# §1 Introduction

Computers have become a vital part of our daily lives over the last few decades. Indeed, computing devices have been used by human for thousands of years. For example, the earliest known calculating device is the early abacus that was believed to be invented in Babylon about 4000 years ago. The concept of an all-purpose, automatic, and programmable computing machine was introduced by British mathematician and astronomer Charles Babbage in the early nineteenth century. In 1936, computer scientist Alan Turing first described the principle of the modern computer in his seminal paper "On Computable Numbers, with an Application to the Entscheidungsproblem". In 1939, with the assistance of his student Clifford E. Berry, theoretical physicist John Vincent Atanasoff built the first electronic digital computer known as the Atanasoff-Berry Computer (ABC). During the 1940s and 1950s, a number of computers such as Colossus, ENIAC, and MANIAC I were built for performing scientific computation. The success of this work led to the birth of a new branch of physics called **computational physics**. Today this branch of physics becomes an essential component of research in diverse areas of physics.

## 1.1 What is Computational physics?

Roughly speaking, computational physics is the calculation of the answers to physics problems using computers. Many physics problems in the real-world are so complicated that it cannot be solved by analytical methods alone. Of course, some of these problems can be solved by doing experiment. But sometimes we seek the answers by theoretical calculations because experiments are impossible or we want to compare experimental results with theoretical predictions. Nowadays, not much calculations in theoretical physics can be done using only the tools of mathematical analysis. More commonly, we use the methods of computational physics to do part or all of the calculation, i. e. the calculation is done *numerically*.

Many elegant calculations are done in computational physics by combining mathematical analysis and numerical methods. The computer is regarded as a tool helping us to perform difficult and massive calculation. For example, finding the determinants of a $1000 \times 1000$ matrix and evaluating a complicate integral. By doing so, it has improved our understanding of physical phenomena from subatomic particles to the observable universe.

Numerical solution of ordinary differential equations is a fundamental issue in computational physics since these equations are encountered in many physics problems. Examples include projectile motion, harmonic motion, and celestial mechanics. Only trivial cases can be solved analytically. Computational physics provides the tools for solving these equations in real problems. These tools are also useful for solving ordinary differential equations in other fields such as engineering and economics.

Another important aspect of computational physics is the modeling of large complex system. For example, the modeling of stock market for the prediction of its performance and the modeling of climate system for the prediction of climate change. These problems are often solved by employing **Monte Carlo methods** which use **random numbers** to solve problems that might be non-random. These methods are simply impossible without computers and have applications which reach far beyond physics.

Other important issues in computational physics include solving nonlinear differential equations, computing integrals and derivatives, and linear algebra calculation such as matrix inversion. Computational physics again provides perfect tools to study these issues which are not only confined to physics.

Just like traditional theoreticians need a working knowledge of analytic mathematics, computational physicists need a working knowledge of **numerical methods** and **computer programming**. So let us first have an overview on computer programming in the next section.

## 1.2 Computer Programming

In order to solve a specific problem using a computer, we must instruct the computer to follow certain procedures and carry out the desired computational task. So it is necessary for us to first transform the problem into an ordered set of logical steps known as **computer algorithm** that a computer can follow. The logical steps in an algorithm can be sequential, parallel, or iterative. How to utilize the properties of a given problem in constructing a fast and accurate algorithm is a very important issue in computational science.

To inform the computer executing the algorithm for solving a problem, we need to translate the algorithm into a working **computer program**. A computer program is a collection of statements, typically written in a well-defined **programming language**. Programming languages allow us to give instructions to a computer using commands that both the computer and the programmer can understand. Every structure in a programming language has a precise form (its **syntax**) and a precise meaning (its **semantics**). Programmers often refer to programs written in human-readable programming language as **source code** and the process of writing an algorithm in a programming language is called **programming** or **coding**.

We can describe a program in a general way using **pseudocode** that can be understand by users of different programming language. Pseudocode is a description of the entire logic of an algorithm expressed in common language in which the logic is decomposed to the level of a single loop or decision. There is no systematic standard form for pseudocode. As an example, below shows the pseudocode for the calculation of the sum and average of five input numbers.

```
Initialize counter to zero
Initialize sum to zero

While counter is less than or equal to five
    Read a number from keyboard
    Add the number to sum
    Add one to counter
End While

Set average to sum divided by five
Print the value of sum
Print the value of average
```

Pseudocode helps the programmers to plan out a program before attempting to write the program in a programming language. A carefully prepared pseudocode program can be converted easily to the corresponding program in any programming language.

It is often easier for us to write a computer program by adopting a systematic approach. We can create a program by breaking down the process into stages according to the information produced in each phase as follows.

1. *Analyze the Problem*. Figure out exactly what is the problem to be solved. Try to understand as much as possible about it. You cannot begin to solve the problem until you really know what is it.

2. *Determine Specifications*. Describe clearly what your program will do. You should not worry about how your program will work at this stage. For simple programs, this involves carefully describing what will be the inputs and outputs of the program and how are they related to each other.

3. *Create a Design*. Formulate the overall structure of the program. The main task is to

design the algorithm(s) that will meet the specifications.

4. *Implement the Design*. Translate the design into a program in a specific computer language and then put the program into the computer.

5. *Debug the Program*. Check whether your program can work properly to produce the expected result. If you find any **errors** or **bugs**, then you should go back and fix them. The process of finding and resolving errors is called **debugging** a program. During this process, your goal is to eliminate any errors. So you should try everything that might cause the program to break down.

6. *Maintain the Program*. You should continue developing the program in response to the needs of the users. Most programs are never really finished and they just keep evolving over years of use.

During the debugging process, we need to eliminate any **syntax errors** and any **runtime errors** in the program so that it runs properly without any complaints. We also need to check whether the program produces the correct result. Checking a program's output is sometimes not a trivial task, but there are some general guidelines. After a program has been debugged so that it is free of syntax and runtime errors, you can perform the following actions to verify that the results of the program is correct.

1. Check if the output looks reasonable. It can be done by comparing the program's output with your predicted output as well as examining whether the output is consistent with your intuition and instincts.

2. Check if the output agrees with any available exact results. Although these results are not available for most cases, but they are sometimes available for special values of the parameters.

3. Check if the same output would be obtained for different "step sizes". Many numerical calculations involve grid-size or step-size parameters. The output of the program should be independent of the values of such parameters. This is another necessary test for the accuracy of a program.

## 1.3 Programming Languages

Over the years, programming languages have evolved from **low-level languages** to **high-level languages**. Low-level languages are used to write programs that work on the specific architecture and hardware of a computer. Programs written in these languages are machine-dependent and are not portable. **Machine language** and **assembly language** are two typical examples of low-level languages (see Figure 1.1).

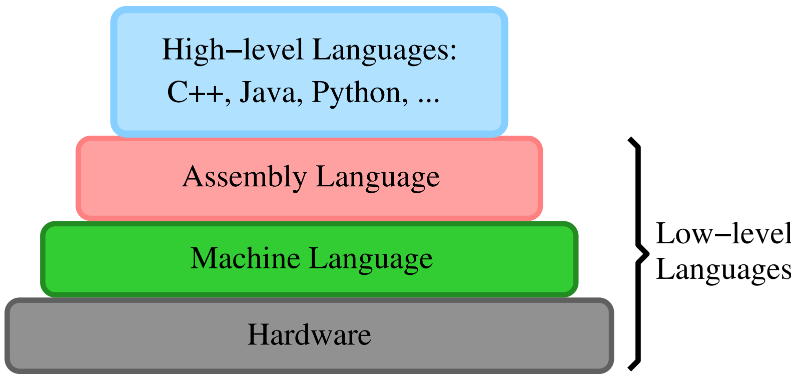
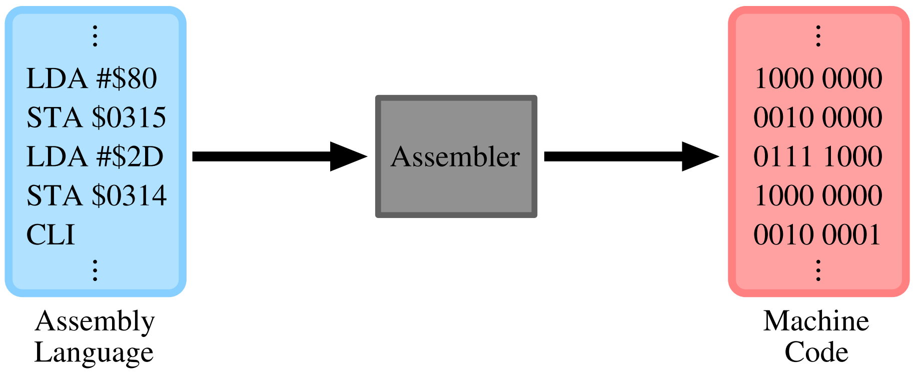


Figure 1.1: Low-level and high-level programming languages

Machine language is the native language of a particular computer that can be directly understood and executed by the computer without any conversion. It is a collection of bit sequences representing the instructions and data in a computer. Each machine language can be used only on one type of computer. A computer program written in machine language is called machine code and it runs very fast in a computer using only a very small amount of memory. However, writing programs in machine language is time-consuming and error prone due to the need of working on numerous bit sequences for machine instructions.



```
⋮
LDA #$80
STA $0315
LDA #$2D
STA $0314
CLI
⋮
```

```
⋮
1000 0000
0010 0000
0111 1000
1000 0000
0010 0001
⋮
```

Figure 1.2: Assembler converts assembly code into executable machine code

Assembly language is an improvement over machine language. Instead of using bit sequences, it represents instructions and data in a computer using **mnemonics**, i. e. English-like abbreviations. Just like machine language, each assembly language can be used only on one type of computer. Assembly code is a computer program written in assembly language and it cannot be directly understood by a computer. Assembly language use a translator program called **assembler** to convert assembly code into executable machine code as shown in Figure 1.2. This translation process is often referred to as **assembling**. By using assembly language, programmers are relieved from remembering the bit sequences for specific machine instructions. However, they still have to memorize various mnemonics that work on different types of computer.

To improve the efficiency of programming, high-level languages were developed in which programming ideas are expressed in human-oriented terms. They are not associated with the unique hardware of a computer and so can work on all computers. **C**, **C++**, **C#**, **Java**, **Python**, and **Visual Basic** are the popular high-level languages in recent years. High-level language allow programmers to write instructions look like everyday English and contain commonly used mathematical notations. A high-level language program need to be either **compiled** or **interpreted** before it can be executed on a computer.

A **compiler** is a translator program that translate a program written in a high-level language into an executable program in the machine language of some computer. When the compiled program is run, the computer outputs the result of the program based on the input as shown in Figure 1.3. Note that a compiled program can run again and again without the compiler or the source code. In addition, many compilers also have the option to produce an objective file first and then link it with other objective files and library files to produce a combined executable file.

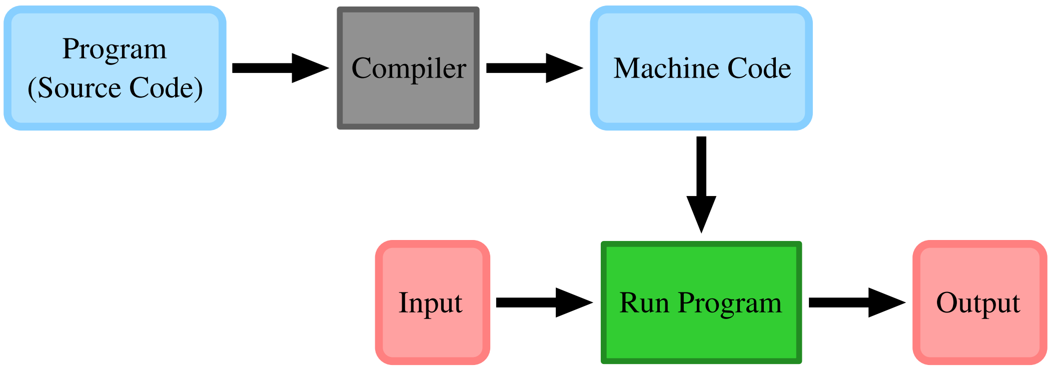


Figure 1.3: Compilation of a high-level language

An **interpreter** is a computer program simulating a computer that understands a high-level language. Instead of translating the source code into the equivalent machine code, the interpreter immediately analyzes and executes the source code instruction by instruction as necessary (see Figure 1.4). Complied programs usually run much faster than interpreted programs since the compilation is done once and for all. However, interpreted programs leads to a more flexible programming environment since they can be developed and run interactively.

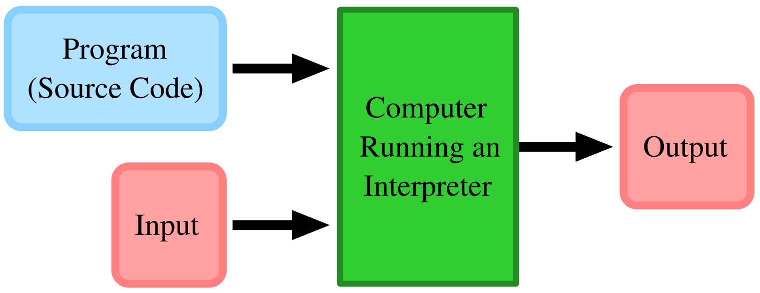


Figure 1.4: Interpretation of a high-level language

Some high-level languages such as Java and Python use a combination of compiling and interpreting processes. For example, when a Python program is executed, the compiler first converts the source code into machine-independent **bytecodes** which can be run on any computers installed with Python interpreter. Then the interpreter translates the bytecode into native machine code and immediately executes the code line by line.

## 1.4 Programming with Python

There are many popular programming languages for scientific computation such as Python, C++, **Matlab**, and **Fortran**. In this course, we will use Python which is a powerful and general-purpose programming language devised by Guido van Rossum in 1989. It is an excellent choice for computational physics for a number of reasons:

1. Python is easy to learn. The syntax and idioms used for basic operations are applied consistently in more advanced usage of the language. Error messages are generally meaningful assessments of what went wrong rather than the generic "crashes" that can occur in compiling other languages.
2. Python has clean and simple syntax which enhances the efficiency of writing programs and in general reduces the chance of creating bugs. If the programming is done right, then the result is a high quality program that is easy to maintain and extend.
3. Python and its associated libraries are free of charge and open source unlike commercial software such as MATLAB.
4. Python has a large library of **modules** (files of Python definitions and statements) and **packages** (collections of relevant modules) that extend its functionality. Many of these are available in the **standard library** provided with the Python interpreter itself. Others libraries that are widely used in scientific computing are **NumPy**, **SciPy**, and **Matplotlib** which can be downloaded separately for free.

5. Python provides cross-platform support. It is available for every commonly available platform such as **Windows**, **Linux**, and **Mac OS**. Although platform-specific extensions exist, it is possible to write code that will run on any platform without any modification.

6. Python is flexible. It is often described as a "multi-paradigm" language that contains the best features from the procedural, object-oriented, and functional programming paradigms. Not much extra effort is required to solve a problem which can only be solved easily with one of these approaches.

However, Python has some disadvantages so that it isn't suitable for every occasion.

1. In general, Python programs require a longer runtime than those programs written in some other fully compiled languages such as C, C++, and Fortran. However, the increase in runtime is insignificant in most applications and the reduced speed of execution is somehow compensated by a much faster speed of development. That is to say, it takes much less time to write and debug a Python program than to do the same thing in C, C++, or Fortran.

2. Hiding the source code of a program to avoid copying or modification by others is a difficult task in Python. Nevertheless, there are still some successful commercial Python programs available in the market.

3. The rapid development of Python has led to compatibility issues between its different versions. The most recent version of Python is version 3 although version 2 is still in active use. Moreover, Python 2 code may not work correctly under Python 3 since it is not backward-compatible. But of course you can update your code to support Python 3.

In this course, we will learn and use Python 3 since it will take over Python 2 by 2020 and it is more user friendly and more self-consistent. As the difference between version 2 and 3 are quite modest, Python 2 programs can work smoothly in Python 3 after making minor modifications. Read the details about how to make such modification on the website **https://docs.python.org/3/howto/pyporting.html**.

## 1.5 Estimation of the Value of $\pi$

To illustrate how the methods of computational physics work, let us estimate the value of $\pi$ using random numbers. Consider a quarter circle of unit radius inscribed in a unit square in the first quadrant of the $xy$-plane as shown in Figure 1.5. Obviously, the ratio of the area of the quarter circle to the area of the square is $\pi/4$. As a result, if we pick $N_s$ points randomly from the square, then the number of these points falling inside the quarter circle $N_c$ is approximately equal to $N_s\pi/4$. Thus the ratio $4N_c/N_s$ would be a good approximation to the value of $\pi$ if $N_s$ is sufficiently large.

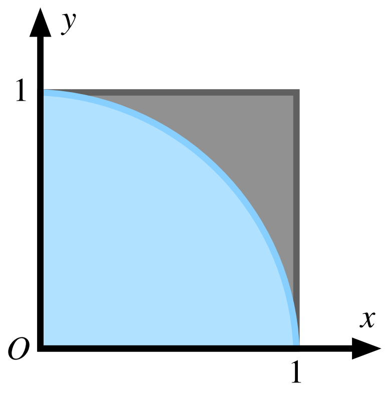


Figure 1.5: A unit quarter circle inscribed in a unit square in the first quadrant

Thus we can estimate the value of $\pi$ using the algorithm with the following pseudocode.

```
Initialize ns to ten millions
Initialize nc to zero

For every integer from 1 to ns
    Generate two random numbers x and y between zero and one
    Add one to nc if the sum of squares of x and y is not bigger than one
End For

Set pi to four times nc divided by ns
Print the value of pi
```

Below is the listing of the Python program `piestimation.py` in which the value of $\pi$ is estimated using the above algorithm.

```python
# piestimation.py
# This program estimates the value of pi using a fixed number of
# random numbers.
# Last Update on 19 Nov 2020 by F K Chow

from math import sqrt
from random import random
```

```
ns = 10000000    # Number of random points in the unit square
nc = 0           # Number of points inside the quarter circle

for i in range(0, ns):
    # Generate a random point in the unit square
    x = random()
    y = random()
    # Decide whether it is inside the quarter circle
    if sqrt(x*x + y*y) <= 1:
        nc += 1

# Compute the estimated value of pi
pi = 4*nc/ns
print('{0:10.6f}'.format(pi))
```

Table 1.1 lists the output of this program. We can see that this method quickly gives the value of $\pi$ to the accuracy of two decimal places. However, the convergence of the estimated value to a high order of accuracy is terribly slow. Therefore, we should use other algorithm to estimate the value of $\pi$ for a result with high degree of accuracy.

Table 1.1: Estimated value of $\pi$ by the program `piEstimation.py` and the corresponding error versus the number of random points $N_s$ picked from the unit square

| $N_s$ | $\pi$ | **Error** |
|---:|---:|---:|
| 10 | 2.400000 | 0.741593 |
| 100 | 3.080000 | 0.061593 |
| 1000 | 3.108000 | 0.033593 |
| 10000 | 3.141200 | 0.000393 |
| 100000 | 3.148040 | −0.006447 |
| 1000000 | 3.140412 | 0.001181 |
| 10000000 | 3.142099 | −0.000507 |