0]
t = np.arange(0.0, 50.0, 0.02)

states = odeint(f, state0, t)

fig = plt.figure()
ax = fig.gca(projection="3d")
ax.plot(states[:, 0], states[:, 1], states[:, 2])
plt.draw()
plt.axis('off');

plt.show()

```



Let's try to understand figure in detail. The trajectory starts near the origin, then swings to the right, and then dives into the center of a spiral on the left. After a very slow spiral outward, the trajectory shoots back over to the right side, spirals around a few times, shoots over to the left, spirals around, and so on indefinitely. The number of circuits made on either side varies unpredictably from one cycle to the next. In fact, the sequence of the number of circuits has many of the characteristics of a random sequence.