COMP 161 - Lecture Notes - 19 - Some Mathematica

In these notes we take a quick side-step from C++ to look at the basic Mathematica we need for Project 2.

Mathematica

Mathematica is a powerful system for all things mathematics[1]. We'll be using it to work with execution time data gathered by C++ programs and written to csv files. The typical way to work with Mathematica is through a NOTEBOOK. With notebooks we can input one or more statements from the Wolfram Language[2] in a single notebook CELL and then evaluate them right there. Notebooks have the advantages of interactive environments like Dr.Rackets interaction window and the physicality of source documents.

[1] https://www.wolfram.com/language/fast-introduction-for-programmers/

[2] https://www.wolfram.com/language/fast-introduction-for-programmers/

In the notes that follow I'll give you statements in Mathematica that you can and show enter into a Notebook. To evaluate a *cell* hit *Shift-Enter*. You can also Evaluate an entire notebook through the Evaluate drop-down menu. When you evaluate a cell, the results are printed within that cell. If you wish to suppress the output of a statement, then end the statement with a semi-colon. I do not recommend putting all your code in a single cell. Use cells like procedures and put little logical chunks of code that carry out clear distinct tasks in a single cell. Just be aware that when you write variables and definition in one cell and use them in another they must first be evaluated before their used.

As functions are introduced you should look them up in the Documentation which is accessible on the web and through the Help drop-down menu. Any text surrounded by (* and *) is a comment and will not be evaluated when the cell is evaluated. You'll soon notice that when calling functions we put arguments in square brackets rather than the parenthesis we're used to from C++ and Racket.

Lists

When we read our csv data into a notebook we'll end up with a list of lists. You can more or less think of this as a table. Each row corresponds to a line from the file and the elements of the row are the values delineated by commas. Task number one is to learn how to manipulate and work with lists[3].

[3] https://reference.wolfram.com/language/guide/ListManipulation.html

Let's start by specifying a few lists by hand before we get to the lists that result from reading our csv files. List literals are written by

comma separated values surrounded by curly braces. Here we define a few variables with list values.

```
threes = {3,3,3,3}
alist = {1,2,3,4,5,6,7}
alolon = {{1,2,3},{5,6,7},{8,9,10},{11,12,13}}
```

We can also use the *Table* function to construct lists. Here we construct the same lists using Table.

```
threes = Table[3,{4}]
alist = Table[i,{i,1,7}
alolon = Table[ 3*(i - 1) + j, {i, 4}, {j, 3}]
```

We select list items using double square brackets in much the same way that we select elements from vectors and strings in C++. The big difference is that the first element of a list is at index 1, not index 0. When selecting from multi-dimensional lists use commas to separate dimensional coordinates.

```
alist[[3]]
alolon[[4,2]]
```

The really wonderful thing is that we can select more than one element at once by passing a list of indexes rather than a single index and that the order of the data is the order in which we list the indexes.

```
alist[[{1, 3, 5}]]
alist[[{3, 5, 1}]]
{4, 52, 3, 6}[[Table[2, {10}]]]
```

We can also use a Span to select a sequence of elements. With spans we can also specify the step size which in turn lets us do reverse spans.

```
blist = Table[20 - i, {i, 1, 20}]
blist[[1 ;; 5]]
blist[[6 ;;]]
blist[[;;10]]
blist[[;;]]
blist[[20 ;; 1 ;; -1]]
blist[[1 ;; 20 ;; 3]]
```

We'll be mostly concerned with 2D lists, so we need to be sure we know how to select rows,

```
alolon[[2]]
```

columns,

```
alolon[[;;,1]]
```

or parts of rows and columns

```
alolon[[;;2,{1,3}]]
```

Making Tables

We're looking at tabular data sets. Let's see how we can build a pretty table from a list of lists. First lets make up a fake, table-like data set.

```
mytab = Table[10*i+j,{i,0,20},{j,0,10}]
```

We might like to view this in an organized table format with TableForm.

```
TableForm[mytab]
```

TableForm has several useful options. Perhaps the most important are for specifying headers.

```
mytab,TableForm[mytab, TableHeadings ->
  {Table[i, {i, 20}], {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"}}]
```

To put captions or labels on graphics, like a table, we use the Labeled function. I find the default spacing to be too tight and tend to add some space between the label and the graphic. It's also convenient to name the finished graphic.

```
tabview =
 Labeled[TableForm[mytab, TableHeadings ->
    {Table[i, {i, 20}], {"a", "b", "c", "d", "e", "f", "g", "h", "i",
      "j"}}],
  "Table 1: Some data showing some relationship",
  Spacings -> {Automatic, 2}]
```

Now that we have pretty table, we probably want to save it as an image file. The *Export* procedure makes this super easy.

```
Export["TimesTable.svg",tabview]
```

Mathematica will determine the file format based on the extension you give in the file name. In this case, I'm exporting as Scalable Vector Graphic[4] because it tends to scale up in size better than formats like png or jpg.

[4] svg https://en.wikipedia.org/wiki/Scalable_Vector_Graphics

Importing csv Data

Importing a csv is as easy as exporting and image.

```
datatab = Import["mydata.csv"]
```

The Import function determines the file format from the extension. For csv files, it will read the data as a 2D list where each line is a row. Here we're saving the resultant list of lists as the variable *datatab*.

Basic Statistics

We concerned with tables of data, so let's make some fake data and use it to explore some basic statistical gathering and data manipulation. Here we'll make a 20 row by 10 column table of random numbers.

```
randtab = RandomVariate[NormalDistribution[],{20,10}]
```

The first thing we might like to do is compute some basic statistics like minimum, maximum, and average. The functions Min and Max will give you the relevant values for the whole table.

```
Min[randtab]
Max[randtab]
```

If we'd like to get the Min/Max by row, then we can use Map. The Map function applies a function to each element of a list. In this case, we'll apply Min/Max to each row.

```
Map[Min,randtab]
Map[Max,randtab]
```

To get the Min/Max for a column we can first Transpose the table. This turns rows into columns and columns into rows. Thus, the Min of the each row in the transpose is the Min of the columns in the original.

```
Map[Min,Transpose[randtab]]
Map[Max,Transpose[randtab]]
```

The function Mean gives you the average for each column. If you want row averages, then Transpose is once again your friend.

```
Mean[randtab]
Mean[Transpose[randtab]]
```

Plotting Points

Mathematica and do a lot of different visualizations. We'll focus solely on plotting points in 2D space with *ListPlot* and work with *randtab* from the previous section. There are three ways to use ListPlot. If you give it a list of values, then it treats those values as the

y-coordinates and assigns each value's location as it's x-coordinate. You see this when you plot a row of the table.

```
ListPlot[randtab[[1]]]
```

Plotting the table causes ListPlot to treat each sub-list[5] as a set of points for that list's location in the main list. So row 1 is treated as a set of points for x equal to 1.

[5] row

```
ListPlot[randtab]
```

When your data is set out incrementally such that list position is the x-coordinate you want, then this is great. If, however, the position in the list is not the x-coordinate, then you need ListPlot option 3– plotting a list of $\{x, y\}$ pairs. The trick is we'll have to build this pairs.

Let's start with a simple case. Our table has 10 columns. Let's say column i corresponds to an x coordinate of $10i$ rather than just plain i[6]. First we need a list of our y-coordinates.

[6] row 1 is 10, 2 is 20, and so forth

```
xs = Table[10*i,{i,10}]
```

Now all we need to do is "zip" these lists together such that the ith element of *ys* matches up with the ith element of the row. There is no Zip function in Mathematica, but in this case we can use Transpose yet again. If we build a list with the first row as our x data and the second as our y data, then the Transpose of that list is our x-y pairs.

```
Transpose[{xs,randtab[[1]]}]
```

It might be nice to have a defined function called Zip so that we can say something like *Zip[xs,ys]* rather than use Transpose directly. If anything, it's a good excuse to see how we write functions in Mathematica. So, here's the definition for Zip!

```
Zip[xs_,ys_] :=
  Transpose[{xs,ys}];
```

The first line is the function header. We same the name followed by the parameter list. Each parameter name is followed by an underscore. The header is separated from the body by :=. The body is then just the Mathematica expression that carries out the intended function. We can now zip our row to the y-coordinates using Zip.

```
points = Zip[xs,randtab[[2]]];
ListPlot[points]
```

Now another case[7]. What if each row is a set of points for an x-coordinate that is not the row's index? Our table has 20 rows. Let's say the row x-coordinates correspond to the first 20 squares:

[7] the case you encounter in project 2

```
rowxs = Table[i^2 , {i,20}]
```

Now we want to zip row *i* of our data table with row *i* of our x-coordinate table. We can solve this problem with Map. While we're at it. Let's make a function called *MakePoints* that takes the list of x-coordinates and the table and maps out the x-coordinates to each row. Let's look at the complete definition for MakePoints then pick it apart.

```
MakePoints[xs_, ytab_] :=
  Map[Zip[Table[xs[[#]], {i, Length[ytab[[#]]]}], ytab[[#]] ] &,
  Table[i,{i,Length[ytab]}]]
```

First we note that we're mapping this Function[8],

[8] It *has* to be a function if it satisfies the Map signature

```
Zip[Table[xs[[#]], {i, Length[ytab[[#]]]}], ytab[[#]] ] &
```

over the list of *ytabs* index values.

```
Table[i,{i,Length[ytab]}]
```

This is actually reminiscent of our vector loops in C++ as we're mapping over index values rather than the table itself.

The odd part here is the function:

```
Zip[Table[xs[[#]], {i, Length[ytab[[#]]]}], ytab[[#]] ] &
```

This is what's often called a LAMBDA EXPRESSION. It's an anonymous function or a function literal. In Mathematica we can write these using the & symbol. Everything to the left of & is the function body. The symbol # is used for the argument. In this case it's a one argument function. If you want to do multi-argument functions, then we just add the argument number after the #[9] symbol. We can tell from the context that the argument of our unnamed function must be an index number. We use it to select an element from *xs* and a row from *ytab*. Let's put this together and state in English what we're seeing.

[9] #1 is the first argument, # the second, and so on

The expression

```
Zip[Table[xs[[#]], {i, Length[ytab[[#]]]}], ytab[[#]] ] &
```

is the function that takes Zips a list of the same length as the #th row of *ytab* containing only the #th element of *xs* with the #th row of *ytab*.

If you don't like this shorthand, then you can use the *Function* function for specifying anonymous functions.

```
Function[i, Zip[Table[xs[[i]], {i, Length[ytab[[i]]]}], ytab[[i]] ]
```

Either way, the Map+Anonymous Function combo is extremely powerful. It can save you the trouble of defining and naming every little function you wish to map on a data set.

Now that we can tag each row with its x-coordinate, we can create and plot our list of points[10].

[10] technically list of lists, but Mathematica doesn't care

```
morepoints = MakePoints[Table[i^2,{i,Length[randtab]}],randtab];
ListPlot[morepoints]
```

ListPlot Spruce Up

The default options for ListPlot are usually pretty good, but there are a few essentials we need to look at to ensure good plots.

- *AxesLabel* To add labels to the axes
- PlotRange *To specify the min/max y-coordinates displayed*

AxisLabel is pretty self-explanatory. PlotRange lets you give exact values or you can use *All* or *Automatic* to let Mathematica figure it out. Automatic is the default and will on occasion lead to cutting off some max values. In which case, All is a good alternative. Explicit values are nice when you need to fit things to a critical range. Here's a few examples.

```
morepoints = MakePoints[Table[i^2,{i,Length[randtab]}],randtab];
ListPlot[morepoints,AxesLabel -> {"Size","Time"},PlotRange -> All]
ListPlot[morepoints,AxesLabel -> {None,"Time"},PlotRange -> {-4,4}]
```

We can then use the Labeled function to add a caption and Export to produce an image file.

```
mypoints =
Labeled[ ListPlot[morepoints,AxesLabel -> {"Size","Time"},PlotRange -> All],
  "Figure 2: Size and Time...",Spacings -> {Automatic,2}]

Export["datapoints.svg",mypoints]
```