1

Programming overview

A programming language useful to this book must provide a minimal set of components that can be used to combine numbers, compare quantities and act on the result of that comparison, repeat operations until a condition is met, and contain functions that we can use to input data, and output results. Almost any language will suffice, but I have chosen to use `Mathematica`'s programming environment as the vehicle. The reasoning is that 1. The input/output functions of `Mathematica` are easy to use, and require little additional preparation[1] 2. We will be focused on the ideas, numerical and otherwise, associated with the methods we study, and I want to draw a clear distinction between those ideas and issues of implementation. This book is not meant to teach you everything you need to know about programming[2] – we will discuss only the bare essentials needed to implement the methods. Instead, we will focus on the physical motivation and tools of analysis for a variety of techniques. My hope is that the use of `Mathematica` allows us to discuss implementation in a homogeneous way, and our restriction to the basic programming structure of `Mathematica` (as opposed to the higher-level functionality) allows for easy porting to the language of your choice.

Here, we will review the basic operations, rendered in `Mathematica`, falling into the broad categories: arithmetic operations, comparisons, loops, and input–output routines. In addition, we must be able to use variable names that can be assigned values, and there is a scoping for these constructions in `Mathematica` similar to C (and many other languages). Functions, in the sense of C, exist in `Mathematica`, and we will use a particular (safe) form, although depending on the context, there are faster (and slower) ways to generate functions. We will bundle almost every set of computations into a function, and this is to mimic

[1] There are libraries to import audio and video, for example, in C, but the resulting internal representation can be difficult to work with. The details of compiling with those libraries correctly linked is also specific to the language and compiler, issues that I want to avoid.
[2] Although, I will employ good programming practice in the examples and accompanying chapter notebooks.

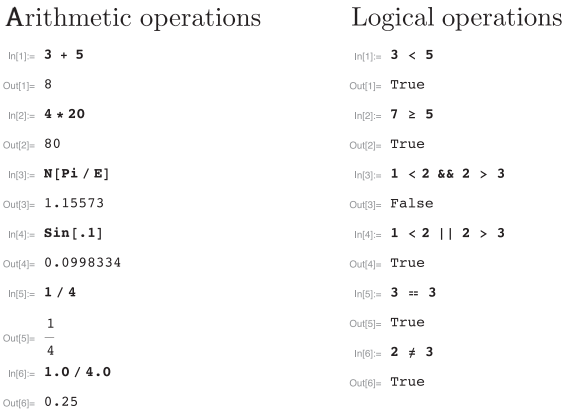

Figure 1.1 Examples of basic arithmetic input and output and logical operations.

good coding practice that is enforced in more traditional languages (for a reason – the logic and readability one gains by breaking calculations up into named constituents cannot be overvalued). Finally, we will look at two important ideas for algorithm development: recursion and function pointers. Both are supported in `Mathematica`, and these are also available in almost any useful programming language.

Beyond the brief programming overview, there are issues specific to numerical work, like consideration of timing, and numerical magnitude, that provide further introduction into the view of physics that we must take if we are to usefully employ computers to solve problems.

1.1 Arithmetic operations

All of the basic arithmetic operations are in `Mathematica`, and some are shown in Figure 1.1 – we can add and subtract, multiply, divide, even evaluate trigonometric functions (arguments in radians, always). The only occasional hiccup we will encounter is the distinction, made in `Mathematica`, between an exact quantity and a number – in Figure 1.1, we can see that when presented with `1/4`, `Mathematica` responds by leaving the ratio alone, since it is already reduced. What we are most interested in is the actual numerical value. In order to force `Mathematica` to provide real numbers, we can wrap expressions in the `N` function, as in `In[3]` on the left of Figure 1.1, or we can insert decimal points, as in `In[6]`. This is more than an aesthetic point – many simple calculations proceed very slowly if you suggest (accidentally or not) that all numbers are exact quantities (all fraction additions, for example, must be brought to a common, symbolic, denominator, etc.).

Aside from integers and real numbers, Mathematica is aware of most mathematical constants, like π and e, and when necessary, I'll tell you the name of any specific constant of interest.

1.2 Comparison operations

We will use most comparison operations, and Mathematica outputs True or False to any of the common ones – we can determine whether a number is greater than, less than, or equal to another number using >, <, == (notice that equality requires a double equals sign to distinguish it from assignment). We denote "less than or equal to" with <=, and similarly for "greater than or equal to" (>=). The logical "not" operation is denoted !, so that inequality is tested using != (not equal). Finally, we can string together the True/False output with "AND" (denoted &&) and "OR" (||). Some examples are shown on the right in Figure 1.1.

1.3 Variables

Unlike C or C++, variables in Mathematica can be instantiated by definition, and do not require explicit typedef-ing. So setting a variable is as easy as typing x = 5.0. The variable has this value (subject to scoping) until it is changed, or cleared (closest to delete that exists in Mathematica) by typing Unset[x].

Variables can take a number of forms: single elements, lists, matrices, etc. For us, variables will be purely numerical (no symbolic variable assignments are allowed – those are generally not available in other languages), and the numbers themselves will be "doubles," i.e. real numbers with maximum precision. We can then define tables and arrays of numbers, again by giving values to a variable name. In Figure 1.2, we see a few different ways of defining variables – first we define and set the variable p to have value 5 – Mathematica will print an output in general, and in the case of defining variables, it prints an output that reminds us of the variable's value. To suppress printing output, we use a semicolon at the end of a line – in the second example on the left in Figure 1.2, we define q to have value 7, and the semicolon tells Mathematica to just set the value without extra verbiage.

We can define variables that are tables of fixed length by specifying the numerical value for each entry, using {...}, as in the definition of x on the left in Figure 1.2. The Mathematica command

```
Table[f[k],{k,start,end,step}]
```

can also be used to generate tables that have values related to index number by the function f[k] – in the definition of the array variable y, we use f[k]=k for "iterator" (a dummy name given to the index used to generate the table) k.

Defining variables

```
In[1]:= p = 5
Out[1]= 5
In[2]:= q = 7;
In[3]:= x = {1.0, 2.0, 3.0, 4.0}
Out[3]= {1., 2., 3., 4.}
In[4]:= x
Out[4]= {1., 2., 3., 4.}
In[5]:= y = Table[k, {k, 1.0, 4.0, .5}]
Out[5]= {1., 1.5, 2., 2.5, 3., 3.5, 4.}
```

Setting variables

```
In[1]:= q = 9;
In[2]:= q
Out[2]= 9
In[3]:= q = 10;
In[4]:= q
Out[4]= 10
In[5]:= x = Table[j^2, {j, 1.0, 10.0, 2.0}]
Out[5]= {1., 9., 25., 49., 81.}
In[6]:= x[[2]]
Out[6]= 9.
In[7]:= x[[2]] = 4.0
Out[7]= 4.
In[8]:= x
Out[8]= {1., 4., 25., 49., 81.}
```

Figure 1.2 Examples of defining and setting variable values.

```
In[1]:= x = 2;
        y = 3;
In[3]:= x + y
Out[3]= 5
In[4]:= X = Table[Sin[j], {j, 0.0, Pi, Pi / 4}]
Out[4]= {0., 0.707107, 1., 0.707107, 1.22465 × 10^-16}
In[5]:= Y = Table[2.0 j, {j, 1, 5}]
Out[5]= {2., 4., 6., 8., 10.}
In[6]:= X[[2]] * Y[[3]]
Out[6]= 4.24264
In[7]:= X - Y
Out[7]= {-2., -3.29289, -5., -7.29289, -10.}
```

Figure 1.3 Using arithmetic operations with variables, table elements, and tables.

Once a variable has been defined by giving it a value, the value can be accessed (by typing the name of the variable as input) or changed (using the operator =) as shown on the right in Figure 1.2, where a table `x` is created, and its second entry set to the value `4.0`. The output of such an assignment is the assigned value, if we want to check the full content of `x`, we can type it as input, as in `In[8]`.

Variables can be used with the normal arithmetic operations, their value replaces the variable name internally, just as in most programming languages. In Figure 1.3, we define `x` and `y`, and add them. We can perform operations on elements of lists, or on the lists themselves (so the final example in Figure 1.3 adds each element

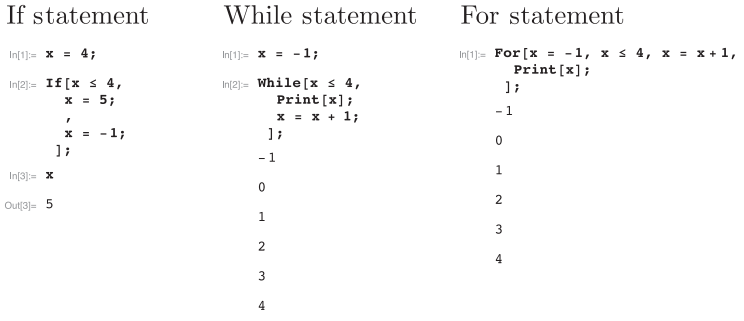

Figure 1.4 Using `Mathematica`'s `If`, `While`, and `For`.

of the lists `X` and `Y` – note that you cannot add together lists of different size). All variable and function names are case-sensitive, so that using `x` and `X` as variable names is unambiguous.

1.4 Control structures

The most important tools for us will be the `if-then-else`, `while` and `for` constructs. These can be used with logical operations to perform instructions based on certain variable values.

The `if-then-else` construction operates as you would expect – we perform instructions *if* a certain logical test returns `True`, and other instructions (*else*) if the test returns `False`. The `Mathematica` structure is:

```
If[test, op-if-test-true, op-if-test-false]
```

In Figure 1.4, we define and set the value of `x` to 4. Then we use the `If` statement to check the value of `x` – *if* `x` is less than or equal to 4, *then* we set `x` to 5, *else* we set `x` to -1.

Using `While` is similar in form – we perform instructions *while* a specified test yields `True`, and stop when the logical test returns `False`. The `Mathematica` command that carries out the `While` loop is

```
While[test, op-if-test-true]
```

An example in which we set `x` to -1 and then add one to `x` if its value is less than or equal to four is shown in Figure 1.4. In this example, we also encounter the i/o function `Print[x]`, which prints the value of the variable `x`.

Finally, "for loops" perform instructions repeatedly while an *iterator* counts from a specified start value to a specified end value – more generally, the iterator is given some initial value, and a logical test is performed on a function of the iterator – while the logical test is true, operations are executed. We can construct a for loop

from a while loop, so the two are, in a sense complementary. In Mathematica, the syntax is:

```
For[j = initialval, f[j], j-update, operations]
```

where j is the iterator, f[j] represents a logical test on some provided function of j, j-update is a rule for incrementing j, and operations is the set of instructions to perform while f[j] returns True – each execution of operations increments j according to j-update. This is easier done than said – an example of using the for loop is shown in Figure 1.4. That example produces the same results as the code in the While example.

1.5 Functions

Writing programs requires the ability to break computational instructions into logically isolated blocks – this aids in reading, and debugging. These isolated blocks are called "functions," generically, a name for anything that takes in input and returns output. Mathematica provides a few different ways to define programming functions. We will use the Module form of function definition – the basic structure is:

```
functionname[input1_, input2_]:= Module[{local variables},
                                   operations;
                                   Return[value];
                                 ]
```

An example of Module in action is shown in Figure 1.5 – but the important thing to remember is that we now have a function that can be called with some inputs, returns some output, and has hidden local variables that are not accessible to the "outside world."

In Figure 1.5, we define the function HelloWorld, that takes a single argument called name – the underscore identifies name as an input. The Module is set up with two local variables, one takes the value of name (generally, a string), and the other is set to one. The function itself prints a friendly greeting, and returns the value stored in localvarx (i.e. one). As a check that the variable localvarx really is undefined as far as the rest of the Mathematica "session" is concerned, the last line in Figure 1.5 calls localvarx – the fact that Mathematica returns the variable name, unevaluated, indicates that it is not currently defined.

We can use all of our arithmetic, logical, and control operations inside the function to make it do more interesting things. As an example, the two functions defined in Figure 1.6 are used to sort an array of numbers in increasing order. The first function is Swap – this takes a list, and two numbers, a and b, as inputs, swaps

```
In[1]:= HelloWorld[name_] := Module[{retval, localvarx},
          retval = 1.0;
          localvarx = name;
          Print["Hello ", localvarx];
          Return[retval];
         ]
In[2]:= X = HelloWorld["Dave"]

        Hello Dave

Out[2]= 1.

In[3]:= Y = HelloWorld[33];

        Hello 33

In[4]:= Y

Out[4]= 1.

In[5]:= localvarx

Out[5]= localvarx
```

Figure 1.5 Example of defining, and then calling, a function in `Mathematica` using `Module`.

```
In[1]:= Swap[inlist_, a_, b_] := Module[{holder, outlist},
          outlist = inlist;
          holder = outlist[[b]];
          outlist[[b]] = outlist[[a]];
          outlist[[a]] = holder;
          Return[outlist];
         ]
In[2]:= InsertionSort[inlist_] := Module[{outlist, indexx, indexy, curelm, Nlist},
          outlist = inlist;
          Nlist = Length[outlist];
          For[indexx = 2, indexx ≤ Nlist, indexx = indexx + 1,
           curelm = outlist[[indexx]];
           indexy = indexx - 1;
           While[indexy > 0 && outlist[[indexy]] > curelm,
            outlist = Swap[outlist, indexy, indexy + 1];
            indexy = indexy - 1;
           ];
           outlist[[indexy + 1]] = curelm;
          ];
          Return[outlist]
         ]
In[3]:= InsertionSort[{5, 2, 4, 6, 1, 3}]

Out[3]= {1, 2, 3, 4, 5, 6}

In[4]:= InsertionSort[{-4, 1, 7, 2, 3, -10}]

Out[4]= {-10, -4, 1, 2, 3, 7}
```

Figure 1.6 Definition of the function `InsertionSort` – this function takes a list and sorts the elements of the list in increasing order (from [12]).

the value of the ath and bth elements of the list, and returns the resulting array. For the `InsertionSort` function, we go through the input array, and sequentially generate a sorted list of size `indexx-1`, increasing `indexx` until it is the size of the entire array. This is an inefficient but straightforward way to sort lists of numbers.

1.6 Input and output

There are a wide variety of `Mathematica` functions that handle various input and output. We will introduce specific ones as we go, I just want to mention two at the start that are of interest to us. The first we have already seen: `Print[ stuff ]` prints whatever you want, and can be used within a function to tell us what is going on inside the function.

The second output command we will make heavy use of is `ListPlot`. This function takes a table and generates a plot with the table values as heights at locations given by the index. Alternatively, if the table consists of pairs of values, then the plot uses the first of each pair as the x location, and the second provides the y (height). `ListPlot` can be used to visualize arrays of data, or function values. A few examples are shown in Figure 1.7.

1.7 Recursion

Most programming languages support a notion of "recursion" – this is the idea that a function can call itself. Recursion can be useful when designing "divide-and-conquer" algorithms. As a simple example of a recursive function, consider `DivideByTwo` defined in Figure 1.8. This function takes a number, and, if it is possible to divide the number by two, calls itself with the input divided by two. If the number cannot be divided by two, the function returns the non-dividable-by-two input. Notice the helper function `IsDivisableByTwo` – this checks divisibility using `Mathematica`'s built-in `Round` command.

An example of the function in action is shown in Figure 1.8 – we are using the `Print` command to see what value the function `DivideByTwo` gets at each call – you can see that it is called four times for the input 88, and returns a concrete result when its input is not divisable by two.

As a more interesting example, we can accomplish the same sorting of numbers idea from `InsertionSort` using recursion. See if you can "sort out" (no pun intended) the recursion in the definition of `MergeSort` shown in Figure 1.9. The helper function `Merge` takes two lists that are already sorted, and combines them to form a sorted list.

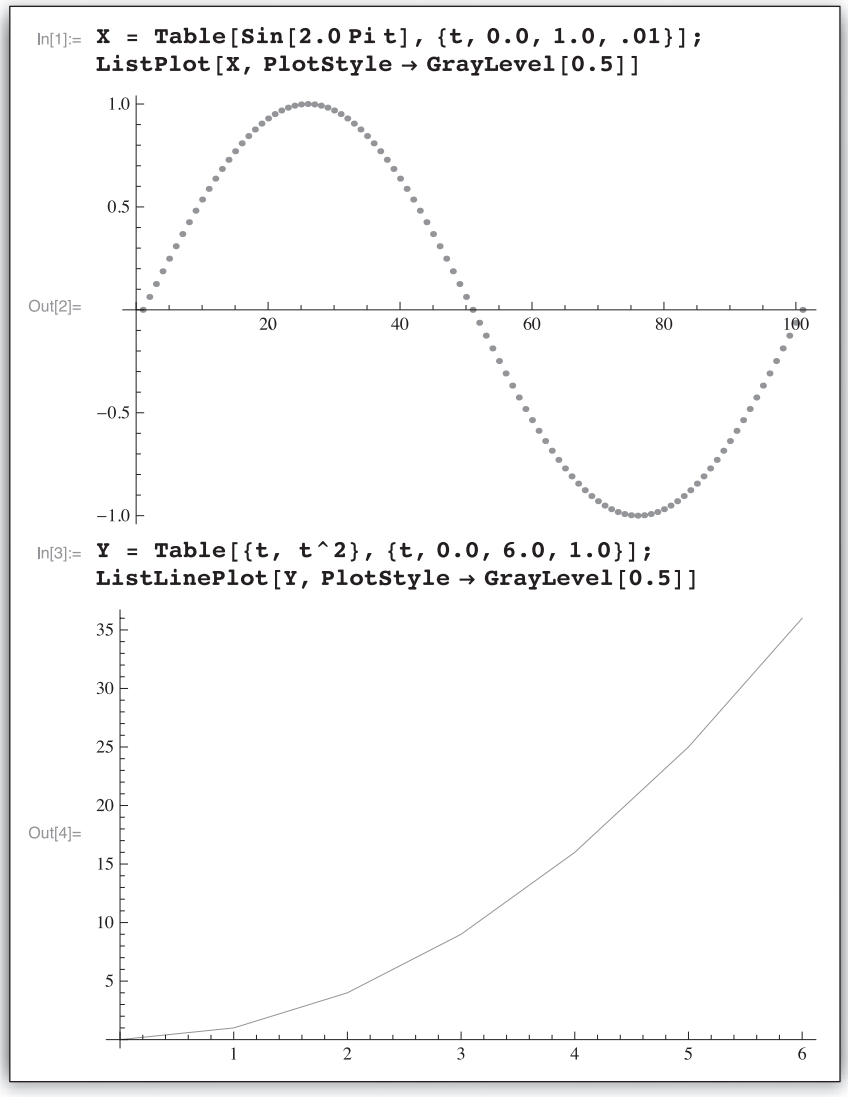

Figure 1.7 Plotting – if the table input to `ListPlot` contains single entries, like the first example above, then the x-axis is the entry number. If, as in the second case, the table contains pairs of numbers, then the first number is taken to be the x value, the second number the y value. The command `ListLinePlot` is identical to `ListPlot` except that the points are connected by lines.

```
In[1]:= IsDivisableByTwo[inx_] := Module[{retval, div, NZERO},
          NZERO = 10^(-10);
          div = inx / 2;
          If[Abs[div - Round[div]] > NZERO,
           retval = False;
           ,
           retval = True;
          ];
          Return[retval];
         ]
In[2]:= DivideByTwo[num_] := Module[{retval},
          Print[num];
          retval = num;
          If[IsDivisableByTwo[retval] == True,
           retval = DivideByTwo[num / 2];
          ];
          Return[retval];
         ]
In[3]:= DivideByTwo[88]

        88

        44

        22

        11

Out[3]= 11
```

Figure 1.8 Example of recursive function definition – you must provide both the recursive outcome (call the function again with modified input) and the final outcome (the definition of the endpoint).

1.8 Function pointers

It is important that functions be able to call other functions – we can accomplish this in a few different ways. One way to make a function you write accessible to other functions is to define it globally, and then call it. That is the preferred method if you have a "helper" function that is not meant to be called by "users." The function `Swap` in the insertion sort example from Figure 1.6 is such a support function – it is not meant to be called by a user of the function `InsertionSort`, it is purely a matter of convenience for us, the programmer.

But sometimes, the user must specify a set of functions for use by a program. In this case, we don't know or care what the names of the functions supplied by the user are – they are user-specified, and hence should be part of the argument of any function we write. This "variable" function is known, in C, as a "function pointer" – a user-specifiable routine. Because of the low-key type-checking in `Mathematica`, we can pass functions as arguments to another function in the same way we pass anything. It is up to us to tell the user what constraints their function must satisfy. As an example, suppose we write a function that computes the time average of some user-specified function $f(t)$ – that is, we want to write a