1 Basics of Mathematica

1.1 Introduction

Mathematica is a notebook-style IDE for the Wolfram language. It has a wide array of built-in functions for data analysis, plotting, optimization problems, maching learning, 3d diagrams etc. Its core features are its extensive use of symbollic manipulation and capacity for functional programming, allowing to design programs with capabilities unobtainable in other languages. This tutorial, will familiarize you with how variable, function definition and application.

Mathematica primarily uses file types with ".nb" and ".m" extensions. The ".m" extension is reserved for packages and is interchangeable with the older ".l" extension. We will be primarily focused on working with notebook files, however all aspects of the language itself will apply for code written in notebook files as well.

1.2 Setting up a Notebook

After loading up the Mathematica program, you will be greeted with the following welcome menu. Hit "New Document" to open a new notebook file.

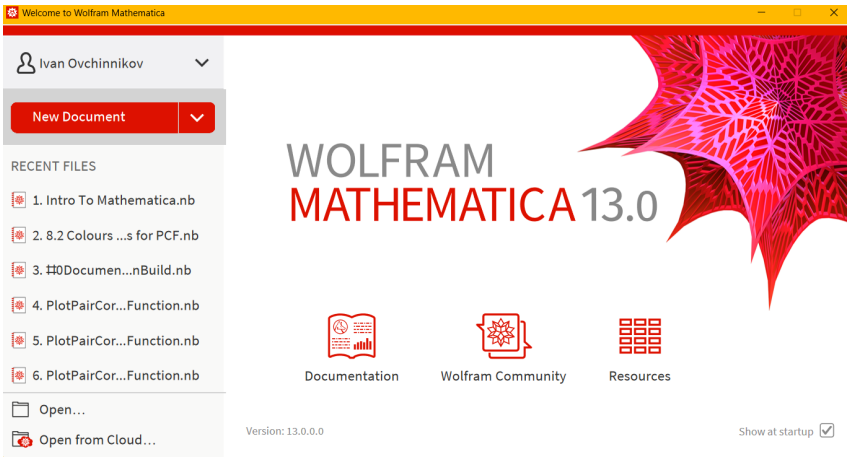


Figure 1: Welcome Page

You should now be able to see an empty notebook that looks something like this:

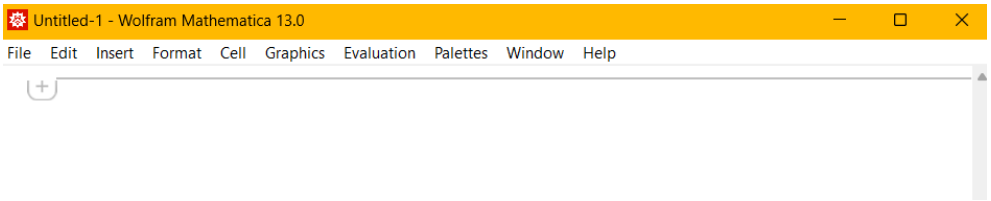


Figure 2: Sample empty Mathematica Notebook

Before we begin with writing code, let's first make a few remarks on the notebook system itself. By default, Mathematicashares all variables between all currently open notebooks. For example, if you had two concurrently open notebooks and defined a variable in the first one. You will be able to access the same variable under the same name in the second notebook. While this is quite convenient, it is also useful to know how to disable this, and keep all definitions local to the notebook. You can do so by setting "Notebook's default context" to "Unique to this notebook" in the evaluation tab

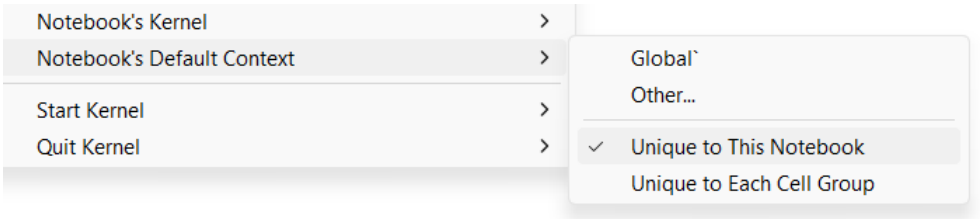
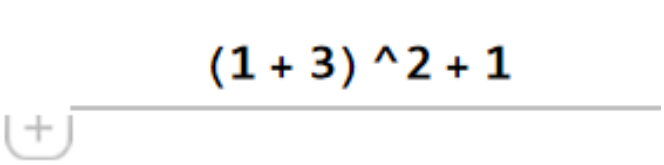


Figure 3: Forgetting to enable local context and then accidently defining different variables with the same name across different notebooks can lead to rather convoluted errors (which are always confusing to figure out the cause of).

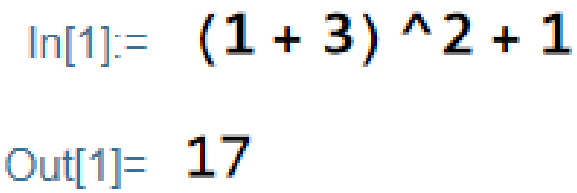
Of course, feel free to switch between Global and unique context to your convenience.

1.3 Cells and Executing Code

Similar to a jupyter notebook, a Mathematica notebook is made up of different cells, in which you can write and execute code. The output of the code in the cell is the displayed in an "output" cell which is generated just below your input cell. To run a cell, click in it with the cursor and simultaneously press "shift" and "enter".



(a) Input cell with simple mathematical expression.



(b) Upon pressing shift-enter, mathematica evaluates the code in the cell and displays the output in an output cell below.

Figure 4: Evaluating a cell in Mathematica

Note that if you saved the file (in the usual way with ctr+s), all outputs will be saved as well. This way when you open the file again, you will be able to see all your evaluations, without having to rerun the code. This is especially convenient when generating plots and other kinds of visuals to share/use later on devices with lower computational power.

1.4 Defining Variables

1.4.1 Numbers and Strings

Now let's see how to define variables. This is nearly identical to any other programming language. Write the variable name, an equal sign and the value you want to assign to the variable in a cell. Now run the cell using shift enter, and that variable will now be accessible throughout the whole notebook (or all your notebooks if you did not define context to be local as in Figure 3).

Note that by default, whatever value you assign to a variable, once you run the cell, will be printed in the output cell. Sometimes you will be defining rather "long" variables (like a list of a few hundred points), and would rather not have that clutter your notebook. To suppress output of any line of code, just put a semicolon after it (this will also suppress output of any command, as we will see later.

```
In[1]:= x = 1

Out[1]= 1
```

(a) Defining a variable and running the cell, prints the value of the variable in the Output cell.

```
In[2]:= y = 2;
```

(b) Placing a semicolon on a line of code, will suppres its output (no output cell was generated as we see)

```
In[3]:= x + y

Out[3]= 3
```

(c) Now that we have defined variables x,y we can use them in other cells.

Figure 5: Defining a variable in Mathematica

Numbers are of course not the only kinds of variable you can have in Mathematica. Another simple example, are strings. Strings are wrapped in double quotes. You can concatenate strings using the "¡¿" operator. Note that variables in Mathematica do not have "strictly assigned" data types, meaning that you can make a variable containing a number, and then assign a string to it (or any other object). To check what type a variable currently has, run its name followed by "[[0]"

```
In[9]:= string1 = "abc";
        string2 = "def";
        string1 <> string2

Out[11]= abcdef
```

(a) Defining and concatenating two strings.

```
In[14]:= string1 = 3;
         string1 = "myString"

Out[15]= myString
```

(b) Variables are not strict. You can save any object type into any variable (regardless of what was saved into it previously).

```
In[16]:= string1[[0]]

Out[16]= String
```

(c) If you want to know the type of a variable, just run its name followed by a "[[0]]"

Figure 6: More about variables

1.4.2 Lists

A very useful type of object in Mathematica are lists. This allows you to save a "list" of different items into a variable. Note that unlike languages like java and C++, the types of objects in the same list need not be the same.

A list is written down by placing a bunch of values separated by commas inside of curly brackets. If you want to specifically access the nth element of a list, write "listName[[n]]". If you want to truncate your list, and consider a list of only the first n elements, write "listName[[;;n]]". Note that Mathematica has **indexing starting with 1**! So [[1]], returns the first element of a list.

```
In[19]:= sampleList = {1, 1, 2, 3, 5}

Out[19]= {1, 1, 2, 3, 5}
```

(a) Defining a list of numbers

```
In[24]:= sampleList[[3]]
         sampleList[[ ;; 3]]

Out[24]= 2

Out[25]= {1, 1, 2}
```

(b) Here we obtain the 3d element of the list we defined in Figure 7a and separately look at the same list but cut after the 3d element.

```
In[23]:= points2D = { {1, 2}, {2, 3}, {3, 4} }
Out[23]= {{1, 2}, {2, 3}, {3, 4}}
```

(c) Lists can have any object in them. Even other lists! This is a common format for saving 2d points. The list above has the following points (1;2); (2;3); (3;4)

Figure 7: Basics of list manipulation

1.5 Functions

1.5.1 Format of Functions

In Mathematica calling a function has the following format:

```
functionName[ var1,var2,....,varn]
```

Where var1 and so forth, are the necessary arguments the function takes.

When code is compiled, the function call is then "replaced" by the value the function outputs.

1.5.2 Defining a function

Following the following format, we will now look at defining a function. This is in some sense very similar to how you define variables. The function name is followed by the variables[1] you want to pass inside the function (their local names that will be used). A ":=" symbol is placed, followed by the code you want your function to output. Here is an example of definitions of a few simple functions:

```
In[26]:= addTwo[x_] := x + 2

In[27]:= addTwo[1]

Out[27]= 3
```

(a) A function that adds two to the number you input.Note that the variables input into the function are highglighted green inside the function.

```
In[30]:= addMult[x_Integer, y_] := (y + x) * x

In[31]:= addMult[3, 4.5]
```

(b) A function that expects the first of its entries to be an integer.

```
In[35]:= addMult[3.1, 4.5]

Out[35]= addMult[3.1, 4.5]
```

(c) The function defined in Figure 8b, expects the first entry to be an integer. If your input is anything else, it does not run.

Figure 8: Defining a few simple functions

Sometimes however, you want to create a more complicated function. One that has multiple steps and has its own local variables defined. For this purpose, we can use the Module (or somewhat similarly Block) commands. The Module command allows you to define local variables, accessible only within the module. It has the following format:

```
Module[ {variables you want to be accessible only in the module}, Whatever code you want to be executed ]
```

Here is an example

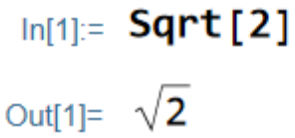
```
In[36]:= complicatedFunction[myList_] := Module[ {num1, num2},
           num1 = myList[[1]] + myList[[2]];
           num2 = myList[[2]] * myList[[3]];

           num1 + num2     ]

In[37]:= complicatedFunction[{1, 2, 3}]

Out[37]= 9
```

Figure 9: We define a function called complex function, within which we can use "local" variables "num1" and "num2". Note that any line that does not have a semicolon, is rreturned as output. Alternatively you can wrap the thing you want to return inside of "Return[]"

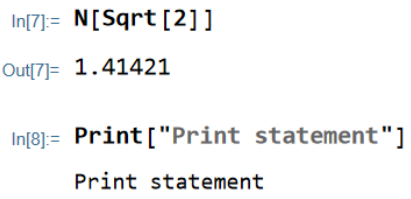
[1]Note that these variables should be preceeded by an underscore. If you want to specify the type of the variable that should be passed into the function, you can write it after the underscore.

1.5.3 Common Functions

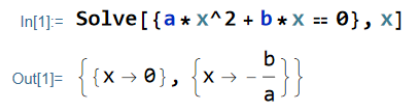
Computational Functions Now let's look at a few common built in functions (and related nuances). First begin by considering some standard numeric manipulations. Take as an example the function Sqrt[], which outputs the square root of the input. We would expect Sqrt[2], to return something like 1.41421. If you were to try it out yourself, you would however quickly find that it simply outputs the symbol $\sqrt{2}$. This is the "symbolic" part of Mathematica. It attempts to manipulate all equation etc. analytically as much as possible, and by default will not give you a numerical output. If you have some expression you want to be evaluated as a number, place it inside the N function (see example below). Due to this symbolic ability, Mathematica is quite good at solving equations (and differential equations) analytically (Solve and DSolve functions). Another key function is the Print command. It can be used to display output, without placing it in an output cell.



(a) Mathematica by default keeps all values as symbols



(b) Use the N command to convert symbolic representation to decimal. Print command does not output content to return cell, simply displaying it.



(c) The Solve and Dsolve can be used to symbolically solve equations (differential equations). It is also possible to solve systems.

Figure 10: Commonly used computational commands

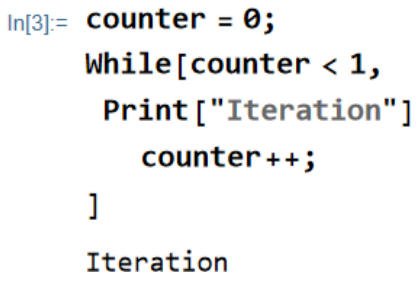
All other common commands (sin,tan,log etc.) are also available with their "natural" naming (note that names of all built in Mathematica functions start with capital letters).

Loops

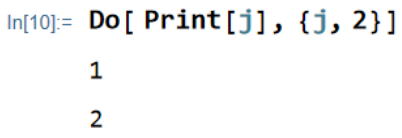
Mathematica provides access to all the standard loops (for,while and do). The while and for loops, have the same overall structure as you would expect in any language. The do loop evaluates some expression, with more specific iteration parameters. Its key advantage over the for loop, is that it automatically localizes the counter variable. It is generally the preferred/more convenient loop to use [2]

The do command has the following possible syntax:

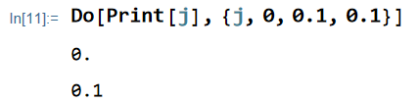
```
Do[ Code to be iterated, { local counter variable name, specifications of what to count over  } ]
```



(a) Standard while loop. Note that the counter variable must be defined externally.



(b) Sample do loop. Iterates over the variable j, from 1 to 2.



(c) Sample do loop with more specifications that have the following syntax: iterator (j), iteration start value (0), iteration end value (0.1), increment step of variable (0.1).

Figure 11: Examples of simple loops. Note that these loops have more possible options and arguments, details on which you can find on Mathematica's official documentation page.

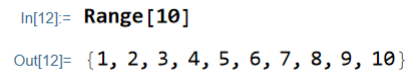
List Manipulation

[2] A more detailed explanation of why the Do loop is better than the For loop can be found in *this stackexchange post*. Point is, don't use For loops.

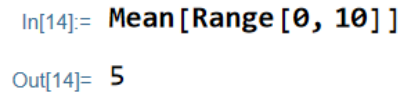
A very important aspect of Mathematica is proper list manipulation. Dealing with large lists can become rather computationally heavy and it is important to be aware of the built in tools to deal with this. One of the natural tendencies, is to generate lists either by continuously appending elements to an empty list. This is usually very inefficient and memory consuming. It is far better to try and employ the built in "table" command. It works very similarly to the do loop, but instead of simply running some code each time, it places the output as an element of a list.

```
Table[What to Put in each element, {iterator variable, iteration options}]
```

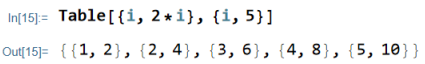
It is worth noting the Range (generates a list of integers from 1 to n) and Mean (returns mean of a list) commands.



(a) Range[n], returns a list of integers from 1 to n.



(b) Range[a, b], returns list of integers from a to b.Mean returns average of numbers in list.

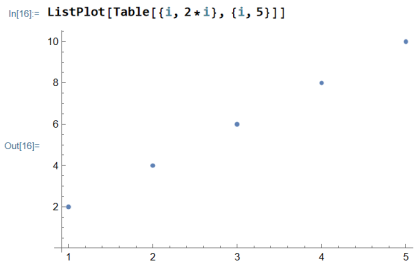


(c) Table used to generate a list of points on a line with equation $y = 2x$

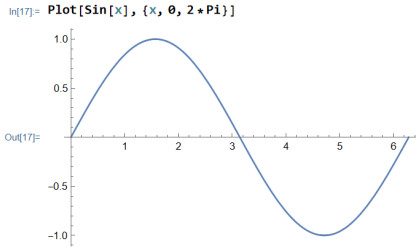
Figure 12: Sample list manipulation commands.

Graphic Functions

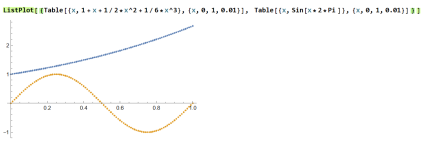
Mathematica also has a wide array of commands for making graphics, plotting etc. Here we will consider the two simplest ones: Plot and List Plot.



(a) ListPlot allows you to plots a list of 2d points (format show in Figure 7c)



(b) Plot allows you to plot a specific function, over some range (in this case from 1 to 2 π.)



(c) It is possible to plot multiple datasets at once. Here you need to modify your input to be of the form: dataset1,dataset2,....

Figure 13: Examples of plot commands. Note that there are many more customizability options (axes labels, tickmarks, colours, marker size/shape etc.) which you can read about on the official Mathematica documentation website.

1.6 Applying Functions

At this point, we have learned about common functions available in Mathematica as well as how to make our own. However, at times, they appeared to be rather clunky to type, with all of the brackets (especially, when you have multiple functions nested in each other). For this purpose (and many others), the Wolfram language has a few "shortcuts" for function application.

The first and most common is the "@" operator. It is used to apply functions that have a single input and has the following format:

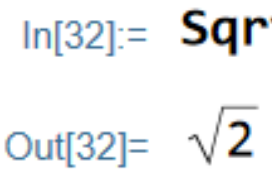
```
myFunction@Input
```

Note that the "@" operator, applies the function to the input coming after the operator. Sometimes, it is more convenient to place the function after the input. For this we have the "//" operator which has the following format:

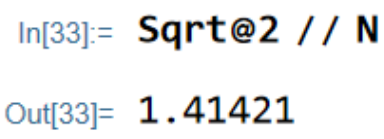
```
Input // myFunc
```

Sometimes, you might want to apply a function to every element in a list. In this case, the @ operator should be preceded by a "/".

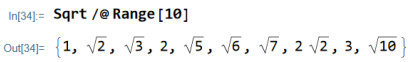
```
myFunction/@List
```



(a) Apply the Sqrt function to a number.



(b) Apply Sqrt function to a number, then convert that to decimal format.



(c) Obtain a list of square roots of integers from 1 to 10.

Figure 14: Examples of shorthand function applications.