In this book, we will be discussing computational modeling of physical systems ranging from classical to quantum systems. The primary factor that makes this possible is the speed of computation offered by digital computers. In this sense, computer modeling is simply a tool to get the job done, just like the abacus or the calculator of the past. It is an important tool, enabling us to simulate a range of systems from the simple projectile motion but with added realism such as drag and spin effects to the complex behaviors of many-body problems or time-dependent quantum systems. Computational modeling is important and often indispensable in the study of these systems.

In the following, we outline a three-step strategy in computational modeling. We also briefly discuss sources of error and introduce the basic elements of Python programming with our first program as well as the required packages for computation and visualization. Installation of these packages and numerical array operations are discussed in the Appendices.

Before computational modeling can begin, there are several required steps to set it up. We generally follow a three-step approach for a given problem:

```
model building → algorithm construction → implementation
```

First, we build a model based on physical laws governing the problem. We usually aim to keep the model as simple as possible but still contains the essential physics. On the one hand, key physical concepts are best understood from the simplest of the models. On the other hand, many problems are sufficiently complex that, without simplifying approximations, they cannot be solved even with the most powerful computers available.

Second, we construct a simulation method, or an algorithm, to solve the problem based on the mathematical description of the physical model. Consideration of the operational behavior of an algorithm can often lead to physical insight about how the physical processes unfold. There is typically a trade-off in the selection of algorithms. The main criteria affecting our choice include accuracy, simplicity, stability, and robustness.

Last, we implement the algorithm in a programming language and present the results accordingly. Here again, we face several possible choices. We choose Python because, among other reasons, it is easy to write and read codes, and is open source. Python is flexible and is made more attractive and powerful for scientific computing due to the availability of external libraries for numerical simulations. For our purpose, four open-source packages are especially useful: SciPy and NumPy, an essential core base for numerical and scientific computing in Python; Matplotlib, a library for data plotting and graphical presentation in 2D and 3D; and VPython, a 3D interactive animation and graphics library.

The NumPy library adds advanced multidimensional array capability that is missing in native Python. The SciPy library provides many fundamental numerical analysis tools, among which the linear algebra routines and special functions are especially useful to us. These routines not only extend the functionalities of Python, but many of them also allow vectorized operations (implicit looping) over whole arrays. We gain the added benefits of execution speeds of compiled codes, as well as simpler and more elegant programs.
The graphics libraries enable us to seamlessly integrate visualization into our Python simulations. Data visualization is becoming an essential part of computer modeling that is increasingly data-centric. We will encounter simulations that yield a large amount of spatial and temporal data. With Matplotlib, we can produce with ease graphical representation of simulation results as simple line graphs, contour and surface plots, and vector fields. For many systems, we also use VPython to animate the process or data sets as dynamic movies. All these techniques help us to more effectively analyze and interpret the simulation results. We can do all this with these packages within four or five dozen lines of code in most cases (see Section 1.B for installation of required packages).

We make every effort to present the physical systems as self-contained models. For mechanical systems, the student should have a good grasp of physics at the introductory level or above. For quantum and thermal systems, some familiarity with the relevant concepts such as the wave function, expectation value, entropy, and so on, will be helpful. We also assume the student to have some programming background. Familiarity with Python is helpful but not required. Past experience has shown that students unfamiliar with Python can successfully learn from the examples and become productive quickly.

1.2 THE SCIENCE AND ART OF NUMERICS

Scientific computing naturally involves numerical computation (number crunching). It uses integers and real values, or floating point numbers. Integer arithmetic on the computer is exact, but its range is limited. It is unsuitable for scientific computing because our physical world encompasses a vast scale over many orders of magnitude. Floating point numbers have a much expanded range, but their operations may not be exact in the mathematical sense, so exactness is sacrificed for enlarged range. Here we briefly discuss floating point operations so we are aware of some of the limitations.

1.2.1 FLOATING POINT OPERATIONS

To check floating point operations, consider $a = \sqrt{1 - x^2}$ for some real value $|x| < 1$. We will use IPython (other interactive shells such as IDLE work as well; see Section 1.B on installing and running programs). At the prompt In [1], enter the following (in notebook mode, press Shift Enter at the end of each input cell to get output):

```
In [1]: %precision 17
x=0.6
a=(1-x*x)**(0.5)
a
Out[1]: 0.8000000000000004
In [2]: 1-a
Out[2]: 0.1999999999999996
```

The “%precision 17” keyword (also called a magic function) tells IPython to print numbers to 17 decimal places. If nothing follows the keyword, default behavior is restored (see below).

Mathematically, we should have exact values $a = 0.8$ and $1 - a = 0.2$. But the numerical values are not exact though very close, agreeing to 16 digits. We see right away that mathematical rules are bent a bit in floating point operations. This is because of the way decimal numbers (floats) are represented in the computer. A float $f$ is internally represented by

$$f = (-1)^s \times M \times 2^E,$$

(1.1)

See http://www.faculty.umassd.edu/j.wang/vp/movie.htm or the Digital Online Companion site for sample movies.
where $S$ is the sign, $M$ a fractional number between $\frac{1}{2} \leq M < 1$ (except 0), known as the mantissa, and $E$ an integer exponent. Python provides an easy way to find $M$ and $E$ for a given $f$ as follows,

```
In [3]: import math as ma # import math library
ma.frexp(0.2)
Out[3]: (0.80000000000000004, -2)
```

In the preceding code, we first import the mathematics library, rename it to `ma`, and use the Python function `frexp()` to find the mantissa and the exponent of 0.2, which are $M = 0.8$, and $E = -2$. Indeed, we have $0.2 = 0.8/4 = 0.8 \times 2^{-2}$, according to Eq. (1.1). Note the mantissa is not exactly 0.8 as seen by the computer. Why?

The reason is due to the representation of a float by a finite number of bits (see Appendix Section 1.A). According to Eq. (1.7), the mantissa is a sum of powers $(\frac{1}{2})^i$, $i = 1, 2, ..., n$, similar to a geometric series. Not all real values can fit exactly in a finite $n$-bit range (see Table 1.2). Python uses double-precision floats, which has an effective $n = 53$ bits. For a fixed exponent, the smallest difference between a pair of mantissas representable on the computer is $\sim 2^{-53} \approx 10^{-16}$. For single-precision floats, the difference is larger. Double precision, the default in Python, is most common in scientific computing, though higher precisions such as quadruple precision are used for special purposes. Regardless, a measure of precision is given by the machine accuracy $\epsilon$, defined as the largest $\epsilon$ value that the sum $1 + \epsilon$ is still 1 in floating point arithmetic (not mathematical arithmetic),

$$1 + \epsilon \equiv 1. \quad \text{(machine accuracy)} \quad (1.2)$$

Let us see what the machine thinks of this:

```
In [4]: 1. + 1.e-15 == 1.
Out[4]: False
In [5]: 1. + 1.e-16 == 1.
Out[5]: True
```

The operator "==" tests whether a condition is true or false. Evidently, the machine accuracy $\epsilon$ is below $10^{-15}$ but above $10^{-16}$.

### 1.2.2 NUMERICAL ERROR

When doing numerical calculations, we should be aware that numerical results are not always mathematically sensical. For instance, using the same example earlier with $x = 10^{-7}$, we obtain the following,

```
In [6]: x=1.e-7
   : a=(1-x*x)**(0.5)
   : a
Out[6]: 0.99999999999999500
In [7]: %precision
   : 1-a
Out[7]: 4.9960036108132044e-15
```

Although the value $a$ is as expected, the accuracy of $1 - a$ is significantly reduced compared to the approximate value $1 - a \sim \frac{1}{2} x^2 = 5 \times 10^{-15}$, a loss of more than 10 significant digits in
double precision. Further reducing $x = 10^{-8}$, the numerical answers become $a = 1$ and $1 - a = 0$. The latter is far worse than a loss of accuracy because there is a qualitative difference between a zero value and a finite but nonzero value, albeit a small one ($\sim 5 \times 10^{-17}$). This occurs because $x^2$ is less than the machine accuracy in this case, so that $a = \sqrt{1 - x^2} = 1$ in floating point operations, and we have a total cancellation error when evaluating $1 - a$.

Because floating point operations are inexact, mathematically equivalent expressions can yield different numerical results. This often leads to various tricks (on the level of art) in the quest for better numerical results. One trick to remedy the preceding situation is to rewrite it equivalently: let $b = \sqrt{(1-x)/(1+x)}$, so that $\sqrt{1 - x^2} = b + bx$, and $1 - a = 1 - b - bx$. Numerically, we have

```
In [8]: x=1.e-8
b=((1-x)/(1+x))**0.5
1-b-b*x
Out[8]: 3.922529011343002e-17
```

The preceding step improves $1 - a$ somewhat, at least it is nonzero, and on the same order of magnitude as the mathematical answer. A better way is to write $1 - a = 1 - \sqrt{1 - x^2} = x^2/(1 + \sqrt{1 - x^2})$, so that continuing from the preceding, we have

```
In [9]: x*x/(1+(1-x*x)**0.5)
Out[9]: 5.0000000000000005e-17
```

There is no cancellation involved. This result is as close as we can get to the mathematical answer within double precision.

The preceding tricks are mathematically equivalent, but numerically different. This simple example illustrates that numerics involves both science and art. Real-world examples are not as simple, but the underlying idea is valid: we need mathematical thinking as well as numerical thinking in scientific computing.

### Round-Off Error

Round-off error can occur in arithmetic operations of floating point numbers for several reasons (see Section 1.A). The floats may not be exactly representable, or some significant bits are lost during shifting. Even if the floats are exactly representable or no shifting is involved, the intermediate results could require more mantissa bits to hold the full accuracy than the working precision could provide, resulting in rounding off.

The severity of round-off error varies. The preceding example is a case of round-off errors due to cancellation of two floats $(1 - a)$ within close proximity of each other, leading to a severe loss of accuracy. We cannot eliminate round-off error. But with proper choice of methods (algorithms), we may be able to control or minimize its impact, as seen earlier.

### Truncation Error

To produce a concrete result, we normally use a practical method, or an algorithm, for the calculation. There is usually a difference between the obtained result and the exact result. This is known as the truncation error. The size of the error should be dependent on the method, and unlike round-off error, it is entirely under our control by the choice of the method.
1.2 The Science and Art of Numerics

FIGURE 1.1 Approximating the circumference of a circle by polygons.

Table 1.1 The Ratio of an \( n \)-Sided Polygon Perimeter to the Circumference of a Unit Circle.

<table>
<thead>
<tr>
<th>( n )</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P/2\pi )</td>
<td>0.900316</td>
<td>0.974495</td>
<td>0.993586</td>
<td>0.998394</td>
<td>0.999598</td>
<td>0.999899</td>
</tr>
</tbody>
</table>

As an illustrative example, let us calculate the circumference of a unit circle approximated by the perimeter of a regular polygon inside the circle shown in Figure 1.1. For an \( n \)-sided regular polygon, adding up all the sides yields the perimeter

\[
P = 2n \sin \left( \frac{\pi}{n} \right). \tag{1.3}
\]

Figure 1.1 shows two polygons, a square, and an octagon. We can see error cumulating as the arc length is replaced by straight-line segments. With increasing \( n \), we expect the approximation to be ever more accurate.

Table 1.1 lists the ratio of the polygon perimeter to the exact circumference of the unit circle for several \( n \) values. Roughly with every quadrupling of \( n \), the numerical result gains one significant digit. The method is stable and allows systematic improvement of accuracy. Most algorithms in our simulations are chosen to minimize or control truncation error.

**Stability**

Often, stability is a factor in controlling truncation error. Consider the golden mean \( \phi = \frac{1}{2}(1 + \sqrt{5}) \sim 1.618034 \). It is a root to the quadratic equation\(^2\)

\[
\phi^2 = \phi + 1. \tag{1.4}
\]

We can multiply both sides of Eq. (1.4) by \( \phi^{n-1} \) to get \( \phi^{n+1} = \phi^n + \phi^{n-1} \). This indicates a recursion relation

\[
y_{n+1} = y_n + y_{n-1}, \quad \text{with } y_0 = \phi. \tag{1.5}
\]

Recursion relations are very useful because they are simple, are fast, and can generate a series of values on the fly. Suppose we want to compute the series \( y_n \) for a range of \( n \) values by the recursion method. We set the seeds \( y_0 = 1 \) and \( y_1 = \phi \) in Eq. (1.5) and obtain \( y_2 \), then \( y_3 \), and so on. To check the accuracy against the exact results, we calculate the relative error \( |y_n - \phi^n|/|\phi^n| \) and plot the results in Figure 1.2. Over the \( n \) range shown, the error for the \( \phi \) series is on the order

\(^2\)Another form for Eq. (1.4) is \( \phi/1 = (\phi + 1)/\phi \), meaning that the ratio \( \phi \) to 1 is the same as \( \phi + 1 \) to \( \phi \). This “perfect” ratio has inspired artists and architects in history.
of \( \sim 10^{-15} \), so small that it is practically at the machine accuracy in double precision. Though there is a slight rise in the error with increasing \( n \) due to inevitable round-off, the recursion method works well in this case.

There is a second root to Eq. (1.4), \( \phi_- = -\phi^{-1} = \frac{1}{2}(1 - \sqrt{5}) \sim -0.618034 \). Like the golden mean, the negative root \( \phi_- \) also satisfies the recursion relation (1.5), so we could follow the same way as given earlier to generate a series of values for \( \phi_- \) via recursion, with the initial seeds set as \( y_0 = 1 \) and \( y_1 = \phi_- \). Repeating the calculation, we find quite a surprise: the relative error growing exponentially, and without bound. This behavior is easy to spot because we plot the results on the semilog scale, which shows the trend (a straight line) and, at the same time, clearly displays data differing over many orders of magnitude. The exponential explosion of the error cannot be attributed to round-off error. Rather, it must be due to the inherent ill-behavior of the algorithm (recursion). A significant digit is lost in roughly every 2 to 3 steps, rendering recursion unstable and useless for the \( \phi_- \) series.

We may wonder why the method works well for one series (\( \phi \)) but fails terribly for another (\( \phi_- \)). The reason is that both \( \phi \) and \( \phi_- \) are present (even a small amount caused by round-off error) in the recursion because both are solutions of Eq. (1.4). The growing \( \phi \) series completely overwhelms the vanishing \( \phi_- \) series. A small error is quickly amplified and washes out the latter in a few steps.\(^3\) The behavior is not related to, and cannot be solved by, higher float precisions. It is the property of the recursion (1.5). There is a silver lining, though: if the upward recursion is unstable, the reverse direction must be stable. We can use this fact to accurately obtain the \( \phi_- \) series by downward recursion (Project P1.3). Stability conditions of some common recursions can be found in Ref. [1].

Through these examples, we demonstrate that proper choice of algorithms is important, and algorithm-induced error (truncation error) is entirely under our control.

\(^3\)A quantitative von Neumann stability analysis is discussed later (Section 6.5.4).
1.3 Fundamentals of Programming and Visualization

straightforward and forgiving language and is easy to learn from examples [54]. Many examples are found in the included programs in the chapters ahead. Next, we discuss the basic elements of Python and introduce the relevant packages to be used: Matplotlib, NumPy and SciPy, and VPython. We assume Python 2.7x is installed first, along with the latest versions of the packages just mentioned (Section 1.B).

1.3.1 PYTHON

Like driving, the only way to learn programming is to just do it. Let us write a Python program to compute and graph one-dimensional motion with constant acceleration $a$. Taking the initial value to be zero, the position $x$ as a function of time $t$ is $x = v_0 t + \frac{1}{2} a t^2$, where $v_0$ is the initial velocity. The program is given here and results in Figure 1.3 (see Section 1.B on how to run programs).

Program listing 1.1 Motion with constant acceleration (motion.py)

```python
import matplotlib.pyplot as plt # get matplotlib plot functions

a, v0 = input('enter a, v0 (eg 1.0, 0.0) : ') # read input

t, h, n = 0.0, 0.1, 20 # init time, step size, number of steps

# time and position arrays for plotting

for i in range(n): # loop for n steps
    ta.append(t)
    xa.append(v0 * t + a * t * t / 2.0)
    t = t + h # update time

plt.figure() # start a figure; no indent

plt.plot(ta, xa, '-o') # plot data
plt.xlabel('t (s)') # add labels
plt.ylabel('x (m)')
plt.show() # show figure
```

As our first full example, Program 1.1 may feel longer than necessary. It is okay if you do not understand fully what is going on here. We will explain the program later, but after working through the rest of this chapter to the middle of Chapter 2, it will become clear. The program has three basic blocks: initialization (lines 1 to 5), computation (lines 7 to 10), and data presentation (lines 12 to end). Most programs generally follow this template.

FIGURE 1.3 The position-time curve from Program 1.1.
Program 1.1 starts by importing from Matplotlib the `pyplot` library, which is the main workhorse of 2D plotting and is short-named to `plt`. Any external library must be imported before its use. To keep separate namespaces, we will follow this convention from now on using short descriptive aliases for several libraries, including VPython as `vp` and NumPy as `np`. The program then reads (line 3, blank lines are ignored) the values $a$ and $v_0$, assumed to be in MKS units, from the keyboard via the `input()` function, which can parse multiple parameters from user input. In this case, we should enter two numbers separated by a comma (also see what happens without the comma). The next line initializes the time, step size, and number of steps, the first two being floats, and the last one an integer. We can assign values to multiple variables on a single line, provided the number of variables on the left is equal to the number of values on the right (called sequence unpacking). This saves print space without sacrificing clarity. In Python, variables need not be declared before assignment. The type of a variable is automatically derived from the assigned object. The same variable could be assigned to a different object later, changing its type. The last line before the loop creates two empty lists to store time and position values, rounding up the initialization block. For readability, it is usually a good idea to group similar constants and parameters together.

After the initialization statements, the `for` loop starts at line 7 and terminates just before the blank line. The `range()` function specifies the number of iterations. Unlike other languages where the programmer controls the appearance of logical blocks for easy identification, Python requires vertical alignment (indentation) for block control, including `while` loops, `if` conditions, function declarations `def()`, and other logical blocks. The next three lines, indented by the same amount, belong to the `for` loop. Inside the loop, the current time and position are appended to the respective lists `ta` and `xa`, and time is incremented. Like the append method, there is a host of methods for manipulating lists, including insert, remove, pop, and so on.

The rest of the program graphs the results (Figure 1.3, $a = 1 \text{ m/s}^2$, $v_0 = 0$). We first open a figure (line 12, it may be omitted if there is only one figure in the program) and plot the data with `plot()` from the Matplotlib library. The plot function accepts the required pair of $(x, y)$ data arrays, plus an optional string specifying color, line style, or symbol types (see Section 1.C). After adding axis labels, the plot is displayed on the screen, ending the program. We can manipulate the figure, zoom in, drag around, or save to a file.

**Conditional Controls**

Besides the “for” loop above, Python has a conditional “while” loop (see Program 2.1)

```python
while (condition):
    ....
```

The conditional `if` statement takes the form (also see Program 2.1)

```python
if (condition):
    ....
elif (condition):
    ....
else:
    ....
```

---

4. There seems to be a specialized library for any task. Want to fly? Try `import antigravity`.
5. Some may prefer to use more descriptive names. We try to use short names if they do not lead to confusion. Here, `ta` and `xa` designate time and position arrays.
Both the `elif` and `else` are optional, and there can be multiple `elif` statements. There is also an inline `if-else` expression that yields a value depending on the condition:

```python
x = a if condition else b
```

In the preceding statement, `x` will be equal to `a` if the condition is true and equal to `b` otherwise (see Program 5.1 for an example).

**Function Definitions and Variable Scope**

We can define functions with the `def` keyword as (see Program 2.5)

```python
def func(x1, x2=1.0):
    ....
    return y1, y2
```

The function may accept input parameters with default values (`x2 = 1.0`). Unlike traditional programming languages, Python functions can return multiple values or none at all. All variables defined inside the function are local and inaccessible outside. The function can access any global variables defined in the main program, but cannot modify them. If the latter is necessary (should be rarely used and very well controlled, see Program 9.2), the variables to be modified must be declared as `global` inside.

**Procedural and Object-Oriented Programming**

With few exceptions, we choose procedural programming for its direct approach to problem solving. We can set up a problem quickly and solve it by building a series of compact functions. There are cases when it is more convenient to use object-oriented programming, such as Brownian motion (Program 10.2), Einstein solids (Program 11.1), and VPython modules (Program S:6.4).

Object-oriented programming combines both methods (i.e., functions) and data into stand-alone entities called objects. It may be more logical and suitable for building models that are expandable (see Program 11.1 and Project P11.2). For example, it would be natural to define particle objects with properties such as position and velocity. When combined with an integration method (Chapter 2), one could instantiate one particle object for projectile motion (Chapter 3), or another one for planetary motion (Chapter 4), all within a unified approach. One could even extend it to \( N \)-body problems. The interested reader may wish to explore this object-oriented approach.

**File Handling**

Among Python file handlers, the easiest way may be to read from and write data to files via the `pickle` module. Structured data records can be loaded or dumped easily (see Program 9.7 for an example). A quick way to handle text files can be found in Program 4.4.

**Python 2.7x and 3.xx Compatibility**

The programs presented assume Python version 2.7x, though they should run without problems in version 3.xx, provided the required packages are 3.xx specific. For our purpose, there are two incompatibilities between the two versions that affect us: print and division operations. First, the `print` statement in 2.7x becomes a function in 3.xx (see Program 3.5), and second, divisions are carried out as floats in 3.xx. To retain forward compatibility, the first line of a 2.7x program should be

```python
from _future_ import print_function, division
```
Python 2.7x treats divisions according to operand types, but 3.xx defaults divisions to float operations. For example, in 2.7x, we have (using IPython)

```
In [1]: 1/2
Out [1]: 0
In [2]: 2.0**(1/2)
Out [2]: 1.0
```

The last result was probably unintended even in 2.7x and a source of unintentional human error. To always use float division like in 3.xx, add the compatibility statement as

```
In [3]: from __future__ import print_function, division
   : 1/2
Out [3]: 0.5
In [4]: 2.0**(1/2)
Out [4]: 1.4142135623730951
```

We believe the programmer should be explicit about whether integer or float divisions are intended. For instance, the following works in both versions.

```
In [5]: 1//2  # integer division
Out [5]: 0
In [6]: 2.**(1./2.)  # float division
Out [6]: 1.4142135623730951
```

We follow the preceding practice in our programs. The included programs run correctly under 3.xx without relying on the division compatibility.

**Built-In Help**

Python has an online help system that can provide information on keywords, modules, and so on. It is invoked by

```
In [7]: help()
help>
```

Enter terms at the help> prompt, for example, `range` or `numpy`, to get help on them. Type `q` to return to the prompt or exit help.

**1.3.2 NumPy AND SciPy**

At its core, the NumPy library offers advanced multidimensional array creation and manipulation functions. In concert with Python, they operate at a high and often abstract level and execute at the speed of compiled codes. They form a unique and powerful combination for scientific computing. Many other libraries require NumPy, including Matplotlib and VPython. Appendix 1.D introduces the basic operations on NumPy arrays. We also use elementary mathematical functions in NumPy, many of which are the so-called universal functions (ufunc), functions that operate on scalar and NumPy array inputs transparently (see Program 4.6 for examples).

We will mainly use the SciPy library for solving linear systems and eigenvalue problems. It has highly specialized routines for linear algebraic problems, including efficient banded matrix
solvers (see Program 8.3). SciPy also offers many special functions useful to us, such as Legendre and Hermite polynomials, spherical Bessel functions, and so on. Like NumPy functions, many SciPy routines are universal functions.

**Speed Boost with Language Extensions**

Though easy to learn and use, flexible yet powerful, Python by itself was not designed for sheer number crunching, a core task in scientific computing. Being an interpretive language, Python parses and translates each statement it encounters, one at a time, into machine instructions. Therefore, explicit, nested looping is slow and expensive, making the use of NumPy array operations practically indispensable in many simulations. Programs 6.7 and 8.2 are such examples where the proper use of NumPy arrays speeds up the code by 50 times or more over explicit looping (see also Programs 6.8 and 7.2 and Section 8.2.2).

Sometimes the speed problem cannot be solved so easily with NumPy alone. In most cases, a small part of the code hogs most of the computing time. Code profiling (Section S:8.2.B) can tell us where the bottleneck is. Are there ways to alleviate, even eliminate, the bottleneck without losing the benefits of Python? The answer is yes: with language extensions. There are several such easy-to-use extensions. Numba is the easiest package that can compile a standard Python function at runtime. It does not require us to change the code or use a different language. We just put the so-called decorator symbol @jit in front of a function, and almost by magic, it runs faster, sometimes by a lot. So Numba is nearly effortless to use. This is shown in a simple example in Program 5.7 for fractals. Cython is a package providing a simple interface so Python can easily call compiled C/C++ codes. Another package is Weave, part of the SciPy library, that can turn inline expressions into C/C++ codes. For Fortran, one can use F2Py, which is part of NumPy. F2Py can convert Fortran codes into Python callable binary modules.

For example, molecular dynamics (Section 11.4) is a typical case where the bottleneck is localized in the computation of forces among the particles (Program 11.7). When we use F2Py extensions to convert that part of the code to Fortran, we get a handsome payoff: a 100x speed gain (see Programs S:11.3 and S:11.4).

When we have a critical speed bottleneck, one of the aforementioned extensions should solve the problem. With just a little effort, we can have the best of both worlds.

**1.3.3 Matplotlib**

Matplotlib is a Python library for data plotting and graphical presentation in 2D and 3D. It is easy to make quality graphs with Matplotlib, interact with them like zooming, and save to standard file formats. At the same time, almost every aspect of a Matplotlib plot is customizable, making it possible to do hard or odd things, such as twin axes with different scales or even data animation (see Figure 3.16, Programs 8.1 and 10.2, and Matplotlib gallery for examples).

Almost all graphs of simulation results in the book are made in situ with Matplotlib. Source codes for a graph can be used as a template for similar types (e.g., Program 4.6 for surface plots), with the user substituting data only. This saves us time as we can simply copy and paste that part of the code. Furthermore, Matplotlib functions should be familiar to Matlab users.

Matplotlib also works in IPython notebooks, so graphs can be embedded in an interactive document. This is illustrated in Figure 1.4. The statement “%matplotlib inline” in input cell 1 tells IPython to show Matplotlib output within the notebook. The first line of input cell 2 imports pylab, which combines both pyplot and NumPy into a single entity (namespace), effectively importing two packages at once. The next line divides the interval $[-10, 10]$ into 100 points using the NumPy function linspace, which returns the points in a NumPy array assigned to $x$ (see Program 4.6). The last line plots the function $\sin(x)/x$, which is shown immediately below. Here, the plotted function accepts an array input automatically, an example of a ufunc. We can also graph functions symbolically with SymPy (see Section 1.B).
FIGURE 1.4 Matplotlib and IPython notebook.

It is also possible to load an existing program such as Program 1.1 in input cell 2. This way, we have a flexible, interactive environment within which both simulation and visualization can be done.

1.3.4 VPython

VPython is a graphics programming library for Python. It makes manipulation and dynamic animation of illuminated three-dimensional objects easy and accessible to the end user. It offers a rich and interactive environment to render the simulation process or data sets as dynamic movies on the screen. Program 2.1 shows a simple example of a bouncing ball (Figure 2.1). We use VPython extensively. In nearly all cases, animation can be turned off (commented out) without affecting the nonvisual aspect of the simulations, or replaced with either Matplotlib output (see Exercise E2.1) or data files for separate analysis.

Starting from VPython 6, a frame-limiting function `rate()` is required in the animation loop for it to function correctly across different computer platforms due to technical reasons. This normally does not pose a problem because without it, most animations would run too fast given the speed of today’s computers. In cases where we need all the speed we can get (e.g., restricted three-body motion, Program 4.7), we can set the limit very high.

VPython programs are most robustly run by themselves (see Section 1.B on ways to run programs). Very recently, however, it has become possible to run VPython animation inline within IPython notebooks, as is the case with inline Matplotlib. To do so, just replace VPython with a wrapper called IVisual (see Section 1.B) and create VPython objects as usual. Figure 1.5 shows a spinning cube in the IPython notebook. The cube is created using the VPython box object with

\[6\]

If the animation does not show in the notebook, restarting the IPython kernel usually fixes it.
default attributes because none is specified in the argument, so it appears as a cube with equal length, height, and width (see Program 2.1). Inside the while loop, the rate statement limits the animation to about 20 loops per second. We make the cube spin using the rotate method, which rotates an object for the specified angle (radians) about an axis vector (see Program 3.8 for a spinning baseball).

Run the example, and interact with the animation. You can view the scene from different perspectives by right-button drag (or Ctrl-drag) to change the camera angle. You can also zoom in or out by middle-button drag (Alt-drag, or left+right on a two-button mouse).

VPython also defines vector objects and operations, including cross and dot products. We make use of these capabilities (see Programs 3.8, S:6.2, and 7.5).

1.3.5 DEBUGGING AND OPTIMIZATION

It is rare not to make programming errors when writing functional programs of more than a dozen or so lines. Debugging can be challenging and time-consuming. Syntax errors are easy to find, but logical errors can be hard to trace. A few things can make debugging easier. Write modular and readable codes, test them often and in isolation if possible, and comment out, rather than delete, the changed lines (delete only when finished). Python is forgiving because it will print out the lines where errors occurred, but the chain-error messages can be bewildering. Try to find the error from the earliest line in the error chain, and print the values of the variables in question. For really hard-to-find bugs, try the integrated debugger built in the Python shell, IDLE. It allows stepping through the code and examination of variables.

Optimization, if necessary, should be done once a working program is at hand. It should keep the loss of clarity and readability to a minimum. Before we start optimizing, find the bottleneck by profiling first (Section S:8.B).
We outlined the three-step strategy in computational modeling, namely: model building, algorithm construction, and implementation. We discussed floating point numerics and its limitations, sources of error, including round-off and truncation errors. We very briefly described the basic Python elements, and the required libraries: Matplotlib, NumPy and SciPy, and VPython. Familiarity with these libraries comes from using them, which has plenty of help in the chapters ahead.

We list below several general references on computational physics. A comprehensive bibliography is included at the end of the book.

- P. DeVries and J. Hasbun, *A first course in computational physics*, 2nd ed. (Jones & Bartlett, 2010) [24]. An introductory computational physics text that emphasized graphical representation of data even in the early days of the first edition, with the graphics-challenged Fortran language. This edition uses the MATLAB environment.

- N. Giordano and H. Nakanishi, *Computational physics*, 2nd ed. (Benjamin Cummings, 2005) [34]. A general computational physics book that uses both Fortran and C as the programming languages. Most examples are available online only.


- M. Newman, *Computational physics*. (CreateSpace Independent Publishing) [68]. A computational physics text using the Python programming language and the graphics libraries, including VPython.

- T. Pang, *An introduction to computational physics*, 2nd ed. (Cambridge University Press, 2006) [70]. A text on traditional computational physics topics as well as special topics such as generic algorithm. Sample programs are given in Java.

- J. Thijssen, *Computational physics*, 2nd ed. (Cambridge University Press, 2007) [89]. A text aimed at graduate level with advanced topics such as quantum physics. Examples in Fortran or C are available online only.

**Exercises**

**E1.1** Find the exponent and mantissa of the following floating point numbers: 0.5, 1.0, 5.0, 10^{10}, and 10^{-10}.

**E1.2** Write down the bit patterns of the following fractions in their fixed-point format (1.6), keeping only the first 4 most-significant bits.

(a) 1/4; (b) 4/5; (c) 5/8; (d) 3/7.

**E1.3** Write a program to find the machine accuracy \( \varepsilon \) on the computer you are using, to within a factor of 2.

**E1.4** (a) Use the sample code with \texttt{struct} (Section 1.4) to find the mantissa bits in double precision (53 bits including the phantom bit) for the floats in Table 1.2. Replace ‘f’ and ‘i’ by ‘d’ and ‘q’, respectively.

(b) Find the exponent of each float. The machine typically includes a bias to the exponent. What is the bias?
E1.5 Derive an expression for the area of a regular $n$-sided polygon inside a unit circle. Approximate the area of the unit circle by the area of the polygon, and calculate the truncation error for the $n$-values listed in Table 1.1.

Projects

P1.1 Write a program to convert a floating point number to its bit pattern. The program should take the input from the user and print out the bit pattern (sign, and the rest in groups of 4 bits) for the float, assuming single precision. Test your program with the results shown in Table 1.2. See if you can break the program into logical modules.

P1.2 Consider the roots of the quadratic equation $ax^2 + bx + c = 0$,

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}$$

(a) Write a program to calculate the roots, taking $a, b, c$ as input from the user, and printing out the answer to the screen. Choose any of the preceding equivalent expressions. Test it on the following numbers:

$$a = 3.0 \times 10^{-3}, \ b = -2.0, \ c = 10^{-4},$$
$$a = 1.2 \times 10^{-17}, \ b = 5.5, \ c = 10^{-13}.$$ 

Are the results correct for all cases? What kind of numerical error is responsible?

(b) To control the error, modify your program so that it correctly computes the roots to within machine precision, no matter the values of $a, b, \text{and} c$.

P1.3 We showed that the recursion relation (1.5) is unstable in the upward direction for $\phi_+ = \frac{1 - \sqrt{5}}{2}$. Thus, it should be stable in the downward direction, $y_{n-1} = -y_n + y_{n+1}$.

Iterate the downward recursion for $N$ times, say $N = 50$, starting with arbitrary seeds, for example, $y_{50} = 0$ and $y_{49} = 1.0$. Normalize your series so $y_0 = \phi^0 = 1$ because the seeds are arbitrary. Compare with the exact results ($\phi^n$). Also plot the relative error as in Figure 1.2. Discuss why it works.

Because the downward recursion is equally valid for the golden mean $\phi = \frac{1 + \sqrt{5}}{2}$, the series obtained earlier should also be applicable to $\phi^n$. Calculate and plot the relative error on the same graph. Why does it not work? Given that the seeds are the same, how does the algorithm “know” what series to converge to?

Fixed Point Numbers

We are familiar with bit patterns representing integers. They can also be used to represent fractional numbers, if we imagine a fixed point placed in front and interpret the $n$-th bit as a fraction $2^{-n}$.

For example, consider the $n$-bit binary pattern $X_{bin}$ given by

$$X_{bin} = .b_1 b_2 \cdots b_n,$$  \hspace{1cm} (1.6)
where \( b_i \) is either 0 or 1. Interpreted as above, the decimal fraction number \( X_{\text{dec}} \) can be written as
\[
X_{\text{dec}} = \sum_{i=1}^{n} b_i 2^{-i}.
\] (1.7)

The range of \( X_{\text{dec}} \) is \( 0 \leq X_{\text{dec}} \leq 1 - 2^{-n} \), and the smallest increment representable is \( 2^{-n} \). By increasing the number of bits \( n \), the increment will be reduced, and finer fractional values, or greater precision, can be achieved. It is also possible to shift the fixed point to different positions to represent a mixture of whole numbers plus fractions. In addition, by adding a sign bit, negative values can be included using two’s complementary binary representation.

**Floating Point Numbers**

Floating point numbers are natural extensions of fixed point numbers. As Eq. (1.1) shows, a nonzero real number \( f \) can be decomposed as \( f = M \times 2^E \) where the mantissa \( M \) is a fraction between \( \frac{1}{2} \leq |M| < 1 \), and the exponent \( E = \text{ceiling}(\log_2 |f|) \), the smallest integer above \( \log_2 |f| \). For example, the number 0.5 has a mantissa \( M = 0.5 \) and exponent 0, and 1.0 has the same mantissa but a different exponent of 1.

Typically, a floating point number is stored in a finite number of bits. For example, single-precision floats use 32 bits (4 bytes) as
\[
S \cdot \underbrace{||| \quad ||| \quad ||| \quad ||| \quad ||| \quad ||| \quad ||| \quad |||}_E \quad \underbrace{||| \quad ||| \quad ||| \quad ||| \quad ||| \quad ||| \quad ||| \quad |||}_M.
\] (1.8)

including 1 bit for the sign, 8 bits for the exponent interpreted as a signed integer between \([-128, 127]\), and 23 bits for the mantissa as a fraction according to Eq. (1.6). Because the mantissa has a lower bound \( M \geq \frac{1}{2} \), the first bit (most significant) of the mantissa in Eq. (1.8) is always 1, and by convention, it is normally not stored explicitly (so-called phantom bit). The mantissa actually represents the next 23 bits, for an effective 24-bit length. This means that single-precision floats are accurate to \( \epsilon = 2^{-24} \sim 6 \times 10^{-8} \), or about 7 digits. The value \( \epsilon \) is called the machine accuracy in Eq. (1.2). The range of single-precision floats is \([2^{-128}, 2^{127}]\), or roughly \(10^{-38}\) to \(10^{38}\).

Double-precision floats, the default in Python, use 64 bits (8 bytes): 1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa (53 effective bits). The machine accuracy for double precision is \( \epsilon \sim 10^{-16} \) (15 digits), and the range is about \(10^{-308}\) to \(10^{308}\). Results exceeding the range will cause under- or overflow.

We can write a program to calculate the mantissa binary pattern for a given float (Project P1.1). However, like the earlier case about the exponent and mantissa (Section 1.2.1), there is an easier way provided by Python via the `struct` conversion module. The following code demonstrates this.

```python
In [1]: import struct as st
   fpack=st.pack('f',6.e-8)  # pack float to bytes
   fint=st.unpack('i',fpack)[0]  # unpack to int
   bin(fint)[-23:]  # mantissa bits
Out[1]: '00000001101100101011001'
```

This code converts a float (\(6 \times 10^{-8}\) in the example) to a byte pattern (string) in memory, which is subsequently converted to an integer. The mantissa is contained in the last 23 bits of the integer in binary representation, excluding the phantom bit. Several other examples including the
### Table 1.2 The Exponent $E$ and Mantissa $M$ of Single-Precision Floats.

<table>
<thead>
<tr>
<th>$\text{float}$</th>
<th>$E$</th>
<th>$M$ (decimal)</th>
<th>$M$ (binary including phantom bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0</td>
<td>4</td>
<td>0.625</td>
<td>1010 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>1.0</td>
<td>1</td>
<td>0.5</td>
<td>1000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>0.1</td>
<td>-3</td>
<td>0.8</td>
<td>1100 1100 1100 1100 1100 1101</td>
</tr>
<tr>
<td>$6 \times 10^{-8}$</td>
<td>-23</td>
<td>0.50331648</td>
<td>1000 0000 1101 001 0101 1001</td>
</tr>
</tbody>
</table>

phantom bit are listed in Table 1.2. The decimal exponent and the mantissa are obtained from `frexp()` explained earlier (Section 1.2.1).

When two floats are to be added, the exponent of the smaller number is adjusted upward so it is equal to the exponent of the larger number, and the remainders are added. For example, suppose we wish to add $1 + 6 \times 10^{-8}$. Using the $E$ and $M$ values from Table 1.2, we have

$$1 + 6 \times 10^{-8} = 0.5 \times 2^1 + 0.50331648 \times 2^{-23} = 2^1 (0.5 + 0.50331648/2^{24}).$$  \hspace{1cm} (1.9)

In machine operation, the second mantissa in the parenthesis in Eq. (1.9) will be shifted to the right 24 times,\(^7\) with the most significant bit (left most) replaced by 0 each time. In single precision, the effective mantissa is zero after the shifts because all the bits are flushed to the right. Consequently, the result from Eq. (1.9) is (numerical, not mathematical)

$$1 + 6 \times 10^{-8} = 2^1 \times 0.5 = 1.$$ \hspace{1cm} (1.10)

We see from the preceding example that, in general, addition of two floats whose relative magnitudes differ by more than the machine accuracy will cause the smaller float to be neglected, and the result is equal to the larger float. Even when the floats are within the machine accuracy, there is usually a loss of accuracy due to the flushed bits. This is the round-off error. Round-off errors also occur in other arithmetic operations even if no shifting is involved because the finite mantissa may not be sufficient to hold the resultant answer. In particular, cancellation error from the subtraction of two floats within close proximity of each other could lead to a complete loss of significant digits.

---

In addition to Python 2.7x, we need the following four packages: NumPy, SciPy, Matplotlib, and VPython. Optional packages include IPython, SymPy (both recommended), and IVisual. IPython is a user-friendly Python shell (see following). SymPy, a symbolic mathematics library for Python, is not used in our numerical computation, but is handy nonetheless for many tasks such as algebra, exercises (see Exercise E2.7 and Exercise E3.10), and so on.

### Python Distributions

The easiest way to get a full installation up and running quickly is through all-in-one distributions such as:

- Python(x,y), https://code.google.com/p/pythonxy/ (Windows only);

---

\(^7\)In binary arithmetic, dividing by 2 is equivalent to shifting the bits to the right by one position.
Each distribution contains all the required packages (and more) listed here, except Anaconda and Canopy, which do not include VPython as of this writing. To add VPython to Anaconda, enter the following conda command

```
$ conda install -c mwcraig vpython
```

This will install VPython built for Anaconda by Matt Craig on both Mac OS and Windows. If you are using Canopy on the Mac OS, there is a ready-made binary installer at vpython.org.

The Python(x,y) and WinPython distributions do include VPython but are for Windows only. WinPython is also portable.

**Custom Installation**

In general, use binary installers if possible, and stick with one version (32- or 64-bit) consistently. All required packages have binary installers at their web sites built for the official python.org Python (Table 1.3). Binaries built for other Python installations including all-in-one distributions are usually not compatible. The online resource of the book has updated installation instructions and the needed binaries for Windows at:

http://www.wiley.com/college/wang

You can also use the `pip` tool to install packages from Python Package Index, https://pypi.python.org/, for any platform including Linux. The typical command line is $ `pip install xyz` for installing package `xyz`. Note that `pip` (included starting Python 2.7.9) is in the `Scripts` directory of a Python installation, so make sure it is in the command path (C:\Python27\Scripts\ on Windows).

The `pip` tool can also install packages in the “wheel” format, for example, $ `pip install xyz.whl`. For Windows, many binaries built by Christoph Gohlke for the official Python including Matplotlib dependencies can be found at www.lfd.uci.edu/~gohlke/pythonlibs/ (referred to as UCI).

Table 1.3 lists the recommended order to install only the needed packages individually (file names are for Windows or Mac at the time of writing, always use the most current versions). Check each package’s installation instructions, and test the installed package before moving on to the next one. When running into a brick wall, ask the instructor or colleagues for help.

To check the installed packages, issue this from a command terminal,

```
$ pip list
```

or start IDLE and enter the following

```
>>> help('modules')
```

This will list the locally installed packages. Use `pip` to install missing dependencies (if any) or other packages.

**Running Programs**

If you have installed the required packages successfully, congratulations! You are about to enter the exciting world of simulations, and you can now run all the programs discussed in the book.
### Table 1.3 Package Installation Guide (W=Windows, M=Mac OS).

<table>
<thead>
<tr>
<th>Package</th>
<th>Files to download or method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>W: python-2.7.8.msi</td>
</tr>
<tr>
<td>python.org</td>
<td>M: python-2.7.8-macosx10.6.dmg</td>
</tr>
<tr>
<td>pip</td>
<td>Download get-pip.py from the site, then run it: $ python get-pip.py (from a command terminal)</td>
</tr>
<tr>
<td>pip.pypa.io</td>
<td></td>
</tr>
<tr>
<td>VPython</td>
<td>W: VPython-Win-32-Py2.7-6.10.exe</td>
</tr>
<tr>
<td>vpython.org</td>
<td>M: VPython-Mac-Py2.7-6.10.dmg</td>
</tr>
<tr>
<td>NumPy</td>
<td>W: numpy-1.8.2-win32-superpack-python2.7.exe</td>
</tr>
<tr>
<td>numpy.org</td>
<td>M: $ pip install numpy</td>
</tr>
<tr>
<td></td>
<td>This can be skipped because VPython includes NumPy.</td>
</tr>
<tr>
<td>SciPy</td>
<td>W: scipy-0.14.0-win32-superpack-python2.7.exe</td>
</tr>
<tr>
<td>scipy.org</td>
<td>M: scipy-0.14.0-py2.7-python.org-macosx10.6.dmg</td>
</tr>
<tr>
<td>Matplotlib</td>
<td>W: matplotlib-1.4.0.win32-py2.7.exe; requires python-dateutil, pyparsing and six (binaries at UCI)</td>
</tr>
<tr>
<td>matplotlib.org</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M: matplotlib-1.3.1-py2.7-python.org-macosx10.6.dmg</td>
</tr>
<tr>
<td>IPython</td>
<td>W: ipython-2.4.0-py27-none-any.whl (from UCI)</td>
</tr>
<tr>
<td>ipython.org</td>
<td>W or M: $ pip install ipython</td>
</tr>
<tr>
<td>SymPy</td>
<td>W: sympy-0.7.5.win32.exe</td>
</tr>
<tr>
<td>sympy.org</td>
<td>M: $ pip install sympy</td>
</tr>
<tr>
<td>IVisual</td>
<td>Install from the command terminal</td>
</tr>
<tr>
<td>python.org/</td>
<td>$ pip install ivisual</td>
</tr>
<tr>
<td>pypi/IVisual</td>
<td>Optional. Some VPython codes may not run properly.</td>
</tr>
</tbody>
</table>

The most direct way to run a program is from a command terminal. For instance, the following will run Program 1.1, motion.py⁸

```bash
$ python motion.py
```

One can also navigate to the folder of source programs and click on them. Programs can be prepared with a text editor and be executed this way.

To run Python interactively, we can use IPython, which is an interactive shell for Python. It can act like an interactive calculator and also integrate codes and other media like graphics in the notebook mode running in a browser (see Figure 1.4). IPython has many other useful features, including online help and tab completion. Start IPython (in Scripts path) in either the standard line mode, or the notebook mode (without [])

```bash
$ ipython [notebook]
```

---

⁸On Windows, the keyword “python” can be omitted. Any program entered at the command line (cmd.exe) with .py extension should automatically invoke Python.
Once in IPython, enter statements line by line (in notebook mode, press \texttt{Shift Enter} at the end of each input cell to execute the cell). For example, you can test SymPy with

\begin{verbatim}
In [1]: %matplotlib inline
In [2]: from sympy import *
x = symbols('x')
plot(sin(x)/x)
\end{verbatim}

and you should see the same output as Figure 1.4.

We can also load an existing program in IPython

\begin{verbatim}
In [1]: load motion
\end{verbatim}

This will load \texttt{motion.py}, Program 1.1, to be edited or executed.

An alternative to IPython is the standard Python integrated development environment (IDLE), with an editor and a debugger. From IDLE, we can write a new program, open an existing one, and run it from a menu-driven system.

VPython also comes with its own shell called VIDLE. It offers several improvements over IDLE, including more graceful handling of exceptions when Matplotlib and VPython functions are used simultaneously. For maximum reliability, run VPython programs from the command line or from IDLE or VIDLE.\footnote{One can also run most VPython programs without VPython installation in a web browser via GlowScript (glowscript.org) or capture output to a movie (see vpython.org).}

\section{The Matplotlib \texttt{plot()} Function}

The \texttt{plot()} function from Matplotlib produces line graphs. It is invoked by

\begin{verbatim}
plt.plot(x, y, string)
\end{verbatim}

where \(x\) and \(y\) are the data set, and \texttt{string} is an optional string specifying the color, line style, or marker type of the curve. There are also many other optional arguments. The string is normally a three-character field, \texttt{CMS}, for color, marker, and line style, respectively. Refer to Table 1.4 for common codes. The order of the character in the string is unimportant. For instance, the string \texttt{g-o} means a green solid line with filled circle markers.

Axis labels as well as text can be added. For example,

\begin{verbatim}
plt.text(xloc, yloc, 'message', rotation=45)
\end{verbatim}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
\textbf{color} & \texttt{b} & \texttt{g} & \texttt{r} & \texttt{m} & \texttt{c} & \texttt{y} & \texttt{k} & \texttt{w} \\
\hline
\texttt{blue} & \texttt{green} & \texttt{red} & \texttt{magenta} & \texttt{cyan} & \texttt{yellow} & \texttt{black} & \texttt{white} \\
\hline
\textbf{marker} & o & s & x & + & D & * & ^ & \texttt{\textdagger} \\
\hline
\textbf{style} & solid & dashed & dots & dash-dot \\
\hline
\end{tabular}
\caption{Common Line Properties for the Matplotlib \texttt{Plot()} Function.}
\end{table}
would put the “message” at \((x_{\text{loc}}, y_{\text{loc}})\), and rotated 45° counterclockwise. Often, the font size looks fine on the screen, but may be too small for print after resizing. Such parameters can be specified via the optional arguments, or via configuration records. For example, the following code would change font size, axis and line thickness, frame spacing, and so on.

```python
import matplotlib.pyplot as plt
plt.rc('lines', linewidth=2) # change line style, fonts
plt.rc('font', size=16)  # one group at a time
plt.rc('axes', linewidth=2)
plt.rc('figure.subplot', right=0.96)  # frame spacing
plt.rc('figure.subplot', top=0.96)
plt.rc('figure.subplot', left=0.1)
```

Many figures in the book are produced with these parameters. For convenience, the code can be included in a file (say `rcpara.py`) and imported before use.

We give a very brief introduction to the usage of NumPy arrays (also called ndarrays). There are many other powerful features that can be understood through use, examples (see programs mentioned at the end of this section), and the NumPy documentation.\(^\text{10}\) Remember, import NumPy before use, and online help is available in IPython or IDLE on any installed packages. To get help on `np.array` for instance, issue

```
In [1]: import numpy as np
In [2]: help(np.array)
```

### Array Creation, Indexing, and Slicing

A NumPy array can be created from any array-like objects:

```
In [3]: np.array([1., 2., 3.])
Out[3]: array([1., 2., 3.])
In [4]: a=np.arange(5)
    
a
Out[4]: array([0, 1, 2, 3, 4])
```

The NumPy `arange` function is just like the Python `range` except it returns an ndarray. The data type (e.g., float or integer) is derived from the assigned objects or specified with `dtype`. NumPy has other functions for initializing arrays, for example, `zeros(5)` and `ones(5)` will create two, 5-element arrays filled with zeros and ones, respectively, with float as the default type. The array `a` will be continuously updated and used in the following examples using IPython notebook mode.

\(^{10}\text{There are also online tutorials, for example, scipy-lectures.github.io.}\)
Array elements can be accessed by an index, either positive or negative,

```
In [5]: a[2]
Out[5]: 2
In [6]: a[-2]
Out[6]: 3
```

A positive index counts from the beginning of the array. Remember that array indices start from zero, thus `a[0]` is the first element (0), and `a[2]` the third element (2). A negative index is allowed, which counts backward from the end. So `a[-1]` refers to the last element (4), `a[-2]` the second last element (3), and so on.

Slicing is triggered by the colon `[:]` symbol, and is an important part of using NumPy arrays. It normally takes the form `a[start:end:step]`, producing an array from `start` up to but not including `end`, in steps `step`. If `start` or `end` are absent, the first or the last elements (inclusive), respectively, are assumed. If `step` is omitted, it defaults to 1. A negative `step` means going in the reverse direction of the array. Some basic slicing examples are the following:

```
In [7]: a[:]
Out[7]: array([0, 1, 2, 3, 4])
In [8]: a[1:3]
Out[8]: array([1, 2])
In [9]: a[:3]
Out[9]: array([0, 1, 2])
In [10]: a[2:]
Out[10]: array([2, 3, 4])
In [11]: a[::1]
Out[11]: array([4, 3, 2, 1, 0])
```

Note that `a[:]` returns the whole array, but `a[]` would be illegal.

**Array Manipulations**

Elements can be inserted or removed from ndarrays. The following example inserts a “9” in front of the original index “1” of `a` and deletes the element at index “2”:

```
In [12]: np.insert(a,1,9)
Out[12]: array([0, 9, 1, 2, 3, 4])
In [13]: np.delete(a,2)
Out[13]: array([0, 1, 3, 4])
```

Each time, a new array is created, and the original array `a` is not modified.

Operations on an array occur in an element-wise, vectorized fashion. For example:

```
In [14]: 2*a
Out[14]: array([0, 2, 4, 6, 8])
In [15]: a+3
Out[15]: array([3, 4, 5, 6, 7])
```

When the scalar “3” is added to the array, it is cast into an array of the same dimension (called broadcasting) internally, and then the arrays are added element by element. Because no explicit
looping is invoked in Python, the speed gain is considerable for larger arrays. For explicit looping, Python lists are faster.

Elements can also be selectively changed via slicing. The following assigns two elements to new values:

```python
In [16]: a[1:3]=[-1, -2]
    a
Out[16]: array([ 0, -1, -2,  3,  4])
```

Basic slicing like the preceding returns a “view” of the array, that is, it points to the elements within the same array. In contrast, slicing of a Python list returns a copy.

If we wish to work with a copy of an ndarray, we can use the `copy()` function. The following copies part of the array and changes the copy without affecting the original array:

```python
In [17]: b = np.copy(a[1:3])
    b
Out[17]: array([-1,  -2])
In [18]: b[:]=[-3, -4]
    b
Out[18]: array([-3,  -4])
In [19]: a
Out[19]: array([ 0, -1, -2,  3,  4])
```

Advanced Indexing

Copies of ndarrays can also be obtained with advanced indexing. This occurs when the index is not simply an integer. The following code obtains a copy with a list index, which triggers advanced indexing. Again, the source array is not affected:

```python
In [20]: c = a[[1,2]]
    c
Out[20]: array([-1,  -2])
In [21]: c[:]=[-5, -6]
    c
Out[21]: array([-5,  -6])
In [22]: a
Out[22]: array([ 0, -1, -2,  3,  4])
```

It is also possible to select elements with a boolean or truth array. A truth array can be created by logical relations such as `<`, `>`, `==`, and so on. For example,

```python
In [23]: a>0
Out[23]: array([False, False, False,  True,  True], dtype=bool)
In [24]: a==0
Out[24]: array([ True, False, False, False, False], dtype=bool)
```

The statement `a>0` tests every element of `a` against 0 and sets the corresponding indices to `True` if they are greater than zero, and `False` otherwise.
When a truth array is an index to another array, only the true elements are taken. This is sometimes called fancy indexing in NumPy. The following are some examples of truth array indexing:

```
In [25]: a[a>0]
Out[25]: array([3, 4])
In [26]: a[a<0]
Out[26]: array([-1, -2])
In [27]: a[a==0]
Out[27]: array([0])
```

The positive elements are selected in the first instance, negative ones in the second instance. Note that the third instance has only a single element, but it is still returned as a one-element array.

Like assignment with slicing, we can also selectively assign elements with a truth array:

```
In [28]: a[a<0]=0
   
   a
Out[28]: array([0, 0, 0, 3, 4])
```

All elements of the array less than zero are set to 0 in the preceding example.

Advanced indexing can be very powerful but also convoluted at the same time, making the code harder to understand. Unless necessary, we prefer the direct approach, for example, to obtain array copies with the `copy()` function.

### Multidimensional Arrays

Multidimensional arrays can be created and handled similarly. For instance, `zeros((2, 5))` will create a 2-row by 5-column array filled with zeros. Each dimension is called an axis, so axis 0 refers to going down the rows and axis 1 to going across the columns.

Next, we create a two-dimensional array and find its dimensions via the `shape` attribute:

```
In [29]: d=np.array([[11,12,13],[21,22,23],[31,32,33]])
   
d
Out[29]: array([[11, 12, 13],
                 [21, 22, 23],
                 [31, 32, 33]])
In [30]: d.shape
Out[30]: (3, 3)
```

The array can be accessed with two indices and sliced like before, but the results may be scalars, one-dimensional arrays (row or column vectors), or two-dimensional subarrays:

```
In [31]: d[1,2]
Out[31]: 23
In [32]: d[0]
Out[32]: array([11, 12, 13])
In [33]: d[:,1]
Out[33]: array([12, 22, 32])
In [34]: d[1:,2]
Out[34]: array([[21, 22],
                 [31, 32]])
```
In this example, \( d[1,2] \) refers to the element at the second row and third column, \( d[0] \) the first (entire) row, and \( d[:,1] \) the second column, both 1D arrays. But \( d[1:,2] \) refers to the second row and above, and all columns up to, but not including, the third column. The result is a 2D subarray. See Program 7.2 for actual uses.

Rows or columns can be inserted or deleted as well. The following deletes the second row along axis 0 and the third column along axis 1, respectively:

\[
\text{In [35]:} \quad \text{np.delete}(d,1,\text{axis}=0) \\
\text{Out[35]:} \quad \text{array}([[11, 12, 13], \\
\quad [31, 32, 33]]) \\
\text{In [36]:} \quad \text{np.delete}(d,2,\text{axis}=1) \\
\text{Out[36]:} \quad \text{array}([[11, 12], \\
\quad [21, 22], \\
\quad [31, 32]])
\]

Operations on ndarrays with advanced indexing can be very efficient and elegant. For instance, we can swap the first and third columns simply by:

\[
\text{In [37]:} \quad d[:,[0,2]]=d[:,[2,0]] \\
\text{d} \\
\text{Out[37]:} \quad \text{array}([[13, 12, 11], \\
\quad [23, 22, 21], \\
\quad [33, 32, 31]])
\]

The target \( d[:,[0,2]] \) refers to the first and third columns of all rows, and the source \( d[:,[2,0]] \) specifies the third and first columns of all rows. Slicing and advanced indexing work together to enable us to swap the columns in a single statement. See Program 9.9 for examples in actual codes.

**Universal Functions**

NumPy arrays of the same shape can be added, subtracted, multiplied, or divided, all element-wise. Python operators like \(+, -, \times, \div, \star\star\) and so on, are context-aware. When acting on ndarrays, operations are carried out element-wise transparently and automatically, irrespective of the type of operands as long as they can be broadcast into an ndarray.

\[
\text{In [38]:} \quad x=\text{np.array}([1.,2.,3.,4.]) \\
\quad x-2 \\
\text{Out[38]:} \quad \text{array}([-1., 0., 1., 2.]) \\
\text{In [39]:} \quad x/2. \\
\text{Out[39]:} \quad \text{array}([0.5, 1. , 1.5, 2. ]) \\
\text{In [40]:} \quad x\times x \\
\text{Out[40]:} \quad \text{array}([1. , 4. , 9. , 16.])
\]

In the first two cases, the second operand is broadcast to an ndarray \([2.,2.,2.,2.]\) before element-wise subtraction/division is carried out. In the third case, no broadcasting is necessary, and the elements are multiplied one to one, amounting to \(x\star\star2\).
In NumPy, the concept of element-wise operations is also extended to functions, called universal functions (ufunc). They operate on scalar and ndarray inputs transparently. For example, with the same $x$ as used previously,

```python
In [41]: np.sqrt(x)
Out[41]: array([1. , 1.41421356, 1.73205081, 2. ])
In [42]: np.maximum(x,x[::-1])
Out[42]: array([4. , 3. , 3. , 4. ])
```

In the first instance, the square root of every element is taken. Next, each element of $x$ is compared to the element of the reversed array, and the greater one is picked. Using ufuncs with ndarrays involves implicit looping. It is not only simple and elegant, but also efficient: the actual coding or computation is often done in compiled languages like CPython or C/C++ for increased speed.

Like `sqrt` and `maximum`, most NumPy functions are ufuncs. Thus, any function we define using NumPy versions of mathematical functions such as `sqrt` is also a universal function.\(^{11}\) See `potential()` in Program 4.6 for an example.

We have described very briefly some features of the NumPy library. There are many other operations and advanced features on NumPy arrays, including concatenation, stacking, sorting, and logical operations, and so on. Numerous examples are found and explained in the accompanying programs, particularly starting from Chapter 4 such as Programs 4.6, 6.8, 7.3, 8.2, 9.7, and S:6.1, to name but a few.

---

\(^{11}\)If necessary, we can also make a non-ufunc into a ufunc by vectorizing it with a NumPy module named, well, you guessed it: `vectorize()`.