Data Visualisation and Analysis Report – Lab 1

Christopher Murdoch, cm151

Demonstrated to Amit on 04/02/22

GitHub

Part one consisted of creating a GitHub repository, setting up the file structure, and creating a GitHub pages site for the exercises. The repository is public and can be viewed at https://github.com/ChrisJMurdoch/DataAnalytics, along with the file structure.

I tried to keep the structuring of files as modular as possible. For this reason, each exercise is encapsulated in its own named folder under exercises. Each of these exercise folders contains a exercise.html and exercise.js.

The repository also contains a vendor folder used to store the third-party libraries that I use, such d3 and Prism.js.

The site can be viewed locally when cloned, but is also hosted at https://chrisjmurdoch.github.io/DataAnalytics/ using GitHub pages.

Exercise 1

Using JavaScript's string interpolation, I added a text element to the screen displaying the d3 version: 7.3.0.

Exercise 2

I created two paragraph elements in HTML, one with a span. I used d3 to select one paragraph and style it blue and bold, and I underlined the span in the other paragraph.

Exercise 3

Putting the d3 append call in a loop, I created 10 divs, each stating their numerical position in the list. I also conditionally-coloured the divs with a lambda:

.style("color", i<=5 ? "red" : "green")

Exercise 4

Following on from Exercise 3, and using *select("div")*, I could select and style the first element of the list. I also changed the text to say "first".

Exercise 5

Many of d3's functions use the builder pattern. That is, the functions return a reference to the element upon which they are called. This means that multiple calls of functions such as *attr, style,* and *text* can be chained together. Chaining these functions, each on a new line, neatens up the code and help keep the intention clear.

Exercise 6

Exercise 6 demonstrates d3's ability to integrate data and HTML elements. Using the *data* function, subsequent calls to a set of elements are mapped to respective data elements. These elements can be accessed with an anonymous function or lambda passed instead of a value for functions such as *attr, style,* and *text*. The following lambda can be used to insert the data as text:

```
.text( (d) => `This element's respective data is ${d}.` );
```

Exercise 7

This exercise is very similar to the last, but involved conditionally-colouring the elements based on their respective data entries. The following code allowed me to do this succinctly:

```
.attr("color", (d) => (d.value>=50 && d.value<=100) ? "red" : "yellow";
```

Exercise 8

The number of existing HTML elements may not match the number of data entries, but d3 allows for an easy fix. The *enter* and *exit* functions allow for the procedural creation and deletion of elements, respectively.

Using the enter function followed by append, I was able to bring the number of elements up to match the data size. Conversely, I used the exit function to remove any extra elements.

Exercise 9

Because d3's core principle is the use of data in documents, it provides functionality to easily retrieve and parse this data. The csv function it provides targets a URL, and asynchronously parses and loads it.

The csv function allows the specification of a parsing function: one that takes in a line of data and outputs the way in which it should be parsed. Provided with a lambda such as:

```
(d) => d.firstname
```

The function parses the whole document, finally returning an array of all the first names present. This runs asynchronously so as to avoid blocking up the main thread's execution.

The csv function returns a promise which, upon fulfilment, triggers the 'then' function with the parsed data. This function is useful for actually using the data, as trying to use it immediately after the csv call will run before the data is read and parsed.

For this exercise, I counted the number of Mr and Mrs present in the Titanic's manifest, the sexes of the passengers, and calculated the average age for males and females onboard.

Exercise 10

Similar to exercise 9, this exercise reads in a CSV of data, and presents aggregate information about the occurrence of heart failure in varying age ranges. As an additional function, I calculated the mortality rates for each age group based on the number of heart failure cases ending in death.

Exercise 11

Scalable Vector Graphics (SVGs) provide an easy way to create graphical displays in HTML documents, and d3 can be used to procedurally manipulate them. This exercise simply generated 4 lines in the shape of a square, displayed on the page.

Exercise 12

This exercise builds a scene from an external file. I used a CSV file with standardised dimension columns to allow for the use of multiple, distinct shapes. The function reads in the CSV and adds the elements automatically so elements in the scene such as squares, circles, ellipses, and text can be managed from the external file.

Exercise 13

This exercise uses buttons with scripted onclick events to add and remove elements using the *enter* and *exit* functions. Starting with a set amount of elements on screen, the user can either add more randomly-placed circles, or remove additional elements.

Exercise 14

This exercise uses the previous data to create a bar chart. By reading in the data and using the value as the length of each bar (multiplied by a scaling factor), each bar can easily be generated to scale.

Exercise 15

Using the value of each bar, divided by the maximum value in the dataset, I was able to map each bar to a value between 0 and 1. Using this value, I linearly-interpolated between white and red to display a colour for each bar based on how high the value is. The code also thresholds lower values to maintain a white colour for the bottom 10% of values.

Exercise 16

Using data to generate shapes for this exercise, I mistyped the square centres and created and interesting shape. Overlapping the circles and squares, I made teardrop-looking shapes that I believe would be useful for highlighting specific points on a graph or map.

Exercise 17

To colour the bars based on their data, the unscaled values had to be used. This meant indexing into the original array without using the scaling function generated for placement on the graph.

Exercise 18

Extending the example to use data from an external file was quite simple, but I had to make sure that the call was run once all of the data had been parsed from the CSV. I also provided a parameter that allows the user to specify which column is read into the data, but the function defaults to thew first column if not specified.

Exercise 19

Writing the code into the function just required exposing the important data as function parameters. To improve code reuse and maintainability, I loaded the code in from the previous exercise instead of copying it all over. This saved on time and meant that later fixes to the used function did not need to be duplicated.

Exercise 20

Creating an axis on each of the four sides of the graph required the transform: translate style function to move the top and right axes. One important thing to note was that the y-axis in CSS, like most GUI systems, starts at the top and increases downward. The top and right axes could also be coloured using the *attr* function.

Exercise 21

Adding an axis to the previous bar chart example was fairly simple, and I converted the existing bars to use the x-scaling functionality in the process.

Exercise 22

Encapsulating the sin line chart code in a function that allows external data loading was very similar to previous examples. Once again, I reused the CSV-parsing data from previous examples to improve maintainability.

Exercise 23

Calling the function on URLs worked, as long as the files were publicly available. I had a few issues with this, but it was solved after using GitHub raw links and refreshing the browser cache.

Exercise 24

To allow for drawing multiple lines on one graph, I added an optional 'svg' parameter. If no value is passed, a new graph will be generated and the line added. If an svg is passed, the line will be plotted on the existing svg with the same axes. Each call of the function returns the svg created/used so it can be added to with subsequent calls.

Exercise 25

By using *selectAll("dots")* followed with an *append("circle")*, I was able to create a custom-styled circle at each datapoint along the line.

Exercise 26

Creating triangles was a little tricker as they aren't supported by SVGs in the same way. I solved this issue by appending 'path' elements with their d attributes set to d3's generated triangle symbols.

As the code was reused from previous exercises, plotting multiple shaped lines was as simple as inserting the shape-addition code into the previous code, exposing the rendered shape name as a parameter.

Exercise 27

I added another parameter to allow the user to specify a number of indices with which the function would use to textually-display the respective points' values. The text was originally written over the points, so I added a vertical offset so that the text sat above the line.

Exercise 28

As the bar charts were already rendered in a very similar way using colour interpolation, modifying the code to use d3's scales was quite simple. I had to make sure to use the real data value, instead of the scaled value.

Exercise 29

I used the colour scales to produce a gradient effect on the line graph points. I used the y values for the lines to slide between two specified colours, but I could have also used the x axis to colour the lines as well.

Exercise 30

Adding more values to the pie chart example, i realised that some colours were appearing multiple times in a row. This is because d3 assigns the colours based on index, and **then** sorts the sections based on size. I solved this by removing the automatic sorting, but it could have similarly been solved by sorting the data itself before providing it to d3's automatic colouring system.

Exercise 31

d3 has a useful *centroid* function that returns the centroid for a specified arc. I had trouble using this for a while as most documentation/forums are very out of date, but after using *pie(data)* instead of *data* as the datum for the centroid call, I was able to position text labels at the centre of each arc.

Exercise 32

I had trouble getting the example for this exercise to work for quite a while, but I found out that the svg images cannot be displayed outside of SVG elements like displayed. Once I put it in an SVG element and transformed it to cover, everything worked fine.

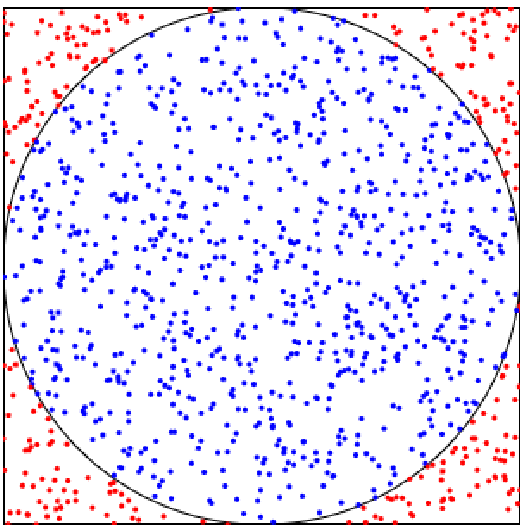
I found a free-to-use vector image of graph paper that I thought would suit the bar chart. I added it to the background and made the bars a little transparent so as to see the grid lines behind them.

Extra 1: Pi Approximation

I'm interested in the Monte Carlo Method for approximation and simulation, and have wanted to create a visualiser to demonstrate the concept. d3 seemed perfect for this case.

Monte Carlo method is named after the famous casino region as it uses randomness over many iterations to approximate different values and functions.

The system I made visually shows samples being placed randomly within a circle and square of the same width/diameter. Pi can then be approximated by dividing the number of samples that land in the circle by the number that land in the square (total samples), and multiplying by four. The approximation gets closer over time as more and more samples are placed.



Approximation formula

```
8937k (Samples in circle) * 4 /
11380k (Samples in square) = 3.14153
```

Accuracy

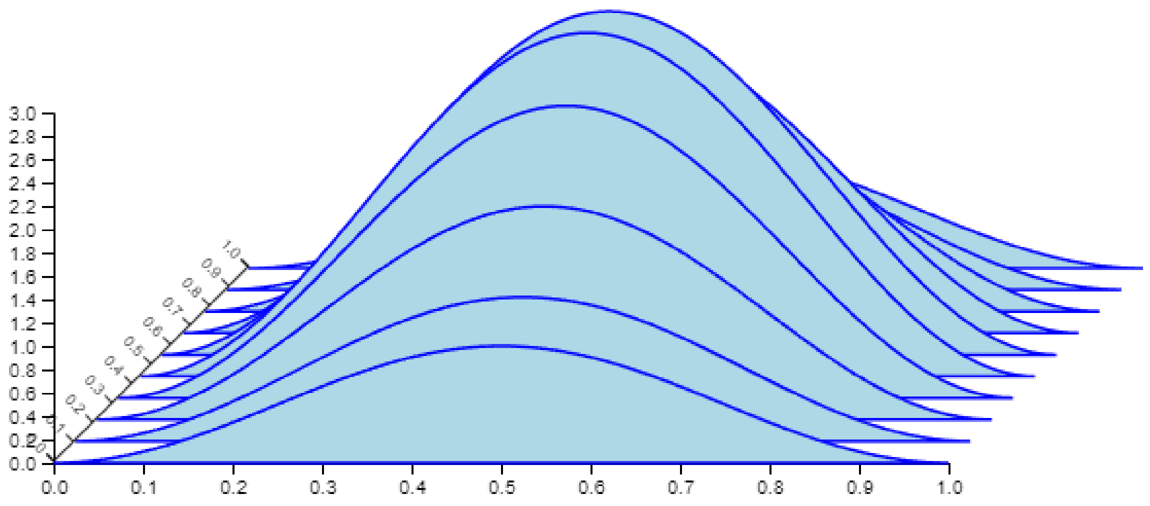
```
PI Real:   3.14159

PI Approx: 3.14153

Accuracy:  99.99811%
```

Extra 2: 3D line graph

Using the transform function to diagonally-translate different lines by varying amounts, I was able to create a basic 3D graphing tool that can display 3-dimensional functions. Each value on the z-axis uses a different 2D layer and they're rendered from back to front using the painter's algorithm to give the illusion of 3D. Any 3D function can be rendered but this example shows: (x, z) => sin(2 πx) sin(2 πz)


3.0
2.8
2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0