F21DV: Lab 1

# Introduction

This report will serve as an accompaniment to html files containing the JavaScript code to solve lab 1 of the F21DV course. It will go through all 16 parts as described in the pdf overview and highlight key aspects of the code which led to the completion of each of the 32 exercises and mention other possible stylistic choices when appropriate. This report assumes some prior knowledge of JavaScript and will focus on describing the use of the d3.js v7 library.

I demonstrated to Ben Kenwright on the 4th of February 6, 2022, showing exercises 16 and 26.

# Part 1

A GitHub repository was created and showed to be helpful with code management. With the help of the desktop app for GitHub, I managed to smoothly operate on both my IDE of choice (WebStorm) and see the progress on my GitHub page. The repository is the following but may be set to private depending on when this link is clicked : <https://github.com/YoussefBonnaire/DataVisualisation>.

# Part 2

The ground up explanation of the d3.js package we are using with JavaScript is the Document Object Model (DOM) selection. Using the key terms “d3.select(“x”)” and “d3.selectAll(“x”) we are able to select the first, or all objects of type x and give them different styles and attributes. The following code shows how to change the first paragraph with to the different styles described:

<p>First paragraph</p>  
<p>Second paragraph</p>  
<script>  
 // change paragraph to use several styles  
 d3.select("p")  
 .style("color", "red") // Changes colour to red  
 .style('font-style', 'italic') // Changes font style to italic  
 .style('font-family', 'sans-serif') // Changes font to sans-serif  
 .style('font-size', '16pt') // Increases text size to 16pt  
 .style('font-weight', 'bolder') // Makes the text bold  
 .style('text-shadow', '2px 2px blue') // adds a blue 3d aspect with a 2px by 2px shadow  
 .style('word-spacing', '20px') // increases the spaces between words to 20px  
 .style('text-transform','capitalize'); // Capitalises the first letter of each word  
</script>

The objects can also be given different class attributes which can be used to select solely on this class using d3.select(“.MyClass”). Here is an example of how to add a class to a div object with the number 3 written inside and then set the colour to purple.

d3.select('body').append('div').text(3).attr('class', 'under-five')

d3.selectAll('.under-five').style('color', 'purple')

It should be explicitly mentioned that we are working with the version 7 of the d3.js package. This version has some useful simplifications compared to older ones such as chaining which allows the following 4 lines of code to be simplified as the 5th one.

var bodyElement = d3.select("body");  
var div = bodyElement.append("div");  
div.text("Hello World!");  
div.style('color', 'green');  
// Making text green in one line of code  
d3.select("body").append("div").text("Hello World!").style('color', 'green');

This allows for cleaner more legible code.

# Part 3

Data in the form can be bound to a DOM object. What this means is you can iteratively use each object in the data source and perform style and attribute changes which will appear in the dom. This is done by using .data(“MyData”) after selecting a DOM. If you have 3 div objects in your body already and your data is composed of 3 datapoints, when you selectAll the div objects, the datapoints will be bound to each div and can be populated as in the following code:

<div></div>  
<div></div>  
<div></div>  
<script>  
 let data = [{name: 'test', val: 1, color: 'red'},  
 {name: 'other', val: 2, color: 'green'},  
 {name: 'b', val: 3, color: 'blue'}];  
 let paragraph = d3.select("body")  
 .selectAll("div")  
 .data(data)  
 .text(function (d, i) {  
 return 'Color: ' + d.color; // Return value is used to set the 'text'  
 }).style('color', d => d.color); // Set text to said color  
</script>

The .style, .attr, and .text attributes are very strong as they can be populated using functions as is shown above. **Note**, the d => d.color is equivalent to function(d){ return d.color}.

# Part 4

In the case where the div elements do not exists and you decide to selectAll(“div”), you may still bind your data to this selection, the important difference then is that you then add the .enter() function which checks for corresponding div elements. If 1 element exist but we have 3 datapoints, only the first will be used, however, after the .enter() function we use the .append(div) function, it will add a div element for any datapoint missing one. An example of this is shown here below as we have not created any span objects prior, but the code will create it dynamically:

var myData = ['a', 4, 1, 'b', 6, 2, 8, 9, 'z'];  
console.log(typeof ('a'))  
var spans = d3.select("body")  
 .selectAll("span")  
 .data(myData)  
 .enter() // checks whether span element exist for data index  
 .append('span') // Adds non existing span elements  
 .text(function (d) {  
 return d;  
 })  
 .style('color', function (d) {  
 console.log(typeof (d))  
 if (typeof (d) == 'string') { // Selects objects of type string to change to blue  
 return 'blue'  
 } else {  
 return 'green'  
 }  
 });

Conversely, when a datapoint of the binded data is removed, calling the .exit() function on the spans variable above will remove the unused dom elements.

# Part 5

Being able to bind data from an array is very useful for a lot of basic activities, however, when we start having larger datasets they are usually provided in the form of external csv, json, tsv or xml files. Using these files in d3 is fairly simple, using for example a csv file, we can use it’s data in two ways:

This first way allows you to have a function acting on the entire data set gathered:

d3.csv(csvfile).then(data => {  
 *DoSomething*(data)  
 )

And this second way allows you to have a function working on each datapoint independently:

d3.csv(csvfile, function(data){  
 *DoSomething2*(data)  
 )

As an example, assume this data has a column for the age of each datapoint; DoSomething2 can count up the number of individuals above a certain age in the following way (having initialized age50 variable before similar to how it was done in exercise 10):

function *DoSomething2* (d) {  
 if (d.age >= 50) {  
 age50 += 1;  
 }

}

# Part 6

SVGs Scalable Vector Graphics (SVG) are a standard way of segmenting the screen to output visuals easily. This code below initialises the SVG and it’s size on the screen adding a border at the end to make it visible.

//Create SVG element  
var svg = d3.select("body")  
 .append("svg")  
 .attr("width", 400)  
 .attr("height", 400)  
 .style("border", '1px solid green');

Then within this svg we can easily append different shapes just like this line for example:

//Create blue line element inside SVG  
svg.append("line")  
 .attr("x1", 100)  
 .attr("x2", 200)  
 .attr("y1", 50)  
 .attr("y2", 50)  
 .attr("stroke", "blue")

Each shape has different attributes used to place it and determine it’s size, for example, a circle would be added in this way (this code is using data bound to the svg):

//Create circle element inside SVG

svg.append("circle")  
 .attr("cx", d.cx\_pos)  
 .attr("cy", d.cy\_pos)  
 .attr("r", d.width / 2)  
 .attr('fill', d.color);  
}

# Part 7

Bar charts use the idea presented here before considerably. After all, bar charts are simply rectangles placed along side each other (or above/under each other) with determined lengths. A crucial part of bar charts in SVGs are the scale which adjusts our rectangles appropriately. We will discuss that further in part 9. These are the simple parts of making a chart.

This part of the code creates the svg container we have our chart in:

var graph = d3.select("body")  
 .append("svg")  
 .attr("width", width)  
 .attr("height", barHeight \* age\_bins.length);

Then using data from a csv as can be seen more explicitly in exercise 14&15.html, we place the elements in which our bars will be located using the following code:

// Initialise g element to contain each bar at specified height  
var bar = graph.selectAll("g")  
 .data(age\_bins)  
 .enter()  
 .append("g")  
 .attr("transform", function (d, i) {  
 return "translate(0," + i \* barHeight + ")"; // Make sure containers are one below the other  
 });

Moreover, we can add the actual rectangles to each element g to show ours bars using the following code:

// Populate bars with data  
bar.append("rect")  
 .attr("width", function (d) {  
 return d \* scaleFactor;  
 })  
 .attr("height", barHeight - 1)  
 .style('fill', function (d) {  
 if (d > 100) {  
 return 'red'  
 } else {  
 return 'green'  
 }  
 });

Finally, we can add the label for each bar with the following code:

bar.append("text")  
 .attr("x", function (d) {  
 return (d \* scaleFactor);  
 })  
 .attr("y", barHeight / 2)  
 .attr("dy", ".5em")  
 .text(function (d) {  
 return d;  
 }).style('fill', function (d) {  
 if (d < 140) { // Specifies ranges for text color based on count  
 return 'yellow'  
 } else {  
 return 'purple'  
 }  
});

# Part 8

Although the shapes have different attribute tags to describe their location and size, with a few changes the same data can draw different shapes. For example given an array with data for the radius of circles, we can create squares with the same side length simply by changing the attribute name:

const data = [10, 15, 20, 25, 30];

This data can be used for both items below

g.append("circle")  
 .attr("cx", function (d, i) {  
 return i \* 100 + 50;  
 })  
 .attr("cy", 100)  
 .attr("r", function (d) {  
 return d \* 1.5;  
 })  
 .attr("fill", function (d, i) {  
 return colors[i];  
 })

The important note here is that while the circle is placed using cx and cy being its center, the rectangle x and y are the top left corner of it.

g.append("rect")  
 .attr("x", function (d, i) {  
 return i \* 100 + 50 / 2;  
 })  
 .attr("y", 100)  
 .attr("width", function (d) {  
 return d \* 1.5;  
 })  
 .attr("height", function (d) {  
 return d \* 1.5;  
 })  
 .attr("fill", function (d, i) {  
 return colors[i];  
 })

# Part 9

The bar chart which was created previously was simple. If the length of one of the bars’ lengths extended past the svg’s width, the bar would continue past the svg without being shown as it only exists within the svg. This can be remedied by using a scale function. For bar charts, the scale to use for most cases is d3.scaleLinear(). To use this you need to add a domain and a range. What this function does is that when it is given any number within the domain, it maps it to a number in the range in a linear way. For example say your data points go from 0 to 100 (your domain) and your svg is only of width 0 to 10 (your range) passing a datapoint with value 60 through the function will return the value 6 and allow your bar to remain in the domain. The code looks as follows:

const data = [50, 400, 300, 900, 250, 1000]

// Initiliase svg width  
var svg = d3.select("body")  
 .append("svg")  
 .attr("width", width)   
 .attr("height", barHeight \* data.length);

// Initialise scale range to fit in svg   
var scale = d3.scaleLinear()  
 .domain([d3.min(data), d3.max(data)])  
 .range([50, width]);

// Selects svg and bind our data to g  
var g = svg.selectAll("g")  
 .data(data)  
 .enter()  
 .append("g")  
 .attr("transform", function (d, i) {  
 return "translate(0," + i \* barHeight + ")";  
 });  
// Creates bars  
g.append("rect")  
 .attr("width", function (d) {  
 return scale(d); // Use scale to keep bars within svg  
 })  
 .attr('fill', 'blue')  
 .attr("height", barHeight - margin)  
 .style('fill', function (d) {  
 if (d <= 100) { // changes bar color based on numeric value of data  
 return 'green'  
 } else if (d >= 500) {  
 return 'red'  
 }  
 })

This code can be seen with all its variables in exercise 17.html. Furthermore, Exercise 18.html and Exercise 19.html extends this to accept data from a csv chart and puts it into a function, the only thing added in these exercises which is important to note is the svg creation must be added into the function if the required outcome is duplicating the chart as calling the function would instead populate the same svg over and over. Please see the code in the files as there would be little benefit to pasting the code into this report.

# Part 10

Adding an axis to a chart using the same scaling idea as placing the bars themselves. To get the x and y axis we need a horizontal (x) scale and a vertical (y) scale. These are defined in the following way:

const width = 400;  
const height = 300;  
var data = [10, 15, 20, 25, 30];  
var svg = d3.select("body")  
 .append("svg")  
 .attr("width", width)  
 .attr("height", height);  
// Define horizontal scale  
var xscale = d3.scaleLinear()  
 .domain([0, d3.max(data)])  
 .range([0, width - 100]);  
// Define Vertical scale  
var yscale = d3.scaleLinear()  
 .domain([0, d3.max(data)])  
 .range([height / 2, 0]);

Following the creation of the scale, the bottom and left axis must be created using d3 functions, d3.axisBottom() and d3.axisTop() which both need the scale to be bound to it.

// Initialise scale for bottom axis  
var x\_axis\_b = d3.axisBottom()  
 .scale(xscale);

// Initialise scale for left axis  
var y\_axis\_l = d3.axisLeft()  
 .scale(yscale);

This axis can then be placed by appending a graph element to the svg, specifying its location using a transform, then calling the axis function as follows:

// Place bottom axis   
svg.append("g")  
 .attr("transform", "translate(50, " + xAxisTranslate\_b + ")")  
 .call(x\_axis\_b)

// Place left axis  
svg.append("g")  
 .attr("transform", "translate(50, 20)")  
 .call(y\_axis\_l);

The same can be done for a stylistic right and top axis which can be seen in Exercise 20.html.

# Part 12 (11 missing)

In order to show how easy it is to use different external files interchangeably, in exercises 22-24 we used json files. These files contain the data of 1000 points which range between -1 and 1 which correspond to sine, cos and tan waves of domain between 0 and 1. These exercises perform the same steps as pervious ones, however, there are a few additional functions. The d3.extent() function was to find the interval of from the min(data) and max(data) which can be a handy simplification used in some cases; furthermore, when creating a line graph, unlike adding a simple line, you must append a path element to the svg to which you bind the data using datum and get to choose the type of line you desire using the d attribute as shown here below (x and y are the x and y scale functions):

svg.append("path")  
 .datum(shape\_data)  
 .attr("fill", "none")  
 .attr("stroke", "steelblue")  
 .attr("stroke-width", 1.5)  
 .attr("d", d3.line()  
 .x(function (d) {  
 return x(d.x)  
 })  
 .y(function (d) {  
 return y(d.y)  
 })  
 );

# Part 13

Additional labeling/markers can be added to the line plots in a very similar fashion, instead of adding a line for the d attribute, we can choose whichever d3 symbol or customized symbol we desire specifying its size:

svg.selectAll("dot")  
 .data(shape\_data)  
 .enter()  
 .append("path")  
 .attr('d', d3.symbol().type(d3.symbolTriangle).size(50))  
 .attr('transform', function (d) {  
 return 'translate(' + x(d.x) + ',' + y(d.y) + ')';  
 })  
 .style("fill", "green");

The notable difference is these symbols do not have a x and why coordinate which can directly be chained on creation and instead needs a transform attribute to be added to place it in the svg.

# Part 14

In This part we observe the ability to have a range/sequence of colours and using said colours in our visualization. To color a graph (or other data bound elements) we use a scale similarly to the scales used to limit the bar chart to the svg. The scale will map our data to a range of colours instead of a number. However, different scales can be used depending on the type of colour pattern you have chosen to use. For example, if you want the colour to change linearly (gradually) through the domain set out by the data, we can use scalelinear like this:

d3.scaleLinear().domain([d3.min(data), d3.max(data)]).range(["white", "blue"]);

If you want an iterative change between multiple colours we would use the scaleordinal function in this way:

d3.scaleOrdinal.domain([d3.min(data), d3.max(data)]).range(["gold", "blue", "green", "yellow"]);

Finally, if you wish to use one of d3’s interpolated colour schemes you would use a sequential scale function like so:

d3.scaleSequential().domain([d3.min(data),d3.max(data)]).interpolator(d3.interpolateViridis);

For all of these colour variations, for bar graphs all that is needed is to use the fill styling to make the colour of the bars dependent on the data as in the following code

bar.style('fill', function (d) {  
 return myColor(d);  
})

A demonstration of a bar graph colored following the viridis colour scheme is present in Exercise 28.html

Colouring a line graph on the other hand can be done completely differently. Instead of creating a normal scale, we can append a LinearGradient to the svg in the following way:

svg.append("linearGradient")  
 .attr("id", "line-gradient")  
 .attr("gradientUnits", "userSpaceOnUse")  
 .attr("x1", 0)  
 .attr("y1", yscale(d3.min(y)))  
 .attr("x2", 0)  
 .attr("y2", yscale(d3.max(y)))  
 .selectAll("stop")  
 .data([  
 {offset: "0%", color: "red"},  
 {offset: "100%", color: "green"},  
 ])  
 .enter()  
 .append("stop")  
 .attr("offset", function(d) { return d.offset; })  
 .attr("stop-color", function(d) { return d.color; });

This linear gradient acts in a similar way to the scales, the y1 and y2 attribute in this case act as the domain of our scale and the data offset and colour data bound to it acts as the range we want our data to be mapped to in those locations. This gradient can the be applied to the line by changing the stroke attribute as such:

svg.append("path")  
 .datum(shape\_data)  
 .attr("fill", "none")  
 .attr('stroke', "url(#line-gradient)")  
 .attr("stroke-width", 1.5)  
 .attr("d", d3.line()  
 .x(function (d) {  
 return xscale(d.x)  
 })  
 .y(function (d) {  
 return yscale(d.y)  
 })  
 );

# Part 15

There are 4 simple steps to creating a pie chart, after having initialized the svg, we must call the d3.arcs() function to decide size and shape (donut or circle) of the chart in this way:

var arc = d3.arc()  
 .innerRadius(120)   
 .outerRadius(radius);

Following this we must call the d3.pie() function to get the required angles from the data:

var pie = d3.pie();

Following this we can bind the data and angles to graph elements within the svg:

//Generate groups

var arcs = svg.selectAll("arc")  
 .data(pie(data))  
 .enter()  
 .append("g")  
 .attr("class", "arc")

Finally we can make the circles appear, in this case using a color map defined here:

var color = d3.scaleSequential().domain([d3.min(data),d3.max(data)]).interpolator(d3.interpolateViridis);

//Draw arc paths  
arcs.append("path")  
 .attr("fill", function (d) {  
 return color(d.value);  
 })  
 .attr("d", arc);

# Part 16

An image can be added as any other element, the only additional attribute one should add is the xlink:href attribute which finds the location of the image on the web or in the same path and then width, height, x, y, transform and many other styles can be added too, an example of this can be seen in Exercise 32.html

# Conclusion

With this report I have discussed all 16 parts of this course work supporting the 32 exercise files accompanying this report describing the basic functionality of D3 selection, data binding, DOM object manipulation, external file use, svg container use, bar, circle and line chart creation, scales for containing data in svgs, scales for data dependent colour schemes, additional markers on line charts, pie charts and adding images.