

Practical Computational Methods using Python

Science & Engineering Applications

Mahendra Verma

Copyright © 2020 Mahendra Verma

All rights reserved.

ISBN:

ISBN-13:

CONTENTS

Preface	1
Part-I	2
Chapter One	4
Chapter Two	24
Chapter Three	40
3.1	41
3.2	49
3.3	64
Chapter Four	72
Chapter Five	106
Chapter Six	122
Chapter Seven	144
Chapter Eight	162
8.1	163
8.2	181
Chapter Nine	188
9.1	189
9.2	201
Part-II	208
Overview	209
Chapter Ten	210
Chapter Eleven	228
11.1	229

11.2	237
11.3	252
Chapter Twelve	258
12.1	259
Ordinary Differential Equation Solvers	266
13.1	267
13.2	271
13.3	278
13.4	283
13.5	289
13.6	294
13.7	301
Chapter Fourteen	306
14.1	307
14.2	320
14.3	325
14.4	332
Chapter Fifteen	342
Solving Diffusion Equation Using Finite Difference Method	343
15.2	351
Burgers Equation	354
Naiver-Stokes Equation: Finite Difference Method	??
Schrodinger	357
Chapter Sixteen	362

Chapter Seventeen	376
PDE Solvers: Elliptic Equations	??
Chapter Eighteen	386
Working with randomness	??
Random numbers	??
Regression	??
Integration	??
Epilogue	??
References	??
Acknowledgements and Credits	??
Appendix A: Error in Lagrange Interpolation	??
Appendix B: Improving Accuracy by Richardson Method	??

PREFACE Hello

PART-I

Introduction to Computers

&

Python Programming

CHAPTER ONE
INTRODUCTION

INTRODUCTION TO COMPUTING

Computers everywhere..

Focus on Computing

1.1 Computer Hardware

A computer is a general-purpose digital device that can perform many tasks, e.g., compute numbers, control other devices, print things, etc. In this book we will discuss how computers aid in numerical computations.

For an efficient use of a car, it is best to know some of its details: its milage, power of the engine, nature of the brakes, etc. Similarly, an optimal use of a computer requires some knowledge about its computational capabilities that include memory capacity, power of the processors, storage capacity, etc. In this chapter, we provide a basic overview of a computer and its components that are critical for numerical computations.

We start with anatomy of a laptop, a device that every college student either owns or has access to.

Exterior of a laptop

On the exterior, a laptop contains a screen, a keyboard, a trackpad, a camera, and some ports of a laptop that are typically located on the left and right sides of the laptop. See Figure 1 for an illustration.

Every student is familiar with the keyboard, trackpad, and camera, which are input devices, and laptop screen, which is an output device. In addition, a laptop has various ports that are shown in [Figure 1](#). Older laptop have USB ports, Audio in/out, firewire port, and ethernet port. Modern laptops have one or several USB-C ports (see [Figure 1\(c\)](#)), which are used for charging the laptop and for connecting to USB drive, external monitors, TV, projector, etc. via a USB-C hub. A laptop is connected to the outside world using these input/output ports and devices such as screen, printer, etc.

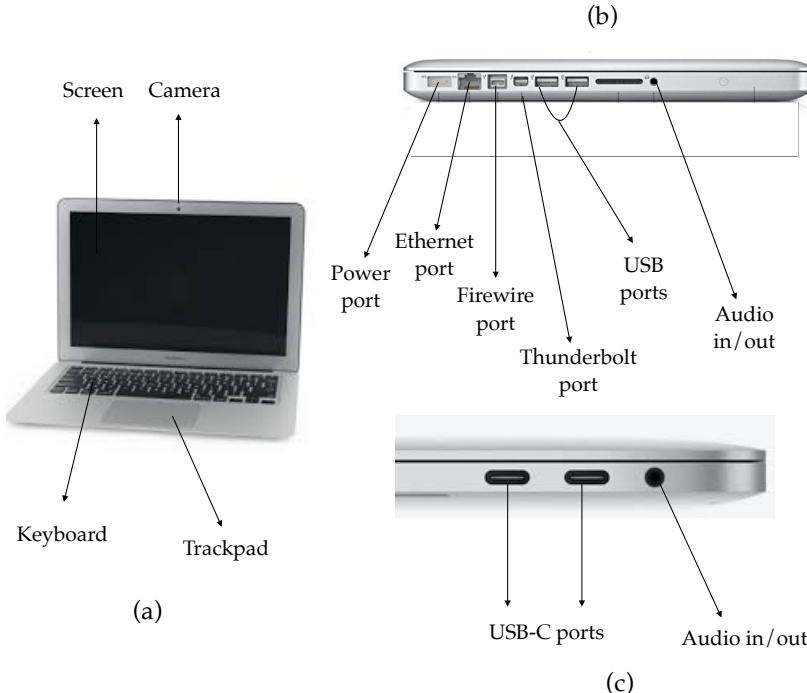


Figure 1: (a) Laptop; (b,c) Ports of a laptop. Adopted from a figure of wikipedia.org

Interior a laptop

After this, we peep inside a laptop. The processor, memory, connections to input/output ports, and many other small units reside on a printed circuit board (PCB) called *Motherboard* or *logic board*, which is illustrated in [Figure 2](#). In addition, hard disk (HD) or solid-state drive (SSD), battery, wireless device, and camera are kept inside the laptop. The inputs to the laptop via ports, keyboard, and camera arrive at the processor, who operates on them and sends the results to the memory or to appropriate output units (for example, screen or printer). The website <https://www.nemolaptops.com/post/2018/09/11/antamoy-of-a-laptop> contains a good overview of these units.

Let us get a deeper perspectives on the processor, memory, and hard disk.

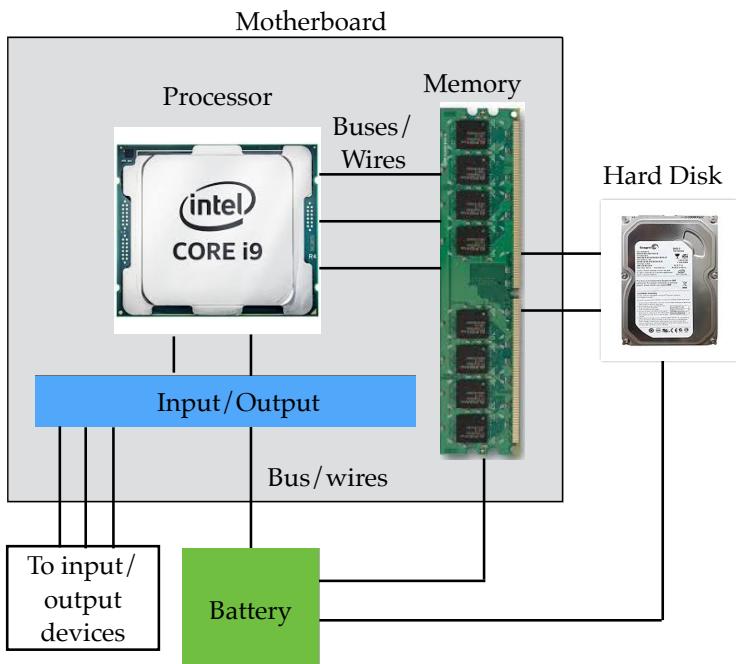


Figure 2: A schematic diagram of the internal parts of a laptop: Processor, Memory, Hard Disk, Battery.



https://www.youtube.com/watch?v=jxPO_O3gCIA

Processor: The processor, also called *central processing unit* (CPU), is the most complex and critical device of a computer. This unit can perform billions of arithmetic and logical operations per second. For example, the processor adds two numbers A and B . An important thing to note is that the numbers A , B , and $A+B$ are stored in a small memory, called *registers*, of the CPU.

Now we provide a quick overview of the capabilities of current laptop/desktop processors (year 2020). One of the best laptop processor available today is AMD Ryzen 9 that has 12 cores (processor subunits) that can operate at 3.8 Giga Hz. The maximum power consumed by this processor is 105 Watts. Intel has processors with similar capabilities. Recently, Apple has announced its own processor, which has comparable performance.

Memory: The data and programs reside in computer's *memory* (green strip inside the motherboard of [Figure 2](#)), also called *random access memory* (RAM). The CPU reads the program and data from RAM and write the results back to it. Note that the memory is active only till the laptop is powered, hence it is temporary. We need to write the data to the hard disk for permanent storage.

A laptop or desktop has RAM in the range of 4 Gigabytes to 64 Gigabytes. Note that 1 byte = 8 bits, and

$$1 \text{ Kilobyte} = 1\text{KB} = 2^{10} \approx 10^3$$

$$1 \text{ Megabyte} = 1\text{MB} = 2^{20} \approx 10^6$$

$$1 \text{ Terabyte} = 1\text{TB} = 2^{40} \approx 10^{12}$$

$$1 \text{ Petabyte} = 1\text{PB} = 2^{50} \approx 10^{15}$$

$$1 \text{ Exabyte} = 1\text{EB} = 2^{60} \approx 10^{18}$$

The clock speed of RAM ranges from 0.667 GHz to 1.6 GHz, which is slower than processor's clock speed. The best laptop RAM available at present, DDR4 (double data rate 4), transfers data at the rate of 4 to 16 Gigabits/second, which is quite slow compared to CPU's data processing capability (see next section). Following strategies are adopted to offset this deficiency of RAM: (a) The motherboard has multiple channels (like roads for data travel) between CPU and RAM; (b) CPU has its own memory called *cache*. The data which is needed immediately is kept in cache for fast access.

Hard disk of Solid-state disk (SSD): These are permanent storage of the computer. The program and data reside here when the laptop is turned off. On turning on of the laptop, the programs and data are transferred to the RAM; this process is called boot up,. Hence, the CPU, RAM, and hard disk continuously interact with each other.

A hard disk is an electro-magnetic device in which magnetic heads read data from the spinning magnetic disks. In these devices, the data transfer rate to RAM is 100-200 Megabytes (MB) per second. Due to the moving parts, such devices are prone to failures, specially in laptops during their movements. In the market, we can buy hard disk with capacities ranging from 1 TB to 12 TB.

On the other hand, a SSD is purely electronic device with no spinning parts. Hence, SSDs are safer than hard disks, but they cost more. The data transfer rate in SSD is around 500 MB per second. The capacity of SSD ranges from 128 GB to 1 TB.

Computers come in different variants, but their basic design remains the same. *Desktops*, which are typically more powerful than laptops, sit on desks. *Workstations* or *compute servers* have strong CPUs and large memory, hence they are more powerful than desktops. Supercomputers that consist of a large number of server-grade

processors and memory units are even more powerful. We will detail servers and supercomputers further in the next section.

Mobiles and tablet too are computers. They too have processors and memory and perform similar operations, but they weaker than laptops.

These *hardware* units by themselves cannot perform any task. A complex program called *Operating System* (OS) makes the system alive. The OS, applications (such as Microsoft Word), and user programs are called *software*, and they will be briefly described in the next chapter.

Conceptual questions

1. Why do computers have hierarchy of memory devices—cache, RAM, hard disk?
2. What are the advantages and disadvantages of USB-C port over USB port?
3. What are the similarities and dissimilarities between the functioning of a computer and a human brain?

Exercises

1. List the following for your laptop/desktop and your mobile phone: RAM size, CPU clock speed, Hard disk or SSD capacity.
2. It is best to see the parts of an opened-up desktop. Don't open your laptop because it is tricky.
3. Does hard disk/solid-state disk sit on the motherboard of a laptop? If not, how is connected to the memory or CPU?

1.2 Servers and Supercomputers

Laptops have limited power and they cannot perform complex tasks, such as banking, weather predictions, simulations of aircrafts. For the above operations, we employ servers or supercomputers.

Fugaku is the fastest supercomputer as of June 2020. It consists of large number of racks, each of which contains many nodes with processors and RAM. See Figure 3 in which each blue box is a rack. The nodes are connected to each other via a fast switch called *interconnect*. Note that *Fugaku* requires 28 Megawatts of power, which is the power consumed by a typical middle-sized town. For reference, we provide the following specifications of *Fugaku*:

- 158,976 nodes each with 48-core A64FX processor (2 GHz clock speed). Total cores: 7,299,072
- Total memory: 4,866,048 GB
- Peak speed: 415,530 Tera floating-point operations per second (FLop / s)
- Interconnect: Tofu interconnect D
- Power requirement: 28,334.5 kilowatts

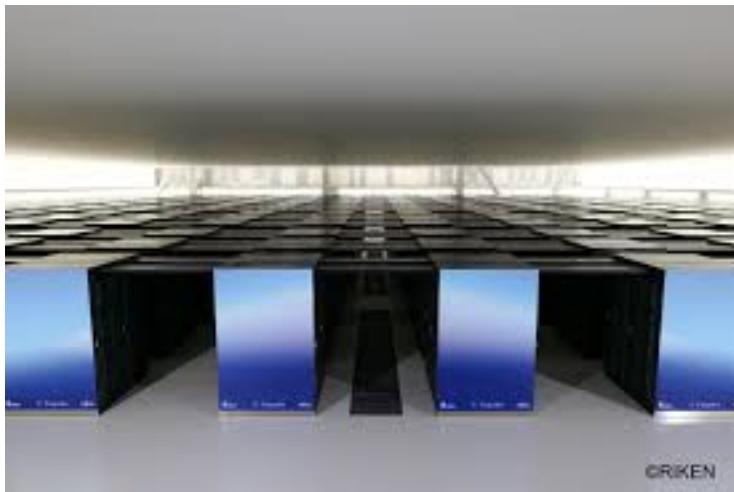


Figure 3: *Fugaku*, world's fastest supercomputer as on June 2020.

Server Processors

The processors of servers or supercomputers are much more powerful

than those of laptops. AMD's *Rome* 7742 (EPYC series) processor built using 7nm technology is the fastest processor as on year 2020. It contains 64 compute cores with 256 MB of L3 Cache. Its base clock speed is 2.25 GHz which could be boosted up to 3.4 GHz. The processor has 8 DDR4 memory channels with per socket memory bandwidth of 204.8 GB/second (see Figure 4).

A Rome processor can perform 16 floating-point operations per clock cycle. Hence, the peak performance of each core can be estimated to be $16 \times 2.24 \approx 35$ Giga floating-point operations/second (*Flops/s* in short). Consequently, a Rome processor can perform $35 \times 64 \approx 2.24$ Tera Flop/s. These data are useful for time estimation of completing a computing job.

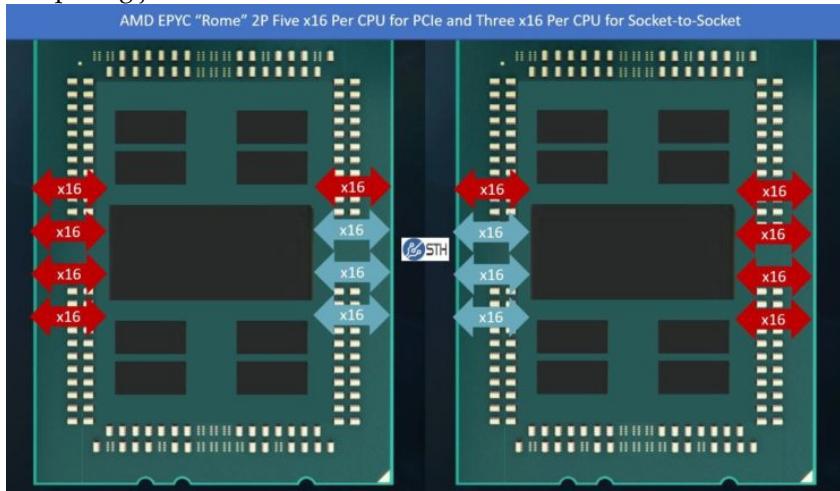


Figure 4: A schematic diagram of AMD's Rome 7742 processor. The compute cores reside inside the black patches. The blue and red bidirectional arrows represent channels.

There are many more server grade processors. For example, Intel's Xeon Gold 6252 built using 14 nm technology has 24 cores. Each core of Fugaku's A64FX performs around 56 Giga Flop/s, thus outperforms Rome 7742. We remark that the processors are getting better each day.

Graphical processing units (GPU) too are being used in supercomputers. Each GPU contains thousands of processors, and it can perform huge number of floating-point operations. GPUs are heavily used in machine learning.

Memory and Interconnect

One of the main bottlenecks in the performance of supercomputers is relatively slower data transfer rate from RAM to CPU. As described in the previous section, the data transfer rate from memory to CPU is around 1 GB per second. However, processors can operate on 1000 GB's of data per second.

Similar bottleneck exists for the internode communication, that is, between the processors across different nodes. One of the fastest interconnect, FDR Infiband, can transfer data at the rate of 56 Gbps. Considering that an interconnect is connected to many nodes, this speed is too slow compared to the rate at which CPU can process data. These bottlenecks are summarised in the following quote, "FLOPS are free, but data communications are expensive".

CPU is like a giant who can compute very fast, but remains idle because people (analogous to RAM) are not giving him enough data to work on. The performance disparity between the CPU and RAM/interconnect is one of the biggest challenges of supercomputing.

With this we close our introductory discussion on servers and supercomputers.

Conceptual questions

1. In what ways supercomputers are helping scientists and engineers in their research?
2. What is the peak computational performance of a supercomputer? Why don't we achieve performance close to the peak value?
3. Why is memory access a major bottleneck for supercomputers?

Exercises

1. A processor with 20 cores can perform 20 floating point per clock cycle. The clock speed of the processor is 3 GHz. Estimate the peak Flop/s of the processor.
2. Visit the top500.org and study the top 10 supercomputers of

the world. Frontera is the 8th ranked machine in the list and consists of Xeon Platinum 8280 28C 2.7 GHz processors. Compute the maximum Flop/s of this processor, and then compute the peak performance of the supercomputer.

1.3 *Memory and Time Complexity*

It is critical to estimate the memory and time requirements for a numerical computation. Significant efforts and resources would be wasted if we do not have proper estimate. For example, matrix multiplication of two $10^5 \times 10^5$ arrays requires 24 GB (see [Example 1](#) below), hence we should not try to run such programs on a laptop.

In the following discussion we present two examples that illustrate how to estimate memory and time complexity of a numerical computation. To estimate memory, we need to keep in mind that storage of an integers and a float variable normally require 4 and 8 bytes respectively.

Example 1: We want to multiply two arrays A and B of sizes $10^4 \times 10^4$ and store the result array C . For this problem we need 3 arrays of 10^8 elements each. Storage of 3×10^8 real numbers requires $8 \times 3 \times 10^8 = 24 \times 10^8$ bytes of storage, which is 0.24 GB.

A simplest algorithm for multiplication of two $N \times N$ arrays requires approximately N^3 multiplications and additions. Therefore, for $N = 10^4$, we need 10^{12} floating-point multiplications and additions. The peak performance of a typical laptop with CPU with 4 cores is 50 Giga Flop/s. Hence, in the best case scenario, the floating-point operations would require $2 \times 10^{12} / (50 \times 10^9) = 40$ seconds.

The retrieval and storage the array elements from/to memory require additional computer time. Due to this time, as well as those for web-browsing, email server, etc., we expect that program to take much larger than 40 seconds. However, we do not expect the run to go much beyond 10 minutes.

For $N = 10^5$, the space and time requirements would be respectively 24 GB and 10^4 minutes that go beyond the capabilities of a typical laptop.

Example 2: For weather prediction, the best resolution for the grid at the surface of the Earth is around 3km x 3km. Thus, we have around 12000×12000 horizontal grid points. Suppose, we take 1000 points along the vertical direction, then the total number of grid points for the simulation is 144×10^9 . At each grid point, we store the three components of the velocity field, pressure, temperature, humidity, and

CO_2 concentration. Hence, to store these seven variables at each grid point, we need $8 \times 7 \times 144 \times 10^9 = 8.064 \text{ TB}$ of memory. Clearly, we need a supercomputer for the operation.

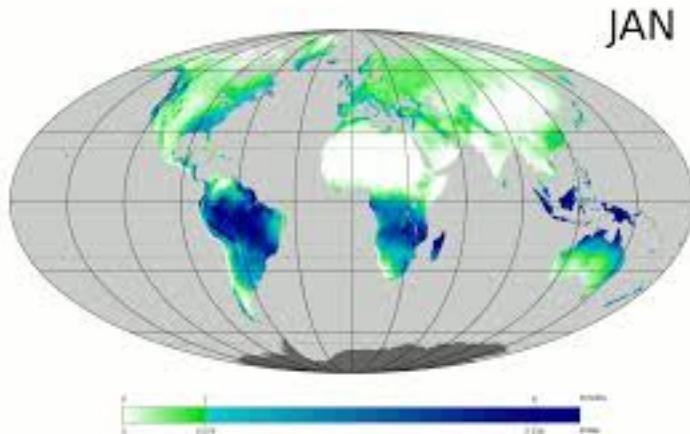


Figure 5: Grid for weather simulation on the surface of the Earth.
Courtesy: Wikipedia

The estimation of time requirement for weather codes is quite complex. We will attempt this estimate in Chapter XXX.

Exercises

1. Estimate the memory requirements for the following computational projects:
 - a. Three-dimensional integer array of size $1000 \times 1000 \times 1000$.
 - b. Three-dimensional float array of size $1000 \times 1000 \times 1000$.
 - c. 10^8 spins that can take values up or down.
2. We want to search for a word in text of 300 words. How many comparisons are required in the worst-case scenario?
3. Estimate the number of comparisons required to search for a word in a dictionary.
4. An atmospheric scientist wants to store the velocity field for

the wind blowing on Earth's atmosphere. Assuming grid resolution of 3km x 3km along the horizontal, and 1 meter along vertical for 10 km of height. What is the storage requirement?

5. You are required to perform matrix multiplication of two NxN matrices and store the result in another matrix. Estimate the largest matrix that you can multiply in your laptop.
6. Consider a matrix multiplication operation $A = B \times C$ where each of them are NxN matrices.
 - a. If you were to run the above on a laptop with 4 GB RAM and quadcore Xeon processor (20GF/per core), what is the max N allowed? How long will it take to perform this operation?
 - b. Do the above exercise for the HPC 2013 system of IIT Kanpur.

1.4 Applications of Computers

Most of the present-day science and engineering problems are very complex. It is impossible to solve many of them analytically. For example, it is very safe to say that no one can write an analytical solution for weather forecast. Computers have become ideal tools for addressing such complex problems.

In this section we provide an overview of various computational applications in science and engineering. I have classified them thematically.

Flows

Computers are used heavily for solving problems in these fields. Some of the leading examples are

- **Weather and climate forecast:** These are complex fluid and nonlinear problems that require enormous computing resources. Major challenges in this field are global warming, forecast for 100 years, formation of clouds and rains, etc. Every nation employs their best computers for these applications.
- **Flows around automobiles, aeroplanes, space vehicles, and rockets:** Such flows are quite complex. Scientists and engineers employ advanced numerical techniques and supercomputers to address these problems. Combustion is another problem with many unsolved issues for which computers are used heavily.
- **Oil exploration:** Due to economic issues, major supercomputing efforts are on oil exploration. In this field, researchers model the flows between the rocks deep inside the Earth.
- **Tectonics and earthquakes:** Researchers are trying to understand the physics of earthquakes, and possibly, predict them.

- **Physics of turbulence:** Due to the efforts over more than a century, some aspects of turbulence are understood. However, there are many unsolved issues in the field. For example, we do not fully understand the convection in the atmosphere, monsoons, global warming, mechanism of generation of magnetic field in the stars and planets. Computers have become handy for getting insights into these problems.
- **Flows in and around stars, blackholes, galaxies, and planets:** Some of the challenges in these fields are planet formation, accretion of matter to the central star, stellar winds, corona heating in the Sun, physics of magnetohydrodynamic turbulence. Many astrophysicists are working on these problems.
- **Quantum turbulence:** Turbulence in superfluids and Bose-Einstein condensates yield interesting insights into quantum world, namely quantum dissipation, multiscale interactions, etc.

Materials and Quantum Systems

- **Simulations of quantum matter:** Schrödinger's equations of Hydrogen atom and quantum oscillator have been solved analytically. However, no other atom (even Helium) or molecule exact solutions. This is due to complex many body interactions. Hence, computers are heavily used to solve such problems. Simulations of complex molecules have immediate practical applications in drug design and material science (e.g. strong material, noncorrosive steel, etc.).
- **Quantum Monte Carlo:** Simulations of collections of quantum particles are performed using quantum Monte Carlo methods. Other popular methods are density functional theory (DFT), particle simulation, etc.

Nonlinear Physics, Health

Most processes in the world are nonlinear. In fact, large number of problems discussed above involve complex nonlinearities, and hence

they can be also be classified in this field. Besides these, the nonlinear problems that are being addressed at present are

- **Epidemic evolution:** This field has become prominent due to COVID-19 pandemic. Epidemic growth occurs via nonlinear interactions among people, government, intervention mechanisms, etc.
- **Understanding brain:** We hardly understand human or animal brain. At present, researchers are attempting to simulate brain and understand its behaviour. Some groups are simulation billions of neurons and their interactions.
- **Network:** Interactions in social network, biological and ecological networks, computer networks are quite complex. Computers provide interesting avenue to learn their dynamics.
- **Human body:** Medical scientists are studying heart, brain, blood flow, ECG, etc. using computers.

Machine learning, Defence, Economics

- **Machine learning:** This field has become very important at present. Here, the emphasis is on understanding the patterns of data, rather than on the underlying physics. For example, a computer learns to contrast a cow with a horse by observing thousands of images of cows and horses.
- **Defence:** Computers are used heavily for war games, surveillance, cracking passwords, and making weapons.
- **Economics:** Large computers are employed in stock market for data management, predictions of stock market, etc.

Complex Physics

- **Dark matter/energy and evolution of the universe:** Researchers are modelling the matter and energy in the universe so as to match with the astronomical observations. Some of the simulations aim to simulate the whole universal.

- **Fusion and Tokomak simulation:** Nuclear fusion and inertial confinement (imploding matter using high-powered lasers) involve very complex physics. Computer simulations come out to be very handy for understanding the complexities. In these problems, we have to deal with charged particles, plasmas, electromagnetic fields, as well as large-scale flow dynamics.
- **Lattice quantum chromodynamics:** Though equations for nuclear matter (quarks and gluons) have been written already, but their solution remains illusive. Hence, researchers simulate the quantum fields of quarks and gluons and try to model properties of nuclei.
- **Accelerator simulations and data analysis:** The particle colliders generate huge amount of data that can be analysed only using computers. Large groups of scientists are involved in such efforts.

There are more applications, but we will stop here.

Exercises

1. Identify home appliances at your house and in your classrooms that rely on computers.

CHAPTER TWO
COMPUTER SOFTWARE AND PYTHON

2.1 Computer software

Computer software is a vast field. This section is not meant to discuss the nuances of various software; here we relate the programming languages to various programs running in a computer.

Operating system and system software

The *Operating System (OS)* makes a computer aware of its hardware—CPU, memory, hard disk—and connected input/output units—computer screen, keyboard, mouse, and printer. For example, a computer responds to the inputs from the keyboard; executes programs; etc. OS loads as soon as the computer is turned on; this process is called *boot up*. The OS performs the following tasks:

1. Memory management
2. Process management
3. Management of input/output devices (keyboard, display, printer, mouse)
4. Computer security
5. Management of application softwares (to be described below)
6. Interactions with users via input/output devices
7. Compilation and execution of user programs

The leading OS of today's computers are Unix and Windows. MacOS, the OS of Apple Computers, is a variant of Unix. Unix itself has many programs, which are categorised into two classes: Unix Kernel and Unix Shell. See [Figure 6](#) for an illustration. Note that OS of mobile devices—*iOS*, *Android*, and *Windows*—have limited capabilities.

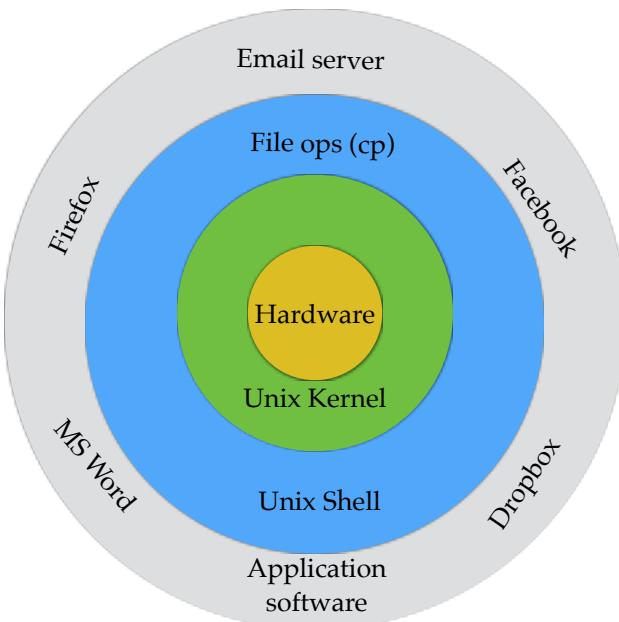


Figure 6: Schematic diagram of various kinds of software in a computer based on Unix OS.

Application softwares

Everyday we employ computers to perform many things: sending/receiving emails, browsing internet, watching movies, playing songs, writing documents, and occasionally writing programs. Each of these tasks are performed using application softwares. For example, we write documents using Microsoft Word or Pages (in macintosh computers); browse internet using Firefox, Internet Explorer, Safari, etc.; watch movies using Mplayer, VLC, etc.

In addition to the above, we can employ computers to record videos for surveillance, control robots and home devices such as air conditioners, TVs. Supercomputers are employed for more complex and data-intensive tasks such as banking, weather forecast, climate

modelling, simulations of airplanes and cars, etc. Note that the application softwares are at the top of Unix shell in system hierarchy. See [Figure 6](#).

System software and programming languages

At the base level, a computer understands instructions written in 0 and 1. Therefore, all the instructions and data are converted to strings of 0's and 1's. For example, an addition of two numbers A and B involves the following operations that are written in 0's and 1's:

1. Get numbers A and B from memory and put them into CPU registers.
2. Add the numbers and put them into register C.
3. Put the result C into the memory.

The addition operation involves bitwise operations, such as $1+0=1$, $1+1=0$ with a carry of 1, etc. The above instructions, coded in binary, are called *object code*.

The object code for the above operation itself is quite tedious. It is close to impossible to write object codes for large tasks, e. g., matrix multiplication. To overcome these difficulties, researchers at *Bell labs* devised a clever way in 1970's: a user writes programs in a higher-level language; then such programs are translated into object codes using system program called *Compilers* and *Interpreters*. This division of task saves the programmer from the drudgery of writing object codes.

The leading program languages are C, C++, Fortran, Java, Python, Matlab, etc. In C, C++, and Fortran, computer programs written completely, after which they are converted to object codes using compilers. Hence, C, C++, and Fortran are called *compiler languages*. On the other hand, Python and Matlab codes could be executed line by line by an interpreter, hence Python and Matlab are called *interpreter languages*. For example,

```
In [1]: x=3
In [2]: x
Out[2]: 3
In [3]: y=9
In [4]: print(x+y)
12
```

In the next section we will provide a overview of Python programming language.

Conceptual questions

1. What are the differences between computer hardware and software? Illustrate your answer using examples.
2. What are the differences between a compile and an interpreter?
3. List all the application and system softwares that you have used in your laptop/desktop.
4. Locate the *ipython* interpreter in your computer.

2.2 Brief Overview of Python

Python is a interpretive language that is very easy to learn and program. It is one of the most attractive and popular languages at present. Python is used for a large number of applications that includes the following:

1. Numerical and scientific computing
2. Data analysis
3. Big data and Machine learning
4. Image processing
5. Software development
6. Interface to experimental devices and control
7. GUI (graphical user interface) development
8. Audio and video applications
9. Internet programming

Cursory view of Python

At present, Python 3 is the standard version, which will be followed in this book. Two major differences between Python 3 and its earlier version, Python 2, are given below:

1. The arguments of print statement are within the brackets in Python 3, but it is not so in Python 2. In Python 2, the division operator “/” performs integer division for two integer operands; for real division, one of the operands of “/” must be float or complex.
2. In Python 3, the integer division and float division operators are different, and they are “//” and “/” respectively,

Using Python interpreter: We can start the Python interpreter from a unix terminal by typing *python* at the prompt. After that we can proceed to write Python statements. For example:

```
(base) ~/python
Python 3.7.4 (default, Aug 13 2019, 15:17:50)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc.
on darwin
Type "help", "copyright", "credits" or "license" for more
information.
```

```
>>> 2+3/4
2.75
>>>
```

In this book we recommend *Ipython* interactive shell with *pylab* option, which imports numpy and matplotlib modules. *Ipython* does not consume as much RAM as GUI package like *Spyder*. For invoking *ipython*, we type the following at the terminal prompt:

```
(base) ~/ipython --pylab
Python 3.7.4 (default, Aug 13 2019, 15:17:50)
Type 'copyright', 'credits' or 'license' for more
information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: MacOSX

In [1]: 2+3/4
Out[1]: 2.75
```

Python modules: A major advantage of Python is its large number of libraries—*numpy*, *matplotlib*, *scipy*, *math*, *pandas*, *sympy*, etc. Technically, Python libraries called *modules*. A brief on some of the important libraries are here.

- *numpy*: *numpy* is a short form for *Numerical Python*. This module contains mathematical functions, such as *sin*, *cos*, *log*, *sqrt*, etc. Also, *numpy* contains optimized functions for array operations.
- *math*: This module contains various mathematical functions. Many mathematical function are common between *math* and *numpy* (trigonometric, logarithm). But, some are not. For example, the function *factorial* exists in *math* module, but not in *numpy*.
- *matplotlib*: This Python module helps create beautiful plots.
- *scipy*: This module contains advanced scientific functions for integration and differentiation, Interpolators, differential equation solvers, Linear algebra operations, special functions, etc.
- *pandas*: This module is useful for data analysis and plotting.

- *turtle*: Using *turtle* module, we can create geometrical figures.
- *sympy*: This module is used for symbolic processing.

In this book we will deal with *numpy*, *math*, *matplotlib*, and *scipy* modules extensively. To use these libraries, we need to import them, which can be done in three ways:

import module: This way we make the module available in the current program as a separate *namespace*. Here, we need to refer to a function of the module using *module.function*. For example,

```
In [1]: import math
In [2]: math.factorial(5)
Out[2]: 120
```

Using *from* module *import* item: Here, the item of the module is referred to in the namespace of the current program. For example,

```
In [3]: from math import factorial
In [4]: factorial(10)
Out[4]: 3628800
```

Aliasing module using *import* module *as* alias: The module is renamed for convenience, or to make it special. For example,

```
In [6]: import math as ma
In [7]: ma.factorial(5)
Out[7]: 120
```

Python documentation and Help

Python is possibly the best documented language. There are a large number of websites and books on various topics of Python. You can just type what you are looking for in google, and you will find it.

Python offers useful online help in the interpreter itself. For example, we can get description of plot, sqrt, etc. by just typing these functions after ?. This is useful if you have forgotten the syntax. Two examples are here:

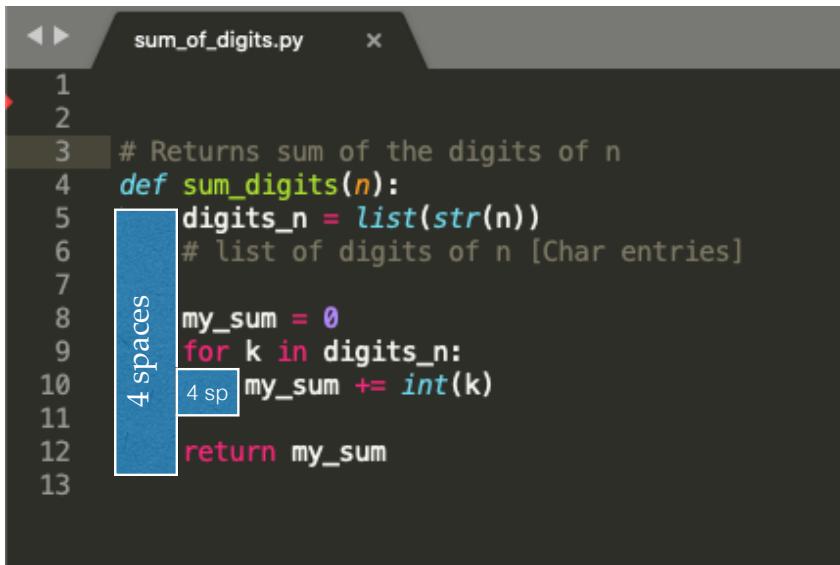
```
In [1]: ? plot
```

```
In [2]: ? sqrt
```

Creating a Python file using an editor

Small Python program can be written in *ipython* window itself. However, it becomes cumbersome to type large codes on the terminal. Large codes are written in a file using a text editor. You may choose any text editor among *Vim*, *IDLE*, *Atom*, *Spyder*, *Sublime*, *Xcode*, *Vi*, etc. However, I recommend *Sublime Text* (<https://www.sublimetext.com>), which is available for all the three platforms: Windows, Mac, and Linux.

Typing in *Sublime* is very easy. A screenshot of a Python code typed in Sublime is shown below (Figure 7):



```
1
2
3 # Returns sum of the digits of n
4 def sum_digits(n):
5     digits_n = list(str(n))
6     # list of digits of n [Char entries]
7
8     my_sum = 0
9     for k in digits_n:
10         my_sum += int(k)
11
12     return my_sum
13
```

Figure 7: A snapshot of a Python file *sum_of_digits.py* created using *sublime* editor.

Note is that a Python program needs to be properly *indented*. That is,

we need to place appropriate number of spaces in the beginning of each line. Note that lines 5, 6, 8, 9, and 12 have 4 spaces, while line 10 has 8 spaces. If we do not provide these spaces, the code will not run. Indentation is a must for Python codes!

We can save the file in the same folder in which *ipython* is running. This file can be executed in *ipython* window using a command ``*run filename*''. For example, a file named *sum_of_digits.py* that contains the function definition of *sum_digits()* is run in *ipython* as follows:

```
In [115]: run sum_of_digits.py  
In [116]: sum_digits(128)  
Out[116]: 11
```

The best way to learn programming is to actually program. Programming is like sports and art, which can be mastered by practice. I recommend that students should just start coding without fear and laziness.

In the next section we will contrast the differences between C and Python.

Exercises

1. Study how to import modules in Python. Import numpy as num and compute $\sin(\pi/2)$ using this module.

2.3 Python vs. C

Python has certain advantages and disadvantages over other programming languages. In this section, we contrast Python with another popular language, C. *This section is meant for advanced students, and it can be skipped by students not familiar with C.*

We list the differences between C and Python in [Table 1](#). A C program has to be written completely and then compiled, while Python codes can be developed piece by piece. Due to this reason and because of extensive Python modules, Python codes are easy to develop. A flipside of this feature is Python codes are not optimised, hence they are slower during the execution than the respective C codes. It is often said that a large Python code can be developed 10 times faster than the corresponding C code, but they are 10 time slower to execute.

[Table 1](#): Comparison between the features of Python and C.

	Python	C
1	An Interpreter language	A compiler language
2	No variable declaration	Variables need to be declared.
3	No need to declare variable type	Defining variable type is mandatory
4	Dynamically typed	Statically typed
5	Large set of libraries available	Limited set of libraries available
6	Easy syntax	Relatively harder syntax
7	Easy testing and debugging	Harder to test and debug.
8	Code development is fast	Code development is slow
9	Used heavily for postprocessing	Used for main solvers
9	Code execution is slow	Code execution is fast.
10	Interpreters do not generate as efficient codes as compilers.	Compilers optimise the object codes by taking into considerations the data structures and loop structures.

11	Parallel programming is not easy	Possible to write parallel program in C
12	Reference Type (see Sec. 4.4)	Value Type

Items 1-8 of [Table 1](#) make programming in Python very easy. Variable declaration is an important programming practice for large codes, but it is relaxed in Python. These features make execution of a Python code slower than C (see item 10). Note however that we can speed up a Python code by making use of fast libraries written in C or C++. For example, Python FFT library pyfftw is quite fast because it makes use of effect library FFTW, which is written in C.

Due to the above reasons, large parallel programs (e.g., for atmospheric and advanced physics applications) are not written in Python. It is best to write such codes in C, C++, or in Fortran with parallel features; compilers generate efficient object codes for such applications by observing the loop and data structures. However, often, prototypes of large codes are written in Python. Once the Python code is tested for complex but small data set, large codes in C or C++ are written. This exercise saves time on the whole; it is in the same spirit as prototyping of large aeroplane first and then work on the real-sized aeroplane.

With this we end our short section on comparison between C and Python. In the next section we will describe how to run Python in your own computer.

Conceptual questions

1. Why are C codes typically faster than Python codes?

2.4 Anaconda Python, Notebooks, and Prutor

In this section we describe some of the popular ways to run Python on a laptop or a desktop.

Anaconda Python

Anaconda Python is one of the most popular Python distributions. It is free for individual use. Anaconda Python (version 3.8) can download from Anaconda's website (<https://www.anaconda.com>) and installed on Windows, Linux, and Mac platforms. Anaconda takes significant disk space because all the packages are installed on the hard disk. Once this is done, you are ready to run Ipython, as well as various Python modules.

Anaconda Python also comes with a GUI (graphical user interface) called *spyder*. To start *spyder*, first launch Anaconda-Navigator, after which click the *spyder* package. A typical Spyder GUI has three windows—Editor, Help, and Ipython consoles. We can type the Python command in the ipython console of Spyder. See [Figure 8](#) for an illustration.

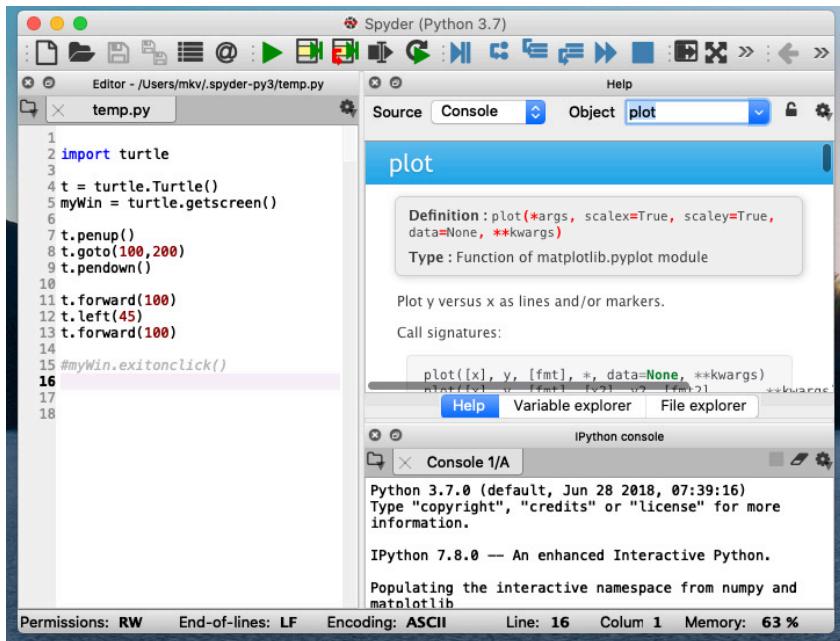


Figure 8: Spyder GUI with Editor, Help, and Ipython consoles.

Python notebooks

Several versions of Python notebooks are available at present. A major advantage of Python notebooks is that you do not need to install Python locally on your computer. We list two major Python notebooks below.

Google's Python notebook is called *Google Colab* that opens by typing <https://colab.research.google.com/notebooks/> on your browser. The browser pops up a window in which you can work with Python. **Figure 9** illustrates how a couple of lines of code can plot the function $y = x^2$. *Jupyter* too offers a Python notebook platform that has similar interface has collab.

my first python.ipynb

File Edit View Insert Runtime To

+ Code + Text RAM Disk Editing

2+3

5

```
[3]: from numpy import *
from matplotlib.pyplot import *
from math import *
```

```
[7]: x = linspace(0,1,100)
plot(x,x**2)
```

[<matplotlib.lines.Line2D at 0x7f56bab14240>]

The figure shows a plot of a parabola $y = x^2$ for $x \in [0, 1]$. The x-axis ranges from 0.0 to 1.0 with major ticks every 0.2. The y-axis ranges from 0.0 to 1.0 with major ticks every 0.2. The curve starts at (0,0) and ends at (1,1), passing through approximately (0.2, 0.04), (0.4, 0.16), (0.6, 0.36), (0.8, 0.64), and (1.0, 1.0).

```
[8]: sin(pi/2)
```

[> 1.0

Figure 9: Working with *collab notebook*

Prutor

Excercises

1. Install Anaconda Python on your computer.
2. Make a plot of $\sin(x)$ vs. x in *collab* and in *Jupyter* notebooks.

CHAPTER THREE

PYTHON DATA TYPES: INTEGERS AND FLOATS

3 . 1 *Integers*

Basic Python data structures are of the following types:

- Integer
- Floating point numbers
- String
- List
- Arrays

In this section we will discuss Integer data type.

Representation of integers

Recall how the numbers are stored in *decimal system* or *base-10* system. The decimal number 953 is

$$953 = 9 \times 10^2 + 5 \times 10^1 + 3 \times 10^0 = 9 \times 100 + 5 \times 10 + 3$$

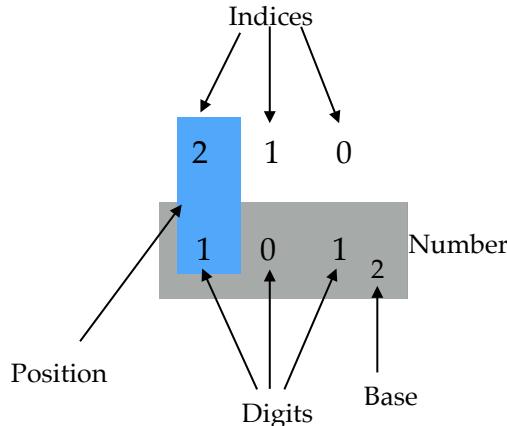
In 953, the digits 9, 5, and 3 are have different weights: 9 multiplied by 100, 5 by 10, and 3 by 1.

Classical computers employ *binary system* (*base-2*), whose digits are 0 and 1. These digits called *bits*. Using these bits, a general binary number is represented as follows:

$$(b_{N-1} b_{N-2} \dots b_2 b_1 b_0)_2 = b_{N-1} \times 2^{N-1} + b_{N-2} \times 2^{N-2} \dots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0, \dots \quad (1)$$

where b_i is the bit at the i^{th} index. A particular example, illustrated in [Figure 10](#), is

$$(101)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (5)_{10}.$$



$$(101)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (5)_{10}$$

Figure 10: An example of binary number number: $(101)_2$

Some other examples binary numbers are

$$(11)_2 = 1 \times 2^1 + 1 \times 2^0 = (3)_{10},$$

$$(1101)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (13)_{10}.$$

We need both positive and negative integers. The first digit is reserved as a sign bit: 0 for positive integers, and 1 for negative integers. Representation of negative numbers in computers will be discussed at the end of the section.

We require many bits to represent large integers. Hexadecimal (hex, in short) system becomes handy in such occasion. In hex system or base 16 system, the digits are 0, 1, 2, ..., 9, A, B, C, D, E, F. Here,

$$\begin{aligned} (h_{N-1} h_{N-2} \dots h_2 h_1 h_0)_2 &= h_{N-1} \times 16^{N-1} + h_{N-2} \times 16^{N-2} + \dots \\ &\quad + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0. \quad \dots(2) \end{aligned}$$

The decimal number $(251)_{10}$ in hexadecimal is $(FB)_{16}$:

$$(1111\ 1011)_2 = (FB)_{16} = 15 \times 16 + 11 = (251)_{10}$$

Octal number system, which has base 8 and digits 0, 1, 2, 3, 4, 5, 6, and

7, are defined similarly. An example of a number in octal system is

$$(11\ 111\ 011)_2 = (373)_8 = 3 \times 8^2 + 7 \times 8 + 3 = (251)_{10}.$$

Thus we show how a number can be represented in different number systems. Note that a number is the same, but its representations are different.

It is intriguing why humans chose decimal system over others. It is possibly because we have 10 fingers to count. Also, the number of digits moderately large numbers (e.g., $(1000)_{10}$) in decimal representation is not too many as in binary representation. Hence, decimal representation appears to be quite convenient.

Conversions between different number systems

We employ decimal number system in our daily lives, but computers employ binary number system. Hence, we need to convert one representation to another. Conversions from binary to decimal, and from hex to decimal are straightforward. We can employ Eq. (1) and Eq. (2) for these conversions.

The converse, conversion from decimal to binary, is performed as follows. Suppose we want to convert $(19)_{10}$ to binary. Since $16 < 19 < 32$, we write $19 = 16 + 3$. Now $3 = 2 + 1$. Therefore, $(19)_{10} = 16 + 2 + 1 = (10011)_2$. Following similar process, we convert a decimal number to hex system. For example, $(251)_{10} = 16 \times 15 + 11$. Therefore, $(251)_{10} = (\text{FB})_{16}$.

Python provides functions for binary, octal, and hex equivalents of a decimal number. For example, the following functions provide binary, octal, and hex equivalents to decimal number $(100)_{10}$.

```
In [18]: x= 100
In [21]: bin(x)
Out[21]: '0b1100100'
In [22]: oct(x)
Out[22]: '0o144'
In [23]: hex(x)
Out[23]: '0x64'
```

In the above code segment, 0b, 0o, and 0x represent respectively the binary, octal, and hex representations. Using inverse functions, we can

obtain decimal representations of numbers in base 2, 10, or hex. For example,

```
In [86]: int('1110',2) # in base 2
Out[86]: 14

In [89]: int('1110',10) # in base 10
Out[89]: 1110

In [91]: int('111F',16) # in base 16
Out[91]: 4383
```

Integers in computers

Computers typically employ 4 bytes or 32 bits to store an integer. Among the 32 bits, the first bit is reserved for *sign* representation: 0 for positive integers and 1 for negative integers. Hence, the range of integers that can be represented in a computer is -2^{31} to $2^{31}-1$, or -2147483648 to 2147483647. Considering that $2^{10} = 1024 \approx 10^3$, we estimate 2^{31} as $2 \times (10^3)^3$ or 2×10^9 .

In python, an integer can of any length. We can demonstrate this statement by the following Python code.

```
In [4]: x=12345678901234567890123456789
In [5]: x
Out[5]: 12345678901234567890123456789

In [7]: x*10
Out[7]: 123456789012345678901234567890
```

In the above example, the integer x contains 29 digits, which is beyond what can be represented in C programming language.

The addition, subtraction, multiplication, and integer-division operators in Python are $+$, $-$, $*$, and $//$ respectively. The expression a^b is evaluated in Python using power operator $**$ as a^{**b} . Note that the operator $/$ is used for real division (to be discussed in the next chapter). For example, $95//45 = 2$, but, $95/45 = 2.11111111111111$.

Often we have expressions involving several integers and operators. Under such situations, we follow the rules given in [Table 2](#).

Table 2: Precedence of arithmetic operators involving integers. Note that $*$, $//$, $\%$ have same precedence.

Operator	Function
<code>**</code> (Highest precedence)	Power
<code>*</code>	Multiplication
<code>//</code>	Integer division
<code>%</code>	Modulus
<code>+, -</code> (Lowest precedence)	Add, subtract

Note that

1. The operators `*`, `//`, and `%` have same precedence, so do `+` and `-`.
2. Operators with highest precedence are evaluated first.
3. In case of multiple operators with same precedence, they are evaluated from left to right.
4. Brackets are used to override precedence.

We illustrate some of the above operations using the following examples.

```
In [45]: 2**31-1
Out[45]: 2147483647
```

```
In [46]: 5//3
Out[46]: 1
```

```
In [47]: 5/3
Out[47]: 1.6666666666666667
```

```
In [48]: 3+16//9//3
Out[48]: 3
```

Representation of negative integers in computers

In Python, the negative numbers are stored as *two's complement*. Here, a negative number b is stored as $2^N - b$, where N is the maximum number of bits.

We illustrate operations involving *two's complement* numbers using an example. For simplicity we consider a 4-bit computer that can store integers in the range -3 to 4 . Note that the first bit is reserved to represent the sign of the number. The positive integers 0 to 4 are represented as `0000`, `0001`, `0010`, `0011`, `0100`. To store -3 , two's complement of 3 is computed as follows:

1. Compute bit-flip operation on 3 : $\sim 3 = \sim 0011 = 1100$, which is called *one's complement* of $+3$.
2. Add one to the above: $\sim 3 + 1 = 1101$, which is called *two's*

complement of +3. -3 is stored as 1101.

Using the same method, we deduce that -1, -2 are stored as 1111 and 1110 respectively. Using two's compliment we can easily perform the subtraction operation, which is

$$a - b = a + (2^N - b) - 2^N.$$

That is, we add a and two's complement of b , which is $(2^N - b)$. The subtraction of 2^N is trivially achieved by bit overflow. For example,

$$4 - 3 = 0100 + 1101 = 0001$$

which is the correct answer. Note the bit overflow of 1 at the end; this bit has been discarded. In [Table 3](#) we present 1's and 2's compliments of binary numbers from 0000 to 0111.

The above idea also applies to other number systems. For decimal system, the respective representation is 10's complement. For example, in a three-digit decimal system, -14 would be stored as $1000 - 14 = 986$. Here too, subtraction can be easily performed using 10's complement.

[Table 3:](#) 1's and 2's compliments of binary numbers from 0000 to 0111.

number	1's compliment	2's compliment
0000 (0)	1111 (-7)	No representation (-8)
0001 (1)	1110 (-6)	1111 (-7)
0010 (2)	1101 (-5)	1110 (-6)
0011 (3)	1100 (-4)	1101 (-5)
0100 (4)	1011 (-3)	1100 (-4)
0101 (5)	1010 (-2)	1011 (-3)
0110 (6)	1001 (-1)	1010 (-2)
0111 (7)	1000 (0)	1001 (-1)

Here we end our discussion on integers in Python.

Conceptual questions

1. Why binary system is employed for computer memory? Why is the decimal system more suited than the binary system for daily lives?

Exercises

1. What are the largest and smallest signed integers that can be stored in 16-bit, 32-bit, 64-bit, and 128-bit machines?
2. Convert the following binary numbers to decimal numbers. Verify your results using Python functions.
(a) 11001 (b) 11111111 (c) 1001001
3. Convert the following decimal numbers to binary numbers. Verify your results using Python functions.
(a) 100 (b) 129 (c) 8192
4. Convert the following decimal numbers to hexadecimal numbers. Verify your results using Python functions.
(a) 100 (b) 129 (c) 8192
5. Convert the following hexadecimal numbers to decimal numbers. Verify your results using Python functions.
(a) 1F (b) DD (c) F54 (d) 555
6. Assume 1-byte storage for positive and negative integers. Compute 1's and 2's compliments for the following binary numbers:
(a) 00101010 (b) 01111111 (c) 00001111
7. Perform the following subtraction operation using 2's compliment and verify your result: 01111111 – 00001111
8. Evaluate the following integer expressions in Python and state your results:
 $3^{**}3, 9 // 2, 9 / 2, 4 - 6 - 3, 4 - (6 - 3), 10 + 90 // 7, 4 // 3 * 7, 4 * 7 // 3,$
 $9 * 2^{**}3$
9. Two servers have 16GB and 96GB RAM. What are the RAM sizes in decimal representation? How many integers can be stored in this RAM?

Practical Computational Methods using Python

3.2 Floating Point and Complex Numbers

To represent real numbers, humans employ decimal number system, but computers uses binary number system. We use several ways to represent real numbers. For example, half is represented as $\frac{1}{2}$ or 0.5, which are fraction notation and decimal-point notation respectively. In computers, the real numbers are represented using binary-point notation. An important point to remember is that in Python, real numbers are also called *float*, which is a short form for *floating point number*.

A real number can be rational or irrational. Irrational numbers (for example, $\sqrt{2}$, π , and e) require infinite number of digits in decimal-point notation. Some rational numbers too require infinite number of digits for their representation. For example, in decimal-point notation, $\frac{1}{3}$ requires infinite number of decimal digits (0.3333...).

First we describe the decimal system of real numbers.

Fixed point numbers

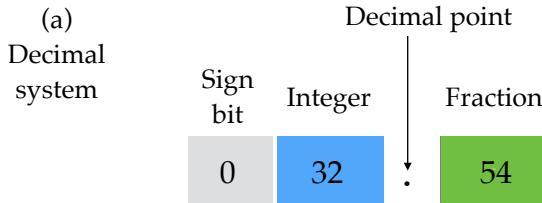
We humans represent real numbers as

$$(d_n d_{n-1} \dots d_0 . d_{-1} \dots d_{-m})_{10} = (d_n 10^n + d_{n-1} 10^{n-1} + \dots + d_0 + d_{-1} 10^{-1} + \dots + d_{-m} 10^{-m}).$$

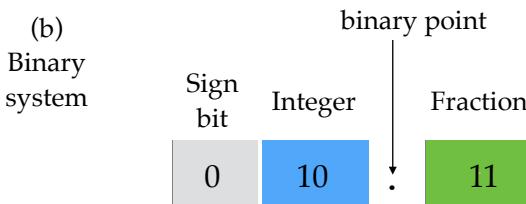
In the above decimal number, the number of digits is $n+m+1$. The number to the left of the decimal point is called *integer*, while to the right of decimal point is called *fraction*. The point separating the integer and fraction is the *decimal point*. Clearly, the weights of the digits decrease as we go rightwards. Note that we also need a sign bit, e.g., 0 for positive and 1 for negative. A specific example is

$$(32.54)_{10} = 3 \times 10^1 + 2 \times 10^0 + 5 \times 10^{-1} + 4 \times 10^{-2},$$

which is illustrated in [Figure 11\(a\)](#). With 1 sign bit and 4 digits (2 for integer and 2 for fraction), the largest and smallest positive (nonzero) numbers in this notation are 99.99 and 00.01 respectively. We can represent 10^4 real numbers in this scheme. The difference between two consecutive real numbers is 0.01, a constant value.



$$(32.54)_{10} = [3 \times 10 + 2 \times 10^0] + [5 \times 10^{-1} + 4 \times 10^{-2}]$$



$$(10.11)_2 = [1 \times 2^1 + 0 \times 2^0] + [1 \times 2^{-1} + 1 \times 2^{-2}]$$

Figure 11: (a) An example of a decimal number, $(32.54)_2$. (b) An example of binary number, $(10.11)_2$

A real number in binary system is

$$(b_n b_{n-1} \dots b_0 . b_{-1} \dots b_{-m})_2 = (b_n 2^n + b_{n-1} 2^{n-1} + \dots + b_0 + b_{-1} 2^{-1} + \dots + b_{-m} 2^{-m}). \dots (3)$$

Here, similar to the decimal system, the middle point, called *binary point*, separates the integer and fraction. For example, as shown in [Figure 11\(b\)](#),

$$(10.11)_2 = 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 2 + 0 + 0.5 + 0.25 = (2.75)_{10}.$$

Similarly,

$$\begin{aligned}
 (11.1011)_2 &= 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} \\
 &= 2 + 1 + 0.5 + 0.125 + 0.0625 = (3.6875)_{10}
 \end{aligned}$$

A real number could also be represented in hex and octal systems by replacing the base in Eq. (3) by 16 and 8 respectively.

In binary system with 1 bit for sign, 5 bits for integer, and 5 bits for fraction, the largest and smallest positive (nonzero) numbers in this notation are $(11111.11111)_2$ and $(0.00001)_2$ respectively; they translate to respectively $(31.96875)_{10}$ and $(0.03125)_{10}$ in decimal system. Thus, the range of numbers that can be represented using fixed point scheme is limited, which is a big obstacle for representing a large range of numbers observed in nature. To achieve this task we use floating point scheme.

Floating point numbers

In this format, a real number is written as

$$\text{real number} = \text{fraction} \times b^{\text{exponent}}.$$

We illustrate this notation in [Figure 12](#). For decimal number, we need sign bits for both fraction (called *mantissa*) and exponent. Besides, we impose a condition that the first digit after the decimal must be nonzero. Hence 540 is allowed as mantissa, but not 054. The largest and smallest positive numbers that could be represented using 2 digits each for mantissa and exponent are 0.99×10^{99} and 0.10×10^{-99} respectively.

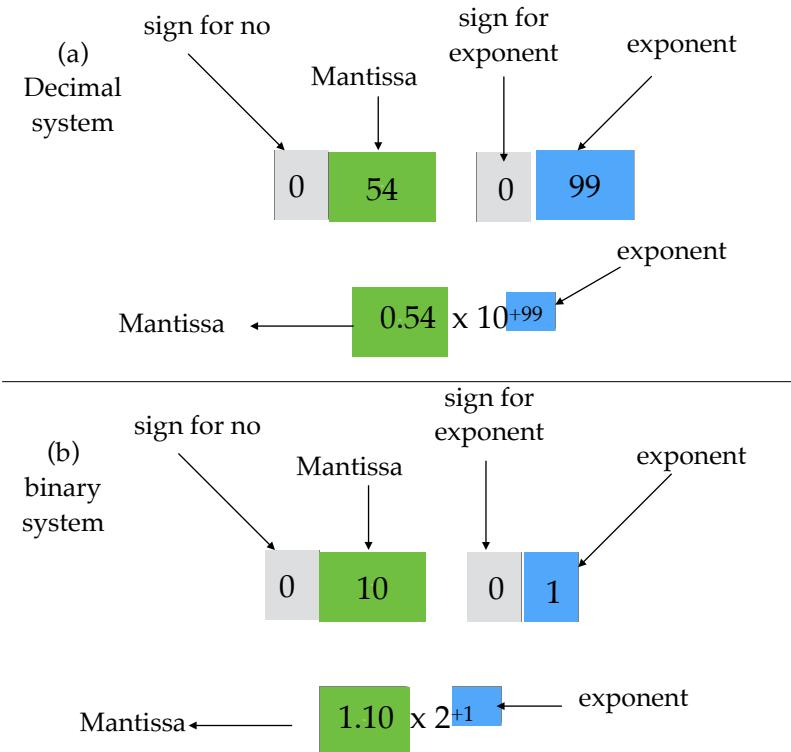


Figure 12: Illustrations of floating point number representations in decimal and binary systems

Thus, the range of numbers in floating point representation is much larger than in fixed point representation. Also note a peculiar property of this number system. For a given exponent E , the gap between two consecutive real numbers is 0.1×10^E , but the gap jumps by a factor of 10 when the exponent is incremented by 1. Interestingly, this feature is observed in nature. In animal kingdom, first comes viruses whose size are multiples of nanometers, after which comes bacterias, which are of the size of micrometers. Then come insects (of size cms) and mammals (of size meters). The size of the animals within a class increases gradually, but there are jumps in the size when we go from one class of animals to another. Physical systems too exhibit similar multiscale structures: nuclei (at femtometer), atoms and

molecules (nanometers), polymers (microns), ..., daily objects (meters), oceans and planets (1000 kms), stars (10^6 km), galaxies (light years), and universe (10^9 light years). The variations in the length scales in these multiscale fractal-like systems have similarities with those in floating point numbers.

The floating point representation of binary number system has similar variations. A constraint is imposed in the binary system: the integer part of the mantissa is always 1 (see [Figure 12](#)). We convert all the numbers to this form. For example, $(1111)_2$ is written as 1.111×2^3 , and $(0.011)_2$ as 1.1×2^{-2} . With 5 bits each for mantissa and exponents (apart from sign bits), the largest and smallest numbers representable in the binary system are 1.1111×2^{31} and 1.00000×2^{-31} respectively; in decimal system, they translated to $(1.96875) \times 2^{31}$ and 1.0×2^{-31} respectively. Similar procedure can be used to represent numbers in hex and octal systems.

In computers, the real numbers are represented using floating point representation, as we describe later in this section.

Conversions between binary and decimal number systems

The integer part of a real number (e.g., 12 of 12.67) is converted to binary as described in the previous chapter. We illustrate the conversions of the fractions $(0.375)_{10}$ and $(0.80)_{10}$ to binary as follows:

	frac	integer			frac	integer	
$0.375 \times 2 = 0.75$	0.75	0	b_{-1}	$0.8 \times 2 = 1.6$	0.6	1	b_{-1}
$0.75 \times 2 = 1.5$	0.5	1	b_{-2}	$0.6 \times 2 = 1.2$	0.2	1	b_{-2}
$0.5 \times 2 = 1$	0	1	b_{-3}	$0.2 \times 2 = 0.4$	0.4	0	b_{-3}
$(0.375)_{10} = (0.011)_2$				$0.4 \times 2 = 0.8$	0.8	0	b_{-4}
				$(0.8)_{10} = (0.11001100\dots)_2$			
(a)				(b)			

Figure 13: Conversion of decimal numbers to binary numbers: (a) 0.375, (b) 0.8.

In Figure 13(a), we convert $(0.375)_{10}$ to binary representation. For the same, we multiply the number by 2, and collect the integer part (0) as the first binary digit after the binary point. Now, the news fraction, 0.75, is multiplied by 2, and the resulting integer part, 1, is collected as the second binary digit. We continue this process until fraction becomes 0. This process yields $(0.375)_{10} = (0.011)_2$. Thus, 0.375 requires finite number of digits in both decimal and binary representations.

This conversion process does not converge for some decimal numbers. As shown in Figure 13(b), $0.8 \rightarrow 1.6 \rightarrow 0.6 \rightarrow 1.2 \rightarrow 0.2 \rightarrow 0.4 \rightarrow 0.8$. Therefore, $(0.8)_{10} = (0.1100\ 1100\ 1100\dots)_2$ with 1100 recurring. Note that $(0.8)_{10}$ has finite number of digits in decimal system, but infinite number of bits in binary system.

Following similar arguments, we deduce that $(\frac{1}{3})_{10} = (0.01\ 01\ 01\dots)_2$ with 01 recurring. Note that $(\frac{1}{3})_{10} = (0.3333\dots)_{10}$. Thus, both decimal and binary representations of $(\frac{1}{3})_{10}$ requires infinite number of digits. However, $(\frac{1}{3})_{10}$ requires only one digit in tertiary (base-3) system: $(\frac{1}{3})_{10} = (0.1)_3$.

More examples:

$$(1/3)_{10} = (0.01\ 01\ 01\ \dots)_2 \text{ with recurrence of } 01$$

$$(1.1)_{10} = (1.0001\ 1001\ 1001\ 1001\ \dots)_2 \text{ with recurrence of } 1001$$

$$(1.5)_{10} = 1+1/2 = (1.1)_2$$

$$(15.0)_{10} = 8+4+2+1 = (1111)_2 = 1.111 \times 2^3$$

$$(150.0)_{10} = 9 \times 16 + 6 = (1001\ 0110)_2 = 1.001\ 0110 \times 2^7$$

$$(1.75)_{10} = 1+1 \times 2^{-1} + 2^{-2} = (1.11)_2$$

In computers, with finite number of bytes allocated for number representation, even in principle, some numbers (e.g., $\frac{1}{3}$ and $\sqrt{2}$) cannot be represented accurately in a computer. We discuss this issue at the end of the section.

We convert a decimal number to hexadecimal number by following the same procedure as above (see [Figure 13](#)). For example, we can convert $(0.375)_{10}$ and $(0.8)_{10}$ to hexadecimal number by multiplying the fraction with 16. We multiply 0.375 with 16 that yields 6, which is h_{-1} . We stop at this stage because the fraction is zero. Otherwise, we need to iterate the above process for the next fraction... and so on till the fraction becomes 0. Hence, $(0.375)_{10} = (0.6)_x$. Note, however, that the iteration does not terminate for some numbers. For example, for 0.8, the next fraction remains as 0.8. You can easily verify that $(0.8)_{10} = (0.CC\dots)_x$.

	frac	integer		frac	integer	
$0.375 \times 2 = 0.75$	0.75	0 b_{-1}	$0.8 \times 2 = 1.6$	0.6	1	b_{-1}
$0.75 \times 2 = 1.5$	0.5	1 b_{-2}	$0.6 \times 2 = 1.2$	0.2	1	b_{-2}
$0.5 \times 2 = 1$	0	1 b_{-3}	$0.2 \times 2 = 0.4$	0.4	0	b_{-3}
$(0.375)_{10} = (0.011)_2$			$0.4 \times 2 = 0.8$	0.8	0	b_{-4}
						$(0.8)_{10} = (0.11001100\dots)_2$
(a)			(b)			

Figure 14: Conversion of a decimal numbers to hexadecimal numbers:
(a) 0.375, (b) 0.8.

Following similar lines, we can convert the other examples given for binary system to hexadecimal system.

$$\begin{aligned}
 (1/3)_{10} &= (0.55 \dots)_x \text{ with recurrence of 5} \\
 (1.1)_{10} &= (1.1999\dots)_x \text{ with recurrence of 9} \\
 (1.5)_{10} &= (1.8)_x \\
 (1.75)_{10} &= (1.C)_x
 \end{aligned}$$

Real numbers in Python

Computers operate real numbers in floating point representation. Typically 64 bits are employed to store a real number. Among the 64 bits, 52 bits are reserved for the mantissa, 11 bits for the exponent, and 1 bit for the sign. The integer part of mantissa is always taken to be 1. There is no sign bit for the exponent, but the exponent is chosen as $E-1024$, where E is the integer value of the 11-bit exponent. Since E lies in the range of 0 to 2047, the exponent varies from -1024 to 1023. In this convention, the real number representable by 64 bits of [Figure 15](#) is

$$\text{Number} = (-1)^{\text{sign}} (1.b_{-1}b_{-2}\dots b_{-52}) \times 2^{E-1023}.$$

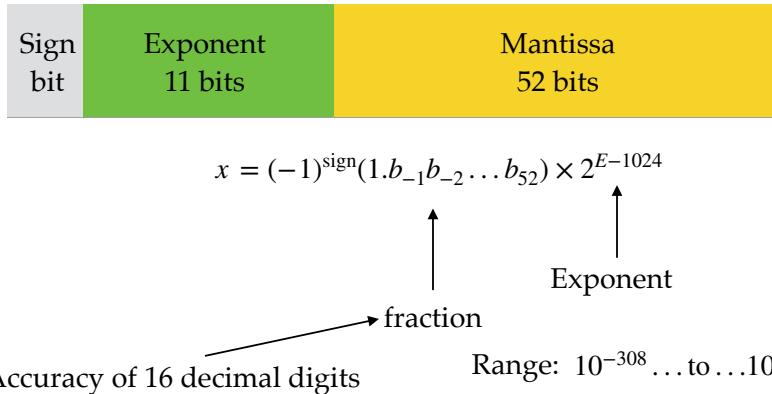


Figure 15: Computer representation of a real number using 64 bits.

Special numbers have special representations in computers are

1. *Actual zero:* $E = 0$, Mantissa = 0.
2. *Infinity:* $E = 2047$, Mantissa = 0; $+\infty$ when sign = 1, but $-\infty$ when sign = 0.
3. *Not a number (NaN):* $E = 2047$ and nonzero mantissa.

Python offers function `float.hex` to convert decimal to hex and vice versa. For example,

```
In [95]: float.hex(1.5)
Out[95]: '0x1.80000000000000p+0'
```

In the above, x stands for hex. The mantissa for $(1.5)_{10}$ is 100.. (52 of them), which corresponds to 13 hex digits, 8000000000000. The exponent is 0, as is evident from p+0.

```
In [1]: float.hex(15.0)
Out[1]: '0x1.e0000000000000p+3'
```

```
In [2]: float.hex(150.0)
Out[2]: '0x1.2c000000000000p+7'
```

```
In [3]: float.hex(-150.0)
Out[3]: '-0x1.2c00000000000000p+7'

In [5]: float.hex(0.8)
Out[5]: '0x1.999999999999ap-1'

In [6]: float.hex(0.375)
Out[6]: '0x1.8000000000000000p-2'

In [7]: float.hex(1/3)
Out[7]: '0x1.55555555555555p-2'

In [8]: (1.1).hex()
Out[8]: '0x1.199999999999ap+0'
```

Note that $+150.0$ and -150.0 differ only in the sign bit. The above results are consistent with the fact that $(15.0)_{10} = 1.111 \times 2^3$ (mantissa 1.E and exponent 3) and $(150.0)_{10} = 1.0010 1100 \times 2^7$ (mantissa 1.2C and exponent 7).

In Python, infinity and NaN are represented as

```
In [165]: math.inf
Out[165]: inf

In [170]: math.nan
Out[170]: nan

In [153]: myzero = '0x0.00000000000000p-1023'

In [154]: float.fromhex(myzero)
Out[154]: 0.0
```

Since $E = 2047$ is reserved for the infinity, the largest positive floating-point number has $E = 2046$ or exponent = 1023. Therefore,

$$\begin{aligned} f_{\max} &= (1.11\dots1)_2 \times 2^{1023} = (1.FFFFFFFFFFFFFF)_{16} \times 2^{1023} \\ &\approx (1.7976931348623157)_{10} \times 10^{308} \end{aligned}$$

```
In [106]: float.fromhex('0x1.FFFFFFFFFFFFFFp+1023' )
Out[106]: 1.7976931348623157e+308

In [104]: sys.float_info.max
Out[104]: 1.7976931348623157e+308

In [105]: (sys.float_info.max).hex()
Out[105]: '0x1.fffffffffffffp+1023'

In [106]: float.fromhex('0x1.FFFFFFFFFFFFFFp+1023' )
Out[106]: 1.7976931348623157e+308
```

Note that $1023 * \log_{10}(2) \approx 308$. Similarly, the smallest positive float

represented in a computer is computed as follows. Since $E = 0$ is reserved for zero, we take $E = 1$ for the smaller float number. Also, mantissa = 0 for this case. Therefore,

$$f_{\min} = (1.000 \dots 000)_2 \times 2^{-1022} \approx (2.2250738585072014)_{10} \times 10^{-308}$$

The corresponding Python statements are

```
In [19]: float.fromhex('0x1.0p-1022')
Out[19]: 2.2250738585072014e-308
```

```
In [106]: sys.float_info.min
Out[106]: 2.2250738585072014e-308
```

```
In [107]: sys.float_info.min.hex()
Out[107]: '0x1.000000000000000p-1022'
```

Also note that the numbers beyond the allowable range are either zero or infinite, depending on whether the number is lower than the minimum number or larger than the maximum number.

```
In [27]: print(1e308)
1e+308
```

```
In [28]: print(1e309)
inf
```

```
In [29]: print(3*1e308)
Inf
```

```
In [33]: print(1e-500)
0.0
```

Some processors employ 32-bit or 16-bit representations for real numbers. Such processors are called 32-bit and 16-bit processors. The floats in these machines are smaller than those of 64-bit processors. Examples of such processors are GPUs (graphical processor units) and those inside cars, air conditioners, TV sets, etc. Applications in these devices do not require good precision.

In a *32-bit* or *single-precision floating-point format*, the exponents and mantissa are represented using 8 and 23 bits respectively. In *16-bit* or *half-precision floating-point format*, the corresponding bits are 5 and 10 respectively. Following the similar lines of arguments as those for 64-

bit machine, we deduce that in a 32-bit machine,

$$-128 * \log_{10}(2) < E < 128 * \log_{10}(2).$$

That is, E lies in the range of -38 to $+38$. Also, the precision $= 2^{-23} \approx 1.19 \times 10^{-7}$ that corresponds to 7 significant digits. Similarly we can deduce that the precision of a 16-bit machine is $2^{-10} \approx 9.7 \times 10^{-4}$, while its exponent E lies in the range of -4.8 to $+4.8$ ($16 * \log_{10}(2)$).

The floating point operators of Python are listed in [Table 4](#). The floating point operations are straightforward hence they are not presented here. Keep in mind the difference between the integer division ($//$) and real division ($/$). As in integers,

1. The operators `*` and `/` have same precedence, so do `+` and `-`.
2. Operators with highest precedence are evaluated first.
3. In case of multiple operators with same precedence, they are evaluated from left to right.
4. Brackets are used to override precedence.

Table 4: Precedence of arithmetic operators involving real numbers

Operator	Function
<code>**</code> (Highest precedence)	Power
<code>*, /</code>	Multiplication, division
<code>+, -</code> (Lowest precedence)	Add, subtract

There are many float functions that are part of numpy module. Some of the functions are `sqrt()`, `log()`, `log10()`, `sin()`, `cos()`, `int()` and `round()`. The function `int()` truncates the real number and returns a integer value, while `round()` rounds off the real number.

```
In [113]: int(13.5)
Out[113]: 13

In [115]: round(13.6)
Out[115]: 14

In [21]: round(13.5)
Out[21]: 14

In [21]: round(13.1)
Out[21]: 13
```

Precision, Round-off or truncation errors

We estimate the precision of a 64-bit computer as follows. The nearest floats differ in bit b_{-52} while keeping E fixed. Consecutively, the difference between consecutive numbers with the same exponent would be $2^{-52} \times 2^{E-1024}$. Therefore, the precision of a 64-bit real computer is

$$P = 2^{-52} \approx 2.22 \times 10^{-16}$$

that corresponds to 16 decimal digits. Therefore, the maximum significant digits for Python float is 16.

The examples in this section illustrate that in general, many real numbers can not be represented precisely due to (a) finite number of bits used; (b) truncation of digits in the expansion of rational numbers (as in $\frac{1}{3}$); (c) conversion from decimal to binary representation (as in $(0.8)_{10}$). Such an error is called *round-off error* or *truncation error*.

On many occasions, exact arithmetic operations and the corresponding computations in a computer do not match due to the truncation error. We illustrate them using an example.

```
In [23]: 0.8*3
Out[23]: 2.4000000000000004

In [24]: 0.8*3-2.4
Out[24]: 4.440892098500626e-16

In [25]: 1.5*3
Out[25]: 4.5

In [26]: 1.5*3-4.5
Out[26]: 0.0
```

Computer-generated $3*0.8$ does not match with exact value of 2.4. The difference $0.8*3-2.4$ is nonzero, and it is around 10^{-16} , which is the machine precision. The above error is because 0.8 is not representable exactly in binary system ($(0.8)_{10} = (0.CC \dots)_x$). The corresponding error for 1.5 is zero because 1.5 is represented exactly in the computer.

Special tricks are employed to achieve better computing precision than 16 digits. For example, π has been computed with precision of billions of digits by employing more terms in the series. In Section 9.1 we will discuss errors and precision in more detail.

We relate the precision of a computer to the finite number of states available to store a number. We have 2^{64} states with 64 bits. Therefore, we can represent 2^{64} distinct numbers, irrespective of schemes (fixed point, float point, or whatever). The rest of the numbers can be stored only approximately. This is the origin of error in arithmetics involving real numbers. Note that such issues do not arise for integers because they are represented accurately, as long as they are within the limit, e.g., -2^{31} to $2^{31}-1$ for a 64-bit machine.

Complex float

A complex number in Python is a combination of two real numbers, one for real part and the other for the imaginary part. The syntax for a complex number assignment is as follows:

```
In [12]: z = 1.0+2.0j
```

where 1.0 and 2.0 are the real and imaginary parts of z , and j represents $\sqrt{-1}$. The operations on complex numbers are illustrated using the following Python statement. Here, $*$ performs complex multiplication, while the functions `abs()` and `conj()` return the absolute value and complex conjugate of the complex number.

```
In [13]: x*x, x*2, abs(x), conj(x)
Out[13]: (2j, (2+2j), 1.4142135623730951, (1-1j))
```

With this we close our discussion on Python's floats and complex numbers.

Conceptual Questions

1. For a computer implementation of real numbers, what are the advantages of floating-point representation over the fixed-point representation. Illustrate it using an example.
2. Nature has multiscale structure. Which among the two, floating-point representation and fixed-point representation, is more suitable for describing nature.

3. Why are the classical computers are 2-bit machines, not 3-bit or 4-bit machines?
4. What is the advantage of Two's complement representation of negative numbers over signed representation?
5. Precision for 32 bit machine.

Problems

1. Convert the following binary numbers to decimal numbers:
(a) 1.01 (b) 11.1 (c) 0.1001 (d) 1111.11 (e) 11.101
2. Convert the following decimal numbers to binary numbers.
Verify your results using Python function.
(a) 1/8 (b) 1/10 (c) 14.0 (d) 7/10 (e) 256.0
3. Convert the following hexadecimal numbers to decimal numbers:
(a) 0.F (b) F.F (c) 1.8 (d) 0.C (e) 0.A
4. Convert the following decimal numbers to hexadecimal numbers.
Verify your results using Python function.
(a) 1.6 (b) 1.4 (c) 0.9 (d) 1/3 (e) 0.256
5. What are the largest and smallest positive floats that can be stored in 32-bit and 64-bit computers? What are the corresponding negative floats? Also state the number of distinct real numbers that can be represented.
6. Give examples of real numbers that have inexact representations in both binary and decimal.
7. Give examples of real numbers that have exact representation in decimal, but inexact in binary.
8. Give examples of real numbers that have exact representation in base-3, but inexact in decimal.
9. In a hypothetical computer, double-precision representation requires 1 bit for sign, 7 bits for the exponent, and remaining bits for the mantissa. What are the smallest and largest real number that can be represented in such a machine. What is the precision of this machine?

3.3 Python Variables and Logical Variables

Python has variables like any programming language. A python variable is associated with an object, which could be integer, float, or of other data type. For example,

```
In [1]: x=8
```

In the above Python statement, x is associated with the object 8 ($x \leftrightarrow 8$). Here, $x = 8$ does not mean equality, rather, $=$ is an assignment operator. We can obtain the value of x by just typing x :

```
In [2]: x  
Out[2]: 8
```

We can perform operations on x , which is equivalent to operating on its associated object 8. For example,

```
In [3]: x**3  
Out[3]: 512
```

We can print x or any expression involving x using the *print* function:

```
In [4]: print(x, x**3)  
8 512
```

We can reassign x to another number, say 5.5, as shown below:

```
In [5]: x = 5.5
```

```
In [6]: x  
Out[6]: 5.5
```

After the statement *In [5]*, x is associated with the new number 5.5, which is a real number. Thus, a Python variable can be assigned to any data object, integer or real or string or array.

In Python, the integers, as well as other data types, are objects. The location and size of the objects can be determined using the *Python methods* (or *functions*), *id()* and *sys.getsizeof()*.

```
In [67]: x = 100  
In [68]: sys.getsizeof(x)
```

```
Out[68]: 28
In [76]: id(x)
Out[76]: 4563550448
In [77]: y=x
In [78]: id(y)
Out[78]: 4563550448
```

The *address* of the object 100 is 4563550448. That is, 100 is stored at this memory location. Note that the variables *x* and *y*, which are used for labelling the same object 100, have same address. In Section 4.4 we cover this topic in more detail.

Note that the size of the *data object* 100 is 28 bytes. Since $x \leq 2^{30}-1$, 4 bytes are used to store *x*, while the remaining 24 bytes are used for storing other attributes of the object. Note that the integers which are greater than 2^{30} but less than 2^{60} take 8 bytes to store. Larger integers require even larger number of bytes.

We can clear all the Python objects of a python session using *reset*:

```
In [10]: reset
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

We remark that several Python statements can be written in a single line by separating them by semicolons. For example,

```
In [47]: x=5; y = 6; z = x*y; print("x = ", x, "y =", y,
"z = ", z)
x = 5 y = 6 z = 30
```

In the print statement, “x =”, “y =”, and “z =” are strings that are printed as is.

Variable names and Keywords

Python variables have names that can be of any length. However, there are certain rules for Python names. They are

- A variable name must start with a letter or the underscore character (_).
- A variable name cannot start with a number.
- A variable name can contain alpha-numeric characters (A-Z, a-z, 0-9)and underscore (_). It cannot contain any other

- character including special characters, such as \$, %, etc.
- Variable names are case-sensitive. For example, *i* and *I* are different variable, so are India and india.

As a good programming practice, you should choose proper names for the variables. For example, for the city temperature, *temperature* is a good variable name, and *temp* is a terrible choice.

Python *keywords* are used for specific purposes, and they should not be used as variable names or function names. Here is the list Python keywords:

and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, ls, lambda, none, nonlocal, not, or, pass, raise, return, True, try, while, with, yield

Logical variables in Python

Logical variables and *logical expressions* play an important role in Python programming. We encounter them everywhere, specially in conditional statements and in loops (see Chapter Five). In honour of George Boole, who pioneered logic, logical variables are also called *Boolean variables*.

Logical data types and logical expressions take two values, *True* or *False*. For example,

```
In [28]: 5 < 6
Out[28]: True

In [29]: 5>6
Out[29]: False

In [33]: 5 == 6
Out[33]: False

In [34]: 5 == 5
Out[34]: True
```

In the above expressions, *<*, *>*, *==* are comparison operators that compares two operands and returns True or False. A list of comparative operator is given in [Table 5](#).

[Table 5](#): A list of comparative operators

Operator	Name	Example
----------	------	---------

<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

We can combine logical expressions using boolean operators *and*, *or*, *xor*, and *not* (*xor* operator is not available in Python). Refer to [Table 6](#) for the results of these operators. For example, *True and True = True*; *True or False = True*; *not True = False*.

[Table 6](#): Truth table ($T = \text{True}$ and $F = \text{False}$)

<i>A</i>	<i>B</i>	<i>A and B</i>	<i>A or B</i>	<i>A xor B</i>	<i>not A</i>
<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>F</i>	<i>F</i>
<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>
<i>F</i>	<i>T</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>T</i>
<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>

In case of multiple boolean variables and operators, the sequence of operations is decided by precedence of operators, which is listed in Table [Table 7](#).

[Table 7](#): Precedence of logical operator (from highest to lowest)

Operator	Function
<code><, <-, >, >=, !=, ==</code>	Comparison
<code>not</code>	Boolean not
<code>and</code>	Boolean AND
<code>or</code>	Boolean OR

We illustrate logical operators using the following examples. First, we start with the *or* operator.

```
In [38]: is_summer = (month == 5) or (month == 6) or
(month == 7)
```

```
In [40]: month = 5
```

```
In [41]: is_summer
Out[41]: True
```

is_summer is a logical variable that takes value *True* when the *month* is May (5th month), June (6th month), or July (7th month); it is *False* otherwise. After this, we illustrate the *and* operator.

```
In [7]: angles_equal = True
In [8]: sides_equal = True
In [9]: is_square = (angles_equal == True) and
(sides_equal == True)
In [10]: is_square
Out[10]: True
In [11]: sides_equal = False
In [13]: is_square = (angles_equal == True) and
(sides_equal == True)
In [14]: is_square
Out[14]: False
```

A quadrangle is a square if and only if all the sides and angles of the quadrangle are equal. The boolean variable *is_square* is *True* when both the conditions are *True*. The variable *is_square* is *false* when either of the two conditions is *false*. The above example illustrates operator the *and*.

After this, we illustrate the *not* operator. A person is well if he/she is not sick. Hence *is_well* = *not(is_sick)*.

```
In [16]: is_sick = True
In [17]: is_well = not(is_sick)
In [18]: is_well
Out[18]: False
```

Some more examples of boolean operations are here.

```
In [96]: P = True
In [97]: Q = False
In [98]: P and Q
Out[98]: False
In [99]: P or Q and P
```

```
Out[99]: True
In [100]: not P and Q
Out[100]: False
```

Another interesting feature of Python is that boolean values of any nonzero Integer, float, and string is *True*, while those of 0 and 0.0 are *False*. See the illustration below.

```
In [108]: bool(5), bool(-5.0)
Out[108]: (True, True)

In [112]: bool(0), bool(0.0)
Out[112]: (False, False)
```

Bitwise operations

At the end of this section we discuss bitwise operations on binary bits. Since 1 and 0 are treated as True and False respectively, we can extend the logical operations listed in [Table 6](#) to binary numbers. For example, $1 \text{ AND } 1 = 1$. In Python binary, AND operator is represented using symbol `&`. The symbols used for other binary operators are listed in [Table 5](#).

[Table 5](#): A list of bitwise operators

Operator	Name	Example
<code>&</code>	Bitwise AND	$5 \& 6 = 4$
<code>!</code>	Bitwise OR	$5 6 = 7$
<code>^</code>	Bitwise XOR	$5 ^ 6 = 3$
<code>~</code>	Bitwise NOT	$\sim 5 = -6$
<code><< m</code>	Bitwise left shift by m bits	$5 << 1 = 10$
<code>>> m</code>	Bitwise right shift by m bits	$5 >> 1 = 2$

We explain the above operations as follows:

- $5 \& 6 = (101)\&(110) = 100 = 4$
- $5 | 6 = (101)|(110) = 111 = 7$
- $5 ^ 6 = (101)^{(110)} = 011 = 3$
- $\sim 5 = \sim(101) = 010$, whose negative equivalence is -6 (for 3 bit numbers, $-(8-2)$). This convention differs from that of [Table 3](#).
- $\sim 11 = \sim(1011) = 0100$, whose negative equivalence is -12 (for 4 bit numbers, $-(16-4)$).

- numbers, $-(16-4) = -12$.
- In fact, $\sim n = -(n+1)$. Prove it!
 - $5 \ll 1 = (0101) \ll 1 = 1010 = 10$
 - $5 \gg 1 = (0101) \gg 1 = 0010 = 2$

Note that shift left by m bits yields multiplication by 2^m , while shift right by m bits yields integer division by 2^m . Thus, we can perform fast multiplication or division by 2^m using bit shifts. Such tricks are used in computers.

Conceptual questions

1. What are the differences between `=` and `==` operators of Python?
2. What is the efficient way to divide an integer by 8?

Exercises

1. In ipython, type the statement: `y = 89`. Obtain the memory location where 89 is stored. What is the output of `id(y)`?
2. Given three logical variables, `A = True`, `B=True`, and `C=False`, evaluate the following logical expressions:
`A and C`, `A or C`, `not C`, `A and B and C`, `A and B or C`, `not A or B or C`, `bool(10.0)`, `bool(0)`
3. Suppose `x = 5` and `y = 10`. What are the outputs of the following logical expressions?
`x < y`, `x <= y`, `x == y`, `x != y`, `x < y or x > y`,
`x<y and x >y`
4. State the results of the following bitwise operations:
`7 & 9, 7 | 8, 7 ^ 9, ~7, ~9, 6 << 2, 6 >> 2`

CHAPTER FOUR

PYTHON DATA TYPES: STRINGS, LIST AND ARRAYS

4.1 *Strings*

A Python *string*, which is an ordered sequence of character, is an important data types of Python. A character is a string with a single element. In Python, characters and strings are enclosed within single quotes ('') or double quotes (""); both these quotes are used interchangeably.

Some examples of characters and string are

```
In [1]: 's'  
Out[1]: 's'  
  
In [2]: "9"  
Out[2]: '9'  
  
In [7]: x = "This is a string."  
  
In [8]: x  
Out[8]: 'This is a string.'
```

Two strings can be added together using *concatenation operator*, +. Multiplication of a string by an integer, a *replication operation*, yields a larger string containing multiples of that string. Note that a string cannot be added to an integer. For example, 3+'9' is not allowed. We illustrate the above operations using the following examples:

```
In [70]: 'SS'+'T'  
Out[70]: 'SST'  
  
In [71]: 5*'S'  
Out[71]: 'SSSSS'  
  
In [72]: x = "This is a string."  
  
In [74]: x + " Hi, string"  
Out[74]: 'This is a string. Hi, string'
```

We also present several important Python escape sequences: \n for Linefeed (or next line), \u for unicode character, and \x for hex-encoded byte.

```
In [121]: y= "abs \ncde"  
  
In [123]: print(y)  
abs  
Cde  
  
In [126]: print('\u0905')
```

```
अ
```

```
In [132]: print('\x55')
U
```

Note that 0905 (*hex*) is a unicode representation for Hindi letter अ, and 55 (*hex*) is a representation of roman letter U. The characters U and अ require two and four bytes respectively for storage.

Two more important issues related to string data type are in order. To continue a string to the next line, one can use \ to indicate a line break to the interpreter. For example,

```
In [173]: 'I am going to the center of the \
...: Earth'
Out[173]: 'I am going to the center of the Earth'
```

Also, we define a *raw string* by prefixing the string using a letter r.

```
In [177]: r'$\alpha$'
Out[177]: '$\\alpha$'

In [178]: print(r'$\alpha$')
```

which is useful while printing symbols using latex (see Section 8.1). Without r, the character a is not printed, as shown below.

```
In [175]: print('$\alpha')
$\alpha
```

Function *str()* converts an integer or a float to the corresponding strings. For example,

```
In [150]: str(345), str(20.9), str(14.9e10)
Out[150]: ('345', '20.9', '149000000000.0')
```

We can read a string from the keyboard using the function *input()*.

```
In [134]: x = input("Enter the value of x =")
Enter the value of x =10

In [135]: x
Out[135]: '10'
```

Note that *input()* returns a string. Hence, in the above statement, x is not an integer. For the same, we need convert the input to an integer using *int()* function as shown below.

```
In [138]: x= int(input("Enter the value of x ="))
Enter the value of x =10
```

```
In [139]: x
Out[139]: 10
```

Here, the function `int()` converts string '10' to integer 10. Similarly we can employ `float()` and `complex()` functions to convert input string to float and complex number respectively. These functions help us read real and complex numbers as shown below.

```
In [145]: complex( input("Enter the value of x =") )
Enter the value of x =4+5j
Out[145]: (4+5j)
```

The function `print()` outputs *string*, *integer*, or a *float* to the screen. For example,

```
In [154]: x=5; y=10.0; z='hello'
In [156]: print('x = ', x, '; ', 'y = ', y, '; ', 'z = ',
z)
x = 5 ; y = 10.0 ; z = hello
```

String formatting in Python

Formatting of data plays a crucial role in input/output. Here we provide a brief introduction to formatting in Python.

Python employs *tuple string formatting* as follows:

```
In [140]: format = "My name is %s and I am %d year old"
In [141]: info = ("Vijay Verma", 30)
In [142]: print (format % info)
My name is Vijay Verma and I am 30 year old
```

The argument to the `print()` function is of the form—*StringOperand % TupleOperand*. Here *StringOperand* contains directives for combining the string, while the *TupleOperand* contains arguments that are substituted for `%s`, `%d`, `%f`, etc. The `%` operator placed between the *StringOperand* and the *TupleOperand* plays an important role.

Now let us print a float 1234.678968 to 2 decimal places, then we format it using “`%.2f`”. Here, the precision 2 is specified after the decimal point and before `f`.

```
In [148]: print("%.2f" % 1234.678968)
1234.68
```

Various formatting options for different data types of Python are listed in [Table 8](#). Most of the directives are quite straight forward. Note that the precision (p) is specified for float and float exponent.

[Table 8:](#) Formatting of different data types in Python

Directive	Data type	Example
%s	string	In [166]: "%s" % "Vijay Verma" Out[166]: 'Vijay Verma'
%i	integer	In [167]: "%i" % 109 Out[167]: '109'
%d	decimal integer	In [169]: "%d" % -109 Out[169]: '-109'
%x	hexadecimal integer	In [170]: "%x" % 255 Out[170]: 'ff'
%o	octal integer	In [171]: "%o" % 255 Out[171]: '377'
%f	float	In [172]: "%.3f" % 1234.678968 Out[172]: '1234.679'
%e	float exponent	In [173]: "%.2e" % 1234.678968 Out[173]: '1.23e+03'
%c	ASCII char	In [176]: "%c" % 98 Out[176]: 'b'

With this we close our discussions on strings.

Conceptual questions

- What are the advantage of special characters such as '\n'?

Exercises

- A book contains 5 million characters. Estimate the memory size required to store this book in the computer.
- What are the outputs of the following Python expressions?
 $2^{*} \text{Hi}'$, ' $\text{Hi}' + 2020$ ', ' $\text{Hi}' + '2020'$ ', $2^{*}3^{*} \text{Hi}'$, ' $\text{Hi}' + ' friend'$, ' $x' + 'y' + '='5'$ ', '\$\alpha\$'
- What is the output of the following Python statement?
In [75]: $x=5; y=10; \text{print}(x+y)'$
In [76]: $\text{print}(r'$\beta')'$

- In [77]: `print ("%x " % 254)`
4. Using python formatting, print floating point numbers to 5 significant digits after decimal. Write them in both float and exponential formats.
5. What are the differences between the following two Python statements?

```
x = input("Enter the value of x =")  
x = float(input("Enter the value of x ="))
```

4.2 List

In Python, a list is an ordered set of objects. Note that the elements of the list could of different data types.

For a list with N elements, the items of the list can be accessed using indices $0, 1, \dots, N-1$; or using indices $-N, -N+1, \dots, -1$. Note that the index in Python starts from 0, and that the last element of the list is accessed using $y[-1]$.

```
In [22]: y = [1,2,'hi']  
In [188]: print (y[0], y[1], y[2], y[-1], y[-2])  
1 2 hi hi 2
```

Table 9: indices of array $y = [1,2,'hi']$

index	0	1	2
index	-3	-2	-1
y	1	2	'hi'

Interestingly, string is also a list. For example,

```
In [189]: z = 'Python'  
In [191]: z[0], z[1], z[2], z[3], z[4], z[5], z[-1], z[-2]  
Out[191]: ('P', 'y', 't', 'h', 'o', 'n', 'n', 'o')
```

The functions associated with list are as follows.

Table 10: List of functions associated with list.

Function	Description
<i>append(element)</i>	Append <i>element</i> at the end of the list.
<i>pop()</i>	Remove the last <i>element</i> from the list. Returns this element
<i>pop(i)</i>	Remove the i^{th} element of the list, and returns this element.
<i>remove(element)</i>	Remove the first occurrence of <i>element</i> from the list.
<i>reverse()</i>	Reverse the list.
<i>sort()</i>	Sort the list.

<i>copy()</i>	Return a copy of the list.
<i>index(element)</i>	Returns the lowest index of the list for the <i>element</i> .
<i>count(element)</i>	Returns the number of occurrence of the <i>element</i> in the list.
<i>insert(ind,element)</i>	Insert <i>element</i> into the list at index <i>ind</i> .

Usage of list functions:

```
In [192]: a = [1, 3, 5, 9, 15]
In [193]: a.append(21)
In [194]: a
Out[194]: [1, 3, 5, 9, 15, 21]
In [196]: a.insert(2,4)
In [197]: a
Out[197]: [1, 3, 4, 5, 9, 15, 21]
In [199]: a.reverse()
In [200]: a
Out[200]: [21, 15, 9, 5, 4, 3, 1]
```

In addition, we can change an element of the list. For example,

```
In [212]: a[0]=100
In [213]: a
Out[213]: [100, 15, 9, 5, 4, 3, 1]
```

We can add two lists using + operator.

```
In [203]: a+[0,1]
Out[203]: [21, 15, 9, 5, 4, 3, 1, 0, 1]
```

We can also convert an input string to a list of characters using *list()* function.

```
In [209]: x=list('abcd')
abcd
In [210]: x
Out[210]: ['a', 'b', 'c', 'd']
```

Two useful list functions of Python are *size(a)* and *len(a)*. For one-dimensional list, these functions yield the size of its argument, *a*.

```
In [89]: a = [100, 15, 9, 5, 4, 3, 1]
In [90]: len(a)
Out[90]: 7
In [91]: size(a)
Out[91]: 7
```

Slicing

A *sublist* (a part of a list) can be accessed using *slice()* function:

```
slice(start, end, step)
```

where

- *start*: Start position of slice. This argument is optional, and its default value is 0.
- *end*: End position of slice. This argument is mandatory.
- *step*: Step of slice. This argument is optional, and its default value is 1.

For example,

```
In [227]: x = [100, 15, 9, 5, 4, 3, 1]
In [236]: s = slice(1,4)
In [237]: x[s]
Out[237]: [15, 9, 5]
In [238]: s = slice(1,4,2)
In [239]: x[s]
Out[239]: [15, 5]
```

Note that index of the array goes up to *end*-1. For example, *In [237]* lists $x[1] \dots x[3]$, not till $x[4]$.

Python offers a shortcut to *slice* function. We can skip defining the intermediate variable *s*, and directly apply *x[1:4:2]* that yields the same result as *In [238]*. That is,

```
In [169]: x[1:4:2]
Out[169]: [15, 5]
```

Also note that for a list x , $x[i:j]$ produces a sublist containing elements between indices i (inclusive) and j (exclusive). In case an argument is skipped, its default value is taken. For example, $x[:j]$ contains elements $x[0], \dots, x[j-1]$; while $x[j:]$ contains elements $x[j], \dots, x[-1]$ (including the last element). Thus $x[:i] + x[I:]$ covers the whole array. Also, $x[i:j:k]$ returns elements with a stride of k . If k is omitted, it is set to 1. We illustrate these features in the following examples.

```
In [227]: x = [100, 15, 9, 5, 4, 3, 1]
In [228]: x[1:4]
Out[228]: [15, 9, 5]
In [229]: x[1:-1]      # -1 last item, but it is skipped.
Out[229]: [15, 9, 5, 4, 3]
In [230]: x[::]
Out[230]: [100, 15, 9, 5, 4, 3, 1]
In [231]: x[::-1]
Out[231]: [1, 3, 4, 5, 9, 15, 100]

In [232]: x[::3]
Out[232]: [100, 15, 9]
In [233]: x[4:]
Out[233]: [4, 3, 1]
In [234]: x[4:-1]
Out[234]: [4, 3]
In [235]: x[4::-1]
Out[235]: [4, 5, 9, 15, 100]
```

Exclusion of $x[j]$ may appear illogical. However, this notation ensures that the length of the sliced list is $j-i$.

A related Python function is `range(start, end, step)` that generates a sequence of integers from `start` to `end-1` in steps of `step`. Here, `start` and `step` are optional. Their default values are 0 and 1 respectively. The rules for the `range()` are same as those for slice. Some illustrative examples of `range()` function are as follows.

1. `range(1,10,2)` will generate a sequence [1, 3, 5, 7, 9].
2. `range(5)` will generate a sequence [0,1,2,3,4]
3. `range(1,5)` will generate a sequence [1,2,3,4]
4. `range(5,1,-2)` will generate a sequence [5,3]

Note that `range()` does not return a *list*. It produces the integers on demand. However, we can access the elements of range object, as well as cast it into a *list*.

Examples:

```
In [239]: x=range(1,10,2)
```

```
In [240]: x[2]
Out[240]: 5
```

```
In [241]: list(x)
Out[241]: [1, 3, 5, 7, 9]
```

Array vs. List vs. String vs. Tuple

Python has a module called `array`, which is similar to `list` with a major difference. All the elements of array are of same data type, and they are stored at contiguous memory locations. The allowed data types for `array` are *signed char*, *unsigned char*, `Py_UNICODE`, *signed short*, *unsigned short*, *signed int*, *unsigned int*, *signed long*, *unsigned long*, *signed long long*, *unsigned long long*, `float`, `double`. We need to specify the data type code for them in the function. See for a list of data type code (<https://www.geeksforgeeks.org/python-arrays/>).

Table 11: Various data types of `array()` along with their codes

code	C Type	Python Type	Size in bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode char	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'L'	signed long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	4

Usage:

```
In [248]: import array as arr
In [249]: a = arr.array('i', [1, 2, 3])
In [250]: a[0]
Out[250]: 1
In [251]: type(a)
Out[251]: array.array
```

In In[249], 'i' stands integers.

Though *string* is a list of characters, it differs from a *list* in a significant way. No character of a *string* can be replaced. Also, neither a character can be appended to a string, nor popped from a string. This is because a *string* is an *immutable* object, but a *list* is a *mutable* (changeable) object. We will discuss these features in more detail in Section 4.4.

Similarly, a *tuple* is *immutable* list. That is, a *tuple* can be indexed and sliced as in a list. However, no element of a tuple can be destroyed or altered. A *tuple* is created using a slightly different syntax. For example, a *tuple* *y* is created as follows:

```
In [244]: y = (1,2,3)
In [245]: y[1]
Out[245]: 2
In [246]: y.pop()
-----
-----
AttributeError                                     Traceback (most
recent call last)
<ipython-input-246-0dece3cea1d5> in <module>
----> 1 y.pop()

AttributeError: 'tuple' object has no attribute 'pop'
```

Clearly, *pop()* function is not allowed for a *tuple*.

In the next section, we describe a new data type called *numpy array*, which are much more efficient than *list* for numerical computing.

Conceptual questions

1. What are the differences between a list and a tuple?

Exercises

1. Given a list $y = [10, 20, 40, 60]$, what the Python outputs for the following list operations? For each operation, start with the original y .
 $y.pop()$, $y.append(10)$, $y.reverse()$, $y.insert(2,3)$
2. Given a list $y = [10, 11, 12, 13, 14, 15, 16]$, what are the Python outputs for the following slicing operations?
 $y[0:2]$, $y[1:3]$, $y[2:-1]$, $y[:-1]$, $y[::-1]$, $y[:2]$, $y[-1:3]$, $y[-1:3:-1]$
3. What is the output of the following Python statements?
 $range(10)$, $range(5,10)$, $range(1,10,3)$, $range(5,1,-1)$
4. For a string $s = \text{"The Sun"}$, what are the outputs of $s[0]$, $s[2]$, $s[3]$, $s[-1]$, $s[-2]$?

4.3 Numpy Arrays

Numpy, an important Python package for scientific computing, provides an efficient implementation of multidimensional arrays. A *numpy* array contains data of same type, and its array size is fixed. The numpy arrays and associated functions are written in C language because of which arrays and associated functions become quite fast.

We need to *import numpy* to use numpy arrays. We could use the following statements:

```
Import numpy as np  
from numpy import *
```

Or, more simply, import *pylab* by invoking the following command at the terminal prompt (see Section 2.2).

```
ipython --pylab
```

Python statements in this section and in some later parts of the book correspond to either the later option or import using *from numpy import **. We meant to keep our discussion as simple as possible by avoiding constructs such as *np.array()*, *np.len()*.

Among the numpy functions, *array()* creates an array. The method to access array elements is same as that for a *list*.

```
In [265]: y=array([5,9])  
  
In [266]: y[1]  
Out[266]: 9  
  
In [267]: y.dtype  
Out[267]: dtype('int64')  
  
In [268]: y*2  
Out[268]: array([10, 18])  
  
In [269]: size(y) # yields count of elements in the array  
Out[269]: 2  
  
In [270]: len(y)  
Out[270]: 2
```

In a mixed array containing *int* and *float*, *numpy* converts types of all the elements to *float*.

```
In [275]: y=array([1.0,2])
```

```
In [276]: y.dtype
Out[276]: dtype('float64')
```

We can perform real operations on the array elements using functions *sqrt*, *log*, *log10*, *sin*, *cos*, etc. For example, for *y* of *In* [265].

```
In [448]: sqrt(y)
Out[448]: array([2.23606798, 3.])
```

The *+*, ***, */* operations on arrays yield respective element-by-element addition, multiplication, and division of the two arrays.

```
In [449]: z = array([1.0, 4.0])
```

```
In [450]: y+z
Out[450]: array([ 6., 13.])
```

```
In [451]: y*z
Out[451]: array([ 5., 36.])
```

```
In [452]: y/z
Out[452]: array([5. , 2.25])
```

```
In [271]: z = y+3
```

```
In [272]: z
Out[272]: array([ 8, 12])
```

For a numpy array *a*, the functions *sum(a)* provides the sum of all the elements of the array, while *prod(a)* provides the product of all the elements. These functions work for integer as well as float arrays.

```
In [7]: a=[2,3,6]
```

```
In [8]: sum(a)
Out[8]: 11
```

```
In [9]: prod(a)
Out[9]: 36
```

Also note that an integer list or a float list can be converted a numpy array as follows:

```
In [312]: y=[3,4]
```

```
In [313]: array(y)
Out[313]: array([3, 4])
```

The following numpy functions create useful arrays:

- *linspace(start, stop, num)*: Returns uniformly distributed numbers from start to stop.
- *arange([start,] stop[, step,])*: Returns numbers from start to stop (excluding) in steps of step. The arguments *start* and *step* are optional; their default values are 0 and 1 respectively.

Examples:

```
In [310]: arange(1,10,1.5)
Out[310]: array([1. , 2.5, 4. , 5.5, 7. , 8.5])
```

```
In [311]: linspace(1,10,6)
Out[311]: array([ 1. , 2.8, 4.6, 6.4, 8.2, 10. ])
```

```
In [317]: arange(5.0)
Out[317]: array([0., 1., 2., 3., 4.])
```

Note that

1. The number stop is excluded in the function *arange()*.
2. *arange()* returns an array, while *range()* does not return a list.
3. In the functions *arange()* and *linspace()*, *dtype* is an optional argument. The *dtype* for these functions is *float*.

In the above examples, the data elements were either *int64* (8-byte integers) or *float64* (8-byte floats). These are default integers and floats for *numpy*. However, *numpy* offers other types of arrays, which are listed in [Table 12](#). If our data consists of positive integers in the range 0 to 255, then it is best to use *uint8*, which uses much less computer memory. Also, operations on *uint8* are much faster than those on *int64*. To create an *int8* array *x*, we insert an argument *dtype='i1'* in the array function. See the example below.

```
In [189]: x=array([1,3,5],dtype='i1')
```

```
In [190]: x.dtype
Out[190]: dtype('int8')
```

To create *numpy* arrays with other data types, use the bracketed string shown in the first column of [Table 12](#).

[Table 12](#): Numpy data types

Data type	Description	Size (bytes)	Range
int_ ('i8')	default type int64	8	-2^{63} to $2^{63}-1$
int8 ('i1')	1-byte integer	1	-128 to 128
int16 ('i2')	2-byte integer	2	32768 to 32767
int32 ('i4')	4-byte integer	4	$-2^{31} \times 2^{31}$ to $2^{31} \times 2^{31}-1$
int64 ('i8')	8-byte integer	8	-2^{63} to $2^{63}-1$
uint8 ('u1')	unsigned 1-byte integer	1	0 to 255
uint16 ('u2')	unsigned 2-byte integer	2	0 to 65535
uint32 ('u4')	unsigned 4-byte integer	4	0 to 4294967295
uint64 ('u8')	unsigned 8-byte integer	6	0 to $2^{64}-1$
float_	default type float64	8	$\sim 10^{-308}$ to 10^{308}
float32 ('f4')	4-byte float	4	$\sim 10^{-38}$ to 10^{38}
float64	8-byte float	8	$\sim 10^{-308}$ to 10^{308}
complex_	default complex128	16	See float64
complex64	(float32,float32)	8	See float32
complex128	(float64,float64)	16	See float64
bool_	Default boolean	1	0, 1

Multi-dimensional arrays

The following examples illustrate creation of a two-dimensional (2D) array and several numpy functions associated with it. We create a two-dimensional array x as follows:

```
In [25]: x=array([[4,5,6], [7,8,9]])

In [26]: x
Out[26]:
array([[4, 5, 6],
       [7, 8, 9]])
```

Numpy array x has 6 elements arranged in 2 rows and 3 columns as

shown below. Note that x is an ordered array, that is, the elements have their assigned places and they cannot be interchanged.

4

5

6

7

8

9

We can access the elements of a 2D array using two indices $[i,j]$ or $[i][j]$. A diagram representing the variations of the two indices of the array x is shown below. Both the indices start from 0.

[0,0]	[0,1]	[0,2]
[1,0]	[1,1]	[1,2]

The first index represents the row, while the second index represents the column. In Python, the column index moves faster than the row index. That is, the data is stored in sequence as [0,0], [0,1], [0,2], [1,0], [1,1], and [1,2]. Note that the array elements also can be accessed as [0][0], [0][1], [0][2], [1][0], [1][1], and [1][2]. This arrangement, called *row-major order*, is also followed in C and C++. However, array storage in Fortran follows *column-major order*. We illustrate the above indexing using the following statement.

```
In [320]: print(x[0,1], x[0][1])
5, 5
```

Numpy contains linear algebra module that helps us compute *eigenvalues* and *eigenvectors* of an array. For example,

```
In [300]: x=array([[0,1],[1,0]])
In [301]: x
Out[301]:
array([[0, 1],
       [1, 0]])
In [316]: det(x) # determinant of x
Out[316]: -1.0
In [323]: y=eig(x)
In [324]: y
Out[324]:
(array([ 1., -1.]), array([[ 0.70710678, -0.70710678],
                           [ 0.70710678,  0.70710678]]))
```

In *Out* [324], $y[0] = [1., -1.]$ are the eigenvalues of x , while $y[1]$ contains

two lists that are the associated eigenvectors (of 1 and -1 respectively). In fact, y is a three-dimensional array, whose elements are accessed as follows:

```
In [328]: print(y[0][0], y[1][1], y[1][1][1])
1.0 [0.70710678 0.70710678] 0.7071067811865475
```

Note that $y[1,1,1]$ gives an error because y is a nonuniform array. We employ $y[i][j][k]$ to access elements of a 3D nonuniform array.

The following array is an example of a uniform 3D three-dimensional array (of size 2x2x2). Here, $y[I,j,k]$ is allowed.

```
In [100]: y = array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
In [101]: y
Out[101]:
array([[[1, 2],
       [3, 4]],
      [[5, 6],
       [7, 8]]])

In [102]: y[1,1,1]
Out[102]: 8
```

We can treat y as 2 layers of 2D arrays. Since y is a uniform array, we can access an element using $y[i,j,k]$.

Creation and operations with numpy arrays

The following *numpy* functions are used to extract size and shape of an array, as well as create new arrays.

size(a, axis = None): Returns the number of elements along a given axis. The default value is total number of elements.

shape(a): Returns the shape of an array.

zeros(shape, dtype=float): Returns a new array of a given *shape* and type filled with zeros.

ones(shape, dtype=float): Return a new array of given *shape* and type filled with ones.

vstack(tuple): Stacks numpy arrays along the first dimension (row-wise).

column_stack(tuple): Stacks numpy arrays along the second dimension (column-wise).

empty(shape, dtype=float): Return a new array of given *shape* and type without initializing entries.

random.rand(shape): Returns a float array of given *shape* with random numbers sampled from a uniform distribution in [0,1].

random.randn(shape): Returns a float array of given *shape* with random numbers sampled from a Gaussian distribution.

random.randint(low, high=None, size=None, dtype=int): Returns an integer array of given size with random numbers sampled from a uniform distribution in [low, high].

concatenate(a1, a2, ...): Concatenates arrays given as arguments.

Examples:

```
In [357]: a = ones((3,3))
```

```
In [358]: a
Out[358]:
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
In [359]: size(a)
Out[359]: 9
```

```
In [360]: shape(a)
Out[360]: (3, 3)
```

We can use the function *resize()* to convert an array from one shape to another.

```
In [361]: a = ones((4,4))
```

```
In [362]: a
Out[362]:
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
```

```
[1., 1., 1., 1.],
[1., 1., 1., 1.]])

In [363]: a.resize(2,8)

In [364]: print(a)
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]]

In [204]: random.rand(3,2)
Out[204]:
array([[0.03303746, 0.2236998 ],
       [0.20059688, 0.1309394 ],
       [0.68722795, 0.54985967]])

In [207]: random.randint(2,5, size=10)
Out[207]: array([3, 4, 3, 2, 2, 4, 3, 4, 2, 4])

In [209]: random.randint(2,5, size=(2,4))
Out[209]:
array([[4, 3, 4, 3],
       [2, 3, 3, 3]])

In [1]: x = array([1,2,3])

In [2]: y = 2*x

In [3]: vstack((x,y))
Out[3]:
array([[1, 2, 3],
       [2, 4, 6]])

In [29]: column_stack((x,y))
Out[29]:
array([[1, 2],
       [2, 4],
       [3, 6]])

In [107]: x=array([3,4,5]); y = array([45, 50])
In [109]: concatenate((x,y))
Out[109]: array([ 3,  4,  5, 45, 50])
```

Numpy function *transpose()* performs transpose operation on an array.
For example, for array *a* of *In* [362],

```
In [365]: a.transpose()
Out[365]:
array([[1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.]])
```

An array is indexed by a *tuple* of integers, as illustrated earlier. *Slicing* an array along any direction follows a similar rule as that for a *list* (discussed in Section 4.2).

Examples:

```
In [375]: y
Out[375]:
array([[ 1.           ,  2.26666667,  3.53333333,
4.8          ],
       [ 6.06666667,  7.33333333,  8.6           ,
9.86666667],
       [11.13333333, 12.4           , 13.66666667,
14.93333333],
       [16.2           , 17.46666667, 18.73333333,
20.          ]])
```

```
In [381]: y[0,:]
Out[381]: array([1.           , 2.26666667, 3.53333333,
4.8          ])
```

```
In [382]: y[0,0:2]
Out[382]: array([1.           , 2.26666667])
```

```
In [383]: y[0,0:3:2]
Out[383]: array([1.           , 3.53333333])
```

```
In [405]: y[0:2,1:3]
Out[405]:
array([[2.26666667, 3.53333333],
       [7.33333333, 8.6           ]])
```

We can also create a subarray by choosing a set of indices.

Examples:

```
In [416]: ia = array( ((1,0), (2,1)) )
In [417]: ja = array( ((0,1), (1,2)) )
In [418]: y[ia,ja]
Out[418]:
array([[ 6.06666667,  2.26666667],
       [12.4           ,  8.6           ]])
```

Here, *ia* and *ja* provide two columns of the subarray.

Another useful function is *meshgrid()*, which is used to create X and Y coordinates at the grid points. For example,

```
In [1]: row_x = linspace(0,0.3,4)
In [2]: col_y = linspace(0,0.2,5)
In [3]: X,Y=meshgrid(row_x, col_y)

In [4]: X
Out[4]:
array([[0. , 0.1, 0.2, 0.3],
       [0. , 0.1, 0.2, 0.3],
       [0. , 0.1, 0.2, 0.3],
       [0. , 0.1, 0.2, 0.3],
       [0. , 0.1, 0.2, 0.3]])

In [5]: Y
Out[5]:
array([[0. , 0. , 0. , 0. ],
       [0.05, 0.05, 0.05, 0.05],
       [0.1 , 0.1 , 0.1 , 0.1 ],
       [0.15, 0.15, 0.15, 0.15],
       [0.2 , 0.2 , 0.2 , 0.2 ]])

In [6]: X[:,0]
Out[6]: array([0. , 0.1, 0.2, 0.3])

In [7]: Y[:,0]
Out[7]: array([0. , 0.05, 0.1 , 0.15, 0.2 ])
```

The above example is an illustration of *meshgrid*. It follows the default indexing option, *indexing* = ‘xy’, in which $X[i,:]$ gives `row_x`, while $Y[:,j]$ gives `col_y`. This choice follows from the way plots are made—x coordinates vary as in `Out[4]` and y coordinates vary as in `Out[5]`. If we need *meshgrid* as in Python array notation, then we need to give option as *indexing* = ‘ij’.

```
In [12]: X1,Y1=meshgrid(row_x, col_y,indexing = 'ij')

In [13]: X1
Out[13]:
array([[0. , 0. , 0. , 0. , 0. ],
       [0.1, 0.1, 0.1, 0.1, 0.1],
       [0.2, 0.2, 0.2, 0.2, 0.2],
       [0.3, 0.3, 0.3, 0.3, 0.3]])

In [14]: Y1
Out[14]:
array([[0. , 0.05, 0.1 , 0.15, 0.2 ],
       [0. , 0.05, 0.1 , 0.15, 0.2 ],
       [0. , 0.05, 0.1 , 0.15, 0.2 ],
       [0. , 0.05, 0.1 , 0.15, 0.2 ]])

In [16]: X1[:,0]
```

```
Out[16]: array([0. , 0.1, 0.2, 0.3])
In [17]: Y1[0,:]
Out[17]: array([0. , 0.05, 0.1 , 0.15, 0.2 ])
```

The x and y meshgrid would be more complex for a nonuniform mesh. The function *meshgrid()* is useful for making *surface plots* and *contour plots*. See Section 8.1 for more details.

Vectorization

A processors can perform many operations in a single clock cycle if the data is arranged nicely in the memory. The gain is much more for multicore processors and supercomputers. We illustrate this idea using a simple example where we multiply two numpy arrays of same size.

```
a = np.random.rand(10**8)
b = np.random.rand(10**8)
c=(a*b)
```

In the above example, the computer operations are streamlined as in an assembly line of a car manufacturer. As a result, a large chunk of element-by-element multiplication could be performed in a single cycle. This process, called *vectorization*, is supported for numpy arrays. Here, optimised array operations like $c = a * b$ is much faster than that performed using a loop.

We illustrate the above by timing the above two code segments. We employ the Python function *datetime.datetime.now()* to capture the current time of the computer.

```
from datetime import datetime
import numpy as np

a = np.random.rand(10**8)
b = np.random.rand(10**8)
c = np.empty(10**8)

t1 = datetime.now()
c=(a*b)
t2 = datetime.now()
print ("for vectorised ops, time = ", t2-t1)

t1 = datetime.now()
for i in range(10**8):
    c[i] = a[i]*b[i]
t2 = datetime.now()
print ("for loop, time = ", t2-t1)
```

In my MacBook Pro 2014 model, vectorised array multiplication took 2.10 seconds, while the multiplication by an explicit loop took 20.06 seconds. Thus, vectorised array multiplication is approximately 10 times faster than the loop-based array multiplication. We remark that we could also capture time using *timeit* module as well:

```
import timeit  
t1 = timeit.default_timer()
```

With this we end our discussion
on numpy arrays.

Conceptual questions

1. What are the differences between list and numpy arrays?
2. Contrast the array access methods of Python, C, and Fortran.

Exercises

1. An image consists of 2048x1536 pixel, with each pixel is represented by 8 bits. Estimate the image size in kilobytes? Construct an numpy array to host this image.
2. Create a numpy *bool* array to work with 10^8 spins that take values 0 and 1. Store 1 in all of them. This arrangement will be useful for many applications related to Ising spins. One problem is that the Ising spins take values ± 1 . How will adopt the *bool* array for Ising spins.
3. Consider an array $x = \text{array}([1,2,3,4,5])$. What are the outputs of $x[1]$, $x[3]$, $x[-1]$?
4. Consider a list of integers $x = [4,5,6]$. Find square root of all the elements using Python. You code should work for a general integer list.
5. For the array $x = \text{array}([1,2,3,4,5])$, what are the outputs of $\text{sqrt}(x)$ and x^{**2} ?
6. What is the output of $\text{linspace}(0,10,11)$?
7. What are the outputs of the following: $\text{arange}(1,5,0.5)$, $\text{arange}(1,5)$, $\text{arange}(5)$, $\text{arange}(5,1,-0.5)$
8. Construct a 3x3 two-dimensional array $A[i,j] = (i+1)*(j+1)$. Find its eigenvalues and eigenvectors. Work with a float32 array.
9. Construct a 3D array of 1000x1000x100 size with random entries. Sum the numbers using *sum()* function and using a

loop. Compare the timings for the two operations. Verify that the mean values of the number matches with the expect result.

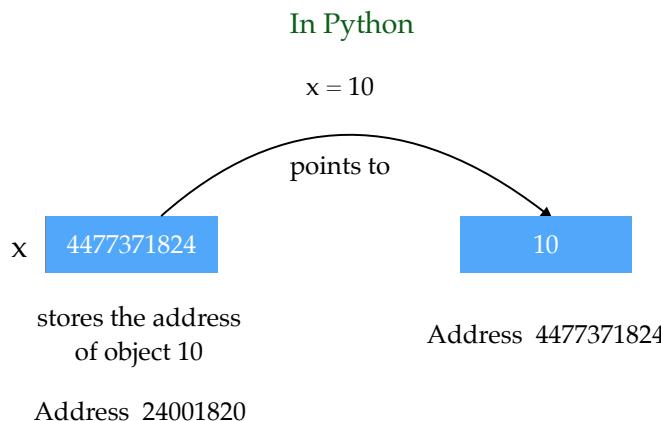
4.4 *Mutable and Immutable Objects in Python*

This section is a somewhat advanced, and it can be skipped by beginner programmers.

The Python data types—*integer*, *float*, *complex*, *string*, *list*, *array*, *tuples*—are objects. Python employs *reference type* to represent these objects. Here, the variable does not store the value of the object, but it stores the address where the value is being stored. We illustrate it using the following code segment.

```
In [32]: x = 10  
In [33]: id(x)  
Out[33]: 4477371824
```

As shown in [Figure 16](#), *x* stores the address (4477371824) of object “10”. Students familiar with C programming language will notice that Python variables are like C pointers.



[Figure 16](#): Illustration of a Python variable and its contents

Among the above Python data types, *integer*, *float*, *complex*, *string*, and *tuples* are immutable, but *list* and *array* are mutable. The value of an immutable object is unchangeable once it has been created. However, value of an mutable object can be changed. In the following discussion we will discuss the subtleties of immutable and mutable

Python objects.

Immutable Python Objects

Let us illustrate the immutability of a Python integer using an example.

```
In [32]: x = 10
In [33]: id(x)
Out[33]: 4477371824

In [34]: id(10)
Out[34]: 4477371824

In [35]: id(11)
Out[35]: 4477371856

In [36]: sys.getsizeof(10)
Out[36]: 28
```

Python statement *In* [32] creates an integer 10, which is an object. The location of the object, 4477371824, is stored in *x*. We can get the location of the object using *id*(10) or *id*(*x*). Every Python object has an *identity*, a *type*, and a *value*. For the object 10, 10 is the value, *id*(10) yields its *identity*, and *integer* is its *type*. We can think of variable *x* as a label to the object.

An important point to note that an integer in Python is an *immutable* object. The value of an immutable object is unchangeable once it has been created. The integer 10 of *In* [32] and *In* [34] are the same object.

Python statement *In* [35] creates another object, integer 11, whose location is 4477371856. Note that 11, which is 28 bytes away, is adjacent to 10. Interestingly an integer alone requires only 8 bytes of storage. The remaining 20 bytes are overheads that is used for storing other information of the object.

Let us explore some more intricacies of immutable object.

```
In [37]: y=x
In [38]: id(y)
Out[38]: 4477371824

In [39]: x = 100
In [40]: id(x)
Out[40]: 4477374704
```

```
In [52]: id(100)
Out[52]: 4477374704
```

```
In [53]: id(y)
Out[53]: 4477371824
```

After $x = 10$; $y = x$



After $x = 100$

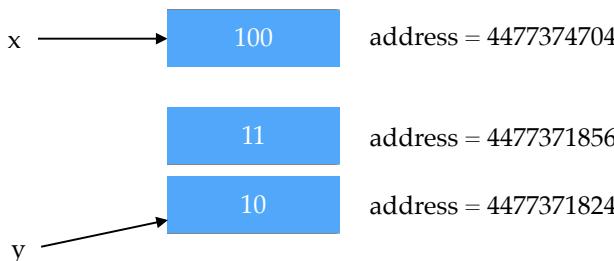


Figure 17: Illustrations of Python's *immutable* objects related to Python statements from *In*[32] to *In*[53].

In Python statement *In* [37], we provide another label *y* to the object 10 of statement *In* [32] (which is same as that of object 10 of *In* [34]). It is verified by statement *Out* [38] that yields the same *id* for *y* as that of object 10 or *x*. See [Figure 17](#).

In statement *In* [39] we associate label *x* to another object 100, whose *id* is 4477374704. Interestingly, *id(y)* is same as before, and it points to integer 10, though *x* now is label to object 100. See Figure 17 for an illustration of the above features.

We can illustrate the immutability of *float*, *complex*, *string* and *tuple* using similar examples.

Mutable Python Objects

List and *array* are mutable objects. We illustrate their mutability using the following examples.

```
In [1]: a = [1,2,3]
In [2]: b=a
In [3]: print(id(a), id(b))
4630230512 4630230512
In [4]: a.pop(1)
Out[4]: 2
In [6]: print(a, b)
[1, 3] [1, 3]
In [7]: print(id(a), id(b))
4630230512 4630230512
```

In the above statements, the *list* [1,2,3] is labelled by *a* and *b*, and they have same address (4630230512). In statement *In[4]*, we pop the element 2 from the list. Consequently, both *a* and *b* represent the new list [1,3]. Note that the addresses of *a* and *b* remain the same after the operation (see *In[7]*). See [Figure 18](#) for an illustration. Contrast the above feature with the earlier example where the address of *x* got changed on reassignment.

We create a copy of *a* using the statement:

```
In [8]: c = a[:]
In [9]: print(c, id(a), id(c))
[1, 3] 4630230512 4627991840
```

Note that the address of *c* differs from that of *a*. Thus, *c = a* and *c = a[:]* have different effects.

after $a = [1,2,3]$



after popping out 2 from $a = [1,2,3]$



after $c = a[:] = [1,2]$

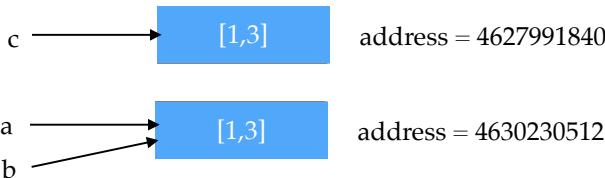


Figure 18: Illustrations of Python's *mutable* objects related to Python statements from *In[1]* to *In[9]*.

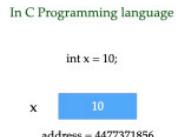
If a points to another list, then the *id* of a is changed, but b continues to point to the original object $[1,2,3]$. Similarly, $a = 5*a$ creates a new list, whose *id* will differ from that of original a . See the following example.

```
In [88]: a = [5,6]
In [89]: print(id(a), ' ', id(b))
4846573760 4630230512
```

Why do we choose lists and arrays as *mutable* objects? In computing, *arrays* and *lists* can be large objects with many elements. It is expensive to copy large arrays, hence it is prudent to use arrays as mutable objects.

Comparison of Python with C, C++, and Fortran

In programming languages *C*, *C++*, and *Fortran*, the data type is *value type*. Here, the variable holds the data value itself in its memory. As shown in [Figure 19](#), the variable *x* stores the integer 10. The address of variable *x* is 4477371824. Contrast this arrangement with that of Python (see [Figure 16](#)).



[Figure 19](#): Illustration of a C variable.

Further, in the following C code segment,

```
x = 5  
y = x
```

The variables *x* and *y* represent two different variables though they have same value 5. Hence, they have different memory addresses. When we reassign *x* to a new value using the following statement:

```
x = 6
```

The program simply changes the content of *x* to 6; it does not create a new integer 6. Contrast the above with the data representation in Python.

Conceptual questions

1. What are the differences between Python variables and C variables?
2. List three main differences between mutable and immutable Python variables.

Exercises

1. Execute the following Python statements: $x = 60.9$; $y = x$; $x = 99$. What are the id's of *x* and *y* at each stage of the code segment.

2. Execute the following Python statements: $x = \text{array}([3,4])$; $y = x$; $x = 99$. What are the id's of x and y at each stage of the code segment.
3. Consider the following Python statements: $a=[1,2,3]$; $b=[4,5,6]$; $a[:]=b$. What do the lists a and b contain after the statement, $a[:]=b$? What are their id's? Test your code in the computer.
4. We change the statements of Exercise (3), to $a=[1,2,3]$; $b=[4,5,6]$; $a=b$. Contrast your results with those of Exercise (3).

CHAPTER FIVE
CONTROL STRUCTURES IN PYTHON

5.1 Simple Statements

Computer programs, including Python codes, consists of the following basic structures:

1. Simple sequential statements
2. Conditional or Branching statements
3. Repetitive structures

In the present section and next two sections, we will cover these structures along with simple example. We start with sequential statements.

In the following sequential code segment, a set of Python statements are written one after the other. These statements are separated by semicolons (;). In a Python file, the semicolons can be avoided. Python interpreter executes the statements one after the other.

```
In [482]: x=50; \
...: y=100; \
...: print(x, ' ', y)
50    100
```

The following sequential code segment swaps two numbers a and b:

```
In [207]: a, b = 5,6
In [208]: tmp = a; a = b; b = tmp
In [209]: a,b
Out[209]: (6, 5)
```

A simple and more straightforward way to swap the numbers is as follows:

```
In [212]: a, b = 5,6
In [213]: b, a = a,b
In [214]: a,b
Out[214]: (6, 5)
```

Sequential statements are quite simple. We end this discussion here.

Exercises

1. Write a program to compute the roots of an equation $a x^2 + b x + c = 0$. Read the values of the parameters a, b, c (assume real) from the keyboard.
2. A projectile is fired with a velocity v and at angle θ from the horizontal. Write a Python code that prints the trajectory of this projectile.

5.2 Conditional Flows in Python

In a computer program, we perform operations based on certain conditions. The code segments related to conditionals are called conditional or branching program structures.

The following Python code segment illustrates a conditional flow of *if statement*.

```
In [491]: x = 3
In [492]: if (x%2 == 1):
...:     print('x is odd')
...:
x is odd
```

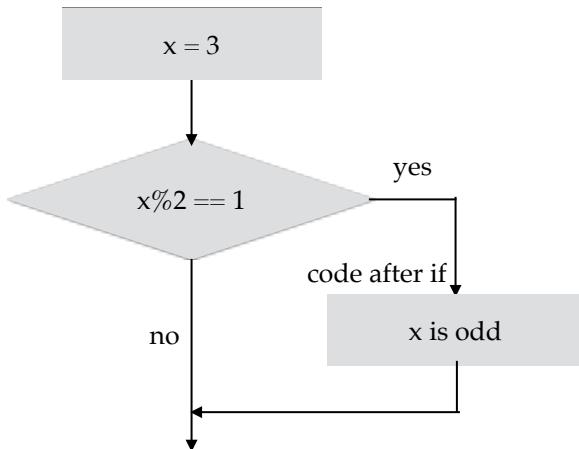


Figure 20: Illustration of an *if* conditional flow in Python. The rhombus contains the condition for the *if* statement.

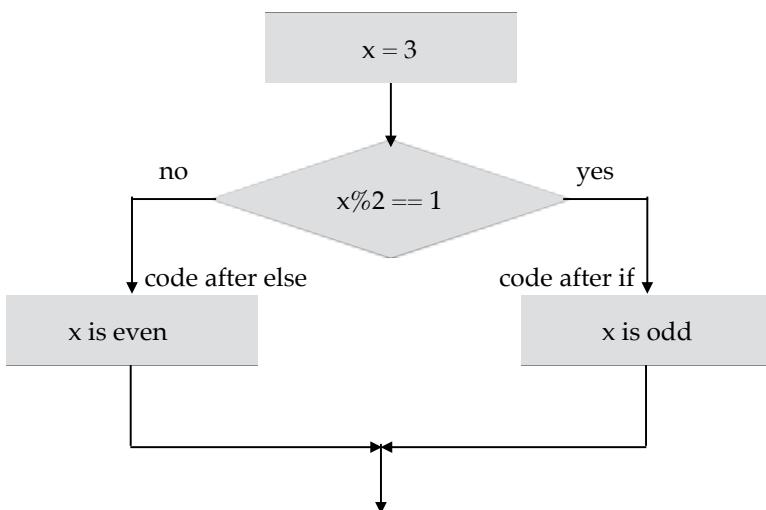
Often, flow control of a program is illustrated using a diagram, which is called *control-flow diagram*. [Figure 20](#) is the control-flow diagrams of the above *if* statement and the related structure.

In the code segment *In [492]*, `if (x%2 == 1)` is a conditional statement. The subsequent statement `print('x is odd')` is executed only if the logical expression `(x%2 == 1)` is True. The indentation (4 blank spaces) of the code block tells the Python interpreter what to execute after the *if* statement. An improper indentation will give an error.

Also, there could be more than one line of code in the *if* branch.

Now we illustrate the conditional flow of *if-else* statement using a code segment and [Figure 21](#).

```
In [493]: x=2
In [500]: if (x%2 == 1):
...:     print('x is odd')
...: else:
...:     print('x is even')
...:
x is even
```



[Figure 21](#): Illustration of an *if-else* conditional flow.

In the code segment, when $(x \% 2 == 1)$, the first print statement, *print ('x is odd')* is executed. Otherwise, the print statement after *else*: *print ('x is even')*, is executed.

The conditional and non-conditional code segments may themselves contain further conditional branches. See the example below, which is illustrated in [Figure 22](#). In this example, among the odd numbers, we further test if the number x is divisible by 3 or not.

```
In [501]: x=27
```

```
In [502]: if (x&2 == 0):
```

```

...: print('x is even')
...: elif(x%3 == 0):
...:     print('x is divisible by 3')
...: else:
...:     print('x is not divisible by 2 or 3')
...:
x is divisible by 3

```

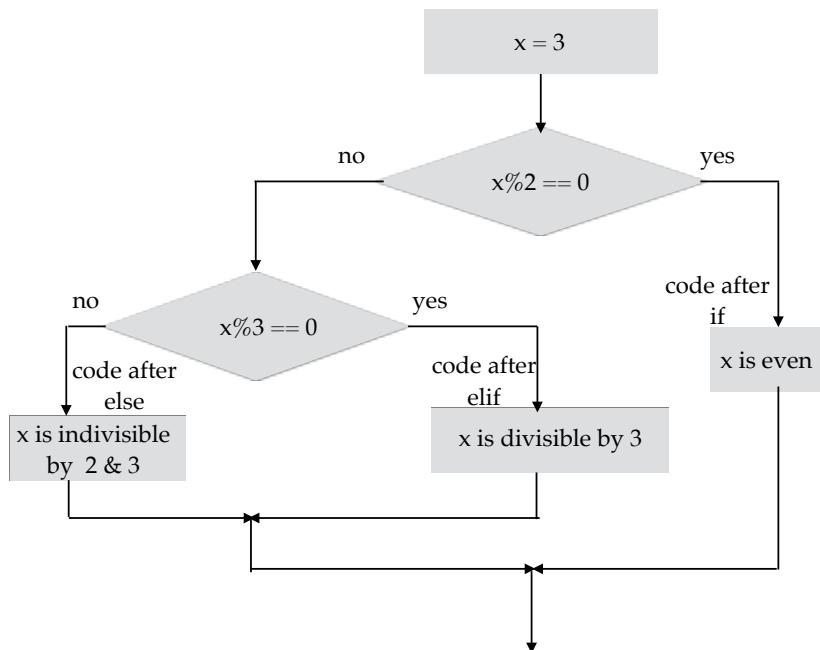


Figure 22: Illustration of an *if-elif-else* conditional flow.

Identification of a leap year involves more complex conditional statements. Consider `year = yr`.

```

If yr is divisible by 400
    then yr is a leap year
else if yr is divisible by 100
    then yr is not a leap year
else if yr is divisible by 4

```

```

then yr is a leap year
else yr is not a leap year

```

Python code for the above computation is as follows.

```

In [37]: yr = 1984
In [61]: if (yr%400 == 0):
....:     print ('A leap year')
....: elif(yr%100 == 0):
....:     print ('Not a leap year')
....: elif(yr%4 == 0):
....:     print ('A leap year')
....: else:
....:     print ('Not a leap year')

```

A shorter but more cryptic code is as follows:

```

In [37]: yr = 1984
In [39]: if (yr%400 == 0):
....:     print('A leap year')
....: elif (yr%4 ==0) & ~(yr%100==0):
....:     print('A leap year')
....: else:
....:     print('Not a leap year')
....:
A leap year

```

Exercises

1. Perfect square is a number whose square root is an integer. Examples of perfect squares are 25, 36. Write a Python code to test if a number is perfect square or not.
2. Armstrong number is a number that is equal to the sum of cube of its digits. One such number is 153. Test if a number is an Armstrong number or not. Hint: Use the functions `str()` and `int()`.
3. A palindrome string is one that reads the same forward and backward. Examples of palindrome strings are 'lal', 'KAK', '010'. Write a Python code that tests whether a string is a palindrome or not.
4. Write a program to compute the roots of an equation $a x^3 + b x^2 + cx + d = 0$. Read the values of the parameters a, b, c, d (assume real) from the keyboard. Test your result for $a = 1, b$

=6, $c = 11$, and $d = 6$.

5.3 Looping in Python

Computer programs typically involve repetitive structures. For example, to compute the sum of a numpy array, we need to repeatedly add the array elements. Python offers two repetitive structures: *for loop* and *while loop*, which are described below.

for loop

The structure of a *for loop* is as follows:

```
for elements in list:  
    body of the loop
```

The body of the loop is executed for all the elements of the list. We illustrate the loop structures using several examples.

Example 1: The following Python code prints the names of all students in the *student_list*.

```
In [508]: student_list = ['Rahul', 'Mohan', 'Shyam',  
'Shanti']  
  
In [509]: for student in student_list:  
...:     print(student, '\n')  
...:  
Rahul  
  
Mohan  
  
Shyam  
  
Shanti
```

[Figure 23](#) illustrates the control flow of for loop. The loop is carried out for all students in *student_list*. We start with index = 0, and continue till the *student_list* is exhausted. Note that the initialisation and index increment are not required in the code segment; they are performed by the Python interpreter internally.

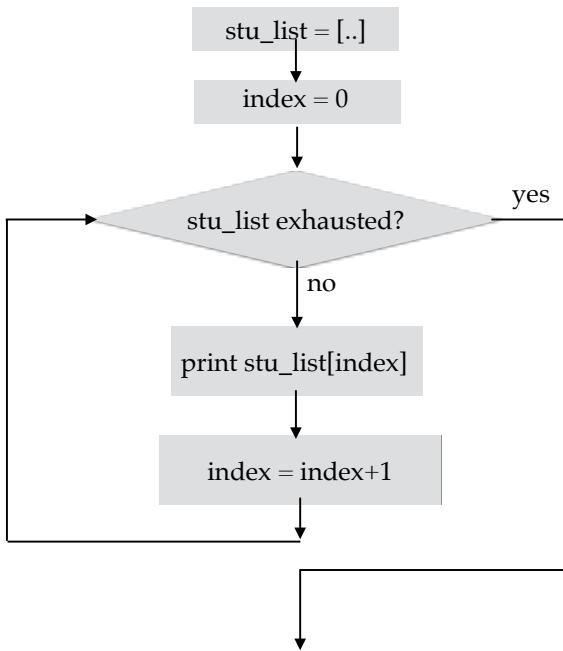


Figure 23: Illustration of the `for` loop of Example 1.

Example 2: The following code computes $\text{factorial}(n)$ using for loop.

```

In [510]: n = 5
....: fact=1
....: for i in range(2,n+1):
....:     fact = fact*i
....:

In [511]: print("Factorial of ", n, '= ', fact)
Factorial of 5 = 120
  
```

Here, `range(2,n+1)` creates a list containing integers 2 to n (excluding $n+1$). Hence, `for loop` is carried out for i ranging from 2 to n .

Example 3: Find the maximum number in a list.

```

In [42]: A = [6, 0, 4, 9, 3]
In [43]: max = -inf
  
```

```
In [44]: for x in A:
...:     if (x > max):
...:         max = x
```

```
In [45]: max
Out[45]: 9
```

In this example, we initialise `max` to $-\infty$ ($-\infty$). After that we loop over all the elements of `A`. If the element $> max$, then we replace `max` with the new element. In the end, `max` contains the maximum number of the list.

Another way to compute the maximum is as follows:

```
In [43]: max = A[0]
```

```
In [44]: for i in range(1, len(A)-1):
...:     if (A[i] > max):
...:         max = A[i]
...:
```

```
In [45]: max
Out[45]: 9
```

Here, we start `max = A[0]`, and loop from `A[1]` to the end of the array. As before, `max` is associated with the new element if it is larger than `max`. This is an index-based solution.

Example 4: In an integer array, send the maximum element to the end of the array.

```
n [46]: A = [6, 0, 4, 9, 3]
```

```
In [47]: for i in range(len(A)-1):
...:     if (A[i] > A[i+1]):
...:         A[i], A[i+1] = A[i+1], A[i]
...:
```

```
In [48]: A
Out[48]: [0, 4, 6, 3, 9]
```

In this example, among $A[i]$ and $A[i+1]$, the larger number is pushed to the right. The loop is carried out from $i=0$ to $n-2$, where n is the length of the array A . The above process is similar to the bubbling operation in which the lightest element rises to the top.

while loop

The structure of a *while* loop is as follows:

```
while (expression):
    body of while loop
```

The body of while loop is executed until condition of *while(expression)* statement remains *True*. The loop is terminated as soon as the condition becomes false. See examples below for illustrations.

Example 5: The following code computes $\text{factorial}(n)$ using a while loop. The control-flow diagram is shown in [Figure 24](#).

```
In [514]: n = 5
....: fact = 1
....: i = 2
....: while (i <= n):
....:     fact = fact*i
....:     i = i+1
....:
```

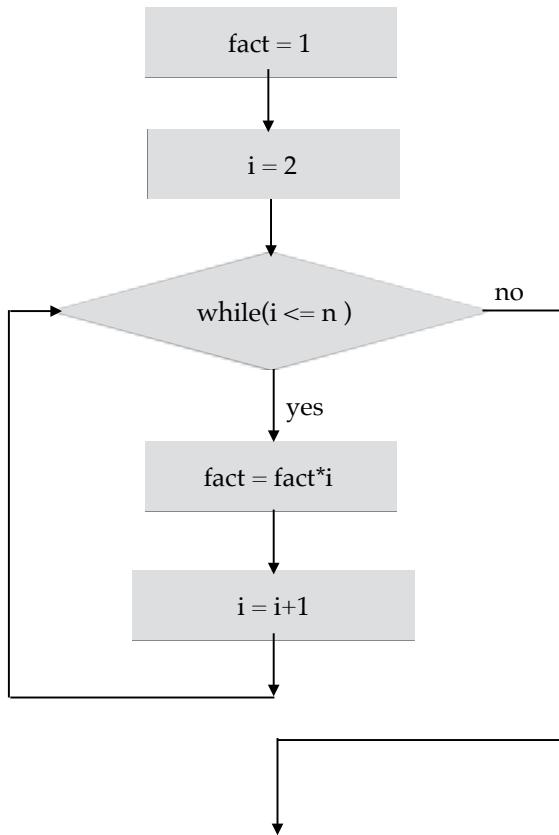


Figure 24: Illustration of the *while* loop of Example 5.

The while loop is executed until the running index $i \leq n$. The loop terminates as soon as $i = n+1$.

Example 6: Compute the greatest common divisor (gcd) of two integers a and b using Euclid's algorithm.

In [525]: $a, b = 2277, 1633$

In [526]: $\text{while } b:$
 $\dots: \quad a, b = b, a \% b$
 $\dots:$

In [527]: $\text{print}(a)$

23

In the above algorithm, $b < a$. The loop is continued till $b = 0$. Within the loop, $a \leftarrow b$ and $b \leftarrow a \% b$. See wikipedia for details.

Infinite loop and *break* statement

It is mandatory that all the loops in a program terminate. If not, the loop will continue forever. Such a loop, called *infinite loop*, must be avoided at all cost. An example of infinite loop is given below.

```
In [54]: i=1
In [55]: while (i>0):
...:     i = i+2
...:
```

You can easily check that the above loop does not terminate. You will have to kill the program using Cntrl-C (control-C). We may wish for a system software that can detect an infinite loop. Unfortunately, using a mathematical proof, Turing showed that it is impossible write such program. This celebrated theorem, called *halting problem*, is part of theoretical computer science.

A loop can be broken using a *break* statement. For example, we can break the above infinite loop using the following *break* structure.

```
In [263]: i=1
In [266]: while (i>0):
...:     i = i+2
...:     if (i>100):
...:         break
```

The above loop terminates as soon as i exceeds 100.

Loop ... *else* and repeat ... until

Python allows usage of *else* after *for* or *while* loop. The *else* block is executed when the loop finishes normally. We illustrate this structure using a code that tests if number n is a prime or not. In this program we use a property that one of the factors of a prime number must be less than or equal to \sqrt{n} .

```
n=15
# i ranges from 2 to sqrt(n) in steps of 1.
for i in range(2, int(sqrt(n))+1):
    if (n%i == 0):
```

```

print ("n is divisible by ", i, ")
Print("hence n is not a prime")
break
else:
    print ("n is a prime")

```

In this example, the *else* structure provides a simple way to contrast the nonprime numbers with prime numbers. Without *else* construct, the code gets more complex.

Python does not provide *repeat ... until* structure. However, it can be easily implemented using the following construct:

```

while (True):
    body of the loop
    if <logical condition>:
        break

```

In the next chapter we will discuss functions in Python.

Exercises

1. Write a Python code to find the minimum number in a list of numbers.
2. Write your own Python function to sum the numbers of a numpy array.
3. Write your own Python code to convert a binary number to decimal number. Assume the number to be a positive integer.
4. Write a Python code to construct a 5x5 two-dimensional array $A[i,j] = (i+1)*(j+1)$.
5. Pingala-Virahanka-Fibonacci numbers are defined as follows: $F_{n+2} = F_{n+1} + F_n$ with $F_0 = 0$ and $F_1 = 1$. Write a Python program to compute the first 15 Pingala-Virahanka-Fibonacci numbers. Compute the ratio F_{n+1} / F_n . Verify that the ratio converges to golden mean $(\sqrt{5}+1)/2$. Argue why does it converge to this number.
6. Iterate the function $x_{n+1} = f(x_n)$ where $f(x) = 4x(1-x)$. Start with $x_0 = 0.3$.
7. Write a computer program to find out the time taken to perform 10^9 addition, subtraction, multiplication, division for both integers and floating point numbers. Generate random numbers for the operation. Do the same for $x^{**}n$ with $n =$

2,4,8,16, and $\exp(x)$ for $x=1.939389$.

CHAPTER SIX
FUNCTIONS IN PYTHON

6.1 Functions in Python

A code segment that is used often can be abstracted as a function. A simple example of a function is $\text{sqrt}(x)$ that computes the square root of x . We do not need to or want to write this often-used function. It is better to use optimised function written by experts.

Functions in a programming language provide the following benefits:

- We can avoid repetition of codes.
- The user codes become smaller and readable with functions.
- Functions help modularise complex problems into smaller tasks. It is much better to write separate functions (each with 100 lines or less) and then combine them rather than writing a very long program, say of 5000 lines of code. Also, different functions can be kept in different files; this arrangement makes programming manageable.
- Modification of a program becomes easier with functions. For example, we may just replace a function by its optimised version. This will not alter the main program at all.

Similar to mathematical functions, a typical Python function takes arguments and returns results after computation. The *return* statement of the function provides the result of the function. See [Figure 25](#) for an illustration.

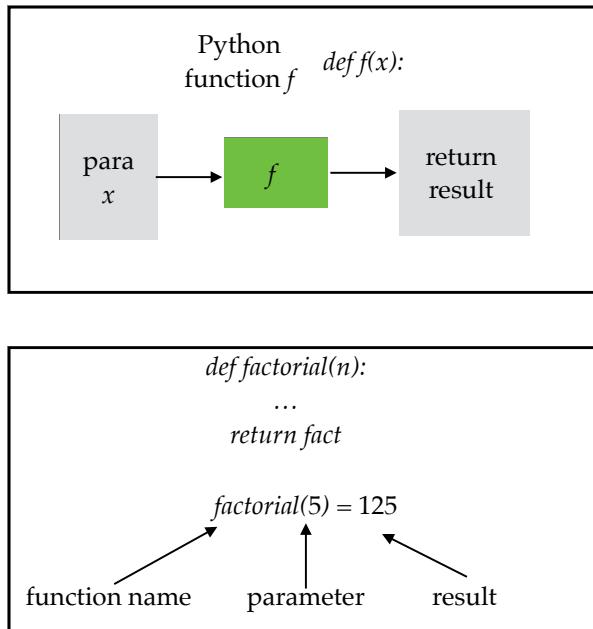


Figure 25: An illustration of a Python function.

A python function appears as follows:

```
def function_name (parameters):
    Body of the function
    Typically includes a return
statement
```

We illustrate several python functions below.

Example 1: Function `factorial(n)` that returns $n!$

```
def factorial(n):
    fact = 1
    for i in range(2,n+1):
        fact = fact*i
    return fact
```

Usage:

```
In [16]: factorial(8)
Out[16]: 40320
```

```
In [17]: factorial(9)
```

```
Out[17]: 362880
```

Example 2: A function that returns the sum the digits of an integer. Here, $\text{str}(n)$ converts integer n to a string of digits, while $\text{int}(k)$ converts digit (e.g., '1') to an integer.

```
import numpy as np

# Returns sum of the digits of n
def sum_digit(n):
    digits_n = list(str(n))
    # list of digits of n [Char entries]

    my_sum = 0
    for k in digits_n:
        my_sum += int(k)

    return my_sum

Usage:
In [94]: sum_digit(456)
Out[94]: 15
```

Example 3: A function that returns the maximum value of an array A .

```
# Returns maximum number of an array A.
def max_array(A):

    my_max = -inf

    for x in A:
        if (x > max):
            my_max = x

    return my_max
```

```
Usage:
In [92]: a=[5, 0, 9, 4]

In [93]: max_array(a)
Out[93]: 9
```

Example 4: A function that tests if number n is a Harshad number or not. Harshad number is one that is divisible by sum of its digits. Example: 21. In the code we make use of the function $\text{sum_digits}(n)$.

```
import numpy as np

# Harshad number is one that is divisible
# by sum of its digits
# is_harshad(n) retuns 1 if n is a Harshad number,
# 0 otherwise.
```

```
def is_harshad(n):
    return not(n % sum_digit(n))
```

Usage:

```
In [95]: is_harshad(21)
Out[95]: True
```

Example 5: A function that tests if number n is a Armstrong number or not. Armstrong number is one that is equal to the sum of cubes of its digits. Example: 153.

```
# An Armstrong number is one that is
# equal to sum of cubes of its digits.

# is_armstrong(n) 1 if n is an Armstrong number,
# 0 otherwise.

def is_armstrong(n):
    digits_n = list(str(n))
    # list of digits of n [Char entries]

    sum_cube = 0

    for k in digits_n:
        sum_cube += int(k)**3

    return (sum_cube == n)
```

Usage

```
In [96]: is_armstrong(153)
Out[96]: True
```

```
In [97]: is_armstrong(154)
Out[97]: False
```

Example 6: A function that tests if a string is a Palindrome or not. A Palindrome string is one that is same as its inverse.

```
# A Palindrome string is one that is same as its inverse.
# Here, x[::-1] returns a new string which is
# reverse of x.
# In x[::-1] means from end to 0 in steps of -1

# is_palindrome(x) 1 if x is Palindrome, 0 otherwise

def is_palindrome(x):
    return(x == x[::-1])
```

Usage

```
In [100]: is_palindrome('ABCD')
Out[100]: False

In [101]: is_palindrome('ABCCBA')
Out[101]: True
```

In the next chapter, we will construct more complex functions.

Local variables and passing arguments in Python

Variables inside a Python function are *local* to the function, and they cannot be accessed outside the function. For example, the variable *fact* of [Example 1](#) is local to the function *factorial(n)*. Note that the function exchanges the result to the caller of the function via a *return* statement. This feature is apparent in the above set of examples.

Inputs to the Python function are via arguments or parameters. Immutable variables (integer, float, string) and mutable variables (list and arrays) behave differently when they are passed as parameters. When an immutable variable (say *x*) is passed as an argument to the function, any change in *x* inside the function is not reflected outside the function. We illustrate this feature using an example.

```
In [6]: a = 5

In [7]: def my_sqr(x):
...:     x = x**2
...:     return x
...:

In [9]: print(my_sqr(a))
25

In [10]: a
Out[10]: 5
```

In the above example, during the function call *my_sqr(a)*, *a* is passed to *x* of *my_sqr(x)* as a parameter. That is, the argument *x* is replaced by *a*, which is 5. Inside the function, $x \rightarrow x^2 = 25$. However, the global variable *a* is unaffected, as is shown in *Out [10]*.

However, when a list or an array, which are mutable variables, is passed as an argument to a function, any alteration in the list or array during the function execution reflects in the caller. See the example below.

```
In [26]: b = [1,2,3]
In [30]: def change_array(x):
...:     x[0] = 100
...:     return
...:

In [34]: b = [1,2,3]
In [35]: change_array(b)
In [36]: b
Out[36]: [100, 2, 3]
```

During the function call `change_array(b)`, list `b` is passed `x` of `change_array(x)`. Inside the function, `x[0]` takes value 100. This change is reflected in `b` after the function execution (see `Out [36]`). This is unlike immutable variables that are unaffected by similar changes inside the function.

The above description captures the essential behaviour of arguments to the function. However, there are some subtle arguments that will be discussed in the next section.

Python function as objects

Python function is object, and its identity is obtained by the function `id()`. This feature makes function and data indistinguishable in many respects. For example, we can pass a function as an argument to a function, as illustrated in the following example.

```
In [138]: def mysqr(x):
...:     return x*x
...:

In [139]: def mycube(x):
...:     return x**3
...:

In [140]: def func(x, f): # f is a function
...:     return f(x)
...:

In [141]: func(2,mysqr)
Out[141]: 4

In [142]: func(2,mycube)
Out[142]: 8
```

lambda function

A lambda function in Python is an anonymous function. Such functions are defined as follows:

lambda arguments: expression

A lambda function can have any number of arguments but only one expression, which is evaluated and returned on a function call. A lambda function can be substituted for an object.

Example 7:

```
In [105]: sqr = lambda x: x**2
```

```
In [106]: print(sqr(5))
25
```

Example 8: The following Python statement computes $\int_0^1 dx \int_0^x dy (xy)$.

We will describe this function in Section 11.3.

```
print (dblquad(lambda x,y: x*y, 0, 1, 0, lambda x: x))
```

Example 8 illustrate the power of lambda function. The integrand, as well as the limits of the integrals, are inserted as arguments of *dblquad*.

With this we close our discussion on Python functions.

Conceptual questions

1. What are the advantages of functions in a programming language?

Exercises

1. Write a Python function that returns second minima of an integer array.
2. Write a function to generate n^{th} Pingala-Virahanka-Fibonacci numbers.
3. Write a function that returns x_n of function $x_{n+1} = f(x_n)$ where $f(x) = ax(1-x)$. Make the a and initial x_0 as parameters of the

function.

4. Write a function that takes two $n \times n$ matrices A and B and returns their matrix product. Solve for 3×3 and 4×4 matrices. You could try for $A[i,j] = i*j/100$ and $B[i,j] = (i+1)*(j+1)/100$.
5. Write a Python function that returns the number of digits of an integer.
6. Write a Python function that returns the number of digits of in the mantissa of a real number.
7. Write your Python function that converts a positive real number in decimal to binary. Assume fixed point format.
8. Write a computer program to discover positions of 4 queens on a 4×4 chess board so that the queens do not kill each other. List all possible solutions.
9. Solve problem 1 for n queens on a $n \times n$ chess board for $n = 5, 6, 7, 8$.
10. Knight's tour: Consider a 4×4 chess board. Write a computer program to construct move sequence of knight tour such that the knight visits every square of the board exactly once. List all possible solutions.
11. Solve problem 3 for the night on $n \times n$ chess board.
12. We have 4 coins with one among them heavier than the rest. Write a computer program to identify the defective coin using least possible weighing. Make a tree diagram for the solution. **Hint:** For 4 coins, weigh coins 1 and 2 in one side, and coins 2 and 3 in the other side.
13. Extend Exercise 5 to 8 coins and to 12 coins. Find solutions for both these cases.
14. We have 4 coins among which one of them is defective (could be lighter or heavier). How will identify the defective coin?
15. Do Exercise 8 for 8 coins and 12 coins.

6.2 Subtleties in Passing argument in Python

This section contains advanced discussions on argument passing in Python language. I recommend that the beginning programmer can skip this section.

As discussed in Section 4.4, Python variables are like pointers. This aspect has subtle effects during the argument passing to a function. We illustrate these effects using several examples.

Example 1: Argument passing of an immutable object (integer) to a Python function

```
In [90]: def test_fn(x):
...:     print('inside function, step 1: ', x, ' ', 
id(x))
...:     x = 999.0
...:     print('inside function, after x = 999.0: ', 
x, ' ', id(x))
...:

In [91]: x = 100.0

In [92]: id(x)
Out[92]: 4846932752

In [93]: test_fn(x)
inside function, step 1:  100.0    4846932752
inside function, after x = 999.0:  999.0    4846935088

In [94]: x
Out[94]: 100.0

In [95]: id(x)
Out[95]: 4846932752
```

In the above Python statements and function, during the execution of *In [91]* x takes value 100.0, and its id is 4846932752. We can treat x as a *global variable*. On function call $test_fn(x)$, the object reference is passed to the function. During the function execution, x remains the same as the global variable in the first line of the function. However, in the second line, x is assigned to 999.0. Following the discussion of Section 4.4, now x of $test_fn()$ points to the new object 999.0, whose address is 4846935088. However, the global variable x remains the same after the function execution. See [Figure 26](#) for an illustration. The variable x inside the function is local to function, and hence any change in x is not reflected in the global variable x .

Argument passing of a mutable object

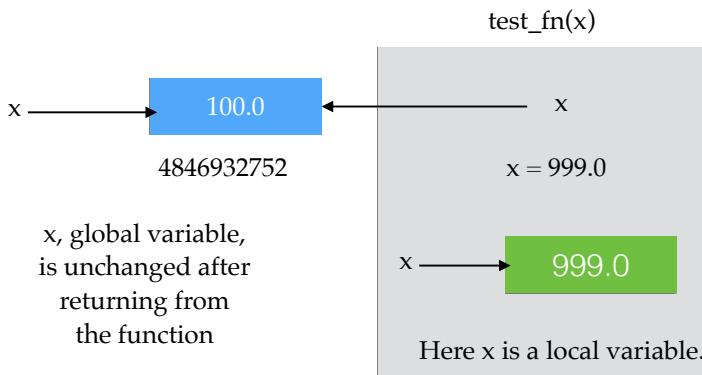


Figure 26: Argument passing of an immutable object to a Python function

Thus, the variables inside a Python function behave *almost* like local variables of C, where the arguments are passed using *call by value* scheme. A main difference between Python and C is that in C, a new variable would have been created at the function call itself.

Example 2: Argument passing of a mutable object (list) to a Python function is illustrated using the following code segment.

```
In [115]: def test_fn_array(x):
    ...:     print('inside function, step 1: ', x, ' ', 
id(x))
    ...:     x[0] = 30
    ...:     print('inside function, after reassignment
of x[0]', x, ' ', id(x))
    ...:

In [116]: x = array([1,2,3])

In [117]: id(x)
Out[117]: 4845358176

In [118]: test_fn_array(x)
inside function, step 1:  [1 2 3]  4845358176
inside function, after reassignment  [30  2  3]
4845358176

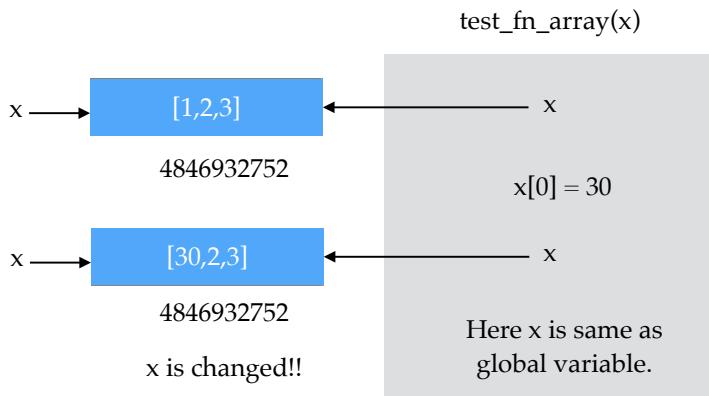
In [119]: x
```

```
Out[119]: array([30, 2, 3])
```

```
In [120]: id(x)
Out[120]: 4845358176
```

Since *array* is a *mutable* object, after the function call *test_fn_array(x)* at *In [118]*, the array *x* inside the function remains the same as the original array. Inside the function the array *x* is altered (*x[0] = 30*). Since the global array *x* is the same object as that inside the function, *Out [119]* prints array *x* as [30, 2, 3]. See [Figure 26](#) for an illustration.

Argument passing of a mutable object



[Figure 26](#): Argument passing of a mutable object to a Python function

Example 3: The following *swap* function for integers does not work.

```
# This function does not swap the vars a,b
# of the main program
# (Reference Type, or call by value in some sense)
def swap_not_working(a,b):
    temp = a
    a = b
    b = temp
    print (a, b, temp, id(a), id(b), id(temp))
    return
```

```
In [323]: a
Out[323]: 5
```

```
In [324]: b
Out[324]: 50
```

```
In [325]: id(a), id(b)
Out[325]: (4453578000, 4453579440)

In [329]: swap_not_working(a,b)
5 5 4453579440 4453578000 4453578000

In [330]: print(a, b, id(a), id(b))
5 50 4453578000 4453579440
```

In the python statement *In* [325] we pass the global variables *a* and *b* as arguments to the function. Inside the function, the variables *a* and *b* have been swapped (see *In* [329]), but the changes are lost when the function exits. This is because *a* and *b* inside *swap_not_working(a,b)* are local variables of the function. According to *In* [330], the global variables remain unaffected by the happenings inside the function.

A side remark: Contrast the above behaviour with C functions. If *a* and *b* were passed as pointers to integers, then *swap* would work. However, the C function will not swap when the arguments are passed by value.

Example 4: Rework the *swap* function so that it works.

```
return → return (a,b)
swap_not_working(a,b) → (a,b) = swap_not_working(a,b)
```

The latter statement passes new *a* and *b* (that are swapped) to global variables *a* and *b*. Another version (simpler one) of *swap* function that works is given below.

```
# Swap a and b
def swap(a,b):
    return b,a

Usage: a , b = swap(a,b)
```

Even a simpler way to swap is the following:

```
In [308]: a, b = b,a
```

Example 5: Swapping of two lists

```
# Swaps two lists a and b
def swap_list(a, b):
    temp = a[:]
    # copies elements of a into tmp (new array).
```

```
# differs from tmp = a
a[:] = b
# copies elements of b into a

b[:] = temp
# copies elements of tmp to b

print (a, b, temp, id(a), id(b), id(temp))
Return

In [365]: a = [1,2]
In [366]: b = [10,20]
In [367]: id(a), id(b)
Out[367]: (140716150723760, 140716130405904)

In [369]: swap_list(a,b)
[10, 20] [1, 2] [1, 2] 140716150723760 140716130405904
140716132670352

In [371]: a, b, id(a), id(b)
Out[371]: ([10, 20], [1, 2], 140716150723760,
140716130405904)
```

In the function *swap_list(a, b)*, the lists *temp* and *a* behave differently than the previous swap example. This is because *list* is mutable. Note that the code will not work if

```
temp = a[:] → temp = a
```

The latter statement does not copy *a* to *temp*; rather gives another name to the same list. You can easily verify why the code will not work.

Example 6: Squaring a numpy array

```
In [47]: b = array([1,2,3])
In [48]: id(b)
Out[48]: 140708012226560

In [49]: def my_sqr(x):
...:     x = x**2
...:     print (x, id(x))
...:     return
...:

In [50]: print(my_sqr(b))
[1 4 9] 140708036299952
None
```

```
In [51]: print(b, id(b))
[1 2 3] 140708012226560
```

In the above example, array b is passed to the function $my_sqr(x)$. Hence, Inside the function, $x = b$. However, the operation $x = x^2$ creates a new array that contains square of elements of b , as is evident from In [50]. Note however that original b remains unaffected by the squaring operation and on function exit (see In [51]). Similar behaviour would be observed for other array functions such as $\sqrt{()}$, $\sin()$, $\exp()$, etc.

The above examples show that the argument passing in Python is tricky. Sometimes codes do not produce expected outputs due to the subtle reasons mentioned above. We need to be careful on such occasions.

Conceptual questions

1. What are the differences between Python local and global variables? Contrast these differences when the arguments are immutable and mutable objects.

Exercises

1. In Example 5, we replace $temp = a[i]$ with $temp = a$. What are the consequence of this change?

6.3 Recursive Functions

In mathematics, a recursive function is one that is defined using its own definition. For example, $\text{factorial}(n)$ can be defined as

$$\begin{aligned} n! &= n(n-1)! \text{ For } n > 1, \text{ and} \\ 1! &= 1 \end{aligned}$$

Such a definition is called a *recursive* definition.

The above function works as follows. For $n = 3$, $3! = 3 \times 2!$, $2! = 2 \times 1!$, and $1! = 1$. The last condition, $1! = 1$, helps terminate the recursive process. After the last step, reverse substitution yields $2! = 2$ and $3! = 3 \times 2 = 6$, which is the answer.

Python and many modern languages offers recursive definition of functions. In the following discussion, we provide some examples of recursive Python functions.

Example 1: Recursive implementation of $\text{factorial}(n)$

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

It works as follows:

```
factorial(3) = 3*factorial(2),
factorial(2) = 2*factorial(1),
factorial(1) = 1.
```

After this, the reverse substitution yields

```
factorial(2) = 2*1 = 2,
factorial(3) = 3*2 = 6. (Answer)
```

The above recursive implementation of $\text{factorial}(n)$ yields the same result as in [Example 1](#) of Section 6.1. However, the recursive function is quite expensive because it makes forward calls to itself, and then does reverse substitutions. Each function calls requires significant bookkeeping (e.g., saving local variables etc.).

Recursion does not provide any real benefit in the implementation of factorial function. However, some problems are much easier to solve using recursion. One such problem is Tower of Hanoi.

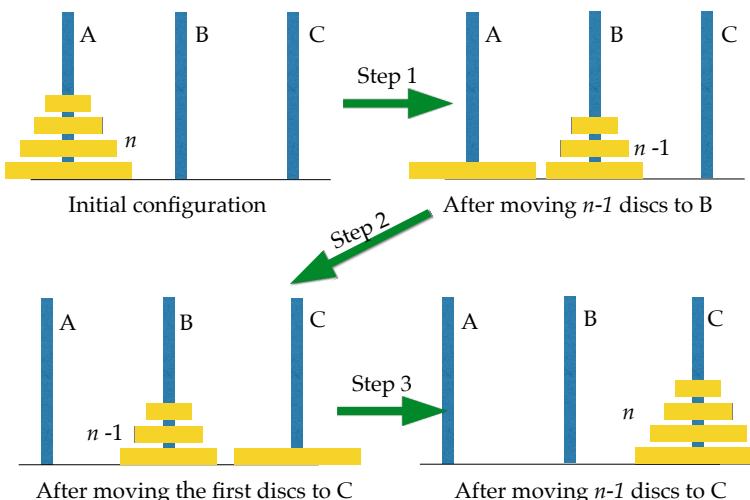
Example 2: Tower of Hanoi: Consider 3 pegs, A, B, C, as shown in the initial configuration of [Figure 27](#). Peg A has n discs, but pegs B and C have none. We need to transfer n disks from peg A to peg C obeying the following rules:

1. Only one disk can be moved at a time.
2. Only the top disk of a peg can be moved.
3. No disk can be placed on top of a smaller disk.

A recursive solution for the above problem is as follows. For $n > 1$,

1. Transfer $n-1$ disks from A to B.
2. Transfer one remaining disk from A to C.
3. Transfer $n-1$ disks from B to C.

However, for $n = 1$, the disk can be transferred in a single step, as long as rule 3 is obeyed. We illustrate the above steps in [Figure 27](#). A Python code is given after the figure.



[Figure 27](#): Illustrations of the three steps for solving the tower of Hanoi

problem.

```
def destwer_of_hanoi(n,source,dest,intermediate):
    if (n==1):
        print("transfer disk ", source , " dest ", dest)
    else:
        destwer_of_hanoi(n-1, source, intermediate, dest)
        print("transfer disk ", source , " dest ", dest)
        destwer_of_hanoi(n-1, intermediate, dest, source)
```

In [2]: run tower_of_hanoi.py

```
In [3]: tower_of_hanoi(3,"A","B","C")
transfer disk A to B
transfer disk A to C
transfer disk B to C
transfer disk A to B
transfer disk C to A
transfer disk C to B
transfer disk A to B
```

In the function definition, we employ *source*, *dest*, and *intermediate* as the peg labels. This is because source and destination pegs vary at different stages of function calls.

Based on the steps of the algorithm, we can estimate the number of moves required to transfer n disks. If $N(n)$ is the total number of moves to transfer n disks, then the recursive definition yields the following relation between $N(n)$ and $N(n-1)$:

$$N(n) = 2 N(n-1) + 1 \dots (4)$$

and $N(1) = 1$

We attempt solution of the form $N(n) = a^n + b$. The condition $N(1) = 1$ yields $a+b = 1$. Substitution of $N(n) = a^n + b$ in Eq. (4) and usage of $a+b=1$ yields

$$a^n = 2 a^{n-1} + 2 - a$$

For $n = 2$, the quadratic equation yields two solutions: $a = -1$ and 2 . The former solution is invalid because $N(n)$ is an increasing function of n . Therefore,

$$N(n) = 2^n - 1,$$

which is the time complexity of the algorithm.

Lastly, Tower of Hanoi has an iterative solution as well. However, it is much more complex to implement. Here, the recursive solution is very intuitive and straight forward.

Example 3: *Sierpinski triangle*: Write a computer program to draw *Sierpinski triangle* shown in Figure 28.

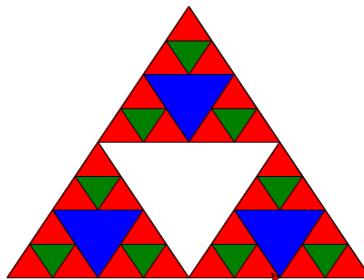


Figure 28: Sierpinski triangle of degree 3.

In Figure 28, we illustrate a Sirpiński triangle of fractal degree 3. Note that each triangle has three sub-triangles, whose dimension is half of its predecessor (for example, blue triangle within a red triangle). Further, the sub-triangles have even smaller triangles (subsub-triangles) within them. For fractal degree 3, the recursive iteration stops at the innermost green triangles. Here, the white, red, blue, and green triangles have *fractal degrees* of 3, 2, 1, 0 respectively. Such objects are *self-similar*. That is, smaller triangles have similar structure as the large ones.

An algorithm to construct Sirpiński triangle is as follows.

```
def draw_sirpinski(vortex_coords, fractal_degree):
    Draw a triangle of fractal_degree.

    if (fractal_degree > 0):
        Find the coordinates of the triangle
            at the left corner
        draw_sirpinski(left_triangle_vortex_coords,
                      fractal_degree-1)

        Find the coordinates of the triangle at
            the right corner
        draw_sirpinski(right_triangle_vortex_coords,
```

```

fractal_degree-1)

Find the coordinates of the triangle at
the top corner
draw_sirpinski(top_triangle_vortex_coords,
fractal_degree-1)

```

The code that produces the Sierpinski triangle with fractal degree of `fractal_degree` is given below. It employs *turtle module* of Python. The *turtle* could move forward by a distance d (`forward(d)`), turn left by any angle θ (`left(θ)`), turn right by an angle ζ (`right(ζ)`), go to a point (x,y) (`goto(x,y)`), and perform several other tasks. Refer to the turtle manual for more details.

```

import turtle

colormap = ['red','green','blue',
'white','yellow','violet','orange']

# Draws a triangle of color 'color' and whose vortices are
# in the list vortex_coords.
def drawTriangle(vortex_coords,color,fractalturtle):
    fractalturtle.fillcolor(color)
    fractalturtle.up()
    fractalturtle.goto(vortex_coords[0][0],
                      vortex_coords[0][1])
    fractalturtle.down()
    fractalturtle.begin_fill()
    fractalturtle.goto(vortex_coords[1][0],
                      vortex_coords[1][1])
    fractalturtle.goto(vortex_coords[2][0],
                      vortex_coords[2][1])
    fractalturtle.goto(vortex_coords[0][0],
                      vortex_coords[0][1])
    fractalturtle.end_fill()

# Returns the mid point of points p1 and p2
def mid_point(p1,p2):
    return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)

# Draws sierpinski triangle of fractal_degree with its
# def
def sierpinski(vortex_coords,fractal_degree,fractalturtle):

    drawTriangle(vortex_coords,colormap[fractal_degree],fractalturtle)
    if fractal_degree > 0:

        sierpinski([vortex_coords[0],
                   mid_point(vortex_coords[0], vortex_coords[1]),
                   mid_point(vortex_coords[0], vortex_coords[2])],
                  fractal_degree-1, fractalturtle)

        sierpinski([vortex_coords[1],

```

```

        mid_point(vortex_coords[0], vortex_coords[1]),
        mid_point(vortex_coords[1], vortex_coords[2])),
        fractal_degree-1, fractalTurtle)

    sierpinsk([vortex_coords[2],
              mid_point(vortex_coords[2],vortex_coords[1]),
              mid_point(vortex_coords[0],vortex_coords[2])],
              fractal_degree-1, fractalTurtle)

def main():
    fractalTurtle = turtle.Turtle()
    Win = turtle.getscreen()
    vortex_coords = [[-200,-100],[0,200],[200,-100]]
    sierpinsk(vortex_coords,3,fractalTurtle)
    Win.exitonclick()

main()

```

The above code appears quite complex, but it is not so. Some of the key features of the code are as follows.

1. The centre of the largest triangle is at the origin, while its three vortices are at $(-200, -100)$, $(0, 200)$, $(200, -100)$. The vortices are stored in the array `vortex_coords`. The vortices of the inner triangles are computed iteratively.
2. In `main()`, we create `fractalTurtle` using the function `turtle.Turtle()`; this `fractalTurtle` is passed as an argument to the functions `sierpinsk()` and `drawTriangle()` because the same turtle moves in these functions.
3. Using the array `colormap`, we set the colours of the triangles of levels 3, 2, 1, 0 as white, red, blue, and green respectively. Note that level n triangles are drawn on top of level $n-1$ triangles, hence only the remnants of the earlier triangles are visible in the figure.
4. The color-filling of the triangles is achieved by the functions `fillcolor()`, `begin_fill()`, and `end_fill()`. The colors are filled after the triangles have been drawn.

Since the turtle moves on turtle console, the above code cannot run in ipython. It is best to run this code on *Spyder* that supports turtle console.

Recursion is a powerful tool for many computational problems, e.g. binary search, Fast Fourier Transform, etc. However, majority of scientific algorithms employ iterative solutions because they are faster than the recursive counterparts.

With this we end our discussion on recursive functions.

Conceptual questions

1. Give or construct five examples of mathematical recursive functions.
2. List five recursive structures found in nature.
3. Why are recursive functions more expensive than iterative functions?

Exercises

1. Write an iterative Python function to generate n^{th} Pingala-Virahanka-Fibonacci numbers.
2. Write an iterative Python function to compute x_n of function $x_{n+1} = f(x_n)$ where $f(x) = ax(1-x)$. Make the a and initial x_0 as parameters of the function.
3. Using turtle module write Python functions to create the following fractal objects: Sierpinski carpet, Koch curve, Lévy fractal. See wikipedia for their description.

CHAPTER SEVEN

SOLVING COMPLEX PROBLEMS USING PYTHON

7.1 Good Programming Practice

Computational problems can be very complex. Here we list a small sample of often-encountered complex computational problems.

1. Compression of a music file.
2. Weather prediction
3. Making document writer
4. Making web browser

Writing computer programs for such applications requires major planning and team work. These problems, of course, are beyond the scope of this introductory book in which we solve much simpler problems. Yet, it is best to learn the best practices of programming during the early stages itself. In this chapter we will illustrate how to address computational problems like a good computer scientist.

In the following we list some of these tips that are helpful for writing good programs:

1. Think about the problem at hand before you start to program. In fact, *first solve the problem, and then write an algorithm, after which start to code.*
2. A computer has a fast processor, but it is dumb. It has to be told exactly what is to be done. A step-by-step process to arrive at the final solution is called an *algorithm*. It is important to write down an algorithm before we start to code. This way, we arrive at a correct and error-free code.
3. Think simple! Simple solutions are easy to code and easy to explain to colleagues. Simple codes often yield efficient results, and they are easy to modify. *KISS* is a popular phrase in computer programming; it means *Keep it simple, stupid!* When you get back to your code after a year, you should be able to understand it without strain.
4. Think carefully about extreme cases. For example, during a loop execution, we need to be cautious not to address an nonexistent array element, that is, the array index should lie in the range of 0 to *len(array)*-1.
5. Make a clean code with good choice of variables. For example, use *mass* as a variable to denote mass of the particle. Do not

- use x for mass! Your code should be readable.
- 6. Follow consistent choice of variables. For example, use i, j, k as loop variables consistently.
- 7. Comment your code, but avoid excessive commenting.
- 8. Computer programming is an art. Write beautiful codes! Of course, mastering this art takes a lot of effort, practice, and patience.

For further guidance on the art of programming, refer to the following books:

- B. W. Kernighan and R. Pike: The practice of Programming
- G. Polya and J. H. Conway: How to Solve It: A New Aspect of Mathematical Method
- R. G. Dormy: How to Solve It by Computer

7.2 prime_numbers

In this section, we focus on three problems related to prime numbers. We will illustrate how to write algorithms and then write the corresponding codes.

Test if an integer is prime or not

Many clever algorithms are available to test whether a given number is prime or not (also called *primality test*). However, in this section we will present a very simple algorithm to test if a integer n is prime or not.

Idea: If n is not a prime, then one of factors of n will be in the range 2 to \sqrt{n} . For example, 15 is divisible by 3, which is less than $\sqrt{15}$. Therefore, we test if n is divisible by integers 2, 3, ..., \sqrt{n} . As soon as we find a factor in this band, we declare n to be non-prime. Otherwise n is declared to be a prime.

Algorithm:

```
function is_prime(n):
    for i: 0 to  $\sqrt{n}$ 
        if ( $n \mod i == 0$ ):
            n is non-prime; return
    else # after the loop
        n is indivisible by a factor.
        Declare n to be prime; return
```

Code:

```
import numpy as np

# Returns 1 if n is prime, 0 otherwise  #s
def is_prime(n):

    # First test if n is divisible by 2.
    if (n%2 == 0):
        return 0

    # Test if i divides n.
    # i ranges from 3 to sqrt(n) in steps of 2.
    for i in range(3, int(np.sqrt(n))+1, 2):
        if (n%i == 0):
            return 0
```

```
    else:
        return 1
```

In the code, we save computation effort by eliminating $i = 2, 4, \dots$ (even numbers) in the first loop. The present code is more efficient than that in Section 5.3.

Generating prime numbers up to n

Here, we present the algorithm—*Sieve of Eratosthenes*—to generate prime numbers up to integer n .

Idea: The algorithm consists of following steps.

1. Construct a sieve of integers from 2 to n .
2. From the sieve eliminate numbers that are multiples of prime numbers $2, 3, 5, 7, \dots, \sqrt{n}$ one after the other.
3. At the end of the process, the remaining numbers in the sieve are the desired prime numbers.

Algorithm: Here we remove the non-prime numbers using *pop()* function.

```
function prime_sieve(n):
    Create sieve (2:n)
    prime = prime_no[0]
    loop (prime***2 < n):
        for the given prime:
            Eliminate all the multiples of prime
            up to the last number n.
            Retain other numbers.
    Pick the next prime from prime_no.
```

Code:

```
import numpy as np

# The function prime(n) returns a list of primes up to n.
# Adopts the Sieve of Eratosthenes algorithm
# Pops the non-prime numbers from the sieve ones
# a prime is able to divide it.

def prime(n):
    nos = list(range(2,n+1))
```

```

# Sieve containing integers from 2 to n

prime_index = 0
prime_now = nos[prime_index]
# First prime

while(prime_now**2 < n):

    # remove all multiples of prime_now from nos
    index = prime_index + 1

    while (index <= (np.size(nos)-1)):
        # Loop till the last number in the Sieve
        # if prime_now divides the number,
        # then delete it from the sieve.
        # Otherwise let it be,
        # move on to the next number
        if ((nos[index] % prime_now) != 0):
            index += 1
        else:
            nos.pop(index)

    # Goto the next prime number
    prime_index += 1
    prime_now = nos[prime_index]

return nos

```

Compute the prime factors of a number

Computing *prime factors* of a number is a very important problem of number theory, and it has wide applications in cryptography, statistical physics, combinatorics, etc. In the following discussion, we present one of the simplest algorithms to obtain a list of prime factors of a number.

Idea: To compute the prime factors of number n :

1. Start with 2. If n is divisible by 2, then add 2 as a factor, and set $n \rightarrow n/2$. Repeat this process until 2 is unable to divide n .
2. Repeat the same process for subsequent prime numbers: 3, 5, 7, ...
3. This process is continued till n becomes 1.

Algorithm:

```

prime_no = [2,3,5, ...]

function prime_factor(n):
    factor = [] # empty arrays

```

```
i=0
while (n > 1):
    loop:
        For prime(i),
        if (prime[i] divides n):
            n ← n/prime[i]; Append prime[i] to factor.
        increment i
    return factor
```

Code:

```
import numpy as np

# Prime numbers up to 100
prime_nos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, \
37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

# Returns prime factors of integer n
def prime_factor(n):
    factor_array = []
    i = 0
    while (n > 1):
        # loop for prime[i]
        # Inner loop for multiple factors of prime[i]
        while(n%prime_nos[i] ==0):
            n /= prime_nos[i]
            factor_array.append(prime_nos[i])

        # Go to the next prime factor
        i += 1

    return factor_array
```

With this we end our preliminary discussions on prime numbers using Python programming.

Exercises

1. Construct first 50 prime numbers.
2. Consider two integers a and b . Compute their prime factors. Using these prime factors, compute the greatest common divisor (GCD) and least common multiple (LCM) of a and b .
3. Twin prime numbers are those who differ by 2. For example, (5,7), (11,13). List all the twin prime numbers up to 500.
4. Consider the primality code discussed in this section. Rewrite the code without the `else` statement.

5. Find first 10000 prime numbers from the internet (for example, <http://mathforum.org/dr.math/faq/faq.prime.num.html>). Let us label them as P_n . Plot P_n vs. n . Also plot $P_{n+1} - P_n$ vs. n . Fit the second plot with $\log(n)$.

7.3 Searching and Sorting

Search is a very important algorithm and it is employed everywhere. Some example of search problems are

1. Searching for a person, research topic, as in google search
2. Recognition of a person in our head by searching through the stored images.
3. Search for a word in a dictionary

Many search algorithm are quite complex. Google has become world famous for their sophisticated search algorithms. Such complex schemes are beyond the scope of this book; here, we present two elementary search algorithms.

As we show below, search in sorted and unsorted arrays have different time complexity. Note that a sorted array is one in which the elements (numbers) are arranged either in increasing or decreasing order. We describe the respective algorithms below.

Searching in an unsorted array

An unsorted array has no pattern. Hence, we need to look for the element to be searched in the whole array. Algorithmically, it is prudent to start searching from the beginning of the array and continue the search operation till the element is found. This algorithm is called a *linear search*. See below the algorithm and code for a linear search for element x in array A . The function returns the array index of x . The linear search process is illustrated in [Figure 29](#).

Algorithm:

```
function linsearch(A, x):  
    Loop:  
        Starting from the first element to the last  
        element of A.  
        Test if the A[i] == x:  
            if success, report the index.  
  
    else: report that x is absent in A.
```

Code: The function returns the array index of x . It returns -1 if x

does not belong to A .

```
# Searches x in an unsorted array A.  
# Return index i, A[i]=x  
# If x does not belong to A, then return -1  
# Linear search  
  
def linsearch(A, x):  
  
    for i in range(len(A)-1):  
        if (A[i] == x):  
            return i  
  
    else: # x does not exist in A  
        return -1
```

Usage:

```
In [382]: A = ['Rohit', 'Sita', 'Ahilya', 'Laxmi']
```

```
In [383]: linsearch(A, 'Sita')  
Out[383]: 1
```

```
In [384]: linsearch(A, 'Ahilya')  
Out[384]: 2
```

```
In [385]: linsearch(A, 'Jaya')  
Out[385]: -1
```

Rohit	Sita	Ahilya	Laxmi
-------	------	--------	-------

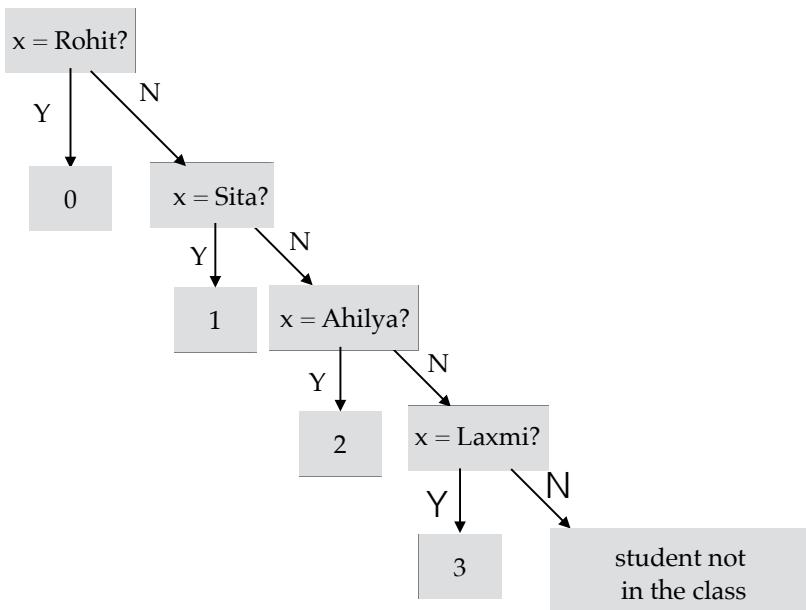


Figure 29: Linear search of an element x in a list $A = [\text{Rohit, Sita, Ahilya, Laxmi}]$. Here, Y and N represent yes and no.

How many comparisons are required in the above search operation? The best and worst scenarios require 1 and n comparisons respectively. It is easy to verify that the average number of comparison is $n/2$. Hence, the time complexity of the search algorithm is $O(n)$, where O stands for the “order of”.

Searching in an sorted array: Binary search

It is faster to search for an element in a sorted array. The algorithm makes use of the sorted nature of the array. Let us we assume that the integer array A is sorted in an ascending order.

Idea: First we locate the mid index of the array and denote it by mid . If

$x = A[mid]$, then search is complete. If $x < A[mid]$, then we search in the left half of the array, else we search in the right half of the array. The above procedure is recursively applied to the reduced arrays. We continue till we succeed. In case of failure, return -1 indicating that the element is not in the list. This procedure is called *binary search*.

Code: The function returns the array index of x . It returns -1 if x does not belong to A .

```
# Search x in an sorted array A.
# Return index i, A[i]=x
# If x does not belong to A, then return -1
# Binary search
def binsearch(A, x):
    n = len(A)
    lower = 0
    upper = n-1
    mid = (lower+upper)//2

    # if x is outside array, return -1
    if ((x < A[0]) | (x>A[-1])):
        return -1

    # If x == A[mid], exit the loop.
    # else, search left or right half depending
    # on x < A[mid] or x>A[mid].
    while (A[mid] != x):
        if (lower == upper):
            # search exhausted. x not found.
            break
        if (x<A[mid]):
            upper = mid-1
        else:
            lower = mid+1
        mid = (lower+upper)//2

    # Check with the returned mid if A[mid]==x.
    if (A[mid] == x):
        return mid
    else:
        return -1
```

Usage:

In [72]: $A = [1, 3, 7, 8, 11, 15, 20, 25, 30]$

In [73]: $\text{binsearch}(A, 11)$
Out[73]: 4

In [74]: $\text{binsearch}(A, 35)$
Out[74]: -1

In [75]: $\text{binsearch}(A, 9)$

```
Out[75]: -1
```

We illustrate the binary search described above using a diagram shown in Figure 30:. The search process starts from the top of the tree ($x = 11$), and goes down to lower branches. In the diagram, the elements are shown inside the nodes (circle), and the index range for the tree at the node ($lower, mid, upper$) is shown by its side. If $lower = upper$, then we list the *middle* at the node.

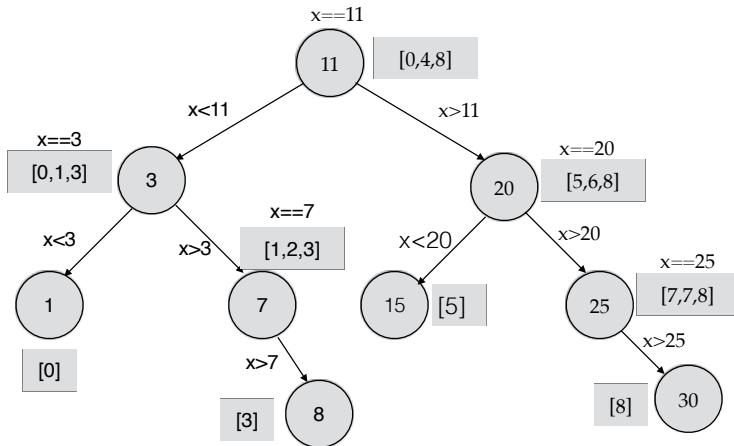


Figure 30: Illustration of binary search tree for the array $A = [1,3,7,8,11,15,20,25,30]$. The nodes contain the array elements, while the list in the side contains index range of the tree at the node.

Recursive code: A recursive code provides a more transparent implementation of binary search. However, this code is more expensive than the iterative code described above.

```
# Search x in an sorted array A between A[lower:upper].
# Return index i, A[i]=x
# If x does not belong to A, then return -1
# Recursive binary search
def binsearch_recursive(A,lower,upper,x):

    mid = (lower+upper)/2
    if ((x < A[lower]) | (x>A[upper])):
        return -1
```

```

    elif ((lower==upper) & (A[mid] != x)):
        return -1
    else:
        if (A[mid] == x):
            ans = mid
        elif (x < A[mid]):
            ans = binsearch_recursive(A,lower,mid-1,x)
        else:
            ans = binsearch_recursive(A,mid+1,upper,x)
    return ans

```

Usage:

```
In [76]: binsearch_recursive(A,0,8,11)
Out[76]: 4
```

We can compute the time complexity of binary search as follows. As illustrated in [Figure 30](#), at each stage, the data is divided into two parts. The array A of the above example has 9 elements. For this array the maximum number of comparisons ($==$, $<$, $>$) required is 5, which is maximum depth of the tree plus one. We need these many comparisons when $x = 30$.

For an array of size n , the corresponding number is approximately $(\log_2(n))$. Hence, the time complexity for binary search is $O(\log_2(n))$, which is much smaller than the $O(n)$ for a linear search. The difference becomes significant for large n . Hence, binary search is more efficient than the linear search.

The above binary search algorithm is for an integer list. However, the algorithm is equally applicable to any ordered list, for example, a language dictionary. In a dictionary, we can search a word much more quickly because the words are sorted. It would be impossible to search for a word among many unsorted words.

Sorting an array: Bubble sort

As described above, a sorted array facilitates efficient search. Hence, sorting an array is an important algorithm. Computer scientists have devised many efficient sorting algorithms. Here, we present *bubble sort*, which is one of the simplest sorting algorithms.

Idea: We first push the largest element of the array to the end of the array. For the same, we start from the first element of the array and move forward comparing the neighbours. On comparison, if $A[i] >$

$A[i-1]$, we swap the integers. As a result, the last element of A becomes the largest number.

After this, we need to push the largest number among $A[0:N-2]$ to the end of $A[0:N-2]$. We follow exactly the same procedure as above. We continue this process till we are left with a single number, which is the smallest number of the array, that stays at $A[0]$.

We implement the above idea in the following program.

Python code:

```
# Compare two neighbours: if A[i] < A[i-1], swap.
# In the first round, A[N-1] is the largest integer.
# In the next round, A[N-2] is the second largest number.
# We carry out this process till the end.

# Sort A in increasing order.
def bubble_sort(A):
    n = len(A)

    for i in range(n):

        # A[n-i-1:n-1] is sorted.
        # So go only up to n-i-2.
        for j in range(0, n-i-1):

            # Swap if the prev number is greater
            # than the last number.
            if A[j] > A[j+1]:
                A[j], A[j+1] = A[j+1], A[j]

In [76]: run bubble_sort.py
In [77]: A = [8,7,3,10,6]
In [78]: bubble_sort(A)
In [79]: A
Out[79]: [3, 6, 7, 8, 10]
```

Figure 31 illustrates the sorting procedure for array $A = [8,7,3,10,6]$. In the first iteration, the program bubbles the largest integer, which is 10. In the next iteration, the second largest integer 8 is bubbled up. This process is continued till the whole array is sorted.

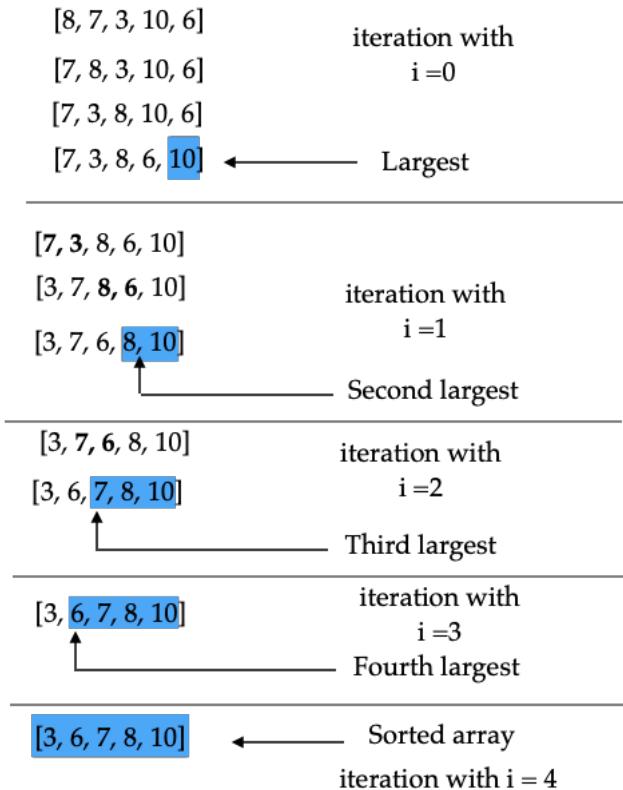


Figure 31: Sorting of array $A = [8, 7, 3, 10, 6]$ using bubble sort. The blue coloured list indicates the numbers that have been sorted.

With this we end our discussion on search and sort algorithms.

Exercises

1. Compare the complexity (number of comparisons) of linear and binary searches for $n = 16, 256, 4096, 2^{30}$.
2. How many comparisons are required to bubble sort an array of size n ?
3. Given an integer array of integers, write a Python function that prints the fourth largest integer of the array.
4. Write a Python function that sorts an array of integers in

- descending order.
5. Consider a list of strings, for example, $A = ["Python", "C", "Java", "Pascal"]$. Write a computer program to sort such a list.

CHAPTER EIGHT

PLOTTING IN PYTHON AND INPUT/OUTPUT

8.1 Plotting with Matplotlib

Experiments, observations, and computer simulations generate data. Most often, just staring at the numbers do not provide insights, but the plots do. For example, the plots of Earth's surface temperature reveal much more insights than the raw data. Similarly, business and demographic data too are presented using plots. This is the reason why most scientific and business presentations and publications contain many plots of various kinds.

Python provides extensive plotting and visualisation tools. These functions are part of *matplotlib module* of Python. In this chapter, we will cover only basic plotting features of *matplotlib*. We start with usage of *pyplot* module of *matplotlib*.

Using Pyplot and files

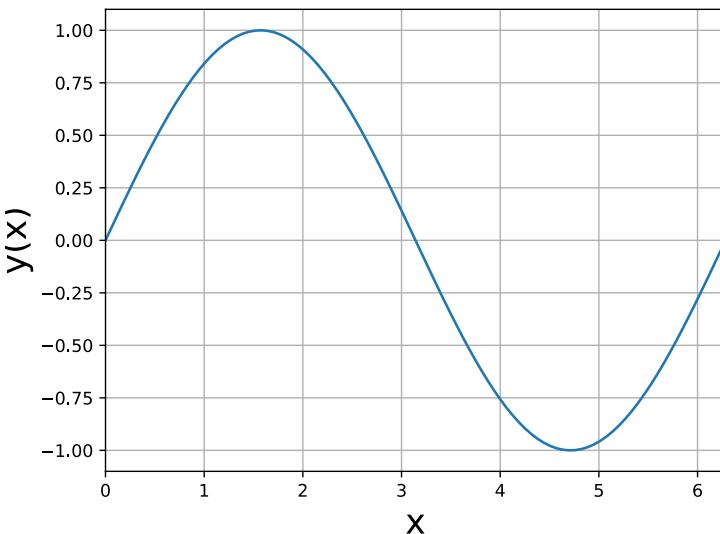
Pyplot within *matplotlib* is an important module for plotting. This module can be imported within *ipython* by typing the following in the terminal.

```
$ipython --pylab
```

We employ the following Python statements to plot $\sin(x)$ for $x = [0, 2\pi]$.

```
In [114]: x=linspace(0,2*pi,100)
In [115]: y=sin(x)
In [116]: plot(x,y); xlabel('x',size=20);
ylabel('y(x)',size=20)
Out[116]: Text(0, 0.5, 'y(x)')

In [117]: xlim([0,2*pi]); grid();
# X axis is fixed as [0,2\pi].
In [118]: savefig("test.pdf")
```



[Figure 32](#): Plot of $y=\sin(x)$.

We save the plot in `test.pdf` using the function `savefig()`. We display the plot in [Figure 32](#). Note that Python allows the plot to be saved in `png`, `jpg`, `svg`, ... formats as well.

It is much more convenient to put the axis labels, legends, etc. in a file and then run the file in *ipython*. It is easier to tinker some Python statements in the file, rather than modify the features interactively in *ipython*. In the following, we present contents of a file that plots $y = x^2$ and $y = x^3$.

```
import numpy as np
import matplotlib.pyplot as plt
from pylab import rcParams
rcParams['figure.figsize'] = 5, 3
# figure of the size 5in x 3in

x = np.linspace(-1,1,40)
y = x**2
y1 = x**3

plt.plot(x,y, 'r.', label = r'$y=x^2$')
plt.plot(x,y1, lw = 3, color = 'g', label = r'$y=x^3$')
# The labels appear in the legend.
# Helpful for identifying the curves
```

```

plt.axhline(0, color='k')
plt.axvline(0, color='k')
# Axis properties

plt.xlim(-1,1)
plt.ylim(-1,1)
# Limits of x and y axes

plt.xlabel(r'$x$', fontsize=20)
plt.ylabel(r'$y$', fontsize=20)
# Axis labels

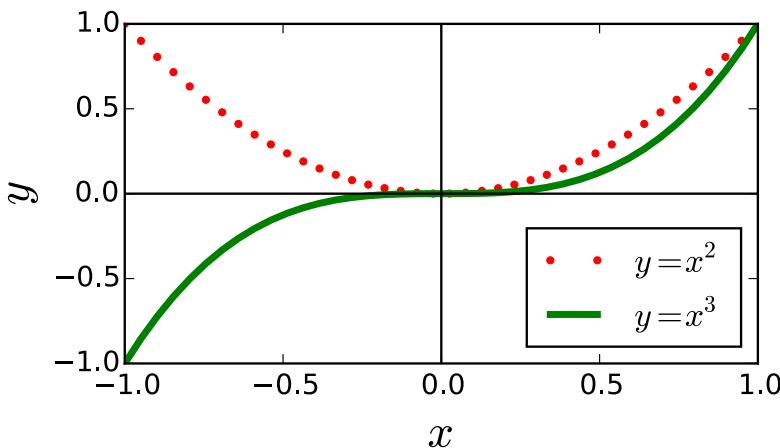
plt.xticks(np.linspace(-1, 1, 5, endpoint=True))
# Ticks on the xaxis.
# 5 points in the interval (-1,1).

plt.legend(loc=4)
# Legends appear at the fourth quadrant,
# which is the bottom right

plt.tight_layout()
# Helps in fitting plots within
# the figure cleanly

plt.savefig('plot2d.pdf')
# Save the plot in a file plot2d.pdf
plt.show()
# Show the plot on the console.

```



[Figure 33:](#) Plot of $y=x^2$ and $y=x^3$ vs. x .

The generated pdf file is shown in [Figure 33](#). In the code comments

we provide a brief description of the *matplotlib* functions, such as *legend*, *xlim*, *xlim*. In the following we make several important remarks on the plot script.

1. Key arguments of the *plot()* function are
 - a. *color*: Color of the curve. They could be blue (b), green (g), red (r), cyan (c), magenta (m), yellow (y), black (k), or white (w).
 - b. *linewidth* (*lw*): Line width of the curve in units of *points*. Note that 72 points = 1 inch.
 - c. *marker*: They could one of the following: hline ('-'), point ('.'), Circle ('o'), triangle_down ('v'), triangle up ('^'), triangle_left ('<'), triangle_right ('>'), octagon ('8'), square ('s'), pentagon ('p'), star ('*'), hexagon ('h' or 'H'), diamond ('d' or 'D'), plus ('+'), x ('x'), and more. See https://matplotlib.org/api/markers_api.html for more details.
 - d. *markersize*: Size of the above markers.
 - e. *linestyle*: solid ('-'), dashed (), chained ('.-'), dotted (':')
 - f. *label*: For the legends.
2. *plt.xticks*, *plt.yticks*: We can place the axis ticks at the desired locations.
3. *plt.show()*: Display all open figures.
4. *plt.legend(loc = x)*: Place legend at location given by x. See [Table 13](#) for the location code.

Table 13: Location code for placing legends in a plot

Location string	Location code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper centre'	9
'center'	10

Objects of matplotlib

Numerous functions and features of matplotlib can be quite confusing. It is best to understand these functions in relation to two key objects of *Matplotlib.pyplot*:

1. *Figure object*
2. *Axes object*

Figure is the top level object containing all the elements of a figure. A *Figure object* contains one or more *Axes objects* with each *axes object* representing a plot inside the figure. *Figure class declaration* is as follows:

```
class matplotlib.figure.Figure(figsize=None, dpi=None,
facecolor=None, edgecolor=None, linewidth=0.0,
frameon=None, subplotpars=None, tight_layout=None,
constrained_layout=None)
```

We create a *figure object* of size 5 inch x 5 inch using the following Python statement:

```
fig = plt.figure(figsize = (5,5))
```

A single plot or several plots are embedded inside the figure. This is done by a function *add_subplot()*. It adds an *Axes* to the figure. The function *add_subplot* can be called in several ways:

```
add_subplot(nrows, ncols, index, **kwargs)
add_subplot(pos, **kwargs)
add_subplot(ax)
add_subplot()
```

For example, in [Example 1](#), we create 4 *Axes objects* within object *fig*. Here

```
Nrows = ncols = 2
```

And the *axes objects* *ax1*, *ax2*, *ax3*, *ax4* are created using

```
ax1 = fig.add_subplot(2,2,1)
ax2 = fig.add_subplot(2,2,2)
ax3 = fig.add_subplot(2,2,3)
ax4 = fig.add_subplot(2,2,4)
```

In the figure they appear at the *top left*, *top right*, *bottom left*, and *bottom right* respectively.

Example 1:

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

fig = plt.figure(figsize = (5,5))

x = np.linspace(0,1,100)

# Generate a grid of 2x2 subplots.

# Axes plot at 1st location
ax1 = fig.add_subplot(2,2,1)
ax1.plot(x,x,label=r'$y=x$')
ax1.set_xlim(0,1)
ax1.set_ylim(0,1)
ax1.legend(loc = 'upper left')
ax1.set_ylabel(r'$y$', fontsize=20)
#ax1.xaxis.set_major_locator(plt.NullLocator())
ax1.xaxis.set_major_formatter(plt.NullFormatter())
ax1.xaxis.set_major_locator(plt.MaxNLocator(3))

# Axes plot at 3rd location
ax2 = fig.add_subplot(2,2,2)
ax2.plot(x,x**2,label=r'$y=x^2$')
ax2.set_xlim(0,1)
ax2.set_ylim(0,1)
ax2.legend(loc = 'upper left')
ax2.xaxis.set_major_formatter(plt.NullFormatter())
ax2.xaxis.set_major_locator(plt.MaxNLocator(3))
ax2.yaxis.set_major_formatter(plt.NullFormatter())
ax2.yaxis.set_major_locator(plt.MaxNLocator(6))

# Axes plot at 3rd location
ax3 = fig.add_subplot(2,2,3)
ax3.plot(x,x**3,label=r'$y=x^3$')
ax3.set_xlabel(r'$y=x^3$')
ax3.set_xlim(0,1)
ax3.set_ylim(0,1)
ax3.legend(loc = 'upper left')
ax3.set_xlabel(r'$x$', fontsize=20)
ax3.set_ylabel(r'$y$', fontsize=20)
ax3.xaxis.set_major_locator(plt.MaxNLocator(3))

# Axes plot at 4th location
ax4 = fig.add_subplot(2,2,4)
ax4.plot(x,x**4,label=r'$y=x^4$')
ax4.set_xlabel(r'$y=x^4$')
```

```

ax4.set_xlim(0,1)
ax4.set_ylim(0,1)
ax4.set_xlabel(r'$x$', fontsize=20)
ax4.legend(loc = 'upper left')
ax4.xaxis.set_major_locator(plt.MaxNLocator(3))
ax4.yaxis.set_major_formatter(plt.NullFormatter())
ax4.yaxis.set_major_locator(plt.MaxNLocator(6))

plt.savefig('plot_axes.pdf')
plt.show()

```

The above code produces

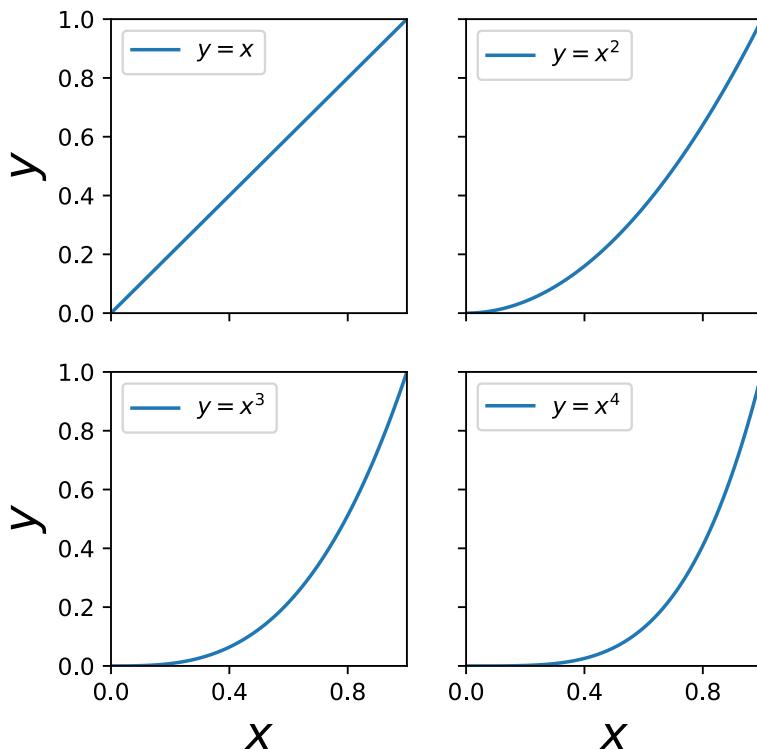


Figure 34: Plot produced by Example 1. It illustrates the usage of Axes and subplots.

An important point to note in the figure is control of the *xticks* and *yticks* in the figure.

```
ax1.xaxis.set_major_locator(plt.MaxNLocator(3))
```

```
ax2.yaxis.set_major_locator(plt.MaxNLocator(6))
```

These statements produce 3 *xticks* and 6 *yticks* respectively. Other interesting options are *NullLocator*, *LogLocator*, *AutoLocator*, *FixedLocator*. The other ticks function is Tick formatter. For example, *NullFormatter()* puts no labels on ticks. Refer to matplotlib documentation for more details.

Python offers many features for producing variety of plots. In this chapter we present some of the key features.

Visualising $z=f(x,y)$

We can visualise a function $z=f(x,y)$ in the following ways:

1. Contour plot
2. Density plot
3. Surface plot

We describe each one of them below:

Contour plot: The following code segment creates a *contour plot* of function $z = x^2 + y^2$.

```
fig = plt.figure(figsize = (3,3))
ax = fig.add_subplot(1,1,1)

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

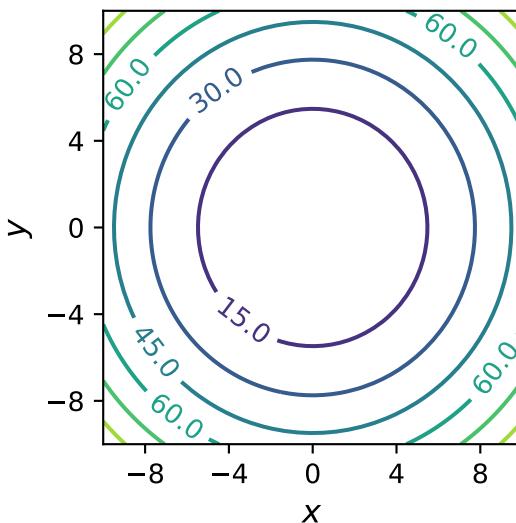
xv, yv = np.meshgrid(x,y)
z = (xv**2 + yv**2)/2

# contour plot, contours (6 contours)
curves = ax.contour(xv,yv,z, 6)
ax.clabel(curves, inline=1, fmt='%.1f', fontsize=10)
```

The function *ax.contour(xv, yv, z, 6)* produces six *contour lines*. *ax.clabel()* controls the formatting of *contour labels*. Alternatively, the function

```
curves = ax.contourf(xv,yv,z, 5, cmap = cm.Reds)
```

produces filled contours with *color map*. The resulting plot is show in [Figure 35](#).



[Figure 35:](#) Contour plot of function $z = x^2 + y^2$. The contours are labelled.

Density plot: We can also make *density plot* of the function $z = x^2 + y^2$ using function `imshow()`, `pcolor()`, and `pcolormesh()`. The function `pcolormesh()` is faster than `pcolor()`. The code segment is given below, and the plots are shown in [Figure 36](#).

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

x = np.linspace(-1,1,100)
y = np.linspace(-1,1,100)
xv, yv = np.meshgrid(x,y)
z = (xv**2 + yv**2)/2

fig = plt.figure(figsize = (6,3), tight_layout= True)
ax1 = fig.add_subplot(121, aspect='equal')

# Using imshow (image show)
c1 = ax1.imshow(z, vmin=abs(z).min(), vmax=abs(z).max(),
extent=[-1,1,-1,1])
ax1.set_title('using imshow')
ax1.set_xlabel('$x$',size=12)
```

```

ax1.set_ylabel('$y$',size=12)

divider = make_axes_locatable(ax1)
cax2 = divider.append_axes("right", size="5%", pad=0.05)
fig.colorbar(c1, cax=cax2)

# Using pcolor

ax2 = fig.add_subplot(1,2,2)
ax2.set_title('using pcolormesh')
c2 = ax2.pcolormesh(xv,yv, z)
ax2.set_xlabel('$x$',size=12)
ax2.set_aspect(aspect=1)
#Another way to set aspect ratio

```

Two important points to note:

1. The arrays `xv` and `yv` provide the `x` and `y` coordinates at the grid points. They are 2D arrays. We compute `z` at each grid point using `xv` and `yv`.
2. The colormap is resized to dimension of the plot using `make_axes_locatable()` function. There are variety of colormaps. Choice of colormap is subjective. For some advice on colormaps, refer to <https://matplotlib.org/tutorials/colors/colorbars.html> and <http://www.kennethmoreland.com/color-advice/>

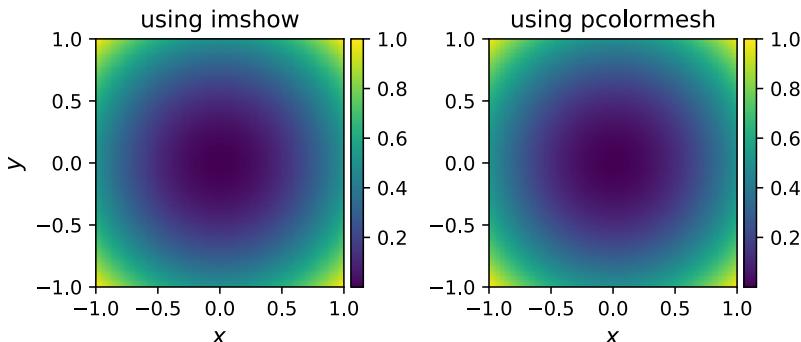


Figure 36: Density plot of function $z = x^2 + y^2$ using functions `imshow()` and `pcolormesh()`, which are in the left and right sides respectively.

Surface plot: In *surface plots*, which are three-dimensional in nature, the z coordinate specifies the value of the function. In the following code segment we present how to make *surface plot* and *wireframe plot* of the function $z = \exp(-x^2 - y^2)$. The functions used are *plot_surface()* and *plot_wireframe()*.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# import matplotlib.cm as cm
from pylab import rcParams
rcParams['figure.figsize'] = 7, 3.5

x = np.linspace(-2, 2, 100)
y = x.copy()
xv, yv = np.meshgrid(x, y)
z = np.exp(-(xv**2 + yv**2))

fig = plt.figure()
ax1 = fig.add_subplot(121, projection = '3d')
ax1.plot_surface(xv, yv, z, rstride=5, cstride=5, color = 'm')

ax2 = fig.add_subplot(122, projection = '3d')
ax2.plot_wireframe(xv,yv,z, rstride=5, cstride=5)
ax2.set_title('Wireframe')
```

In the above code, we make use of *mpl_toolkits.mplot3d.Axes3D module* to set the axis as three dimensional. This is achieved using

```
ax1 = fig.add_subplot(121, projection = '3d')
```

The surface plots are shown in [Figure 37](#).

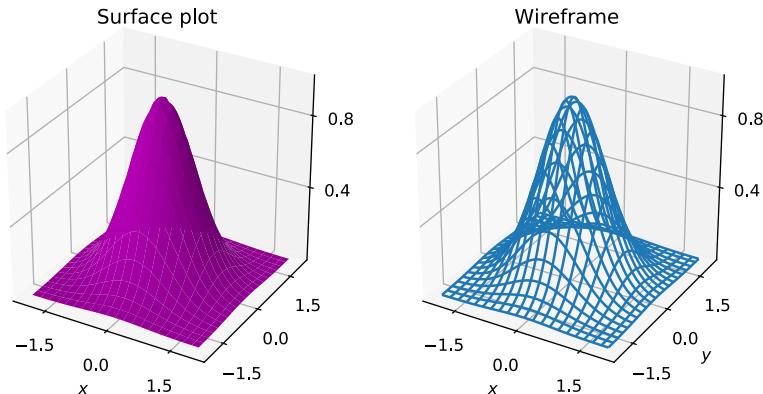


Figure 37: Surface plots of function $z = x^2 + y^2$ using functions `plot_surface()` and `wireframe()`, which are in the left and right sides respectively.

Vector plots

The following code segment produces 2D and 3D *vector plots* for the fields $\mathbf{v} = (-x, -y)$ and $\mathbf{v} = (-x, -y, -z)$ respectively. Here, we employ the function `quiver()`.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm
from pylab import rcParams
rcParams['figure.figsize'] = 7, 3.5

# Vector plot for the field v = (-x, -y)
L = 10
x = np.linspace(-L,L,10)
y = x
xv, yv = np.meshgrid(x,y)

fig = plt.figure()
ax1 = fig.add_subplot(121, aspect ='equal')
ax1.quiver(xv, yv, -xv, -yv)

#3D vector plot for the field v = (-x, -y)
ax2 = fig.add_subplot(122, projection = '3d')

x = np.linspace(-L,L,5)
y = x
z = x
xv, yv,zv = np.meshgrid(x,y,z)
```

```
ax2.quiver(xv, yv, zv, -xv, -yv, -zv)
```

The 2D and 3D *vector plots* are shown respectively in left and right sides of [Figure 38](#).

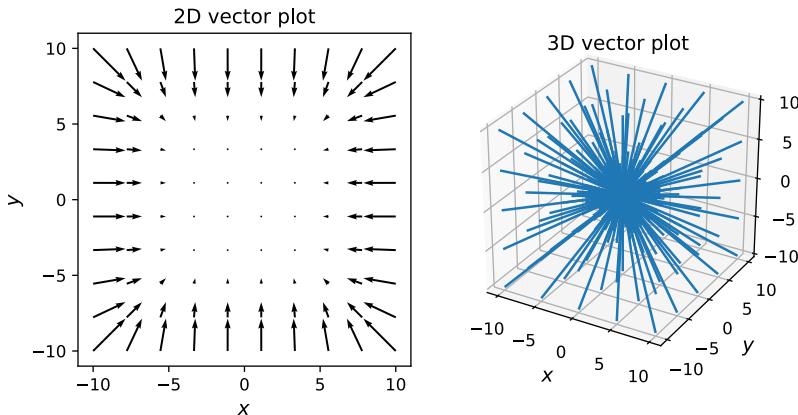


Figure 38: (Left figure) 2D vector plot of the vector field $\mathbf{v} = (-x, -y)$.
 (Right figure) 3D vector plot of the vector field $\mathbf{v} = (-x, -y, -z)$.

Plotting particle trajectory

Using Python we can easily plot 3D trajectory of a particle. The following code segment yields the helical trajectory of a particle whose coordinates are described by

$$\mathbf{r} = (x, y, z) = (b \sin(t), b \cos(t), b t),$$

Where b is a constant, and t is time that varies from 0 to 8π .

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from pylab import rcParams
rcParams['figure.figsize'] = 3,3.5

fig = plt.figure()
axes = fig.add_subplot(111, projection = '3d')
t = np.linspace(0, 8 * np.pi, 1000)
b = 1
r = b

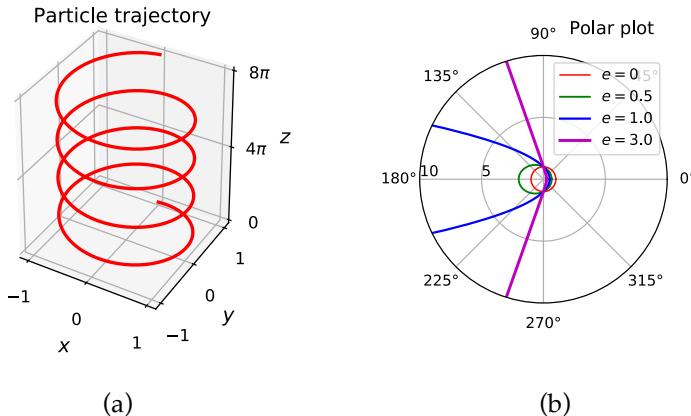
x = r * np.sin(t)
y = r * np.cos(t)
```

```

z = b*t
axes.plot(x, y, z, lw=2, color='r')

```

The generated plot is shown in [Figure 39\(a\)](#). The code employs *mpl_toolkits.mplot3d.Axes3D module* for the 3D plot.



[Figure 39](#): (a) 3D plot of a helical trajectory of a particle. (b) Radial polar plot of various conic section curves, whose eccentricities are listed in the legend.

Radial-polar plot

Using Python we can make plots in radio-polar (r - θ) coordinate system. The following code segment plots equations for conic section curves:

$$r = 1 / (1 + e \cos(\theta))$$

```

import numpy as np
import matplotlib.pyplot as plt
from pylab import rcParams
rcParams['figure.figsize'] = 4,3
phi = np.linspace(0, 2*np.pi, 100)

```

```
r1 = 1.0/(1 + 0*np.cos(phi)) #Circle
r2 = 1.0/(1 + 0.5*np.cos(phi)) #Ellipse
r3 = 1.0/(1 + 1*np.cos(phi)) #Parabola
r4 = 1.0/(1 + 3*np.cos(phi)) #Hyperbola

fig = plt.figure()
axes = fig.add_subplot(111, projection = 'polar')

axes.set_title('Polar plot')
axes.plot(phi, r1, lw=1, color='r', label=r'$e = 0$')
axes.plot(phi, r2, lw=1.25, color='g', label=r'$e = 0.5$')
axes.plot(phi, r3, lw=1.5, color='b', label=r'$e = 1.0$')
axes.plot(phi, r4, lw=2, color='m', label=r'$e = 3.0$')
plt.ylim(0, 10)
axes.set_rgrids([5,10], angle=180)
plt.legend()
```

Here, the statement,

```
axes = fig.add_subplot(111, projection = 'polar')
```

plays a key role. The *projection = 'polar'* instructs the interpreter to make a radial-polar plot. The resulting plot is shown in [Figure 39\(b\)](#).

Histogram

Python helps us plot a histogram of a given data set. The following code creates a histogram of random numbers which are Gaussian distributed with mean of $\frac{1}{2}$ and standard deviation of 1.

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

avg = 0.5 # mean of distribution
sig = 1.0 # standard deviation of distribution
x = avg + sig * np.random.randn(100000)
# generates 100000 random numbers with normal distribution

fig = plt.figure(figsize = (3.5,2.5))
num_bins = 50
n, bins = plt.hist(x, num_bins)
# Creates histogram of x using num_bins bins.
```

The above code generates 10^5 random numbers with Gaussian distribution with average of $\frac{1}{2}$ and standard deviation of 1.0. After this, the numbers are divided into 50 (*num_bins*) bins and a histogram is created, which is displayed in [Figure 40](#). The function *plt.hist()* provides following information about the histogram:

n: array of size *num_bin*; *n*[*i*] contains the number of data points in *i*th bin.

bins: array of size *num_bin*+1; it contains the edges of the bins.

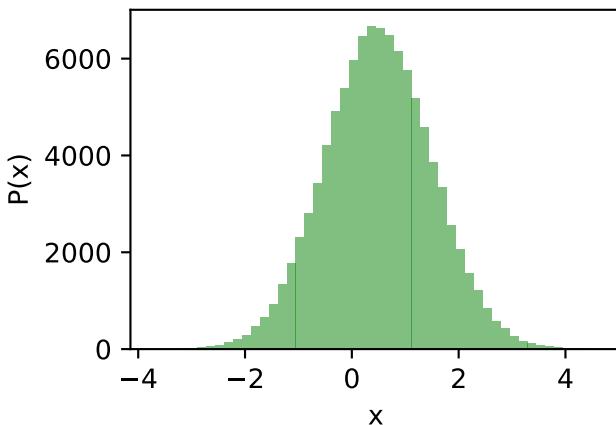


Figure 40: Histogram of random numbers which are Gaussian distributed with mean of $\frac{1}{2}$ and standard deviation of 1.

Animation

It is very easy to make animations using Python. The following code animates the function

$$f(x,t) = \exp(-(x-bt)^2 / \sigma^2)$$

where *t* is time, and *b* and σ are constants. The code is a modified version of the original code written by Jake Vanderplas (<http://jakevdp.github.com>).

```
"""
Matplotlib Animation Example

author: Jake Vanderplas
email: vanderplas@astro.washington.edu
website: http://jakevdp.github.com
license: BSD
Please feel free to use and modify this, but keep the
above information. Thanks!

```

```

"""
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

# First set up the figure, the axis, and the plot element
# we want to animate
fig = plt.figure()
ax = plt.axes(xlim=(0, 2), ylim=(-0.5, 1.5))
line, = ax.plot([], [], lw=2)

# initialization function: plot the background of each
# frame
def init():
    line.set_data([], [])
    return line,

# animation function. This is called sequentially
def animate(i):
    x = np.linspace(-2, 2, 100)
    y = np.exp(-((x-0.01*i)**2)/0.01)
    line.set_data(x, y)
    return line,

# call the animator. blit=True means only re-draw
# the parts that have changed.
# interval in milisec. There are 200 frames.
anim = animation.FuncAnimation(fig, animate,
                               init_func=init, frames=200, interval=20, blit=True)

# save the animation as an gif using
# the writer='PillowWriter')

anim.save('gaussian.gif',writer='matplotlib.animation.PillowWriter')

```

A brief explanation of the above code is as follows. The function

```
anim = animation.FuncAnimation(fig, animate,
                               init_func=init, frames=200, interval=20, blit=True)
```

animates the time-dependent curves generated by function *animate()*. The background frame is generated by *init()*. These two functions generate *line* object , which is a curve, for *matplotlib*. The output of the animation is saved in gaussian.gif file using *PillowWriter* of *matplotlib*. The animation writers that could be used are *ffmpeg*, *avconv*, etc.

With this we end this section on plotting and creating animations.

Exercises

1. Plot $\log(x^2)$ vs. x and $\cos^2 x$ vs. x .
2. Make contour, density, and surface plots for the potential of a line charge of line charge density of λ . Make appropriate scaling for the plot.
3. Plot the electric field for the above line charge.
4. Plot the potential and electric field of an electric dipole. You will need to moderate the singularity near the origin.
5. Plot the wavefunction of the ground state and first-excited state of linear oscillator.
6. The wavefunction of Hydrogen atom for $l=1$, $m = 0$ is xxx. Make a 2D contour plot of the wavefunction.
7. Generate a million random number with uniform distribution in the interval (0,4). Make a histogram of the data.
8. Animate a moving sin wave: $f(x,t) = \sin(k x - \omega t)$ for a given k and ω .
9. Consider a free particle modelled as a gaussian wave packet. Animate its motion.

8.2 *Input/Output in Python*

In scientific computing, we often encounter large datasets. For example:

- Weather data: temperature, humidity, wind velocity of Earth's atmosphere
- Wind velocity in a simulation of a moving car
- Quantum wave function of a complex molecule
- Population data of a country
- Matter density inside a star or galaxy

Such datasets are stored in the hard disk and they need to be *read* by a computer program for further processing. In reverse, we *save* or *write* the data generated by a computer program to a hard disk.

Numerical data can be stored in hard disk in various formats. Some of the popular formats are ASCII, binary, HDF, HDF5, CSV, NASACDF, etc. In this book we will describe only ASCII and HDF5 format. A brief description of some of the above formats are given below.

- *ASCII*: Here, the numbers are stored as set of characters. For example, "13.5" consists of 4 characters including the decimal point, all of whom need to be stored. In ASCII representation, we truncate the binary number to a limited precision. Hence, there is a loss of accuracy during storage. Similar loss of accuracy occurs during the data reading from the hard disk.
- *Binary*: Recall the discussion of Chapter Three in which we showed how numbers are represented in binary format, e.g., $(2.75)_{10} = (10.11)_2$. In this scheme, the binary number (e.g., the 8 bytes for float) is stored in the hard disk. Clearly, binary representation is accurate because the number is saved as is.
- *HDF5* or *Hierarchical Data Formats*: This is a complex data format for storage. HDF5 supports binary format. Refer to <https://www.hdfgroup.org/solutions/hdf5/> for more details.
- *CSV* or *Comma Separated Values*: It is a plain text format. It is popular because Microsoft Excel uses CSV format for data import/export.

Python can read/write numerical data in all the above formats. In addition, Python can also work with movie (e.g., mp4, png) and image (png, pdf) formats. In this book we will not discuss media formats. Also note that in the past we read data from the keyboard using `input()` function and wrote output to the screen using `print()` function. These are specific to standard input (keyboard) and standard output (screen).

Now we start input/output of ASCII data, which is simplest of all formats.

Input/Output for Numpy arrays in ASCII Data

Python's `numpy module` provides functions `savetxt()` to save an array into a file, and `loadtxt()` to load an array to the memory from a txt file. We illustrate these functions using the following code segment.

```
import numpy as np
# create data

month = np.linspace(1,12,12)

temperature = np.array([5.92, 9.12, 15, 20, 35, 40, 39,
34, 27, 22, 10, 9])

# Saving the arrays to text files month.txt
# and temperature.txt.
np.savetxt("month.txt", month)

np.savetxt("temperature.txt", temperature)

#Loading arrays from the txt files.
month_read = np.loadtxt("month.txt")
temperature_read = np.loadtxt('temperature.txt')
```

After `np.savetxt()`, the arrays are saved in `month_read` and `temperature_read` arrays. A couple of important points to note. We can view the saved txt files. For example, a part of file `month.txt` appears as follows:

```
In [192]: cat month.txt
1.0000000000000000e+00
2.0000000000000000e+00
3.0000000000000000e+00
4.0000000000000000e+00
```

Note that, by default, the data is saved in exponential format. We could save the data in float format with finite precision (here, up to 3

decimal places) as follows.

```
np.savetxt("month.txt", month, fmt="%0.3f", delimiter=",")
```

After this we discuss how to save and read HDF5 data.

Input/Output for HDF5 data

In the following code segment we write two arrays x and y to a HDF5 file 'data.h5'. Here we use *h5py module* of Python. The file object hf is created to write the arrays to the hard disk. The file object hf contains information about the file size, file type (integer, float, character), and the data; it is not same as the filename 'data.h5' that holds the data.

```
import numpy as np
import h5py

x = np.arange(0,2*pi,0.01)
y = np.sin(x)

# Create file object, hf, for writing
hf = h5py.File('data.h5', 'w')

#Creates dataset array_x and array_y
hf.create_dataset('array_x', data=x)
hf.create_dataset('array_y', data=y)

# close the file
hf.close()
```

As described in the comments, the function $hf.create_dataset()$ creates datasets $array_x$ and $array_y$ (for x and y arrays). These datasets are saved in the hard disk. We close the file at the end. For reading the file data.h5, we reverse the operation.

```
# Create file object hf2 for reading data.h5 file
hf2 = h5py.File('data.h5', 'r')

# keys() tells us datasets stored in the file
hf2.keys()

# Returns dataset objects array_x_read & array_y_read
# They are not numpy arrays
array_x_read = hf2.get('array_x')
array_y_read = hf2.get('array_y')

# Converts dataset objects to numpy arrays
x_read = np.array(array_x_read)
y_read = np.array(array_y_read)
```

```
#Close the file
hf2.close()
```

As described in the code, the `hf2` is created to read the datasets `array_x` and `array_y` from the hard disk. The function `hf2.keys()` tells us the datasets stored in the file, while the function `hf2.get()` reads the datasets to objects `array_x_read` and `array_y_read`. These objects are converted to numpy arrays `x_read` and `y_read` using `array()` function. We close the file at the end.

Reading Text files

Now, we briefly describe how to read/write text files, which differ from `loadtxt()` and `savetxt()` that read numerals. In the following discussion we show how read or write strings (not numerals). Python supports many access modes for the files; they are listed in [Table 14](#).

First we write strings to a file using the following set of statements.

```
f = open('data.txt', "w")
str_list = ["This is month of June. \n", "It is very hot
here. \n", "Expect rains next month. \n"]
f.write("Hi! \n")
f.writelines(str_list)

f.close()
```

[Table 14:](#) Access modes for files

argument	format	Access mode
r	text	read-only
w	text	write
a	text	append
r+	text	read and write
rb	binary	read-only
wb	binary	write
ab	binary	append
rb+	binary	read and write

The function `writelines()` writes the three strings of the list. Note that `f.write(20)` will give an error because 20 is not a string. Instead, you could use `f.write('20')`. The file `data.txt` has four lines that are separated by special character '`\n`' that stands for *newline* character.

We can view the data.txt using *cat* or *vi* commands.

```
In [40]: cat data.txt
Hi!
This is month of June.
It is very hot here.
Expect rains next month.
```

Now, we proceed to read the contents of a file, which is a large string, using *readline()* and *read()* function.

```
f = open('data.txt', "r+")
# read 2 lines of the file
print(f.readline())
print(f.readline())

#read the remaining file
print(f.read())

#To read the file from the beginning, use seek(0)
# seek(n) takes the file pointer to the nth byte from the
beginning
f.seek(0)
# The file content is a big string
str_read = f.read()

# Read all the lines of the file
for line in f:
    print(line, end='')
```

We can also use the *print* function with an additional argument, *file* = *file_name*, to send the data to the *file_name*. For example, the following statement prints variables *x* and *y* to file *f*.

```
print(x, y, file = f)
```

We can read/write binary data using similar procedure. As shown in [Table 14](#), we use options ‘wb’ and ‘rb’ to write and read binary files.

```
f = open('data.bin', 'wb')
a = [1,2,5]

# bytearray() returns a byte representation of a
bin_a = bytearray(a)
f.write(bin_a)
f.close()

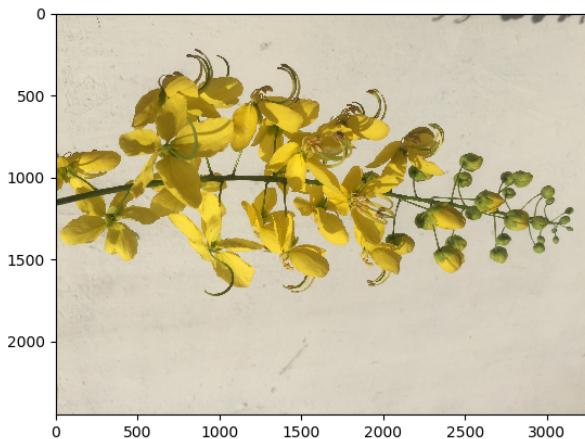
f = open('data.bin', 'rb')
a_read = list(f.read())
f.close()
```

Reading an image in Python

Often we need to import images and rework them. Here we show how to read an image in python using *mpimg* function.

```
In [65]: import matplotlib.image as mpimg  
In [69]: imag = mpimg.imread('amaltas.jpg')  
In [70]: plt.imshow(imag)
```

The function *mpimg.imread(file)* generates an array *imag(n, m, 3)* that contains the pixel map of the picture of the file. The three layers of the array provide the values of red, and green, blue components of the image. We print the image using *plt.imshow(imag)* that regenerates the original picture (see [Figure 41](#)). The pixels along the *x* and *y* axes are shown in the figure.



[Figure 41:](#) Output of *plt.imshow(im)*

With this we close our discussion on input/output from/to files.

Exercises

1. Download the yearly population data of India from worldometer.org. Read the txt data using `loadtxt()`. Plot the yearly population.
2. Download the COVID-19 data for India and several other countries from worldometer.org and plot the data.
3. Create an array of 100 random numbers. Save this array in a file in (a) ASCII format, (b) binary format, (c) HDF5 format. Read the array back from the files. Verify that you recover the original array.
4. Take picture of yourself. Read its pixel values using `matplotlib.image` library.

CHAPTER NINE

ERRORS & NONDIMENSIONALIZATION

Synopsis

Drop of water.. Napoleon

9.1 *Error Analysis*

Most numerical or computer solutions have errors due to various reason—precision of computer, measurement of data, approximate algorithm, etc. In this section we will briefly describe the origins of these errors and steps for minimize them.

An important issue in error analysis is *precision* and *accuracy*, which will be addressed first.

Precision vs. Accuracy

Precision and accuracy are often considered to be the same thing. However, they are not! Contrasting the two yields interesting insights into the error analysis, as we show below.

A marksman's aim is to hit the bullseye all the time. See [Figure 42](#). The distance of the shot from the bullseye is a measure of *inaccuracy*, while scatter of shots is a measure of *imprecision*. The shots are considered to be accurate if they are close to the centre, and precise if they are clustered together. In [Figure 42](#), case (a) corresponds to accurate and precise shooting, (b) to precise but inaccurate, (c) to accurate but imprecise, and (d) to inaccurate and imprecise. Needless to say that a marksman desires accurate and precise shots at the target.

Even though both accuracy and precision of shots depends on the quality of the gun and the skills of the marksman. However, on the whole, precision could be associated with the gun and accuracy with the skills of the marksman. In addition, precision is related to random error, while accuracy to the systematic error.

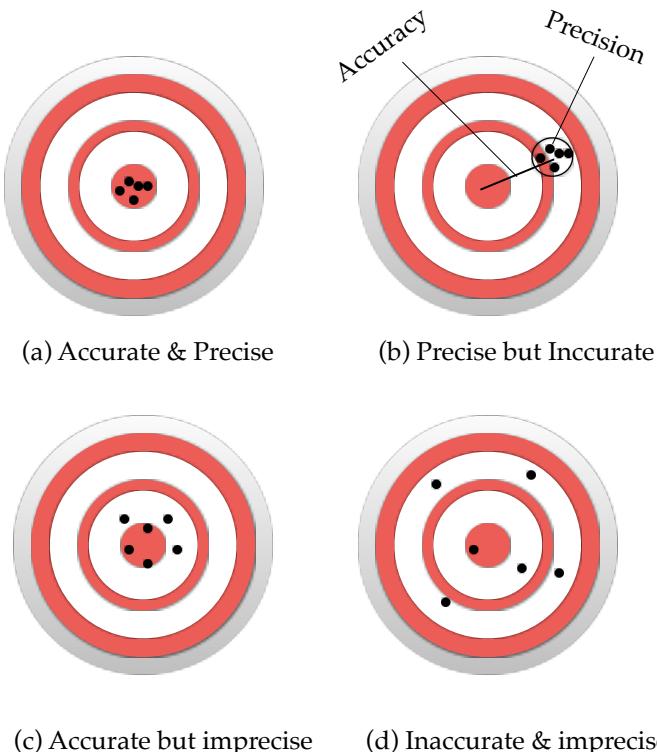


Figure 42: Illustration of accuracy and precision in target practice.
 (a) Accurate and precise; (b) precise but inaccurate; (c) accurate but imprecise; (d) inaccurate and imprecise.

Now, let us relate the above to computer solution. We take weather forecast as an example. Such forecasts are preformed using computer simulations given weather data. There are two types of errors in forecast:

- a) The temperature forecast may show systematic deviation from the real value, possibly due to some approximation in algorithm or due to some new issue that may have arisen (e.g., a sudden appearance of a tornado). This systematic error is analogous to accuracy in the target practice.
- b) The temperature forecast over a week exhibits a large scatter

from the mean temperature. This error is related to precision, and it may be due to random error in data collection and/or due to random numerical errors (e.g., truncation error).

We will illustrate similar scenario in other examples in this section.

Significant digits is a concept related to precision. The measurement using a ruler with millimeter marks is accurate only up to a millimeter. Hence, a reported measurement 10.34 cm is incorrect. For this ruler, we can trust only one digit beyond the decimal place of the measurements in units of cm. A correct measurement would be 10.3 cm, 1.9 cm, etc. The number of reliable digits in a real number is called *significant digits*. Clearly, significant digits matter on the accuracy of the concerned device and measurements. We illustrate significant digits using some examples.

- The numbers 10.3, 10.300, 010.30 have only 3 significant digits.
- 0.0019 has only two significant digits.

Before proceeding further, we describe some measures of errors:

Error: Error is defined as the difference between the actual value (V_{actual}) and the measured or computed value (V_{comp}), that is

$$\text{Absolute error} = V_{\text{actual}} - V_{\text{comp}}.$$

Absolute error: Absolute error is defined as the absolute value of the difference between the actual value (V_{actual}) and the numerical value (V_{num}), that is

$$\text{Absolute error} = |V_{\text{actual}} - V_{\text{comp}}|.$$

Relative error: Relative error is defined as the ratio of absolute error and the numerical value:

$$\text{Relative error} = |V_{\text{actual}} - V_{\text{comp}}| / |V_{\text{actual}}|$$

Relative error is a better measure of errors. This is because an error of 0.1 kg for the weight of a baby is significant, but it is not so for an elephant.

After this discussion, we categorise the errors encounter in numerical computations.

Errors in numerical solutions

Numerical errors can be categorised into the following two broad categories:

Round-off error: Round-off errors occur because most real numbers cannot be accurately represented in a computer. For example we showed in Section 3.2 that $0.8*3-2.4 \neq 0$ because 0.8 cannot be represented accurately in memory. Note that such truncation of floats occurs at almost all stages of a computation.

In addition, instruments record measurements only up to certain precision. For example, suppose the real temperature is 25.789° , but the thermometer may record it as 25.7° . The above truncation error too is called round-off error. When such measurements are fed in computer applications, e.g., weather forecast, industrial controls, the corresponding measurement errors limit the precision of numerical results.

The round-off errors are typically random, and not reproducible. That is, if we perform these computations on an another system, the result would be slightly different, but within the precision limit (here, we are ignoring the chaos theory in which errors have amplifying effects). Thus, simulation results of an ensemble of an application are expected to be scattered similar to [Figure 42](#). Such an scatter of results would be enhanced with larger round-off errors. Due to random distribution, round-off errors are presented in terms of absolute error.

Round-off errors are expect to affect the precision of a computer simulation. This is analogous to the precision in the shooting exercise discussed above (see [Figure 42](#)).

Systematic error: Most realistic numerical algorithms involve certain approximations. For example, $\sin(x)$ and $\exp(x)$ are computed using truncated Taylor series. Naturally, such series solutions have *approximation errors*, which are systematic in nature, and they are related to the accuracy of the result. This is similar to the inaccuracy of [Figure 42\(b\)](#). We will discuss the series solution in more detail later in

this section. By definition, systematic errors tend to have a definite sign, and they are reproducible. Hence, systematic errors are often presented along with sign ($V_{\text{actual}} - V_{\text{comp}}$).

Even though we broadly classify the computational errors in the above two categories, many issues in numerical computations are more complex. The numerical solution of differential equations depend quite critically on the discretization scheme, for example, time-difference Δt for ordinary differential equations. Since differential equations assume that $\Delta t \rightarrow 0$, any discretization leads to errors that could be random and/or systematic. Space discretization in PDE (partial differential equation) solvers too lead to similar issues. Error propagation is another complex issue that we will briefly sketched at the end of this section.

Note that integer operations do not have round-off or truncation errors. For example, we get a definite result for n^{th} prime number. It is a different matter that some integer problems remain unsolved. One such problem is finding the most optimum route for a travelling salesman touring n cities.

Next we discuss syntax and logical errors that are encountered during program development.

Errors during program development

The random and systematic errors occur when a computer program is already working. However, we encounter errors during program development itself. Such errors are classified in two categories:

Syntax errors: Python interpreter understand only valid Python statements. For example, statement “ $x =^ 4$ ” is syntactically (or grammatically) incorrect and it leaves the computer clueless. A computer is dumb and it will not apply *intuition* to understand incorrect statements. When a computer encounters a grammatically incorrect Python statement, it raises a flag to the programmer that the code has a *syntax error*. The programmer has to correct such statements to proceed further.

Logical error: A syntactically correct program may not yield correct results. An obvious example is an infinite loop. Or, we may have some other error in the algorithm. Such errors, called *logical errors*,

need to be fixed by the programmer.

Syntactically and logically correct programs may still not yield correct answers due to numerical errors described above. In the following discussion, we will describe general properties of numerical errors encountered in arithmetic operations and in series solution.

Errors in arithmetic operation

As described in Section Floating Point and Complex Numbers, real numbers are prone to round-off errors. Consequently most arithmetic operations (such as sum, multiplication, division) have numerical errors. We discuss the errors in subtraction and multiplication operations below.

Let us estimate the error in a subtraction operation. Imagine two floats x and y that are represented in computers as x_c and y_c , where

$$x_c = x (1 + \varepsilon_x); \quad y_c = y (1 + \varepsilon_y)$$

with ε_x and ε_y as small numbers. Even though the actual answer is $r = x - y$, computer will yield

$$r_c = x_c - y_c = x (1 + \varepsilon_x) - y (1 + \varepsilon_y).$$

Therefore, the relative error is

$$\frac{r_c}{r} = 1 + \epsilon_r = 1 + \varepsilon_x \frac{x}{r} - \varepsilon_y \frac{y}{r}$$

The error ϵ_r explodes when $r \rightarrow 0$, that is, when we subtract two nearly equal numbers. We estimate the relative error as

$$\langle \epsilon_r \rangle = \frac{x}{r} \sqrt{\langle \varepsilon_x^2 \rangle + \langle \varepsilon_y^2 \rangle}$$

where $\langle \cdot \rangle$ stands for averaging. The above analysis shows that we need to be cautious while subtracting nearly equal numbers.

Following similar procedure we can estimate the relative error in a multiplication operation. For operation $r = x y$, the relative error is

$$\frac{r_c}{r} = \frac{x_c y_c}{x y} = 1 + \epsilon_r = 1 + \epsilon_x + \epsilon_y.$$

Example 1: Let us estimate the error in numerical computation of

$$r = 1.234589111 \times 10^{-9} - 1.234589110 \times 10^{-9}.$$

The exact answer is 10^{-18} . However, according to Python,

```
In [89]: 1.234589111e-9-1.234589110e-9
Out[89]: 1.0000001492112815e-18

In [98]: 1.0000001492112815e-18/1e-18 - 1
Out[98]: 1.492112815526525e-07
```

Clearly, $\epsilon_r = 1.492112815526525 \times 10^{-7}$, which is significant considering $r = 10^{-18}$. The error would be worse for 32-bit float operation.

A question is why worry about such small numbers. It turns out that we encounter such small numbers in physics. The atomic and nuclear sizes are in nano (10^{-9}) and femto meters (10^{-15}) respectively. In the next section, we will present strategies on how to avoid very large or very small numbers.

Example 2: The roots of a quadratic equation

$$a x^2 + b x + c = 0$$

Are

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For small c , $\sqrt{b^2 - 4ac}$ is close to $|b|$. Hence, one of the solutions is close to zero, which is vulnerable to round-off errors, as described above. The finite-value solution is

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \text{ for } b > 0, \text{ and}$$

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ for } b < 0.$$

It is best to estimate the small-value solution x_2 using the formula $x_1 x_2 = c/a$ that leads to $x_2 = c/(a x_1)$. In particular for $a = b = 1$, and $c = 10^{-4}$,

```
In [100]: (-1 -sqrt(1-4*1e-4))/2      #x1
Out[100]: -0.9998999899979994
In [103]: x2
Out[103]: -0.00010001000200050015
```

Thus, $x_1 = -0.9998999899979994$ and $x_2 = -0.00010001000200050015$. We verify that $x_1 + x_2 = -1 = -b/a$, thus the computed roots are quite accurate.

Example 3: For large x , $\sqrt{x+1} - \sqrt{x} \rightarrow 0$. It is best to evaluate the above using

$$\sqrt{x+1} - \sqrt{x} = \frac{1}{\sqrt{x+1} + \sqrt{x}}$$

For $x = 10^{18}$, $\sqrt{x+1} - \sqrt{x} = 0$, but the correct answer is

$$\frac{1}{\sqrt{x+1} + \sqrt{x}} = 5 \times 10^{-10}.$$

Errors in series expansion

Approximate solutions can be improved using better approximations. For example, we compute $\cos(x)$ and $\exp(x)$ using the following series approximations:

$$\begin{aligned}\cos(x) &= \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \dots \\ \exp(x) &= \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots\end{aligned}$$

In the following code, we compute $\cos(x)$ and $\exp(x)$ for $x = 1$ using

series solution with n terms, with n ranging from 1 to 19.

```

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

plt.tick_params(axis='both', which='minor', labelsize=10)

plt.figure(figsize = (3,2))

N = 20
x = 1

# for exp(x)
print ("Exp(x) expansion for x=1")

##
error_expx = []

sum = 0
for n in range(0,N):
    sum += x**n/math.factorial(n)
    error_expx.append(np.exp(x)-sum)
    print (n, sum, np.exp(x)-sum, x**(n+1)/
math.factorial(n+1))

x_axis = np.arange(1,N+1,1)
error_expx = np.array(error_expx)

plt.semilogy(x_axis, abs(error_expx), 'g.-', label='error
in $\exp(x)$')

# for cos(x)
print ("cos(x) expansion for x=1")

error_cosx = []

sum = 0
for n in range(0,N):
    sum += (-1)**n * x**(2*n)/math.factorial(2*n)
    error_cosx.append(np.cos(x)-sum)
    print (n, sum, np.cos(x)-sum, x**(2*n+2)/
math.factorial(2*n+2))

x_axis = np.arange(1,N+1,1)
error_cosx = np.array(error_cosx)

plt.semilogy(x_axis, abs(error_cosx), 'b.-', lw = 2,
label='error in $\cos(x)$')

plt.xticks([1,4,8,12,16,20])

plt.xlabel(r'$n$', fontsize=10)
plt.ylabel('Errors', fontsize=10)

```

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

We compute the numerical error for each n and plot them. As shown in Figure 43, $\cos(x)$ converges to the actual answer for $n = 8$ with error around 1.11×10^{-16} . For $\exp(x)$, similar accuracy is achieved for $n = 17$. This is because $\exp(x)$ converges slower than $\cos(x)$.

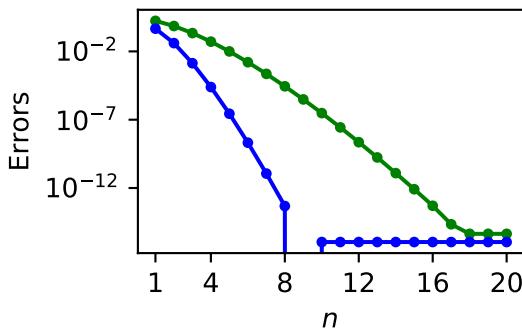


Figure 43: Errors in series solution for $\cos(x)$ (blue curve) and $\exp(x)$ (green curve) using various terms of the series.

The above series solutions are one of the simplest approximate solutions. Ramanujam came up with the following amazing series solution for π that yields correction answer up to 7 significant digits for three terms itself. (reference: wikipedia.org).

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}.$$

In later parts of the book, we will discuss various approximate solutions for computing derivatives, integrals, roots of equations etc.

In Figure 44 we illustrate how better approximations take us closer to the actual answer. A marksman improves his/her accuracy with better techniques, while better algorithms improve the accuracy of numerical solutions.

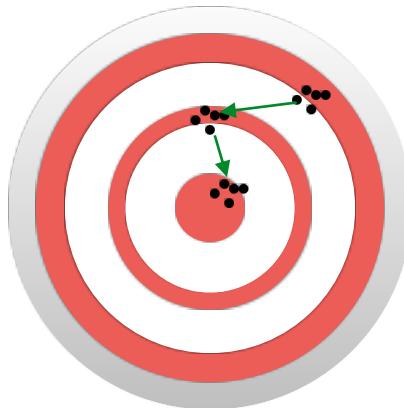


Figure 44: A schematic diagram illustrating how approximate solutions successively approach the actual answer with better approximations.

Propagation of numerical error

We illustrate propagation of numerical error using the example of weather prediction. We run the weather code so as to forecast weather for five to seven days. Propagation of uncertainty in such simulations is a complex issue and it is a hot topic of research. Yet, we know from experience that the forecast gets worse for longer duration. This is because errors tend to add up. Simplistically, in a series of computation, the error could add up in two ways:

In a random manner: The errors in every step is random and uncorrelated, hence the total error after N steps will be approximately $\epsilon\sqrt{N}$, where the error in each step is ϵ .

In a systematic manner: Systematic error add up algebraically. Here, the total error after N steps is approximated as ϵN , where ϵ is error in each step.

Another important issue in multistep numerical scheme is stability of the solution. We will discuss stability of numerical solution in XXX.

According to chaos theory, in chaotic systems (which are many), the errors grow exponentially with time. With this, most numerical solutions would become useless due to the propagation of round-off errors. However, large weather simulations and other applications yield meaningful results most of the time. This contradiction is unresolved and it is a topic of present-day research. Among many factors, multiscale interactions may possibly be suppressing the errors to some degrees. It is known that in multiscale systems, the energy cascade from large scales to small scales lead to an effective dissipation. It is possible that such dissipation may be responsible for the suppression of errors at small scales.

With this we close our discussion on error analysis.

Conceptual questions

1. Contrast round-off error and systematic error. Give examples.
2. Series solutions have numerical errors. Which category among round-off error and systematic error do they belong to?

Exercises

1. Compute the round off error for the following operation: $1.1*8$, $1/3*7$, $1.5*2$, $0.2*9$.
2. What are the significant digits for the following real numbers: 0.0019 , 0.00190 , 01.980 ?
3. Using Python code, find roots of quadratic equation $a x^2 + b x + c = 0$ for $a = 1$, $b = 2000.00001$, and $c=0.002$. Employ 32-bit float variables. Compare your numerical result with the exact result.
4. Compute $\cos(x)$ ($x=1.0$) using a series solution by taking various number of terms and compute the error. Compare the results with those for $\sin(x)$ discussed in this chapter.
5. Compute $\exp(-x)$ ($x=1.0$) using a series solution and compute the error. Compare the results with those for $\exp(x)$ discussed in this chapter.

9.2 Nondimensionalization of equations

Very small or very large numbers are prone to severe round-off errors. Also, it is not very convenient to work with very large or small numbers. To circumvent such problems, atomic physicists, nuclear physicists, and astrophysicists employ respectively Angstrom (\AA) (10^{-10} meter), fermi (10^{-15} meter), and lightyear as units of length.

Similarly convenient units are chosen for time, velocity, energy etc. For details on choice of appropriate scales, I refer the reader to Appendix B of my book, Introduction to Mechanics.

In this section, we go a step further. Using appropriate scales, we nondimensionalize several equations. For numerical simulations, it is best to work with *nondimensionalized equations* due to the following reasons:

1. Many parameters of the original equations are eliminated in the nondimensionalized equations. For example, Planck's constant (6.63×10^{-34} Joule second), which is a tiny number, does not appear in the nondimensionalized Schrödinger's equation of Hydrogen atom (to be discussed below). The parameters of the nondimensionalized equations are of the order of unity.
2. A nondimensionalized equation has a much fewer parameters than its dimensional counterpart. Hence we need fewer simulations (also experiments) to get to solve the equation. Using these techniques aeronautical engineers perform experiments on smaller prototypes of aircrafts and deduce the flow behaviour of the real aircraft.
3. The variable range of nondimensionalized equations is of the order of unity. Thus, we avoid small or large numbers and thus minimise round-off errors.
4. We understand the system better in terms of nondimensionalized equations. Also, it is easier to present the relevant numbers. For example, we state that the size of the Hydrogen atom is 0.53\AA , rather than $0.53 \times 10^{-10} \text{ m}$.

We illustrate the nondimensionalization procedure using the following examples.

Nondimensionalization of projectile motion with air drag

We can solve the motion of a projectile of mass m moving under gravity (acceleration due to gravity = g) and viscous damping force of $\gamma\mathbf{v}$, where \mathbf{v} is the velocity of the projectile (see Verma, Introduction to Mechanics, Section 9.1.2). For initial velocity $(v_x(0), v_y(0))$, the solution $(x(t), y(t))$ is

$$x(t) = \frac{m v_x(0)}{\gamma} \left[1 - \exp\left(-\frac{\gamma}{m}t\right) \right]$$

$$y(t) = -\frac{mg t}{\gamma} + \frac{m}{\gamma} \left[\frac{mg}{\gamma} + v_y(0) \right] \left[1 - \exp\left(-\frac{\gamma}{m}t\right) \right]$$

The above solution looks quite complex with so many parameters. We can nondimensionalize this solution using the following transformation. Here the time scale is v_y/g , while the length scales along x and y directions are $v_x(0)v_y(0)/g$ and $[v_y(0)]^2/g$ respectively.

$$t' = t / [v_y(0)/g]$$

$$x' = x / \left[v_x(0)v_y(0)/g \right]$$

$$x' = x / \left[v_x(0)v_y(0)/g \right]$$

Substitution of the above yields the following solution

$$x' = \frac{1}{\alpha} \left[1 - \exp(-\alpha t') \right]$$

$$y' = -\frac{t'}{\alpha} + \frac{1}{\alpha} \left[\frac{1}{\alpha} + 1 \right] \left[1 - \exp(-\alpha t') \right]$$

where $\alpha = \gamma v_y(0)/(mg)$. We can analyse the above solution by varying α . The trajectories for various α 's are exhibited in [Figure 45](#). Note that we can easily convert (x', y', t') to (x, y, t) using the above transformations and compute the real trajectories.

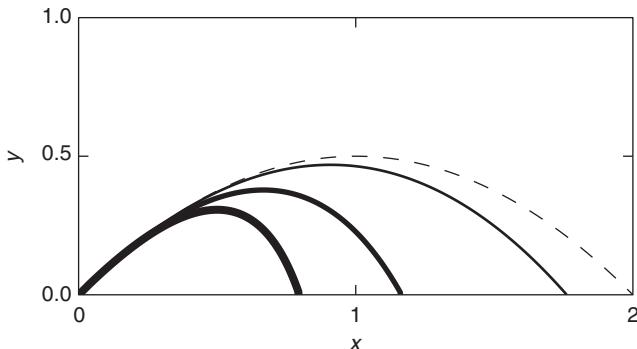


Figure 45: Trajectories of projectile for various values of $\alpha = \gamma v_y(0)/(mg)$. $\alpha=0$: drag-less (dashed line), while $\alpha = 0.1, 0.5$, and 1 (curves with increasing thickness).

The nondimensionalized solution has many advantages, namely,

1. The original equation has four parameters: $v_x(0)$, $v_y(0)$, g , and γ , but the nondimensionalized version has only one. Thus, we have reduced the complexity tremendously without sacrificing anything.
2. The nondimensionalized solution is much cleaner and can be plotted very easily. It is less prone to errors.
3. In the nondimensionalized version we avoid small or large numbers. For example, the range of the projectile could be in tens of kilometres, but x' and y' range from 0 to 2.

Nondimensionalization of equation for classical oscillator

The equation of a forced oscillator of mass m and spring constant k is

$$m \frac{d^2x}{dt^2} + k x = F_0 \cos(\omega_f t)$$

where F_0 and ω_f are the force amplitude and frequency. Note that the above equation has four parameters: m , k , F_0 and ω_f . The spring constant k is related to the natural frequency ω_0 as $k = m \omega_0^2$. Hence, the oscillator equation can be rewritten as

$$\frac{d^2x}{dt^2} + \omega_0^2 x = \frac{F_0}{m} \cos(\omega_f t)$$

The oscillator has a natural time scale, which is ω_0^{-1} . We nondimensionalize the above equation using this time scale:

$$t = t'/\omega_0, \dots \quad (5)$$

where t' is the nondimensional time. We rewrite the oscillator equation in terms of x and t' that yields

$$\frac{d^2x}{dt'^2} + x = \frac{F_0}{m\omega_0^2} \cos\left(\frac{\omega_f}{\omega_0} t'\right) \dots (6)$$

The above equation provides us a length scale, $F_0/(m\omega_0^2)$, using which we nondimensional x :

$$x = x' \left(\frac{F_0}{m\omega_0^2} \right) \dots (7)$$

Substitution of above x in Eq. (5) yields

$$\frac{d^2x'}{dt'^2} + x' = \cos\left(\frac{\omega_f}{\omega_0} t'\right) \dots (8)$$

The above equation, a nondimensionalized version of the original oscillator equation, has only one parameter, ω_f/ω_0 , which is a significant reduction from the four parameters the original equation has.

In computer, we solve Eq. (8) for various values of ω_f/ω_0 , including extreme values, say 0.001, 0.1, 1, 10, 1000, and study the properties of the oscillator. We do not need to solve the original equation by varying four parameters. Note that we can always get $x(t)$ using Eqs. (5, 7).

Nondimensionalization of Schrödinger's equation for Hydrogen atom

Schrödinger's equation for Hydrogen atom in CGS units is

$$-\frac{\hbar^2}{2m_e} \nabla^2 \psi - \frac{e^2}{r} \psi = E \psi$$

Where ψ is the wavefunction, \hbar is Planck's constant, m_e is the reduced mass of the proton and electron, e is the charge of the electron, and E is the energy of the system. Using dimensional analysis we can deduce the following length scale, r_a , and energy scale, E_a , of the system:

$$r_a = \frac{\hbar^2}{m_e e^2}; \quad E_a = \frac{e^2}{r_a}$$

Note that r_a is the Bohr radius. We nondimensionalize Schrödinger's equation using the above scales, that is,

$$r = r' r_a; \quad E = E' E_a$$

Substitution of the above in Schrödinger's equation yields

$$-\frac{\hbar^2}{2m_e r_a^2} \nabla'^2 \psi - \frac{e^2}{r_a r'} \psi = E' \frac{e^2}{r_a} \psi$$

or

$$-\frac{1}{2} \nabla'^2 \psi - \frac{1}{r'} \psi = E' \psi$$

which is the nondimensionalized Schrödinger's equation, a simpler version of the original equation. Some of the nondimensionalized wave functions are

$$\begin{aligned} \psi_{1,0,0} &= \frac{1}{\sqrt{\pi}} \exp(-r') \\ \psi_{2,0,0} &= \frac{1}{\sqrt{32\pi}} (2 - r') \exp(-r'/2) \\ \psi_{2,1,0} &= \frac{1}{\sqrt{32\pi}} r' \exp(-r'/2) \cos \theta \\ \psi_{2,1,\pm 1} &= \frac{1}{\sqrt{64\pi}} r' \exp(-r'/2) \sin \theta \exp(\pm i\phi) \end{aligned}$$

Note that the normalisation condition in terms of nondimensionalized variables is $\int d\mathbf{r}' |\psi|^2 = 1$.

Nondimensionalization of Schrödinger's equation for linear oscillator

The Schrödinger's equation for a linear oscillator of mass m and frequency ω is

$$-\frac{\hbar^2}{2m} \nabla^2 \psi + \frac{1}{2} m \omega^2 x^2 \psi = E \psi$$

Using $\sqrt{\hbar/(m\omega)}$ and $\hbar\omega$ as length and energy scales respectively, we derive the nondimensional equation as the following:

$$-\frac{1}{2} \nabla^2 \psi - \frac{1}{2} x^2 \psi = E' \psi$$

The nondimensional stationary wavefunctions for the oscillator are

$$\begin{aligned}\psi_0 &= \frac{1}{\sqrt{\pi}} \exp(-x'^2/2) \\ \psi_1 &= \frac{1}{\sqrt{\pi}} \sqrt{2}y \exp(-x'^2/2) \\ \psi_2 &= \frac{1}{\sqrt{\pi}} \frac{1}{\sqrt{2}} (2x'^2 - 1) \exp(-x'^2/2) \\ \psi_3 &= \frac{1}{\pi^{1/4}} \frac{1}{\sqrt{3}} (2x'^3 - 3x') \exp(-x'^2/2)\end{aligned}$$

Clearly, nondimensionalization simplifies the equations and corresponding solution. This is in addition to the other benefits, such as reduction in number of parameters, variables of the order of unity, etc.

Conceptual questions

1. What are the benefits of nondimensionalization of equations? How does it help computer simulations?

Exercises

1. What are the length scales for the quantum oscillator? Nondimensionalize the time-independent Schrödinger's equation for the oscillator. Also, nondimensionalize the wavefunction. Compute the $\langle x \rangle$, $\langle x^2 \rangle$, and $\langle E \rangle$ for the ground and first excited states.
2. Using the nondimensionalized wavefunctions, compute the $\langle r \rangle$, $\langle r^2 \rangle$, and $\langle E \rangle$ for ψ_{100} , ψ_{200} , ψ_{210} , and ψ_{211} states.
3. The equation of a damped oscillator is
$$m \frac{d^2x}{dt^2} + 2\gamma \frac{dx}{dt} + kx = F_0 \cos(\omega_f t)$$
where γ is the damping coefficient. What are the time scales of this equation? Nondimensionalize the above equation using one of the time scales.

PART-II

Numerical Methods

“Numerical computations can yield exact result with clever algorithms.”

OVERVIEW

Flow chart

CHAPTER TEN
INTERPOLATION

10.1 Lagrange Interpolation

In experiments and simulations, we record values of a function at finite number of points. For further processing we construct a smooth function that passes through these points. A process of construction of such a function is called *interpolation*. We illustrate this process using several example.

- Temperature is recorded on the surface of at discrete locations for weather predictions. Using interpolation we construct a temperature function to find temperature at intermediate points.
- TV picture. gaming. GPU

In addition, interpolation forms a basis for many numerical algorithms: integration, differentiation, differential equation solver, etc. This observation will become evident when we discuss these schemes in future.

In Figure 46(a), we exhibit an extrapolated curve that passes through the data points (x_i, y_i) , which are called *knots*. For a smooth interpolation curve, the uncertainty in the values of the data points need to minimal. However, if the data points have significant spread due to random errors, the we employ regression analysis to find best-fit curves through the data points. See Figure 46(b) for an example. Regression analysis will be discussed in XXX.

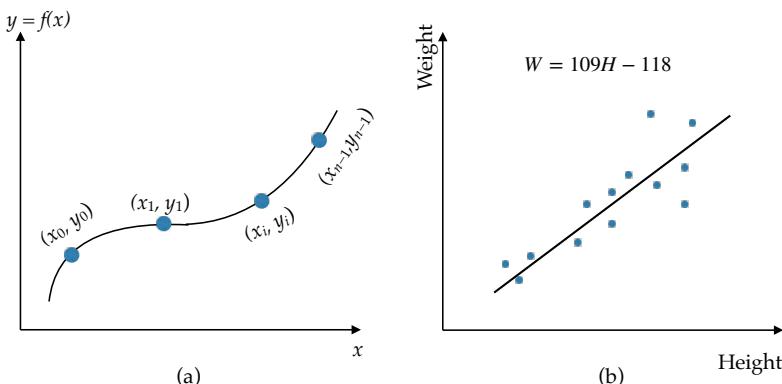


Figure 46: (a) Interpolated function $f(x)$ passes through the data

points. (b) The best-fit curve through a set of points.

There are many interpolation schemes, Lagrange interpolation, Hermite interpolation, divided difference, spline etc. We start with 1D Lagrange interpolation.

Lagrange Interpolation in 1D

Figure 46(a) Illustrates 1D interpolation through a set of data points. Lagrange constructed a polynomial that passes through the data points. For n data points, the degree of polynomial is $n-1$.

It is simplest to describe the linear linear interpolation through two points, (x_0, y_0) and (x_1, y_1) , of the function $f(x)$. Note that $f(x_0) = y_0$ and $f(x_1) = y_1$. See [Figure 47](#) for an illustration.

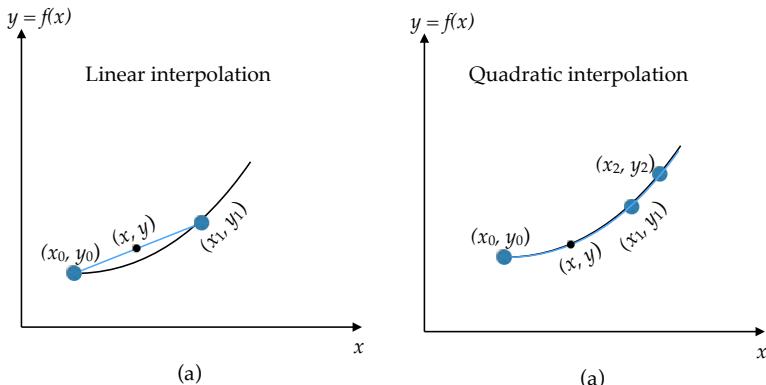


Figure 47: (a) Linear interpolation through (x_0, y_0) and (x_1, y_1) of the curve. (b) Quadratic interpolation through (x_0, y_0) , (x_1, y_1) , and (x_2, y_2) .

It is straightforward to derive the linear interpolating function $P_2(x)$ passing through the two points as

$$P_2(x) = \frac{x - x_1}{x_0 - x_1} y_0 + \frac{x - x_0}{x_1 - x_0} y_1 = L_0(x)y_0 + L_1(x)y_1$$

where $L_0(x)$ and $L_1(x)$ are linear functions only of x . Note that $L_0(x_0) = 1$; $L_0(x_1) = 0$; $L_1(x_1) = 1$; $L_1(x_0) = 0$. Note that in general $P(x) \neq f(x)$. However, they are equal when $f(x)$ is a linear function of x .

Now we generalise the above to n data points: $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$.. Lagrange constructs the following interpolating function passing through these points:

$$P_n(x) = \sum_{j=0}^{n-1} L_j(x)y_j$$

where

$$L_j(x) = \prod_{i,i \neq j} \frac{(x - x_i)}{(x_j - x_i)}$$

Note that $L_j(x_k) = \delta_{jk}$. The interpolation function $P_n(x)$ is a $(n-1)^{\text{th}}$ degree polynomial. For $n = 2$, $P_2(x)$ is the formula for the linear interpolation described above, while for $n = 3$, $P_3(x)$ is quadratic interpolation of $f(x)$:

$$\begin{aligned} P_3(x) &= \frac{x - x_1}{x_0 - x_1} \frac{x - x_2}{x_0 - x_2} y_0 + \frac{x - x_0}{x_1 - x_0} \frac{x - x_2}{x_1 - x_2} y_1 + \frac{x - x_1}{x_2 - x_1} \frac{x - x_1}{x_2 - x_1} y_2 \\ &= L_0(x)y_0 + L_1(x)y_1 + L_2(x)y_2 \end{aligned}$$

In general, the above $P(x)$ is only an approximation of $f(x)$, but they are the same function if $f(x)$ is quadratic. See [Figure 47\(b\)](#) for an illustration.

Like any numerical computation, it is important to compute the error, $f(x) - P(x)$, for Lagrange interpolation. Using reasonably complex arguments, it has been shown that the error for $P_n(x)$ is

$$E_n(x) = f(x) - P_n(x) = \frac{f^{(n)}(\zeta)}{n!} \prod (x - x_i) \quad \dots(9)$$

where ζ is an intermediate point. The proof of the above statement is somewhat technical, hence we detail it in [Appendix A: Error in Lagrange Interpolation](#). Note that if $f(x)$ is a polynomial of degree $(n-1)$, then $f^{(n)}(\zeta) = 0$, and hence $f(x) - P_n(x) = 0$. Thus, the Lagrange interpolation function is same $f(x)$.

The derivation $E(x)$ tells about the existence ζ , but its computation is quite complex. Hence, error estimation too is quite involved. For practical purposes, we can only estimate a bound on the error.

The following Python code computes $P(x)$ given that $xarray$ and $yarray$ contain x_i and y_i respectively. After that we also compute the product $\prod (x - x_i)$ for the estimation of error.

```
# P(x) = Pi_i (x-x_i)/(x_j-x_i) y_i
# returns value at x
def Lagrange_interpolate(xarray,yarray,x):
    n = len(xarray)
```

```

ans = 0
for j in range(n):
    numr = 1; denr = 1;
    for i in range(n):
        if (j != i):
            numr *= (x-xarray[i])
            denr *= (xarray[j]-xarray[i])
    ans += (numr/denr)*yarray[j]
return ans

```

Usage:

```

xarray = np.array([3,4])
yarray = np.array([1/3.0,1/4.0])
P2 = Lagrange_interpolate(xarray,yarray,x)

```

Example 1: Given $f(x) = 1/x$, we construct interpolating Lagrange polynomials using 2, 3, and 4 points. Using the function `Lagrange_interpolate()` and the two points $(2,1/2)$ and $(5,1/5)$ we construct a linear $P(x)$ illustrated in [Figure 48\(a\)](#). After this we employ two sets of three points, $\{(2,1/2), (4,1/4), (5,1/5)\}$ and $\{(2,1/2), (3,1/3), (5,1/5)\}$, to construct quadratic interpolating functions, which are illustrated in [Figure 48\(b,c\)](#) respectively. In the end, we use 4 points $\{(2,1/2), (3,1/3), (4,1/4), (5,1/5)\}$ to derive a $P_4(x)$ illustrated in [Figure 48\(d\)](#).

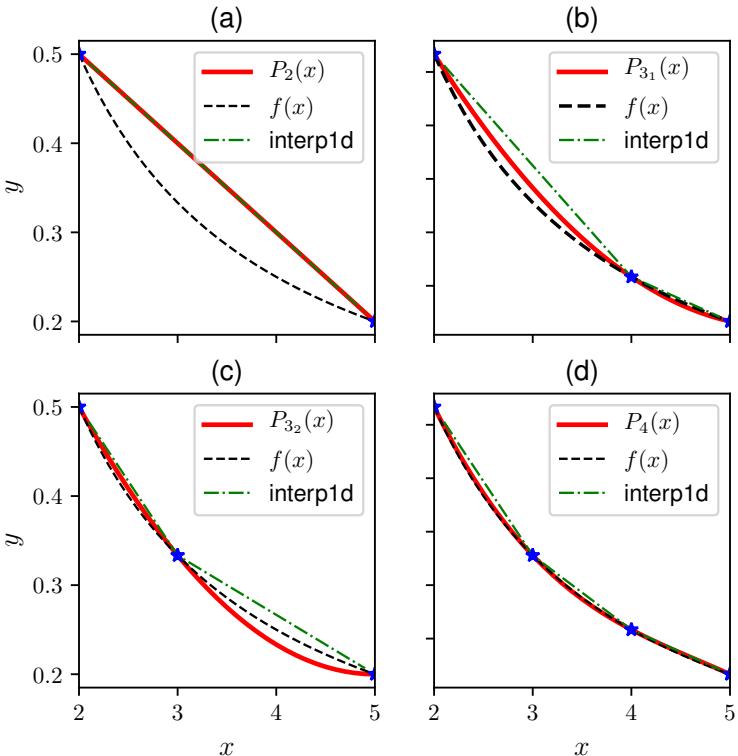


Figure 48: (a) Linear interpolation using 2 points. (b,c) Quadratic interpolation using three points. (d) Interpolation using 4 points. The blue stars represent the data points, the actual function $f(x)$ is shown using black dashed lines, the polynomials using red solid line, and interpolation function using Python’s *interp1d* using green dashed-chained line.

Example 2: Error analysis of the data of Example 1. We plot the error $E(x) = f(x) - P(x)$ in [Figure 49](#). Note that $E(x) = 0$ at the knots. As expected, error decreases with the increase of n .

We estimate the upper bound on $E_2(x) = f(x) - P_2(x)$ as follows. Since

$$E_2(x) = \frac{f^{(n)}(\zeta)}{2}(x-2)(x-5)$$

$E_2(x)$ is maximum for somewhere in the middle of the interval.

The product $(x-2)(x-5)$ takes maximum value of 2.25 at $x = 3.5$. In addition, $\max(f^{(2)}(\zeta)) = 1/4$ at $\zeta = 2$. Therefore,

$$|E_2(x)| < \frac{1}{8} \times 2.25 = 0.28125$$

which is the case, as shown in Figure 49. We can perform similar error estimation for other polynomials.

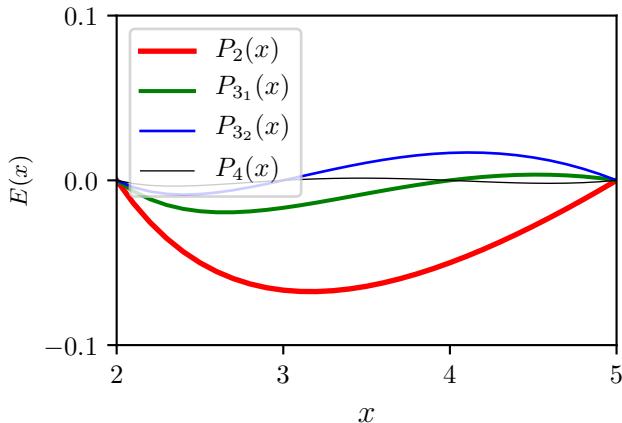


Figure 49: The plots of errors $E(x) = f(x) - P(x)$ for various Lagrange polynomials passing through the data of Example 1.

In addition, we also compute error at a specific point, say at $x = 3.5$, which is the middle point in $[2,5]$. The errors $E_n(3.5) = 1/3.5 - P_n(3.5)$ for $P_2(x)$ to $P_4(x)$ are -0.06429 , -0.00804 , 0.01071 , and 0.00134 respectively. Here too the error decreases with increase of n .

Lagrange interpolation in using *scipy's interp1d*

Python's has function *interp1d* within the *scipy* module. A Python code for the same is given below:

```
from scipy import interpolate

xarray = np.array([2,5])
yarray = np.array([1/2.0,1/5.0])
f = interpolate.interp1d(xarray,yarray)
# f contains the interpolation function.

xinter = np.arange(2.1,5,.1)
```

```
P2_scipy = f(xinter)
# P2_scipy is an array containing interpolated values for
xinter array.
```

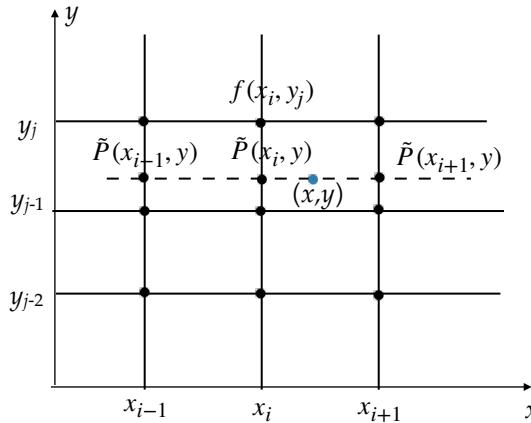
Here, f contains the interpolation function. We use f to generate interpolated values at x coordinates stored in *winter* array. The interpolated values are in *P2_scipy* array. A point to note that the range of *xinter* must be within extreme abscissa values of the data.

In [Figure 48](#), we exhibit the interpolated function obtained using *interp1d*. Note that *interp1d* produces piece-wise linear functions.

Lagrange interpolation in 2D

The Lagrange interpolation scheme described for 1D curves can be easily extended to two dimensions (2D). Two-dimensional interpolation is very useful for image processing.

Assume Cartesian 2D mesh with points as (x_i, y_i) . Consider an arbitrary point (x, y) where we wish to interpolate the function $f(x, y)$. We denote the interpolating function as $\tilde{f}(x, y)$. See Fig [Figure 50](#) for an illustration:



[Figure 50](#): Interpolation in a 2D mesh.

We estimate $f(x, y)$ using x_i interpolation first.

$$P(x, y) = \sum_i \prod_{i', i' \neq i} \frac{(x - x_{i'})}{(x_i - x_{i'})} \tilde{P}(x_i, y)$$

Now we proceed to estimate $\tilde{P}(x_i, y)$ as

$$\tilde{P}(x_i, y) = \sum_j \prod_{j', j' \neq j} \frac{(y - y_{j'})}{(y_j - y_{j'})} f(x_i, y_j)$$

substitution of which in $P(x, y)$ yields

$$\begin{aligned} P(x, y) &= \sum_i \prod_{i', i' \neq i} \frac{(x - x_{i'})}{(x_i - x_{i'})} f(x_i, y) \\ &= \sum_i \sum_j \prod_{i', i' \neq i} \frac{(x - x_{i'})}{(x_i - x_{i'})} \prod_{j', j' \neq j} \frac{(y - y_{j'})}{(y_j - y_{j'})} f(x_i, y_j) \end{aligned}$$

Hence

$$P(x, y) = \sum_i \sum_j L_{i,j}(x, y) y_{i,j}$$

Where

$$L_{i,j} = \prod_{i', i' \neq i} \prod_{j', j' \neq j} \frac{(x - x_{i'})}{(x_i - x_{i'})} \frac{(y - y_{j'})}{(y_j - y_{j'})}$$

Similarly, interpolating polynomials to 3D functions can be derived as

$$P(x, y, z) = \sum_i \sum_j L_{i,j,k}(x, y, z) y_{i,j,k}$$

Where

$$L_{i,j,k} = \prod_{i', i' \neq i} \prod_{j', j' \neq j} \prod_{k', k' \neq k} \frac{(x - x_{i'})}{(x_i - x_{i'})} \frac{(y - y_{j'})}{(y_j - y_{j'})} \frac{(z - z_{k'})}{(z_k - z_{k'})}$$

Python function interp2d

In summary, the accuracy of Lagrange interpolation increases with more points. However, too many points produce higher-order polynomials that induce spurious oscillations. Also, the computational

complexity increases for larger number of points. Thus, we need to use optimum number of points for interpolation. In case of large number of points, piece-wise interpolation can be employed.

There are several other polynomial-based interpolation schemes, such as Hermite interpolation and Newton's divided difference. We will not discuss them here due to lack of space and time. Rather, we move to a different interpolation scheme called splines, which is topic of the next section.

Conceptual questions

1. List examples of scientific applications where interpolation is used.
2. Series solutions have numerical errors. Which category among round-off error and systematic error do they belong to?

Exercises

1. For $f(x) = \sin(x)$ with $x = [0, \pi/2]$, construct Lagrange interpolating polynomials using 2, 3, and 4 points. Plot these functions, as well as the errors for them. Repeat the exercise when the interval is $x = [0, \pi]$.
2. For $f(x) = \exp(x)$ with $x = [0, 2]$, construct Lagrange interpolating polynomials using 2, 3, and 4 points. Analyse the errors associated with them.
3. Consider a 2D function $f(x,y) = \exp(x+y)$ in the domain $x = [0,1]$ and $y = [0,1]$. Construct Lagrange interpolating polynomials using 2, 3, and 4 points along each directions. Make contour plots for the interpolating polynomials along with the original function.
- 4.

10.2 Splines

For large number of points, we need to employ piece-wise interpolation. However, the interpolating function is not smooth at the knots. For example, the piece-wise linear function generated by `interp1d` has kinks at the knots (see [Figure 48](#)). Splines, to be discussed in this section, help us achieve smoothen the interpolating functions.

In this section we will discuss cubic splines, which are solutions of the beam equation:

$$EI \frac{d^4}{dx^4} f(x) = F(x) \quad (1)$$

where E is the Young's modulus of the material, I is the second moment of the beam's cross-section, and $F(x)$ is the applied force. In the spline framework, the force is assumed to be active at the nodes, which are x_i 's with $i=0:(n-1)$. See [Fig](#) for an illustration. The equation for such a beam is

$$EI \frac{d^4}{dx^4} f(x) = F_i \delta(x - x_i) \dots \quad (10)$$

Integration of the above equation around yields $f''(x)$ with discontinuities around the nodes, as shown in [Figure 51](#). Integration of $f''(x)$ yields $f'(x)$ with a kink. Subsequent integration yields smooth curves for $f(x)$ and $f'(x)$.

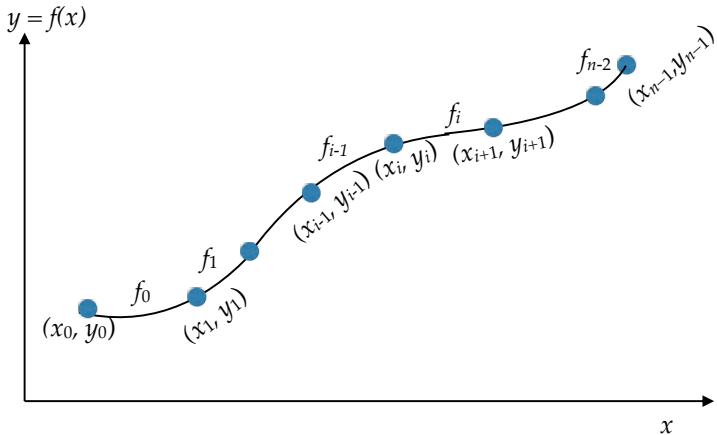


Figure 51: Illustration of a spline with nodes as (x_i, y_i) and the piece-wise functions as $f_i(x)$.

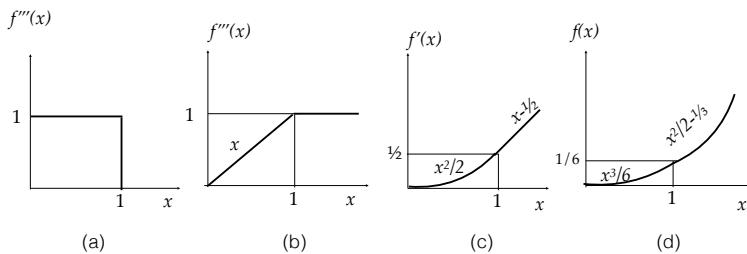


Figure 52: Function $f(x)$ and its derivatives for a spline.

Our objective is find smooth $f(x)$ given (x_i, y_i) at the nodes. As shown in [Figure 52](#), $f(x)$ and its derivatives satisfy the following properties:

1. The second derivative, $f''(x)$, is piecewise linear in each interval (x_i, x_{i+1}) .
2. The function $f(x)$ and its first derivative are continuous and smooth everywhere, including at the nodes.
3. The function $f(x)$ passes through (x_i, y_i) .

Following property (1), the linear interpolation of $f'(x)$ yields

$$f_i''(x) = f''(x_i) \frac{x_{i+1} - x}{x_{i+1} - x_i} + f''(x_{i+1}) \frac{x - x_i}{x_{i+1} - x_i}$$

Double integration of the above equation yields $f(x)$ in the interval (x_i, x_{i+1}) as

$$\begin{aligned} f_i(x) &= f''(x_i) \frac{(x_{i+1} - x)^3}{6h_i} + f''(x_{i+1}) \frac{(x - x_i)^3}{6h_i} \\ &+ \left[\frac{y_i}{h_i} - \frac{h_i}{6} f''(x_i) \right] (x_{i+1} - x) + \left[\frac{y_{i+1}}{h_i} - \frac{h_i}{6} f''(x_{i+1}) \right] (x - x_i) \end{aligned}$$

where $h_i = x_{i+1} - x_i$. The two constants of integration have determined using the conditions: $f_i(x_i) = y_i$ and $f_{i+1}(x_{i+1}) = y_{i+1}$. Similarly, for the interval (x_{i-1}, x_i) , the function is

$$\begin{aligned} f_{i-1}(x) &= f''(x_{i-1}) \frac{(x_i - x)^3}{6h_{i-1}} + f''(x_i) \frac{(x - x_{i-1})^3}{6h_{i-1}} \\ &+ \left[\frac{y_{i-1}}{h_{i-1}} - \frac{h_{i-1}}{6} f''(x_{i-1}) \right] (x_i - x) + \left[\frac{y_i}{h_{i+1}} - \frac{h_{i-1}}{6} f''(x_i) \right] (x - x_{i-1}) \end{aligned}$$

Now we impose an additional condition that the first derivative at the nodes is continuous at both sides. Applying this condition to the node at $x = x_i$, i.e., $f_i(x_i) = y_i$ and $f_{i+1}(x_{i+1}) = y_{i+1}$, we obtain

$$\begin{aligned} &-f''(x_i) \frac{h_i}{2} - \left[\frac{y_i}{h_i} - \frac{h_i}{6} f''(x_i) \right] + \left[\frac{y_{i+1}}{h_i} - \frac{h_i}{6} f''(x_{i+1}) \right] \\ &= -f''(x_i) \frac{h_{i-1}}{2} - \left[\frac{y_{i-1}}{h_{i-1}} - \frac{h_{i-1}}{6} f''(x_{i-1}) \right] + \left[\frac{y_i}{h_{i-1}} - \frac{h_{i-1}}{6} f''(x_i) \right] \end{aligned}$$

which yields

$$\frac{h_{i-1}}{6} f''(x_{i-1}) + \frac{1}{3} (h_i + h_{i-1}) f''(x_i) + \frac{h_i}{6} f''(x_{i+1})$$

$$= \frac{y_{i+1}}{h_i} - y_i \left(\frac{1}{h_i} + \frac{1}{h_{i-1}} \right) + \frac{y_{i-1}}{h_{i-1}}$$

We obtain $(n-2)$ linear equations for nodes at $i = 1 \dots (n-2)$. However we have n unknowns [$f''(x_i)$ for $i = 0 \dots (n-1)$]. To solve this problem, we use one of the following boundary conditions:

1. *Parabolic run-out*: We assume that $f''(x)$ are constant on both end intervals, i.e., $f''(x_0) = f''(x_1)$ and $f''(x_{n-1}) = f''(x_{n-2})$. Hence $f(x)$ is quadratic in these intervals. With the above two conditions, we have n equations to determine the n unknowns.
2. *Free end*: We assume that $f'' = 0$ at both the ends, or $f''(x_0) = f''(x_{n-1}) = 0$. With this constraint, we can determine $(n-2)$ $f''(x_i)$'s using $(n-2)$ equations.
3. *Cantilever end*: An intermediate condition between the cases 1 and 2, i.e., $f''(x_0) = \lambda f''(x_0)$ and $f''(x_{n-1}) = \lambda f''(x_{n-2})$ with $0 \leq \lambda \leq 1$. As in case (1), we have n equations to determine the n unknowns.
4. *Periodic spline*: We assume that the data is periodic with y_0 identified with y_{n-1} , or $y_0 = y_{n-1}$. With this we have $(n-1)$ points, $(n-1)$ matching conditions, and $(n-1)f''(x_i)$ to determine.

With one of the above four boundary conditions, we have equal number of unknown and linear equations. These equations can be solved using matrix methods that will be discussed in Section 18.1. In the following, we take a simple case with four points that can be solved analytically.

Example 1: We work out the splines for the data of Example 1 discussed in Section 10.1. We consider the same four points $\{(2, 1/2), (3, 1/3), (4, 1/4), (5, 1/5)\}$ as before. Let us use the free-end boundary condition for this example. Note that $h_i = 1$. Therefore $f''_0 = f''_3 = 0$, and

$$\begin{aligned} \frac{2}{3}f''_1 + \frac{1}{6}f''_2 &= y_2 - 2y_1 + y_0 = \frac{1}{4} - \frac{2}{3} + \frac{1}{2} = \frac{1}{12} \\ \frac{1}{6}f''_1 + \frac{2}{3}f''_2 &= y_3 - 2y_2 + y_1 = \frac{1}{5} - \frac{2}{4} + \frac{1}{3} = \frac{1}{30} \end{aligned}$$

The solution of the above equations are $f''_1 = 3/25$ and $f''_2 = 1/50$. Using these values and $f''_0 = f''_3 = 0$, we construct the piece-wise function and plot it. The plot is shown in [Figure 53](#). Python's `scipy` function `interpolate.splrep()` computes the spline parameters (`tak`). We can

compute the value of the curve by supplying x array to $\text{interpolate.splev}(x, tck)$. The following Python code segment computes the splines using free-end boundary condition (as described above) and using scipy's splrep and splev functions.

```
### x-y data
xd = np.array([2,3,4,5])
yd = np.array([1/2.0,1/3.0,1/4.0,1/5.0])
### double derivatives
fdd = np.array([0,3/25,1/50,0])

def f(i,x):
    first_term = (fdd[i]*(xd[i+1]-x)**3 + fdd[i+1]*(x-
    xd[i])**3)/6
    second_term = (yd[i]-fdd[i]/6)*(xd[i+1]-x) + (yd[i+1]-
    fdd[i+1]/6)*(x-xd[i])
    return first_term + second_term

# range of x
x0r = np.arange(2,3.1,0.1)
x1r = np.arange(3,4.1,0.1)
x2r = np.arange(4,5.1,0.1)
y0r = f(0,x0r);
y1r = f(1,x1r);
y2r = f(2,x2r);

xtot = np.concatenate((x0r, x1r, x2r))
ytot = np.concatenate((y0r, y1r, y2r))

# Using spline, output in tck
# Using tck, splev
tck = interpolate.splrep(xd,yd)
x=np.arange(2,5.1,0.1)
y = interpolate.splev(x, tck)
```

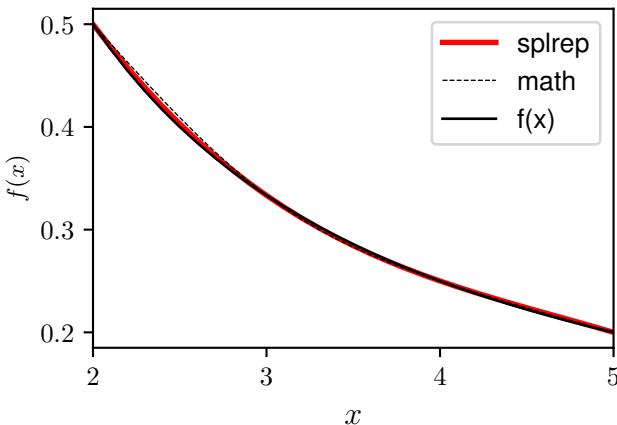


Figure 53: For the data $\{(2, 1/2), (3, 1/3), (4, 1/4), (5, 1/5)\}$, the plots of splines computed using (a) free-end boundary condition (black dashed curve) and (b) scipy's `splrep` and `splev` functions. The actual function $f(x) = 1/x$ is shown as solid black curve. All the curves are close to each other. See [Figure 54](#) for errors.

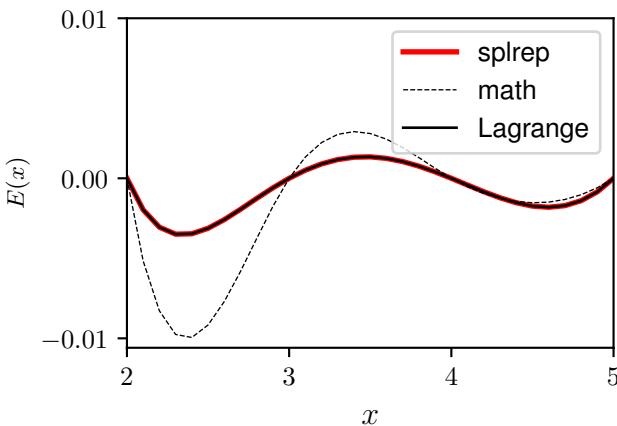


Figure 54: The plots of [Figure 53](#), error $E(x) = f(x) - P(x)$ for spline calculated mathematically (black dashed curve), spline computed by `splrep` and `splev` functions (red curve), and Lagrange polynomial with 4 points (solid black curve).

Even though the splines are quite close to the actual function, there are small errors ($E(x) = f(x) - P(x)$), which are plotted in [Figure 54](#). The accuracy of Lagrange polynomial and spline computed using `splrep` are comparable. However, the error of splines computed using free-end boundary condition is quite significant. This is because $f''(x_0) = 1/4$, not zero as is assumed in free-end boundary condition. The error at the other end, $x=5$, is less because $f''(x_3) = 2/25$, that could be considered close to zero.

Other forms of splines are cubic splines, B-Splines, etc. In these splines, the forms of the polynomials differ from the ones discussed in this section.

In summary, splines provide smooth interpolating functions. For large number of points, determination of splines requires solution of matrix equation. Fortunately, the matrix is tridiagonal that can be easily solved using the method that will be described in [Section 18.1](#).

In this chapter we assumed that the sampling points (x_i, y_i) have zero noise. In the presence of random noise, regression analysis are used to determine the underlying function $f(x)$. We will discuss regression in [Regression](#).

Conceptual questions

1. In what ways spline interpolation is better than Lagrange interpolation.

Exercises

1. Rework spline for Example 1 with parabolic run-out boundary condition. Use linear equation solver of Python.
2. Construct periodic spline for $f(x) = \sin(x)$ with $x = [0, 2\pi]$ using appropriate number of points. Rework the spline with parabolic run-out boundary condition.
3. For $f(x) = \exp(x)$ with $x = [0, 2]$, construct spline with free end boundary condition of $f''(x)$.

CHAPTER ELEVEN
NUMERICAL INTEGRATION

11.1 Newton-Cotes Formulas

Integration, loosely speaking, summing up of a function in an interval, is encountered in all streams of science and engineering. For example, the velocity of a particle is computed by integrating acceleration, which is force divided by mass. In turn, an integration of the velocity yields the position of the particle. For these reasons, *numerical integration* is an important topic of computational science.

Numerical integration, also called *quadrature*, of a function $f(x)$ from $x=a$ to $x=b$ is written as

$$I = \int_a^b f(x) dx = \sum_{i=0}^{n-1} w_i f(x_i) \dots (11)$$

where w_i 's are the weights and x_i 's are abscissas. The integration schemes can be classified into two broad categories:

Simple methods: Here we choose x_i as evenly spaced n points and then compute w_i . If error is beyond admissible limit, the number of points is increased.

Complex methods: Here x_i and w_i are chosen in such a way the integral has minimum error.

Newton-Cotes Formulas

Newton-Cotes scheme belongs to class of simple methods. Here we divide the interval (a,b) into m equal divisions with $\Delta x = (a-b)/m = h$. The abscissas are located at $x_j = a+j h$, where $j = 0:m$. Note that the number of points $n = m+1$. Here, m denotes the degree of Newton-Cotes scheme.

We approximate the function $f(x)$ using the Lagrange interpolation:

$$P_n(x) = \sum_{j=0}^{n-1} L_j(x) f(x_j)$$

which is substituted in Eq. (11) that yields

$$I = \int_b^a f(x) dx \approx \int_b^a P_n(x) dx = \sum_{j=0}^{n-1} f(x_j) \int_a^b L_j(x) dx = h \sum_{j=0}^m C_j^{(m)} f(x_j)$$

... (12)

where

$$C_j^{(m)} = \frac{1}{h} \int_b^a L_j(x) dx \dots (13)$$

Since $L_j(x)$'s are independent of the data $f(x_j)$, we conclude that $C_j^{(m)}$'s are independent of $f(x_j)$. For $f(x) = 1$, Eq. (12) yields $I = (b-a)$. Hence, we conclude that $\sum_j C_j^{(m)} = m$. Note that weights $w_j = h C_j^{(m)}$.

Now let us compute the $C_j^{(m)}$ for some of the Legendre polynomials discussed in Section 10.1. For $n = 2$ (two points), the polynomial is

$$P_2(x) = \frac{x-b}{a-b} f(a) + \frac{x-a}{b-a} f(b)$$

An integration of the above yields

$$I = \int_b^a P_2(x) dx = \frac{h}{2} (f(a) + f(b)).$$

This method, called *trapezoid rule*, is accurate up to machine precision for linear functions. For example, $\frac{1}{3}$ will still be approximated to a finite number of digits. See [Figure 55\(a\)](#) for an illustration. The area between $f(x)$ and the blue straight line is the error in the integral. Thus, $C_0^{(1)} = C_1^{(1)} = \frac{1}{2}$.

For the quadratic Lagrange polynomial, $P_3(x)$ with abscissas at $x=a, (a+b)/2, b$, we obtain

$$I = \int_b^a P_3(x) dx = \frac{b-a}{6} (f(a) + 4f((a+b)/2) + f(b))$$

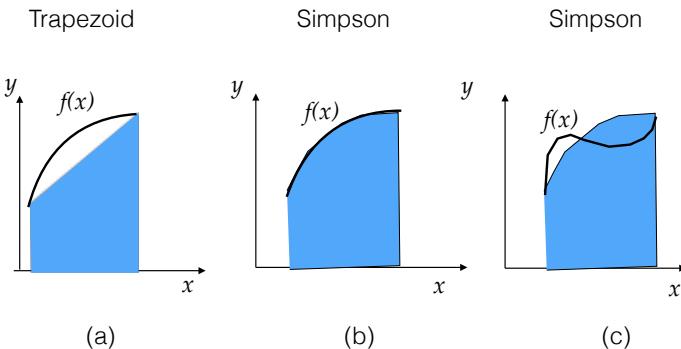


Figure 55: (a) Trapezoid rule. (b,c) Simpson rule. The numerical integral is represented by the blue coloured region.

Hence, $C_0^{(2)} = C_2^{(2)} = \frac{1}{3}$ and $C_1^{(2)} = \frac{4}{3}$. This method, called *Simpson's rule*, is accurate for polynomials up to third-degree polynomials, as shown in Figure 55(b,c). In figure(c), for a cubic $f(x)$, the filled and unfilled areas cancel each other. The coefficients $C_j^{(m)}$ for larger m are derived similarly using higher-order Lagrange polynomials. These coefficients are listed in Table 15.

Table 15: Coefficients for the Newton-Cotes integration schemes.

m	$C_j^{(m)}$	$C_j^{(m)}$	$C_j^{(m)}$	$C_j^{(m)}$	$C_j^{(m)}$	$C_j^{(m)}$
1	$\frac{1}{2}$	$\frac{1}{2}$				
2	$\frac{1}{3}$	$\frac{4}{3}$	$\frac{1}{3}$			
3	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{3}{8}$		
4	$\frac{14}{45}$	$\frac{64}{45}$	$\frac{8}{15}$	$\frac{64}{45}$	$\frac{14}{45}$	
5	$\frac{95}{288}$	$\frac{125}{96}$	$\frac{125}{144}$	$\frac{125}{144}$	$\frac{125}{96}$	$\frac{95}{288}$

We remark that Newton-Cotes scheme with $n=4$ is also called *Simpson's 3/8 rule*.

Error analysis for Newton-Cote's method

In 10.1 we derived that the error in Lagrange interpolation with n

points is

$$E_n(x) = f(x) - P_n(x) = \frac{f^{(n)}(\zeta)}{n!} \prod (x - x_i)$$

Hence the error in the integration by the Newton-Cotes scheme with n points is

$$E_n = \int_a^b E_n(x) dx = \frac{f^{(n)}(\zeta)}{n!} \int_a^b dx \prod (x - x_i) \dots (14)$$

The above integral is proportional to h^{n+1} for even n , but it vanishes for odd n . Note however that the error is nonzero for odd n unless $f(x)$ is a polynomial of degree n or lower. Hence, to estimate error for this case, we invoke the next-order Lagrange polynomial that passes through the points $(a, a+h, a+2h, \dots, b, b+h)$, but integrate $E_n(x)$ in the interval $[a, b]$. Hence, a more accurate estimate of the error for odd n is

$$E_n = \int_a^b E_n(x) dx = \frac{f^{(n+1)}(\zeta)}{(n+1)!} \int_a^b dx \left[(x - b - h) \prod (x - x_i) \right] \dots \quad (15)$$

The errors for various Lagrange polynomials are listed in [Table 16](#). We can get the coefficients and the prefactors for the errors using Python's `scipy.integrate.newton_cotes(m, 1)`, where m is the degree of the scheme. The argument 1 indicates that the abscissas are equally spaced. The function returns a tuple whose first element is a list of coefficient, while the second argument contains the prefactor for the error.

[Table 16:](#) Errors for the Newton-Cotes integration

m	Error
1	$(h^3/12)f^{(2)}(\zeta)$
2	$(h^5/90)f^{(4)}(\zeta)$
3	$(3h^5/80)f^{(4)}(\zeta)$
4	$(8h^7/945)f^{(6)}(\zeta)$
5	$(275h^7/12096)f^{(6)}(\zeta)$

6

$$(9h^9/1400)f^{(8)}(\zeta)$$

We illustrate the above estimation for $n = 3$ with $a = 0$, $b=1$, and $h=\frac{1}{2}$. For these parameters, Eq. (14) yields

$$E_3 = \frac{f^{(3)}(\zeta)}{3!} \int_0^1 dx x(x - 1/2)(x - 1) = 0$$

because the integrand is an odd function around $x = \frac{1}{2}$. However, an application of Eq. (15) with knots at $x= 0, 1/2, 1$, and $3/2$ yields the following error estimate:

$$E_3 = \frac{f^{(4)}(\zeta)}{4!} \int_0^1 dx x(x - 1/2)(x - 1)(x - 3/2) = \frac{h^5}{90} f^{(4)}(\zeta) = \frac{1}{2880} f^{(4)}(\zeta)$$

Since $f^{(4)}(\zeta)=0$ for a third-order polynomial, Simpson's method is accurate for polynomials up to cubic order. Since $f^{(n)}(\zeta)=0$ for n^{th} -order polynomial, we can conclude that Newton-Cotes scheme with even n points provides accurate integrals for polynomials of degree $(n-1)$ or lower. However, for odd n points, the integral is accurate for polynomials of degree n or lower; this is due to further cancelations of the areas around some of the knots. See Figure [Figure 55\(c\)](#) for an illustration. The above statements on accurate integration using numerical methods may sound like an oxymoron, but it is true. Careful computations with powerful algorithms can yield results with no errors.

Example 1: We compute $\int_0^1 dx x^3$ and $\int_0^1 dx x^4$ using Newton-Cotes method for various n 's. The Python function used for this computation is given below:

```
def integ_Newton_Cotes(m, f, a, b):
    x = np.linspace(a,b, m+1)
    coeff, error = newton_cotes(m, 1)
    h = (b-a)/m
    return (h * np.sum(coeff * f(x)))
```

```
In [74]: print(integ_Newton_Cotes(4, lambda x: x**4, 0, 1))
0.2
```

Trapezoid rule yields the integral to be $\frac{1}{2}$ for both of them, but it is far away from the exact results, $\frac{1}{3}$ and $\frac{1}{5}$. However, integration with $n \geq 3$ yields correct value for $\int_0^1 dx x^3$, with $n \geq 4$ provides correct integral for $\int_0^1 dx x^4$. This is consistent with the discussion of this section.

Example 2: We compute $\int_0^{\pi/2} dx \sin x$ numerically by dividing the interval $(0, \pi/2)$ into various intervals and employing Newton-Cotes schemes to them. We employ $n = 2, 3, 4, 5, 6$ abscissa points. The exact answer is 1, but integrals using $n = 2, 3, 4, 5, 6$ are 0.7853981633974483, 1.0022798774922104, 1.001004923314279, 0.9999915654729927, 0.9999952613861667 respectively. We plot the errors in

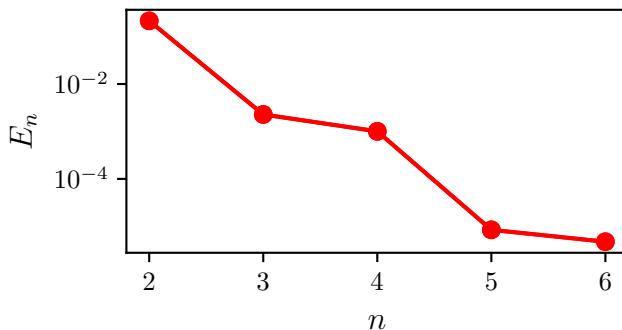


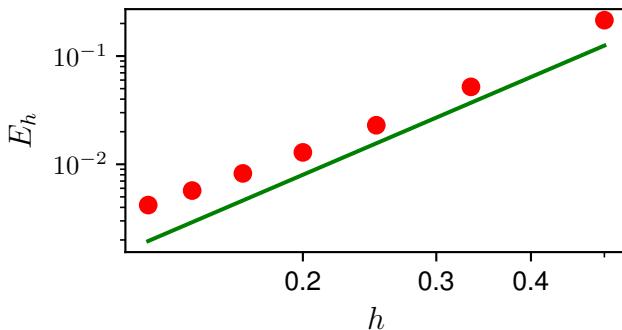
Figure 56: Example 2: Errors in integral $\int_0^{\pi/2} dx \sin x$ computed using Newton-Cotes method with $n = 2, 3, 4, 5, 6$ points.

The figure shows that error drops when we go from 2 to 3, but not so much from 3 to 4. Similarly, error does not drop significantly when n increases from 5 to 6. This is because the integrals using odd n 's are accurate up to $O(h^{n+1})$, which is also the accuracy with $n+1$ even points. See [Table 16](#) for details.

We can also integrate a function by dividing the intervals $(b-a)$ into many segments and by summing the integrals computed using Newton-Cotes method for all them. For example, we divide an interval (a,b) into $n-1$ segments, and employ trapezoid rule to all of them. This operation yields the following formula for the integral:

$$I = \int_a^b f(x) dx = \frac{h}{2} \left[f_0 + 2 \sum_{i=1}^{n-2} f_i + f_{n-1} \right]$$

We compute $\int_0^{\pi/2} dx \sin x$ using the above formula by dividing the interval $(0,\pi/2)$ into 2 to 9 intervals and employing Trapezoid rule on each segment. In [Figure 57](#) we plot error E_h vs. h for h varying from $(\pi/2)/2$ to $(\pi/2)/9$. As shown in the figure, E_h is proportional to h^3 , which is consistent with the error estimate of trapezoid rule.



[Figure 57](#): Plot of error E_h vs. h for the integral $\int_0^{\pi/2} dx \sin x$ when h varies from $(\pi/2)/2$ to $(\pi/2)/9$. The green line represents h^3 curve.

In the above illustrative examples, we know the answers to the integrals. However, in practical situations when we do not know the answers, we compare the integral $I(h)$ for h with that for a smaller interval, say $I(h/2)$. If the error, $|I(h)-I(h/2)|$, is less than the tolerance limit, then we can take $I(h/2)$ as the final numerical answer.

With this we close our discussion on Newton-Cotes schemes. In the next section we will describe Gaussian quadrature, which is more

accurate than Newton-Cotes method.

Conceptual questions

1. Comment on usefulness of Lagrange interpolation to the derivation of Newton-Cotes formulas.
2. For Newton-Cotes method, it is better to use odd number of points for integration. Why?

Exercises

1. Integrate $\int_0^1 x^\alpha dx$ for $\alpha = 2,3,4,5,6$ points using 2,3,4,5 points and compare the results with the exact ones. Are your results consistent with the formulas for the error estimates?
2. Integrate $\int_0^5 (x^6 - 5x^5 + 3)dx$ using Newton-Cotes method with 2,3,4,5 points and compare the results with the exact one. Also divide the interval into many segments and sum the integrals for each segment. Employ Trapezoid and Simpson rules for the latter integrals.
3. Integrate $\int_0^1 \exp(-x)\cos(x)dx$ using Newton-Cotes method with 2,3,4,5 points.
4. Integrate $\int_0^\infty \exp(-x^2)dx$ using Newton-Cotes method. What strategy would you adopt to get accurate integral upto 5%.
5. Compute the time period of an oscillator of mass 2 units and whose potential is $10x^2$.
6. Compute the time period of a pendulum whose length is 1 meter. The pendulum oscillates between -30 degrees to 30 degrees.

11.2 Gaussian Quadrature

Gaussian quadrature is more accurate than Newton-Cotes method. Newton-Cotes scheme with even N and odd N points provides accurate integrals for polynomials of $(N-1)$ and N degrees respectively. However, Gaussian quadrature with N points yields accurate integrals for polynomials of degree $(2N-1)$.

In Gaussian quadrature, we make both x_j and w_j variables to achieve higher accuracy for the integral:

$$I = \int_a^b f(x) dx \approx \sum_{j=0}^{N-1} w_j f(x_j)$$

If we demand exact quadrature for $f(x) = 1, x, x^2, \dots, x^{2N-1}$, we will have $2N$ equations using which we can obtain the aforementioned $2N$ unknowns.

Example 1: Let us compute x_j and w_j for $N=2$ for $[-1,1]$. We require the integrals for $f(x) = 1, x, x^2, x^3$ to be exact, which yields the following four equations:

$$\begin{aligned} 2 &= w_0 + w_1 \\ 0 &= w_0 x_0 + w_1 x_1 \\ 2/3 &= w_0 x_0^2 + w_1 x_1^2 \\ 0 &= w_0 x_0^3 + w_1 x_1^3 \end{aligned}$$

whose solutions are $w_0 = w_1 = 1$ and $x_0 = -x_1 = 1/\sqrt{3}$.

The above procedure is quite tedious and problem specific. In the following discussion, we provide a general formulation based on orthogonal polynomials. Note that our formulation should yield an exact quadrature for any polynomial of degree $2N-1$ or less with N data points. We denote the integrand by $f(x)$.

In Gauss quadrature, we take $N+1$ orthogonal polynomials $\phi_i(x)$ ($i = 0:N$) that satisfy the orthogonality relation:

$$\int_a^b h(x)\phi_i(x)\phi_j(x) = \delta_{ij}\gamma_i \dots(16)$$

where $h(x)$ is the weight function for the integral (different from the weights w_i for the integral), and γ_i 's are constants. Here, $\phi_N(x)$ is a polynomial of degree N and others are of lower degree. In Gauss quadrature, the N abscissas are the roots of $\phi_N(x)$, while the weights are

$$w_j = -\frac{a_N\gamma_N}{\phi'_N(x_j)\phi_{N+1}(x_j)} \dots(17)$$

In terms of these abscissas and weights, the integral is

$$I = \sum_{j=0}^{N-1} w_j f(x_j)$$

Derivation of Gauss quadrature

The derivation involves an interesting application of functional analysis and orthogonal polynomials (Ralston and Rabinowitz). This discussion can be skipped by those who are not interested in the mathematical details. The steps involved are as follows.

Step 1: First we assume that $f(x)$ is a polynomial of degree $2N-1$ or lower. A division of such $f(x)$ with $\phi_N(x)$ yields

$$f(x) = q_{N-1}(x)\phi_N(x) + r_{N-1}(x) \dots(18)$$

where the quotient $q_{N-1}(x)$ and remainder $r_{N-1}(x)$ are polynomials of degree $N-1$ or lower. Hence, $q_{N-1}(x)$ and $r_{N-1}(x)$ are orthogonal to $\phi_N(x)$. Therefore, using the orthogonality property of the polynomials we deduce that

$$\int_a^b h(x)q_{N-1}(x)\phi_N(x)dx = 0$$

Therefore,

$$\int_a^b h(x)f(x)dx = \int_a^b h(x)r_{N-1}(x)dx$$

Note that for a general $f(x)$, the integral with the quotient is not zero, and it is the error for the Gaussian quadrature.

Step 2: We simplify the integral further. $\phi_N(x)$ is a polynomial of degree N , hence it has N roots (assume real), which are denoted by x_j with $j=0:N-1$. Note that

$$f(x_j) = q_{N-1}(x_j)\phi_N(x_j) + r_{N-1}(x_j) = r_{N-1}(x_j)$$

Since $r_{N-1}(x)$ are polynomials of degree $N-1$ or lower, we expand it using Lagrange polynomials whose knots are located at the roots of $\phi_N(x)$. That is,

$$r_{N-1}(x) = \sum_j r_{N-1}(x_j)L_j(x) = \sum_j f(x_j)L_j(x)$$

Therefore,

$$\int_a^b h(x)f(x)dx = \sum_j f(x_j) \int_a^b h(x)L_j(x)dx = \sum_j w_j f(x_j)$$

$$\text{where } w_j = \int_a^b h(x)L_j(x)dx.$$

Step 3: Further, we derive w_j in terms of orthogonal polynomials. Since x_j 's are roots of $\phi_N(x)$, $\phi_N(x) = A_N \prod_i (x - x_i)$, while $\phi'_N(x) = A_N \prod_{i,i \neq j} (x_j - x_i)$. Note that A_N is the coefficient of x^N in $\phi_N(x)$.

Therefore,

$$L_j(x) = \frac{\phi_N(x)}{(x - x_j)\phi'_N(x)}$$

Using these properties, we deduce that

$$w_j = \int_a^b dx h(x) L_j(x) = \frac{1}{\phi'_N(x_j)} \int_a^b \frac{h(x)\phi_N(x)}{x - x_j} dx \quad \dots(19)$$

For further simplification, we employ Christoffel-Darboux identity:

$$\sum_{i=0}^N \frac{\phi_i(x)\phi_i(y)}{\gamma_i} = \frac{\phi_{N+1}(x)\phi_N(y) - \phi_N(x)\phi_{N+1}(y)}{a_N\gamma_N(x - y)}$$

where $a_m = A_{m+1}/A_m$, and γ_N is a constant of orthogonality relation (see (16)). In Christoffel-Darboux identity, substitution of $y = x_j$, a zero of $\phi_N(x)$, yields

$$\sum_{i=0}^N \frac{\phi_i(x)\phi_i(x_j)}{\gamma_i} = -\frac{\phi_N(x)\phi_{N+1}(x_j)}{a_N\gamma_N(x - x_j)}$$

We multiply the above equation with $h(x)\phi_0(x)$ and integrate in the interval $[a,b]$. In addition, we employ the orthogonality property and the fact that $\phi_0(x)$ is a constant. Consequently the above equation simplifies to

$$\frac{\phi_0(x_j)\gamma_0}{\gamma_0} = -\frac{\phi_{N+1}(x_j)}{a_N\gamma_N} \phi_0(x) \int_a^b dx \frac{h(x)\phi_N(x)}{(x - x_j)}$$

Hence,

$$\int_a^b dx \frac{h(x)\phi_N(x)}{(x - x_j)} = -\frac{a_N\gamma_N}{\phi_{N+1}(x_j)}$$

Substitution of which in Eq. (19) yields the weights as

$$w_j = -\frac{a_N\gamma_N}{\phi'_N(x_j)\phi_{N+1}(x_j)}$$

which is same as Eq. (17). This completes the derivation of Gauss quadrature formulas. The integral is exact when $f(x)$ is a polynomial of order $2N-1$ or lower.

Error in Gaussian quadrature: A division of a general $f(x)$ with $\phi_N(x)$

yields

$$f(x) = q(x)\phi_N(x) + r_{N-1}(x)$$

where $q(x) = \sum_j b_j \phi_j(x)$. Therefore, the integral of the quotient of Eq. (18) yields

$$\int_a^b h(x)q(x)\phi_N(x)dx = b_N \gamma_N$$

which is the error in the integral. The other terms of $q(x)$ ($j \neq N$) vanish due to the orthogonality properties of the polynomials. After algebraic manipulation we derive the error term as

$$E_{\text{int}} = \frac{\gamma_N}{A_N^2(2N)!} f^{(2N)}(\zeta) \quad \dots \quad (20)$$

In the following discussion we discuss specific implementations of Gaussian quadrature.

Quadrature based on Legendre polynomials

For Gauss quadrature with finite a and b , it is best to employ Legendre polynomials ($G_n(x)$) whose orthogonality properties are

$$\int_{-1}^1 G_i(x)G_j(x)dx = \frac{2}{2j+1} \delta_{ij}$$

Hence, $a = -1$, $b = 1$, $h(x) = 1$, $\gamma_j = 2/(2j+1)$. In [Table 17](#) we list the Legendre polynomials $G_0(x) \dots G_5(x)$; these polynomials are also depicted in [Figure 58](#).

Table 17: Legendre polynomials

$G_0(x) = 1$	$G_3(x) = \frac{1}{2}(5x^3 - 3x)$
$G_1(x) = x$	$G_4(x) = \frac{1}{8}(65x^5 - 70x^3 + 15x)$

$$G_2(x) = \frac{1}{2}(3x^2 - 1)$$

$$G_5(x) = \frac{1}{8}(231x^6 - 315x^4 + 105x^2 +$$

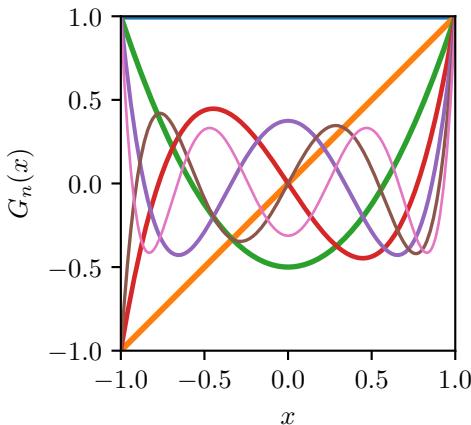


Figure 58: Plots of Legendre polynomials $G_0(x) \dots G_6(x)$ with curves of decreasing line thickness.

It can be shown that

$$A_j = \frac{(2j)!}{2^j(j!)^2}, \quad a_N = \frac{2}{N+1}$$

Using the above properties and an identity

$$(1-x^2)G'_j(x) = (j+1)xG_j(x) - (j+1)G_{j+1}(x)$$

we can show that

$$w_j = \frac{2}{(1-x_j^2)[G'_N(x_j)]^2}$$

The roots x_j and the weights w_j can be computed from the polynomials. In [Table 18](#) we list them for $N = 1..5$. The Python module `scipy.special.roots_legendre()` contains these quantities (see the Python code below). Note that the weights and abscissas of Example 1 match with that shown in [Table 18](#).

Table 18: Roots and weights of Legendre polynomials

N	x_j	w_j
2	$\pm 1/\sqrt{3}$	1
3	0	$8/9$
	$\pm 1/\sqrt{3/5}$	$5/9$
4	± 0.339981	0.347855
	± 0.861136	0.568889
5	0	$128/225$
	± 0.538467	0.478629
	± 0.90618	0.236927

To employ Gauss Quadrature $\int_a^b f(x')dx'$ for an arbitrary interval $[a,b]$,

we employ a liner transformation $x' = \alpha x + \beta$ and impose the condition that $x' = a \rightarrow x = -1$ and $x' = b \rightarrow x = 1$. Consequently, $\alpha = (b-a)/2$ and $\beta = (b+a)/2$. Therefore,

$$I = \int_a^b f(x')dx' = \frac{b-a}{2} \int_{-1}^1 f(\alpha x + \beta)dx \quad \dots(21)$$

Now the above integral can be computed using Legendre polynomials in the interval $[-1,1]$.

Example 2: We compute $\int_{-1}^1 x^8 dx$ using Gauss quadrature. The actual answer is $2/9$. For the Gauss quadrature described above with $N=2,3,4,5$, the errors are -0.19753086419753085 , -0.0782222222222211 , -0.01160997732426308 , and 0 respectively. This is consistent with the fact that for $N=5$, we expect that quadrature to be exact for any polynomial of degree 9 or lower. For the above computation, we use the following code:

```
from scipy.special import roots_legendre
def f(x):
    return x**8

for n in range(2,6):
    xi = roots_legendre(n)[0]
```

```
wi = roots_legendre(n)[1]
yi = f(xi)

In = sum(yi*wi)
print(n, In, In-2/9)
```

Note that `scipy.special.roots_legendre(n)` returns a tuple whose elements are the roots and weights of $P_n(x)$.

Example 3: We compute $\int_0^1 \exp(x)dx$ numerically. The actual answer is $e-1$. However, for $N=2,3,4,5$, the Gauss quadrature of Eq. (21) yields approximate integral with errors as -0.0003854504515410362 , $-8.240865234654393e-07$, $-9.329670369595533e-10$, $-6.534772722943671e-13$. These integrals are quite accurate considering that we are using maximum of 5 data points.

Laguerre-Gauss quadrature

We often encounter integral of the form

$$\int_0^\infty e^{-x} f(x) dx \quad \dots(22)$$

The above computation using Newton-Cote's scheme will be very expensive because we have to take many intervals to reach $x = \infty$. Here, Laguerre-Gauss quadrature offers an attractive option.

For the above integral, we employ Laguerre polynomials ($L_m(x)$) whose orthogonality relation is

$$\int_0^\infty e^{-x} L_i(x) L_j(x) dx = \delta_{i,j}$$

Therefore, $a = 0$, $b = \infty$, $w(x) = e^{-x}$, and $\gamma_j = 1$. Some of the low-order Laguerre polynomials are listed [Table 19](#) and exhibited in [Figure 59\(a\)](#). Since the values of the functions are quite large, we employ `plt.yscale('symlog')` for plotting in logscale for negative numbers as well.

Table 19: Laguerre polynomials

$$\begin{array}{ll} L_0(x) = 1 & L_2(x) - \frac{1}{2}(x^2 - 4x + 2) \\ \hline L_1(x) = -x + 1 & L_3(x) = \frac{1}{6}(-x^3 + 9x^2 - 18x + 6) \end{array}$$

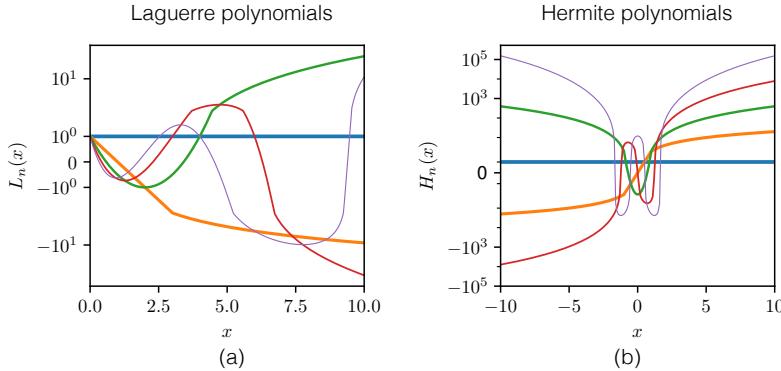


Figure 59: Plots of (a) Laguerre polynomials and (b) Hermite polynomials in semiology format. The thickness of the curves decreases for higher order polynomials.

These polynomials satisfy the property $A_j = (-1)^j / j!$ and $a_N = -1 / (N+1)$. The weights for the Laguerre-Gauss quadrature are

$$w_j = \frac{1}{(N+1)L'_N(x_j)L_{N+1}(x_j)} = -\frac{1}{NL_{N-1}(x_j)L'_N(x_j)}$$

Using the above ingredients and the identity

$$x_j L'_N(x_j) = -NL_{N-1}(x_j) = (N+1)L_{N+1}(x_j)$$

We deduce that

$$w_j = \frac{1}{x_j[L'_N(x_j)]^2} = \frac{x_j}{(N+1)^2[L_{N+1}(x_j)]^2}$$

The roots x_j and the weights w_j of the leading-order Laguerre-Gauss polynomials are listed in [Table 20](#).

Table 20: Roots and weights of Laguerre polynomials

N	x_j	w_j
2	0.585786	0.853553
	3.41421	0.146447
3	0.415775	0.711093
	2.29428	0.278518
4	6.28995	0.0103893
	0.322548	0.603154
5	1.74576	0.357419
	4.53662	0.0388879
6	9.39507	0.000539295
	0.26356	0.521756
7	1.4134	0.398667
	3.59643	0.0759424
8	7.08581	0.00361176
	12.6408	0.00002337

Now, using the x_j 's and w_j 's we compute the integral

$$\int_0^\infty e^{-x} f(x) dx = \sum_{j=0}^{N-1} w_j f(x_j),$$

Example 4: Let us compute $\int_0^\infty \exp(-x)x^n dx$ using Lagurre-Gauss quadrature. The exact value of the integral is $n!$. For $n=4$, the integral with two points $N=2$ is 20.000000000000007, which is incorrect. However, the quadrature yields correct result, which is 24, for $N=3$ and beyond. Similarly, for $n=9$, the quadrature yields correct value for $N=5$ and beyond.

Example 5: For a stationary state of Hydrogen atom, $\langle f(r) \rangle$ is

$$\langle f(r) \rangle = \int d\mathbf{r} f(r) |\psi(\mathbf{r})|^2$$

Where $\psi(\mathbf{r})$ is the wavefunction. The ground state of the H-atom is

$\psi_{1,0,0} = \frac{1}{\sqrt{\pi}} \exp(-r')$, where $r' = r/r_a$ is nondimensional radius with r_a is the Bohr radius. For this state,

$$\langle r' \rangle = \frac{1}{\pi} \int_0^\infty e^{-2r'} 4\pi(r')^3 dr' = \frac{4}{16} \int_0^\infty e^{-2r'} (2r')^3 d(2r')$$

Which has same form as Eq. (22). We compute the above integral using Laguerre-Gauss quadrature that yields $\langle r' \rangle = 3/2$ or $\langle r \rangle = (3/2)r_a$. Similarly, we can derive that

$$\langle \frac{1}{r'} \rangle = \frac{1}{\pi} \int_0^\infty e^{-2r'} 4\pi r' dr' = \frac{4}{4} \int_0^\infty e^{-2r'} (2r') d(2r') = 1$$

Therefore, the average potential energy is $\langle U \rangle = (-e^2/r_a)\langle 1/r' \rangle = -(\epsilon^2/r_a)$. Virial theorem tells us that the average kinetic energy $\langle KE \rangle = 2\langle U \rangle$. Therefore, $\langle E_0 \rangle = \langle U \rangle + \langle KE \rangle = -(e^2/2r_a)$. Other quantities can be computed in a similar manner. Note that $\langle 1/r \rangle \neq 1/\langle r \rangle$.

Example 6: We compute the average radius, $\langle r \rangle$, for Hydrogen atom's $\psi_{2,0,0}$ state, whose normalised wavefunction (Section 9.2) is

$$\psi_{2,0,0} = \frac{1}{\sqrt{32\pi}} (2 - r') \exp(-r'/2).$$

For this state, we find that

$$\langle r' \rangle = \frac{1}{32\pi} \int_0^\infty e^{-r'} (2 - r')^2 4\pi(r')^3 dr' = 6$$

Hence, $\langle r \rangle = 6 r_a$. Similarly, we find that $\langle 1/r' \rangle = 1/4$. Hence $\langle U \rangle = (-e^2/r_a)\langle 1/r' \rangle = (-e^2/4r_a)$ and $\langle E \rangle = (-e^2/8r_a) = \langle E_0 \rangle / 4$.

Hermite-Gauss quadrature

For integrals of the form $\int_{-\infty}^{\infty} e^{-x^2} f(x) dx$, we employ Hermit's

polynomials that satisfy the following orthogonal relation:

$$\int_{-\infty}^{\infty} e^{-x^2} H_i(x) H_j(x) dx = j! \sqrt{2\pi} \delta_{ij}$$

Hence, $a = -\infty$, $b = \infty$, $h(x) = \exp(-x^2)$, $\gamma_j = j! \sqrt{2\pi}$. Some of the low-order Legendre polynomials are listed in Table 21, and depicted in Figure 59(b).

Table 21: Hermite polynomials

$H_0(x) = 1$	$H_3(x) = 8x^3 - 12x$
$H_1(x) = 2x$	$H_4(x) = 16x^4 - 48x^2 + 12$
$H_2(x) = 4x^2 - 2$	$H_5(x) = 32x^5 - 160x^3 + 120x$

For the Hermite-Gauss quadrature, the weights w_j are

$$w_j = \frac{2^{N-1} N! \sqrt{\pi}}{N^2 [H_{N-1}(x_j)]^2}$$

which are listed in Table 22 along with the abscissas.

Table 22: Roots and weights of Hermite polynomials

N	x_j	w_j
2	$\pm(\sqrt{2})/2$	$(\sqrt{\pi})/2$
3	0	$2(\sqrt{\pi})/3$
	$\pm(\sqrt{6})/2$	$(\sqrt{\pi})/6$
4	± 0.524648	0.804914
	± 1.65068	0.0813128
5	0	0.945309
	± 0.958572	0.393619
	± 2.02018	0.0199532

Example 7: Let us compute $\int_{-\infty}^{\infty} \exp(-x^2) x^4 dx$ using Hermite-Gauss quadrature. The exact value of the integral is $(3/4)\sqrt{\pi}$. Numerical computation yields the correct result for $N=3$ and beyond. For $N=2$,

the numerical integral yields 0.4431134627263788. For $\int_{-\infty}^{\infty} \exp(-x^2) x^n dx$, the quadrature will yield a correct value when $N=(n/2)+1$ and beyond.

Example 8: Hermite-Gauss quadrature is useful for solving quantum oscillator problem. We take the wavefunctions of the oscillator given in Section 9.2 and compute $\langle f(x) \rangle$ for ψ_n using

$$\langle f_n(x) \rangle = \int dx f(x) |\psi_n(x)|^2$$

We observe that $\langle 1 \rangle = 1$ indicating that the wavefunctions are normalised. In addition, we observe that for ψ_n , $\langle x^2 \rangle = n + \frac{1}{2}$. Therefore, average potential potential energy $\langle U \rangle = \langle x^2 \rangle / 2 = (n + \frac{1}{2}) / 2$. Using Virial theorem, we infer that $\langle T \rangle = (n + \frac{1}{2}) / 2$ and total energy $= (n + \frac{1}{2})$. In dimensional form, total energy is $(n + \frac{1}{2})\hbar\omega$.

With this we end our discussions on numerical integration. We have not covered many topics, such as singular integrals, multidimensional integrals, etc. We encourage the reader to browse through references for such topics.

Conceptual questions

1. Why is Gaussian quadrature more accurate than Newton-Cotes method?
2. With N data points, list the polynomials that can be integrated accurately using Gaussian quadrature.
3. Derive the expression for the error in Gaussian quadrature (Eq. (20)).
4. In this chapter, we describe one-dimensional integrals. Generalize the formulas derived here to two-dimensional and three-dimensional integrals.

Exercises

1. Integrate $\int_0^5 (x^6 - 5x^5 + 3)dx$ using Gauss quadrature with $N = 2, 3, 4, 5$ knots and compare the results with the exact one. Compare the accuracy of the results with those from Newton-Cotes schemes.
2. Compute the following integrals using Gauss quadrature:
 $\int_0^\infty \exp(-x)(x^6 - 5x^5 + 3)dx$ and $\int_0^\infty \exp(-x^2)(x^6 - 5x^5 + 3)dx$. How many knots do you need for the accurate integration of the above?
3. Integrate $\int_0^\infty \frac{x^3}{\exp(x) - 1} dx$ using Gaussian quadrature. This integral appears in black-body spectrum.
4. Integrate $\int_0^1 \frac{x^4 \exp(x)}{(\exp(x) - 1)^2} dx$ using Gaussian quadrature.
5. Integrate $\int_0^\infty \frac{\sin(x)}{x} dx$, $\int_0^1 \exp(-x)\cos(x)dx$, $\int_0^\infty \exp(-x)\cos(x)dx$ using Gaussian quadrature.
6. Consider the $\psi_{2,1,0}$ and $\psi_{2,1,\pm 1}$ wave functions of Hydrogen atom. Show that these wavefunctions are normalised. Compute the average radius and energies for these quantum states. It is best to work with nondimensionalized wavefunctions.
7. Consider the ground state and first, second, and third excited states of quantum oscillator. Compute $\langle x^3 \rangle$, $\langle x^4 \rangle$, $\langle x^6 \rangle$, $\langle x^8 \rangle$ for the above functions. Work with nondimensionalized wavefunctions.
8. Consider the ground state of a two-dimensional linear quantum oscillator. Compute the energy of this state.
9. Compute the time period of a particle of mass m and energy E that executes a periodic motion in the potential fields:
 - (a) $U(x) = -U_0/\cosh^2 x$ with $-U_0 < E < 0$
 - (b) $U(x) = U_0 \tan^2 x$

11.3 Python's quad & Multidimensional Integrals

In this section we will describe how to compute multidimensional integrals, and how to compute integrals using Python's quad functions. At the end, we will discuss some examples how to deal with complicated integrals.

Multidimensional integrals

It is quite straightforward to generalise 1D quadrature schemes to higher dimensions. In the following discussion we illustrate how to work out the 2D integral $\int_a^b dx \int_0^{g(x)} dy f(x, y)$. As shown in [Figure 60](#), for the x integral, we choose abscissas at $\{x_0, x_1, \dots, x_{n-1}\}$. After that, at each abscissa, we choose knots along the y direction. The coordinates of y -abscissas are $\{y_0, y_1, \dots, y_{n-1}\}$. Now, the integral is performed as follows:

$$\int_a^b dx \int_0^{g(x)} dy f(x, y) = \sum_i w_i \int_0^{g(x)} dy f(x_i, y) = \sum_i w_i \sum_j w_{i,j} f(x_i, y_j)$$

Note that the weights along the y -direction will depend on x_i . The above formula works for both Newton-Cotes and Gauss quadrature. The above formula can be implemented using 1D integrals discussed earlier.

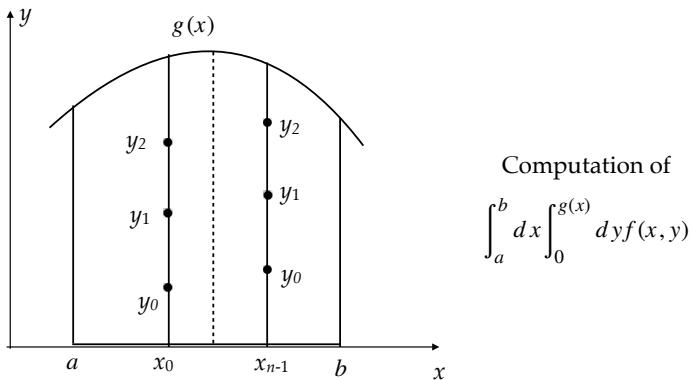


Figure 60: A diagram illustrating how to perform 2D quadrature. We compute the functions at the dotted points.

Python's functions for integration

Fortunately, Python offers functions for integrations. Python's function *quad* (a part of *scipy.integrate* module) computes 1D integrals using Gaussian quadrature. The usage of the function is as follow.

```
from scipy.integrate import quad
print (quad(lambda x: x,0,1)) # 1D
```

The above code computes $\int_0^1 x \, dx$ and yields $\frac{1}{2}$ as an answer.

For 2D and 3D integrals, the corresponding functions are *dblquad* and *tplquad* respectively. In the following, *dblquad* and *tplquad* compute the following integrals:

$$\int_0^1 dx \int_0^x dy (xy); \sim \sim \sim \int_0^1 dx \int_0^x dy \int_0^{x+y} dz (xyz)$$

```
from scipy.integrate import dblquad, tplquad
print (dblquad(lambda x,y: x*y, 0, 1, 0, 1, lambda x: x))
```

```
print (tplquad(lambda x,y,z: x*y*z, 0, 1, 0, lambda x: x,
0, lambda x,y: x+y))
```

In the above examples, lambda functions provide a convenient means to pass the functions and integral limits as anonymous functions. The results from Python are $1/8$ and 0.1180555555555555 ($17/144$), which are exact values of the integrals.

We also point out that *quad()* function can handle improper integrals (whose one or both the limits are ∞ or $-\infty$). For example,

```
In [121]: print(quad(lambda x: np.exp(-x**2), 0, inf))
(0.8862269254527579, 7.101318390472462e-09)
```

Python's functions *trapz* and *simps* integrates numerical data using trapezoid and Simpson's rules. For example,

```
In [92]: y = [1,2,3]
In [94]: from scipy.integrate import simps
In [95]: simps(y)
Out[95]: 4.0
```

Here, h is assumed to be unity. We could also give sampling points as arguments of *simps* and *trapz*. For example, $x=[0.1, 0.2, 0.25]$; *simps(y,x)*.

Tricks for integrals

Even though *quad()* can handle improper integrals, it is important to know how to handle such cases. We will illustrate some of the tricks that are used to handle such integrals, as well as singular integrals.

Example 1: One of the limits of the integral $\int_0^\infty \exp(-x^2)dx$ is at ∞ . To avoid this, we make a change of variable $y = x/(1+x)$. In terms of y , the integral is

$$\int_0^1 \frac{\exp(-(y/(1-y))^2)}{(1-y)^2} dy$$

that has finite limits. Incidentally, both the integrals can be compute quite easily using Gauss quadrature.

Example 2: To evaluate $\int_1^\infty \frac{\exp(-x)}{(1+x)^2} dx$, we rewrite the integral as

$$\int_1^\infty \frac{\exp(-x)}{(1+x)^2} dx = \int_0^\infty \frac{\exp(-x)}{(1+x)^2} dx - \int_0^1 \frac{\exp(-x)}{(1+x)^2} dx$$

It is best to compute the first integral using Laguerre-Gauss quadrature and the second using Legendre-Gauss quadrature.

Example 3: The integral $\int_0^{\pi/2} \frac{\cos(x)}{x^{1/3}} dx$ is singular at $x = 0$. We can avoid singularity by subtraction or by performing the integral by parts. Avoidance of singularity is as follows:

$$\int_0^{\pi/2} \frac{\cos(x)}{x^{1/3}} dx = \int_0^{\pi/2} \frac{\cos(x) - 1}{x^{1/3}} dx + \int_0^{\pi/2} \frac{1}{x^{1/3}} dx$$

The first term is nonsingular. The second term, though singular, is easily integrated. The integral by parts is as follows:

$$\int_0^{\pi/2} \frac{\cos(x)}{x^{1/3}} dx = \frac{3}{2} x^{2/3} \cos(x) \Big|_0^{\pi/2} + \frac{3}{2} \int_0^{\pi/2} x^{2/3} \sin(x) dx$$

The first term is zero, while the second term is nonsingular.

Exercises

- Compute the following integrals using Python's quad functions: Integrate $\int_0^1 \frac{x^4 \exp(x)}{(\exp(x) - 1)^2} dx$, $\int_0^\infty \frac{x^3}{\exp(x) - 1} dx$, $\int_0^\infty \exp(-x)(x^6 - 5x^5 + 3) dx$, $\int_0^1 dx \int_0^{x^2} dy \sin^2(xy)$, $\int_0^1 dx \int_0^x dy \int_0^{x+y} dz (x^2 y^3 z^4)$.
- Compute the integrals of Examples 1, 2, and 3.
- Compute the following singular integral $\int_0^{\pi/2} \frac{\cos^2(x)}{x^{1/2}} dx$.

CHAPTER TWELVE
NUMERICAL DIFFERENTIATION

12.1 Computing Numerical Derivatives

Differentiation is an important operation in science and engineering. For example, the velocity of a particle is computed by taking derivative of its position. The derivatives of temperature and population tells us about their temporal variations. Also, physical systems are often described using differential equations, in which derivatives play an important role. In this chapter we will focus on numerical differentiation.

According to Newton and Leibniz, the formula for a derivative of a function $f(x)$ is

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

Mathematically, $h \rightarrow 0$, but h is finite but small in numerical computations. Hence, numerical $f'(x)$ is an approximation of the actual derivative. For the derivation for a formula for $f'(x)$, we start with the Lagrange polynomial with two points (x_i, x_{i+1}) . For brevity, we denote the Lagrange polynomial with $f(x)$ itself.

$$f(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}} f_i + \frac{x - x_i}{x_{i+1} - x_i} f_{i+1}$$

The derivatives of the above function at both the points, x_i and x_{i+1} , are the same. Yet, notationally, we write them as

$$\text{Forward difference: } f'(x_i) \approx D_+ f \approx \frac{f_{i+1} - f_i}{h_i}$$

$$\text{Backward difference: } f'(x_{i+1}) \approx D_- f \approx \frac{f_{i+1} - f_i}{h_i}$$

where $h_i = x_{i+1} - x_i$. Here, the *forward difference* is the derivative computed at x_i looking toward the forward point x_{i+1} . The *backward difference* is the derivative computed at x_{i+1} looking backward towards x_i . See [Figure 61](#) for an illustration. The computed slopes (the blue lines of the figure) are approximations to the actual slopes (green lines).

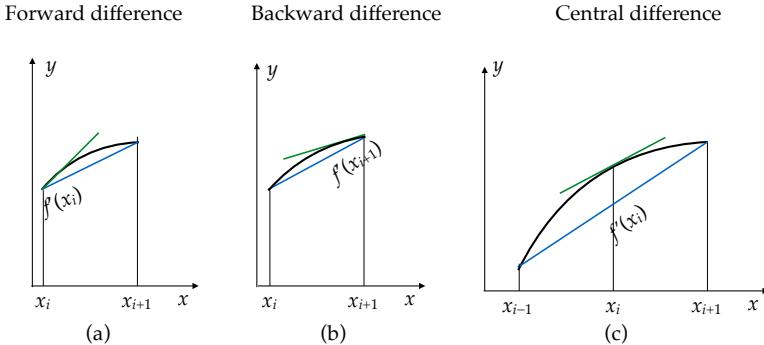


Figure 61: Schematic diagrams exhibiting (a) forward difference, (b) backward difference, and (c) central difference. The green lines are the actual slopes, while blue lines represent the computed slopes.

We expect the accuracy of the derivatives to improve with more number of points. Hence, we work with the following Lagrange polynomial passing through three points (x_{i-1}, x_i, x_{i+1}) :

$$f(x) = \frac{(x - x_i)(x - x_{i+1})}{(h_{i-1} + h_i)(h_{i-1})} f_{i-1} + \frac{(x - x_{i-1})(x - x_{i+1})}{h_{i-1}(-h_i)} f_i + \frac{(x - x_{i-1})(x - x_i)}{(h_{i-1} + h_i)(h_i)} f_{i+1} \dots (23)$$

The first and second derivatives of the above function at x_i are

$$f'(x_i) = -\frac{h_i}{(h_{i-1} + h_i)(h_{i-1})} f_{i-1} + \left(\frac{1}{h_{i-1}} - \frac{1}{h_i} \right) f_i + \frac{h_{i-1}}{(h_{i-1} + h_i)(h_i)} f_{i+1}$$

$$f''(x_i) = \frac{2}{(h_{i-1} + h_i)(h_{i-1})} f_{i-1} - \frac{2}{h_{i-1} h_i} f_i + \frac{2}{(h_{i-1} + h_i)(h_i)} f_{i+1}$$

The aforementioned complex expression gets simplified when the points are equidistant, that is, when $h_{i-1} = h_i = h$. Under this assumption, the first derivatives at the three points are

$$\text{Forward difference: } f'(x_{i-1}) = D_+ f = \frac{-3f_{i-1} + 4f_i - f_{i+1}}{2h}$$

$$\text{Backward difference: } f'(x_{i+1}) = D_- f = \frac{f_{i-1} - 4f_i + 3f_{i+1}}{2h}$$

$$\text{Central difference: } f'(x_i) = \frac{1}{2}(D_+ + D_-)f = \frac{f_{i+1} - f_{i-1}}{2h}$$

The central difference at $x = x_i$, exhibited in [Figure 61\(c\)](#), is more accurate than the forward and backward differences. The second derivatives at the three points are equal (see Eq. (23)):

$$f''(x_i) = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} = D_+ D_- f = D_- D_+ f$$

We can compute even more accurate derivatives with more number of points. In [Table 23](#) we list the first and second derivatives for 2, 3, and 5 equidistant points. Note the usefulness of Lagrange interpolation formulas for the derivative computations.

Table 23: Coefficients for various formulas for the derivatives. The last column is the error in the derivative.

	f_{i-2}	f_{i-1}	f_i	f_{i+1}	f_{i+2}	Error
<i>Forward</i>						
hf'_i			-1	1		$O(h)$
$2hf'_i$			-3	4	-1	$O(h^2)$
$h^2f''_i$			1	-2	1	$O(h)$
<i>Backward</i>						
hf'_i		-1	1			$O(h)$
$2hf'_i$	1	-4	3			$O(h^2)$
$h^2f''_i$	1	-2	1			$O(h)$
<i>Central</i>						
$2hf'_i$		-1	0	1		$O(h^2)$
$12hf'_i$	1	-8	0	8	1	$O(h^4)$
$h^2f''_i$		1	-2	1		$O(h^2)$
$12h^2f''_i$	-1	16	-30	16	-1	$O(h^4)$

Errors in the derivatives

We can estimate the errors in the numerical derivatives using the error formula for the Lagrange polynomials, which is Eq. (9) of Section 10.1. For n data points, the error is

$$E_n(x) = f(x) - P_n(x) = \frac{f^{(n)}(\zeta)}{n!} \prod_i (x - x_i)$$

The derivative of the above at $x = x_j$ yields the error in the derivative at that point:

$$\Delta f'(x_j) = \frac{d}{dx} E_j(x) |_{x=x_j} = \frac{f^{(n)}(\zeta)}{n!} \prod_{i,i \neq j} (x_j - x_i) \dots (24)$$

Therefore, for equidistant points, $\Delta f'(x_j) \sim h^{n-1}$, which are listed in [Table 23](#). The error formula also tells us that the first-derivative computed using n points is accurate for any polynomial of $(n-1)^{\text{th}}$ degree.

Derivation using Taylor Series

We can also derive the formulas for the derivatives using Taylor's series, which is

$$f(x \pm h) = f(x) \pm h f'(x) + \frac{h^2}{2} f''(x) \pm \dots$$

Using the above formula we derive that

$$\begin{aligned} a f(x-h) + b f(x) + c f(x+h) &= (a+b+c)f(x) + (c-a)h f'(x) \\ &\quad + (c+a)f''(x) h^2/2 + (c-a)f'''(x) h^3/6 + \dots \end{aligned}$$

For derivation of $f'(x)$ up to $O(h^2)$, we set

$$A+b+c = 0; \quad c-a = 1/h; \quad \text{and} \quad c+a = 0.$$

whose solution is $b = 0$, $c = 1/(2h)$, and $a = -1/(2h)$. Hence,

$$f'(x_i) = \frac{f(x+h) - f(x-h)}{2h} = \frac{f_{i+1} - f_{i-1}}{2h}$$

with error as $(c-a)f'''(x)h^3/6 = f''(x)h^2/6$, consisted with the formulas listed in [Table 23](#). Using similar analysis, we can also calculate a formula for $f''(x)$.

It may be tempting to employ many points to compute the derivatives. However, for large n with small h , the gains in accuracy is offset by the machine precision. For example, with 5 points and $h = 10^{-4}$, the error in derivatives is of the order of $h^4 \sim 10^{-16}$, which is close to machine precision. Hence, it is not advisable to use too many points for the derivative computations.

Using Python *gradient* function

Python's numpy offers a function called *gradient* to compute numerical differences. For a given array, *gradient* computes second-order accurate central differences in the interiors and either first- or second-order accurate differences at the edges. The second argument of the gradient function is the spacing between the consecutive points.

```
In [207]: y
Out[207]: array([0. , 0.25, 1. ])

In [208]: gradient(y) # first-derivative, same as
gradient(y,1)
Out[208]: array([0.25, 0.5 , 0.75])

In [210]: gradient(y,2) # gradient(y)/2
Out[210]: array([0.125, 0.25 , 0.375])
```

[Example 1](#): We take $f(x) = x^2$ and compute the first and second derivatives using three points located at $x = 0, \frac{1}{2}, 1$. The values of the function at these points are 0, $1/4$, 1 respectively. We employ the formulas of [Table 23](#) for derivative computations.

With two points, the forward first-order derivatives are $f'(0) = (f(\frac{1}{2})-f(0))/h = 0.5$ and $f'(\frac{1}{2}) = (f(1)-f(\frac{1}{2}))/h = 1.5$, while the backward first-order derivatives are $f'(1) = (f(1)-f(1/2))/h = 1.5$ and $f'(\frac{1}{2}) = (f(1/2)-f(0))/h = 0.5$. All these derivatives differ from the exact values.

With three points, we obtain forward difference $f'(0) = 0$, backward difference $f'(1) = 2$, and central difference $f'(1/2) = 1$. The second-order derivative $f''(0) = f''(1/2) = f''(1) = 2$. All these derivatives are accurate. This is because the derivatives of a quadratic function can be

computed exactly using three points.

Example 2: We compute the derivatives for $f(x) = \exp(x)$ using 3 points located at $x = 0, \frac{1}{2}, 1$. The forward difference $f'(0)$, backward difference $f'(1)$, and central difference $f'(1/2)$ are 0.8766032543414677, 2.559960402576623, 1.718281828459045 respectively. They differ from the exact values. The second derivatives at these points are same (1.6833571482351548), and they differ from the exact values.

We also remark that we can compute the accurate derivatives using Fourier transforms. We will discuss these computations in Chapter XXX.

In the next chapter we will describe how to solve ordinary differential equations using computers.

Conceptual questions

1. Why do the numerical derivatives have errors?
2. It is not recommended to use many points (say 8-10) for the derivative computations. Why?
3. Derive Eq. (24).
4. Derive a formula for the third derivative using four-point Lagrange interpolating polynomial.

Exercises

1. Discretize the function $f(x) = x^3$ using 2,3,4,5 points in $[0,1]$. For each case, compute the first and second derivatives at each point using forward, backward, and central differences. Compute the errors and compare them with the error laws, e.g., errors for the first derivative is proportional to h^n . For what n do you expect accurate results?
2. Repeat Exercise 1 for $f(x) = 1/x$ in the interval $[1,2]$.
3. Repeat Exercise 1 for $f(x) = \sin x$ in the interval $[0, \pi/2]$.

CHAPTER THIRTEEN

*ORDINARY DIFFERENTIAL EQUATION: INITIAL
VALUE PROBLEMS*

Synopsis

“Science is a differential equation.”

Alan Turing

13.1 General Overview

Many natural laws are best expressed using differential equations: ordinary differential equations (ODE) and partial differential equations (PDE). For example, Newton's equation of motion is a second-order differential equation for the position of a particle. Schrödinger's equation, a PDE, describes the quantum world. The fluids flows in nature are described using Navier-Stokes equation. In this chapter we will describe how to solve ODE's using numerical algorithms. In Chapters XXX we will cover PDEs.

In science and engineering we come across differential equations of all orders, but most of them are either first- or second-order equations. For example, Newton's equation of motion, $\ddot{p} = m\ddot{x} = F(x, \dot{x}, t)$, is second-order DE (differential equation) in time. However, for numerical computations, it is convenient to work with first-order DE's. Fortunately, any n^{th} -order DE's can be reduced to n first-order DE's. For example, $\ddot{p} = F(x, t)$ is equivalent to a set of two first-order DEs: $m\dot{x} = p$ and $\dot{p} = F(x, t)$.

A n^{th} -order ODE requires n initial conditions. For example, Newton's equation, $m\ddot{x} = F(x, \dot{x}, t)$, requires two initial conditions, $x(0)$ and $\dot{x}(0)$. In this chapter, we solve the differential equations using these initial conditions. This is the reason why we refer to the above as *initial value problem*. In Chapter Seventeen we will solve higher-order (2nd or higher) differential equations using *boundary values*, for example, $x(0)$ and $x(1)$ for Newton's equation of motion.

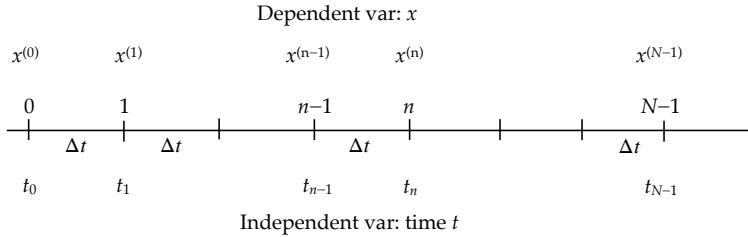
In the following discussion we will describe how to solve a first-order DE numerically. Let us consider the following DE with t as an independent variable and x as a dependent variable:

$$\frac{dx}{dt} = \dot{x} = f(x, t) \quad \dots (25)$$

In this chapter we solve the above equation given initial condition $x(t = 0)$.

For numerical computation we discretize the total time into $N-1$ steps with markers at $t_0, t_1, \dots, t_n, t_{n+1}, \dots, t_{N-1}$. We denote the variable x at these times as $x^{(0)}, x^{(1)}, \dots, x^{(n)}, x^{(n+1)}, \dots, x^{(N-1)}$. For simplicity we

assume the time markers to be equidistant with $t_{n+1} - t_n = \Delta t$ for all n . See [Figure 62](#) for an illustration. We contrast the indices of the dependent and independent variables using superscripts and subscripts to avoid any ambiguity.



[Figure 62](#): The total time is divided into $N-1$ steps. The time and x at the i^{th} marker are t_i and $x^{(i)}$ respectively.

An integration of Eq. (25) from t_n to t_{n+1} yields

$$x^{(n+1)} - x^{(n)} = \int_{t_n}^{t_{n+1}} f(x, t) dt$$

Now we employ some of the integration schemes studied in Section 11.1 to approximate the integral $\int_{t_n}^{t_{n+1}} f(x, t) dt$. The solution with different schemes are

Euler's forward method: $x^{(n+1)} = x^{(n)} + (\Delta t) f(x^{(n)}, t_n)$

Euler's backward method: $x^{(n+1)} = x^{(n)} + (\Delta t) f(x^{(n+1)}, t_{n+1})$

Midpoint method: $x^{(n+1)} = x^{(n)} + (\Delta t) f(x^{(n+\frac{1}{2})}, t_{n+\frac{1}{2}})$

Trapezoid method: $x^{(n+1)} = x^{(n)} + (\frac{1}{2})(\Delta t) [f(x^{(n)}, t_n) + f(x^{(n+1)}, t_{n+1})]$

These methods, called ODE solvers, prescribe how to advance from time t_n to t_{n+1} . We illustrate the *Euler forward*, *Euler backward*, and *midpoint methods* in [Figure 63](#).

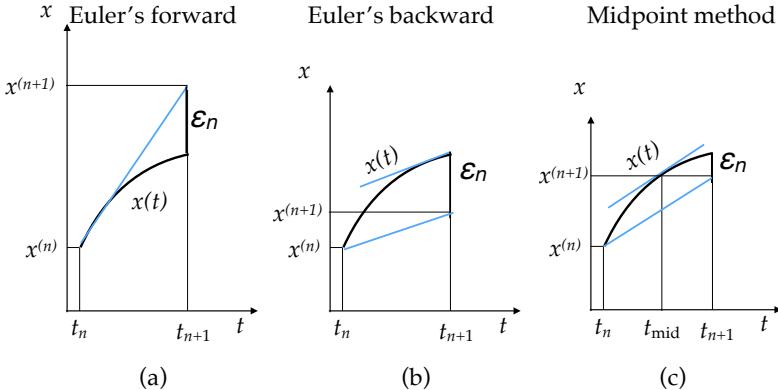


Figure 63: Schematic diagrams exhibiting (a) Euler's forward method, (b) Euler's backward method, and (c) midpoint method. The blue lines touching $x(t)$ represent the slopes at the respective points. ε_n is the error between the actual value and the computed value at $t = t_{n+1}$.

In Euler's forward method, we compute $x^{(n+1)}$ using the slope at t_n . Since we go forward in time, the slope $f(x^{(n)}, t_n)$ can be easily computed. Such schemes are called *explicit schemes* because the unknown $x^{(n+1)}$ depends explicitly on the knowns $x^{(n)}$, t_n , and t_{n+1} . The difference between the computed $x^{(n+1)}$ and the actual value is the error ε_n .

In Euler's backward method, $x^{(n+1)}$ is computed using the slope $f(x^{(n+1)}, t_{n+1})$ at t_{n+1} . Since the slope depends on the unknown variable itself, this scheme is an example of *implicit schemes*. In general, implicit schemes are more complex to implement.

In the *trapezoid method*, the slope is average of the slopes at $x^{(n)}$ and $x^{(n+1)}$. Therefore, this method is called a *semi-implicit method*. The midpoint method involves $x^{(n+\frac{1}{2})}$, which is also an unknown; we need to come up with a way to estimate $x^{(n+\frac{1}{2})}$.

An integration of Eq. (25) from t_{n-1} to t_{n+1} using midpoint method yields

$$x^{(n+1)} = x^{(n-1)} + (\Delta t) f(x^{(n)}, t_n)$$

This explicit scheme is called *Leapfrog method*. In a related method, a usage of Simpsons's $\frac{1}{3}$ rule for the integration from t_{n-1} to t_{n+1} yields

$$x^{(n+1)} - x^{(n-1)} = \frac{1}{3}(\Delta t) \left[f(x^{(n-1)}, t_{n-1}) + 4f(x^{(n)}, t_n) + f(x^{(n+1)}, t_{n+1}) \right]$$

In both these scheme the $x^{(n+1)}$ depends on $x^{(n)}$ and $x^{(n-1)}$, hence they are called a multi-step method.

In this chapter, we will briefly describe all the above schemes. We start with Euler's forward method.

Conceptual questions

1. What are the importance of numerical solution of differential equations?
2. Relate the ODE solvers to the numerical methods of integration?
3. Why do the numerical solutions of ODEs have errors?
4. Derive an ODE solver that is based on 3/8 Simpson rule.

13.2 Euler Forward Method, Accuracy & Stability

Euler's forward method is the simplest ODE solver. In this section we will solve the ODE $\dot{x} = f(x, t)$ using this method. The initial condition for the ODE is $x(t=0) = x^{(0)}$.

As described in previous section, we discretize time into $N-1$ steps with markers at $t_0, t_1, \dots, t_n, t_{n+1}, \dots, t_{N-1}$. We time advance the ODE from t_0 to t_1 using

$$x^{(1)} = x^{(0)} + (\Delta t) f(x^{(0)}, t_0)$$

Similarly, we time advance the ODE from t_1 to t_2, \dots , from t_n to t_{n+1}, \dots , and finally, from t_{N-2} to t_{N-1} . The time stepping in the intermediate step is

$$x^{(n+1)} = x^{(n)} + (\Delta t) f(x^{(n)}, t_n)$$

The desired $x(N-1)$ is the value of the dependent variable at $t = t_{N-1}$.

Example 1: We solve $\dot{x} = -x$ numerically using Euler's forward scheme described above. We take $x(0) = 1$, final time = 10, and $(\Delta t) = 0.01$. The numerical result, shown in [Figure 64\(a\)](#), matches quite well with the exact result $\exp(-t)$, which is shown as red dashed line in the figure.

Example 2: We solve $\dot{z} = -i\pi z$ numerically using $z(0) = 1$, final time = 10, and $(\Delta t) = 0.01$. In [Figure 64\(b\)](#) we plot $\text{Im}(z)$ vs. $\text{Re}(z)$, where $z(t)$ is the numerically computed value of z at time t . The figure shows that $|z|$ increases with time. This is contrary to the exact solution, which is $z(t) = \cos(\pi t) + i \sin(\pi t)$. We will discuss the instability of the solution in later part of this section.

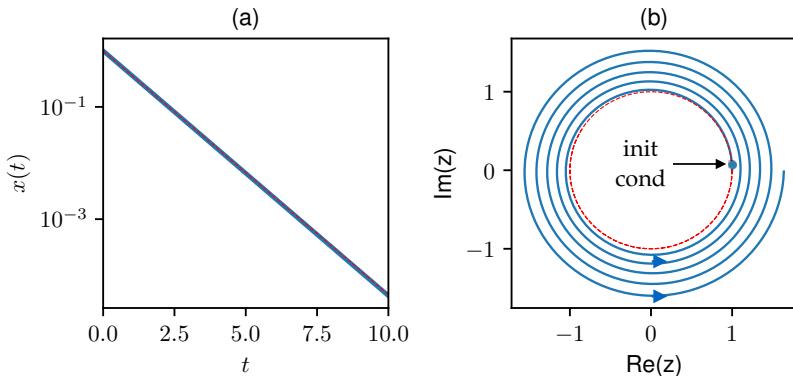


Figure 64: (a) Example 1: Plot of numerically-computed $x(t)$ vs. t (blue solid line). (b) Example 2: Plot of $\text{Im}(z)$ vs. $\text{Re}(z)$. In both the figures, the exact solutions are shown as red dashed curves.

Example 3: Let us take a nonlinear DE, $\dot{x} = -t \exp(-x)$. We solve it numerically using Euler's forward scheme with $x(0) = 1$, final time = 2, and $(\Delta t) = 0.01$. The exact solution of the above equation is $x(t) = \log(e-t^2/2)$. In [Figure 65\(a\)](#) we plot $x(t)$ vs. t for both exact and numerical solutions. Both the solutions are close to each other. We remark that the above solution is valid before the emergence of singularity at $t = \sqrt{2e}$.

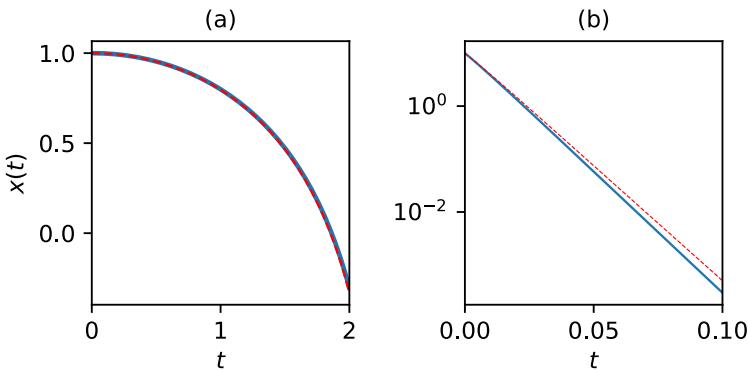


Figure 65: (a) Example 3 and (b) Example 4: Plots of numerically-

computed $x(t)$ vs. t as solid blue line and the exact solution as dashed red line.

Example 4: We solve another nonlinear DE $\dot{x} = x^2 - 100x$ using $x(0)=10$ and $(\Delta t) = 0.001$. The exact solution is $x(t) = 100 / (9 \exp(100t) - 1)$. In [Figure 65\(b\)](#) we exhibit the numerical and exact solutions.

Accuracy

Now we describe the error in Euler's forward method. An application Taylor series yields,

$$\begin{aligned} x^{(n+1)} &= x(t_n + \Delta t) = x(t_n) + (\Delta t)\dot{x}(t_n) + \frac{(\Delta t)^2}{2}\ddot{x}(t_n) + \dots \\ &= x^{(n)} + (\Delta t)f(x^{(n)}, t_n) + \frac{(\Delta t)^2}{2}\frac{df}{dt}|_{t_n} + \dots \quad \dots(26) \end{aligned}$$

Euler's forward method capture the first two terms of the expansion. Hence, for every time step, the error in this scheme is

$$\text{Error} = \varepsilon_n = (\frac{1}{2})(\Delta t)^2 \ddot{x}(x(n), t_n) \quad \dots(27)$$

Since the systematic errors for ODE solvers tend to add up, the error in N steps of the ODE is $(\frac{1}{2})N(\Delta t) \ddot{x}(x(n), t_n)$.

Example 5: Consider a DE $\dot{x} = \alpha x$ whose exact solution is $x(t) = x(0) \exp(\alpha t)$. In one step of this solution,

$$x^{(n+1)} = x^{(n)} \exp(\alpha(\Delta t)) = x^{(n)}(1 + \alpha(\Delta t) + (\frac{1}{2})(\alpha(\Delta t))^2 + \dots)$$

However, Euler's forward scheme yields $x^{(n+1)} = x^{(n)}(1 + \alpha(\Delta t))$. Thus, to leading order, the error in this scheme is $(\frac{1}{2})(\alpha(\Delta t))^2 x^{(n)}$, which is consistent with the formula of Eq. (27).

Stability of a DE is also an important concept. The notion of stability of DE's differs from that of a dynamical system. We illustrate this concept for a simple DE $\dot{x} = \alpha x$. The stability of nonlinear DEs will be discussed after that.

The equation $\dot{x} = \alpha x$ has an exact solution, which is $x(t) = x(0) \exp(\alpha t)$. For $\alpha < 0$, the solution converges to zero as $t \rightarrow \infty$. We expect

that the numerical solution to converge in a similar manner. However, this is not guaranteed as we show below.

The solution of Euler's forward method, $x^{(n)} = x^{(0)} (1+\alpha(\Delta t))^n$, converges to zero asymptotically as long as $(\Delta t) < 1 / |\alpha|$. However, for $|\alpha|(\Delta t) > 1$ or $(\Delta t) > 1 / |\alpha|$ with $\alpha < 0$, $x^{(n)}$ oscillates around zero with its magnitude growing with time, which is contrary to the exact solution. Such a numerical scheme where numerical solution diverges contrary to converging exact solution is called an *unstable scheme*. Note that for $\alpha > 0$, both exact and numerical solutions diverge.

Based on the stability issues, numerical schemes are classified into the following categories:

A Method is *stable* if it produces a bounded solution when the solution of the DE is bounded; otherwise it is *unstable*.

Conditionally stable: If the method is stable for a set of parameters and unstable for another set of parameters.

Unconditionally stable: If the method is stable for all parameter values.

Unconditionally unstable: If the method is unstable for all parameter values.

For complex α , exact solution is $x(t) = x(0) \exp(\alpha t)$; the amplitude $x(t)$ of grows with time when $\text{Re}(\alpha) > 0$ and decreases when $\text{Re}(\alpha) < 0$. The numerical solution by Euler's forward scheme, $x^{(n)} = x^{(0)} (1+\alpha(\Delta t))^n$, too behaves similar to the analytic solution, except for the case when $|1+\alpha(\Delta t)| > 1$ with $\text{Re}(\alpha) < 0$. In this case, the exact solution converges to zero, but the numerical solution diverges. Hence, the system is unstable for $|1+\alpha(\Delta t)| > 1$ when $\text{Re}(\alpha) < 0$. See [Figure 66](#) for an illustration.

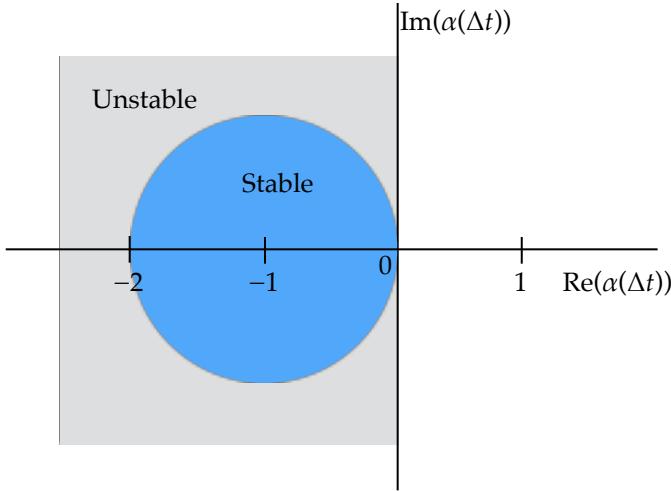


Figure 66: The stable region (blue circle) and unstable region (grey zone outside the circle) for Euler's forward method.

In the above discussion we dealt with stability of a linear equation $\dot{x} = \alpha x$. How is this result relevant to nonlinear equations, such as $\dot{x} = -t \exp(-x)$ of Example 3? For such equations, we linearize $f(x,t)$ around $(x^{(n)}, t_n)$ that yields

$$f(x, t) = f(x^{(n)}, t_n) + (x - x^{(n)}) \frac{\partial f}{\partial x} \Big|_{(x^{(n)}, t_n)} + (t - t_n) \frac{\partial f}{\partial t} \Big|_{(x^{(n)}, t_n)}$$

Now we shift the origin to $(x^{(n)}, t_n)$ using a transformation: $x' = x - x^{(n)}$ and $t' = t - t_n$. In terms of these variables, the differential equation is

$$\dot{x}' = \alpha x' + \beta + \gamma t'$$

where $\beta = f(x^{(n)}, t_n)$, and $\alpha = \partial f / \partial x$ and $\gamma = \partial f / \partial t$ are derivatives computed at $(x^{(n)}, t_n)$. Using the same arguments as for $\dot{x} = \alpha x$, we deduce that the equation $\dot{x}' = \alpha x' + \beta + \gamma t'$ would be locally unstable around $(x^{(n)}, t_n)$ when $\partial f / \partial x < 0$ and $|(\Delta t) \partial f / \partial x| > 1$.

Thus, we can deduce the stability criteria for linear as well as nonlinear equations. Note that the above linearization process is similar to what is done for the computation of Lyapunov exponent in

nonlinear dynamics.

Example 5: We analyse the stability of Euler's forward method for solving $\dot{x} = -100x$ ($\alpha = -100$). We solve the DE using $\Delta t = 0.021$ and 0.001 and initial condition $x(0)=10$. For $\Delta t = 0.021$, $\alpha h = -2.1$. Hence, $x^{(n)} = x^{(0)} (1+\alpha(\Delta t))^n$ oscillates around 0, as shown in Figure 67. For $h = 0.001$, we observe that $\alpha(\Delta t) = 0.1$, hence, the numerical solution is stable with the numerical result close to the exact result $10\exp(-100t)$.

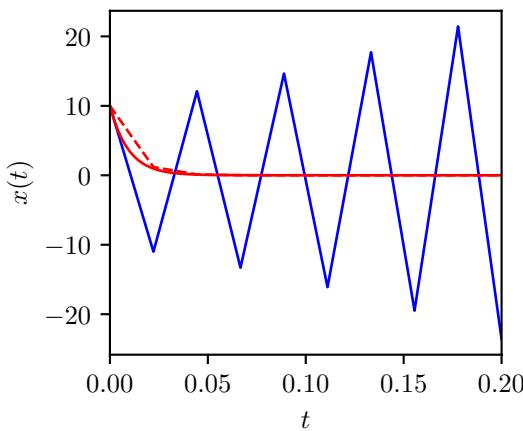


Figure 67: Solution of $\dot{x} = -100x$ using Euler's forward method using $(\Delta t) = 0.021$ (unstable, blue oscillatory curve) and 0.001 (stable, red curve). The exact result (dashed curve) is close to the solution for $(\Delta t) = 0.001$ (red solid curve).

Example 6: Let us explore the instability criteria for the Examples 1 to 4. In Table `#tab(Euler_stability_examples)` we list the regions of stability and instability. In Example 3, $\partial f/\partial x = t \exp(-x) > 0$. Hence, the system is unconditionally stable (stable for all Δt). For Example 4, $\partial f/\partial x = 2x - 100 \approx -100$ because x is small asymptotically. Therefore, Euler's forward method is stable for $\Delta t < 1/100$ and unstable for $\Delta t > 1/100$.

Thus, Euler's forward method is conditional stable for rows 1, 3, and 5; unconditionally unstable for row 2; and unconditionally stable for row 4.

Table !tab(Euler_stability_examples): Regions of stability and instability for Examples 1-4.

Equation	Stability regime	Instability regime
$\dot{x} = -x$	$\Delta t < 1$	$\Delta t > 1$
$\dot{z} = -i\pi z$	None	all h
$\dot{x} = \alpha x$	$\Delta t < 1 / \alpha $ (for $\alpha < 0$)	$\Delta t > 1 / \alpha $ (for $\alpha < 0$)
$\dot{x} = -t \exp(-x)$	all h	None
$\dot{x} = x^2 - 100x$	$\Delta t < 1 / 100$	$\Delta t > 1 / 100$

With this we end our discussion on Euler's forward method.

Conceptual questions

- What is meant by instability of an differential equation solver? How is this notion of stability different from dynamical instability.
- Argue that errors in Euler's forward method is of systematic type.

Exercises

- Solve the following differential equations using Euler's forward method. Compare your result with analytical one and estimate the error. State the parameter regimes of stability.
 - $\dot{x} = 5x$ with $x(0) = 1$
 - $\dot{x} = -10x$ with $x(0) = 1$
 - $\dot{x} = -x^2$ with $x(0) = 1$
 - $\dot{x} = \sin(t)$ with $x(0) = -1$
 - $\dot{x} = \cos^2(t)$ with $x(0) = 0$
 - $\dot{x} = x^{-2}$ with $x(0) = 1$
 - $\dot{x} = x^3 - 50x$ with $x(0) = 10$

13.3 *Implicit Schemes*

In this section we describe Euler's backward method and Trapezoid method that are implicit schemes.

Backward or implicit Euler Method

Euler's backward method to solve the differential equation $\dot{x} = f(x, t)$ is similar to Euler forward method, except one critical difference: To time advance from t_n to t_{n+1} we compute the slope at $t = t_{n+1}$. That is,

$$x^{(n+1)} = x^{(n)} + (\Delta t) f(x^{(n+1)}, t_{n+1}) \dots (28)$$

The above equation is an implicit equation because the right-hand-side (RHS) contains the unknown $x^{(n+1)}$. Hence, Euler's backward method is called an *implicit scheme*.

Accuracy: In order to estimate the error for this scheme, we expand $f(x^{(n+1)}, t_{n+1})$ of Eq. (28) around $t = t_n$ that yields

$$x^{(n+1)} = x^{(n)} + (\Delta t) f(x^{(n)}, t_n) + (\Delta t)^2 \frac{df}{dt} + \frac{(\Delta t)^3}{2} \frac{d^2 f}{dt^2} + \dots$$

Comparison of the above with the solution of Eq. (26) yields the error in Euler's backward scheme as

$$\text{Error} = -(\tfrac{1}{2})(\Delta t)^2 \ddot{x}(x(n), t_n) \dots (29)$$

which is just opposite of the error for Euler's forward method. Euler's backward method overestimates $x^{(n+1)}$, but Euler forward method underestimates it.

Stability: After this we explore the stability issues of Euler's backward method. We again start with the analysis of $\dot{x} = \alpha x$ and test the instability for real and negative α . For this equation,

$$x^{(n+1)} = x^{(n)} + \alpha(\Delta t)x^{(n+1)}$$

Therefore,

$$x^{(n+1)} = x^{(n)} / (1 - \alpha(\Delta t)).$$

For negative α , $1 / |(1 - \alpha(\Delta t))| < 1$. Hence, the system is unconditionally stable. For complex α with $\text{Re}(\alpha) < 0$, $1 / |(1 - \alpha(\Delta t))| = 1 / ((1 - \alpha_{\text{re}}(\Delta t))^2 + (\alpha_{\text{im}}(\Delta t))^2)^{1/2}$, which is also less than unity. Therefore, Euler's backward scheme is unconditionally stable for complex α as well. For nonlinear equations, a simple extension of the above analysis as in Section 13.2 shows that their treatment using Euler's backward method is also unconditionally stable.

We can rework all the examples of last section using Euler's backward method. The Examples 1 and 2 that are of the form $\dot{x} = \alpha x$ are straightforward to solve. For Examples 3 and 4, Euler's backward implementations are

$$x^{(n+1)} = x^{(n)} - \alpha(\Delta t) t_{n+1} \exp(-x^{(n+1)}) \dots (30)$$

$$x^{(n+1)} = x^{(n)} - \alpha(\Delta t) [(x^{(n+1)})^2 - 100 x^{(n+1)}] \dots (31)$$

The former equation is solved using one of the methods to be discussed in Section XXX. The latter equation is quadratic that can be solved easily; however, we need to choose the root for which $x^{(n+1)} \approx x^{(n)}$.

Example 1: We solve $\dot{x} = x^2 - 100x$ using Euler's backward method with $(\Delta t) = 0.02$. We take the initial condition as $x(0)=10$. As shown in [Figure 67](#), the numerical solution (solid line) is stable and is larger than the exact result (dashed line, $10\exp(-100t)$) for most part. Note that Euler's forward method is stable for $(\Delta t) < 0.01$. For time stepping, we solve Eq. (31) and take the following root; the other root is unsuitable for time advancement.

$$x^{(n+1)} = \frac{1}{2(\Delta t)} \left[(100(\Delta t) + 1) - \sqrt{(100(\Delta t) + 1)^2 - 4(\Delta t)x^{(n)}} \right]$$

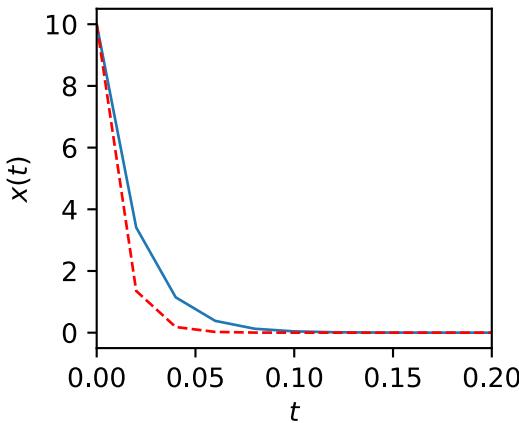


Figure 68: Solution of $\dot{x} = x^2 - 100x$ using Euler's backward method with $\Delta t = 0.02$ (blue curve). The exact result is shown as a dashed curve.

Trapezoid method

Now we discuss the accuracy and stability issues of Trapezoid method where the time stepping is performed using the following formula:

$$x^{(n+1)} = x^{(n)} + (\frac{1}{2})(\Delta t) [f(x^{(n)}, t_n) + f(x^{(n+1)}, t_{n+1})]$$

Since the above scheme is a combination of explicit and implicit schemes, hence it is called *semi-implicit method*.

In the above formula, an expansion of $f(x^{(n+1)}, t_{n+1})$ using Taylor series yields

$$x^{(n+1)} = x^{(n)} + (\Delta t) f(x^{(n)}, t_n) + \frac{(\Delta t)^2}{2} \frac{df}{dt} + \frac{(\Delta t)^3}{4} \frac{d^2f}{dt^2} + \dots$$

Thus, Trapezoid method is second order accurate with the leading order error as

$$\text{Error} = \left(\frac{(\Delta t)^3}{3!} - \frac{(\Delta t)^3}{4} \right) \ddot{x}(t) = -\frac{(\Delta t)^3}{12} \ddot{x}(t) \dots (32)$$

To analyse the stability of the above equation, we start with the equation $\dot{x} = \alpha x$. For this equation,

$$x^{(n+1)} = \left(\frac{1 + \alpha(\Delta t)/2}{1 - \alpha(\Delta t)/2} \right) x^{(n)}$$

It is easy to show that

$$\left| \frac{1 + \alpha(\Delta t)/2}{1 - \alpha(\Delta t)/2} \right| < 1$$

for real but negative α , as well as for complex α with $\text{Re}(\alpha) < 0$. However, the above quantity is unity for purely imaginary α . We thus prove that the trapezoid method is unconditionally stable.

In summary, trapezoid method is second order accurate and unconditionally stable. Hence, we can choose any value of Δt for integration. However, an implementation of implicit schemes poses difficulties for nonlinear equations.

It is important to note that stability and accuracy are two different things. Stability is a must, that is, we need to choose either a unconditionally stable method or an appropriate Δt for a conditionally stable method. The accuracy of solvers vary for different solvers. The choice of a solver and Δt depends on the requirement.

Conceptual questions

1. Show that the accuracies of forward and backward Euler methods are same.
2. What are the advantages and disadvantages of backward or implicit ODE schemes?

Exercises

1. Solve the following differential equations using Euler's backward method. Compare your result with analytical one.
 - $\dot{x} = 5x$ with $x(0) = 1$
 - $\dot{x} = -10x$ with $x(0) = 1$
 - $\dot{x} = x^{-2}$ with $x(0) = 1$
 - $\dot{x} = x^2 - 50x$ with $x(0) = 10$

2. Repeat Exercise 1 for trapezoid method.

13.4 Higher-order methods

A major advantage of implicit schemes described in the previous section is that they are stable. However, solving implicit equations poses challenges. This difficulty is partially surmounted in predictor-corrector and Runge-Kutta schemes where $x^{(n+1)}$ of the implicit term is estimated and plugged in the implicit equation. We describe these schemes below.

Predictor-Corrector (PC)

Implicit Scheme based on Trapezoid rule is

$$x^{(n+1)} = x^{(n)} + (\frac{1}{2}) (\Delta t) [f(x^{(n)}, t_n) + f(x^{(n+1)}, t_{n+1})] \dots (33)$$

The $x^{(n+1)}$ of the RHS of the above equation is *predicted* as follows:

$$\text{Predictor step: } x^{(n+1)*} = x^{(n)} + (\Delta t) f(x^{(n)}, t_n)$$

This value is now substituted for $x^{(n+1)}$ of the RHS of Eq. (33), that is,

$$\text{Corrector step: } x^{(n+1)} = x^{(n)} + (\frac{1}{2}) (\Delta t) [f(x^{(n)}, t_n) + f(x^{(n+1)*}, t_{n+1})]$$

The above scheme is accurate up to $O(h^2)$. The proof is exactly same as that given for trapezoid method. For the stability analysis, as usual, we solve the equation $\dot{x} = \alpha x$ using the above method that yields

$$x^{(n+1)} = (1 + \alpha(\Delta t) + (\alpha(\Delta t))^2 / 2) x^{(n)}$$

The inner region of the oval of Figure 69 is stable, while those outside with $\text{Re}(\alpha) < 0$ is unstable.

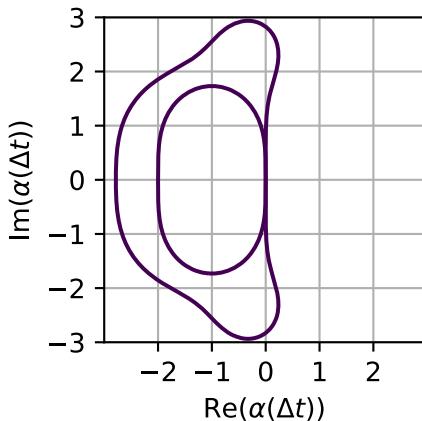


Figure 69: Stability regions for (a) the predictor-corrector and RK2 methods (oval region); (b) RK4 method (larger enclosure).

Runge-Kutta Methods

Runge-Kutta (RK) methods are similar to the predictor-corrector method in spirit. A major difference however is that RK methods make use of intermediate points between t_n to t_{n+1} . Here we describe the second-order and fourth-order RK methods.

Second-order RK method (RK2): A ODE solver based on mid-point rule is

$$x^{(n+1)} = x^{(n)} + (\Delta t) f(x^{(n+\frac{1}{2})}, t_{n+\frac{1}{2}}) \dots (34)$$

In RK2 method, the intermediate point $x^{(n+\frac{1}{2})}$ is estimated using Euler's forward method, that is,

$$k_1 = (\Delta t) f(x^{(n)}, t_n); \quad x^{(n+\frac{1}{2})*} = x^{(n)} + k_1 / 2$$

Now we substitute the above in Eq. (34) that yields

$$x^{(n+1)} = x^{(n)} + (\Delta t) f(x^{(n+\frac{1}{2})*}, t_{n+\frac{1}{2}}) \dots (35)$$

This is the RK2 method that involves two $f()$ computations per step. It is easy to show that this scheme has the same accuracy ($O((\Delta t)^2)$) and stability condition as the predictor-corrector method. See [Figure 69](#) for an illustration.

Third-order RK method (RK3): RK3 scheme involves more intermediate steps than RK2 method. The predictors for RK3 are

$$\begin{aligned} k_1 &= (\Delta t) f(x^{(n)}, t_n); \quad x^{(n+1/2)*} = x^{(n)} + k_1 / 2 \\ k_2 &= (\Delta t) f(x^{(n+1/2)*}, t_{n+1/2}); \quad x^{(n+1)**} = x^{(n)} - k_1 + 2 k_2 \\ k_3 &= (\Delta t) f(x^{(n+1)**}, t_{n+1}) \end{aligned}$$

Now, the corrector for RK3 is

$$x^{(n+1)} = x^{(n)} + (1/6)(k_1 + 2k_2 + 2k_3 + k_4)$$

Another version of RK3 is as follows:

$$\begin{aligned} k_1 &= (\Delta t) f(x^{(n)}, t_n); \quad x^{(n+1/3)*} = x^{(n)} + k_1 / 3 \\ k_2 &= (\Delta t) f(x^{(n+1/3)*}, t_{n+1/3}); \quad x^{(n+2/3)**} = x^{(n)} + (2/3) k_2 \\ k_3 &= (\Delta t) f(x^{(n+2/3)**}, t_{n+2/3}) \\ x^{(n+1)} &= x^{(n)} + (1/4)(k_1 + 3k_2) \end{aligned}$$

RK3 method is third-order accurate with error $O((\Delta t)^4)$. Note that RK3 involves three $f()$ computations per step.

Fourth-order RK method (RK4): The predictors for RK4 are

$$\begin{aligned} k_1 &= (\Delta t) f(x^{(n)}, t_n); \quad x^{(n+1/2)*} = x^{(n)} + k_1 / 2 \\ k_2 &= (\Delta t) f(x^{(n+1/2)*}, t_{n+1/2}); \quad x^{(n+1/2)**} = x^{(n)} + k_2 / 2 \\ k_3 &= (\Delta t) f(x^{(n+1/2)**}, t_{n+1/2}); \quad x^{(n+1)***} = x^{(n)} + k_3 \\ k_4 &= (\Delta t) f(x^{(n+1)***}, t_{n+1}) \end{aligned}$$

Now, the corrector for RK4 is

$$x^{(n+1)} = x^{(n)} + (1/6)(k_1 + 2k_2 + 2k_3 + k_4)$$

For the equation, $\dot{x} = ax$, RK4 method yields

$$x^{(n+1)} = \left(1 + \alpha(\Delta t) + \frac{1}{2}(\alpha(\Delta t))^2 + \frac{1}{6}(\alpha(\Delta t))^3 + \frac{1}{24}(\alpha(\Delta t))^4 \right) x^{(n)} + HOT \quad (36)$$

which shows that RK4 method is accurate up to the fourth order, and it has error $O((\Delta t)^5)$. RK4 method involves four $f()$ computations per step. Using Eq. (36) we can also construct the stability region in the ah plane. This region is depicted as an inverted D-shaped curve in [Figure 69](#).

ODE solver based on higher-order derivative

For the equation, $\dot{x} = f(x, t)$, using Taylor series, we can compute $x^{(n+1)}$ to whichever order we may choose:

$$x^{(n+1)} = x^{(n)} + (\Delta t)\dot{x} + \frac{(\Delta t)^2}{2}\ddot{x} + HOT$$

The second-order term is computed as

$$\ddot{x} = \frac{\partial f}{\partial t} + \dot{x} \frac{\partial f}{\partial x}$$

substitution of which in the series yields

$$x^{(n+1)} = x^{(n)} + \frac{(\Delta t)}{2} \left[2\dot{x} + (\Delta t) \frac{\partial f}{\partial t} + n(\Delta t)\dot{x} \frac{\partial f}{\partial x} \right] + HOT$$

which is accurate up to $O((\Delta t)^2)$.

Some of the multistep methods, e.g., Adam-Bashforth scheme, is derived using the above approach (see Chapra and Canale).

Using Python's *odeint*

Python has an *scipy.integrate* module whose function *odeint()* helps us

solve ODEs. Its usage is shown below. It takes the RHS function $f(x,t)$, initial condition, and time array (t), and it returns the values of the dependent variable $x(t)$ at each element of t .

```
from scipy.integrate import odeint

def f(x,t):
    return x**2-x

tinit = 0; tfinal=2.0; dt = 0.2
n = int((tfinal-tinit)/dt)+1
t = np.linspace(tinit, tfinal, n)

xinit = np.array(1.1)
x=odeint(f,xinit,t)
```

Example 1: We solve $\dot{x} = x^2 - x$ numerically in the interval $[0,2]$ using $x(0) = 1.1$ and $h = 0.2$. The numerical results for Euler forward, RK2, RK4 methods, and `ode_int()` are shown in Figure 70(a). RK4 and `ode_int` yield results close to the exact result, but RK2 and Euler methods have significant errors. The errors for Euler, RK2, and RK4 methods, shown in Figure 70(b), are proportional to h , h^2 , h^4 respectively, as expected.

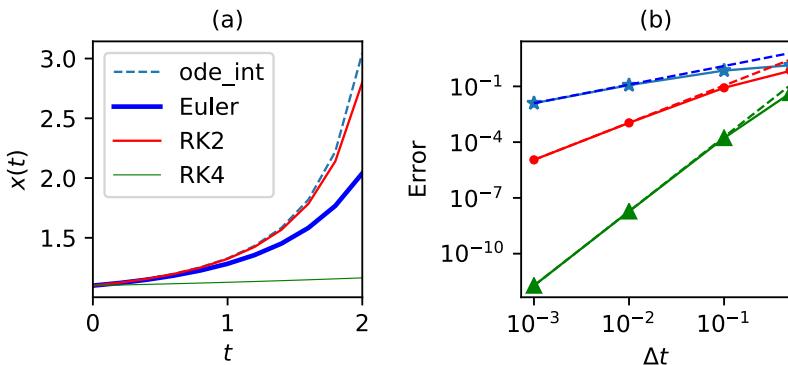


Figure 70: (a) Numerical solutions of $\dot{x} = x^2 - x$ using Euler forward, RK2, RK4 methods, and `ode_int()`. Here, $\Delta t = 0.2$ $x(0) = 1.1$ (b) Plots of errors for Euler (blue stars), RK2 (red dots), and RK4 (green triangles) methods for various Δt show that the errors for these schemes are proportional to Δt , $(\Delta t)^2$, and $(\Delta t)^4$ (dashed lines)

respectively.

Conceptual questions

1. What are the advantages and disadvantages of RK2 and RK4 methods over implicit schemes?

Exercises

1. Solve the following differential equations using predictor-corrector, RK2, and RK4 methods. Compare your result with analytical one.
 - $\dot{x} = 5x$ with $x(0) = 1$
 - $\dot{x} = -10x$ with $x(0) = 1$
 - $\dot{x} = x^{-2}$ with $x(0) = 1$
 - $\dot{x} = x^2 - 50x$ with $x(0) = 10$
2. Solve the equations of Exercise 1 for $h = 0.0001, 0.001, 0.01$, and 0.1 , and study the errors. How do the errors vary with h ?

13.5 Multistep Method

In this section we briefly describe several multistep methods. In such methods, $x^{(n+1)}$ depends on $x^{(n)}$, as well as on the values at earlier times, such as $x^{(n-1)}$, $x^{(n-2)}$, $x^{(n-3)}$. The name, multistep, comes due to this reason.

The simplest multistep method is leapfrog that will be described below.

leapfrog method

In leapfrog method, the time-stepping is done as

$$x^{(n+1)} = x^{(n-1)} + 2(\Delta t) f(x^{(n)}, t_n) \dots\dots(37)$$

The leapfrog method is very useful, specially for time-reversible systems because Eq. (37) is symmetric under time reversal ($t \rightarrow -t$). That is, if we wish to go from t_n to t_{n-1} , the prescription of leapfrog scheme is

$$x^{(n-1)} = x^{(n+1)} - 2(\Delta t) f(x^{(n)}, t_n)$$

which is same as Eq. (37). It is easy to verify that Euler's method, as well as RK2 and RK4 methods are not symmetric under time reversal. In Section 13.6 We will discuss this issue for Hamiltonian systems.

Still, there are certain critical issues with this method, as we describe below. For the equation $\dot{x} = \alpha x$, the equation becomes

$$x^{(n+1)} = x^{(n-1)} + 2\alpha(\Delta t) x^{(n)} \dots\dots(38)$$

whose solution is of the form ρ^n with ρ determined from the quadratic equation $\rho^2 = 1 + 2\alpha(\Delta t)\rho$. The solutions of the quadratic equation are

$$\rho_{1,2} = \alpha(\Delta t) \pm \sqrt{(\alpha(\Delta t))^2 + 1}$$

A general solution of Eq. (38) is

$$x^{(n)} = c_1 \rho_1^n + c_2 \rho_2^n$$

Where c_1 and c_2 are constants that are determined using initial condition. Among the two solutions, ρ_1 is genuine but ρ_2 is a spurious. This feature is evident from the Taylor expansion of $\rho_{1,2}$:

$$\begin{aligned}\rho_1 &\approx 1 + \alpha(\Delta t) + \frac{1}{2}(\alpha(\Delta t))^2 + HOT \\ \rho_2 &\approx -1 + \alpha(\Delta t) - \frac{1}{2}(\alpha(\Delta t))^2 + HOT\end{aligned}$$

where HOT represents higher order terms. For small αh , $\rho_2 \approx -1$, hence $(\rho_2)^n$ will induce oscillations in $x(t)$. This is a signature of the spurious solution. Thus, the leapfrog scheme is *unconditionally unstable* due to ρ_2 . However, if this instability due to ρ_2 could be suppressed (as described below), then the solution would be second-order accurate.

We can avoid ρ_2 in a following manner. A first-order ODE requires only one initial condition. Hence, we can tweak $x^{(0)}$ and $x^{(1)}$ so as to make $c_2 = 0$. For example, our $x^{(0)}$ and $x^{(1)}$ satisfy the following relations:

$$\begin{aligned}x^{(0)} &= c_1 + c_2 \\ x^{(1)} &= c_1\rho_1 + c_2\rho_2\end{aligned}$$

The constant c_2 automatically vanishes if $x^{(1)} = \rho_1 x^{(0)}$. This strategy can be used to turn off the spurious ρ_2 for the initial condition. Unfortunately, the spurious ρ_2 could reappear at a later stage due to instabilities arising out of round-off errors. To overcome this difficulty, following measures are recommended:

- Perform a step with another method often. It is better to use second-order scheme such as RK2 method so as to match the accuracy of leapfrog method.
- Once in a while, average the results of several successive steps to nullify the spurious oscillations.
- Test whether $x^{(n+1)} / x^{(n)} = \rho_1$. A violation of this equality signals the presence of the spurious component ρ_2 .

Once the spurious component of the solution has been suppressed, the leapfrog method works quite nicely. It is specially useful for solving time-reversible systems such as Hamiltonian systems, Schrödinger's

equation. We will address some of these systems later in this book.

In addition to the spurious component, the leapfrog method has several other disadvantages: (a) We need to store more variables ($x^{(N-2)}$ and $x^{(N-1)}$) if we need to restart the system from the final value. (b) For the leapfrog method, the initialisation procedure described above is more complex than Euler's method. For convenience, Euler's forward method is often used as a initial step. But, such a initialisation is prone to instability.

Adams-Bashforth method

In this scheme,

$$x^{(n+1)} = x^{(n)} + (\Delta t) \sum_{m=0}^{k-1} \beta_{k,m} f(x^{(n-m)}, t_{n-m})$$

with $\beta_{k,m}$ listed in Table [Chapra and Canale].

β	$m = 0$	$m = 1$	$m = 2$	$m = 3$	$m = 4$
$\beta_{1,m}$	1				
$2\beta_{2,m}$	3	-1			
$12\beta_{3,m}$	23	-16			
$24\beta_{4,m}$	55	-59	37	-9	
$720\beta_{5,m}$	1901	-2774	2616	-1274	252

As illustrations, we list the two lowest-order Adams-Bashforth schemes:

$$\begin{aligned} x^{(n+1)} &= x^{(n)} + 2(\Delta t) f(x^{(n)}, t_n) \\ x^{(n+1)} &= x^{(n)} + ((\Delta t)/2) [3f(x^{(n)}, t_n) - f(x^{(n-1)}, t_{n-1})] \end{aligned}$$

Adams-Moulton method

This scheme is very similar to Adams-Bashforth except that its RHS has a term dependent on $x^{(n+1)}$. In this scheme,

$$x^{(n+1)} = x^{(n)} + (\Delta t) \sum_{m=0}^{k-1} \beta_{k,m} f(x^{(n+1-m)}, t_{n+1-m})$$

With $\beta_{k,m}$ listed in Table [Chapra and Canale].

β	$m = 0$	$m = 1$	$m = 2$	$m = 3$	$m = 4$
$\beta_{1,m}$	1				
$2\beta_{2,m}$	1	1			
$12\beta_{3,m}$	5	8	-1		
$24\beta_{4,m}$	9	19	-5	1	
$720\beta_{5,m}$	251	646	-264	106	-19

The three lowest-order Adams-Moulton schemes are listed below:

$$\begin{aligned} x^{(n+1)} &= x^{(n)} + 2(\Delta t) f(x^{(n+1)}, t_{n+1}) \\ x^{(n+1)} &= x^{(n)} + ((\Delta t)/2) [f(x^{(n+1)}, t_{n+1}) + f(x^{(n)}, t_n)] \\ x^{(n+1)} &= x^{(n)} + ((\Delta t)/12) [5 f(x^{(n+1)}, t_{n+1}) + 8 f(x^{(n)}, t_n) - f(x^{(n-1)}, t_{n-1})] \end{aligned}$$

Conceptual questions

1. Why is leapfrog method to solve ODEs unstable?
2. What are the benefits and disadvantages of multistep methods?

Exercises

1. Solve the following differential equations using leapfrog and second-order Adams-Basforth methods. Compare your result with analytical one.
 - $\dot{x} = 5x$ with $x(0) = 1$
 - $\dot{x} = -10x$ with $x(0) = 1$
 - $\dot{x} = x^{-2}$ with $x(0) = 1$
 - $\dot{x} = x^2 - 50x$ with $x(0) = 10$

13.6 Solving a System of Equations

So far we solved a single first-order ODE. However, in practice, we encounter a set of ODEs. For example, chemical reactions with N species are described using N rate equations, which are first-order ODEs. The dynamics of N particles is described by $3N$ second-order ODEs. In this chapter we discuss how to numerically solve such a set of ODEs.

Solving a set of ODEs

We consider N first-order ODEs whose independent variable is t and dependent variables are x_0, x_1, \dots, x_{N-1} . The i^{th} equation of the set is written schematically as

$$\dot{x}_i = f_i(t, x_0, x_1, \dots, x_{N-1})$$

where the RHS function $f_i(x_j)$ could be linear or nonlinear functions of the variables, x_0, x_1, \dots, x_{N-1} , and t .

To solve the set of equations given initial condition $\{x_i(t=0)\}$, we discretize time into many intervals and evolve the equations from $t = 0$ to the final time. For example, if the dependent variables are $\{x_i^{(n)}\}$ at time $t = t_n$, then we time advance the variables using Euler's forward method as follows:

$$x_i^{(n+1)} = x_i^{(n)} + (\Delta t) f_i(t, x_0^{(n)}, x_1^{(n)}, \dots, x_{N-1}^{(n)})$$

where $(\Delta t) = t_{n+1} - t_n$. It is straight forward to compute the RHS and hence $\{x_i^{(n+1)}\}$. Other schemes such as RK2 can be implemented in a similar manner.

Time stepping equation in Euler's backward method, an implicit scheme, yields the following N coupled equations that could be either nonlinear or linear:

$$x_i^{(n+1)} = x_i^{(n)} + (\Delta t) f_i(t, x_0^{(n+1)}, x_1^{(n+1)}, \dots, x_{N-1}^{(n+1)})$$

Solving such a set of equations could be challenging. In Chapters xxx we will discuss how to solve such equations.

Now we will discuss how to solve coupled equations of mechanics.

Solving equations of motion in mechanics

The equation of motion of a particle moving in 1D is $m\ddot{x} = f(x, \dot{x}, t)$, where m is the mass of the particle; x , \dot{x} , \ddot{x} are respectively particle's position, velocity, and acceleration; and $f(x, \dot{x}, t)$ is the force on the particle. We can convert the above second-order ODE to the following two first-order equations:

$$m\dot{x} = p; \quad \dot{p} = f(x, p, t)$$

where p is the momentum of the particle. We can solve the above two equations using the methods described in this chapter. For a class of systems for which the force $f(x) = -dV/dx$, the total energy, $p^2/2m + V(x)$, is constant in time; such systems are called Hamiltonian or energy conserving systems.

We take a concrete example of a simple harmonic oscillator for which the force is $-x$ (ignoring the prefactor). For convenience, we have set $m = 1$. Hence, the equations of motion of an oscillator are

$$\dot{x} = p; \quad \dot{p} = -x \quad \dots(39)$$

This system is a Hamiltonian system. We can solve the above equations using Euler's forward scheme as follows:

$$x^{(n+1)} = x^{(n)} + (\Delta t) p^{(n)}; \quad p^{(n+1)} = p^{(n)} - (\Delta t) x^{(n)} \quad \dots(40)$$

Even though Euler's forward method yields an approximate solution to the oscillator with sufficiently small h , it has certain deficiencies. Equation (39) is symmetric under time reversal, but the Euler's time-stepping scheme is not (see Eq. (40)).

Fortunately, a modified version of leapfrog method discussed in Section 13.5 is suitable for solving equations that are time-reversal symmetric. In *leapfrog method*, the position x and the momentum p are computed at staggered time grid, as shown in [Figure 71](#).

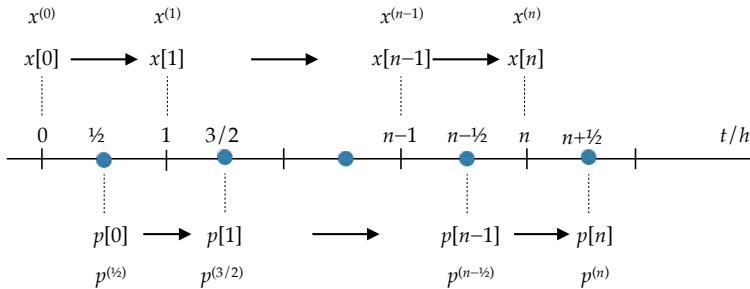


Figure 71: In the leapfrog method, $x(t)$ are computed at $t = (\Delta t), 2(\Delta t), \dots, n(\Delta t), \dots$, while $p(t)$ are computed at $(\Delta t)/2, 3(\Delta t)/2, \dots, (n+\frac{1}{2})(\Delta t), \dots$. For the Python program, we store these variables in arrays $x[0:N-1]$ and $p[0:N-1]$ as shown.

In the first timestep, we compute

$$p^{(1/2)} = p^{(0)} - ((\Delta t)/2) x^{(0)}$$

After this, we time advance both x and p using the following method:

$$x^{(n)} = x^{(n-1)} + (\Delta t) p^{(n-1/2)}; \quad p^{(n+1/2)} = p^{(n-1/2)} - (\Delta t) x^{(n)} \quad \dots \dots (41)$$

The following Python function implements the above scheme. For the computation of energy, we interpolate the velocity field at the integer grid points of [Figure 71](#).

```
def f(x,v):
    return -x # force for the oscillator
def leap_frog(f, tinit, tfinal, dt, initcond):
    n = int((tfinal-tinit)/dt)+1 # n divisions
    t = np.linspace(tinit, tfinal, n)
    y = np.zeros((n,2))
    y[0][0] = initcond[0];
    y[0][1] = initcond[1] + f(y[0][0],y[0][1])*dt/2
    # init vel at t=1/2

    for k in range(1,n):
        y[k][0] = y[k-1][0] + y[k-1][1]*dt
        y[k][1] = y[k-1][1] + f(y[k][0],y[k][1])*dt
```

```
return t, y
```

In [Figure 72\(a\)](#) we exhibit the time series of $x(t)$ and $p(t)$ for initial condition $x(0) = 1$ and $p(0) = 0$ with $\Delta t = 0.001$. To check the correctness of our solution, we compute the total energy, $(x^2 + p^2)/2$, whose exact value is $\frac{1}{2}$ throughout the trajectory. As shown in [Figure 72\(b\)](#), the error in the energy of the order of 10^{-7} ; this error is due to the truncation error in the leapfrog method (see the Taylor series expansion in Section 13.5). In addition, the error is oscillatory, unlike that for Euler's forward method for which the error grows with time. For the same h , the error in Euler's method is approximately 10^4 times larger. We also remark that a more advanced scheme, called PFERL method, provides more accurate results for energy conserving systems.

According to the leapfrog method, we go from t_{n+1} to t_n using the following steps:

$$x^{(n)} = x^{(n+1)} - (\Delta t) p^{(n+\frac{1}{2})}; \quad p^{(n-\frac{1}{2})} = p^{(n+\frac{1}{2})} + (\Delta t) x^{(n)} \quad \dots\dots (42)$$

which are same as Eqs. (41). Thus, the time-advance equations for leapfrog method are time-reversal symmetric. We demonstrate this feature in [Figure 72\(a\)](#) where the time-reversed time series (from $t = \pi$ to $t = 0$), represented using dashed curves, overlaps with the forward time series.

This feature is responsible for the approximate energy conservation observed for leap frog method. Another advantage of the leapfrog method is the it preserves the phase space volume during the evolution.

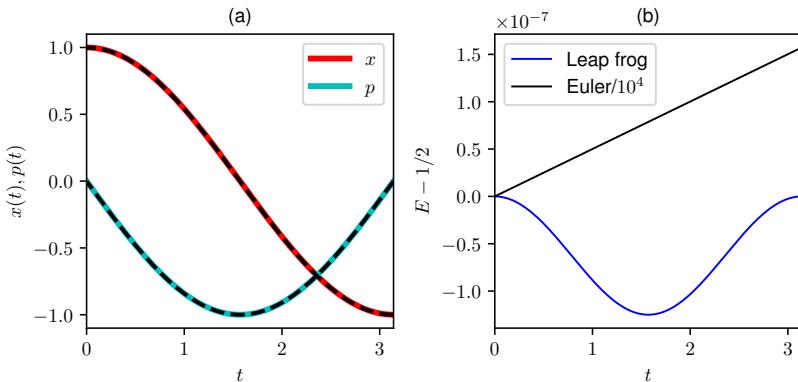


Figure 72: For a simple harmonic oscillator: (a) Plots of $x(t)$ and $p(t)$ computed using the leap frog method. The time-reversed $x(t)$ and $p(t)$ are shown as dashed black curves. (b) The error in energy, $E - 1/2$, for the leapfrog and Euler methods.

Phase space plot, which is the plot of x vs. p , provides useful insights into the system dynamics. In [Figure 73](#), we present the phase space plots of simple oscillator, and that of Duffing oscillator whose equation of motion is $\ddot{x} = x - x^3$. The three difference curves in [Figure 73\(b\)](#) are obtained by starting the ODEs with initial conditions $(x[0], p[0]) = (0.5, 0.5), (0.4, 0)$, and $(-0.4, 0)$.

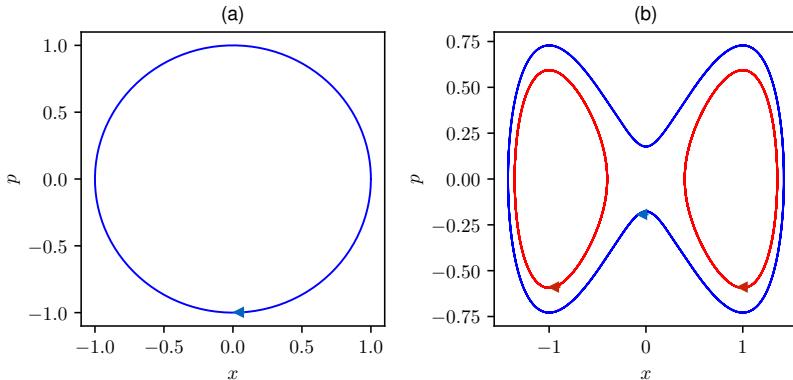


Figure 73: Phase plots (a) of a simple oscillator ($\ddot{x} = -x$) and (b) of a Duffing oscillator ($\ddot{x} = x - x^3$).

It is important to note that in the phase space, the trajectories computed using Eq. (42) are same as the forward-time trajectories, except that they move in anticlockwise direction. Such trajectories are unphysical. The corresponding physical trajectories are obtained by making a transformation $p \rightarrow -p$ (or flipping the trajectories around $p = 0$ axis). For Hamiltonian or energy conserving systems, such trajectories are indeed observed in reality. This is another statement of time reversal symmetry.

There are many equations that are not symmetric under time reversal. A simple example of such equation is $\dot{x} = x$, that has a growing solution $x(t) = \exp(t)$. Under the transformation, $t \rightarrow -t$, the equation gets transformed to $\dot{x} = -x$, whose solution $x(t) = \exp(-t)$ decreases with time. Hence, the original equation and the time-reversed equation, as well as their solutions, are very different. It can be verified that the phase space trajectories of the two systems are also very different.

For solving mechanical systems, we could also use `odeint()` function to solve a system of equation. For example, a code segment for solving oscillator problem using `odeint()` is as follows:

```
def ydot_osc(y,t):
    return ([y[1], -y[0]])

yinit = np.array([1, 0])
y=odeint(ydot_osc,yinit,t)
```

Conceptual questions

1. List examples of physical systems that are described by a set of ODEs.
2. Prove that the total energy is conserved for simple oscillator. Show that the equations are time-reversal symmetric.

Exercises

1. Using leapfrog method solve the nondimensionalized forced oscillator $\frac{d^2x'}{dt'^2} + x' = \cos\left(\frac{\omega_f}{\omega_0}t'\right)$, which is Eq. (8) of Section 9.2. What are the solutions for the oscillator whose parameters are $m = \frac{1}{2}$ kg, $\omega_0 = 10$ Hz, $\omega_f = 1$ Hz, and $F_0 = 0.1$ N. How does the result change if $\omega_0 = 0.1$ Hz.
2. Solve the equation for a damped oscillator without forcing. Choose appropriate parameters. Draw the phase space plots. It is better to use nondimensional equation.
3. Repeat Exercise 2 for forced damped oscillator.
4. Construct phase space plots for the Duffing oscillator whose

- equation of motion is $\ddot{x} = x - x^3$.
5. Solve $\ddot{x} = -x + x^3 + \sin(2t)$ using RK2 method for different initial conditions.
 6. Solve Newon's equation of motion for a pendulum with a massless string of length l and point bob of mass m . Plot the angle and angular velocity as a function of time. Also make the phase space plot. It is better to use non-dimensional equation and leapfrog method.
 7. Solve Lorenz equation numerically. The equations are $\dot{x} = P(y-x)$; $\dot{y} = x(r-z) - y$; $\dot{z} = xy - \beta z$. Make 3D plots of (x, y, z) for three sets of parameters: (1) $P = 10, r = 0.5, \beta = 1$; (2) $P = 10, r = 2, \beta = 1$; (3) $P = 10, r = 28, \beta = 1$.

13.7 Stiff Equations

Many natural and engineering systems involve several or many length and time scales. For example, a damped oscillator has three time scales: damping time scale, time period of the oscillator, and time period of the forcing. Atmospheric flows too have length scales ranging from meters to thousands of kilometres. A turbulent flow has vortex structures at all scales.

Thus, the differential equations of multiscale systems has many timescales. The equation whose the lowest timescale and the largest timescale are far apart are called *stiff equations*. Numerical simulation of such equations is quite challenging, which is the topic of this section.

Solving $\dot{x} = x^2 - 10^6x$

Let us consider an ODE: $\dot{x} = x^2 - 10^6x$, which is a variant of Example 3 of Section 13.2. The timescales of this equation are $1/x$ and 10^{-6} . When x is $O(1)$, the two timescales are far apart, hence the system is *stiff*. Here, the linear term with a very short time scale is the stiff term. For this case, the numerical implementation of Euler's forward method is conditionally stable for $\Delta t < 10^{-6}$, which is too small for comfort. This difficulty is overcome using two methods.

Semi-implicit scheme: We semploy a semi-implicit scheme for the stiff term that yields

$$x^{(n+1)} = x^{(n)} + (\Delta t)(x^{(n)})^2 - \frac{\Delta t}{2}10^6[x^{(n)} + x^{(n+1)}]$$

whose solution is

$$x^{(n+1)} = \left(\frac{1 - 10^6(\Delta t)/2}{1 + 10^6(\Delta t)/2} \right) x^{(n)} + h(x^{(n)})^2$$

which is unconditionally stable. The above equation is stable for any Δt ; we do not need tiny Δt (10^{-6} or lower).

Exponential trick: We can combine the stiff term $-ax$ with \dot{x} using a change of variable: $x' = x \exp(at)$. In terms of x' ,

$$\dot{x}' = x^2 \exp(at)$$

that can be solved using $x'^{(n+1)} = x'^{(n)} + (\Delta t) (x'^{(n)})^2 \exp(at_n)$ or

$$x'^{(n+1)} = \left[x'^{(n)} + (\Delta t) (x'^{(n)})^2 \right] \exp(-a(\Delta t))$$

The above equation can have any Δt for time advancement, not tiny Δt as in the original equation.

Solving a set of equation

Let us consider two linear equations:

$$\dot{x} + ax = y; \quad \dot{y} = -y$$

$$\text{or, } \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} -a & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The eigenvalues of the matrix in the above equation are $-a$ and -1 . Hence, the timescales of the above equations are $1/a$ and 1 . When a is very large, the timescales $1/a$ and 1 are widely separated, thus making the above system stiff. For Euler forward method, the above equations are stable only for $\Delta t < 1/a$ that could be very demanding.

We can employ the strategies described for $\dot{x} = x^2 - 10^6 x$ to overcome the above difficulty. We employ semi-implicit scheme to stiff term that yields

$$x^{(n+1)} = x^{(n)} - a(\Delta t) \left(\frac{x^{(n)} + x^{(n+1)}}{2} \right) + (\Delta t)y^{(n)}; \quad y^{(n+1)} = (1 - \Delta t)y^{(n)}$$

or

$$x^{(n+1)} = \left(\frac{1 - a(\Delta t)/2}{1 + a(\Delta t)/2} \right) x^{(n)} + (\Delta t)y^{(n)}; \quad y^{(n+1)} = (1 - \Delta t)y^{(n)}$$

The above equations are stable for $\Delta t < 1$, which is a major gain

compared to $h = 1/a$ required for Euler forward method.

We can also employ the exponential trick to the above set of equations. Using the change of variable, $x' = x \exp(at)$, the equation $\dot{x} + ax = y$ is transformed to

$$\dot{x}' = y \exp(at)$$

Whose solution is

$$x^{(n+1)} = [x^{(n)} + (\Delta t) y^{(n)}] \exp(\alpha(\Delta t))$$

The other equation, $\dot{y} = -y$, is easily time advanced using $y^{(n+1)} = (1 - \Delta t) y^{(n)}$. Clearly, the new set of equations are unconditionally stable.

This is how we solve the stiff equations numerically. With this we end this long chapter on ODE solvers.

Conceptual questions

1. Why solving an equation with two very different timescales is difficult using Euler's forward method? How do we solve such equations.

Exercises

1. Solve the equation $\dot{x} = x^2 - 10^6 x$ using the methods described in this section.
2. Solve the equation $\dot{x} + ax = y; \quad \dot{y} = -y$ for $a = 10^6$ using the methods described in this section.
3. Solve the equation numerically for an unforced oscillator when the viscous time scale is much larger than the time period of the oscillator.
4. Solve the equation $\dot{x} = \sin(t) - 10^5 x$ using the methods described in this section.

CHAPTER FOURTEEN
FOURIER TRANSFORM AND SPECTRAL METHODS

14.1 Fourier Transform

Many natural phenomena have multiple scales. For example, Earth's atmospheric flow has structures whose size range from meter to thousands of kilometres. Stellar and galactic flows have even wider range of scales. Fourier transform is one of the tools to analyse such multiscale systems.

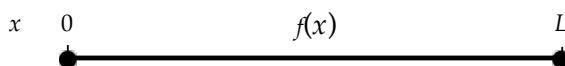
In this chapter we will first introduce Fourier transform. After later sections, we will use Fourier transform to solve various partial differential equations.

One-dimensional Fourier Transform

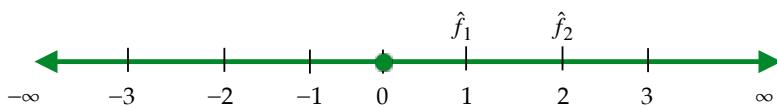
We consider a periodic, piece-wise continuous, and differentiable function $f(x)$ of period L , that is $f(x+L) = f(x)$. In 1807, Fourier showed that such a function can be expanded as the following infinite series:

$$f(x) = \sum_{-\infty}^{\infty} \hat{f}_n \exp(i k_n x) \dots \dots (43)$$

where $i = \sqrt{-1}$; $k_n = 2n\pi/L$ with n as an integer from $-\infty$ to $+\infty$; and \hat{f}_n is the *Fourier coefficient* or *Fourier mode*. The constants k_n 's are called *wavenumbers*. See Figure 74 for an illustration. In the following discussion, we assume that $f(x)$ could be real or complex. However, \hat{f}_n is always complex.



(a) real space



(b) Fourier space

Figure 74: Fourier transform of $f(x)$ to \hat{f}_n . $f(x)$ is a periodic function with period of L . The index of \hat{f}_n is an integer that takes values from $-\infty$ to ∞ .

The functions $\{\exp(ik_nx)\}$ form an *orthogonal basis* (just as orthogonal polynomials), that is,

$$\int_0^L \exp(ik_n x) \exp(-ik_m x) dx = L \delta_{nm}$$

where *Kronecker delta function*, $\delta_{nm} = 1$ if $n = m$ and 0 otherwise. Using the above orthogonality property, we can invert Eq. (43) as follows:

$$\hat{f}_n = \frac{1}{L} \int_0^L f(x) \exp(-ik_n x) dx \dots\dots (44)$$

In literature, the equations (44) and (43) are called *forward Fourier transform* (real to Fourier) and *inverse Fourier transforms* (Fourier to real). The space of wavenumbers is called *wavenumber space*, *Fourier space*, or *spectral space*. Note that representations of the function in real and spectral spaces are equivalent.

The quantity $[f(x)]^2/2$ is the *energy density*, and its integral over the whole space provides the *total energy* of the system. The quantity $|\hat{f}_n|^2/2$ is called the *modal energy*. Parseval's theorem relates the real space energy to the modal energy as follow.

Parseval's theorem: The average energy in real space equals the sum of modal energy of all the Fourier modes. That is,

$$\frac{1}{L} \int_0^L dx \frac{1}{2} (f(x))^2 = \sum_{-\infty}^{\infty} \frac{1}{2} |\hat{f}_n|^2$$

Example 1: We consider a box of length 2π and compute the Fourier transform of $\cos(3x) + i \sin(3x)$, $\sin(2x)$, $4\cos(4x)$, $8\cos^2(x)$ using Eq. (44). Note that the functions are periodic in the domain. The results are

For $\cos(3x) + i \sin(3x)$: $\hat{f}_n = \delta_{(3)n}$

$$\text{For } \sin(2x): \hat{f}_n = \frac{1}{2i}(\delta_{(2)n} - \delta_{(-2)n})$$

$$\text{For } 4\cos(4x): \hat{f}_n = 2(\delta_{(4)n} - \delta_{(-4)n})$$

$$\text{For } 8\cos^2(x): \hat{f}_n = 4\delta_{(0)n} + 2(\delta_{(2)n} - \delta_{(-2)n})$$

The average energy of the functions are $\frac{1}{2}$, $\frac{1}{4}$, 4, and 12 respectively, both in real and Fourier space.

Example 2: We compute the Fourier transform of $f(x) = \exp(-(x/a)^2/2)$, which is not a periodic function of x , using a trick. We consider $f(x)$ to be in a box of size L with $L \gg a$ and repeat this box on both sides (see Figure 75).

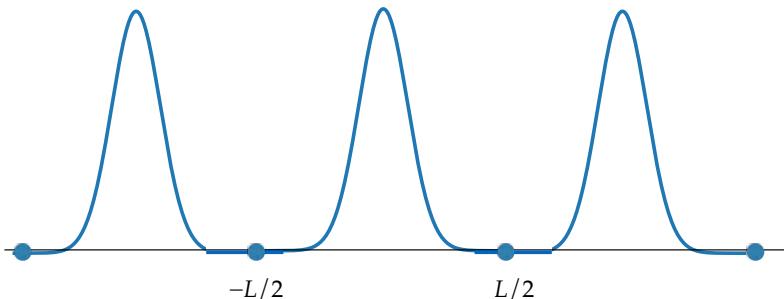


Figure 75: To compute the Fourier transform of a nonperiodic function $f(x) = \exp(-(x/a)^2/2)$, we construct $f(x)$ in a large box ($L \gg a$) and repeat it on both sides. We take Fourier transform of this periodic function.

Under these circumstances,

$$\begin{aligned}\hat{f}_n &= \frac{1}{L} \int_{-L/2}^{L/2} \exp[-\frac{1}{2}(x/a)^2] \exp(-ik_n x) dx \\ &= \frac{1}{L} \exp[-\frac{1}{2}(k_n a)^2] \int_{-L/2}^{L/2} \exp[-(x/a + ik_n a)^2/2] dx \quad \dots\dots (45)\end{aligned}$$

Numerical Fourier transform will yield the above result. Analytical form of the above integral is quite complex (an error function).

For analytical calculations, we take $L \rightarrow \infty$, for which k_n becomes a continuum and the sum of Eq. (43) is replaced by an integral. We perform the following transformation: $k_n \rightarrow k$; $\sum \rightarrow \int dk$ and $L \hat{f}_n = \hat{f}(k) \sqrt{2\pi}$ that leads to the following redefinitions of Fourier transforms:

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(k) \exp(ikx) dk \quad \dots\dots(46)$$

$$\hat{f}(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) \exp(-ikx) dx \quad \dots\dots(47)$$

Using this definition, the Fourier transform of $\exp(-(x/a)^2/2)$ gets simplified and yields

$$\hat{f}(k) = a \exp[-\frac{1}{2}(ka)^2] \quad \dots\dots(48)$$

In mathematics, Eqs. (46, 47) are the equation of Fourier transform, while Eq. (43) is called Fourier series. However, in this book, following the convention of computational science, we will call Eqs. (43, 44) Fourier transforms.

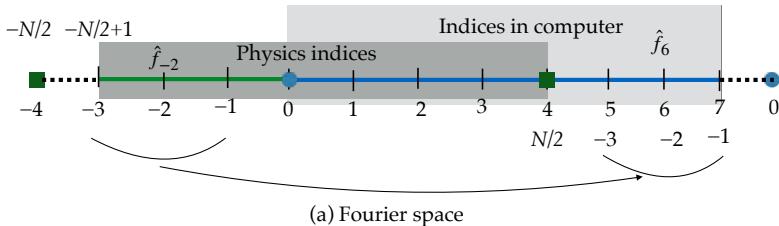
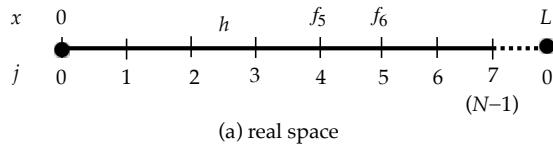
1D Discrete Fourier Transform (DFT)

For numerical computations, the discretize $f(x)$ and convert the integral of Eq. (44) to a sum. We divide the interval $[0, L]$ into N segments with the real space points at $x_j = jL/N$ with $j = 0:N-1$. Periodicity of the function implies that $x_N = x_0$, hence, $f(x_N)$ is not stored. See [Figure 76](#). The relationship between $\{f_j\}$ and $\{\hat{f}_n\}$ is linear with determinant being nonzero. Hence, there should be N Fourier coefficients with wavenumbers ranging from $-N/2+1$ to $N/2$. With the above arrangements, the definition of the transforms, called *Discrete Fourier Transform (DFT)*, are

$$\text{Inverse DFT: } f_j = \sum_{-N/2+1}^{N/2} \hat{f}_n \exp \left[i \frac{2\pi j n}{N} \right] \quad \dots\dots(49)$$

$$\text{Forward DFT: } \hat{f}_n = \frac{1}{N} \sum_{j=0}^{N-1} f_j \exp \left[i \frac{-2\pi j n}{N} \right] \dots\dots(50)$$

It is easy to verify that the Fourier coefficients satisfy the property, $\hat{f}_n = \hat{f}_{n+N}$. Using this property we shift the Fourier coefficients for $n = (-N/2+1):(-1)$ to $(N/2+1):(N-1)$. With this arrangement, the Fourier coefficients are conveniently saved in an array with indices 0 to $N-1$. See [Figure 76](#) for $N = 8$. Also, note that among the Fourier modes, \hat{f}_0 and $\hat{f}_{N/2}$ are real; in fact, \hat{f}_0 is average of f_j 's.



[Figure 76:](#) Storage of $\{f_j\}$ and $\{\hat{f}_n\}$ for $N = 8$, for which $f_0 = f_8$. Python stores \hat{f}_n in an array with indices 0:7; the modes $\hat{f}_{-3}, \dots, \hat{f}_{-1}$ are transferred to $\hat{f}_5, \dots, \hat{f}_7$.

In a special case when $f(x)$ is real, $\{\hat{f}_n\}$ satisfy the relation: $\hat{f}_{-n} = \hat{f}_n^*$. Hence, the coefficients for positive n 's are stored.

For DFT, Parseval's theorem translates to

$$\frac{1}{N} \sum_j \frac{1}{2} |f_j|^2 = \sum_n \frac{1}{2} |\hat{f}_n|^2$$

Forward and inverse DFTs prescribed above require N^2 addition and N^2 multiplications (N for each mode $\times N$ modes). In 1965, Cooley and Tukey devised an optimised algorithm, called *Fast Fourier Transform (FFT)*, that can perform the above task using $O(N \log(N))$ floating-point

operations. This is a big saving for very large N . Python employs FFT for Fourier transforms.

Python's numpy module offers functions listed in [Table !tab\(1DFFT\)](#). Note however that Python *fft* and *ifft* functions have opposite normalisation, which is

$$\begin{aligned} \text{ifft: } \hat{f}_j &= \frac{1}{N} \sum_{n=-N/2+1}^{N/2} \hat{f}_n \exp \left[i \frac{2\pi j n}{N} \right] \\ \text{fft: } \hat{f}_n &= \sum_{j=0}^{N-1} f_j \exp \left[i \frac{-2\pi j n}{N} \right] \end{aligned}$$

Hence, we need to incorporate this factor after the evaluation of the function.

[Table !tab\(1DFFT\)](#): Python's 1D FFT functions of numpy module

Function	Operation
<i>fft(f,N)</i>	Forward FFT (of complex f of size N)
<i>ifft(fk,N)</i>	Inverse FFT (of fk of size N)
<i>rfft(f,N)</i>	Forward FFT (of real f of size N)
<i>irfft(fk,N)</i>	Inverse FFT (of fk to real f ; size = N)

Example 3: We compute DFT of $\cos(3x) + i \sin(3x)$, $\sin(2x)$, $4\cos(4x)$, $8\cos^2(x)$ in a box of length 2π . We divide the box 2π into 15 segments and compute $\{f_j\}$ at those grid points 0:15. We employ *fft* for the first function, and *rfft* for the rest of them. The results are same as that of Example 2, as expected. A code segment for fft computation is as follows:

```
L = 2*np.pi; N = 16; dx = L/N
j = np.arange(0,N)
x = j*dx
y = np.cos(3*x) + 1j*np.sin(3*x)
yk = np.fft.fft(y,N)/N
```

Mutlidimensional-dimensional Fourier Transform

It is straightforward to generalise 1D Fourier transform to multidimensional Fourier transforms. We consider a function $f(\mathbf{x})$ in a box (L_1, L_2, \dots, L_d) where d is the space dimension. Note that the

argument $\mathbf{x} = (x_1, x_2, \dots, x_d)$ is a vector.

The function $f(\mathbf{x})$ can be expanded in terms of Fourier basis as follows:

$$\text{Inverse transform: } f(\mathbf{x}) = \sum_{\mathbf{n}} \hat{f}_{\mathbf{n}} \exp(i \mathbf{k} \cdot \mathbf{x})$$

where the wavenumber component $k_l = 2\pi n_l / L_l$ with $l = 1:d$; and $\mathbf{n} = (n_1, n_1, \dots, n_l, \dots, n_d)$ is lattice vector with n_l taking values from $-\infty$ to $+\infty$. An inversion of the above equation yields

$$\text{Forward transform: } \hat{f}_{\mathbf{n}} = \frac{1}{\prod L_j} \int d\mathbf{x} f(\mathbf{x}) \exp(-i \mathbf{k} \cdot \mathbf{x})$$

For the discrete Fourier transform, we discretize the real space into $\prod N_j$ small blocks with N_1 section along x_1 , N_2 section along x_2, \dots, N_d section along x_d . Hence, at the lattice point (j_1, j_2, \dots, j_d) ,

$$\mathbf{x} = \{(L_1/N_1)j_1, (L_2/N_2)j_2, \dots, (L_d/N_d)j_d\}$$

With this notation, multi-dimensional transform is

$$\begin{aligned} \text{Inverse } f_{\mathbf{j}} &= \sum_{\mathbf{n}} \hat{f}_{\mathbf{n}} \exp\left(2\pi i \frac{j_l n_l}{L_l}\right) \\ \text{Forward } \hat{f}_{\mathbf{n}} &= \sum_{\mathbf{j}} f_{\mathbf{j}} \exp(-2\pi i \frac{j_l n_l}{L_l}) \end{aligned}$$

The Fourier modes satisfy the following:

$$\hat{f}_{(n_1+a_1N_1, n_2+a_2N_2, \dots)} = \hat{f}_{(n_1, n_2, \dots)}$$

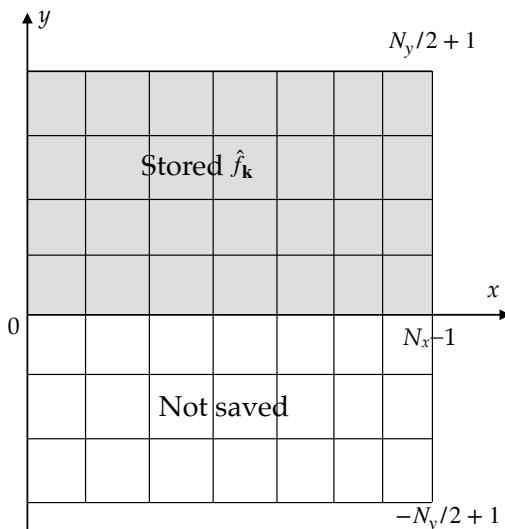
The above properties hold for real field, as well as complex field. For real $f(\mathbf{x})$, the Fourier coefficients satisfy another important property: $\hat{f}_{(-\mathbf{n})} = \hat{f}_{(\mathbf{n})}^*$, called *reality condition*. Due to this property, we save nearly half of all the Fourier modes.

Python's numpy module offers functions for multidimensional FFT. These functions are listed in [Table #tab\(multi_DFFT\)](#). For 2D, the corresponding functions are *fft2*, *ifft2*, *rfft2*, *irfft2*. For real data, 2D *rfft2(f)* returns an array of size $(N_1, N_2/2+1)$ that contains Fourier

modes with $0 \leq n_2 \leq N_2/2+1$. See [Figure 77](#) for an illustration.

Table !tab(multi_DFFT): Python's multidimensional FFT functions (numpy module).

Function	Operation
<i>fftn(f)</i>	Forward FFT (of complex f)
<i>ifftn(fk)</i>	Inverse FFT (of fk)
<i>rfftn(f)</i>	Forward FFT (of real f)
<i>irfftn(fk)</i>	Inverse FFT (of fk to real f)



[Figure 77:](#) The Fourier coefficients \hat{f}_n for 2D real data. Only \hat{f}_n in the grey zone are stored due to the reality condition.

Example 4: Consider the functions, $\cos(x+y)$ and $8 \cos(x) \sin(y)$, which are periodic in a $(2\pi)^2$ box. We compute the Fourier transforms of these functions as follows. Since $\cos(x+y) = (\frac{1}{2}) [\exp(i(x+y)) + \exp(-i(x+y))]$, we deduce that

$$\hat{f}_n = \frac{1}{2} [\delta_{n_x(1)} \delta_{n_y(1)} + \delta_{n_x(-1)} \delta_{n_y(-1)}]$$

That is, $\hat{f}_{(1,1)} = \hat{f}_{(-1,-1)} = 1/2$. Using similar arguments, we show that

$$\hat{f}_n = -2i[\delta_{nx(1)}\delta_{ny(1)} - \delta_{nx(-1)}\delta_{ny(-1)} + \delta_{nx(1)}\delta_{ny(-1)} + \delta_{nx(-1)}\delta_{ny(1)}]$$

The following Python code performs the above task.

```
L = 2*np.pi; Nx = 16; Ny = 16; dx = L/Nx; dy = L/Ny

f = np.zeros((Nx, Ny))
for jx in range(0,Nx):
    for jy in range(0,Ny):
        f[jx,jy] = 8*np.cos(jx*dx)*np.sin(jy*dy)

fk = np.fft.fftn(f)/(Nx*Ny)
```

Example 5: We compute the Fourier transform of $4 \sin(2x + 3y + 4z)$ in a $(2\pi)^3$ periodic box using the similar procedure as Example 4. The result and the corresponding Python code is given below.

$$\hat{f}_n = -2i[\delta_{nx(2)}\delta_{ny(3)}\delta_{nz(4)} - \delta_{nx(-2)}\delta_{ny(-3)}\delta_{nz(-4)}]$$

```
L = 2*np.pi; Nx = 16; Ny = 16; Nz = 16; dx = L/Nx; dy = L/Ny; dz = L/Nz

f = np.zeros((Nx, Ny, Nz))
for jx in range(0,Nx):
    for jy in range(0,Ny):
        for jz in range(0,Nz):
            f[jx,jy,jz] =
4*np.sin(2*jx*dx+3*jy*dy+4*jz*dz)

fk = np.fft.fftn(f)/(Nx*Ny*Nz)
```

Computing derivatives using Fourier Transforms

We can compute accurate derivatives using Fourier transforms as follows. We take the first derivative of Eq. (43) that yields

$$\frac{df(x)}{dx} = \sum_{-\infty}^{\infty} ik_n \hat{f}_n \exp(ik_n x) \quad \dots\dots(51)$$

Therefore, the Fourier transform of $f'(x)$ is $\hat{f}'_n = ik_n \hat{f}_n$, and that of $f''(x)$ is $\hat{f}''_n = -k_n^2 \hat{f}_n$. Using similar arguments we obtain the Fourier transform of $\partial f / \partial x_j$:

$$FT \left[\partial f / \partial x_j \right] = i k_j \hat{f}$$

Note that $f'(x)$ of Eq. (51) is exact as long as all the Fourier modes are included in the series. Note that every signal has a wavenumber cutoff, say k_{\max} . Therefore, the sum of Eq. (51) needs to be carried out from $-k_{\max}$ to k_{\max} , not $-\infty$ to ∞ . The above k_{\max} also determines minimum number of real-space points required for FFT computations. We need two sampling points for λ_{\min} , hence minimum grid spacing $h = \lambda_{\min}/2$. Therefore,

$$k_{\max} h \approx \pi \quad \dots \dots (52)$$

This condition is called *Nyquist criteria*. In addition, we require that $L = \lambda_{\max}$.

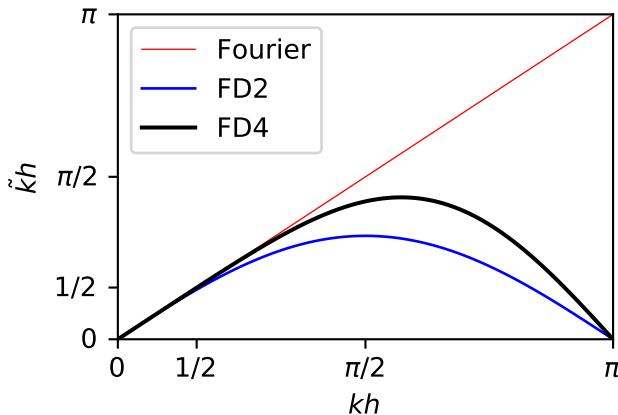
Let us compare the spectral derivative with the derivatives computed using finite difference method. If $f(x) = \exp(ikx)$, then using simple algebra, we can deduce that $f'(x_j) = i \tilde{k}_j f_j$ with

$$\begin{aligned} \tilde{k}h &= \sin(kh) \quad \text{for central difference scheme} \\ \tilde{k}h &= (4/3)\sin(kh) - (1/6)\sin(2kh) \quad \text{for fourth-order difference scheme} \end{aligned} \dots \dots (53)$$

which are illustrated in [Figure 78](#). Note however that the exact derivative is $f'(x_j) = ikf_j$, which is the limit achieved only when $h \rightarrow 0$. However, on practical front, $\tilde{k} \approx k$ up to $kh = 1/2$. See [For accurate simulation, it is desired that we resolve derivatives at the smallest scale, which puts a constraint that](#)

$$k_{\max} h \approx 1/2 \quad \dots \dots (54)$$

A comparison this relationship with Eq. (52) shows that for accurate computations, the grid resolution for finite-difference should $2\pi \approx 6$ times larger than that for Fourier method. The accuracy is improved using higher-order finite-difference derivatives.



[Figure 78](#): Plot of \tilde{f}_{kh} vs. kh for Fourier transform, and second-order and fourth-order finite difference methods.

Example 6: We compute the first derivative of $\cos(2x)$ using Fourier transform and finite difference. The Fourier transform of $\cos(2x)$ is nonzero for $\hat{f}_{-2} = \hat{f}_2 = 1/2$, multiplication of which with ik leads to $\hat{f}'_2 = i$ and $\hat{f}'_{-2} = -i$. The inverse Fourier transform of \hat{f}'_n yields $f'(x) = -2 \sin(2x)$, which is an exact answer. For the above computation, $N = 16$ is more than enough. A code segment for this computation is as follows:

```
kx = np.linspace(0, N//2, N//2+1)
fk_p = 1j*kx*fk    # fk is the Fourier transform of f(x)
fp = np.fft.irfft(fk_p,N)*N
```

We employ $np.gradient(f)/h$ (central difference method) to compute the first derivative of $\cos(2x)$. As shown in [Figure 78](#), the derivatives computed using Fourier transform are accurate, while that using FD has significant errors.

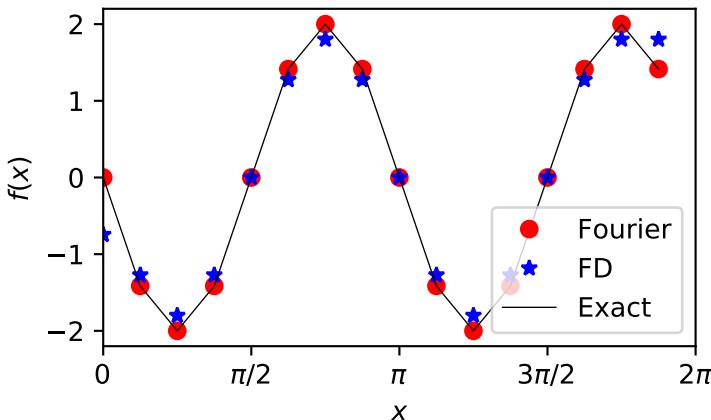


Figure 78: Example 6: Derivatives of $\cos(2x)$ computed using Fourier transform and Finite difference method.

Some additional important properties of Fourier transforms are as follows. Here, for convenience, we index the Fourier mode with its wavenumber.

- The Fourier transform of constant function, C , is $C \delta_{k(0)}$.
- The Fourier transform of a product of two real functions $f(x)$ and $g(x)$ is a convolution. That is, $(fg)_k = \sum_p f_{k-p} g_p$
- Parseval's theorem: $\langle f(x)g(x) \rangle = \sum_p \Re[f_p^* g_p]$
- Fourier transform of correlation function is the power spectrum. That is,

$$\frac{1}{L_x L_y L_z} \int d\mathbf{r} \langle f(\mathbf{r})g(\mathbf{r} + \mathbf{l}) \rangle \exp(-i(\mathbf{k} \cdot \mathbf{l})) = f(-\mathbf{k})g(\mathbf{k})$$

We end this section with a remark that *sin* and *cosine transforms* too are part of Fourier transforms. However, we skip them here due to lack of space. Another important topic that has been skipped here is *aliasing*.

Conceptual questions

1. What are the benefits of Fourier transform for computational methods?
2. How does the derivatives computed using Fourier transform compare those computed using finite difference? Derive the formulas of Eqs. (53).

Exercises

1. Using Python's functions, compute Fourier transforms of the following real functions that are periodic in domain $[0,2\pi]$: $\cos(6x) + \sin(4x)$; $\cos^3(2x)$; $\sin^4(4x)$. Compute the total energy of these functions and verify Parseval's theorem.
2. Using Python's functions, compute Fourier transforms of the following complex functions that are periodic in domain $[0,\pi]$: $\cos(2x) + i \sin(4x) + \cos(6x)$.
3. Consider the velocity field $\mathbf{u}(x,y) = 4[\hat{x} \sin 2x \cos 2y - \hat{y} \cos 2x \sin 2y]$ that are periodic in a π^2 box. Using Python's functions, compute the Fourier transforms of velocity field. Compute the average energy in real space and in Fourier space, and verify Parseval's theorem.
4. Consider a function $f(x,y,z) = 16 (\sin 2x \cos 5y \sin 16z + \sin x + \cos y)$ that is periodic in a $(2\pi)^3$ box. Compute the Fourier transform and energy of $f(x,y,z)$.
5. Consider $f(x) = \exp(-12.5 x^2)$. What is the length scale of this system? Compute the Fourier transform of this function in the domain $[-5, 5]$. Does the numerical answer look similar to the theoretical formula?
6. Compute Fourier transform of $f(x,y) = \exp[-12.5 ((x-10)^2 + (y-5)^2)]$ in the domain $[0:20] \times [0:20]$.
7. Verify Nyquist criteria using FFT functions of Python. For example, show that we need minimum four points to represent $\cos(2x)$.

14.2 Solving PDEs Using Spectral Method: Diffusion Equation

Partial differential equations (PDE) can be solved by employing Fourier transform or spectral method. The basic steps of spectral method are as follows. Consider the following PDE:

$$\frac{\partial \phi}{\partial t} = f(\phi, \frac{\partial \phi}{\partial x}, \frac{\partial^2 \phi}{\partial x^2}, t) \quad \dots\dots(55)$$

Here, the dependent variable $\phi(x,t)$ is a function of x and time t . The RHS of the equation contains a function dependent on ϕ and its spatial derivatives, as well on t . The domain of space variable x is $[0:L]$.

In spectral method, we assume that the function ϕ is periodic with a period of L . We take Fourier transform of Eq. (55) that yields ODEs for each Fourier mode of the system:

$$\frac{d\hat{\phi}_k}{dt} = \hat{f}_k$$

The conversion of a PDE to a set of ODEs is a major simplification. These ODEs could be solved using the methods described in Ordinary Differential Equation Solvers. Note that every physical system has a finite number of Fourier modes. In computer simulations we take these many modes or fewer of them.

In this chapter we solve the following equations using spectral method:

- Diffusion equation
- Wave equation
- Burgers eqation
- Navier-Stokes equation

Solving Diffusion Equation Using Spectral Method

The diffusion equation is

$$\partial_t \phi = \kappa \frac{\partial^2 \phi}{\partial x^2}$$

where $\partial_t \phi$ is a shorthand for $\partial \phi / \partial t$. The diffusion equation is one of the most important equations of physics and it describes diffusion of temperature, pollution, etc. in a static medium. Note that the diffusion time (time scale of the system) is L^2 / κ , where L is the system size. In a diffusion time, the field ϕ would have diffused considerably, for example, ϕ_{\max} could have gone down by a factor $1/2$.

In 1807, Fourier devised Fourier series to solve the diffusion equation given initial condition $\phi(x, t=0)$. Fourier's procedure to solve the diffusion is as follows. The diffusion equation in Fourier space is

$$\partial_t \hat{\phi}_k = -\kappa k^2 \hat{\phi}_k \dots \dots (56)$$

whose solution is

$$\hat{\phi}_k(t) = \hat{\phi}_k(t=0) \exp(-\kappa k^2 t) \dots \dots (57)$$

Using the initial condition, $\phi(x, t=0)$, we can compute

$$\hat{\phi}_k(t=0) = \frac{1}{L} \int_0^L \phi(x, t=0) \exp(-ikx) dx$$

Using $\hat{\phi}_k(t=0)$ we can compute $\hat{\phi}_k(t)$. An inverse transform from $\hat{\phi}_k(t)$ to $\phi(x,t)$ yields the final solution.

For Gaussian initial condition, $\phi(x, t=0) = \exp(-(x/a)^2/2)$, the analytical solution of $\hat{\phi}_k(t=0)$ is quite complex, as discussed in Example 2 of Section 14.1. It is much easier to compute $\hat{\phi}_k(t=0)$ approximately for a finite L . After this, using Eq. (57) and by performing inverse transform of $\hat{\phi}_k(t)$, we obtain $\phi(x,t)$. The following code segment performs this task.

Even though Eq. (56) has an exact solution, it is important to understand the evolution of this equation with Euler's forward method (from t_n to t_{n+1}):

$$\hat{\phi}_k^{(n+1)} = \hat{\phi}_k^{(n)}[1 - \kappa k^2(\Delta t)]$$

Note that $t_{n+1} - t_n = \Delta t$. Clearly, Euler's forward method is stable only if

$$|1 - \kappa k^2(\Delta t)| < 1, \text{ or } \Delta t < \frac{2}{\kappa k^2}$$

Note that Δt depends on wavenumber k , which is because the diffusion time scales for different modes are different. However, we need to choose a single Δt for all the modes, and it has to be the smallest among all of them:

$$\Delta t < \frac{2}{\kappa k_{\max}^2}$$

Note that $k_{\max} = 2\pi/\lambda_{\min} = 2\pi/(2h) = \pi N/L$, where h is the grid spacing and N is the total number of modes. Hence,

$$\Delta t < \frac{2h^2}{\kappa \pi^2}$$

Euler's forward method is unstable for a larger Δt . Other schemes, such as RK2 method, would have a similar Δt requirement. The above constraint on Δt is called *Courant–Friedrichs–Lowy (CFL)* condition. Note that a wide range of Δt , from $2/(\kappa k_{\max}^2)$ to $2/(\kappa k_{\min}^2)$, is related to the stiffness of the equations.

Example 1: We solve the diffusion equation in a 2π box with $\exp(-2(x-\pi)^2)$ as an initial condition. We choose $\kappa = 1$, $t_{\text{final}} = 1$ units. The initial and final fields are shown in [Figure 79](#). The field spreads out as time progresses. A code segment for the above computation is given below:

```

kappa = 1; tf = 1
L = 2*np.pi; N = 64; h = L/N
j = np.arange(0,N); x = j*h

f = np.exp(-2*(x-np.pi)**2)
fk = np.fft.rfft(f,N)/N
kx = np.linspace(0, N//2, N//2+1)

# Final states
fk_t = fk*np.exp(-kappa*kx**2*tf)    # in Fourier space

```

```
f_t = np.fft.irfft(fk_t,N) # in Real space
```

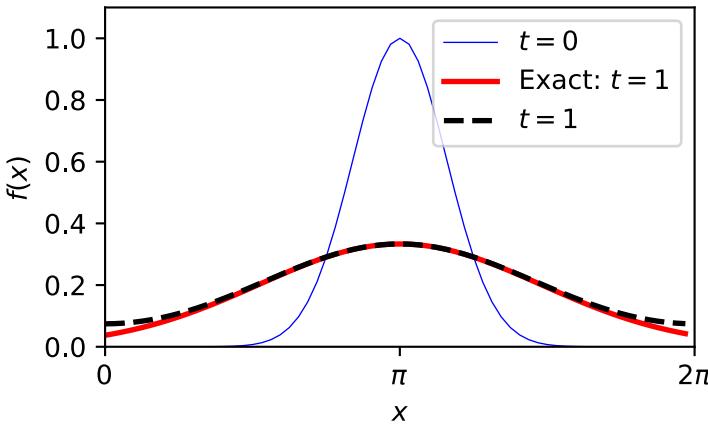


Figure 79: Evolution of a function $\exp(-2(x-\pi)^2)$ under diffusion. The numerical solution (black dashed line)

Equation (56) can be solved exactly, hence the above time marching procedures are not required for diffusion equation. However, we need such schemes for nonlinear equations—Burgers and Navier-Stokes equations—that will be described later in this chapter.

Example 2: We solve the diffusion in the infinite domain using Fourier transform following the formulas of Example 2 of Section 14.1. The Fourier transform of $\exp(-2(x-\pi)^2)$ is

$$\hat{\phi}(k, t=0) = a \exp\left[-\frac{1}{2}(ka)^2\right] \exp(-ik a)$$

where $a = \frac{1}{2}$. Hence, following the Example 1, we deduce that

$$\hat{\phi}(k, t) = a \exp\left[-\frac{1}{2}(ka)^2 - \kappa k^2 t\right] \exp(-ik a)$$

whose inverse transform yields

$$\phi(x, t) = \frac{a}{a'} \exp\left[-\frac{1}{2a'^2}(x - \pi)^2\right]$$

where $a' = \sqrt{a^2 + \kappa t}$. With time, the above $\phi(x,t)$ spreads with decreasing amplitude. The exact $\phi(x,t=2)$ in [Figure 79](#) matches quite closely with the numerical solution.

The procedure to solve 2D or 3D diffusion equation is very similar. We just need to employ multidimensional FFT functions. In later sections of this chapter we will solve more complex problems.

Conceptual questions

1. Make a qualitative comparison between the complexities of analytical and numerical solutions. How does it fare for finite L ?

Exercises

1. Solve the diffusion equation in a $[-\pi, \pi]$ box with $\exp(-8x^2)$ as an initial condition. We choose $\kappa = 1$, $t_{\text{final}} = 5$ units. Make a movie of the profile. Compare the numerical result with analytical results for infinite box.
2. Repeat Exercise 1 for a box $[-4\pi, 4\pi]$. Also, study the variations with respect to κ .
3. Work out the asymptotic solution (large t) for Example 2.

14.3 Solving Wave and Burgers Equation

In this section we solve several PDEs—wave, Burgers, and KdV equations—using spectral method.

Wave equation

A version of wave equation is

$$\partial_t \phi + c \partial_x \phi = 0 \quad \dots\dots(58)$$

where c is the velocity of the wave. The field $\phi(x)$ moves along $+x$ direction for positive c . Needless to say that wave equation is found every where: pressure waves, electromagnetic waves, elastic waves, etc.

We solve Eq. (58) given initial condition $\phi(x,t=0)$. Following exactly the same procedure as for the diffusion equation, we write Eq. (58) in Fourier space that yields a set of ODEs for all the Fourier modes:

$$\frac{d}{dt} \hat{\phi}_k = -i c k \hat{\phi}_k \quad (59)$$

whose solutions are

$$\hat{\phi}_k(t) = \hat{\phi}_k(0) \exp(i k c t) \quad \dots\dots(60)$$

For the initial condition $\phi(x,t=0) = \zeta(x)$, $\hat{\phi}_k(0) = \hat{\zeta}_k$, which is substituted in Eq. (60). With this, the general solution for the wave equation is

$$\phi(x, t) = \sum_k \hat{\zeta}_k \exp[i k (x - c t)] = \zeta(x - c t)$$

which is a $\zeta(x)$ travelling rightwards with a velocity of c .

Equation (59) has a simple solution, hence there is no real necessity to employ computational method for this equation. Still, it is instructive to consider the behaviour of explicit methods for the wave equation because they provide clues for solving nonlinear wave equations.

Let us employ Euler's forward method to Eq. (59) that yields

$$\hat{\phi}_k^{(n+1)} = (1 - i c k (\Delta t)) \hat{\phi}_k^{(n)}$$

where $t_{n+1} - t_n = \Delta t$. The above equation is unconditionally unstable. Hence, we need to employ either an implicit scheme or the exponential trick to solve Eq. (59) computationally.

Burgers equation

Burgers equation, given below, is a flow equation for a fully-compressible fluid:

$$\partial_t u + u \partial_x u = v \partial^2 u$$

where v is the kinematic viscosity of the fluid. In comparison to the diffusion equation discussed in Section 14.2, Burgers equation has an additional nonlinear term, $u \partial_x u$.

In Fourier space, we obtain the following set of equations for the Fourier modes:

$$\frac{d}{dt} \hat{u}_k = -i \frac{k}{2} \sum_p u_{k-p} u_p - \nu k^2 \hat{u}_k = -\hat{N}_k - \nu k^2 \hat{u}_k \dots \dots (61)$$

where k is the wavenumber, and the nonlinear term \hat{N}_k is a convolution. To solve this equation we can employ one of the methods discussed in Ordinary Differential Equation Solvers. First, we employ Euler's forward method. Following the prescription of Section 13.2, we observe that $\partial f / \partial \hat{u}_k = -\nu k^2$. Thus, the stability condition is unaffected by the nonlinear term. An application of Euler's forward scheme to Eq. (61) yields

$$\hat{u}_k^{(n+1)} = \hat{u}_k^{(n)} [1 - \nu k^2 (\Delta t)] - (\Delta t) \hat{N}_k$$

Therefore, the time-stepping scheme is stable when

$$|1 - \nu k^2 (\Delta t)| < 1, \text{ or } \Delta t < 2 / \nu k^2$$

We need to choose the lowest Δt , which is $2 / (\nu k_{\max}^2) = 2h^2 / (\nu \pi^2)$ (see

Section 14.2 for details). Here, h is the grid spacing. The set of equations have a range of time scales: $2/(vk_{\max}^2)$ to $2/(vk_{\min}^2)$. In addition, the nonlinear term too has its own time scales; the smallest time scale for the nonlinear term is h/u_{rms} , where u_{rms} is root mean square velocity. Hence, the equations for \hat{u}_k are stiff.

As discussed in Section 13.7, we can solve the stiffness problem by employing a semi-implicit scheme for the linear term or by using the exponential trick. Under one of the semi-implicit scheme, called *Crank-Nicolson scheme*,

$$\begin{aligned}\hat{u}_k^{(n+1)} &= \hat{u}_k^{(n)} - \nu k^2 (\Delta t) \frac{1}{2} \left(\hat{u}_k^{(n)} + \hat{u}_k^{(n+1)} \right) - (\Delta t) \hat{N}_k \\ \text{or, } \hat{u}_k^{(n+1)} &= \hat{u}_k^{(n)} \left[\frac{1 - \nu k^2 (\Delta t)/2}{1 + \nu k^2 (\Delta t)/2} \right] - (\Delta t) \hat{N}_k\end{aligned}$$

which is stable. For the exponential trick, we make a change of variable, $\hat{u}'_k = \hat{u}_k \exp(\nu k^2 t)$, that transforms Eq. (61) to

$$\frac{d}{dt} \hat{u}'_k = -\hat{N}_k \exp(\nu k^2 t)$$

An application Euler's forward scheme to the above equation yields

$$\hat{u}_k^{(n+1)} = \left[\hat{u}_k^{(n)} - (\Delta t) \hat{N}_k^{(n)} \right] \exp(-\nu k^2 (\Delta t))$$

We could also employ more advanced time-stepping schemes, such as RK2 and RK4 methods, for the above integration.

The computation of the convolution takes $O(N^2)$ floating-point operations, where N is the total number of Fourier modes, hence, it is very expensive. Fortunately, FFT enables us to compute the nonlinear term using $O(N \log(N))$ operations, which is much faster than $O(N^2)$ for large N . As shown in [Figure 80](#), we transform \hat{u}_k to $u(x)$ using IFFT, then compute $[u(x)]^2$ and perform forward FFT to obtain \hat{N}_k . The array multiplication takes $O(N)$ operations, hence, the overall time complexity of the nonlinear computation is $O(N \log(N))$. Since $[u(x)]^2$ is computed in real space, the present method is also called *pseudo-spectral method*.

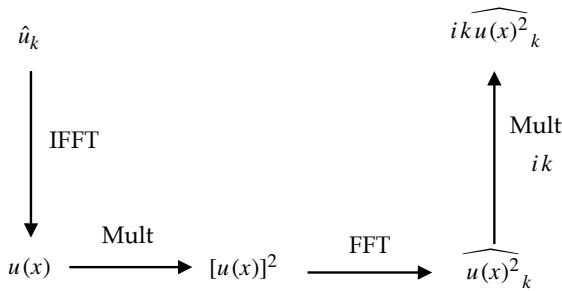


Figure 80: A schematic diagram illustrating computation of the nonlinear term $i k \widehat{u(x)^2}_k$.

Example 1: We simulate Burgers equation in $x = [0, 2\pi]$ with $v = 0.1$ up to final time of 1 unit. We start with initial state of $\sin(x)$. The evolution of $u(x)$ to a well formed shock is shown in [Figure 81](#). A code segment for the simulation is given below. We employ RK2 method for the time evolution.

```

nu = 0.1; tf = 1; dt = 0.001; nsteps = int(tf/dt)
L = 2*np.pi; N = 64; h = L/N
j = np.arange(0,N); x = j*h

kx = np.linspace(0, N//2, N//2+1)
exp_factor_dtby2 = np.exp(-nu*kx**2*dt/2)
exp_factor = np.exp(-nu*kx**2*dt)

def comput_Nk(fk):
    f = np.fft.irfft(fk,N)*N
    f = f*f
    fk_prod = np.fft.rfft(f,N)/N
    return (1j*kx*fk_prod)

f = np.sin(x) # initiation condition
fk = np.fft.rfft(f,N)/N # FT(f)

for i in range(nsteps+2):
    Nk = comput_Nk(fk)
    fk_mid = (fk - (dt/
2)*Nk)*exp_factor_dtby2
    Nk_mid = comput_Nk(fk_mid)
    fk = (fk - dt*Nk_mid)*exp_factor
  
```

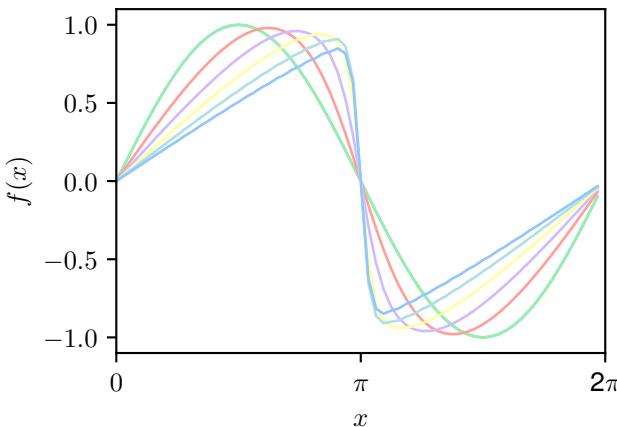


Figure 81: Example 1: Evolution of $u(x)$ for Burgers equation from $\sin(x)$ to a triangular profile with shock in between.

KdV equation

Korteweg–de Vries (KdV) is a prototype equation for shallow water waves. One version of the KdV equation is

$$\partial_t u + u \partial_x u = \kappa \partial^3 u \quad \dots \dots (62)$$

The nonlinearity of KdV equation is same as that of Burgers equation. The only difference between the two equations is that the viscous term of Burgers equation is replaced by a dispersive term ($\kappa \partial^3 u$) in KdV equation. Consequently, $\int (u^2) dx$ is conserved for KdV equation, but it is dissipated in the Burgers equation.

In Fourier space, the equation for the Fourier mode with wavenumber k is

$$\frac{d}{dt} \hat{u}_k = -\hat{N}_k - i\nu k^3 \hat{u}_k \quad \dots \dots (63)$$

where the nonlinear term \hat{N}_k is same as that for the Burger equation. Therefore, KdV equation could be computed using FFT as in Burgers equation. An implementation of the diffusive term using Euler's

forward method is unstable, hence either leapfrog method or the exponential trick is employed for the time integration.

Example 2: We simulate KdV equation for $\kappa = 0.1$ up to 2 time unit. The function $u(x)$ starts from $\sin(x)$ and evolves to two *solitons*. For simplicity, we employ RK2 method. The code is same as that for Example 1 except that the `exp_factor` is replaced by `exp_factor = np.exp(-1j*nu*kx**3*dt)`, and that we employ leapfrog method for time advancement.

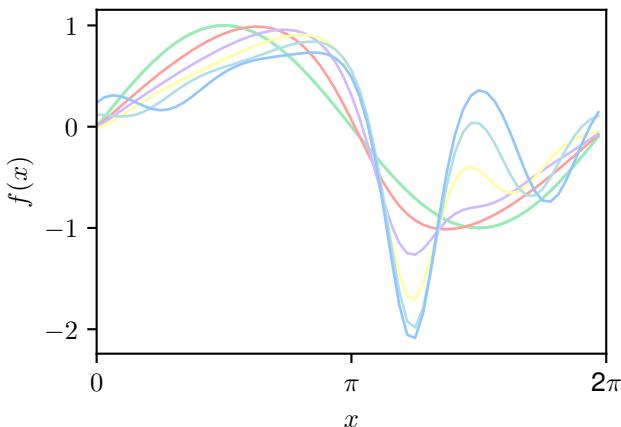


Figure 81: Example 2: Evolution of $u(x)$ for KdV equation from $\sin(x)$ to structures called *solitons*.

Conceptual questions

1. The solution of Burgers equation shows sharp shocks for very small viscosity. What kind of numerical problems do you anticipate for such situations?
2. Study the analytical solution of Burgers equation when $v = 0$. Try to simulate this energy conserving time-stepping schemes.
3. Study the KdV equation and its soliton solutions. You may refer to Drazin's book on Solitons.

Exercises

1. Solve the wave equation numerically in a periodic box of $[-4\pi, 4\pi]$. Take $c = 1$ and $\exp(-8x^2)$ as an initial profile. Which time-stepping scheme is suitable for this problem?
2. For Exercise 1, make a movie of the wave motion.
3. Solve the Burgers equation of Example 1 using Euler's forward method and compare your results with that shown in [Figure 81](#). Also study the behaviour of the solution under the variation of viscosity.
4. Solve Example 1 for $\nu = 10, 1, 0.01, 0.001$ with $\sin(x)$ as initial condition. What are the grid requirements for these simulations?
5. Prove that the exact solution of Burgers equation with $\nu = 0$ is x/t with shocks in between. How does the solution with $\nu = 0.001$ compare with the exact solution?
6. Show that the KdV equation conserves the total energy $\int(\frac{1}{2})u^2 dx$. What is the value of the total energy for Example 2. Does RK2 method conserve the total energy? Try out the conservation with the leapfrog method.
7. Solve KdV equation for Example 2 with $\kappa = 0.02$. Compare your results with those of Example 2.
8. It has been shown that for a KdV equation $\partial_t u + u \partial_x u = \kappa \partial_x^3 u$, one of the exact solutions is a travelling solution $(U/2) \operatorname{sech}^2[(\sqrt{U}/2)(x - Ut - x_0)]$ with a velocity of U . Demonstrate it numerically.
9. Solve KPZ equation $\partial_t h = \frac{1}{2}(\partial_x h)^2 + \nu \partial_x^2 h$ numerically in a periodic box of $[0, 2\pi]$. Take $\nu = 0.1$.

14.4 Spectral Solution of Navier-Stokes Equation

In this section we will describe how to solve Navier-Stokes equation and Gross-Pitaevskii equation using spectral method.

Solving Navier-Stokes equation

Navier-Stokes equations describe fluid flows, which are seen everywhere—from your home to the universe. In general, flow properties depend on the box geometry, boundary condition, degree of compressibility, initial condition, etc. In this chapter, we make a simplified picture. We focus on an incompressible flow far away from the walls. To a good approximation, such flows can be simulated in a periodic box using spectral simulation. This is what we will describe in this section.

Navier-Stokes equations under an incompressible limit are given below:

$$\partial_t \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \nu \nabla^2 \mathbf{u} \quad \dots\dots(64)$$

$$\nabla \cdot \mathbf{u} = 0 \quad \dots\dots(65)$$

The latter equation implies that the flow is incompressible or fluid density is constant. Given initial condition, we solve the above equations in a periodic box of size $L_x \times L_y \times L_z$. The numerical method for the above equations is very similar to that for Burgers equation, except two major differences: (a) Equation (64) has more nonlinear terms than Burgers equation. (b) We need to take into account pressure that is solved using the constraint equation, Eq. (65).

We write down the following equation for the Fourier mode $\hat{u}_{\mathbf{k},i}$, where $i = 1, 2, 3$ stands for the component of the field:

$$\begin{aligned} \frac{d}{dt} \hat{u}_{\mathbf{k},i} &= -ik_j \widehat{u_j(\mathbf{r}) u_i(\mathbf{r})}_\mathbf{k} - ik_i \hat{p}_\mathbf{k} - \nu k^2 \hat{u}_{\mathbf{k},i} \\ &= -\hat{N}_{\mathbf{k},i} - ik_i \hat{p}_\mathbf{k} - \nu k^2 \hat{u}_{\mathbf{k},i} \quad \dots\dots(66) \\ k_i \hat{u}_{\mathbf{k},i} &= 0 \end{aligned}$$

where $\hat{p}_\mathbf{k}$ is the Fourier transform of the pressure field, and $\hat{N}_{\mathbf{k},i}$ is Fourier transform of the nonlinear term $(u_j \partial_j u_i)$. Note that i in front of $\hat{N}_{\mathbf{k},i}$ is $\sqrt{-1}$. We divide the real space so that there are $M = N_1 \times N_2 \times N_3$

grid points. The wavenumber components of the modes are $(2\pi/L_i)n_i$ where n_i takes values from $[-N_1/2+1:N_1/2, -N_2/2+1:N_2/2, 0:N_3/2+1]$. The reality condition is $\hat{\mathbf{u}}_{-\mathbf{k}} = [\hat{\mathbf{u}}_{\mathbf{k}}]^*$.

The nonlinear term is computed using the following steps (see Figure 82): (a) Compute $\hat{\mathbf{u}}(\mathbf{k}) = \text{IFFT}(\mathbf{u}(x))$; (b) compute the products $u_i u_j$; (c) compute the Fourier transform of the nonlinear terms; (d) multiply $\text{FT}(u_i u_j)$ by ik_j that yields $ik_j \widehat{u_j(\mathbf{r}) u_i(\mathbf{r})}_{\mathbf{k}}$. This computation requires $O(M \log(M))$ operations.

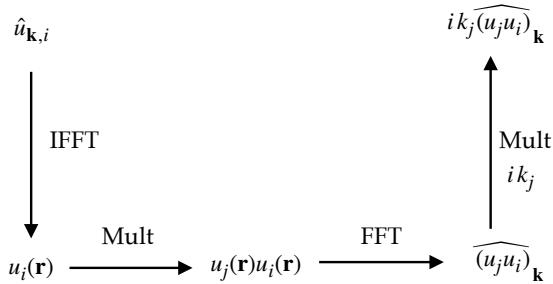


Figure 82: A schematic diagram illustrating computation of the nonlinear term $\hat{N}_{\mathbf{k},i} = ik_j \widehat{u_j(\mathbf{r}) u_i(\mathbf{r})}_{\mathbf{k}}$.

Taking a divergence of Eq. (64) yields

$$-\nabla^2 p = \nabla \cdot [\mathbf{u} \cdot \nabla \mathbf{u}] = \nabla \cdot \mathbf{N}$$

whose Fourier transform provides the pressure as

$$\hat{p}_{\mathbf{k}} = i \frac{1}{k^2} \mathbf{k} \cdot \hat{\mathbf{N}}_{\mathbf{k}}$$

which is substituted in Eq. (66). We time advance $\hat{u}_{\mathbf{k},i}$ once all the terms of RHS have been computed. Application of Euler's forward method yields

$$\hat{u}_{\mathbf{k},i}^{(n+1)} = [1 - \nu k^2(\Delta t)]\hat{u}_{\mathbf{k},i}^{(n)} + [\hat{N}_{\mathbf{k},i}^{(n)} - i k_i \hat{p}_{\mathbf{k}}^{(n)}] \Delta t$$

The method is stable only if (see Section 14.3)

$$\Delta t < \tau_\nu = \frac{2h^2}{\nu \pi^2}$$

The nonlinear term provides another important small timescale, $\tau_{NL} = h/u_{rms}$. The ratio of the two time scales is $\tau_v / \tau_{NL} \approx u_{rms}h/\nu$. Therefore, for viscous flows (Reynolds number $Re \lesssim 1$), τ_v is smaller of the two time scales and $\Delta t \approx \tau_v$. However, for turbulent flows ($Re \gg 1$),

$$\frac{\tau_\nu}{\tau_{NL}} \approx \frac{u_{rms}h}{\nu} \approx \frac{Re}{N} \approx \frac{Re}{Re^{3/4}} \approx Re^{1/4}$$

Here we use the fact that for turbulent flows $N = Re^{3/4}$. The above relation shows that $\tau_{NL} < \tau_v$ for $Re \gg 1$. Hence, for turbulent flows, Δt is determined by the nonlinear term, and CFL condition is not relevant in this case.

Example 1: For $L = 1$, $\nu = 10^{-4}$, $N = 1000$, $u_{rms} = 1$ leads to $h = .$ For these parameters, $\tau_v = 10^{-2}$ and $\tau_{NL} = 10^{-3}$.

We urge the students to write a Python code for solving Navier-Stokes equations in 2D and 3D. In 1D, $\nabla \cdot \mathbf{u} = 0$ implies that $\mathbf{u} = \text{constant}$. Hence, such flows are interesting.

Assessment of spectral method

A major advantage of spectral method is the accuracy. As shown in this chapter, the derivatives computed using Fourier transforms are accurate. This feature helps in accurate simulations of differential equations.

Spectral simulations have severe limitations as well. Fourier transform operate on functions that are periodic in space. Using sin transforms, we can extend to functions satisfying vanishing boundary conditions at the walls. In addition, the domain geometry for Fourier basis functions are idealised, e.g., box. Using more specialised functions such as Bessel function and Legendre polynomials, we could employ spectral methods to cylindrical and spherical geometries.

However, real applications involve more complex geometries and boundary conditions. For example, flows in a curved pipe or in a room with windows and fans. Finite difference, finite volume, finite elements methods are employed to solve such applications. In the next chapter we will cover finite difference method.

Conceptual questions

1. How does the time complexity of Navier-Stokes equation compare with that of Burgers equation? Why is it hard to perform large-resolution fluid simulations?

Exercises

1. Solve the incompressible Navier-Stokes equations in a 2D periodic box of dimension $(2\pi)^2$. Take TG vortex as an initial condition and $v = 0.1$. Evolve the velocity field till 1 unit of time. You may use $(32)^2$ grid.
2. Repeat problem 1 for in a 2D periodic box of dimension $(2\pi)^3$ and $(32)^3$ grid.
3. Solve Euler equation, which is Navier-Stokes equation without viscosity, in 2D and 3D. Employ random velocity field as an initial condition. To conserve the total kinetic energy, employ leapfrog method.

14.5 Spectral Solution of Schrödinger Equation

In this section we will solve Schrödinger equation for several quantum systems. To maintain the conservation of probability, which is $\int d\mathbf{r} |\psi(\mathbf{r})|^2 = 1$, for such systems, we employ energy-conserving schemes, such as leapfrog method. We also use exponential trick for time integration to overcome instability issues. A general sketch of the solver is as follows.

Time-dependent Schrödinger equation for a particle moving in a potential of $V(\mathbf{r})$ is

$$i\hbar\partial_t\psi = \left[-\frac{\hbar^2}{2m}\nabla^2 + V(\mathbf{r}) \right] \psi$$

We nondimensionalize the above equation using r_a as length scale and \hbar/E_a as time scale, where E_a is the energy scale. We make a change of variable $t = t'(\hbar/E_a)$ and $\mathbf{r} = \mathbf{r}'r_a$ and derive the following nondimensional Schrödinger equation:

$$i\partial_{t'}\psi = \left[-\frac{\alpha}{2}\nabla'^2 + V'(\mathbf{r}') \right] \psi$$

where $\alpha = \hbar^2/(mr_a^2E_a)$ and $V' = V/E_a$. We take Fourier transform of the above equation and obtain

$$\partial_{t'}\hat{\psi}_{\mathbf{k}'} = -i\frac{\alpha}{2}k'^2\hat{\psi}_{\mathbf{k}'} - i\widehat{(V'\psi)}_{\mathbf{k}'} \dots \dots (67)$$

Time integration of the linear term using Euler's forward scheme is unstable, hence we employ the exponential trick and make a change of variable $\tilde{\psi}_{\mathbf{k}'} = \hat{\psi}_{\mathbf{k}'} \exp(i\frac{\alpha}{2}k'^2t')$. As a result, Eq. (67) gets translated to

$$\partial_{t'}\tilde{\psi}_{\mathbf{k}'} = -i\widehat{(V'\psi)}_{\mathbf{k}'} \exp(i\frac{\alpha}{2}k'^2t')$$

whose integration using leapfrog method is

$$\hat{\psi}_{\mathbf{k}'}^{(n+1)} = \hat{\psi}_{\mathbf{k}'}^{(n-1)} \exp(-i\alpha k^2(\Delta t')) - i(2\Delta t') \widehat{V'\psi}_{\mathbf{k}'} \exp(-i\frac{\alpha}{2}k^2(\Delta t')) ..(68)$$

We could also employ more accurate schemes. In the following example, we solve for the wavefunctions of a free particle and simple harmonic oscillator.

Example 1: We solve $\psi(x)$ for a free particle (mass m) whose initial wavefunction is $\exp(ik_0x)$. The energy of the particle is $(\hbar k_0)^2/2m$. We choose $1/k_0$ as the length scale. Therefore, $\alpha = 1$. Note that $V' = 0$. Therefore, Eq. (67) yields $\hat{\psi}_{k'}(t) = \hat{\psi}_{k'}(t = 0)\exp(-ik^2t'/2)$. Since $\hat{\psi}_{k'}(t = 0) = \delta_{k'(1)}$ we obtain $\hat{\psi}_{k'}(t) = \exp(-ik^2t'/2)\delta_{k'(1)}$, whose inverse Fourier transform yields the wavefunction at time t as

$$\psi(x, t) = \exp(ix - it'/2) = \exp(ik_0x - i\hbar k_0^2 t/(2m))$$

Since the above analytical solution is straightforward, we do not present a numerical solution for this problem.

Example 2: We solve $\psi(x)$ for a free particle with Gaussian packet as an initial condition. Here, $\psi(x, t=0) = (a\sqrt{\pi})^{-1/2} \exp(-(x/a)^2/2) \exp(ik_0x)$. We choose a as the length scale and $m/(\hbar a^2)$ as time scale. Hence $\alpha = 1$. Nondimensional $\psi(x', t=0) = \pi^{-1/4} \exp(-x'^2/2) \exp(ik_0ax')$. Following Example 1, we obtain

$$\hat{\psi}_{k'}(t) = \hat{\psi}_{k'}(t = 0)\exp(-ik^2t'/2)(69)$$

The forward Fourier transform of $\psi(x, t=0)$ is

$$\hat{\psi}_{k'}(t = 0) = \pi^{-1/4} \exp(-\frac{1}{2}(k'_0 - k')^2)$$

Substitution of the above in Eq. (69) and its subsequent inverse transform yields

$$\psi(x, t) = \left(\sqrt{\pi}(1 + it')\right)^{-1/2} \exp(ik_0a(x' - it'/2)) \exp\left(-\frac{(x' - k_0at')^2}{2(1 + it')}\right)$$

To solve this problem numerically, we consider $\psi(x)$ in a periodic box of length L with $L \gg a$ and $L \gg 1/k_0$. We employ numerical forward and inverse transforms (complex to complex) to obtain the final result. Here we take $a = 0.1$, $k_0 = 10$, $x = [-5\pi, 5\pi]$. The numerical result for the wavefunction at $t' = 2$ is very close to the analytical result. See Figure 83 for an illustration. A code segment for the above computation is given below:

```
tf = 2; L = 10*np.pi; N = 256; h = L/N
j = np.arange(0,N); x = j*h-L/2+h

# initcond
a = 0.1; k0 = 10; k0a = k0*a
f = 1/np.sqrt(np.sqrt(pi))*np.exp(-
x**2/2)*np.exp(1j*k0a*x)
fk = np.fft.fft(f,N)
kx_pos = np.linspace(0, N//2, N//2+1)
kx_neg = np.linspace(-N//2+1,-1,N//2-1)
kx = np.concatenate((kx_pos, kx_neg))*(2*pi/L)

fk_t = fk*np.exp(-1j*kx**2*tf/2)
f_t = np.fft.ifft(fk_t,N)*N. # final solution in real
space
```

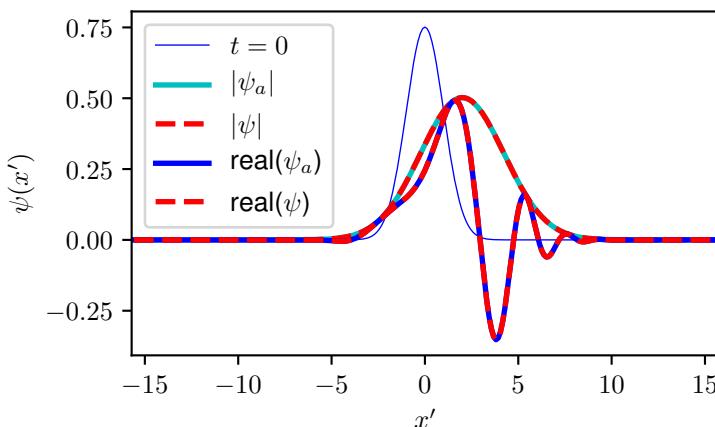


Figure 83: Example 2: Numerically computed wavefunction of a free particles with Gaussian initial condition. The subscript a represents the analytical solution of the wavefunction.

Example 3: We numerically solve the wavefunction for linear harmonic oscillator whose equation is

$$i\hbar\partial_t\psi = -\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + \frac{1}{2}m\omega^2x^2\psi$$

where m, ω are respectively the mass and frequency of the oscillator. We choose $\hbar/E = 1/\omega$ as the time scale and $\sqrt{\hbar/(m\omega)}$ as the length scale. With this, the nondimensional equation is

$$i\partial_t\psi = -\frac{1}{2}\frac{d^2\psi}{dx^2} + \frac{1}{2}x^2\psi$$

For the oscillator, in Eq. (68) we substitute $\alpha = 1$ and $V' = x'^2/2$ and solve for $\psi(x, t)$. For numerical computing we take finite box whose dimension is much larger than the length scale. Here we choose $L = 6\pi$ and $\psi(x', t=0) = (1/\sqrt{2}) \pi^{-1/4} \exp(-x'^2/2)[1+\sqrt{2}x']$, which is a combination of the ground state and the first excited state. We time advance using leap-frog method (along with the exponential trick) up to 2 nondimensional time unit. In Figure 84 we present the numerically computed wavefunction at $t' = 0, 1, 2$. The final state matches with the exact wavefunction, $(1/\sqrt{2}) \pi^{-1/4} \exp(-x'^2/2) [\exp(-it'/2) + \sqrt{2}x' \exp(-3it'/2)]$.

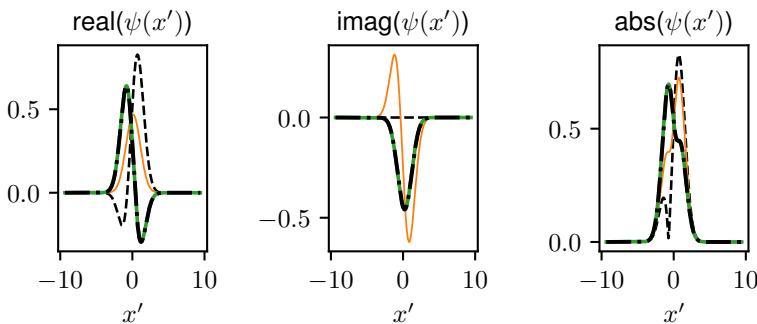


Figure 84: Numerically computed wavefunction of the simple oscillator of Example 3. The figure exhibits the real, imaginary, and absolute values of the wavefunction at $t' = 0, 1, 2$ using curves with increasing thicknesses. The initial and final wavefunctions are represented using thin and thick black-dashed curves respectively.

Solving Gross–Pitaevskii equation

Gross–Pitaevskii equation describes the behaviour of macroscopic wavefunction in quantum mechanics. The equation for the quantum wavefunction ψ , which is a complex function, is

$$i\hbar\partial_t\psi = \left[-\frac{\hbar^2}{2m}\nabla^2 + V(\mathbf{r}) + g|\psi|^2 \right]\psi \quad \dots\dots(70)$$

where \hbar is the Planck constant, g is a constant, and m is the mass of the quasi-particle that is moving in a potential $V(\mathbf{r})$. In spectral space, the equation for the Fourier mode $\hat{\psi}_{\mathbf{k}}$ is

$$i\hbar\partial_t\hat{\psi}_{\mathbf{k}} = \frac{\hbar^2 k^2}{2m}\hat{\psi}_{\mathbf{k}} + \widehat{(V\psi)}_{\mathbf{k}} + \widehat{(g|\psi|^2\psi)}_{\mathbf{k}}$$

which can be solved following a similar procedure as employed for Navier-Stokes equation.

In summary, for Schrödinger equation, it is important points to keep in mind are as follows: (a) the conservation of quantum probability, $\int d\mathbf{r} |\Psi|^2 = 1$; (b) Schrödinger equation is symmetric under time-reversible operation ($t \rightarrow -t$ and $\Psi \rightarrow \Psi^*$); (c) Application of Euler's forward method of the linear term is unstable. Therefore, we need to employ an accurate and stable method. The simplest scheme is leapfrog method with the exponential trick. More advanced method like PFERL (see Appendix) are more accurate. We urge the students to solve the above equation for $V(\mathbf{r}) = 0$.

Conceptual questions

1. What are the complexities for the numerical simulation of Schrödinger equation.
2. Describe similarities and dissimilarities between the numerical methods for solving Navier-Stokes and Gross–Pitaevskii equations.

Exercises

1. Solve Schrödinger equation for a linear harmonic oscillator using RK2 method and compare the results with those of Example 3.
2. Solve Schrödinger equation for free particles in 2D and 3D. Use Gaussian packet as an initial condition.
3. Solve Schrödinger equation for a linear harmonic oscillator in 2D.
4. Solve Gross–Pitaevskii equation with $V=0$ and $g=1$. Nondimensionalize the equation before proceeding to solve the equation. Employ leapfrog method along with the exponential trick. What is the error in the $\int d\mathbf{r} |\Psi^2|$?

CHAPTER FIFTEEN

SOLVING PDES USING FINITE DIFFERENCE METHOD

Synopsis

“Among all of the mathematical disciplines the theory of differential equations is the most important... It furnishes the explanation of all those elementary manifestations of nature which involve time.” Sophus Lie

15.1 General Overview & Diffusion Equation Solver

In this chapter we will discuss another prominent method called *finite difference (FD)* method for solving PDEs. FD method is less accurate than the spectral method, but they can be applied to more complex geometries and boundary conditions. However, in this introductory book, we limit ourselves to simple geometries.

We will use this method to solve the following set of PDEs:

- Diffusion equation
- Wave equation
- Burgers equation
- Navier-Stokes equation
- Schrödinger equation

The basic steps of finite difference method for 1D systems are as follows. Consider the following PDE for ϕ :

$$\frac{\partial \phi}{\partial t} = f(\phi, \frac{\partial \phi}{\partial x}, \frac{\partial^2 \phi}{\partial x^2}, t) \quad \dots\dots(71)$$

We discretize the domain $[0:L]$ at N points and label them as x_0, x_1, \dots, x_{N-1} . The separation between two neighbouring points is h , and the function at x_i is denoted by ϕ_i . Note that the end points are located at x_0 and x_{N-1} . For vanishing boundary condition, $\phi_0 = \phi_{N-1} = 0$. However, as in spectral codes, for periodic boundary condition, $\phi_0 = \phi_N$, where $x_N = x_{N-1} + h$ is outside the domain. See Figure [Figure 85](#) for an illustration.

At each x_i , we compute the RHS function and label it as f_i . It is best to use accurate and vectorized schemes for the spatial derivatives during the computation of f_i .

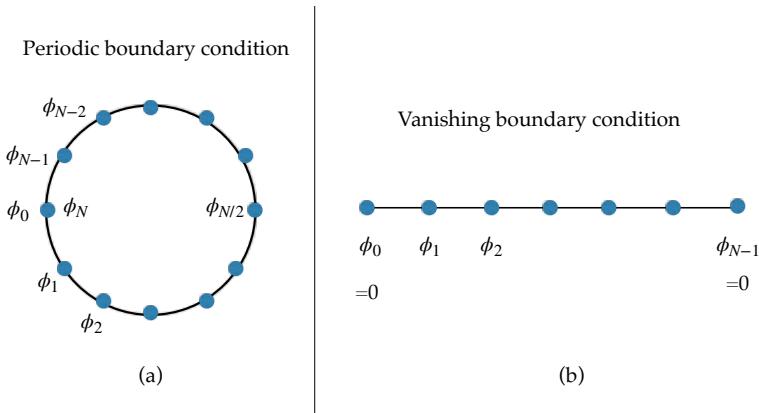


Figure 85: (a) Periodic boundary condition. (b) Vanishing boundary condition with $\phi_0 = \phi_{N-1} = 0$. In (a), $\phi_0 = \phi_N$, but ϕ_N is not stored.

The discretization process yields ODEs for each ϕ_i :

$$\frac{d}{dt}\phi_i(t) = f_i(t)$$

which can be solved using one of the time-stepping schemes. For example, in Euler's forward method, time stepping from $t^{(n)}$ to $t^{(n+1)}$ is achieved using

$$\phi_i^{(n+1)} = \phi_i^{(n)} + (\Delta t)f_i$$

As described earlier, we follow different indexing convention for the space grid (x_i) and time grid ($t^{(n)}$). Note that time varies from initial time $t^{(0)}$ to the final time t_{final} . See [Figure 86](#) for an illustration.

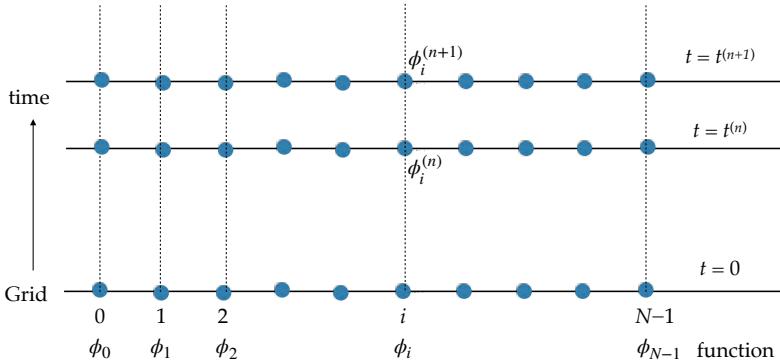


Figure 86: The domain x is discretized at N points. The functions at these points are $\{\phi_i\}$. The function evolves from $t = 0$ to final time via $t = t^{(n)}$.

Thus, FD method helps in converting a PDE to a set of ODEs at discrete positions in real space. Solution of these ODEs yield the final solution. Note however that the solution has significant errors due to two-step discretization, first in space and then in time. Due accurate derivative computations, the spectral methods less error than FD method.

We start with solution of the diffusion equation using FD method.

Solving diffusion equation using FD method

We discussed in Section 14.2, the diffusion equation is

$$\partial_t \phi = \kappa \frac{\partial^2 \phi}{\partial x^2}$$

with $\kappa > 0$ and real ϕ . We solve the above equation for a periodic boundary condition. For the same we discretize the domain at N points. We compute $\partial^2 \phi / \partial x^2$ using central difference. Consequently, the equation for ϕ_i is

$$\frac{d}{dt} \phi_i(t) = \frac{\kappa}{h^2} (\phi_{i+1} - 2\phi_i + \phi_{i-1})$$

Application of Euler's forward scheme to the above equation yields

$$\phi_i^{(n+1)} = \phi_i^{(n)} + \frac{\kappa \Delta t}{h^2} (\phi_{i+1}^{(n)} - 2\phi_i^{(n)} + \phi_{i-1}^{(n)}) \dots \dots (72)$$

The above equation is accurate up to $O(\Delta t)$. This equation is also unstable, as demonstrated below.

The analytical solution of the diffusion equation vanishes asymptotically, which is also required for the numerical solution. Note that $\phi(x)$ can be expanded as a Fourier series with basis functions $\exp(ikx)$, where k ranges from $2\pi/L$ to π/h . For stability, the coefficients for all the basis functions must decay as well. Therefore, to test the stability of Eq. (72), we substitute

$$\phi(x) = \exp(ikx)f(t) \quad \text{or} \quad \phi_i^{(n)} = \exp(ikx_i)f^{(n)}$$

in Eq. (72) and test if $f(t)$ grows or decays with time. The evolution equation for $f(t)$ is

$$f^{(n+1)} = f^{(n)} \left[1 + \frac{2\kappa \Delta t}{h^2} (\cos(kh) - 1) \right] \dots \dots (73)$$

Clearly, $f^{(n)}$ decays with time for $0 < \cos(kh) < 1$. However, it may grow with oscillations if $(2\kappa(\Delta t)/h^2)(\cos(kh)-1) < -1$. The most vulnerable Fourier mode for stability is one with the largest wavenumber ($kh = \pi$) for which $\cos(kh) = -1$. Hence, a sufficient condition for stability is

$$\frac{4\kappa \Delta t}{h^2} < 1 \quad \text{or} \quad \Delta t < \frac{h^2}{4\kappa}$$

In Section 14.1 we showed that for the same accuracy, h for a FD method is $2\pi \approx 6$ times smaller than that for a spectral method. Also, as shown in Section 14.2, for the stability of a spectral method, $\Delta t < 2h^2/(\kappa\pi^2)$. Based on these two limits, we conclude that $(\Delta t)_{\text{FD}} \approx (1/32)(\Delta t)_{\text{spectral}}$.

Euler's forward method is accuracy of $O(\Delta t)$. However, the accuracy of a RK2 method is $O((\Delta t)^2)$; in RK2 scheme,

$$\phi_i^{\text{mid}} = \phi_i^{(n)} + \frac{\kappa \Delta t}{2h^2} (\phi_{i+1}^{(n)} - 2\phi_i^{(n)} + \phi_{i-1}^{(n)})$$

$$\phi_i^{(n+1)} = \phi_i^{(n)} + \frac{\kappa \Delta t}{h^2} (\phi_{i+1}^{\text{mid}} - 2\phi_i^{\text{mid}} + \phi_{i-1}^{\text{mid}})$$

Example 1: We solve the diffusion equation in a 2π box with $\exp(-2(x-\pi)^2)$ as an initial condition. We take $\kappa = 1$, $N = 64$, $\Delta t = 0.001$, $t_{\text{final}} = 1$ units. The initial and final fields are shown in Figure 87. The solution using FD method is quite close to the exact result (see Section 14.2), as well as to the spectral solution. The difference between the spectral and FD solutions is of the order of 10^{-4} at each point. Note $\Delta t < \kappa/h^2$, hence, our method is stable.

A code segment for the above computation is given after the figure.

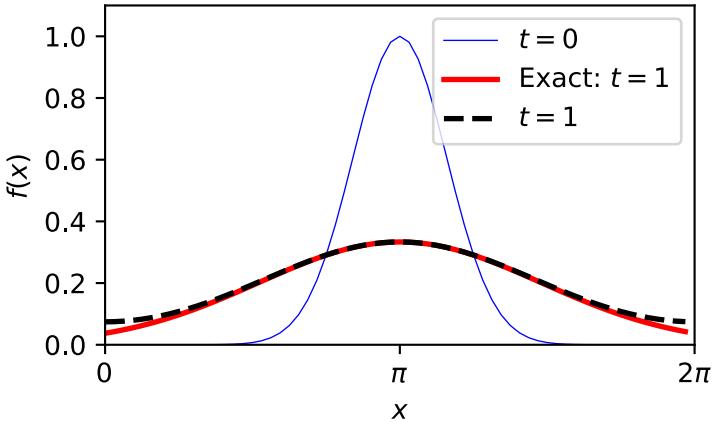


Figure 87: Evolution of a function $\exp(-2(x-\pi)^2)$ under diffusion. The numerical solution (black dashed line) is close to the exact solution (red curve), except at the end points.

```
init_temp = np.exp(-2*(x-pi)**2)
f = np.zeros(N+2); f_mid = np.zeros(N+2);
f[1:N+1] = init_temp; f[0] = init_temp[-1]; f[N+1] =
init_temp[0]

# Evolution with RK2 method.
for i in range(nsteps+2):
    f_mid[1:N+1] = f[1:N+1] +(prefactor/
2)*(f[0:N]-2*f[1:N+1]+f[2:N+2])
    f_mid[0] = f_mid[N]; f_mid[-1] = f_mid[1]

    f[1:N+1] = f[1:N+1] +
prefactor*(f_mid[0:N]-2*f_mid[1:N+1]+f_mid[2:N+2])
```

```
f[0] = f[N]; f[-1] = f[1]
```

To exploit vectorization features in Python, we save the field in array $f[1] \dots f[N]$, with $f[0] = f[N]$ and $f[N+1] = f[1]$ due to periodic boundary condition. With this arrangement, we can easily compute the second derivative, f'' , using central difference scheme at the end point using vectorized operations. See the code-segment above. We follow this convention for all our FD implementations.

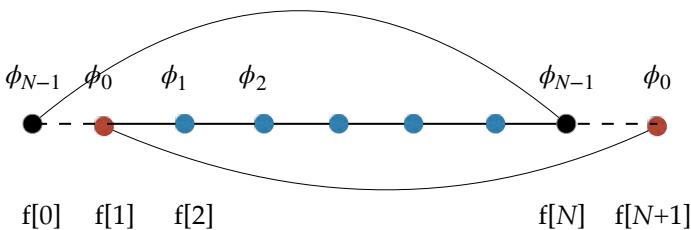


Figure 88: For vectorised operations, we save array $\phi_0 \dots \phi_{N-1}$ in $f[1]$ to $f[N]$ in an $N+2$ sized array. We also impose $f[0] = f[N]$ and $f[N+1] = f[1]$.

Application of Implicit scheme

The stability criteria $\Delta t < h^2 / (4\kappa)$ is too stringent for small h (say 10^{-4}). For such a scenario, it is better to employ a semi-implicit scheme. One such scheme for FD method is Crank Nicolson method, which is,

$$\begin{aligned}\phi_i^{(n+1)} = & \phi_i^{(n)} + \frac{\kappa \Delta t}{2h^2} \left[(\phi_{i+1}^{(n)} - 2\phi_i^{(n)} + \phi_{i-1}^{(n)}) \right. \\ & \left. + (\phi_{i+1}^{(n+1)} - 2\phi_i^{(n+1)} + \phi_{i-1}^{(n+1)}) \right]\end{aligned}$$

$$\text{or, } -\frac{\kappa \Delta t}{2h^2} \phi_{i-1}^{(n+1)} + \left(1 + \frac{\kappa \Delta t}{h^2} \right) \phi_i^{(n+1)} - \frac{\kappa \Delta t}{2h^2} \phi_{i+1}^{(n+1)}$$

$$= \phi_i^{(n)} + \frac{\kappa \Delta t}{2h^2} \left[\phi_{i+1}^{(n)} - 2\phi_i^{(n)} + \phi_{i-1}^{(n)} \right] \dots (74)$$

We have $(N-1)$ equations of type (74) for $\phi_0, \phi_1, \dots \phi_{N-1}$ with periodic boundary condition (see [Figure 85](#)). These equations can be written in the following matrix form:

$$\begin{pmatrix} X & Y & & & Y \\ Y & X & Y & & \\ & Y & X & Y & \\ & .. & .. & .. & \\ Y & & & Y & X \end{pmatrix} \begin{pmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \dots \\ \phi_{N-1} \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ \dots \\ r_{N-1} \end{pmatrix} \dots (75)$$

where $X = (1+\kappa(\Delta t)/h^2)$, $Y = -\kappa(\Delta t)/(2h^2)$, and r_i 's are the RHS of Eq. (74). The matrix in the above equation is a variant of tridiagonal matrix. Algorithms to solve tridiagonal matrices will be discussed in Section

For the vanishing boundary condition, $\phi_0=\phi_{N-1}=0$, the corresponding matrix equation is (leaving out ϕ_0 and ϕ_{N-1} that are zeros)

$$\begin{pmatrix} X & Y & & & \\ Y & X & Y & & \\ & Y & X & Y & \\ & .. & .. & .. & \\ & & & Y & X \end{pmatrix} \begin{pmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \dots \\ \phi_{N-2} \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \dots \\ r_{N-2} \end{pmatrix} \dots (76)$$

This equation too is solved using matrix solver.

Conceptual questions

1. State examples in physics where diffusion equation is used.
2. Contrast the advantages and disadvantages of spectral and FD methods for solving the diffusive equation.

Exercises

1. Solve Example 1 for different grid sizes and Δt and identify

the optimum Δt and grid size for simulating the diffusion equation of Example 1.

2. Solve 1D diffusion equation for $\kappa = 10, 1, 0.1, 0.01$ for vanishing boundary condition. Compare your result based on the time scale.
3. Employ higher-order space derivative for solving the diffusion equation. Compare the errors here with that for second-order derivative.
4. Solve the diffusion equation in a 2D box with $\kappa = 0.1$. Take $\exp(-50 r^2)$ as an initial condition. Study your result for different grid sizes and Δt . Make sure to employ vectorization for the 2D solution.

15.2 Solving Wave Equation

In this section we will solve the following wave equation using FD method:

$$\partial_t \phi + c \partial_x \phi = 0 \quad \dots\dots(77)$$

where c , a positive number, is the speed of the wave. The above equation represents a wave travelling rightwards with a speed of c . Note that the above equation is also called *advection equation* since the field ϕ is advected by the wave motion.

We employ central-difference scheme for the space-derivative computation and Euler's forward method for time stepping that yields

$$\phi_i^{(n+1)} = \phi_i^{(n)} - \frac{c \Delta t}{2h} (\phi_{i+1}^{(n)} - \phi_{i-1}^{(n)})$$

The above equation is unstable. To prove this statement, we substitute $\phi_i^{(n)} = \exp(ik x_i) f^{(n)}$ in the above equation that yields

$$f^{(n+1)} = f^{(n)} [1 - i c \Delta t \frac{\sin kh}{h}].$$

Since $|1 - ic(\Delta t) \sin(kh)/h| > 1$, the integration scheme is unstable.

However, Euler backward scheme, also called *upwind scheme*, is stable. The proof is as follows. The upwind scheme is

$$\phi_i^{(n+1)} = \phi_i^{(n)} - \frac{c \Delta t}{h} (\phi_i^{(n)} - \phi_{i-1}^{(n)}) \dots\dots(78)$$

Substitution of $\phi_i^{(n)} = \exp(ik x_i) f^{(n)}$ in the above equation yields

$$f^{(n+1)} = f^{(n)} \left[1 - \frac{c \Delta t}{h} \{1 - \exp(-ikh)\} \right] = f^{(n)} [1 - \alpha + \alpha \exp(-ikh)].$$

Where $\alpha = c(\Delta t)/h$. Using the identities $|z_1 + z_2| \leq |z_1| + |z_2|$ and $|z_1 - z_2| \geq |z_1| - |z_2|$, we show that the stability condition $|1 - \alpha + \alpha \exp(-ikh)| < 1$ if and only if

$$\alpha = c(\Delta t) / h < 1 \dots\dots(79)$$

Note that the wave covers distance h in time h/c . For a proper integration of the wave propagation, Δt must be lower than h/c , which is the consistent with the above bound.

Let us examine the nature of solution with central difference and upwind schemes. The error in central difference scheme is $h\partial^3\phi/\partial x^3$, a dispersive term (see KdV equation) because of which this scheme is unstable. On the other hand, upwind scheme has error of the form $h\partial^2\phi/\partial x^2$, which is dissipative, as in Burgers equation.

What about the wave travelling to the left? The equation for such waves is $\partial_t\phi - c\partial_x\phi = 0$. It is straightforward to show that Eq. (78) is unstable for time-stepping $\partial_t\phi - c\partial_x\phi = 0$. An appropriate time-marching scheme for a left-moving wave is

$$\phi_i^{(n+1)} = \phi_i^{(n)} + \frac{c\Delta t}{h}(\phi_{i+1}^{(n)} - \phi_i^{(n)}) \dots\dots(80)$$

Substitution of $\phi_i^{(n)} = \exp(ikx_i)f^{(n)}$ in the above equation yields

$$f^{(n+1)} = f^{(n)} \left[1 + \frac{c\Delta t}{h} \{ \exp(ikh) - 1 \} \right] = f^{(n)} [1 - \alpha + \alpha \exp(ikh)]$$

Following the same procedure as that for the right-moving wave, we deduce that the above time-marching scheme is stable if and only if $\alpha = c(\Delta t)/h < 1$, which is same as Eq. (79).

A careful inspection of Eqs. (78, 80) reveal that for the right-moving wave, the information propagates from site $i-1$ to i . However, for the left-moving wave, the corresponding propagation is from site $i+1$ to i . This observation is consistent with causality. The upwind scheme gets its name due to the above reason.

Example 1: We solve the wave equation for $c = 10$. We use $L = 10\pi$, $h = L/128$, and $\Delta t = 0.001$. We run the program till $t = 1$. We take a Gaussian waveform, shown as black dashed curve in [Figure 89](#) as an initial condition. [Figure <\\$n#figure:wave_FD](#) also illustrates that the solution is unstable for central difference scheme, but it is stable but dissipative for upwind scheme.

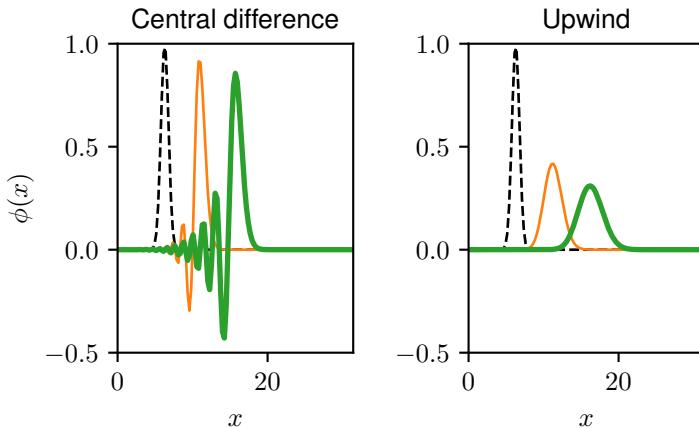


Figure 89: Numerical solution of wave equation using FD method:
 (a) Central difference scheme, which is unstable. (b) Upwind scheme,
 which is stable but dissipative. The black-dashed, orange-thin, and
 green-thick curves represent the waves at $t = 0, 0.5, 1$ respectively.

Conceptual questions

1. Compare the complexities of spectral and FD solutions of a wave equation.
2. Why the solution of a wave equation with central difference scheme unstable?

Exercises

1. Solve the wave equation $\partial_t \phi = \partial_x \phi$ numerically using central difference and upwind schemes. Compare the results with that for $\partial_t \phi = -\partial_x \phi$
2. For Exercise 1, study how the accuracy of the solution depends on parameters such as h and c ?

15.3 Burgers and Navier-Stokes Equations

In this section we will employ FD method to solve Burgers equation that contains advection and diffusion terms. The Burgers equation is

$$\partial_t u + u \partial_x u = \nu \partial_x^2 u$$

where u is the velocity field, and ν is the kinematic viscosity of the fluid. Note however that the advection velocity u is not constant.

This system has two time scales for instability: $\tau_\nu = h^2/\nu$ from the diffusion term and $\tau_{\text{NL}} = h/u_{\text{rms}}$ from the advection term, where h is the grid size. The ratio $\tau_\nu / \tau_{\text{NL}} = u_{\text{rms}} h / \nu$. We choose smaller of the the two timescales as Δt . We employ upwind scheme for the nonlinear term and central difference scheme for the diffusion term. Euler's forward scheme is employed for time-stepping. Consequently,

$$u_i^{(n+1)} = u_i^{(n)} - \frac{\Delta t}{h} u_i^{(n)} (u_i^{(n)} - u_{i-1}^{(n)}) + \frac{\nu \Delta t}{h^2} (u_{i+1}^{(n)} - 2u_i^{(n)} + u_{i-1}^{(n)}) \quad (\text{for } u_i^{(n)} > 0)$$

$$u_i^{(n+1)} = u_i^{(n)} - \frac{\Delta t}{h} u_i^{(n)} (u_{i+1}^{(n)} - u_i^{(n)}) + \frac{\nu \Delta t}{h^2} (u_{i+1}^{(n)} - 2u_i^{(n)} + u_{i-1}^{(n)}) \quad (\text{for } u_i^{(n)} < 0)$$

Alternatively we could also apply RK2 for the nonlinear term and central difference for the diffusion term. This scheme provides convergent solution for a reasonable length of time.

Example 1: We solve the Burgers equation for $\nu = 0.1$, $N = 128$, and $\Delta t = 0.001$. We take $\sin(x)$ as the initial condition. We carry out our simulation for final time of 1 unit using RK2 method. We exhibit the evolution of $u(x)$ in [Figure 90](#), with the final profile (purple-thick curve) exhibiting a clear shock front.

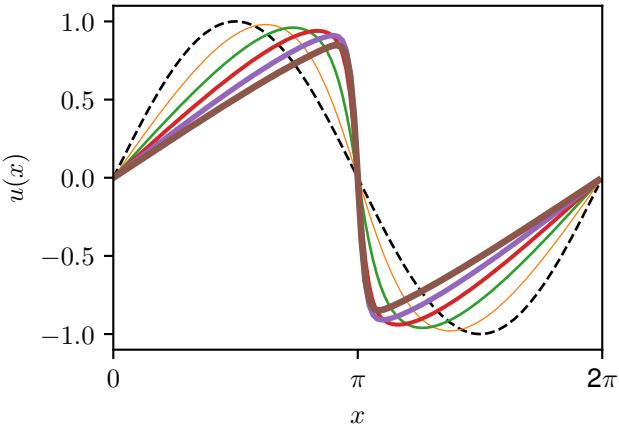


Figure 90: Numerical solution of Burgers equation with $\nu = 0.1$, $N = 128$ with initial condition of $\sin(x)$, which is shown as black dashed curve. The $u(x)$ at $t = 0.2, 0.4, 0.6, 0.8$, and 1 are exhibited using curves with increasing thicknesses.

Incompressible hydrodynamic equations

Now, we provide a brief discussion on how to solve *incompressible Navier-Stokes equations* using FD method. The governing equations are

$$\begin{aligned} \partial_t \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u} &= -\frac{1}{\rho} \nabla p + \nu \nabla^2 \mathbf{u} \\ \nabla \cdot \mathbf{u} &= 0 \\ \nabla^2 p &= -\rho \nabla \cdot [\mathbf{u} \cdot \nabla \mathbf{u}] \end{aligned}$$

Further, we assume that the density is constant. The last equation is *pressure Poisson equation*. Here we present a simple method, called *pressure correction* (Chorin's projection), to solve these equations.

The numerical scheme involves the following steps. First, given $\mathbf{u}^{(n)}$, we compute the intermediate velocity field \mathbf{u}^* using the following equation:

$$\frac{\mathbf{u}^* - \mathbf{u}^{(n)}}{\Delta t} = -\mathbf{u}^{(n)} \cdot \nabla \mathbf{u}^{(n)} + \nu \nabla^2 \mathbf{u}^{(n)}$$

The pressure, $p^{(n+1)}$ is determined using the following Poisson equation:

$$\nabla^2 p^{(n+1)} = \frac{\rho}{\Delta t} \nabla \cdot \mathbf{u}^*$$

Now, using \mathbf{u}^* and the correct pressure, $p^{(n+1)}$, we advance to $\mathbf{u}^{(n+1)}$:

$$\mathbf{u}^{(n+1)} - \mathbf{u}^* = - \frac{\Delta t}{\rho} (\nabla p^{(n+1)})$$

The solution of above equations is quite complex, and it is beyond the scope of this book.

Conceptual questions

1. Compare the accuracies of spectral and FD solutions of the Burgers equation.
2. Read more about Navier-Stokes equation and pressure correction scheme.

Exercises

1. Solve Example 1 for $\nu = 10, 1, 0.01, 0.001$ with $\sin(x)$ as an initial condition. What are the grid requirements for these simulations?
2. Compare the errors of the solution of Example 1 with the corresponding spectral solution.
3. Solve KdV equation $\partial_t u + u \partial_x u = \kappa \partial_x^3 u$ with $\kappa = 0.1$ using FD method. Compare the solution with the spectral one.
4. Write a program for solving 2D incompressible Navier-Stokes equation.

15.4 Schrödinger equation

In this section we will solve time-dependent Schrödinger equation using FD method. Since $\int |\psi|^2 dx (=1)$ is conserved, we employ leapfrog method that conserves this quantity.

For simplicity we solve 1D nondimensional Schrödinger equation for Hamiltonian H , which is $-(\frac{1}{2})\partial_{xx} + V(x)$. The Schrödinger equation is

$$i \partial_t \psi = H\psi$$

We decompose ψ into its real and imaginary parts: $\psi = R + iI$. The equations for R and I are as follows:

$$\begin{aligned}\partial_t R &= [-(\frac{1}{2})\partial_{xx} + V(x)] I \\ \partial_t I &= -[-(\frac{1}{2})\partial_{xx} + V(x)] R\end{aligned}$$

Note that the above equations have similar form as those for the position and momentum of an oscillator ($\dot{x} = Ap$; $\dot{p} = -Ax$). Hence, following the method adopted for a linear oscillator (see Section 13.6), we time advance R and I as follows:

$$\begin{aligned}R^{(n)} &= R^{(n-1)} + (\Delta t)[-(\frac{1}{2})\partial_{xx} + V(x)] I^{(n-\frac{1}{2})} \\ I^{(n+\frac{1}{2})} &= I^{(n-\frac{1}{2})} - (\Delta t)[-(\frac{1}{2})\partial_{xx} + V(x)] R^{(n)}\end{aligned}$$

Following the convention of FD methods, we discretize the space domain at N points. At the i^{th} site, the evolution equations (with central difference scheme) are

$$\begin{aligned}R_i^{(n)} &= R_i^{(n-1)} + (\Delta t) [-(I_{i+1}^{(n-\frac{1}{2})} - 2I_i^{(n-\frac{1}{2})} + I_{i-1}^{(n-\frac{1}{2})}) / (2h^2) + V_i I_i^{(n-\frac{1}{2})}] \\ I_i^{(n+\frac{1}{2})} &= I_i^{(n-\frac{1}{2})} - (\Delta t)[(R_{i+1}^{(n)} - 2R_i^{(n)} + R_{i-1}^{(n)}) / (2h^2) - V_i R_i^{(n)}]\end{aligned}$$

At the first time step, we time advance $I_i^{(0)}$ to $I_i^{(\frac{1}{2})}$, and in the last, we advance $R_i^{(n)}$ to $R_i^{(n+\frac{1}{2})}$. See [Figure 91](#) for an illustration. We remark that the numerically-computed wavefunction is approximate.

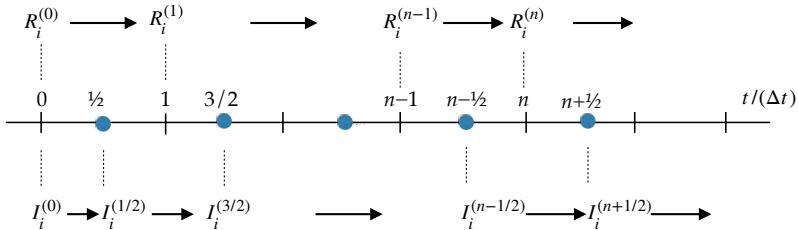


Figure 91: In the Schrödinger-equation solver using leapfrog method, at i^{th} site, R_i are computed at $t = (\Delta t), 2(\Delta t), \dots, n(\Delta t), \dots$, while I_i are computed at $(\Delta t)/2, 3(\Delta t)/2, \dots (n+1/2)(\Delta t), \dots$

Example 1: We solve $\psi(x)$ for a free particle ($H = -(\frac{1}{2})\partial_{xx}$) with nondimensional Gaussian packet $\psi(x, t=0) = \pi^{-1/4} \exp(-x^2/2) \exp(ix)$ as an initial condition. The numerical result for the FD code at $t' = 2$ is very close to the analytical result, as well as to the spectral result. See Figure 92 for an illustration.

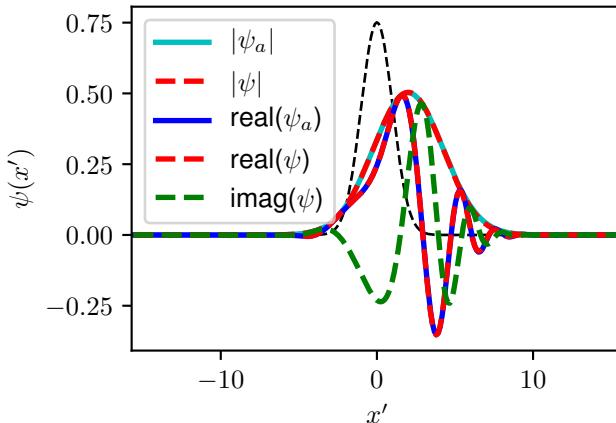
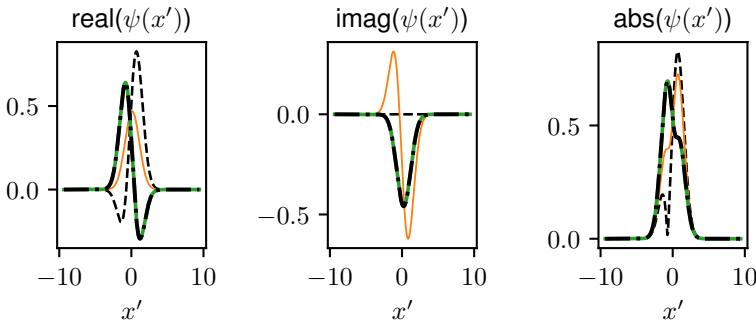


Figure 92: Example 1: Numerically computed wavefunction of a free particles with a Gaussian initial condition. The subscript a represents the analytical solution of the wavefunction, which is close to

the FD solution.

Example 2: We solve for the wavefunction of a linear harmonic oscillator whose Hamiltonian is $H = -(1/2)\partial_{xx} + (1/2)x^2$. We employ same initial condition and parameters as those used for the spectral method, that is, $L = 6\pi$, $\Delta t = 0.001$, $t_f = 2$, and $\psi(x', t=0) = (1/\sqrt{2}) \pi^{-1/4} \exp(-(x')^2/2)[1 + \sqrt{2}x']$. The numerical results for $t' = 0, 1, 2$ are shown in [Figure 93](#). The final state matches with the exact wavefunction, $(1/\sqrt{2}) \pi^{-1/4} \exp(-(x')^2/2)[\exp(-it'/2) + \sqrt{2}x' \exp(-3it'/2)]$.



[Figure 93](#): Numerically computed wavefunction of a simple oscillator of Example 2. The real, imaginary, and absolute values of the wavefunction at $t' = 0, 1, 2$ are illustrated using curves with increasing thicknesses. The initial and final wavefunctions are represented using thin-dashed and thick-dashed curves respectively.

Solving Schrödinger equation using split-operator method

We present an alternative time-stepping scheme called *split-operator method* for Schrödinger equation. In this method, we start with

$$|\psi(t + \Delta t)\rangle = \exp(-i\hat{H}\Delta t/2)\exp(-i\hat{H}\Delta t/2)|\psi(t)\rangle$$

$$\text{or, } \exp(i\hat{H}\Delta t/2)|\psi(t + \Delta t)\rangle = \exp(-i\hat{H}\Delta t/2)|\psi(t)\rangle$$

Now we expand the exponential operator to the first-order in Taylor

series that yields

$$[1 + i\hat{H}\Delta t/2]|\psi(t + \Delta t)\rangle = [1 - i\hat{H}\Delta t/2]|\psi(t)\rangle \quad \dots\dots(81)$$

Scalar product of Eq. (81) with $\langle x |$ yields the following equation

$$\begin{aligned} [1 + i(\Delta t/2)(-\partial_{xx}^2 + V)]\psi(x, t + \Delta t) &= [1 - i(\Delta t/2)(-\partial_{xx}^2 + V)]\psi(x, t) \\ \text{or, } -i \frac{\Delta t}{2h^2} \psi_{i-1}^{(n+1)} + \left[1 + i \frac{\Delta t}{2} V + i \frac{\Delta t}{h^2}\right] \psi_i^{(n+1)} - i \frac{\Delta t}{2h^2} \psi_{i+1}^{(n+1)} \\ &= -i \frac{\Delta t}{2h^2} \psi_{i-1}^{(n)} + \left[1 + i \frac{\Delta t}{2} V + i \frac{\Delta t}{h^2}\right] \psi_i^{(n)} - i \frac{\Delta t}{2h^2} \psi_{i+1}^{(n)} \end{aligned}$$

which is a tridiagonal matrix that is solved using a matrix solver (see Section).

Assessment of FD method

In this chapter we introduced FD method for simple problems. FD is a very powerful method that is heavily used for simulating equations with complex geometries and complex boundary conditions. Some of the leading applications are flows in pipes and Earth's atmosphere, electromagnetic waves for radar applications, stresses in structures, etc. The grids for such problems are typically nonuniform and complex. These topics are covered in specialised books on FD methods.

Note that spectral methods cannot simulate PDEs with complex geometries and complex boundary conditions. This is a major drawback of spectral method compared to FD method. However, for the same grid resolution, the errors in a FD method is larger than that in a spectral method. For similar accuracy we need higher grid resolution for FD method compared to spectral method. Thus, FD and spectral methods have their advantages and disadvantages.

Conceptual questions

- For the numerical solution of Schrödinger equation, compare

- the benefits and disadvantages of spectral and FD methods.
2. Describe similarities and dissimilarities between the FD methods for solving Navier-Stokes and Schrödinger equations.

Exercises

1. Solve the Schrödinger equation for a linear harmonic oscillator using RK2 method. Take the parameters of Example 2. Compare your results with those of Example 2.
2. Solve the Schrödinger equation for a free particle in 2D and in 3D. Use Gaussian packet as an initial condition.
3. Solve the Schrödinger equation for a linear harmonic oscillator in 2D.
4. Solve Gross–Pitaevskii equation with $V=0$ and $g=1$ using FD method. What is the error in the $\int d\mathbf{r} |\Psi^2|$ in this method?

CHAPTER SIXTEEN
SOLVING NONLINEAR ALGEBRAIC EQUATION

16.1 *Solvers*

Root finding of a nonlinear equation is an important exercise in science and engineering, especially for optimisation. Some examples of applications requiring roots are as follows: Roots for polynomials for Gauss quadrature; fixed points nonlinear equations and nonlinear maps; finding minima or maxima during optimisation.

In this chapter we will discuss how to find roots for a single nonlinear equation. The methods discussed here can be generalised to multiple nonlinear equations. In Chapter xxx, we will discuss how to compute the solution of a set of linear equations.

In this chapter we will describe some of the important root-finding algorithms: bisection method, Newton-Raphson's method, secant method, and relaxation method. Our objective is to solve the equation $f(x) = 0$. We start with root finding using bisection method.

Bisection method

In this scheme, we start with two points—one to the left of the root, and the other to the right of the root. We go closer to the root iteratively as the following.

1. We plot $f(x)$ vs. x and identify two points x_l and x_r that are on the left and right sides of the root x^* . See [Figure 94](#). We bracket the root between x_l and x_r .
2. Compute the mid point $x = (x_l + x_r)/2$.
3. If $|f(x)| \leq \varepsilon$, where ε is the tolerance level, then we have found the root. Stop here.
4. Otherwise, $|f(x)| > \varepsilon$. In this case, if $x < x^*$, x replaces x_l . Otherwise x replaces x_r .
5. We repeat the above process till $|f(x)| \leq \varepsilon$.

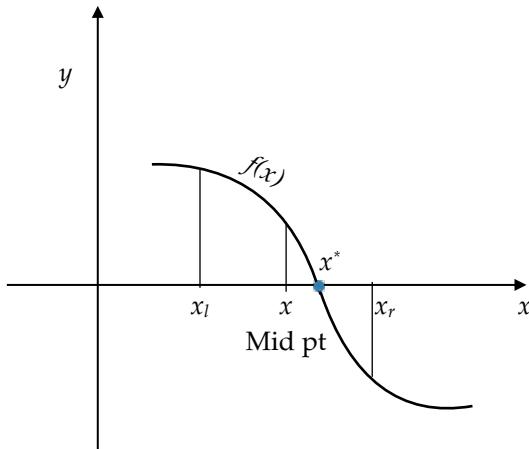


Figure 94: Root finder using bisection method. We start with x_l and x_r and get to the root iteratively.

Now we estimate the error in bisection method. Let us denote the two starting points as x_0 and x_1 , and later x 's as x_2, x_3, \dots, x_n , etc. Note that

$$x_{n+1} = (\frac{1}{2})(x_n + x_{n-1}) \quad \dots \dots (82)$$

Hence, the error, $\varepsilon_n = x^* - x_n$, evolves as

$$\varepsilon_{n+1} = (\frac{1}{2})(\varepsilon_n + \varepsilon_{n-1})$$

That is, the error is nearly halved in each step.

Example 1: Using bisection method, we find root of the equation $f(x) = \exp(x) - 6$. The correct answer is $\log(6) \approx 1.791759469228055$. First, we plot the function in [Figure 95\(a\)](#) and choose with $x_0 = 3$ and $x_1 = 1$, which are on the opposite sides of the root. After this we iterate to compute x_n 's using Eq. (82). We plot x_n vs. n in [Figure 95\(b\)](#), according to which the iteration converges to 1.7890625 for the tolerance level of $\varepsilon = 0.001$. A code segment is given below:

```
def f(x):
```

```

        return np.exp(x)-6

eps = 0.001; N = 10
x = np.zeros(N); x[0] = 3; x[1] = 1

for i in range(2,N):
    mid = (x[i-2] + x[i-1])/2
    if (abs(mid-x[i-1]) < eps):
        x[i] = mid
        break
    if f(x[i-2])*f(mid) > 0:
        x[i] = mid
    else:
        x[i-1] = x[i-2]; x[i] = mid
    print("i, mid = ", i, mid)

print ("estimated root ", mid)

```

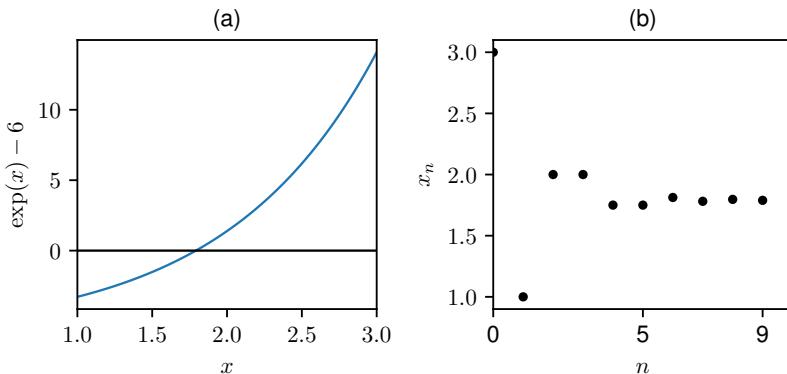


Figure 95: Example 1: Finding the root of $f(x) = \exp(x) - 6$ using bisection method. (a) plot of $f(x)$ vs. x . (b) Plot of x_n vs. n .

Example 2: We employ bisection method to compute the root of $f(x) = x^{1/3}$. We choose $x_0 = 1.7$ and $x_1 = -0.4$ and iterate Eq. (82). The solution converges to $x^* \approx 0$ (within precision of 10^{-5}) in 18 iterations.

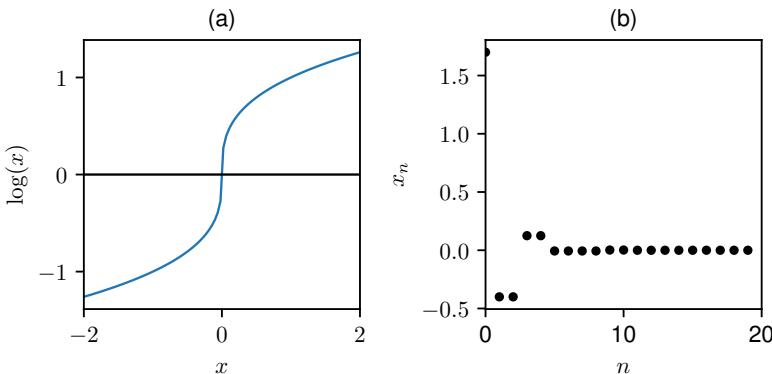


Figure 96: Example 2: Finding the root of $f(x) = x^{1/3}$ using bisection method. (a) plot of $f(x)$ vs. x . (b) Plot of x_n vs. n .

Newton-Raphson method

Newton-Raphson method provides a faster convergence to the root than the bisection method. In this method, we start with x_0 and compute x_1 using the slope at x_0 . As shown in Figure 97,

$$f'(x_0) = \frac{f(x_0)}{x_0 - x_1}, \text{ hence, } x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

For the $(n+1)^{\text{th}}$ step,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \dots\dots(83)$$

The process is continued until $|x_{n+1} - x_n| < \varepsilon$, the tolerance level.

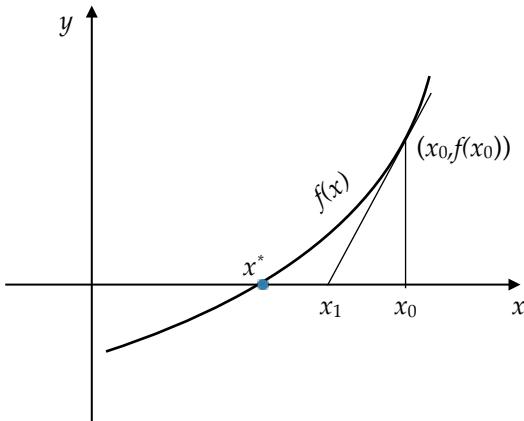


Figure 97: Root finder using Newton-Raphson method.

Now let us analyse the error for this method. We denote the root using x^* , hence $f(x^*)=0$. Rewriting of Eq. (83) and its Taylor expansion around x^* yields

$$x^* - x_{n+1} = x^* - x_n + \frac{f(x^*) - \epsilon_n f'(x^*) + (1/2)\epsilon_n^2 f''(x^*) + HOT}{f'(x^*) - \epsilon_n f''(x^*) + HOT}$$

$$\text{or, } \epsilon_{n+1} \approx -\frac{1}{2}\epsilon_n^2 \frac{f''(x^*)}{f'(x^*)} \quad \dots\dots(84)$$

Thus, the error converges as ϵ_n^2 as long as $f'(x^*)/f''(x^*)$ is finite. The quadratic convergence of Newton-Raphson method is a faster than the linear convergence of bisection method. However, Newton-Raphson method fails to converge if $|f'(x^*)/f''(x^*)|$ is large. For this case, the iteration diverges, that is, $\epsilon_{n+1} > \epsilon_n$. See [Figure 98](#) for an illustration.

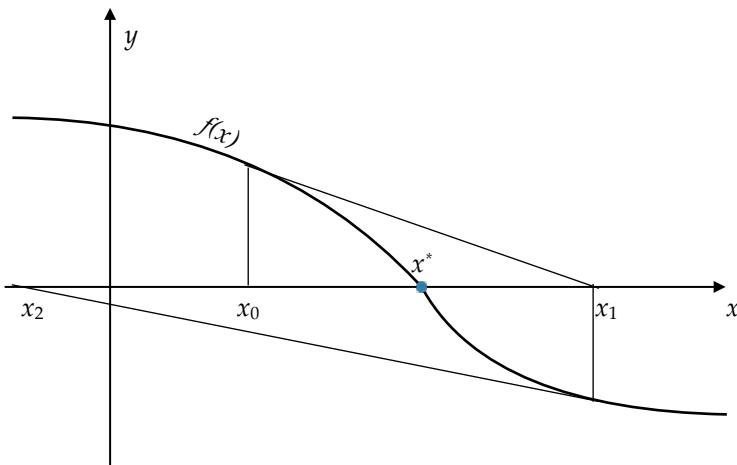


Figure 98: When $|f'(x^*)/f(x^*)|$ is large, $\varepsilon_{n+1} > \varepsilon_n$. Thus, we go further away from the root.

Example 3: We solve for the root of $f(x) = \exp(x) - 6$ using Newton-Raphson method. We start with $x_0 = 3$ and iterate using Eq. (83). For the tolerance level of $\varepsilon = 0.001$, the iteration converges to 1.7917594693651107 in 5 iteration. See [Figure 99\(a\)](#) for an illustration. A code segment for the computation is given below:

```
for i in range(1,N):
    x[i] = x[i-1] - f(x[i-1])/
    np.exp(x[i-1])
    if (abs(x[i]-x[i-1]) < eps):
        break
print ("estimated root ", x[i])
```

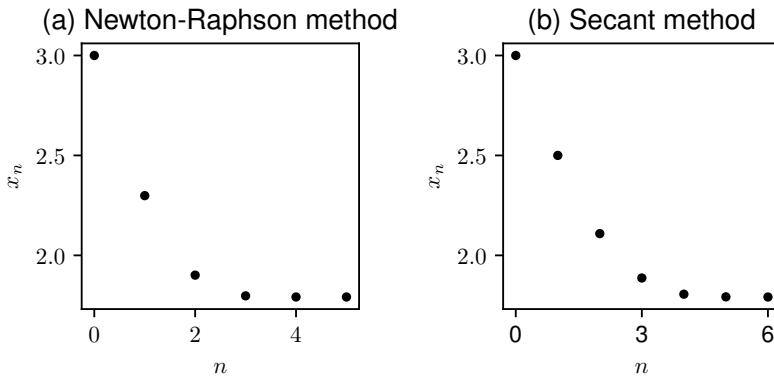


Figure 99: Finding the root of $f(x) = \exp(x) - 6$. Plots of x_n vs. n for (a) Newton-Raphson method (Example 3); (b) secant method (Example 5).

Example 4: Newton's Raphson method fails to compute the root of $f(x) = x^{1/3}$. We start with $x_0 = 1$ and iterate Eq. (83). As shown in Figure 100, the solution x_n diverges from the root $x^* \approx 0$. This is because of the divergence of $|f'(x^*)/f(x^*)|$ near the origin.

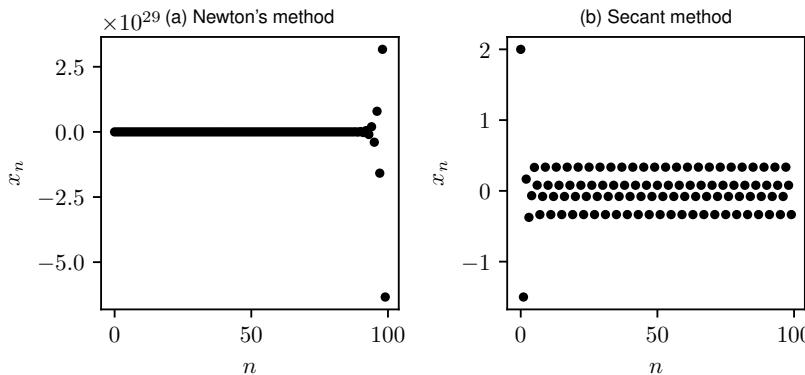


Figure 100: Finding the root of $f(x) = x^{1/3}$. Plots of x_n vs. n for (a) Newton-Raphson method (Example 4); (b) secant method (Example 6).

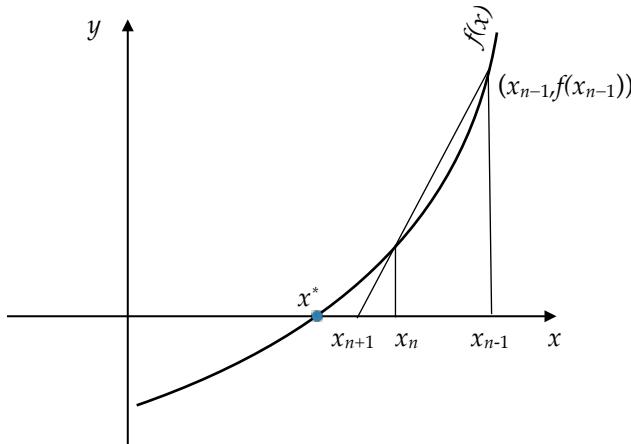
Secant method

Often it may be impractical to compute $f'(x)$. It is specially so when $f(x)$ is provided as a time series. In that case we estimate $f'(x_n)$ of Eq. (83) using two points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$ as follow (see [Figure 101](#)):

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Substitution of the above $f'(x_n)$ in Eq. (83) yields

$$x_{n+1} = x_n - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \quad \dots\dots(85)$$



[Figure 101](#): Root finder using Secant method.

This method is called secant method due to the secant passing through the points $(x_n, f(x_n))$ and $(x_{n-1}, f(x_{n-1}))$. Also, the convergence of the secant method is similar to that of Newton-Raphson method, that is, errors here converge as in Eq. (84).

Example 5: We solve for the root of $f(x) = \exp(x) - 6$ using the secant method. We start with $x_0 = 3$ and $x_1 = 2.5$, and take tolerance level of $\varepsilon = 0.001$. The iteration of Eq. (85) that converges to 1.791764077555014

in 6 steps. See [Figure 99\(b\)](#) for an illustration.

Example 6: The secant method fails to compute the root of $f(x) = x^{1/3}$ because $|f'(x^*)/f''(x^*)|$ diverges near the origin. However, unlike Newton-Raphson method, x_n 's oscillate around 4 values near the origin. See [Figure 100\(b\)](#) for an illustration.

Relaxation Method

Often we need solve equations of the form $f(x) = x$, which is equivalent to finding root of $F(x) = f(x) - x$. Let us denote the solution of $f(x) = x$ by x^* , that is, $f(x^*) = x^*$. Following the notation of dynamical systems, we refer to x^* as a *fixed point (FP)*.

To find x^* , we start with $x = x_0$ and compute $x_1 = f(x_0)$. After this we compute $x_2 = f(x_1)$, and continue further. The $(n+1)^{\text{th}}$ step is

$$x_{n+1} = f(x_n) \quad \dots \dots (86)$$

For some fixed points, called *stable fixed points*, the iteration converges, that is, $|x_{n+1} - x_n| < \varepsilon$. The fixed points for which the iteration does not converge are called *unstable fixed points*.

[Figure 101](#) illustrate two iterations. The iterations near the unstable FP diverge, while those near the stable FP converge. The diagram of [Figure 101](#) is also called a web diagram because the arrows form a web.

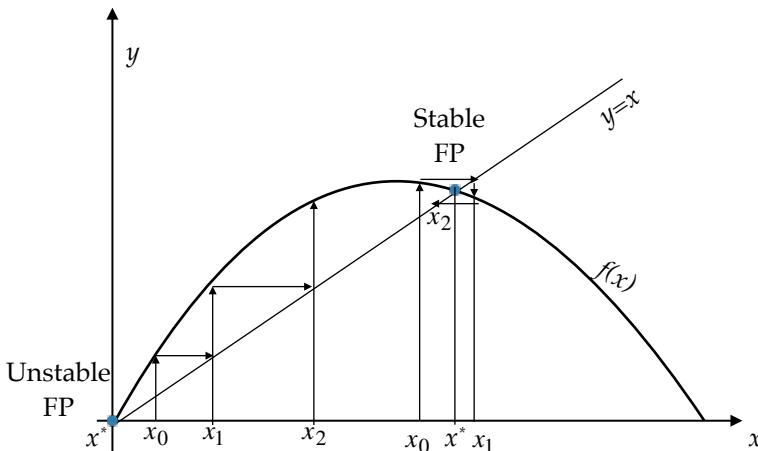


Figure 101: The web diagram for finding the fixed points of $f(x)$ using the relaxation method.

There is a simple criteria for convergence. We show below that the iteration converges to the *stable fixed point* x^* when $|f'(x^*)| < 1$. On the other hand, the iteration diverges when $|f'(x^*)| > 1$. We prove the above statement using the following arguments.

We subtract x^* from Eq. (86) that yields

$$x_{n+1} - x^* = f(x_n) - f(x^*) = (x_n - x^*)f'(x^*)$$

$$\text{or } \varepsilon_{n+1} = \varepsilon_n f'(x^*)$$

Clearly, $|\varepsilon_{n+1}| < |\varepsilon_n|$ when $|f'(x^*)| < 1$. Hence, $|f'(x^*)| < 1$ is the necessary condition for the convergence of the relaxation method. The iteration process diverges when $|f'(x^*)| > 1$.

How do we compute the unstable FP considering that the standard iteration diverges? **Figure 101** provides us a hint. The reversed arrows converge to the unstable fixed point. Hence, the iteration

$$x_n = f^{-1}(x_{n+1}) \quad \dots \dots (86)$$

will lead us to the unstable FP. Note that reversal of the arrows is

equivalent to going back in time.

Example 7: We find the solutions of $f(x) = 2.5 x(1-x) = x$ (see [Figure 102\(a\)](#)) using relaxation method. This equation has two solutions: $x^* = 0$ and $x^* = 1 - 1/2.5 = 0.6$. Note that $f'(0) = 2.5$ and $f'(0.6) = -0.5$. Hence, $x^* = 0$ is an unstable FP, while $x^* = 0.6$ is a stable FP. To reach to the stable FP, we start with $x_0 = 0.1$. The forward iteration take us to $x^* = 0.6$. The black dots of [Figure 102\(b\)](#) represent x_n 's during the iteration.

To obtain the unstable FP using relaxation method, we start with $x_{n+1} = 0.2$ and iterate backwards, that is, $x_n = f^{-1}(x_{n+1})$. We finally reach near the FP $x^* = 0$. The red circles of [Figure 102\(b\)](#) represent x_n for this iteration. Note that $f^{-1}(y)$ has two solutions. However, we employ

$$x = f^{-1}(y) = \frac{a - \sqrt{a^2 - 4ay}}{2a}$$

which is closer to $x^* = 0$.

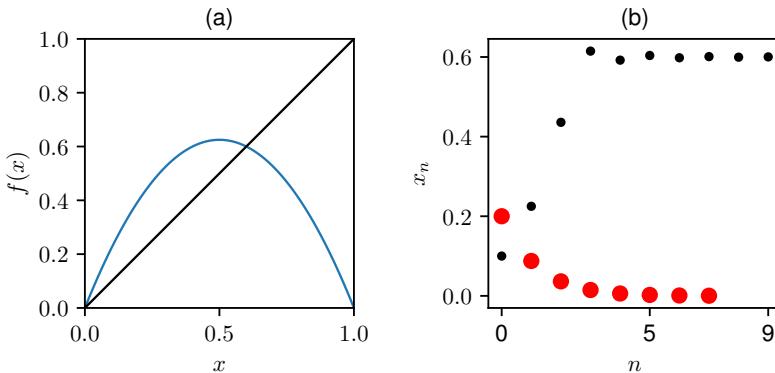


Figure 102: Example 7: Finding the roots of $f(x) = 2.5 x(1-x)$. (a) Plot of $f(x)$ vs. x . (b) Plot of x_n vs. n ; black dots for $x^* = 0.6$ and red circles for $x^* = 0$.

Python's root finder

Python's `scipy` module has several functions that finds roots of

equations. These functions are *brentq*, *brenth*, *ridder*, and *bisect*, whose arguments are function whose root need to found and the bracketing interval. Here we discuss how to use the function *brentq*. For example, the following code segments find the roots of $g(x) = x^2 - 1$ and $f(x) = \sin(\pi x) - x$.

```
In [308]: g = lambda x: x**2 - 1
In [309]: brentq(g, 0.5, 1.5)          # finds the root in
           the interval [0.5, 1.5]
Out[309]: 0.9999999999999993
In [310]: brentq(g, -1.5, -0.5)
Out[310]: -0.9999999999999993
In [312]: f =lambda x: np.sin(np.pi*x) - x
In [313]: brentq(f,-0.1, 0.1)
Out[313]: 0.0
In [314]: brentq(f,0.5, 1)
Out[314]: 0.7364844482410411
```

Assessment of root finders

Let us review the root finders discussed in this chapter. The bisection method is quite robust and simple, and it is guaranteed to converge. The convergence however may be slow. On the other hand, Newton-Raphson and secant methods converge quickly. These two methods however fail if $|f'(x^*)/f(x^*)|$ diverge. We illustrate this issue using $f(x) = x^{1/3}$ as an example. The bisection method can find the root of $f(x) = x^{1/3}$, but Newton-Raphson and secant methods fail to do so.

The relaxation method is an efficient and direct method to find the solution of $f(x) = x$. This method is very useful in dynamical systems.

We will root finders in the next chapter where we discuss boundary value problems.

Conceptual questions

1. Provide examples in physics where we need to find roots of nonlinear equations.
2. What are the merits and demerits of the bisection, Newton-Raphson, and secant methods for finding roots of nonlinear

equations.

Exercises

1. Find root of $f(x) = \log(x)$ using the bisection, Newton-Raphson, and secant methods. Compare their convergence rates.
2. Find all the roots of $f(x) = 65x^5 - 70x^3 + 15x$ numerically.
3. Give an example of a function for which Newton-Raphson method fails to find roots.
4. Find the solutions of $\sin(\pi x) = x$ using relaxation method.
5. Find the root of $f(x) = \tan(2x) - x$ in the interval $[-1/2, 1]$ using relaxation and Newton-Raphson methods.

CHAPTER SEVENTEEN
BOUNDARY VALUE PROBLEMS

17.1 Shooting Method

In Chapter Thirteen we solved ordinary differential equations (ODE) given initial conditions. In this chapter we will solve the differential equations using *boundary conditions*.

A n^{th} -order ODE requires n boundary values. For example, Newton's equation, $m\ddot{x} = F(x, \dot{x}, t)$, requires two boundary conditions, say $x(0)$ and $x(1)$. The equations for eigenvalue problems such as $d^2y/dx^2 = -a^2 y$, as well as time-independent Schrödinger equation, require two boundary conditions. Solution of splines, which is a fourth-order ODE, requires four boundary conditions. In this chapter we will cover the first three problems mentioned above.

For illustration of the method, we solve the following equation

$$d^2y/dx^2 = -a^2 y \quad \dots\dots(87)$$

given boundary conditions $y(0) = 0$ and $y(1) = 2$. Here, x is the independent variable, while $y(x)$ is the Unknown function. We solve this problem in the following manner.

We evolve Eq. (87), a 2nd-order ODE, as an initial value problem using $y(0) = 0$ and two guessed slope $y'(0)$ in such a way that one solution undershoots the target while the other one overshoots, as shown in [Figure 103\(a,b\)](#) for $a = 1$ and 5 respectively. After this, we employ the bisection method to converge to the final solution. Note that the end point $y(1)$ is a function of slope, hence we keep adjusting the slope using the bisection method until $|y(1)-2| < \varepsilon$, where ε is the tolerance limit.

In [Figure 103](#) we illustrate the evolution of $y(x)$ towards the final solution for $a = 1$ and 5 . For the former, we take first two slopes, $y'(0)$, as 0.1 and 2 , but for the latter, the first two slopes are 0.1 and -1 . The latter choice of -1 is somewhat counterintuitive. For the tolerance limit $\varepsilon = 0.01$, we converge to the final answer in 6 and 4 iterations respectively. A code segment for this process is given below:

```
def ydot(y,x):
    return ([y[1], -a**2*y[0]])

end_val = 0.1; a = 5; tollerance = 0.01;
x = np.linspace(0,1,100)
slope_t = -1
yinit = np.array([0, slope_t])
```

```

y = odeint(ydot, yinit, x)
ax2.plot(x,y[:,0],'r--', lw=1)

slope_b = 0.1
yinit = np.array([0, slope_b])
y = odeint(ydot, yinit, x)
ax2.plot(x,y[:,0],'g--', lw=1)

iter = 0
while ((abs(y[-1,0]-end_val) > tollerance) and (iter <
20) ):
    slope_mid = (slope_t+slope_b)/2
    yinit = np.array([0, slope_mid])
    y = odeint(ydot, yinit, x)
    ax2.plot(x,y[:,0])

    if (y[-1,0]>end_val):
        slope_t = slope_mid
    else:
        slope_b = slope_mid
    iter = iter +1

```

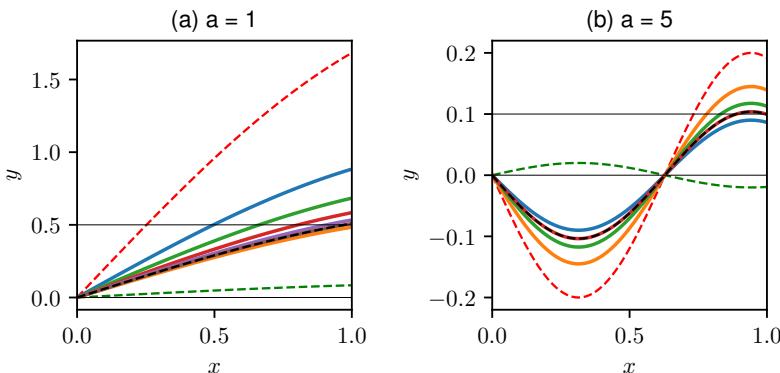


Figure 103: The solutions of $y'' = -a^2 y$ for (a) $a = 1$ and (a) $a = 5$ using the shooting method. The first two trial solutions are shown as red and green dashed curves below and above the real solution (black dashed curve).

For boundary condition $y(0) = 0$ and $y(1) = A$, the equation $y'' = -a^2 y$ has an exact solution, $y(x) = A \sin(ax)/\sin(a)$ that matches with the numerical solution.

Example 1: Let us solve $\ddot{x} = -a^2 x$ for boundary condition $x(t=0) = 0$ and $x(t=1)=A$. Physically, we want to start the mass of the oscillator from

the origin in such a way that it reaches $x = A$ at $t = 1$. Clearly, this problem is exactly same as the earlier example with a change of variable $x \rightarrow t$ and $y \rightarrow x$.

Example 2: We solve the equation of a pendulum $l d^2\phi/dt^2 = -g \sin\phi$ given boundary condition $\phi(0) = 0$ and $\phi(1) = \pi/4$. That is, we require the pendulum to start from $\phi(0) = 0$ and reach $\phi = \pi/4$ at $t = 1$ sec. We take $l = 1$ m and $g = 9.8$ m/sec². We solve the above problem using shooting method. We start with the first two slopes of 0.1 and 5. For the tolerance leve of 0.01 we reach the final solution in 8 iteration. See Figure 104 for an illustration.

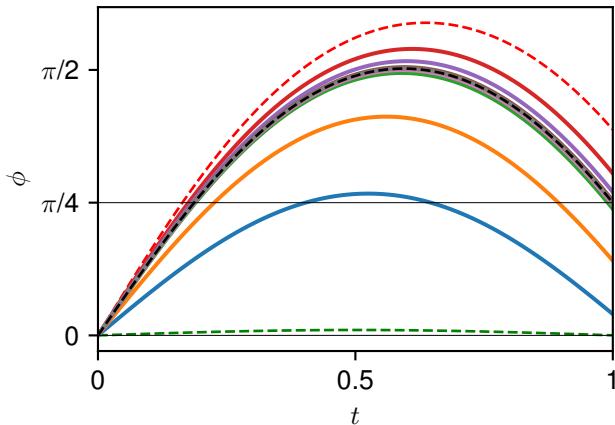


Figure 104: Solution of $d^2\phi/dt^2 = -g \sin\phi$ using shooting method. The first two trial solutions are green and red dashed curves. The final curve is the black dashed curve.

Conceptual questions

1. Provide examples of boundary value problems in science and engineering.
2. Compare the essential methods for solving boundary value problems and initial value problems.

Exercises

1. The shells from the Bofors gun leave with an speed of 1 km/sec. The shell is required to hit a target 10 km away. Use shooting method to compute the angle of launch of the shell. Assume the surface to be flat. Ignore air friction. Solve numerically.
2. Redo Exercise 1 in the presence of viscosity. Assume kinematic viscosity of air to be 0.1 cm²/sec.
3. Solve the equation $\ddot{x} = -x - \gamma\dot{x}$ with $\gamma = 0.1$ unit for the boundary conditions $x(0) = 0$ and $x(1) = 1$.
4. Numerically solve the equation $\varepsilon y'' + y' = 0$ for $\varepsilon = 0.001$ and the boundary conditions $y(0) = 0$ and $y(1) = 1$. This is an stiff equation.

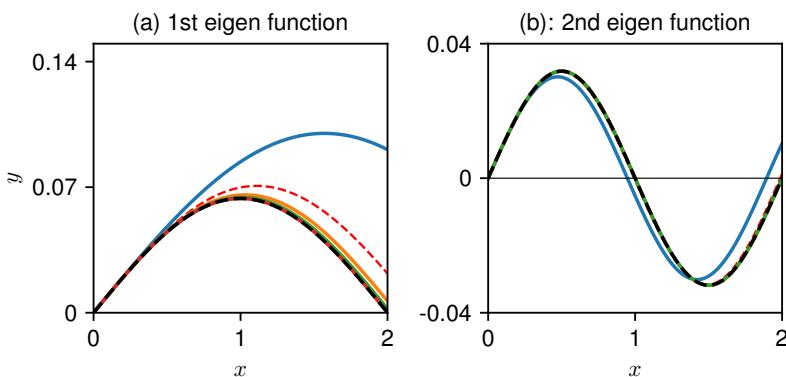
17.2 Eigenvalue Calculation

In this section we show how to compute the eigenvalue λ of the equation

$$y'' = \lambda y \quad \dots\dots(88)$$

for the boundary condition $y(0)=0$, $y(2)=0$. We also fix the slope at $x=0$, say $y'(0) = A$. Under these conditions the end point depends on λ . Hence, we iterate λ in such a way that $y(2)=0$. For a given λ , the function $y(x)$ that satisfies Eq. (88) is called eigenfunction corresponding to λ . In the following discussion we employ secant method to compute λ .

Equation (88) has infinite many eigenvalues. We can obtain these eigenvalues by starting the iteration with guessed value of λ . For the first eigenvalue, we start the iteration process $\lambda = -2$ and -1 and employ the secant method to reach the final λ . In 4 iterations we converge to -2.47 (within precision of 0.01), which is quite close to the exact value $\pi^2/4$. The second eigenvalue (π^2) can be reached by starting the iteration process from $\lambda = -11$ and -10 . We illustrate the above eigenfunctions in [Figure 105](#).



[Figure 105](#): First two eigenfunctions of equations $y'' = \lambda y$.

A code segment for computing the first eigenvalue is given below:

```
def ydot(y,x):
```

```

    return ([y[1], a*y[0]])

yinit = np.array([0, 0.1]) #slope = 0.1
a = -2
y = odeint(ydot, yinit, x)
yend = y[-1,0]
ax1.plot(x,y[:,0],'r--', lw=1)

a_prev = a; a = -1

iter = 0
while ((abs(a-a_prev) > tollerance) and (iter < 10)):
    y = odeint(ydot, yinit, x)
    ax1.plot(x,y[:,0])

    yend_prev, yend = yend, y[-1,0]
    a_prev, a = a, a - yend*(a-a_prev)/(yend-yend_prev)
    iter = iter + 1

```

For the vanishing boundary conditions at both ends, $y'' = \lambda y$ has exact solution, which is

$$y(x) = \sin(\sqrt{-\lambda}x)$$

with negative λ . Since $y(2) = 0$, we obtain $2\sqrt{-\lambda} = n\pi$, where n is an integer. This relation yields the eigenvalues as $\lambda_n = -(n\pi/2)^2$ with the first two eigenvalues as $-\pi^2/4$ and $-\pi^2$. The numerical results match with the analytical results quite well.

Solving eigenvalues with potentials

Now we make Eq. (88) a bit more complex by introducing a potential $V(x)$:

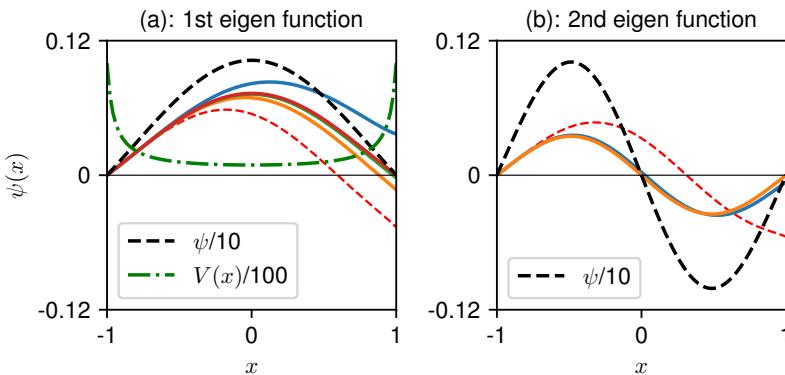
$$-(\frac{1}{2})y'' + V(x)y = Ey \dots\dots (89)$$

The time-independent nondimensionalized Schrödinger equation has the above form with $y(x)$ as the wavefunction of a particle moving under potential $V(x)$. In addition to Schrödinger equation, Eq. (89) describes many other physical systems, for example, a wave moving in a medium whose refractive index is varies in space.

We can solve Eq. (89) following the same procedure as described earlier in this section. Let us solve this equation in the domain $[-1, 1]$ with

$$V(x) = \frac{1}{1-x^2} \dots\dots(90)$$

Note that $V(x)$ takes a large value near the wall, hence it is similar to a square-well potential. In [Figure 106](#) we plot the eigenfunctions for the ground state and the first excited state of the system. The corresponding energies (eigenvalues E) are 2.30 and 6.24 respectively (within precision of 0.01). The figure also illustrates $V(x)$ as a chained curve.



[Figure 106](#): The eigenfunctions of equations $-(\frac{1}{2})y'' + V(x)y = Ey$: (a) Ground state, (b) the first excited state.

Typically the iterative solution of Eq. (89) is not normalized, that is $\int_{-1}^1 |y(x)|^2 dx \neq 1$. However, we can easily construct the normalised wavefunction as follows:

$$y_{\text{norm}}(x) = \frac{1}{\sqrt{\int_{-1}^1 |y(x)|^2 dx}} y(x)$$

For a particle in a box, $V(x) = 0$. Therefore, the governing equation is $y'' = -2Ey$. Hence, for the same boundary condition, the two lowest eigenvalues E for the particle in a box would be $\pi^2/8$ and $\pi^2/2$. Clearly, as expected, the eigenvalues for $V(x)$ of Eq. (90) are larger than those for the particle in a box.

Exercises

1. Compute the third and fourth eigenvalues of ODE $y'' = \lambda y$ for the boundary condition $y(0) = y(2) = 0$.
2. Compute the third and fourth eigenvalues for the quantum problem discussed in this section.
3. Solve for the eigenvalues and eigenfunctions of a quantum oscillator. Employ the boundary condition that the eigenfunctions vanish at $\pm\infty$. For numerical simulation, solve nondimensional Schrödinger equation in a large box.
4. Compute the eigenvalues and eigenfunctions of the equation $x^2y'' + x y' + (x^2 - a^2)y = 0$ for the boundary conditions $y(0) = y(1) = 0$.
5. Compute the eigenvalues and eigenfunctions of the equation $x^2y'' + x y' + (x^2 - a^2)y = 0$ for the boundary conditions $y(0) = 1$ and $y(1) = 0$.

CHAPTER EIGHTEEN
LINEAR ALGEBRA SOLVERS

18.1 *Solution of Algebraic Equations*

Linear algebra finds applications in all branches of science and engineering. In this book we came across a set of linear algebraic equations, for example in splines, solution of several linear ODEs, application of implicit schemes to ODE solvers (FD method), solution of Laplace and Poisson's equation. The dynamics of a linear system is described by $\dot{\mathbf{x}} = A \mathbf{x}$. We find applications of linear algebra in quantum mechanics. In electrical engineering, a complex circuit is expressed as a set of linear equations.

There are numerous numerical solvers in linear algebra. In this chapter we will discuss only two kinds of solvers: solution of linear algebraic equations and eigenvalue solvers. For detailed discussions, a reader is referred to xxx.

Solution of a set of linear equations

A system of linear equations is described as

$$A \mathbf{x} = \mathbf{b}$$

where \mathbf{x} is an unknown vector, A is a matrix, and \mathbf{b} is a known vector. For example, the equations $x_0 + x_1 = 1$ and $x_0 - x_1 = 2$ are written concisely as

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

where $A = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, $\mathbf{x} = [x_0, x_1]$ is the unknown vector, and $\mathbf{b} = [1, 2]$ is the known vector. The solution of the above equation is $\mathbf{x} = A^{-1}\mathbf{b}$, where A^{-1} is the inverse of A :

$$A^{-1} = \frac{1}{\det(A)} \text{adj}(A) \quad \dots\dots(91)$$

Here, $\text{adj}(A)$ is the adjoint of A , and $\det(A)$ is the determinant of the matrix (for a $n \times n$ matrix):

$$\det(A) = \sum \text{sign} \times a_{0,j_0} a_{1,j_1} \dots a_{n-1,j_{n-1}}$$

where $\text{sign} = +1$ for even permutations of j_i 's, and $\text{sign} = -1$ for odd permutations. Note that $\det(A)$ has $n!$ terms, hence the number of computations required to compute the determinant is $O(n^n)$, which is enormous for large n . Therefore, the inverse of a matrix and the solution of $Ax = \mathbf{b}$ are not computed using Eq. (91). Gauss elimination, to be discussed below, is one of the popular methods for solving linear equations.

Gauss elimination method

We write equation $Ax = \mathbf{b}$ with all its elements as

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots \\ a_{10} & a_{11} & a_{12} & \dots \\ a_{20} & a_{21} & a_{22} & \dots \\ \dots \\ a_{n-1,0} & a_{n-1,1} & a_{n-1,n-1} & \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

We solve for \mathbf{x} in three steps:

- Reduce the matrix A to an upper triangular matrix in the following manner.
- Solve for x_{n-1} from the equation of the last row.
- Solve for other x_i 's using back substitution.

The above steps are given below in more detail.

(a) *Gauss elimination*: First we eliminate a_{i0} for $i = 1:(n-1)$ by subtraction of (row 0) $\times(a_{i0}/a_{00})$ from row i . This operation yields

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots \\ 0 & a_{11}^{(1)} & a_{12}^{(1)} & \dots \\ 0 & a_{21}^{(1)} & a_{22}^{(1)} & \dots \\ \dots \\ 0 & a_{n-1,1}^{(1)} & a_{n-1,n-1}^{(1)} & \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_{n-1}^{(1)} \end{bmatrix}$$

with $a_{ij}^{(1)} = a_{ij} - a_{i0} \frac{a_{0j}}{a_{00}}$. The element a_{00} , which is called pivot, plays a critical role here. After this step, we eliminate a_{i1} for $i = 2:(n-1)$, and so on. This process finally leads to

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & \\ 0 & a_{11}^{(1)} & a_{12}^{(1)} & \cdots & \\ 0 & 0 & a_{22}^{(2)} & \cdots & \\ \cdots & & & & \\ 0 & 0 & 0 & a_{n-1,n-1}^{(n-1)} & \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_{n-1}^{(n-1)} \end{bmatrix} \dots\dots(92)$$

Here, $a_{ij}^{(m)}$ represents the the matrix element after m^{th} level of elimination.

(b): The last equation yields $x_{n-1} = \frac{1}{a_{n-1,n-1}^{(n-1)}} b_{n-1}^{(n-1)}$.

(c) *Back substitution:* We solve for x_{n-2} using the second last equation, which is

$$a_{n-2,n-1}^{(n-2)} x_{n-2} + a_{n-2,n-1}^{(n-2)} x_{n-1} = b_{n-2}^{(n-2)}$$

We continue this process till all x_n 's have been computed.

Let us analyse the time complexity of Gauss elimination method. The number of multiplication during the elimination is

$$(n-1)*n + (n-2)*(n-1) + \dots = O(n^3)$$

The number of multiplication in back substitution process is

$$1+2+\dots+(n-1) = O(n^2)$$

Hence, the total number of multiplication is of the order of n^3 , which is much smaller than $O(n^n)$ discussed earlier. That is why we choose Gauss elimination for solving linear equations.

Example 1: Let us solve the following equation using Gauss

elimination method.

$$\begin{bmatrix} 1 & 1 & 0 \\ 3 & 2 & 1 \\ 0 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 10 \\ 11 \end{bmatrix}$$

Step 1: row 2 – (row 1)x3

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 11 \end{bmatrix}$$

Step 2: row 3 + row 2

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 12 \end{bmatrix}$$

The final matrix is of the upper triangular form. We solve for x_2 using the equation of the third row, $4x_2 = 12$, that yields $x_2 = 3$. After this, we compute x_0 and x_1 using back substitution. The equation of the second row is $-x_1 + x_2 = 1$ that yields $x_1 = 2$. Substitution of x_1 in the equation of the first row, $x_0 + x_1 = 3$, yields $x_0 = 1$. Thus, the solution of the three linear equations is $\mathbf{x} = [1, 2, 3]$.

Gauss elimination method has several problematic issues:

1. Consider a new vector \mathbf{b} but the same A . To solve for this system, we will need to redo the whole elimination operation. Instead, it is preferable to compute the inverse of A using which we compute \mathbf{x} for any \mathbf{b} ($\mathbf{x} = A^{-1} \mathbf{b}$). The following discussion contains the procedure for computing A^{-1} in $O(n^3)$ steps.
2. The aforementioned method breaks down if any of the pivots (a_{ii}) is zero. This issue however can be salvaged by interchanging rows so that all the pivots are nonzero.

LU decomposition

Any matrix A can be decomposed into a product of a lower diagonal matrix (L) and an upper diagonal matrix (U), along with a permutation

matrix P for pivoting. That is,

$$A = PLU$$

The U matrix has the same form as that in Eq. (92), while in L , the lower part of the matrix are nonzero and the diagonals are all 1:

$$L = \begin{bmatrix} 1 & 0 & 0 & \dots \\ l_{10} & 1 & 0 & \dots \\ l_{20} & l_{21} & 1 & \dots \\ \dots & & & \\ l_{n-1,0} & l_{n-1,1} & l_{n-1,2} & 1 \end{bmatrix}$$

In this short chapter, we do not detail the process of LU decomposition that takes $O(n^3)$ operations. Refer to xxx for more details.

After the LU decomposition we can easily compute \mathbf{x} for any equation $A\mathbf{x} = \mathbf{b}$ as follows. First,

$$LU\mathbf{x} = P^{-1}\mathbf{b} = \mathbf{b}'$$

Then, we solve for \mathbf{y} from the equation $L\mathbf{y} = \mathbf{b}'$, after which solve for \mathbf{x} for the equation $U\mathbf{x} = \mathbf{y}$.

We can compute the inverse of a matrix using Given LU decomposition. We compute \mathbf{x}_i of the following equation:

$$A\mathbf{x}_i = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

The vector in the RHS has zeros everywhere, except 1 in the $(i-1)^{\text{th}}$ row. This process is carried out to compute for $i = (0, n-1)$. It is easy to show that

$$A^{-1} = [\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_i, \dots, \mathbf{x}_{n-1}]$$

Python's `scipy.linalg` module has functions to solve linear algebra

equations, as well as for LU decomposition. We illustrate these functions using the following code segments.

Codes for solving $Ax = b$ are

```
In [176]: from scipy import linalg
In [177]: A = np.array([[1,1,0], [3,2,1], [0,1,3]])
...:
...: b = np.array([3,10,11])
In [178]: x = linalg.solve(A,b)    # solves Ax = b
In [179]: x
Out[179]: array([1., 2., 3.])
```

The statement $P, L, U = \text{linalg.lu}(A)$ yields

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1/3 & 1/3 & 1 \end{bmatrix}; \quad U = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & -1/3 \end{bmatrix}; P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

You can easily verify that $PLU = A$. Note that we use `numpy.dot(A,B)` for the matrix multiplication of two matrices A and B . We can compute x using the following statements:

```
In [184]: y = linalg.solve(L, np.dot(linalg.inv(P),b))
...:
...: x = linalg.solve(U, y)
```

The other useful linear algebra functions are

- `np.det(A)` to compute the determinant of matrix A
- `np.trace(A)` to compute the trace of matrix A
- `np.inv(A)` to compute the inverse of a matrix A
- `np.norm(a)` to compute the norm of a vector or matrix a .
- `np.inner(a,b)` and `np.outer(a,b)` for computation of inner and outer products of two vectors a and b
- `np.linalg.matrix_power(A, n)` to compute A^n
- `np.linalg.expm(A)` to compute $\exp(A)$
- `np.eye(n,n)` to create an $n \times n$ identity matrix

You can easily verify these functions.

Triangular matrix

Triangular matrices have nonzero entries along the main diagonal, as well as several diagonals above and below. The lower diagonal elements can be eliminated using the Gauss elimination method, after which the matrix equation can be solved using back substitution. In this chapter we consider only one diagonal above and one diagonal below.

The elimination and backward substitution of triangular matrices have computational complexities of $O(n)$. Hence, it is more efficient to employ specialised functions, rather than usual Gauss elimination method. Python's `scipy.linalg.solve_banded(l_and_u, ab, b)` function performs the above task. The arguments of `solve_banded` are

l_and_u: A tuple containing the number of non-zero lower and upper diagonals

b: The vector **b**

ab: Banded matrix containing lower, middle, and upper diagonals.

For example, for the matrix

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 3 & 2 & 1 \\ 0 & 1 & 3 \end{bmatrix}, \text{ the matrix } ab = \begin{bmatrix} - & 1 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & - \end{bmatrix}$$

In the matrix *ab*, the top row is the upper diagonal, the middle row is the diagonal, and the bottom row is the lower diagonal. We can put any number for the – in the matrix *ab*. However, it cannot be left without any number.

We illustrate the above functions as follows:

```
In [192]: ab = np.array([[0,1,1],[1,2,3],[3,1,0]])
In [193]: b = np.array([3,10,11])
In [195]: linalg.solve_banded((1,1),ab,b)
Out[195]: array([1., 2., 3.])
```

Before closing this section we remark that Python has a special class called matrix. Matrix has same features as 2D numpy arrays.

Conceptual questions

1. State specific examples in science and engineering where linear algebra is used.
2. What is the time complexity of Gauss elimination method?
3. How many multiplication are required for solving a triangular matrix?

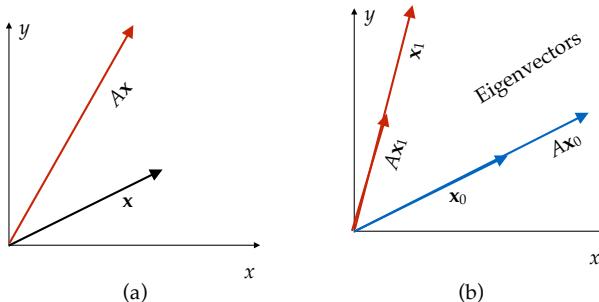
Exercises

1. Solve the following equations using Python functions: $x + y = 2$; $x - y = 3$.
2. Solve the following equations using Python functions: $2y + z = -6$; $x - 2y - 3z = 0$; $-x + y + 2z = 3$.
3. Perform LU decomposition of the matrix `numpy.array([[1,4,7], [2,5,8], [3,6,9]])`. Does it have an inverse?
4. Consider a matrix $A = np.array([[1,2,3],[2,1,4],[3,4,2]])$. Compute the determinant, trace, norm, and inverse of A . In addition, compute $\exp(A)$ and A^3 .
5. Consider five points $\{(2,1/2), (3,1/3), (4,1/4), (5,1/5), (6,1/6)\}$. Construct a spline that goes through these points. Use free-end boundary conditions.
6. Consider the matrix equation for the implicit scheme discussed in Section 15.1. Solve Eqs. (75) and (76) for $N = 8$.

18.2 Eigenvalues and Eigenvectors

Eigenvalues and eigenvector play a critical role in science and engineering applications, especially in the field of dynamical systems and quantum mechanics. In this section we provide a brief description on how to compute eigenvalues and eigenvectors of a matrix.

Typically, for a matrix A and vector \mathbf{x} , the vector $A\mathbf{x}$ is not in the same direction as \mathbf{x} . However, for some special vectors, the vector $A\mathbf{x}$ is in the same direction as the original vector \mathbf{x} . Such special vectors are called *eigenvectors*. See [Figure 107](#) for an illustration.



[Figure 107:](#) (a) For a typical vector \mathbf{x} , $A\mathbf{x}$ is not in the same direction of \mathbf{x} . (b) For eigenvector \mathbf{x}_0 , $A\mathbf{x}_0$ is in the same direction as \mathbf{x}_0 . Similarly, for the second eigenvector, $A\mathbf{x}_1$ is in the same direction as \mathbf{x}_1 .

The eigenvectors satisfy the following equation:

$$A \mathbf{x}_i = \lambda_i \mathbf{x}_i$$

where λ_i , called *eigenvalue* corresponding to \mathbf{x}_i , could be real or complex. For a $n \times n$ matrix, there are n eigenvalues, but there could be n or less than n eigenvectors. Note that all the eigenvalues need not distinct.

The eigenvalues and eigenvectors have many interesting, but we will not discuss them here. In addition, we will not discuss the QR algorithm which is often used for the computation of eigenvalues and eigenvectors.

Numpy.linalg has several functions for computing eigenvalues

and eigenvectors. They are

- `numpy.linalg.eigen(A)`: It provides the eigenvalues and eigenvectors of any matrix A .
- `numpy.linalg.eigh(A)`: It provides the eigenvalues and eigenvectors of Hermitian or real-symmetric matrix A .
- `numpy.linalg.eigvals(A)`: It returns only the eigenvalues (sans eigenvectors) of any matrix A .
- `numpy.linalg.eigvalsh(A)`: It returns only the eigenvalues (sans eigenvectors) of a Hermitian matrix A .

We illustrate the usage of the above functions using the following examples.

Example 1: Consider the following matrix:

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

Using the function `eigh(A)`, we find the eigenvalues to be 1 and 3, and the corresponding eigenvectors to be $\mathbf{x}_0 = [-1/\sqrt{2}, 1/\sqrt{2}]$ and $\mathbf{x}_1 = [1/\sqrt{2}, 1/\sqrt{2}]$ respectively. The eigenvectors are illustrated in [Figure 108](#) are perpendicular to each other.

```
In [83]: eigh(A)
Out[83]:
(array([1., 3.]), array([[ -0.70710678,  0.70710678],
 [ 0.70710678,  0.70710678]]))
```

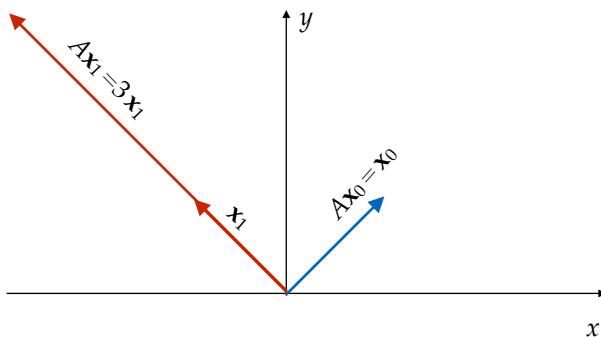


Figure 108: Illustrations of eigenvectors \mathbf{x}_0 and \mathbf{x}_1 of the matrix $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$. For the eigenvectors, $A\mathbf{x}_0 = \mathbf{x}_0$ and $A\mathbf{x}_1 = 3\mathbf{x}_1$.

Example 2: Find the eigenvalues of the matrix `np.array([[1,1,0], [3,2,1], [0,1,3]])` using Python functions.

```
In [212]: A = np.array([[1,1,0], [3,2,1], [0,1,3]])
In [213]: vals, vecs = np.linalg.eig(A)
In [214]: vals
Out[214]: array([-0.41421356,  2.41421356,  4.           ])
In [215]: vecs
Out[215]:
array([[ 0.56151667,  0.33655677,  0.22941573],
       [-0.79410449,  0.47596315,  0.6882472 ],
       [ 0.23258782, -0.81251992,  0.6882472 ]])
```

In `vecs`, the eigenvectors are arranged as columns. These vectors are orthogonal.

Example 3: Find the eigenvalues of the matrix `array([[1, 1,1], [1, 1,1],[1,1,1]])`.

```
In [216]: A = np.array([[1, 1,1], [1, 1,1],[1,1,1]])
In [217]: vals, vecs = linalg.eigh(A)
In [218]: vals
```

```

Out[218]: array([-2.37213427e-17,  8.88178420e-16,
3.00000000e+00])

In [219]: vecs
Out[219]:
array([[ 0.          ,  0.81649658, -0.57735027],
       [-0.70710678, -0.40824829, -0.57735027],
       [ 0.70710678, -0.40824829, -0.57735027]])

```

Here, the two eigenvalues are zeros.

Example 4: Let us solve $\dot{\mathbf{x}} = A\mathbf{x}$ using the properties of eigenvalues. When A has n distinct eigenvectors, it is easy to show that the general solution

$$\mathbf{x}(t) = \sum c_i \mathbf{x}_i \exp(\lambda_i t)$$

The coefficients c_i are computed using the initial conditions.

Example 5: Let us compute the largest eigenvalue of a matrix A using iteration. A vector gets stretched maximally along the eigenvector corresponding to the largest eigenvalue. We denote this vector using \mathbf{S} . Hence, after a large number of iterations, a normalised vector \mathbf{x} would be along \mathbf{S} . After that, we can compute the largest eigenvalue using $\text{norm}(\text{dot}(A, \mathbf{x})) / \text{norm}(\mathbf{x})$.

```

In [25]: A = np.array([[1,1,0], [3,2,1], [0,1,3]])

In [26]: x = np.array([1, 1, 1])

In [27]: x = x/norm(x)
In [31]: for i in range(20):
...:     x = dot(A,x)
...:     x = x/norm(x)
...:

In [32]: x  # (vector S)
Out[32]: array([0.22941573, 0.6882472 , 0.6882472 ])

In [34]: norm(dot(A,x))/norm(x)
Out[34]: 4.0000000001217835

```

Conceptual questions

1. State the importance of eigenvectors and eigenvalues in science and engineering.

2. In Example 1, show that $\mathbf{x}(t) = \sum c_i \mathbf{x}_i \exp(\lambda_i t)$ is a general solution of $\dot{\mathbf{x}} = A\mathbf{x}$.

Exercises

1. Compute eigenvalues and eigenvectors of the following matrices:
 - `numpy.array([[1,4,7],[2,5,8],[3,6,9]])`
 - `numpy.array([[1,1],[1,-1]])`
 - `numpy.ones([5,5])`
 - `np.array([[0,1],[-1,0]])`
1. Compute largest eigenvalue of the array $[[1,4,7],[2,5,8],[3,6,9]]$ using the iterative procedure discussed in the class.
2. Demonstrate using several numerical examples that the eigenvalues of a Hermitian matrix is real.
1. Solve the equation $\ddot{x} = -x - \gamma \dot{x}$ with $\gamma = 0.1$ using matrix method. Take initial conditions as $x(0) = 1$ and $\dot{x}(0) = 0$.
2. For a matrix A , compute the second largest eigenvalue and smallest eigenvalue iteratively.
3. Consider a matrix $A = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$. Compute the eigenvalues and eigenvectors of this matrix for a given θ . What does this matrix do to a vector. Illustrate it using an example.

LIMITATIONS OF COMPUTATIONS

Unexpressible: Gita

NOT COVERED

Curvilinear probs. Circle

19 *Appendix C: PFERL method*