Tampere University

Safwane Benbba

# COMPARISON OF D3.JS AND CHART.JS AS VISUALISATION TOOLS

Bachelor's thesis
Faculty of Engineering and Natural Sciences
Examiner: Kari Systä
April 2021

# ABSTRACT

Safwane Benbba: Comparison of D3.js and Chart.js as visualisation tools
Bachelor of Science Thesis
Tampere University
Science and Engineering
April 2021

---

The web is a good platform for data visualisation due to its accessibility and outreach. JavaScript can be used to create such visualisations, but it is easier to do so using libraries expressly created for that purpose. D3.js and Chart.js are two examples of such libraries. Though both are free to use, for a long time the two were hardly mentioned together.

This thesis aims to analyse the core functionalities of the two libraries and compare them on different metrics. By going through the differences between D3.js and Chart.js, this works aims to find out the practical ramifications of those differences and the kind of applications each library is best suited for.

The results of the work carried out for this thesis indicate that D3.js is faster and tends to use less memory than Chart.js, except when it must render a high amount of object, in which case its memory performance breaks down. D3.js offers more features than Chart.js, but its usage is far more complex and requires more time and effort from the developer. As a result, D3.js is best suited for applications where performance is important or the visualisation is complex, whereas Chart.js is better used for straightforward charts with simple requirements.

Keywords: D3.js, Chart.js, JavaScript, data visualisation.

# CONTENTS

# 1. INTRODUCTION

Data visualisation is a collection of methods and techniques designed to visually summarise up raw information. The discipline can be considered a subset of data science. More formally, data visualisation may be defined as “The use of interactive, dynamic, and responsive visual representations of data to amplify cognition” and data as “Gathered, collected, modeled and produced details, calculations, and measurements, often assumed as facts, and forming the basis of reasoning, analysis, and understanding.” [1]

The aim is to transform raw data into something useful. This can involve developing complex statistical and theoretical methods, finding and exploring relationships, designing a way to show the results of such modelling, and then producing the visualisations.

These definitions might give the impression that the discipline is a modern invention, but while there is no comprehensive history of data visualisation, one can advent the rise of visualisation techniques to the works of William Playfair in the 18th and 19th centuries. The Scottish engineer introduced three large types of diagrams: time series and histograms in 1786, and pie charts in 1801 [2]. Figure 1 showcases an example of a time series created by Playfair.

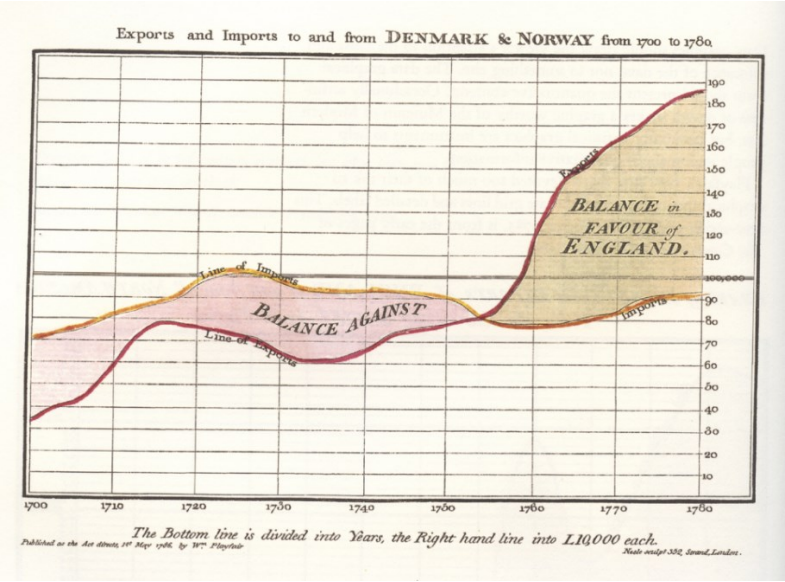



***Figure 1.*** *Playfair’s time series of foreign trade deficit*

Modern digital tools, and especially web-based platforms introduce interactivity and dynamism to the discipline. Such tools make it possible to visualise data that is constantly changing or that is coming directly from the user as an input. New web standards, such as Canvas and SVG, give users extensive freedom in presenting their data while harnessing the full power of JavaScript to achieve this goal.

While JavaScript has full access to the APIs needed to create the most appealing visualisations, doing so directly is unwieldy. To solve this problem, communities have gathered to develop open-source libraries that utilise the flexibility of JavaScript to produce extensive and dynamic data visualisations. These libraries tend to prove superior to “vanilla” JavaScript because they support a data-driven development process. They boast features specifically designed for data visualisation and allow users to quickly create and prototype.

This thesis aims to compare the most popular JavaScript visualisation libraries by analysing their core functionalities, performance, as well as their suitability to different kinds of applications. After going through important concepts and theories in data visualisation, the different visualisation libraries will be presented alongside key examples that will allow practical comparison and analysis.

Data visualisation as a discipline is explored in chapter 2, followed by a presentation of relevant web-based visualisation technologies. Chapter 3 is dedicated to an overview of Chart.js and D3.js as visualisation libraries, while chapter 4 delves into a deeper analysis of both libraries. The final chapter includes a conclusion of the results.

# 2. WEB-BASED DATA VISUALISATION

Data visualisation benefits from all the advantages of the web as a platform. Beyond the numerous browsers on the market, accessing the web generally does not require any other software installation, furthering the outreach and accessibility of web-based visualisations. There are various methods for the implementation of such visualisations, though standards have greatly evolved since the use of Java applets and Flash. This chapter aims to take a brief look into data visualisation as a discipline, then dives into the web standards that are at the base of the creation of visualisations.

## 2.1 Data visualisation

Data visualisation is a broad topic, but the core idea is simple: turning raw data into meaningful visual constructs. Today, large advances in technology means that a lot more data is available and is thus required to be visualised in order to extract information from it.

Numerous fields are served by data visualisation. Journalism, for instance, is essentially about gathering relevant information and disseminating it in an interesting and accessible way [3]. Today, a number of newsrooms are turning to data visualisation, creating visually rich data visualisations to improve understanding of particular stories [4].

From a business standpoint, decisions involve uncertainty and are linked to problems that require quantitative analysis [5]. The right visualisation eases the decision-making by presenting a large amount of data in an easy to work with format. The UK Government, for instance, has a long-running data visualization programme and it has improved the quality of data-driven policy, making decisions more quickly [6].

There are multiple ways the same data can be visualised, which also means that the choice must be conscious and done for the right reasons. A successful historical visualisation is John Snow’s map of the cholera outbreak. Today known as a dot map, Snow’s visualisation shows that the households suffering the most from the outbreak were all using the same well as their water supply.

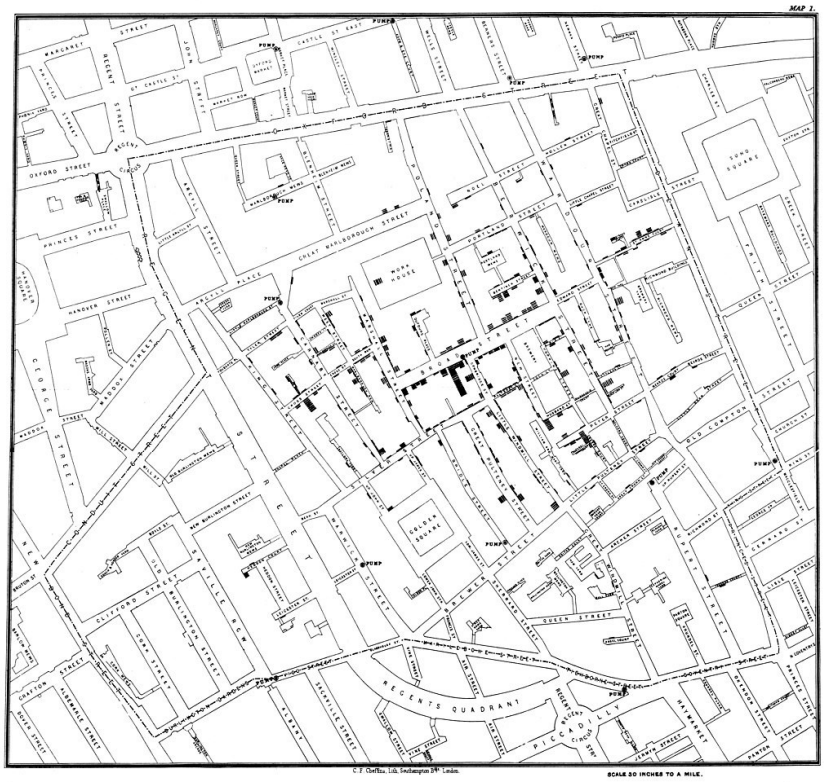



***Figure 2.*** *John Snow's map showing clusters of cholera cases [7]*

When generalised to the entire city of London, a clear link between water sources and cholera outbreaks could be drawn. This allowed Snow to conclude that cholera is water-borne disease, contradicting the then-popular miasma theory.

## 2.2 Web-based visualisation technologies

Various graphics rendering standards are in use in the web today. This thesis looks at the most popular ones, SVG (Scalable Vector Graphics) and Canvas which are the standards used by D3.js and Chart.js respectively.

### 2.2.1 SVG

Scalable Vector Graphics is a data format used to define vector graphics. It is inspired by the formats VML (supported by Microsoft) and PGML (supported by Adobe and Sun). Unlike the previous two, SVG is recommended by the W3C (World Wide Web Consortium).

The format is based on XML and allows the user to define graphical elements for the web [8]. It is mainly used to display graphs and charts but it also supports complex illustrations.

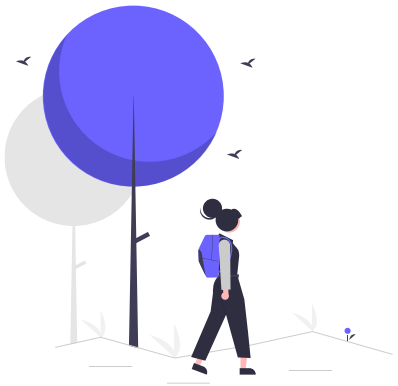


***Figure 3.*** *Example of an illustration rendered with SVG*

SVG can be combined with CSS to define the style of graphical elements, for instance specifying the colour and font of text. SVG graphical elements also support DOM manipulation, which means JavaScript can be used to implement, for instance, interactive graphical elements [9].

The main advantage of using SVG is its vectorial nature – SVG graphical elements can be resized with no loss of quality [9]. Support for basic geometrical shapes such as rectangles and ellipses is included, but more complex shapes can be drawn using paths [10].

### 2.2.2 HTML Canvas

Ever since the arrival of HTML5, web applications have enjoyed access to many new features that make the medium more attractive. One of those new elements is <canvas>. It is an area of pixels initially transparent which may be accessed by JavaScript code to draw various shapes, from simple charts to complex video games.

An advantage of using Canvas is that it is a standard developed by W3C and works seamlessly with HTML and JavaScript. It is also suitable for graphics intensive applications as it has good performance and is hardware accelerated in most browsers. Finally, Canvas enjoys excellent support on mobile platforms.

Since the usage of Canvas is diverse and modular, there is currently no miracle tool or IDE to produce the necessary code without working directly with JavaScript.

As opposed to SVG, Canvas follows a “fire and forget” paradigm where the current state of the graphics is not kept in memory. When making a change to a graphic rendered in Canvas, all of the shapes must be redrawn.

# 3. OVERVIEW OF D3.JS CHART.JS

Web-based visualisation technologies such as SVG and Canvas can be used to present content visually, but then the coding is left to the developer in order to transform data into visualisations. This does not lend itself to creating web applications that are specifically designed for this environment, but the standards constitute the base on which visualisation libraries operate on. These libraries provide a level of abstraction over the interface and abstract the programmer from the implementation details.

Web-based visualisation libraries therefore have a potential advantage over off-the-shelf implementations. They provide a set of features with which programmers can create interactive visualisations on the basis of a standard specification. Such systems also facilitate the construction of Web applications in the context of common Web standards such as HTML, CSS, and JavaScript.

## 3.1 D3.js

D3.js is a JavaScript library meant for visualising digital data in a graphical and dynamic format. It is a powerful tool when it comes to conforming to W3C's standards that uses the common web technologies which are SVG, JavaScript and CSS. D3.js is the official successor of the previous framework Protovis [11]. Unlike other libraries, D3.js allows for a deeper and more extensive control of the final visual result. It has made its appearance in numerous impressive visualisations [12].

The library was developed as response to a growing need for robust and flexible data visualisation that is also web-accessible. D3.js's versatile nature allows it to go beyond simply creating pie charts and line graphs, allowing developers to incorporate complex maps or interactive diagrams without resorting to cumbersome and outdated technologies such as Flash or Java applets.

### 3.1.1 Selection

D3.js uses selection to target the right DOM elements which will be used for creating a visualisation. The library uses the same selection paradigm as CSS3. A selection is made by querying a selector, a string that identifies elements of the DOM. D3.js employs two functions for selecting respectively a single element (*select()*) or a collection of elements (*selectAll()*) corresponding to the selector passed as a parameter [13].

```
1. const selection1 = d3.select("p");
2. const selection2 = d3.selectAll("div");
```

D3.js also supports chaining selections. In the above example, *d3* represents the entire document. It can be substituted with any element of the DOM in order to only select from the descendants of said element.

### 3.1.2 Data binding

Data is directly bound to DOM element in D3.js. Selection objects possess the *data()* method which allows binding an array passed as a parameter to the DOM encapsulated by the selection. This is achieved by adding a *__data__* property to the target element which holds the bound data. If the array passed to the data() method contains objects, a function can be additionally passed as a parameter to the method which will be executed on each element of the array to extract the desired value [13].

Calling the data() method on a selection returns a new object containing 3 sets: update selection, enter selection and exit selection. The sets describe how the data relates to the DOM elements.

```
<html>
    <head>
        <script src="https://d3js.org/d3.v6.min.js"></script>
    </head>
    <body>
        <ul>
            <li></li>
            <li></li>
        </ul>
          <script>
              const elements = [{
                  id: 1,
                  name: 'Iron',
                  symbol: 'Fe'
                  }, {
                  id: 2,
                  name: 'Oxygen',
                  symbol: 'O'
                  }, {
                  id: 3,
                  name: 'Bismuth',
                  symbol: 'Bi'
              }];

              const selection = d3.select('ul')
                  .selectAll('li')
                  .data(elements);
              selection.text(element => element.name)
              console.log(selection)
          </script>
      </body>
  </html>
```

***Figure 4.*** *Selection and data binding with D3.js*

The enter, exit and update sets are stored as properties inside the selection object and can be made available as parameters of a function. Figure 4 demonstrates a selection of a list of 2 items, to which an array of 3 items is bound. The properties of the selection object are printed out in the browser's console using console.log and can be seen in figure 5.


***Figure 5.*** *Properties of a D3.js selection*

The sets, or properties, represent the DOM elements that can or can't be matched with data:

- Update selection: this is the base selection and it is held in the _groups property. It contains the DOM elements that can be updated with a value. In the example, the original selection contains 2 list elements and is bound to an array of size 3, therefore the list elements are bound to the first two elements in the array, Iron and Oxygen.
- Enter selection: this is the selection that contains the DOM elements that should be created to bind any unbound data and it is held in the _enter property. In the example, a new list element must be created to which the last element of the array will be bound.
- Exit selection: this is the selection that contains existing DOM elements to which no data is currently bound. In the example, there are no DOM elements in this selection because there is more data than there are DOM elements.

The update selection can be processed by calling methods directly on the base selection while the enter and exit selection can be processed by calling the methods *enter()* and *exit()* and then chaining with the relevant method.

Multiple methods are supported by selection objects, some of which can act on every single DOM element encapsulated by the selection. Since the iterating is done over the elements included in the *_group* property, it is crucial to adopt an update pattern that considers changing data. It must be possible to create new DOM elements to accommodate extra data while getting rid of superfluous DOM elements if the data size shrinks.

As can be seen in line 101 of appendix A, one way of achieving this pattern is simply using the *join()* method which takes care of creating and removing elements depending on the size of the data.

### 3.1.3 SVG

D3.js does not require using SVG, but the standard is a good base for creating graphics with D3.js since it sits directly on the DOM. SVG elements behave as objects whose attributes can be specified as properties.

An SVG element can either be injected in the DOM with D3.js using the *append()* method on a selection, or it can be directly created in the HTML code. The *attr()* method is used to set the height and width of the SVG element.

```
1. const selection = d3.select('#svg_container')
2.                     .append("svg")
3.                     .attr("width", width).attr("height", height)
4.
```

In appendix A, a bar chart is prepared by creating an HTML that will act as a container for the SVG, which will be added with D3.js. SVG shapes can then be added following the enter-update-exit pattern using the *join()* method, to which the data will be bound using the *data()* method.

### 3.1.4 Scales

Creating a visualisation comes with the challenge of fitting the data in the medium you wish to present it in — screens in our case. A good visualisation must be able to take in datasets of varying sizes and fit them in the same amount of pixels.

In appendix A, the chart is created with bars whose heights are correlated with atomic mass  in an area of size 800x500 pixels. Two important concepts are to be considered here:

- The dataset's domain: this denotes the minimum and maximum values in our data. In the example, hydrogen has the lowest atomic mass 1.008 u, zirconium has the highest one at 91.2242 u. The domain is therefore [1.008, 91.2242].

- The dataset's range: this represents the values to which the data is mapped. In the example the smallest bar must be 5 pixels high, therefore the range starts at 0 and ends at the height allocated to the bars minus five. In line 52 of the appendix, the order is reversed so that the bars are drawn from bottom to top rather than top to bottom.

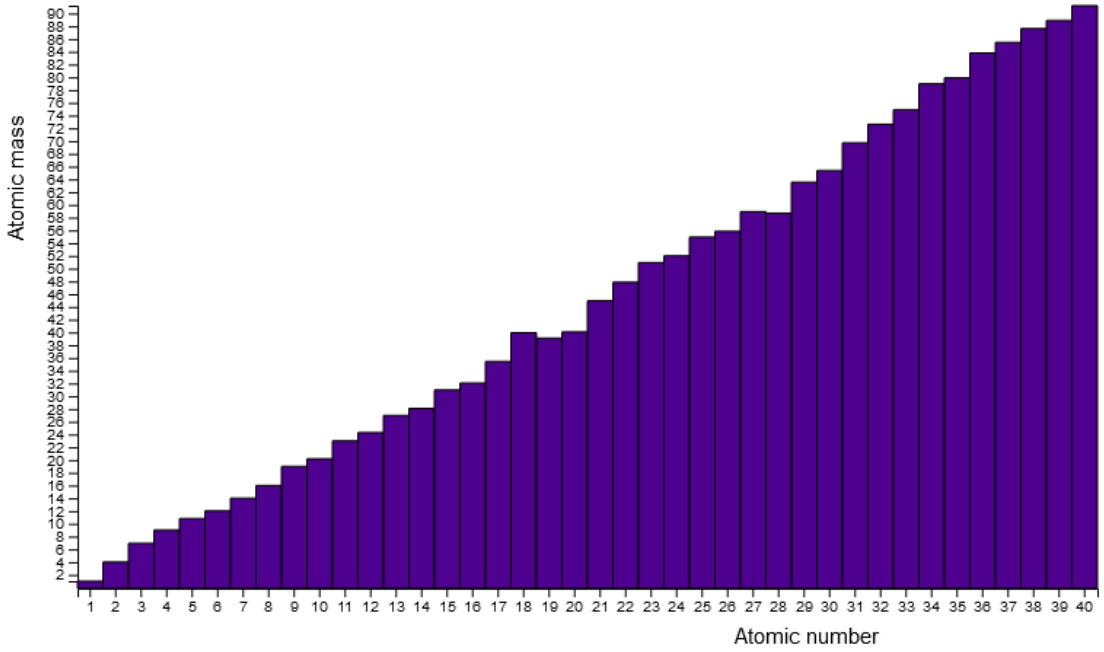
D3.js provides multiple methods that are responsible for scaling the mapping of our values [14]. Those methods take as arguments the dataset's domain, range, and return a function that will map the data so that it all fits in the frame. The library also offers several kinds of scales that fit different uses cases, such as continuous, sequential, quantile or ordinal. For our example, we used the simplest continuous scale: the linear scale.

```
1. let y_scale = d3.scaleLinear()
2.                     .domain([d3.min(elements,
3.                              e => e.atomic_mass),
4.                              d3.max(elements,
5.                              e => e.atomic_mass)])
6.                     .range([effective_height - 5, 0])
7.
```

Since we want to keep our dataset flexible, we programmatically calculate the dataset's domain using the D3.js methods *min()* and *max()* that can be used with accessors to retrieve a specific property from the objects contained in the data array [15]. As can be seen in lines 104 and 106, the resulting function is used to calculate the height and position of each bar with the guarantee that it will not go beyond the dedicated frame.

### 3.1.5 Axes

Charts generally require axes against which the data can be read. They are important for the reader to be able to interpret the visual elements. D3.js provides methods to generate and add the axes to our chart, such as axisBottom() or axisLeft(), depending on where the axis should be positioned [16].




The method *axisBottom()* is used to generate a bottom-positioned axis. The method takes as argument the scale which the axis will visualise. In line 75 of appendix A, the function is passed as a parameter to the *call()* method which is called on top of an SVG element, the g element in the example, which serves as a container for SVG elements. The y-axis is implemented in a similar way by using the *axisLeft()* method in line 88. The *tick()* method is used to specify how many tick lines are to be drawn on the axis. The argument is dynamically calculated depending on how much vertical space is available. Labels for the axes are added using the *append()* method. The position of the text is available as properties and can be changed using the *attr()* method.

### 3.1.6 Interactivity

A major benefit web-based visualisations have over other media is user interaction. D3.js offers this by taking advantage of mouse events supported by browsers [17]. Common events include hovering the mouse over an element (onmouseover) or clicking (onclick).

D3.js provides the method on() for selection objects which takes as a parameter the event type and a call back function. In D3 v6.0, the call back function has access to the current event and the data bound to the element on which the event happens [18]. In the appendix A example, an event listener for the mouseover event is attached to each bar of the bar chart.

```
1. .on("mouseover", (event, element) => {
2.                 tooltip.style("visibility", "visible")
```

```
3.                 .html(element.name
4.                        + "</br> Atomic mass: "
5.                        + element.atomic_mass
6.                        + "</br> Atomic number: " + element.number)
7.                 .style("top", event.y-100+"px")
8.                 .style("left", x_scale(element.number)+"px")
9.                 d3.select(
10.                   event.currentTarget).style("fill", "#8b62ac")
11.                             })
12.
```

The call back is responsible for styling and repositioning the tooltip, a previously created HTML element that will hold information on the hovered bar. CSS is used to make it invisible until a relevant event is triggered. The position can be calculated using information extracted from the bound data as well as the position at which the event was triggered, which is provided by the *y* property of the *event* object. The event object also makes it possible to select the element on which the event was triggered, making it possible to restyle it. The mouseout event is used to hide away the tooltip and revert the style changes done on the hovered bar.

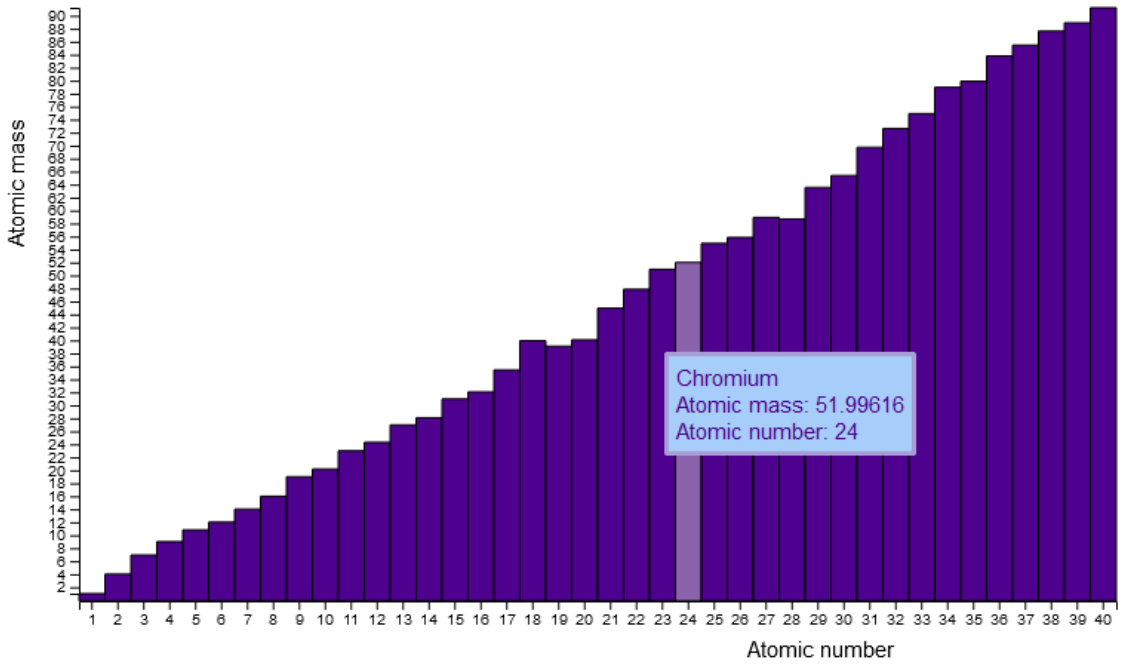



***Figure 6.*** *Hover interaction in a bar graph created with D3.js*

The end result shown in figure 6 is a bar chart with which the user can interact using the mouse. The simple interaction allows the user to access in-depth information about the visualised data without overcrowding the area dedicated to the chart.

### 3.1.7 3D

D3.js does not offer 3D visualisation support out of the box beyond using pure JavaScript and SVG. That being said, there are a number of tools that can be paired with D3.js to achieve 3D capabilities.

One such tool is Extensive 3D (X3D), a fairly new attempt at standardising 3D rendering in HTML. It is maintained by the Web3D Consortium [19]. The standard is cross platform and as long as it is supported by the browser, does not require the installation of additional applications to render 3D models. An example of a 3D visualisation created with D3.js and X3D can be seen in figure 7.

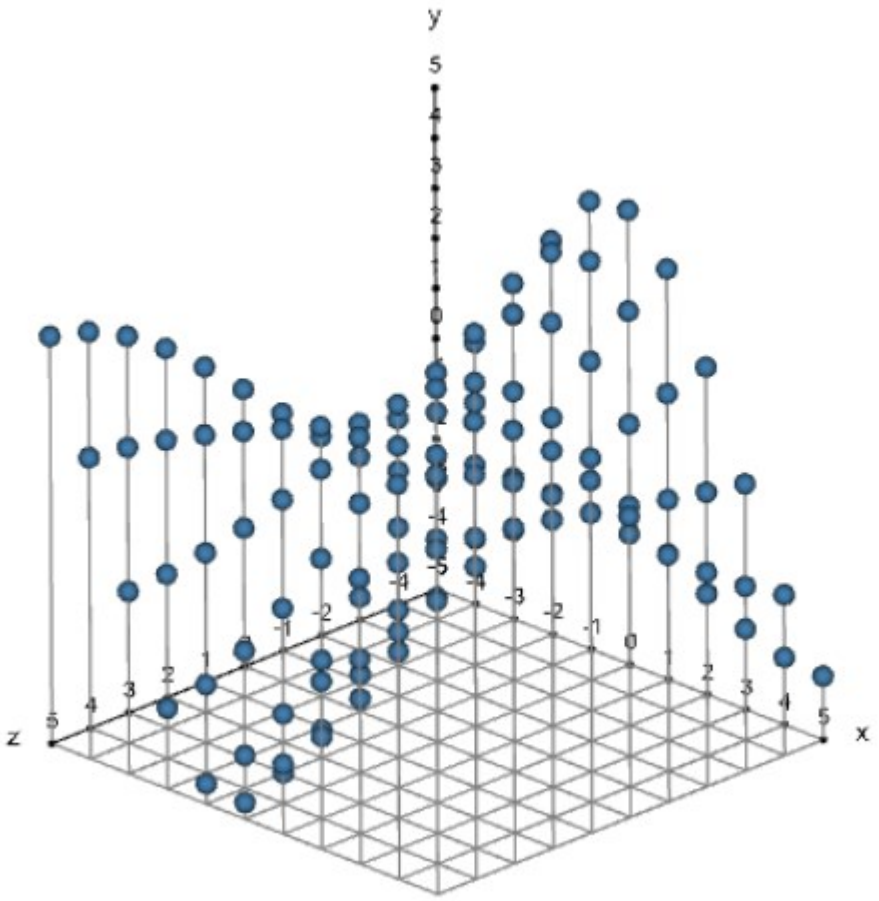



***Figure 7.*** *Example of scatter plot with X3D with D3.js [20]*

Alternatively, it is also possible to use the D3.js plugin d3-3d which is meant for 3D visualisation [21]. The plugin uses the browser's coordinate system and orthographic projection to visualise data in 3D. Figure 8 showcases an example of a visualisation created with the plugin.

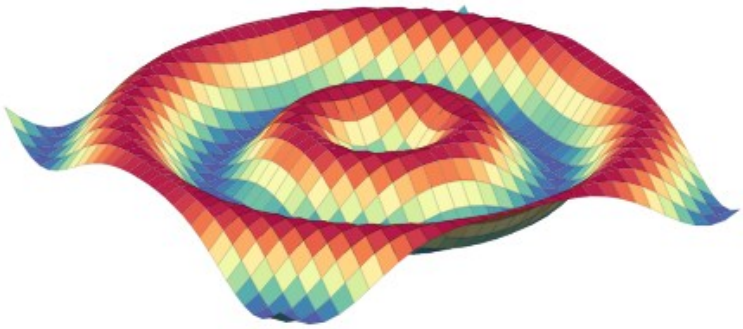


***Figure 8.*** *Example of 3D surface plot with d3-3d [22]*

## 3.2 Chart.js

Chart.js is a free open-source JavaScript library used for data visualisation. The library uses HTML5 canvas to render different types of charts and is one of the most popular JavaScript charting libraries on Github.

The allure of Chart.js comes from its simple usage. Basic knowledge of HTML, CSS and JavaScript allows one to quickly get charts up and running with it. Chart.js has its own cheat sheet for beginners. It is written in plain English, which makes the steps for adding data and arranging the layout as clear as possible [23]. The library pays the price for this by being relatively limited in scope and features. It currently supports eight different types of charts: bar, line, area, radar, polar area, scatter, bubble, pie.

Creating a visualisation with Chart.js follows 3 simple steps:

1. Setting up the canvas element in which the chart will be rendered
2. Configuring the options of the chart
3. Supplying the chart with the data to be visualised

Little programming knowledge is required, though JavaScript can be used to extend some functionalities.

### 3.2.1 Configuration

The configuration is a collection of properties which dictate the behaviour of a chart. These include the data to be visualised, fonts, styles, tooltips, and so on [24]. The most barebone configuration specifies the type of the chart and the data. The remaining options are simply set to the default values included in the library.

```
1. const config = {
2.             type: 'bar',
3.             data,
4.             options: {}
5.           };
```

In lines 47-57 of appendix B, the options property is further to expanded to include custom settings for the tooltips since the default behaviour is not suitable. Options can be configured for the entire chart, a specific dataset or a specific chart type.

### 3.2.2 Data

The data property of the configuration contains the datasets to be visualised and their labels. The style of a dataset can also already be configured at this point. Chart.js automatically matches the data with the label of the same index. If the data provided consists of objects, the data to be parsed is specified with the *parsing* property.

```
1. data: elements,
2. parsing: {
3.     xAxisKey: 'number',
4.     yAxisKey: 'atomic_mass'
5. }
```

In this snippet corresponding to lines 36-40 of appendix B, *elements* is an array with objects describing chemical elements. This configuration maps the atomic numbers in the x-axis against atomic masses in the y-axis.

### 3.2.3 Interactivity

Graphics drawn in canvas do not inherently support event handlers [25]. A common way of implementing them is to retrieve the cursor's position and use it to mathematically work out which element it is pointing to. Chart.js does that behind the scenes and provides helpers to find the data coordinates on which an event occurred [26].

Chart.js also provides interactions out of the box as a default options for all chart types. This includes clicking on the legend to toggle a dataset's visibility on and off and data points tooltips.

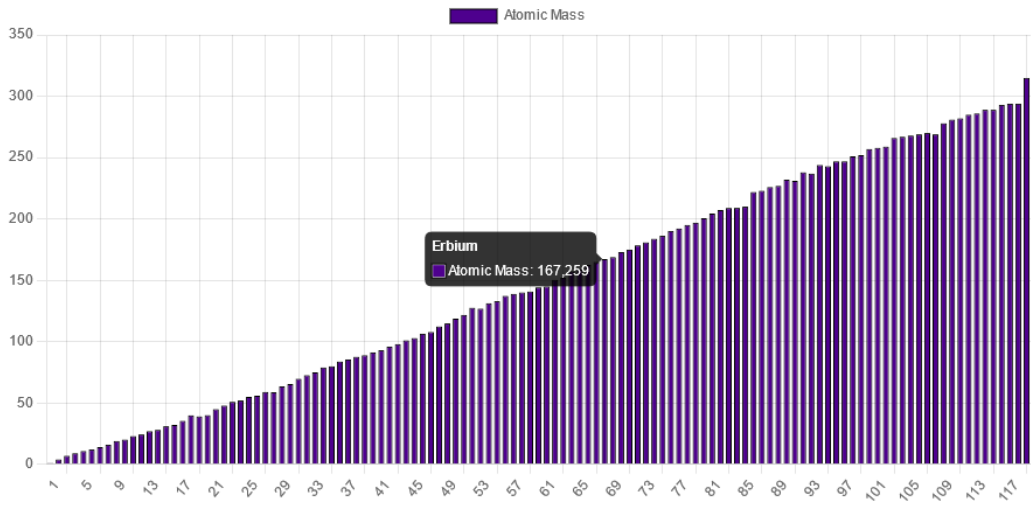



***Figure 9.*** *Hover interaction in a bar graph created with Chart.js*

Tooltips are triggered by hover and include the selected data point by default as it can be seen in figure 9. The functionality can be further customised by configuring callback functions [27]. For instance, it is possible to change the label text in the tooltip by using the *label* callback. In lines 50-53 of appendix B, the tooltip is customised by using the *title* callback. The function changes the title of the tooltip to reflect the name of the element corresponding to the hovered bar as seen in figure 8.

### 3.2.4 Axes and scales

Chart.js includes appropriate axes with all the charts it supports by default. The programmer does not need to worry about configuring them for basic charts. The library also supports multiple X and Y axes [28].

Chart.js supports multiple scale types which can be set in the chart's configuration:

- **Cartesian axes:** linear, logarithmic, category, time, timeseries [29].
- **Radial:** linear radial axis [30].

The user may also define minimum and maximum values for the scale, among many other options for customisation [28].

# 4. ANALYSIS AND COMPARISON OF CHART.JS AND D3.JS

By comparing the two libraries, this thesis sets out to answer the following questions:

1. What are the main differences between Chart.js and D3.js?
2. What are the practical ramifications of these differences?

Different types of data were chosen to conduct the comparison. Nowadays, users expect snappy websites that serve their content with limited delay. Therefore, it was deemed important to compare how well each library performs with datasets of varying sizes. Another point of interest is the offering of each library in terms of features. A library would naturally not be suitable for a certain use case if it cannot offer the required functionality. Finally, as development time is an important component of project management, especially in Agile teams, the ease of use of each library will also be considered.

## 4.1 Performance

The performance tests for Chart.js and D3.js were run by implementing line charts with datasets of increasing size. The performance was measured using browser tools and JavaScript.

### 4.1.1 Generating the data

The data was generated randomly using a Python script to produce 6 different datasets with the following sizes: 10, 100, 1000, 10 000, 100 000, 1 000 000. The data consists of random numbers between 0 and 100 associated with integers from 0 to X-1, where X is the size of the dataset. To avoid large jumps between consecutive data points, each data point is within a 0 to 10 distance from the previous one.


***Figure 10.*** *Example of a D3.js line chart*

The data was stored in JSON format within separate files to ensure that all the tests are run with the same datasets. Figure 10 is an example of a line chart created with D3.js with the data generated for performance tests.

### 4.1.2 Code implementation

The tests were run on line charts inspired from online examples [31] [32] [33]. The charts were implemented with a sober design and only the most basic chart features were kept. Since Chart.js provides default options, it was judged that running tests with the default setup would produce the most reasonable results. Therefore, efforts were made to make the D3.js line charts look the most like the Chart.js ones. All of the charts include x and y axes and a grid.

Since D3.js uses SVG, two different implementations were used. The first one includes only a single line represented by a single SVG element. The second adds a visible circle for each data point throughout the line, each one represented by a SVG element. This means that while in the first implementation there is only one SVG element that represents the data, in the second one there are as many SVG elements as there are data points. This distinction was omitted for Chart.js since it uses Canvas, as the visibility of the individual data points made no performance difference.

### 4.1.3 Test methodology

Mozilla Firefox was chosen as the browser on which the tests are run due to its robust developer tools [34]. To ensure equal conditions, the CPU and RAM loads were kept low.

Three different data types were measured:

- Load time: represents the time delta between the moment the load event handler returns and the moment the navigation starts. This measurement was carried out using the Navigation Timing API [35].
- Execution time: represents the time it takes for the JavasScript code to execute. This does not take in consideration the part of the code that loads the JSON data since that is unrelated to the libraries. This measurement was carried out using the web console's *time()* method [36].
- Memory: represents the memory heap of the tab in which the chart is opened. This measurement is provided by the Firefox Developer Tools [37].

Each measurement was taken 10 times and the mean was computed.

### 4.1.4 Results

All the taken measurements can be found in appendix C.

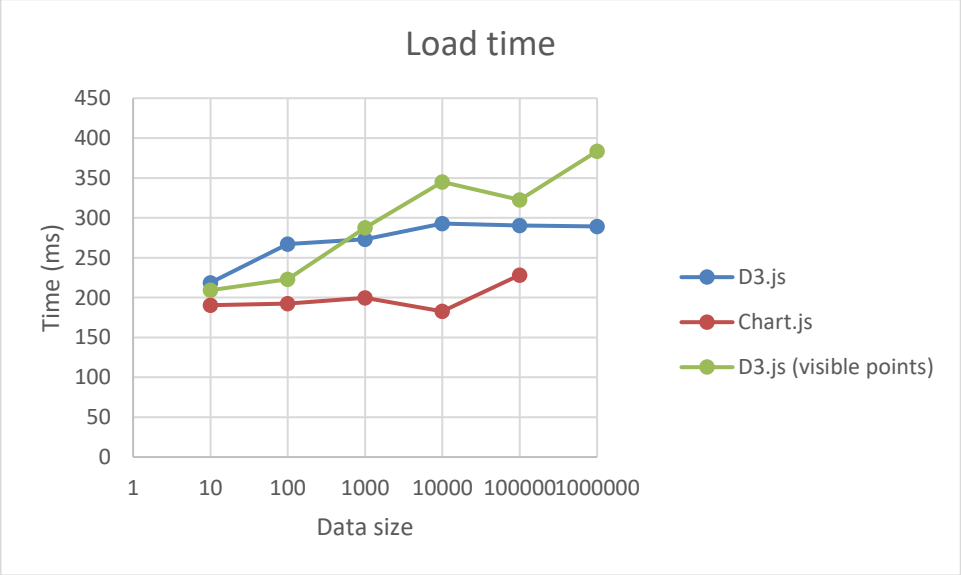



***Figure 11.*** *Load time*

In terms of the loading time, there was no discernible difference between the different implementations as shown in figure 11. This is likely not a great way of measuring the performance as the rendering continues after the page finishes loading.

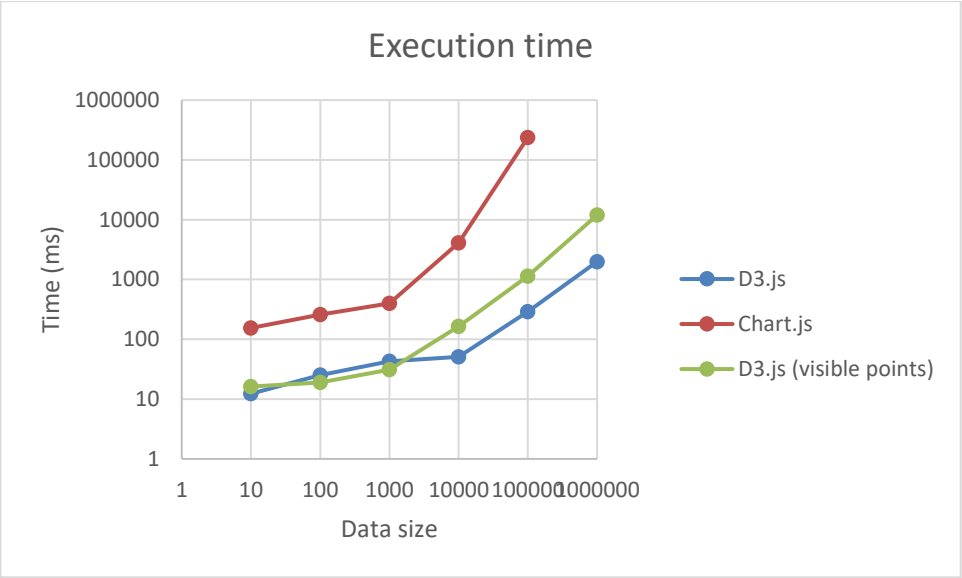



***Figure 12.*** *Execution time*

Figure 12 shows a massive difference in execution times. Chart.js is heavily outperformed by D3.js, with visible points or otherwise. Chart.js was entirely unable to render a million data points as the browser tab crashed after a moment.

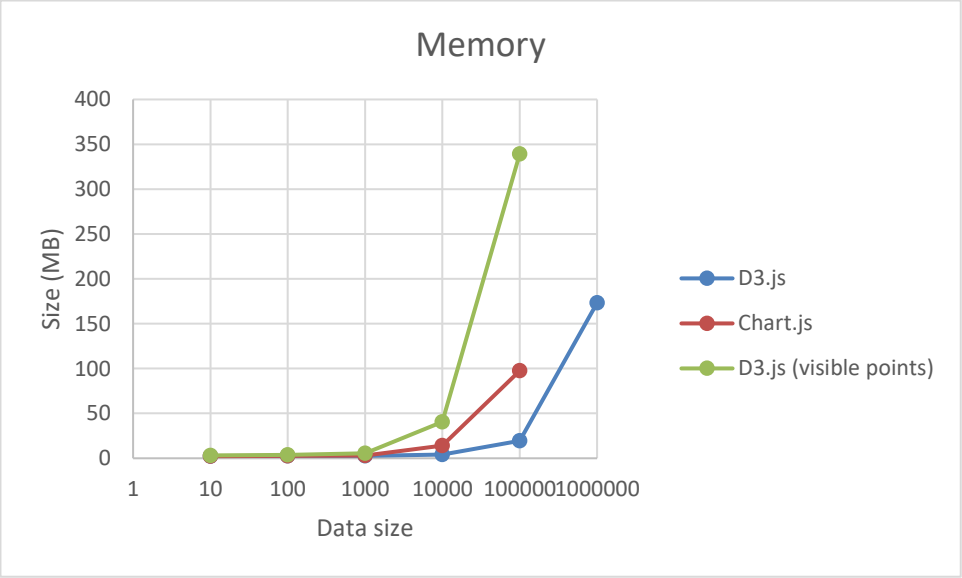



***Figure 13.*** *Memory usage*

Significant memory performance differences are shown in figure 13. D3.js outperforms Chart.js when only rendering a line, but its performance rapidly breaks down if it also has the render circles for each data point. The memory heap at one million data points was so massive (estimated at over 2 GB using Windows' Task Manager) that Firefox Developer Tools were unable to measure it.

### 4.1.5 Discussion

There are significant differences in performance between the two libraries. When it comes to execution time, it is surprising that Chart.js doesn't perform as well as D3.js given that Canvas typically has better rendering performance than SVG [38]. It is unclear why D3.js performs better than Chart.js in this aspect, but it can be theorised that D3.js' development involves more optimisation given its popularity compared to Chart.js.

When using D3.js, rendering a single line was unsurprisingly faster than rendering data point circles. Creating a new SVG element for each new data point adds significant overhead compared to only a single SVG element for all the data.

In terms of memory, Chart.js is again surprisingly lacking compared to D3.js. It can be theorised that Chart.js by default keeps track of every single data point for interactivity purposes, whereas D3.js does not do that if it is not asked to. This theory is reinforced by the fact that D3.js' memory usage was exceedingly poor when rendering circles for data points.

Both libraries performed well with relatively small datasets. Neither libraries are particularly suitable for rendering extremely large data points with SVG or Canvas. D3.js especially so given the massive memory usage with SVG. An alternative would be to make use of the GPU. D3.js can be paired with WebGL to achieve that [39]. Chart.js is entirely unsuitable for this use case given that it only supports Canvas.

It must be kept in mind that an important difference between SVG and Canvas is what happens when the scene must be changed. Since elements are kept track of with the DOM in SVG, individual elements can be changed without affecting the scene. With Canvas, the entire scene is redrawn when there is a change [25]. This has important performance ramifications for highly interactive visualisations, but this thesis does not explore that side for the performance comparison.

## 4.2 Features

Both Chart.js and D3.js feature the basic components that make up a chart – axes, scales, basic shapes. They also support common data formats such as CSV and JSON.

Chart.js boasts a massive amount of customisable settings and scriptable options using callback function, but features can also be extended so much before hitting a wall. The library for instance has no native support for 3D and unlike D3.js, cannot be extended to accommodate it.

Additionally, Chart.js only supports 8 different chart types. The library is only meant for visualising data in the form of charts. D3.js, on the other hand, is only restricted by the developer's imagination. The library can be used for simple charts, but may also visualise data in other formats. For instance, chord diagrams [40] or a map [41].

The organisation *Observable* features on their website a repository of impressive visualisations created with D3.js, the vast majority of which cannot be replicated with Chart.js. [42]

## 4.3 Ease of use

Development time is an important factor to consider when shipping a project, which can be directly correlated to the ease of use of a library. After analysing Chart.js and D3.js, a subjective assessment of this metric can be made.

Chart.js is a powerful choice for saving time and effort. All that is needed to create a basic chart is to copy the basic configuration for the desired chart type from the documentation, bind the data and then adjust the settings as needed. Minimal coding is required since the library handles that behind the scenes. The end product is elegantly presented with colours, animations, legends, and hover interactivity. Since the library is primarily configuration based, making use of all the features is as easy as following the documentation while editing the configuration.

On the other hand, D3.js has a much steeper learning curve and requires developer to invest time and effort into each desired feature. The library supports the basic components of a visualisations, but they must be programmatically assembled by the developer. The onus is on the developer to make sure the components behave nicely with each other. Method chaining and the use of callback functions is the main pattern a programmer must follow.

## 4.4 Use cases

After careful analysis of the differences between Chart.js and D3.js, it stands out each library is suitable for different use cases. When considering performance, D3.js stands out as being more suitable for visualisations with large datasets. That might be a counterintuitive conclusion since SVG is considered less performant than Canvas at higher object count, but the data shows D3.js makes up for it by being more optimised than Chart.js. Additionally, as discussed previously, the SVG element count of certain visualisation types such as line charts does not increase with the dataset size. For visualisations with smaller datasets, there is no clear-cut choice: both libraries are suitable.

Comparatively, it is much easier to decide which library to pick if the application requirements are precisely known beforehand. If the data is best visualised in a format that is not supported by Chart.js, then the developer is forced to turn to D3.js. There is little compromise in this regard as Chart.js has no extensibility options when it comes to chart types beyond merely combining different ones.

# 5. CONCLUSION

As shown by the overview, both D3.js and Chart.js provide all the functionalities needed to create high quality charts for the web. D3.js pushes the boundaries of web rendering by providing the tools to create more impressive visualisations while Chart.js only provides the tools to create 8 different types of charts.

Performance wise, D3.js renders charts faster than Chart.js and uses less memory, though at high object count D3.js uses far more memory than Chart.js. This possibly hints at better optimisation on the side of D3.js despite its usage of SVG, as opposed to Chart.js with uses Canvas, a more performant standard.

This thesis only looked at D3.js when used with SVG. Since the library can also be paired with Canvas, future research may look into D3.js's performance when used with the standard.

# 6. REFERENCES


1. Nathalie Henry Riche CHNDkSC. Data-driven storytelling. First edition ed.: CRC Press LLC; 2018.
2. Michael Friendly DJD. Milestones in the history of thematic cartography, statistica l graphics, and data visualization. 2001 January;: 71.
3. Bill Kovach TR. The Elements of Journalism: What Newspeople Should Know and the Public Should Expect. revised ed.: Three Rivers Press; 2007.
4. Helen Kennedy WARLHMEAKWW. Data Visualisations: Newsroom Trends and… | DataJournalism.com. [Online]. [cited 2021 April 18. Available from: https://datajournalism.com/read/handbook/two/working-with-data/experiencing-data/data-visualisations-newsroom-trends-and-everyday-engagements.
5. Lloyd CJ. Data-Driven Business Decisions. 1st ed.: Wiley; 2011.
6. Government U. National Data Strategy - GOV.UK. [Online].; 2020 [cited 2021 April 18. Available from: https://www.gov.uk/government/publications/uk-national-data-strategy/national-data-strategy.
7. Snow J. On the Mode of Communication of Cholera London: C.F.; 1854.
8. Introduction — SVG 2. [Online]. [cited 2021 6 April. Available from: https://www.w3.org/TR/SVG2/intro.html.
9. SVG: Scalable Vector Graphics | MDN. [Online].; 2021 [cited 2021 April 17. Available from: https://developer.mozilla.org/en-US/docs/Web/SVG.
10. Basic shapes - SVG: Scalable Vector Graphics | MDN. [Online].; 2021 [cited 2021 April 17. Available from: https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Basic_Shapes.
11. Bostock M. Protovis. [Online].; 2011 [cited 2021 April 26. Available from: https://mbostock.github.io/protovis/.
12. Techslides. Over 1000 D3.js Examples and Demos | TechSlides. [Online].; 2013 [cited 2021 April 6. Available from: http://techslides.com/over-1000-d3-js-examples-and-demos.
13. GitHub - d3/d3-selection: Transform the DOM by selecting elements and joining to data. [Online].; 2021 [cited 2021 April 17. Available from: https://github.com/d3/d3-selection.
14. GitHub - d3/d3-scale: Encodings that map abstract data to visual representation. [Online].; 2020 [cited 2021 April 7. Available from: https://github.com/d3/d3-scale.
15. d3-array/README.md at master · d3/d3-array · GitHub. [Online].; 2021 [cited 2021 April 7. Available from: https://github.com/d3/d3-array/blob/master/README.md#min.
16. GitHub - d3/d3-axis: Human-readable reference marks for scales. [Online].; 2021 [cited 2021 April 17. Available from: https://github.com/d3/d3-axis.
17. GitHub. [Online].; 2021 [cited 2021 April 7. Available from: https://github.com/d3/d3-selection#handling-events.
18. Fil. D3 6.0 migration guide / D3 / Observable. [Online].; 2020 [cited 2021 April 7. Available from: https://observablehq.com/@d3/d3v6-migration-guide.
19. What is X3D? | Web3D Consortium. [Online]. [cited 2021 April 14. Available from: https://www.web3d.org/x3d/what-x3d.
20. Voorhees H. 3D scatter plot using d3, x3dom - bl.ocks.org. [Online].; 2020 [cited 2021 April 14. Available from: http://bl.ocks.org/hlvoorhees/5986172.
21. Nieke S. GitHub - Niekes/d3-3d: D3.js plugin for 3d visualization. [Online].; 2021 [cited 2021 April 14. Available from: https://github.com/Niekes/d3-3d.
22. Nieke S. 3D Surface Plot in D3.js with d3-3d - bl.ocks.org. [Online].; 2021 [cited 2021 April 14. Available from: https://bl.ocks.org/Niekes/e920c03edd7950578b8a6cded8b5a1a5.

23. Chart.js | Chart.js. [Online].; 2021 [cited 2021 April 17. Available from: https://www.chartjs.org/docs/latest/.
24. Configuration | Chart.js. [Online].; 2021 [cited 2021 April 17. Available from: https://www.chartjs.org/docs/latest/configuration/.
25. Appel R. Web Dev Report - Working with Graphics on the Web: Canvas vs. SVG | Microsoft Docs. [Online].; 2012 [cited 2021 April 17. Available from: https://docs.microsoft.com/en-us/archive/msdn-magazine/2012/june/web-dev-report-working-with-graphics-on-the-web-canvas-vs-svg.
26. Interactions | Chart.js. [Online].; 2021 [cited 2021 April 17. Available from: https://www.chartjs.org/docs/latest/configuration/interactions.html.
27. Tooltip | Chart.js. [Online].; 2021 [cited 2021 April 17. Available from: https://www.chartjs.org/docs/latest/configuration/tooltip.html.
28. Axes | Chart.js. [Online].; 2021 [cited 2021 April 17. Available from: https://www.chartjs.org/docs/latest/axes/.
29. Cartesian Axes | Chart.js. [Online].; 2021 [cited 2021 April 17. Available from: https://www.chartjs.org/docs/latest/axes/cartesian/.
30. Linear Radial Axis | Chart.js. [Online].; 2021 [cited 2021 April 17. Available from: https://www.chartjs.org/docs/latest/axes/radial/linear.html.
31. Line Chart | Chart.js. [Online].; 2021 [cited 2021 April 17. Available from: https://www.chartjs.org/docs/latest/charts/line.html.
32. d3noob. Simple graph with grid lines in v6 - bl.ocks.org. [Online].; 2020 [cited 2021 April 17. Available from: https://bl.ocks.org/d3noob/566424623105398bc614f3cd89f87259.
33. d3noob. Scatterplot with v6 - bl.ocks.org. [Online].; 2020 [cited 2021 April 17. Available from: https://bl.ocks.org/d3noob/5680dd0089abdc5b15f188d5efe48852.
34. Firefox Developer Tools | MDN. [Online].; 2021 [cited 2021 April 17. Available from: https://developer.mozilla.org/en-US/docs/Tools.
35. Navigation Timing API - Web APIs | MDN. [Online].; 2020 [cited 2021 April 17. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Navigation_timing_API.
36. Console.time() - Web APIs | MDN. [Online].; 2021 [cited 2021 April 17. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Console/time.
37. Memory - Firefox Developer Tools | MDN. [Online].; 2021 [cited 2021 April 17. Available from: https://developer.mozilla.org/en-US/docs/Tools/Memory.
38. Smus B. Performance of canvas versus SVG | Boris Smus. [Online].; 2009 [cited 2021 April 17. Available from: https://smus.com/canvas-vs-svg-performance/.
39. Eberhardt C. Rendering One Million Datapoints with D3 and WebGL. [Online].; 2020 [cited 2021 April 16. Available from: https://blog.scottlogic.com/2020/05/01/rendering-one-million-points-with-d3.html.
40. Bostock M. Directed Chord Diagram / D3 / Observable. [Online].; 2020 [cited 2021 April 17. Available from: https://observablehq.com/@d3/directed-chord-diagram?collection=@d3/d3-chord.
41. Bostock M. Hexbin Map / D3 / Observable. [Online].; 2019 [cited 2021 April 17. Available from: https://observablehq.com/@d3/hexbin-map.
42. D3 / Observable. [Online]. [cited 2021 April 18. Available from: https://observablehq.com/@d3?tab=notebooks&type=public.

# 7. APPENDICES

## 7.1 Appendix A: bar chart created with D3.js

```
<!doctype html>
<html>

<head>
    <style>
        .bar {
            fill: #4e008e;
            stroke: black;
            stroke-width: 1;
              stroke-linejoin: round;
         }

         #tooltip {
             position: absolute;
             font-family: Open Sans, sans-serif;
             font-size: 15px;
             z-index: 0;
             background-color: #A7CDFA;
             color: #4e008e;
             border: solid;
             border-color: #A89ED6;
             padding: 5px;
             border-radius: 2px;
             visibility: hidden;
         }
    </style>
    <script src="https://d3js.org/d3.v6.min.js"></script>
</head>

<body>
    <div id="tooltip"></div>
    <div id="svg_container"></div>

    <script>
        fetch("./PeriodicTableJSON.json")
            .then(response => {
                return response.json();
            })
            .then(data => {
                const elements = data.elements.slice(0, 40);

                const margin = 50

                let width = 800;
                let height = 500;

                effective_height = height - margin * 2;
                effective_width = width - margin * 2;

                let y_scale = d3.scaleLinear()
```

```
51.                     .domain([d3.min(ele-
   ments, e => e.atomic_mass), d3.max(ele-
   ments, e => e.atomic_mass)])
52.                     .range([effective_height - 5, 0])
53.
54.                 let y_axis_scale = d3.scaleLinear()
55.                     .domain([d3.max(ele-
   ments, e => e.atomic_mass), d3.min(ele-
   ments, e => e.atomic_mass)])
56.                     .range([5, height - margin * 2])
57.
58.                 atomic_numbers = []
59.                 elements.forEach(e => {
60.                     atomic_numbers.push(e.number)
61.                 });
62.
63.                 let x_scale = d3.scaleBand()
64.                     .domain(atomic_numbers)
65.                     .range([margin, width - margin]);
66.
67.                 let bar_width = (width - margin * 2) / ele-
   ments.length;
68.
69.                 const selection = d3.select('#svg_container')
70.                     .ap-
   pend("svg").attr("width", width).attr("height", height)
71.
72.                 selection.append('g')
73.                     .call(d3.axisBottom(x_scale))
74.                     .attr('transform', `trans-
   late(0,${height - margin})`)
75.                     .append("text")
76.                     .attr("transform", "scale(1.5)")
77.                     .attr("text-anchor", "end")
78.                     .attr("x", width / 2)
79.                     .attr("y", margin / 2)
80.                     .style("fill", "black")
81.                     .text("Atomic number");
82.
83.                 let tooltip = d3.select("#tooltip");
84.
85.                 selection.append('g')
86.                     .call(d3.axisLeft(y_scale)
87.                         .ticks(height / 12))
88.                     .attr('transform', `translate(${mar-
   gin},${margin})`)
89.                     .append("text")
90.                     .attr("transform", "rotate(-
   90) scale(1.5)")
91.                     .attr("y", -margin / 2)
92.                     .attr("x", -margin)
93.                     .attr("text-anchor", "end")
94.                     .style("fill", "black")
95.                     .text("Atomic mass");
96.
97.                 selection.selectAll('.bar')
98.                     .data(elements)
99.                     .join('rect')
100.                    .attr('class', 'bar')
101.                    .attr('x', (element, i) => x_scale(ele-
   ment.number))
```

```
                    .attr('y', element => y_scale(ele-
   ment.atomic_mass) + margin)
                    .attr('width', bar_width)
                    .attr('height', element => effec-
   tive_height - y_scale(element.atomic_mass))
                    .on("mouseover", (event, element) => {
                        tooltip.style("visibility", "visible")
                            .html(element.name
                                + "</br> Atomic mass: " + ele-
   ment.atomic_mass
                                + "</br> Atomic number: " + el-
   ement.number)
                            .style("top", event.y - 100 + "px")
                            .style("left", x_scale(element.num-
   ber) + "px")
                        d3.select(event.currentTar-
   get).style("fill", "#8b62ac")
                    })
                    .on("mouseout", (event, element) => {
                        tooltip.style("visibility", "hidden");
                        d3.select(event.currentTar-
   get).style("fill", null)
                    });

            });
    </script>
</body>

</html>

```

The file PeriodicTableJSON.json can be found at https://github.com/Bowserinator/Periodic-Table-JSON.

## 7.2 Appendix B: bar chart created with Chart.js

```
<!doctype html>
<meta charset="UTF-8">
<html>

<head>
  <style>
  </style>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
</head>

 <body>
   <div>
     <canvas id="myChart"></canvas>
   </div>

   <script>
     fetch("./PeriodicTableJSON.json")
       .then(response => {
         return response.json();
       })
       .then(json_data => {
         const elements = json_data.elements;
```

```
23.
24.          atomic_numbers = [];
25.          elements.forEach(e => {
26.            atomic_numbers.push(e.number)
27.          });
28.
29.          const data = {
30.            labels: atomic_numbers,
31.            datasets: [{
32.              label: 'Atomic Mass',
33.              backgroundColor: '#4e008e',
34.              borderColor: 'black',
35.              borderWidth: 1,
36.              data: elements,
37.              parsing: {
38.                xAxisKey: 'number',
39.                yAxisKey: 'atomic_mass'
40.              }
41.            }]
42.          };
43.
44.          function get_tooltip(context) {
45.            const element = context[0].raw;
46.            return element.name;
47.          }
48.          const config = {
49.            type: 'bar',
50.            data,
51.            options: {
52.              plugins: {
53.                tooltip: {
54.                  callbacks: {
55.                    title: function (context) {
56.                      return get_tooltip(context);
57.                    }
58.                  }
59.                }
60.              }
61.            }
62.          };
63.          var myChart = new Chart(
64.            document.getElementById('myChart'),
65.            config
66.          );
67.        });
68.    </script>
69.  </body>
70.
71.  </html>
72.
```

## 7.3 Appendix C: performance measurements

***Figure 14.*** *Performance measurements for line charts created with D3.js*

| Data size | Load time (ms) | Execution (ms) | Memory (MB) |
|---|---|---|---|
| 10 | 218,6 | 12,3 | 2,52 |
| 100 | 267,2 | 25,2 | 2,71 |
| 1000 | 273 | 42,8 | 2,78 |
| 10000 | 292,8 | 51,1 | 4,17 |
| 100000 | 290,4 | 289,8 | 19,57 |
| 1000000 | 289 | 1987,3 | 173,17 |

***Figure 15.*** *Performance measurements for line charts created with Chart.js*

| Data size | Load time (ms) | Execution (ms) | Memory (MB) |
|---|---|---|---|
| 10 | 190,5 | 154,8 | 2,35 |
| 100 | 192,3 | 259,3 | 2,67 |
| 1000 | 199,5 | 397,3 | 3,13 |
| 10000 | 182,7 | 4081,5 | 14,27 |
| 100000 | 228,3 | 236577,2 | 97,56 |

***Figure 16.*** *Performance measurements for line charts with visible data points created with D3.js*

| Data size | Load time (ms) | Execution (ms) | Memory (MB) |
|---|---|---|---|
| 10 | 209,2 | 16,2 | 3,21 |
| 100 | 222,8 | 18,9 | 3,65 |
| 1000 | 287,4 | 31,2 | 5,67 |
| 10000 | 345,1 | 164,7 | 40,62 |
| 100000 | 322,4 | 1134,3 | 339,27 |
| 1000000 | 383,5 | 11950,5 | |