Chapter 2

Programming

Mathematics provides the foundations of our models and of the algorithms we use to solve them. Computers are the engines that run these algorithms. Computers are also invaluable for simulation and visualization. Simulation and visualization build intuition, and intuition completes the loop by feeding into better mathematics. This chapter provides a brief introduction to scientific computing.

Companion code for this and other chapters is available in the accompanying Jupyter code book, written in Python (see page xiv). When the first edition of this book was published in 2009, the choice of Python was viewed as surprising. But Python now lies at the heart of a great many applications in engineering, machine learning, artificial intelligence and data science. It also features an outstanding just-in-time compiler and easy access to parallelized computation, which we discuss in more detail below.

At the same time, there are other excellent scientific computing environments and preferences across them vary widely. To avoid constraining the reader, in the current edition, almost all algorithms presented within in the hardcopy textbook have been shifted to pseudocode, rather than one specific language.

2.1 Algorithms

In this introductory section we discuss algorithmic foundations and work through simple examples.

2.1.1 Iteration and Flow Control

Many of the problems we study in this text reduce to a search for algorithms. The language we will use to describe algorithms is called *pseudocode*. Pseudocode is an informal way of presenting algorithms for the benefit of human readers, without getting tied down in the syntax of any particular programming language. It's a good habit to begin every program by sketching it first in pseudocode.

Our pseudocode rests on the following four constructs:

if–then–else, **while,** **repeat–until,** and **for**

The general syntax for the **if–then–else** construct is

```
if condition then
    first sequence of actions
else
    second sequence of actions
end
```

The condition is evaluated as either true or false. If found true, the first sequence of actions is executed. If false, the second is executed. Note that the **else** statement and alternative actions can be omitted: If the condition fails, then no actions are performed. A simple example of the **if–then–else** construct is

```
if there are cookies in the jar then
    eat them
else
    go to the shops and buy more
    eat them
end
```

The **while** construct is used to create a loop with a test condition at the beginning:

```
while condition do
    sequence of actions
end
```

The sequence of actions is performed only if the condition is true. Once they are completed, the condition is evaluated again. If it is still true, the actions in the loop are performed again. When the condition becomes false the loop terminates. Here's an example:

```
while there are cookies in the jar do
    eat one
end
```

The algorithm terminates when there are no cookies left. If the jar is empty to begin with, the action is never attempted.

The **repeat–until** construct is similar:

```
repeat
    sequence of actions
until condition
```

Here the sequence of actions is always performed once. Next the condition is checked. If it is true, the algorithm terminates. If not, the sequence is performed again, the condition is checked, and so on.

The **for** construct is sometimes called a definite loop because the number of repetitions is predetermined:

```
for element in sequence do
    do something
end
```

For example, the following algorithm computes the maximum of a function f over a finite set S using a **for** loop and prints it to the screen.[1]

```
set c = −∞
for x in S do
    set c = max{c, f(x)}
end
print c
```

In the **for** loop, x is set equal to the first element of S and the statement "set $c = \max\{c, f(x)\}$" is executed. Next x is set equal to the second element of S, the statement is executed again, and so on. The statement "set $c = \max\{c, f(x)\}$" should be understood to mean that $\max\{c, f(x)\}$ is first evaluated, and the resulting value is assigned to the variable c.

Exercise 2.1 Modify this algorithm so that it prints the maximizer rather than the maximum. Explain why it is more useful to know the maximizer.

Let's consider another example. Suppose that we have two arrays A and B stored

[1] That is, it displays the value to the user. The term "print" dates from the days when sending output to the programmer required generating hardcopy on a printing device.

in memory, and we wish to know whether the elements of A are a subset of the elements of B. Here's a prototype algorithm that will tell us whether this is the case:

```
set subset = True
for a in A do
    if a ∉ B then set subset = False
end
print subset
```

Next, as an example of a more sophisticated problem, suppose we wish to model flipping a biased coin with probability p of heads, and have access to a random number generator that yields uniformly distributed variates on $[0, 1]$. The next algorithm uses these random variables to generate and print the outcome (either "heads" or "tails") of ten flips of the coin, as well as the total number of heads.[2]

```
set H = 0
for i in 1 to 10 do
    draw U from the uniform distribution on [0, 1]
    if U < p then                                   // with probability p
        print "heads"
        H = H + 1
    else                                            // with probability 1 − p
        print "tails"
    end
end
print H
```

Note the use of indentation, which helps maintaining readability of our code.

Exercise 2.2 Consider a game that pays \$1 if and only if three consecutive heads occur within ten flips. Otherwise the game pays zero. Modify the previous algorithm to generate a round of the game and print the payoff. If you can, write a routine to run the game 10,000 times and record the outcome. Use this data to make an estimate of the probability that the game pays \$1.

Exercise 2.3 Let b be a vector of zeros and ones. The vector corresponds to the employment history of one individual, where 1 means employed at the associated point in time, and 0 means unemployed. Write an algorithm to compute the longest (consecutive) period of employment.

[2] What is the probability distribution of this total?

2.1.2 Application: Bisection

As a more extensive example, let's look at the *bisection algorithm*. You have probably implemented bisection as a child, when you played this game: First, player A thinks of a secret number n between 1 and 100. Player B must find n with the minimum number of guesses, receiving only "yes" or "no" replies. The right strategy is for player B to ask if $n \leq 50$. If the answer is yes, then B repeats the same logic on $1, \ldots 50$ by asking if n is less than 25. If no, then the logic is repeated in the other direction, by asking if n is less than 75, and so on.

Here's the same idea applied to approximating the root of a function $f\colon [\alpha, \beta] \to \mathbb{R}$, where $f(\alpha)$ and $f(\beta)$ have different signs (i.e., $f(\alpha)f(\beta) < 0$). We assume here for convenience that f is continuous and has exactly one root (i.e., one $x \in (\alpha, \beta)$ such that $f(x) = 0$). Other inputs to the algorithm are M, a maximal number of iterations (to bound runtime) and ϵ, a small number. If M is sufficiently large, then the algorithm finds and prints a value x such that $|f(x)| < \epsilon$.

```
set i = 1, a = α, and b = β
while i ≤ M do
    set c = (a + b)/2        // take the midpoint of the current interval
    if |f(c)| < ε then
        print "Approximate root of f is c"
        stop
    end
    set i = i + 1
    if f(c)f(a) < 0 then                    // signs differ so root is in (a, c)
        set b = c                // set (a, b) = (a, c) (choose lower subinterval)
    else                                    // root not in (a, c) so must be in (c, b)
        set a = c                // set (a, b) = (c, b) (choose upper subinterval)
    end
end
print "Exceeded maximum iteration value M, bisection failed."
```

The **stop** keyword indicates that execution of the algorithm should terminate whenever it is encountered. The key piece of logic is that $a < b$ and $f(a)f(b) < 0$ implies that the root lies in (a, b). The algorithm works by using this logic to iteratively select a subinterval of the domain which is half as long as the previous one and guaranteed to contain the root.

Exercise 2.4 Implement the bisection algorithm in your favorite programming language. Test it on the function $f(x) = \sin(4(x - 1/4)) + x + x^{20} - 1$, over the interval $[0, 1]$. The unique root of f on $[0, 1]$ is ≈ 0.408.

A solution in Python can be found in the code book (see page xiv).

2.2 Program Design

In this section we discuss how to combine algorithms into programs, considering both flexibility and efficiency.

2.2.1 User-Defined Functions

Algorithms solve programming problems. Programming involves more than just implementing algorithms, however. Another issue is design: How should we construct programs that contain multiple interacting algorithms while retaining clarity and readability as our projects grow?

The first step along the road to good program design is learning to break programs up using functions. Functions are the primary tool through which programmers implement the time-honored strategy of divide and conquer: problems are split into smaller subproblems, which are then coded up as functions. The main program then coordinates these functions, calling on them to do their jobs at the appropriate time.

Functions allow programmers to isolate and test individual algorithms or steps of logic. Functions also allow programmers to isolate variables to local scope: for most programming languages, changing the value of variables local to the function does not affect the value of global variables, even if they share the same name. Finally, due to their ability to isolate logic and scope, functions can be targeted for optimization by a compiler—more on this below.

Let's look at a particular algorithm and how to implement it as a function. The algorithm we consider is the (discrete) *inverse transform method* for generating random draws from a discrete probability distribution. More specifically, we take S to be a finite set and ϕ to be a function from S to $\mathbb{R}_+$ with the property $\sum_{x \in S} \phi(x) = 1$. Throughout, $\phi(x)$ represents the "probability assigned to x." Suppose we have the ability to generate random variables that are uniformly distributed on $(0, 1]$. We now want to generate random draws from S that are distributed according to ϕ.

Let W be uniformly distributed on $(0, 1]$, so that, for any $a \leq b \in (0, 1]$, we have $\mathbb{P}\{a < W \leq b\} = b - a$, which is the length of the interval $(a, b]$.[3] Our problem will be solved if we can implement a function $z \mapsto \tau(z)$ from $(0, 1]$ to S such that $\tau(W)$ has distribution ϕ. In other words,

$$\mathbb{P}\{\tau(W) = x\} = \phi(x) \quad \text{for all } x \in S$$

[3] The probability is the same whether inequalities are weak or strict.

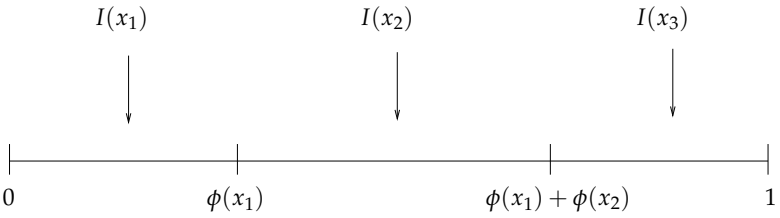

Figure 2.1 Partition $(I(x))_{x \in S}$ created by ϕ

One technique is as follows. First we divide the unit interval $(0, 1]$ into disjoint subintervals, one for each $x \in S$. The interval corresponding to x is denoted $I(x)$ and is chosen to have length $\phi(x)$. More specifically, when $S = \{x_1, \ldots, x_N\}$, we take

$$I(x_i) := (\phi(x_1) + \cdots + \phi(x_{i-1}),\ \phi(x_1) + \cdots + \phi(x_i)]$$

with $I(x_1) = (0, \phi(x_1)]$. You can easily confirm that the length of $I(x_i)$ is $\phi(x_i)$ for all i. Figure 2.1 gives the picture for $S = \{x_1, x_2, x_3\}$.

Now consider the function $z \mapsto \tau(z)$ defined by

$$\tau(z) := \sum_{x \in S} x \mathbb{1}\{z \in I(x)\} \qquad (z \in (0, 1]) \tag{2.1}$$

where $\mathbb{1}\{z \in I(x)\}$ is one when $z \in I(x)$ and zero otherwise.

Exercise 2.5 Prove: for all $x \in S$, we have $\tau(z) = x$ if and only if $z \in I(x)$.

The random variable $\tau(W)$ has distribution ϕ. To see this, pick any $x \in S$, and observe that the $\tau(W) = x$ precisely when $W \in I(x)$. The probability of this event is the length of the interval $I(x)$, which, by construction, is $\phi(x)$. Hence $\mathbb{P}\{\tau(W) = x\} = \phi(x)$ for all $x \in S$ as claimed.

A pseudocode implementation of the function $z \mapsto \tau(z)$ is given in algorithm 2.1.

A direct Python implementation is shown in listing 2.1. In the accompanying Jupyter code book we verify via simulation that the function performs as expected. Similar implementations can be produced in Julia and other scientific computing environments.

The algorithm displayed in listing 2.1 can be improved upon in practice. For example, instead of using `for` loop, we can apply a variation on the bisection algorithm from §2.1 to find the interval containing z more efficiently.

Algorithm 2.1: The function $z \mapsto \tau(z;\phi)$

Data: the set of points $S = \{x_1, \ldots, x_N\}$ and a distribution ϕ on S
Function $\tau(z)$
    set $a = 0$
    for i in $1, \ldots, N$ **do**
        set $b = a + \phi(x_i)$
        if $a < z \leq b$ **then return** x_i
        set $a = b$
    end

```
def tau(z, S, phi):
    """
    Evaluates tau(z) given array_like S and phi.
    """
    a = 0
    for i, x in enumerate(S):
        b = a + phi[i]
        if a < z <= b:
            return x
        a = b
```

Listing 2.1 Direct implementation of the function τ

```
input numpy as np
def tau(z, S, phi):
    i = np.searchsorted(np.cumsum(phi), z)
    return S[i]
```

Listing 2.2 Efficient implementation of the function τ

```
def tau_factory(S, phi):
    Phi = np.cumsum(phi)

    def tau(z):
        i = np.searchsorted(Phi, z)
        return S[i]

    return tau
```

Listing 2.3 Implementing τ via a closure (factory function)

One such implementation is given in listing 2.2. It applies NumPy's `searchsorted` function, which employs a form of bisection, to the task of locating the correct interval. All common scientific computing environments provide a function analogous to `searchsorted`.

Even though the function `tau` in listing 2.2 is efficient, there are still some improvements we can make. For example, it is quite likely that we will want to call `tau` at many different `z`, while holding `S` and `phi` fixed. In this case, it would be less cumbersome if we could pass `S` and `phi` once and then call `tau` with `z` alone.

In addition, the cumulative sum of `phi` is recomputed every time the function `tau` in listing 2.2 is called. This is clearly inefficient.

Listing 2.3 solves both these problems by employing what's known as a *closure*. An outer function called `tau_factory` is used to compute the cumulative sum of `phi` and create a function `tau` that acts on this sum and the array `S`. It then returns the function `tau`.

We can create and use the function `tau` using code such as this:

```
phi = 0.2, 0.5, 0.3
S = 0, 1, 2
tau = tau_factory(S, phi)
tau(0.1)                    # only need to supply the parameter z
```

Notice how `tau` retains access to the data `S` and `phi` even after the function `tau_factory` has finished executing. The outer function is sometimes called a *factory function*.[4]

Python, Julia, R and MATLAB all offer the ability to use closures.

2.2.2 Object Oriented Programming

Some programming languages, such as Python, support *object-oriented programming* (OOP), which essentially means that functions and the data they act on are bundled together into *abstract data types* (ADTs). MATLAB and Julia offer variations on this idea. I personally use Python's OOP facilities routinely, although I write only small, lightweight data types for organizing closely related data.

For those who are interested, we now cover several examples of OOP style, using Python. Readers who prefer procedural styles or other computing environments can safely skip the remainder of this section.

In Python, a *class* definition is a blueprint for such an ADT, describing what kind of data it stores, and what functions it possesses for acting on these data. An *object* is an *instance* of the ADT; an individual realization of the blueprint, typically with its own unique data. Functions defined within classes are referred to as *methods*.

To illustrate the key ideas, we will build a simple class to represent and manipulate polynomial functions. The data in this case are the coefficients $(a_0, \ldots, a_N)$, which define a unique polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_N x^N = \sum_{n=0}^{N} a_n x^n \qquad (x \in \mathbb{R})$$

To manipulate these data we will create two methods, one to evaluate the polynomial from its coefficients, returning the value $p(x)$ for any x, and another to differentiate the polynomial, replacing the original coefficients $(a_0, \ldots, a_N)$ with the coefficients of p'.

Consider, first, listing 2.4, which *sketches* a class definition in pseudo-Python. This is *not* real Python code—it is intended to give the feeling of how the class definition might look, while omitting some boilerplate. The name of the class is `Polynomial`, as specified after the keyword **class**. The class definition consists of three methods. Let's discuss them in the order they appear.

The first method is called `initialize()`, and represents a *constructor*, which is a special method most languages provide to build (construct an instance of) an object from a class definition. Constructor methods usually take as arguments the data needed to set up a specific instance, which in this case is the vector of coefficients

[4] The technique adopted in listing 2.3 is called a closure because the data available in the outer function is enclosed in the nested function.

Listing 2.4 (polyclass0.py) A polynomial class in pseudo-Python

```
class Polynomial:

    def initialize(coef):
        """Creates an instance p of the Polynomial class,
        where p(x) = coef[0] x^0 + ... + coef[N] x^N."""

    def evaluate(x):
        y = sum(a*x**i for i, a in enumerate(coef))
        return y

    def differentiate():
        new_coef = [i*a for i, a in enumerate(coef)]
        # Remove the first element, which is zero
        del new_coef[0]
        # And reset coefficients data to new values
        coef = new_coef
```

$(a_0, \ldots, a_N)$. The function should be passed a list or tuple, to which the identifier `coef` is then bound. Here `coef[i]` represents a_i.

The second method `evaluate()` evaluates $p(x)$ from x and the coefficients. The third method is `differentiate()`, which modifies the data of a `Polynomial` instance, rebinding `coef` from $(a_0, \ldots, a_N)$ to $(a_1, 2a_2, \ldots, Na_N)$. The modified instance represents p'.

Now that we have written up an outline of a class definition in pseudo-Python, let's rewrite it in proper Python syntax. The modified code is given in listing 2.5. Before working through the additional syntax, let's look at an example of how to use the class:

```
data = [2, 1, 3]
p = Polynomial(data)   # Creates instance of Polynomial class
p.evaluate(1)          # Returns 6
p.coef                 # Returns [2, 1, 3]
p.differentiate()      # Modifies coefficients of p
p.coef                 # Returns [1, 6]
p.evaluate(1)          # Returns 7
```

An instance `p` is created by a call of the form `p = Polynomial(data)`. Behind the scenes this generates a call to the constructor method, which realizes the instance as an

Listing 2.5 (`polyclass.py`) A polynomial class, correct syntax

```
class Polynomial:

    def __init__(self, coef):
        """Creates an instance p of the Polynomial class,
        where p(x) = coef[0] x^0 + ... + coef[N] x^N."""
        self.coef = coef

    def evaluate(self, x):
        y = sum(a*x**i for i, a in enumerate(self.coef))
        return y

    def differentiate(self):
        new_coef = [i*a for i, a in enumerate(self.coef)]
        # Remove the first element, which is zero
        del new_coef[0]
        # And reset coefficients data to new values
        self.coef = new_coef
```

object stored in memory, and binds the name `p` to this instance. As part of this process a namespace for the object is created, and the name `coef` is registered in that namespace and bound to the data `[2, 1, 3]`.[5] The attributes of `p` can be accessed using `p.attribute` notation, where the attributes are the methods (in this case `evaluate()` and `differentiate()`) and instance variables (in this case `coef`).

Let's now walk through the new syntax in listing 2.5. First, the constructor method is given its correct name, which is `__init__`. The double underscore notation reminds us that this is a special Python method—we will meet another example in a moment. Second, every method has `self` as its first argument, and attributes referred to within the class definition are also preceded by `self` (e.g., `self.coef`).

The idea with the `self` references is that *they stand in for the name of any instance that is subsequently created.* As one illustration of this, note that calling `p.evaluate(1)` is equivalent to calling

```
Polynomial.evaluate(p, 1)
```

This alternate syntax is more cumbersome and not generally used, but we can see how `p` does in fact replace `self`, passed in as the first argument to the `evaluate()` method.

[5] To view the contents of this namespace type `p.__dict__` at the prompt.

And if we imagine how the `evaluate()` method would look with `p` instead of `self`, our code starts to appear more natural:

```
def evaluate(p, x):
    y = sum(a*x**i for i, a in enumerate(p.coef))
    return y
```

Before finishing, let's briefly discuss another useful special method. One rather ungainly aspect of the `Polynomial` class is that for a given instance `p` corresponding to a polynomial p, the value $p(x)$ is obtained via the call `p.evaluate(x)`. It would be nicer—and closer to the mathematical notation—if we could replace this with the syntax `p(x)`. Actually this is easy: we simply replace the word `evaluate` in listing 2.5 with `__call__`. Objects of this class are now said to be *callable*, and `p(x)` is equivalent to `p.__call__(x)`.

Exercise 2.6 Drawing on listing 2.3 and the discussion concerning the inverse transform method in §2.2.1, write a class with instance data `S` and `phi` that provides two methods: a method to evaluate the function $\tau(z)$ for any given z, and a method to generate a draw from `S` according to the distribution represented by `phi`.

2.2.3 High Performance Computing

When it comes to programming, which languages are suitable for scientific work? Since the time it takes to complete a programming project is the sum of the time spent writing the code and the time that a machine spends running it, an ideal language would minimize both these terms.

Designing such a language is not an easy task. There is an inherent trade-off between human time and computer time, due to the fact that humans and computers "think" differently: Flexible, high level languages that cater well to the human brain are, in general, hard for machines to optimize (i.e., convert into efficient machine code). For example, if we reduce flexibility by insisting that a variable x can point only to floating point numbers, we inconvenience the programmer but free the computer to specialize associated machine code to floating point operations.

Using the flexibility/efficiency trade-off, we can divide languages into (a) robust, lower level languages such as Fortran and C/C++, which execute quickly but can be a chore when coding and debugging, and (b) the more nimble, interactive higher level languages, such as Python, MATLAB and R. By design, these languages are easy to write with and debug, but execution can be orders of magnitude slower. As a consequence, a paradigm for scientific computing developed where programmers write most of the code in a high level language and then call out to Fortran or C code when heavy lifting is required.

There are several problems associated with this traditional approach. First, there is always a nonzero quantity of boilerplate code required when gluing two languages together. Such boilerplate is tedious to maintain and makes the code base less accessible. Second, increasing the number of languages means increasing the number of compilers (or interpreters), which in turn increases complexity and drives up maintenance costs.

For these reasons, in recent years, scientists at the forefront of computational and numerical methods have shifted towards computing environments with high performance just-in-time (JIT) compilers. As suggested by the name, JIT compilers generate machine code on the fly, at runtime. Python (through the Numba library) and Julia offer state-of-the-art JIT compilers based on the LLVM architecture. These JIT compilers can convert well written Python or Julia into extremely efficient machine code—as efficient as the best implementations in C, C++ or Fortran.

Modern high quality JIT compilers also allow programmers to parallelize execution of JIT-compiled code across multiple threads or target execution on a GPU.

Since these technologies change fast, we refrain from studying any listings in this text. Instead, readers are invited the explore just-in-time compilers on their own.[6]

[6]One possible source of information is https://quantecon.org, which presents lectures in Python and Julia that heavily exploit their JIT compilers.