# Lab 2 Report

DEMONSTRATED TO: AMIT PAREKH   ON: 25/02/2022

CAMERON DOUGLAS – H00295794

## Lab Findings

**In this section I will briefly outline my findings from each lab exercise, along with relevant discussions on how I achieved these. Discussion about the extra exercises I added will be done at the end of this section.**

**Part 1:** This part focussed on incorporating CSS aspects into the D3 projects. Using CSS, I was able to set activity for different attributes by their class or type, based on mouse events occurring to other elements within the document.

**Exercise 1:**

Re-using the code to generate a line graph from the first lab, I was able to add a class attribute to each of the datapoints, then by using CSS keyframes, I could reference these nodes and change their colour on mouseover to create a pulsing effect.

**Exercise 2:**

I created a simple website using D3, appending a span and a div on the body of a page for each element in a list of data. The span contained the index label and the div, the value of the data. Then using the CSS + notation, was able to define span:hover + div{} behaviour which selects the next div element and I set its display from none to inline so the div containing the value appears on mouseover of the label.

**Part 2:** This part focused on moving the event handling from the CSS into D3 and gives far more control and increased functionality over the outcomes of the mouseover. Using the d3.on(mouseover), I am able to now add and modify elements based on the events occurring elsewhere in the document.

**Exercise 3:**

Within the .on(mouseover) attribute, I created an anonymous function which selects the object and then sets its style attributes, these attributes are then returned to their original state in the .on(mousout) attribute.

**Exercise 4:**

I first created an SVG and appended a circle, and then similarly to exercise 3, I was able to modify the attributes of the circle within the .on(mouseover) block.

**<u>Exercise 5:</u>**

I first append a blank SVG to the body and then in its on mouseover block, I first exit any text which may already be there (this ensures that there is only one text label and not a trail of old labels), I then append a new text element, setting its x and y position to be the first and second elements in the d3.pointer(event) array respectively. I then set the text to also be these values so the text follows the mouse with the coordinates of the mouse.

**<u>Part 3:</u>** This part introduced the use of transitions to seamlessly move between attribute changes, rather than the abrupt changes that have been occurring up to now. By using D3.transition() and its submethods such as duration(), ease() and delay(), I can control these transitions to create more interesting and dynamic element changes.

**<u>Exercise 6:</u>**

I added a transition() after the transition to red, which changes the style to green, using duration(2000), I define that this change happens over 2 seconds.

**<u>Exercise 7:</u>**

After changing the colour to green, I also define that the width and height attributes also change in the same transition.

**<u>Exercise 8:</u>**

I moved the code from the previous two exercises into a .on(mouseover) block so that the update only occurs when the mouse is over the SVG element. I then also define a .on(mouseleave) to reset the size and colour of the square back original.

**<u>Exercise 9:</u>**

I added two extra elements and used easeElastic and easeLinear over the same transition to demonstrate the difference between them.

**<u>Exercise 10:</u>**

Adding to the previous circle exercise, I added the easeBounce attribute to the transition in both the mouseover and mouseout.

**<u>Exercise 11:</u>**

In this example, I replaced the circle with an SVG element and applied the same transition principles used in the previous exercises.

**Exercise 12:**

I first created a new bar3 variable and added it to the SVG, I then added a .transition() for bar3 in the update function and set its delay to 4 seconds using the .delay() function.

**Exercise 13:**

I wrote a function shrink(), which after a delay, resets all of the bars to their original size in the same way that the example code increases their size in update().

**Exercise 14:**

In the update and shrink methods, I also change the bars colours, they are originally blue, update changes them to red and shrink goes back to blue.

**Part 4:** This part introduced transitions and mouseover events to a more tangible and real world application, combining the learning about data presentation from the previous lab exercise, with the transitions introduced in this lab. It also introduced the handler functions which were passed to the mouseover function by name, rather than writing an inline function to create cleaner and more readable code.

**Exercise 15:**

I implemented the starter code provided. This code utilises the .on(mouseover) block and passes a function by name which is then used on mouseover. This function mutates the element's scale and colour and then also appends a text element with the element's value.

**Exercise 16:**

In the mouseover function, I use anonymous functions to get the x and y coordinates (which will be year and value respectively) I then set the x and y attributes of the text element to these values.

**Exercise 17:**

I used a colour range which is bounded by the minimum and maximum values from the data, I then call this colour range with the value of each bar in the mouseover function, so that on mouseover, the bar changes to a colour based on its value.

**Part 5:** This part introduced the merge() function which allows us to smoothly interpolate between datapoints. This means that we can switch data on graphs and charts smoothly, without re-drawing the whole chart to create a slicker and more appealing visualisation.

**Exercise 18:**

I created a third dataset and added an HTML button to switch to this new dataset. This button calls the update function which uses merge() to seamlessly switch between datasets, rather than popping one in and out

**Exercise 19:**

I added a parameter to the update function which allows me to specify a colour, when update is called on button press, I can set the colour I want each chart to appear in.

**Exercise 20:**

I selectAll rect elements in the SVG and then on mouseover, I append a text element with the bar's value above the bar.

**Exercise 21:**

I appended a left and right axis to the SVG which are scaled just as the previous two axes are.

**Exercise 22:**

I first declare a function called setupAxes which takes in a dataset and then draws the appropriate axes, in the update function, I simultaneously remove the old axes and then call setupAxes. This exit and enter happen almost instantaneously so the axes appear as though they just update rather than disappear and then re-draw. In update, I then check if the new dataset has fewer elements than the current dataset, if so, I remove as many bars as necessary. I then merge the new data in as before, creating a seamless swap of the bars onto the same axes.

**Exercise 23:**

Combining the line graph exercises from the previous lab with my solution from the previous exercises, I was able to swap between two line graphs rather than bar charts.

**<u>Part 6:</u>** This part introduced the theory behind interpolation and outlines how D3 and JavaScript are able to interpolate between different datatypes such as dates, colours and numbers. It also introduces the attrTween() function, which interpolates between angles to smoothly draw in paths, such as those used in Pie Charts.

**<u>Exercise 24:</u>**

Type of returned function is: function

intr(0.2) => 16.2,34.4,5.2

The function returned is called an 'interpolator', given two values 'a' and 'b', it takes in a parameter 't' with the domain [0,1]. 't=0' will return 'a' and 't=1' will return 'b'. Values in between will interpolate between 'a' and 'b', biased towards one or the other based on 't'.

**<u>Exercise 25:</u>**

'cc' is the interpolator between red and green. The color returned from cc(0.5) is: rgb(128, 64, 0)

**<u>Exercise 26:</u>**

'date1' is: Wed Feb 16 2022 00:00:00 GMT+0000 (Greenwich Mean Time) and 'date2' is: Wed Jun 07 2000 00:00:00 GMT+0100 (British Summer Time). Interpolating between the two gives the date: Wed Apr 13 2011 00:30:00 GMT+0100 (British Summer Time). This is the point exactly half way between the two dates.

**<u>Exercise 27:</u>**

I added an additional dataset, and then in update, I append a new path and merge it with the original path, this creates a smooth transition between the two charts.

**Part 7:** The final part in the lab introduced the Force layout. This allows for a 'simulation' to be run which updates using ticks and can apply physical forces to otherwise static D3 objects. In this part, I experimented with centre forces, collision, dragging, links and radial forces, all with the goal of creating interesting and interactive visualisations.

**Exercise 28:**

I used a colour range which is bounded from 0 to *bound* (i.e. smallest and larges possible values in the data). Then when I create the nodes, I call this range, passing it the radius of the node, which colours it based on that radius.

**Exercise 29:**

I manually created a dataset using an inline array. This dataset is passed as before to the forceSimulation.

**Exercise 30 and 31:**

Using mouseover like before, I obtain the value of the calling node and then append a text element above the node with the text of the value. I also change the colour attribute of the node to highlight it on mouseover.

**Exercise 32:**

Also in an inline array, I defined a set of links, this takes a source and target index node. When initialising the simulation, I add an extra attribute .force('link',…) which takes in the array of links and defines a distance which the nodes are separated.

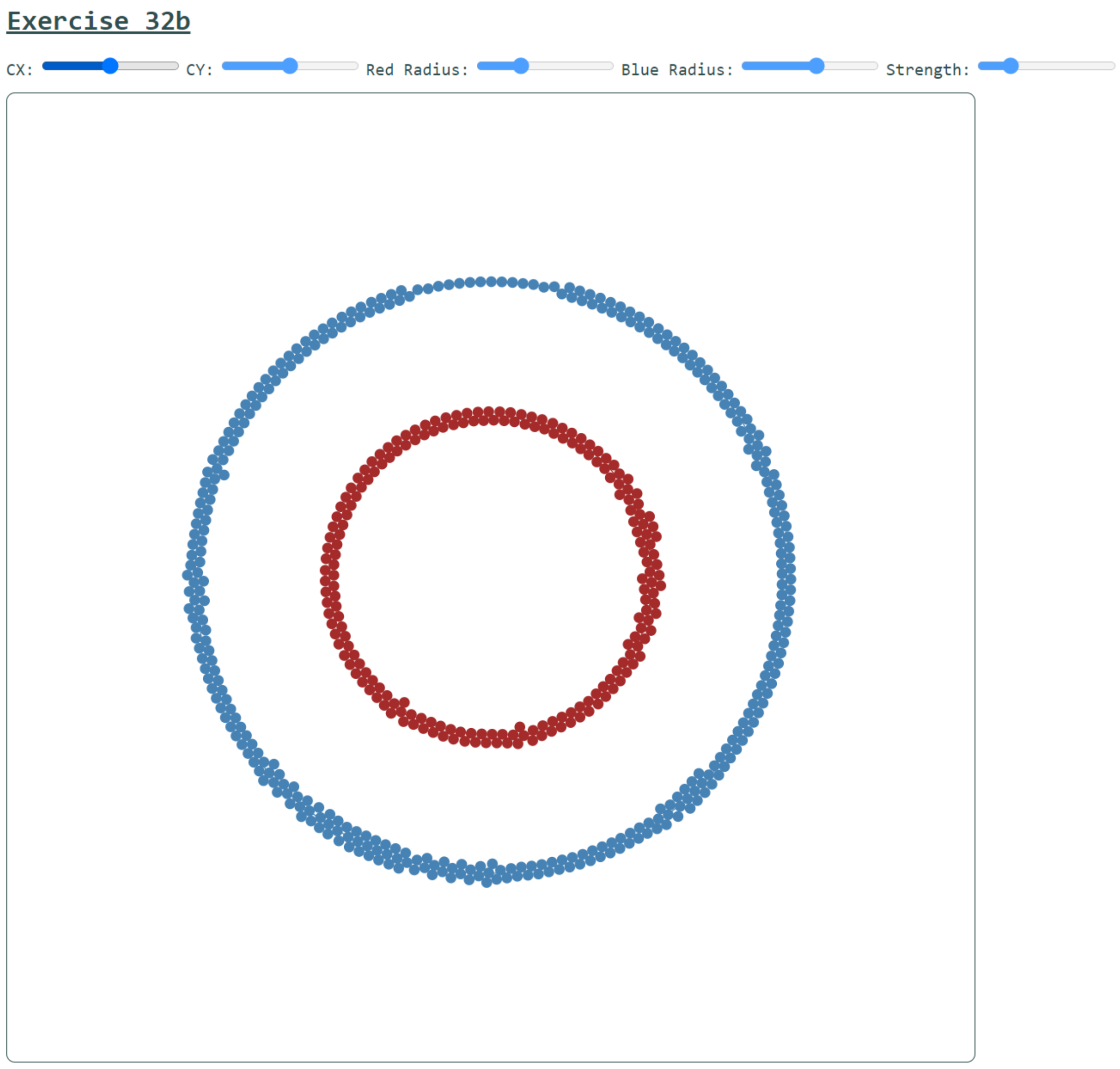
I then also added dragging functionality so that you could drag the connected pairs together. The function drag() is called on the node elements and defines the behaviour for each stage of the drag: start, drag and end. When the element is being dragged, its coordinates are set to the x and y values of the mouse, obtained from the event. Because of the force link defined earlier, if one element is dragged, it's linked nodes are also dragged along with it.

## Extra Exercise:

### Exercise 32b:

For this exercise I created an interactive display of forceRadial. I draw two large circles which act as guide circles to show where the force is being applied. I then create a set of random nodes of two types. I then setup a simulation which has a radial force. For each node, they get assigned a ring based on their type, they are then randomly distributed around the two rings.

Then using on input functions, I update the simulation and the guide circles with the new data from the sliders, allowing the user to control: x and y coordinates of the circles, radius of the circles, and the strength of the radial force.


Exercise 32b
CX:
CY:
Red Radius:
Blue Radius:
Strength: