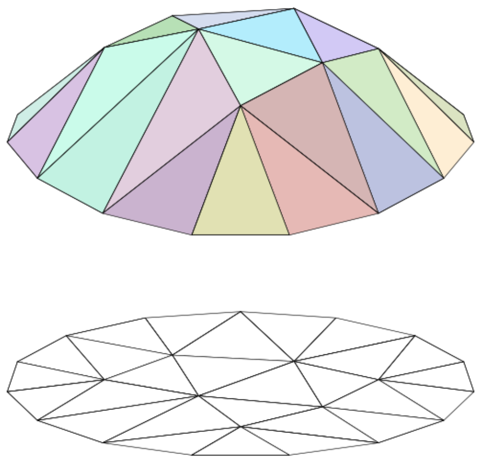
An Introduction to



Mathematica Programming

for Undergraduate Research

Robert D. French

Dr. Samuel Jator

February 5, 2013

Contents

Preface

This book is written especially for the students of MATH 3120 and 3130 at Austin Peay State University in Clarksville, TN. It is intended to introduce the reader, whether versed in programming or not, to the basic elements of *Mathematica*.

Computer algebra systems and symbolic programming provide a valuable means of exploring research topics at the undergraduate level. In addition, *Mathematica*'s facilities for functional programming provide a means for students to learn solid programming habits.

This book features an introduction to basic usage of *Mathematica* for those who have never used it before, and for those who have never done any programming before. It is our aim that this preliminary section "If you have never used *Mathematica* . . ." will provide enough background to get started with the remainder of the book.

We hope that this book will serve to clarify and make *Mathematica* a more accessible programming language. We hope this book illuminates the joy of programming, and that those who read this book will make exciting discoveries because of it.

Happy Programming, and Best Wishes!

Samuel N. Jator & Robert D. French
Austin Peay State University
August 2, 2024

If you have never used *Mathematica* ...

This pamphlet assumes you have done at least a little programming before, possibly in C, C++, FORTRAN, Visual Basic, or some other language. If not, *that is completely fine*. We will walk you through some simple programming concepts here so that you can get your feet wet, and then you will be ready to take on chapter 1.

If you have done some programming before, but have never used *Mathematica*, it will still benefit you to spend some time with this section, because it will walk you through some tricks to make using *Mathematica* more pleasant.

1. Notebooks and Cells

When you start *Mathematica* for the first time, it will create a new *notebook* for you. A notebook is just a type of document, like a Microsoft Word document, but it is specially designed to make *Mathematica* programming easier.[1] Notebooks are broken into cells, which you can think of like paragraphs. Each cell can have code in it, and they can be evaluated[2] independently.

You can create new cells by clicking in a blank region of the docoument. You will see the cursor displayed sideways, and that is *Mathematica*'s way of telling you that the cursor is in between cells or that it is not inside a cell. As soon as you start typing anything, a new cell will be created, your cursor will hop inside, and anything you type will be displayed there.

Cells can have different "Styles" depending on what you want to use them for. By default, when a new cell is created it will be an "Input" cell, which means you can type code in it and evaluate it. By right-clicking on the vertical bar on the right side of a cell, you can bring up a menu that will allow you to change the style of the cell. Changing your cell style to "Title", "Section", "Subsection", etc. as appropriate can help you organize your work and will make your presentations look more professional.

When you type some code in a cell, you can evaluate it by pressing "Shift+Enter"[3]. In this book, the result of evaluations will be displayed next to a $\hookrightarrow$ symbol. For example:

[1]Note that for most programming languages, you type your code in a *plain text* document, or one that has no special formatting in it. This is not the case in *Mathematica*; if you open a notebook in Vim or Notepad, you will see that your code is surrounded by lots of formatting directives that you probably don't want to type by hand!

[2]To "evaluate" some code simply means to run it.

[3]In case you are not familiar with this convention, it means to hold the "Shift" key and then press the "Enter" button

```
(* Cell 1.1 *)
5 + 7
↪ 12
```

In *Mathematica*, the results of your computation will be displayed in an *output cell* that is attached to the input cell you just evaluated, so it is important to note that we present things in a slightly more compact fashion. Also, you will notice that the above cell is labelled Cell 1.1 because it is the first cell in Section 1. Likewise, the next cell will be labelled Cell 1.2:

```
(* Cell 1.2 *)
cellnumber = 2;
''This is the '' <> ToString[cellnumber] <> ''nd cell!''
↪ ''This is the 2nd cell!''
```

Please feel encouraged to follow along by typing the code you see here into a *Mathematica* notebook and evaluating them. Much like mathematics, you cannot learn any programming language simply by reading a book – you must also solve problems and tinker with examples.

2. Fancy Typing and the Palette

When you start *Mathematica*, it should load a set of buttons in a thin vertical window to the right of your notebook. If it does not, you can access this palette by going to "Window" → "Palettes" → "Basic Math Input" in the menu bar. The palette gives you ways to make your code look more mathematical. For example, there are buttons for writing integrals, summations, exponents, matrices, greek letters, etc[4]. If you press any of these buttons, it will insert the necessary symbols into the current cell and you will see several small boxes in the positions where one would expect symbols. Pressing "Tab" will allow you to move between these boxes, and type any valid *Mathematica* expression in them.

2.1. Subscripts, Exponents, and Fractions. There are also keyboard shortcuts for many of the symbols in the palette. For example, to type a fraction, you can press "Ctrl+/" and your cursor will automatically be placed in the numerator. To type the fraction $\frac{2}{3}$ in *Mathematica*, you simply need to press "Ctrl+/" and then "2", "Tab" (which moves you to the denominator), and then "3".

Likewise, to type x^3, you simply type "x" and then "Ctrl+6" and then *Mathematica* will give you a black box above and to the right of the "x" character and move your cursor there. Then type "3", and you will have x^3.

You can of course combine elements in expression boxes. For example, to write $x^{\frac{2}{3}}$, you can type "x" and then "Ctrl+6" to get an expression box in the exponent position, then type "Ctrl+/" to turn that box into a fraction, then hit "2", "Tab", and then "3".

2.2. Greek Symbols (Things like π, θ, and α). Using the "Esc" button, you can generate these symbols. "Esc+pi+Esc" will get you the π symbol. You can look at Table 1 to see a longer list of symbols. These symbols are also available in the "Basic Math Input" palette.

[4]Indeed, many of the basic features of LaTeXare available in *Mathematica* to help you typeset mathematical expressions, but you will still need the full power of LaTeXin order to prepare your research for publication.

Symbol	*Mathematica* code
π	“Esc” pi “Esc”
α	“Esc” alpha “Esc”
β	“Esc” beta “Esc”
θ	“Esc” theta “Esc”
ϕ	“Esc” phi “Esc”
η	“Esc” eta “Esc”
χ	“Esc” chi “Esc”

TABLE 1. A table of greek letters

2.3. Sums, Integrals, and Derivatives. These are best accessed by using the corresponding buttons in the “Basic Math Input” palette. For example, clicking on the large capital Sigma will give you a summation sign with two input boxes below it and one above. You can type expressions in these boxes and press “Tab” to go to the next. You can also construct integrals and derivatives in this fashion. Here are some things to try:

(1) Find the sum $\sum_{n=1}^{10} 2^n$
(2) Make *Mathematica* show you that $\int_0^\pi \sin(x)dx = 1$
(3) Find the derivative of $\sin(x)$

3. Functions

Mathematica has tons of built-in functions. It has trig functions like `Sin` and `Cos`. It also has functions for plotting graphs and generating lists or tables of data (see “**??**” and “Lists, `Map`, and `Table`”). In addition, you can create your own functions, but that will be discussed in Chapter 3, “Defining Your Own Functions”. For now, it is important to go over a few basics about how functions work in *Mathematica*.

For example, we know from trig that $\sin(\frac{\pi}{2}) = 1$. To see this for yourself in *Mathematica*, type the following into a cell:

```
(* Cell 3.1 *)
Sin[π/2]
```

and press “Shift+Enter” to evaluate the cell. Below, you will see an output cell generated containing the number 1. However, there are other functions in *Mathematica* aside from normal math functions. For example, you can use the `Map` function to produce a list of sine values for a given set of domain points.

To see what we are talking about more clearly, let’s consider the set $\{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}, 2\pi\}$. We can calculate the corresponding range values very simply by applying `Sin` to these values via `Map`.

```
(* Cell 3.2 *)
Map[Sin, {0, π/2, π, 3π/2, 2π}]
↪ {0, 1, 0, −1, 0}
```

where the $\hookrightarrow$ on the last line indicates the return value from Cell 3.2.

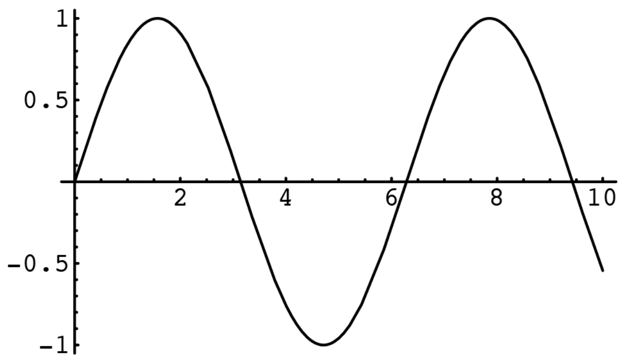


FIGURE 1. `Sin[x]` on $[0, 10]$

4. Variables

Mathematica's variables are very simple. Variables can hold numbers, lists, matrices, functions, symbols, or graphs. For now, we will just look at how to store some numbers as variables and see how it affects a plot.

```
(* Cell 4.1 *)
Amplitude = 1;
UpperBound = 10;
Plot[Amplitude * Sin[x], {x, 0, UpperBound}]
```

When you execute this, you will see that $\sin(x)$ will be drawn on the interval $[0, 10]$, as depicted in Figure 1. Try doing the following by changing the values for `Amplitude` and `UpperBound` in Cell 4.1:

(1) Plot $2\sin(x)$ on $(0, 10)$
(2) Plot $-\sin(x)$ on $(0, 10)$
(3) Plot $\pi\sin(x)$ on $(0, 2\pi)$

CHAPTER 1

A Review of Some Stuff You've Probably Seen Before

Before beginning this chapter, it is assumed that you know a few things about programming already. Specifically, you will need to know how to

(1) assign values to variables
(2) evaluate *Mathematica* cells

and if you don't, that's quite alright. Just see the section "*If you have never used* Mathematica ..." on page iii, and even if you have done a bit of programming before, we will walk you through some basic *Mathematica*.

1. Variable Assignment

The first thing we need to know in any language is how to assign variables. Probably you are thinking "*I learned this in CSCI 1010!*", but *Mathematica* is a subtle language, and does not always work as you might expect if you are coming from C++ or FORTRAN.

```
(* Cell 1.1 *)
i = 1;
j = 2
k := 3;
l := 4
```

Looking at this example, we can see that there are four slightly different ways to "assign" values to a variable, so let's discuss this a bit. If you put this code into a *Mathematica* cell, you will see that, upon evaluating the cell, it will output `2`.

So, if we assigned four numbers to four variables, why do we only see one output? The are two key items here. One is the semicolon (;) at the end of the first and third lines. This tells *Mathematica* to suppress the output of that calculation. Generally, it is appropriate to put a semicolon at the end of every line of code in a cell except for the last one. This is because you *usually* want to group your code into cells in such a way that each cell achieves one result, computes one item (or related set of items), or builds one data structure. When debugging, it can sometimes be handy to remove individual semicolons in order to investigate whether each line behaves as you expect.

The second key item is the `:=` operator. This is called the *Set Delayed* operator. This is different from the `=` operator[1] in one important way: it does not assign the value to that variable immediately. Rather, it tells *Mathematica* to wait until `k` or `j` is used and then evaluate the right hand side of that expression.

[1]also called the *Set* operator

What that means, in terms of the above example, is that, at the moment of evaluation, `i` and `j` are numeric variables that contain the values 1 and 2, but `k` and `l` are just symbols that do not yet contain any value. This might be hard to see with the previous example, so let's look at one that's slightly more involved:

```
(* Cell 1.2 *)
i = 1;
j = i + 10
k := i + 10
```

When we evaluate this code, we see that the output is simply `11`[2]. Now we set up a short experiment: We create a few separate cells, and examine what happens to `j` and `k` when we change the value of `i`.

```
(* Cell 1.3 *)
{j,k}
```

```
(* Cell 1.4 *)
i = 2;
{j,k}
```

```
(* Cell 1.5 *)
i = 20;
{j,k}
```

```
(* Cell 1.6 *)
i = x;
{j,k}
```

When we evaluate these cells, we note that they each produce an ordered pair (also called a *list*) as output. This is just to help us see what happens to `i` and `j` when we change the value of `i`.

In Cell 1.3, nothing interesting happens. We note that `i` and `j` are both `11` like we expected them to be. But now let's evaluate Cell 1.4 and see what happens. The output from Cell 1.4 tells us that `j` is still `11`, but that `k` has been updated to reflect the new value of `i`. This is because the definition we gave for `k` in Cell 1.2 is re-evaluated *every time* we use `k` in an expression.

If we evaluate Cell 1.5, we see the same thing has happened: `j` is still `11`, but `k` is now `30`, reflecting the fact that we changed the value of `i` again.

When you evaluate Cell 1.6, you will see that is does something slightly different. The explaination is simple, but we leave it as an exercise so that you are forced to think about it!

Delayed Evaluation is most frequently used when defining functions (See Defining Your Own Functions). For now, this is as far as we need to go with this topic. You might be thinking: "*Then why did we even bother?!*" but, the misuse of this operator is the cause of many bugs, and much of the *Mathematica* code you are likely to find on the internet contains a wild `:=` when it shouldn't. Straightening this out now will spare you headaches, and will promote friendship between you and your code.

[2]Using what we discussed earlier, can you figure out why this is? See problem 1

2. Invoking Functions

In *Mathematica*, the *invocation operator* is `[]`. This means that, given a function `F`, you can invoke it on an argument `x` by writing `F[x]`. If you do not use the invocation operator, your function will be treated like a variable. Let us look at how this works in a few different scenarios:

```
(* Cell 2.1 *)
Cos[0]
↪ 1
```

This is an example of invoking the `Cos` function directly for a single argument. If we want to evaluate `Cos` for a list of arguments, we can instead treat it as a variable and pass it as an argument to the function `Map`:

```
(* Cell 2.2 *)
Map[Cos, {0, 0.1, 0.2, 0.3, 0.4, 0.5}]
```

Evaluating this cell will give you all of the values for `Cos` for the domain points $\{0, 0.1, \ldots, 0.5\}$. The difference here is that we did not invoke the `Cos` function directly, but rather we passed it as a variable to the `Map` function, which applied `Cos` in turn to each of the data points in our list. Some functions are *listable*, which means that when passed a list, they automatically act on each element of the list separately. For example, all of the built-in trig functions are listable, so instead of using `Map` as we have above, we could instead write:

```
(* Cell 2.3 *)
Cos[{0, 0.1, 0.2, 0.3, 0.4, 0.5}]
```

Taking advantage of listable functions allows you to write code that is more elegant and compact[3]. Many other arithmetic operations are also listable, like addition, multiplication, and exponents.

We can put the famous trig identity $\sin^2(\theta) + \cos^2(\theta) = 1$ to the test with the following code[4]:

```
(* Cell 2.4 *)
thetaValues = {0, π/4, π/2};
Sin[thetaValues]^2 + Cos[thetaValues]^2
↪ {1,1,1}
```

Of course, not all functions are listable, and sometimes using this feature can make your code harder to read and thus debug. That is why `Map` is a trustworthy alternative.

Exercises.

(1) Explain why the code in Cell 1.2 produces only the single output of `11`
(2) Explain why Cell 1.6 produces `{11,x + 10}` as an output.
(3) Explain, using as much detail as you can, what is happening in Cell 2.4.
(4) Think about scalar multiplication and vector addition from Linear Algebra. Do you think these operations could be expressed as listable functions in *Mathematica*? Why or why not?

[3]As with anything, it is possible to go overboard with this technique and write code that is utterly incomprehensible. Use it when it makes sense, and avoid it when you think it might be confusing. Code that is easy to read will win you the respect of peers and professors alike.

[4]It is worth your while to play around with this code until it "clicks" for you.

CHAPTER 2

The Kernel, Variable State, and Scope

1. What is the Kernel?

In *Mathematica*, all of your coding is done in a *notebook*, and all of the output of your code is displayed there as well. However, the calculations themselves are done in *an entirely separate program*[1], and this program is called a *kernel*. There are many reasons for doing these calculations in a separate program from your notebooks:

(1) You can still edit your notebooks while long computations are running
(2) You can share variables and data between notebooks
(3) You can manage multiple kernels, (and thus multiple long-running computations) from a single notebook
(4) You can run computations on multiple kernels *on other computers*

So we see that this separation of kernel and notebook is very powerful. But what does it mean in terms of your research? Specifically, while you are working on your code, the values you calculate and the variables you assign them to will be stored in your "Local Kernel". This assignment of values to variables is called "State", and it's just a fancy computer science term for "The values of your variables at a given time".

Most of the time, these are just technical points that can be ignored, but understanding how the kernel works will make all the difference in the world when you begin to debug your research program.

1.1. Quitting the Kernel. This is kindof like an emergency reset for your program. Quitting the Kernel will basically erase the values for all the variables in your notebook (because they are stored in this separate program which you are about to quit). For example, open a new notebook and evaluate the following code:

```
(* Cell 1.1 *)
NumEggs = 5
```

Now, in a new cell, evaluate this code:

```
(* Cell 1.2 *)
Print[‘‘There Are ’’ <> ToString[NumEggs] <> ‘‘ eggs in
a Programmer’s Dozen’’];
```

And *Mathematica* will display the text "There are 5 eggs in a Programmer's Dozen"[2]. Now go up to the menu and select "Kernel" → "Quit Kernel" → "Local". This will cause the kernel you are using to go away, and your variable state will go

[1]This is an example of a Service in a Service Oriented Architecture, and if you are interested in software engineering, you should check this out.

[2]As opposed to 13 in a "Baker's Dozen".

away with it. Now re-evaluate Cell 1.2 and see what you get. It will tell you that there are "NumEggs" in a Programmer's Dozen. Wait. . . What?

What's happening here is that when *Mathematica* tries to find a value for the variable `NumEggs`, it notices that one does not exist, so in order to keep your code from exploding[3] a value is fabricated for you. The value given is a *symbol object*, which is just a placeholder, kinda like a string, which corresponds to the name "NumEggs".

Symbols will be discussed in more detail in Chapter **??**, but for now they aren't important. The thing to focus on here is that the variable `NumEggs` no longer contains the value `5` because you quit the kernel in which it was stored.

1.2. Debugging with a Secondary Notebook. When dealing with large *Mathematica* projects, it can sometimes be handy to play around with a particular line of code, or analyze the output from one section before going on to the next cell. You could do this all in one notebook, but that could get sloppy, and you run the risk of messing up some code that is already working[4].

For example, let's suppose you have the following block of code that does not appear to be working correctly:

```
(* Cell 1.3 *)
EvaluateDerivativeAtAPoint[f_,x_] := Block[{fPrime}[5],
fPrime = f';
fPrime[x]
];
(* Cell 1.4 *)
EvaluateDerivativeAtAPoint[Sin, π]
↪ -1
(* Cell 1.5 *)
EvaluateDerivativeAtAPoint[x^2 + x, π]
↪ (x + x^2)'[π]
```

Why does it give the right answer for sin but some weird expression for x^2+x? Well, let's open a new notebook and play around with a couple of things. You can open a new notebook by going to "File"→"New" or pressing "Ctrl+N". In the new notebook, type the following code:

```
(* Cell 1.6 *)
(* Second Notebook *)
f = x^2 + x;
fPrime = f';
fPrime[π]
↪ (x + x^2)'[π]
```

The idea here was to repeat the basic logic of our `EvaluateDerivativeAtAPoint` function to see if we could reproduce the problem, and indeed we have. It looks like maybe the third line is not invoking the function? Or maybe the second line is

[3]Which is what would happen in other dynamic languages like Ruby or PHP.

[4]Although, you are protecting yourself from code loss by using version control, right? If not, check out Appendix **??**, "**??**".

[5]Using `Block` is just a way to tell *Mathematica* that `fPrime` should be treated as a *local* variable, which means its value will not be seen outside of the `Block` statement. More on this in Section 2, "Scoping Variables with `Block`"

not taking the derivative for some reason? One way to settle that is to remove the third line and then see what happens:

```
(* Cell 1.7 *)
(* Second Notebook *)
f = x^2 + x;
fPrime = f'
↪ (x + x^2)'
```

Okay, so that looks a little funny, but we remember that it gave the right answer for sin, right? So let's duplicate this cell[6] and try again with `Sin` just to make sure:

```
(* Cell 1.8 *)
(* Second Notebook *)
f = Sin;
fPrime = f'
↪ Cos[#1] &
```

My goodness, what is happening here? It is not obvious from looking at it, but that is shorthand *Mathematica* syntax for a new function. It is equivalent to writing `Function[x,Cos[x]]`, which simply means *"Here is a new function that takes x as an argument and gives Cos[x] as its value"*. This will be covered in more detail in Chapter 3, "Defining Your Own Functions", but for now we don't have to worry about the specifics.

We can make a guess that maybe, for some reason, the `'` operator won't work on plain expressions like x^2+x, but it will work on things that *Mathematica* recognizes as full-fledged functions like `Sin` and `Cos`. So, how do we make *Mathematica* recognize x^2+x as a function? Let's try using `Function` to define our expression as a proper function of the variable `x`:

```
(* Cell 1.9 *)
(* Second Notebook *)
f = Function[x, x^2 + x];
fPrime = f'
↪ Function[x, 1 + 2x]
```

And we can see now that we have `Function[x, 1 + 2x]` as our result, which means *"Here is a new function that takes x as an argument and gives 1 + 2x as its value"*, and indeed this is what we want for the derivative of x^2+x. Now let us re-introduce the line of code we removed earlier:

```
(* Cell 1.10 *)
(* Second Notebook *)
f = Function[x, x^2 + x];
fPrime = f';
f'[π]
↪ 1 + 2π
```

So now we understand the nature of the bug: it wasn't that our `EvaluateDerivativeAtAPoint` function was wrong, rather it was that we were invoking it with the wrong *type* of argument. We gave it the *expression* x^2+x when in fact we should have given it the *function* `Function[x, x^2 + x]`. Now we can go back to our first notebook and correct the code in Cell 1.5:

[6]you can click on the bar on the right side of the cell to select it, hit "Ctrl-C" to copy, and then "Ctrl-V" to duplicate the contents into a new cell.

```
(* Cell 1.11 *)
EvaluateDerivativeAtAPoint[Function[x, $x^2+x$], $\pi$]
↪ $1+2\pi$
```

and now we are in good shape. Using this debugging strategy, we were able to fool around with fixing the bug in one notebook without messing up the code in our main notebook. This is an excellent habit, and it will save you mountains of trouble.

2. Scoping Variables with `Block`

Whenever you use a new variable in *Mathematica*, the kernel makes a new entry for that variable in the global symbol table. This can be a hassle when debugging, because you may want to re-evaluate a cell over and over while you change it, and sometimes values from previous calculations can sneak in and mess up your code. For example, let's say that you are building a list of intermediate expressions for some larger calculation. Maybe you have to calculate the value of a function and its derivative at certain points and store them in a list[7]:

```
(* Cell 2.1 *)
PointsForFunction = {0, 1, 2};
PointsForDerivative = {3, 4};
ListOfValues = {};

(* Cell 2.2 *)
f = Function[x, $x^2+x$];
AppendTo[ListOfValues, Map[f, PointsForFunction]]
↪ {0, 2, 6}

(* Cell 2.3 *)
fPrime = Function[x, $2x$];
AppendTo[ListOfValues, Map[fPrime, PointsForDerivative]]
↪ {0, 2, 6, 6, 8}
```

Which looks all well and good, except we put the wrong derivative for `fPrime`! Let's fix that and re-evaluate Cell 3:

```
(* Cell 2.4 *)
fPrime = Function[x, $2x+1$];
AppendTo[ListOfValues, Map[fPrime, PointsForDerivative]]
↪ {0, 2, 6, 6, 8, 7, 9}
```

Hmmm. . . that can't possibly be right. It seems like we have too many values here, don't you think? Let's re-evaluate Cell 3 again and see if it does the same thing:

```
(* Cell 2.5 *)
fPrime = Function[x, $2x+1$];
AppendTo[ListOfValues, Map[fPrime, PointsForDerivative]]
```

[7]This may seem like a silly example, but it is one of the first steps in deriving BDF, Numerov, or Adams-style ODE solvers

```
↪ {0, 2, 6, 6, 8, 7, 9, 7, 9}
```

Okay, it looks like our list is growing by 2 units every time we run this cell. What could possibly be going on? Let's re-evaluate Cell 2 and see what we get:

```
(* Cell 2.6 *)
f = Function[x, x^2 + x];
AppendTo[ListOfValues, Map[f, PointsForFunction]]
↪ {0, 2, 6, 6, 8, 7, 9, 7, 9, 0, 2, 6}
```

Probably, clever reader, you have figured out the bug by now – Every time we re-evaluate Cells 2 and 3, new values are added to our `ListOfValues`, but we never bothered to reset it, so it just kept growing!

Now, how do we resolve such a puzzle? The answer is that we can use the `Block` function to keep our variables contained to just one region of the code. That will allow us to re-run cells over and over while we make adjustments, and *Mathematica* will automatically reset the variables for us each time. Check this out:

```
(* Cell 2.7 *)
PointsForFunction = {0, 1, 2};
PointsForDerivative = {3, 4};
ListOfValues = Block[{temporaryList},
temporaryList = {};
f = Function[x, x^2 + x];
AppendTo[temporaryList, Map[f, PointsForFunction]];
fPrime = Function[x, 2x + 1];
AppendTo[temporaryList, Map[fPrime, PointsForDerivative]];
temporaryList
]
↪ {0, 2, 4, 7, 9}
```

You can re-run that as many times as you like, and you'll never have to worry about values from old calculations sneaking into your list.

CHAPTER 3

Defining Your Own Functions

Defining one's own functions is very easy in *Mathematica*. For example, suppose you want a function that converts from Cartesian coordinates to polar coordinates. This is a map of the form $(x, y) \mapsto (r, \theta)$ using the relationships $x = r\cos(\theta)$ and $y = r\sin(\theta)$. We also want an inverse function of the form $(r, \theta) \mapsto (x, y)$ that will convert Polar coordinates to Cartesian. Let's see how this would look in code:

```
(* Cell 0.1 *)
CartesianToPolar[{x_,y_}]:= {√(x²+y²), ArcSin[y/√(x²+y²)]};
PolarToCartestian[{r_,θ_}]:= {r Cos[θ], r Sin[θ]};
```

$$\texttt{CartesianToPolar[\{x\_,y\_\}]:= }\{\sqrt{x^2+y^2}\texttt{, ArcSin}\left[\frac{y}{\sqrt{x^2+y^2}}\right]\}\texttt{;}$$

We can invoke these functions by using the `[]` operator, as discussed in the section "Invoking Functions". Let's see what happens when we do.

```
(* Cell 0.2 *)
CartesianToPolar[{1,1}]
↪ {√2, π/4}
```

Since these two functions are one another's inverse, we can compose them and recover our original input:

```
(* Cell 0.3 *)
point = {1,1};
PolarToCartesian[CartesianToPolar[point]];
↪ {1, 1}
```

1. Using `Block` for Scope

Notice how, in the above definition of `CartesianToPolar`, we calculate $x^2 + y^2$ in two different places. This isn't the end of the world, but it's a little inelegant, so let's clean it up by calculating it first and using it later as a variable:

```
(* Cell 1.1 *)
CartesianToPolar[{x_,y_}]:= (
r = √(x²+y²);
{r, ArcSin[y/r]}
);
```

That looks much nicer. Cleaning up the code in this way will make your code more readable, and that will make it easier to debug and share with others.

However, there is a small side effect: the value stored in `r` will "leak" out of your function. To see what this means, let's do an example:

```
(* Cell 1.2 *)
CartesianToPolar[1,1];
r
↪ √2
```

Now, why should `r` even exist outside of where it is defined in the `CartesianToPolar` function? That's because *Mathematica* makes all new variables *global* by default. This means that unless you indicate otherwise, variables created anywhere in your program will be accessible from anywhere else. Global variables are not necessarily bad, but if you don't keep an eye on them they can be a total nightmare.

Fortunately, *Mathematica* provides a construct known as a *block* which allows you to set up some local variables and automatically erase them when you're done. You can make a block by calling the `Block` function like this:

```
(* Cell 1.3 *)
CartesianToPolar[{x_,y_}]:= Block[z,
z = √(x^2 + y^2);
{z, ArcSin[y/z]}
];
```

Now if you repeat the experiment from Cell 5, you will see that no new value is stored in the variable `z`. That is because the block automatically set `z` up as a local variable and erased it when the block ended. Keeping variables like this contained in a block will help keep your code clean, because it will keep cells from adversely interacting with one another.

2. Accepting Functions as Arguments

Frequently it is useful to take a second function as an argument for a function you have defined. There are no specific use cases for this, but if you are familiar with the technique, you will be able to recognize when it will be helpful.

As an example, let's say that you are preparing a report and you would like all of your graphs to be displayed uniformly. You could manually go throuhg your code and set all of your `Plot` settings to be the same, or you could define a single abstract function to take certain arguments and plot all of your graphs in the same style.

Let's assume that in this report, we will be plotting a comparions of *Exact Solutions* versus *Approximate Solutions* for a set of differential equations. That means we want our custom plot function to take two arguments

(1) The Exact Solution as a *Mathematica* function
(2) The Approximate Solution as a list of points to plot

so we know that the *signature* of our function will look like the following:

```
(* Cell 2.1 *)
AwesomePlot[ExactSoln_, ApproxSoln_, LowerBound_, UpperBound_]:=
```

This is because we will need to accept not only the Exact Solution and the Approximate Solution, but also the interval on which those solutions should be

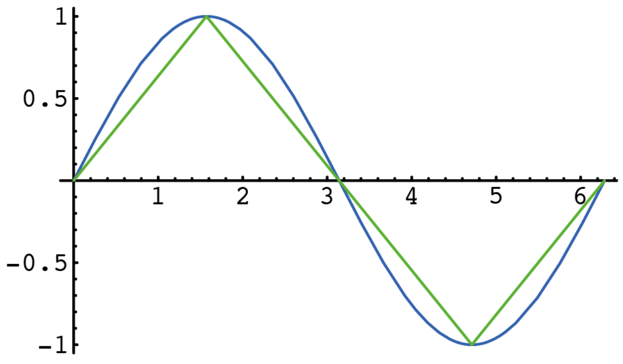


FIGURE 1. `AwesomePlot[Sin,{0,1,0,-1,0},0,2π]`

plotted. This is part of the normal information we would pass to `Plot`[1]. Now we want to display both of these functions on the same graph, preferebly in different colors, so that we may compare them. Thus the body of our function may look like:

```
(* Cell 2.2 *)
ExactGraph = Plot[ExactSoln[x], {x, LowerBound, UpperBound},
PlotStyle -> Blue];

n = Length[ApproxSoln];
StepSize = (UpperBound - LowerBound)/(n - 1);
ApproxPoints = Table[{ApproxSoln[[i]],(i - 1) * StepSize
+ LowerBound},{i,1,n}];
ApproxGraph = ListPlot[ApproxPoints, PlotStyle -> Red];
Show[ExactGraph, ApproxGraph]
```

And we can see that this will form a very handy tool that will make sure all of the plots in your report are very consistent, and it will make your code much, much cleaner.

3. Recursive Functions

It is sometimes handy to define a function that turns around and calls itself. This can come up in Math when building recurrence relations, but it can also be a stylistic technique to help keep your code clean and readable. Clever use of recursive functions can make code more concise, and thus more transparent and demonstrable. As stated many other times in this book, code clarity is vital to collaboration.

3.1. Calculating the Fibonacci Sequence. The Fibonacci Sequence is a famous integer sequence that appears unexpectedly in nature and aesthetics. For many, it is the first clue that the world we live in is inherently mathematical. The nth term in this sequence, labelled F_n, can be calculated as follows:

[1]You may recall that normally `Plot` is invoked by using an expression, not a function variable, as the first argument (i.e. `Plot[Sin[2 x], {x,0,1}]`). This is because `Plot` evaluates its arguments in a non-standard way.

$$F_0 = 0 \tag{1}$$
$$F_1 = 1 \tag{2}$$
$$F_n = F_{n-1} + F_{n-2} \tag{3}$$

Now, how do we build this in *Mathematica*? Notice that in Equation 3, F_n is defined in terms of previous members of the sequence. Let's rephrase these equations in function notation and see if it becomes a little clearer:

$$F(0) = 0 \tag{4}$$
$$F(1) = 1 \tag{5}$$
$$F(n) = F(n-1) + F(n-2) \tag{6}$$

So now we can see that the function $F(n)$ *depends on itself.* That is what the notion of a recursive function is all about! So, let's try to build this in *Mathematica*:

```
(* Cell 3.1 *)
F[n_] := F[n - 1] + F[n - 2];
```

We can hit "Shift+Enter" to evaluate this cell, and we get no complaints from *Mathematica*, so everything must be fine, right? Let's try to calculate `F[2]` and see if we get 1:

```
(* Cell 3.2 *)
F[2]
↪ Recursion depth exceeded
```

So. . . that's not cool. . . But what does that error even mean? It means that our function `F` called itself over and over and over again until *Mathematica* said "Enough! This thing looks as though it will never stop, so I'm going to stop it for you!". But why did it not stop at `F[0] = 0` like Equation 4? Well, we didn't tell it when to stop, so when it got to `n = 0`, it called `F[0]` which called `F[-1]` and `F[-2]` and kept right on rolling down into the negative numbers.

That being said, how do we make a recursive function stop at a certain point? As with most things in *Mathematica*, it's pretty easy – we just tell it when, like this:

```
(* Cell 3.3 *)
F[0] = 0;
F[1] = 1;
F[n_] := F[n - 1] + F[n - 2];
```

What happens here is that the function `F` actually has different rules that it can execute depending on what the input is. This is the *Mathematica* equivalent of a piecewise-defined function. It is as though we had written:

$$F(n) = \begin{cases} 0 & \text{for} \quad n = 0 \\ 1 & \text{for} \quad n = 1 \\ F(n-1) + F(n-2) & \text{for} \quad n \geq 2 \end{cases} \tag{7}$$

When a function has many different rules with different patterns, *Mathematica* will try to match them *in the order they are given*[2]. The first two rules are expressed without _ signs in the arguments. This indicates that they should match exactly the value given. This means the first rule is matched *only* when `n = 0` and it is skipped otherwise.

There are many other types of wilcards, and we will explore these in the next section. It is recommended that you do exercises 5 and 6 before proceeding.

3.2. Building your own Derivative operator. Exercises.

(1) In Cell 1, why were the functions defined with their variables inside a list ({})? What would be different if the curly braces were le ft off? How would this affect composition in Cell 3?
(2) Consider the code given for `AwesomePlot` and explain why `n - 1` yields a more accurate `StepSize` than `n` would.
(3) Expand the code for `AwesomePlot` so that it takes a string to use as the title for the plot
(4) Expand the code for `AwesomePlot` so that it automatically labels plots as “Figure 1”, “Figure 2”, etc without explicitly takin g an argument. Hint: It should contain a variable that gets incremented every time you call `AwesomePlot`.
(5) Build a recursive function that can generate the Fibonacci Word of length `n`.
(6) Build a recursive function that can test the Collatz Conjecture for a given Natural number `n`.

[2]Try quitting the kernel and moving line 3 to the top in Cell 1. Re-evaluate the cell and see if you can make a conjecture about why the sequence does not finish in this case.

CHAPTER 4

Lists, `Map`, and `Table`

This chapter introduces what computer scientists refer to as *functional programming.* To understand some of the impact of this, we begin by discussing the fundamental data structure of functional programming, the List.

A List is similar to an *array* that you might have encountered in other programming languages. One of the main differences is that Lists are designed to grow, whereas arrays are designed to take up a fixed amount of memory.

Lists in *Mathematica* can be constructed very simply by the following statement:

```
(* Cell 0.1 *)
A = List[];
```

or equivalently

```
(* Cell 0.2 *)
A = {};
```

however, the former style should be preferred as it is more explicit[1]. Lists can be grown by *appending* elements to them. For example, in order to create the list $1, 2, 3, 4$, we could do the following:

```
(* Cell 0.3 *)
AppendTo[A, 1];
AppendTo[A, 2];
AppendTo[A, 3];
AppendTo[A, 4];
```

Of course, we could also define this list explicitly as follows:

```
(* Cell 0.4 *)
A = {1,2,3,4};
```

and this is of course much more concise. Generally, if a list can be defined without doing any calculations, i.e. if it is a constant, you will define it all at once as we have done here. However, if the list must be built up programmatically, it is necessary to use the `AppendTo` function as described above.

We can also access list elements directly by using the `[[]]` operator. For example, to get the first element of `A`, we do as follows:

```
(* Cell 0.5 *)
A[[1]]
```

Note that unlike most programming languages, *Mathematica* begins indexing lists with `1` rather than `0`. This is good, because it is in keeping with most mathematical

[1]Good programmers always aim for their code to be *clear* and *explicit.* This makes it easier for others to read their code and understand its meaning, and that is good for friendship.

notation for dealing with vectors and matrices. We can also assign values explicitly to a list by using the `[[]]=` operator like so:

```
(* Cell 0.6 *)
A[[2]] = 5;
```

You cannot grow a list in this fashion. That is, for a list like `A` that has 4 elements, you cannot add a fifth element by

```
(* Cell 0.7 *)
A[[5]] = 5.0;
```

This is because, like arrays in C or Java, lists have a fixed width, and *Mathematica* will not allocate more memory for them unless you explicitly tell it to (using the `AppendTo` function).[2]

1. The `Map` Function

You may have seen the `Map` function mentioned previously in this book. Now we will discuss it in detail. `Map` takes two arguments: a *Mathematica* function `F`, and a List of domain elements for `F`. Recall the following example from Chapter 1, "Stuff You've Seen Before":

```
(* Cell 1.1 *)
Map[Cos, {0, 0.1, 0.2, 0.3, 0.4, 0.5}]
```

This coode generates a list of range values for the `Cos` function that correspond to the domain points $\{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$. In languages like C, FORTRAN, or Python, you may have used a "for-loop" to do a task like this. You would have to declare an empty array, then declare an index, and then apply the function to each element of the domain array. However in *Mathematica* and other languages that support Functional Programming, the `Map` function can do all of this for you at once.

You may have the occasional programming task that involves transforming a list from one form to another. For example, suppose you have a list of complex numbers that you'd like to plot in the plane:

```
(* Cell 1.2 *)
(* You can type i as:  ''Esc''+ii+''Esc'' *)
SomeComplexNumbers = {1 + 2i, 2 + 3i, 4 + 5i, 6 + 7i};
```

The first thing we need to do is figure out how to transform each number into a point in the plane. The rule we will use for this is $z \mapsto (Re[z], Im[z])$, which means that the x value of the point will be the Real part of the complex number, and the y value of the point will be the Imaginary part of the number. Let's express this in *Mathematica* as follows:

```
(* Cell 1.3 *)
ComplexToCartesian[z_]:={Re[z],Im[z]};
```

You can try that on a few numbers to get a feel for how it works. Now that we know how to transform an individual number, how can we easily apply it to a whole list of numbers? That's right, we use `Map`.

[2]This is a much different model than languages like Perl or Ruby that automatically grow an array when you make an "out of bounds" assignment. This strategy, while handy, can slow down your program because memory will be allocated frequently in small chunks rather than infrequently in large chunks.

```
(* Cell 1.4 *)
Map[ComplexToCartesian, SomeComplexNumbers]
↪ {{1,2},{2,3},{4,5},{6,7}}
```

So we see that we have successfully transformed a list of complex numbers into a list of ordered pairs. If we check the documentation for `ListPlot`[3], we see that it will accept a list of ordered pairs and plot them. As such:

```
(* Cell 1.5 *)
OrderedPairs = Map[ComplexToCartesian, SomeComplexNumbers];
ListPlot[OrderedPairs]
```

2. The `Table` Function

`Table` is a very handy function that will generate Lists, Matrices[4], 3D Arrays[5], etc based on an expression. For Example,

```
(* Cell 2.1 *)
Table[a_{i,j}, {i,1,2},{j,1,3}]
```

$$\hookrightarrow \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix}$$

You can see in Cell 1 that `Table` takes an expression as its first argument. That expression should contain some symbols, or "Dummy Variables" that will be used for iteration. The other arguments given to `Table` are lists that describe how the "Dummy Variables" should iterate. In this particular example, we build a matrix with two rows and three columns.

If we wanted to build a function to transpose a square matrix, we could do so as follows using the `Table` function:

```
(* Cell 2.2 *)
SimpleTranspose[matrix_]:= Block[{n},
n = Length[matrix];
Table[matrix[[j,i]],{i,1,n},{j,1,n}]
];
```

This function simply measures the length of the given matrix, and then creates a table whose i, j-entry is the j, i-entry of the original matrix, resulting in the transpose[6].

[3]Hint: check the documentation. Relying on the documentation is a great way to become more fluent in *Mathematica*

[4]Matrices are just lists of lists

[5]3D Arrays are lists of lists of lists

[6]`SimpleTranspose` only works for square matrices. *Mathematica*'s built-in `Transpose` function works for rectangular matrices as well

APPENDIX A

Finding Help in *Mathematica*

1. Documentation Center

You can access *Mathematica*'s built-in documentation center by pressing "Shift" + "F1" from anywhere in the application. The Help Center provides the official documentation for all fo the built-in functions you will use, and for many additional packages that are included in *Mathematica* but not enabled by default.

Each function has its own page in the Help Center. Try searching for `Plot`. You will see several things:

(1) A summaray of usage detailing all the ways that `Plot` can be invoked, and what the differences are.
(2) A section showing further examples and more sophisticated tricks.
(3) A list of related functions, such as `Plot3D` and `Manipulate`.

One of the difficulties in reading the documentation given in the Help Center is that it references other documentation in the Help Center, and you may get the feeling that you are going in circles. This is quite normal, and no cause for worry. Every programmer encounters this difficulty when learning a new language, though they may not admit it. The trick really is to keep following the trail until you build up enough context that what you are reading starts to make sense.

It is important, in situations like these, to put aside (momentarily) the problem for which you are searching and focus on what is presented in the documentation. Play with the examples that are given. Try changing parts of them, rearranging the arguments, throwing in different numbers or values. If you mess it up, you can always just close the Help Center and come back to it later. Above all, it is vital to *be patient*.

The single most frustrating experience you can have while programming is to spend ages hunting through the documentation for *The Wrong Way to Do It*. The docs, you see, are designed to show *The Right Way to Do It*, and it may sometimes take a while to recognize that for what it is.

You may trust that the authors have spent many long sessions dumbfounded in front of the *Mathematica* Help Center only to realize in a great flurry of excitement that the answer is completely different than we initially expected. One cannot rush that sort of thing; this stuff just takes time.

2. Stack Overflow

One of the greatest resources for *Mathematica* help on the internet is StackOverflow/Mathematica. Also becoming very popular is mathematica.stackexchange.com, which is a separate site dedicated enitrely to *Mathematica* issues. Both of these should serve you well in your quest to find answers to your problems.

StackOverflow is a forum in which users are encouraged to give insightful answers in order to receive *points*, which amount to social capital. Thus, the answers you will find on StackOverflow are consistently of a higher quality than those you will find on other programming-related websites.

On StackOverflow, all questions are *tagged* according to which programming language or platform they pertain to. The link above will take you directly to the *Mathematica* questions, and from there you can search for more specific information about your question.

Generally, the probability that the issue you have run into when programming in *Mathematica* (or any language) is unique is very low, so the odds are in your favor that someone has encountered a very similar problem before you. Thus, the challenge is to alter your search terms judiciously until you stumble on a problem that seems to fit the issue you are dealing with.

One way to help with this process is to speculate a few guess about what the problem might be. For example, are you using some functions whose behavior you don't quite understand? Maybe you getting a weird output that doesn't look like what you think it should? Searching on StackOverflow for things like "ListPlot no graph" is more likely to get you useful results than "no graph".

2.1. Asking Questions on Stack Overflow. Accounts on StackOverflow are free, and joining this community will help you learn a great deal about both *Mathematica* and programming in general. Searching through other people's questions and the suggested answers can be very informative, and it is a good intellectual exercise to try to solve some on your own (and you may even be rewarded with profile points!).

Of course, membership on StackOverflow also allows you to post questions. Understand though that the community expects you to have done some work before you post a question. Generally, before posting new questions, it is advisable to:

(1) Try your query on a major search engine like Google or Bing. At least try all the links on the first page to see if they have anything helpful to offer.
(2) Try searching WolframAlhpa. While this is not exactly a search engine in the usual sense, its results usually contain *Mathematica* code which is occasionally helpful.
(3) Try the *Mathematica* documentation. You can access this by pressing Shift+F1 while running *Mathematica*. Specifically, you will want to look at the examples that are available on the documentation page for each function that you might have questions about. Frequently functions can take different arguments, or give different output depending on parameters or options, so it may be that you need to invoke your function in a slightly different way.

If you have done these things and still not found the answer to your question, then it will be cool to post your question on Stack Overflow. One thing to note is that while people on Stack Overflow are usually very eager to help, they need enough information about what you are trying to do to be able to understand where you might be running into trouble. Generally, posting a single line of code may not be enough. Also, keep in mind that folks on the internet are very unlikely to be familiar with your research, so it is important to track down the issue as specifically as possible.