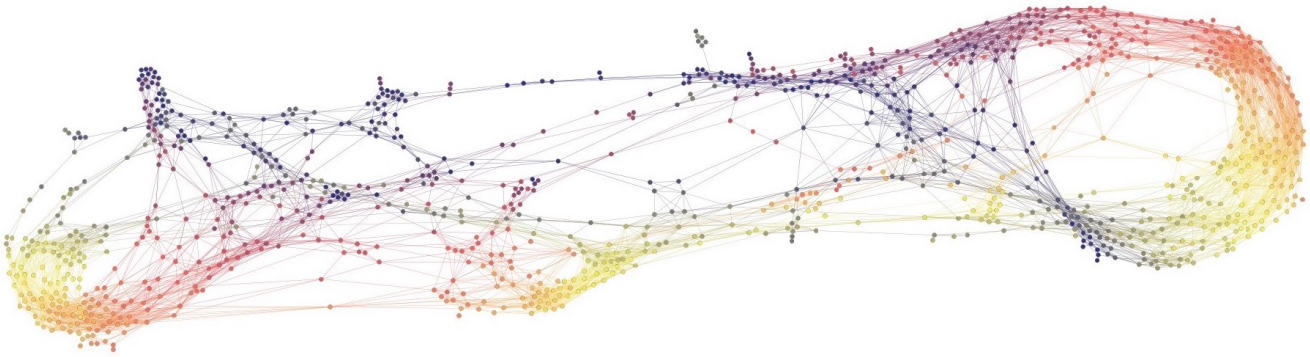
Data visualisation technologies

Implementing data visualisation using Svelte

Jan Aerts

Table of Contents



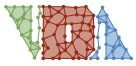
Prof Jan Aerts
Visual Data Analysis Lab
http://vda-lab.github.io
jan.aerts@vda-lab.io | jan.aerts@kuleuven.be

Teaching assistants: Jelmer Bot, Jannes Peeters, Dries Heylen (email: `firstname.lastname@uhasselt.be`)

This material is part of different courses on data visualisation, including at the universities of Leuven and Hasselt (Belgium), and at the European Bioinformatics Institute.

An online (and interactive) version is available at http://vda-lab.gitlab.io/datavis-technologies.

Exercises are provided by Jelmer Bot.

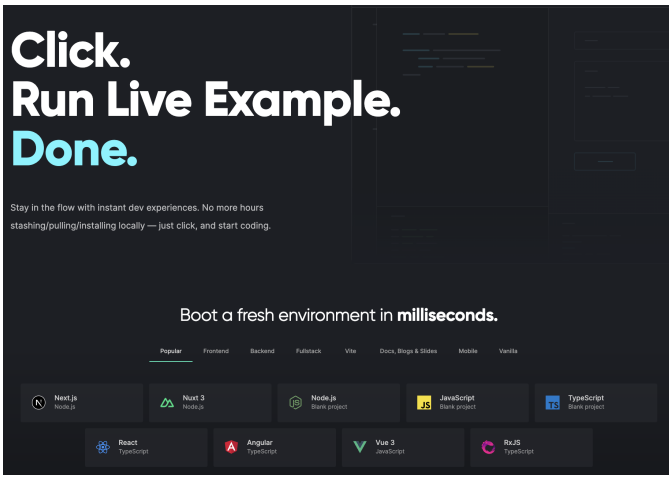


Chapter 1. Environment setup

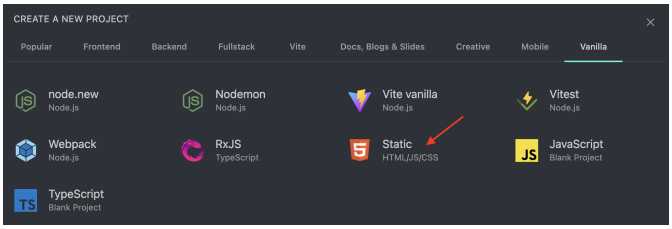
In this part of the course, we will focus on the programming aspect. We will use standard web technologies for this: HTML, CSS and javascript.

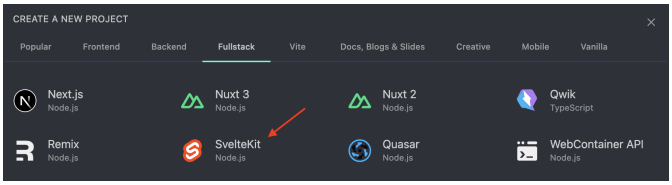
We will use stackblitz, an "instant fullstack web IDE for the javascript ecosystem" (source: Stackblitz developer site). Nevertheless, we also mention below how to set up the necessary stack on your own computer.

1.1. Stackblitz



There are many different types of projects that stackblitz has templates for. For data visualisation, you might use a vanilla HTML/CSS/javascript setup, or (as in this course) Sveltekit.





1.2. Local installation

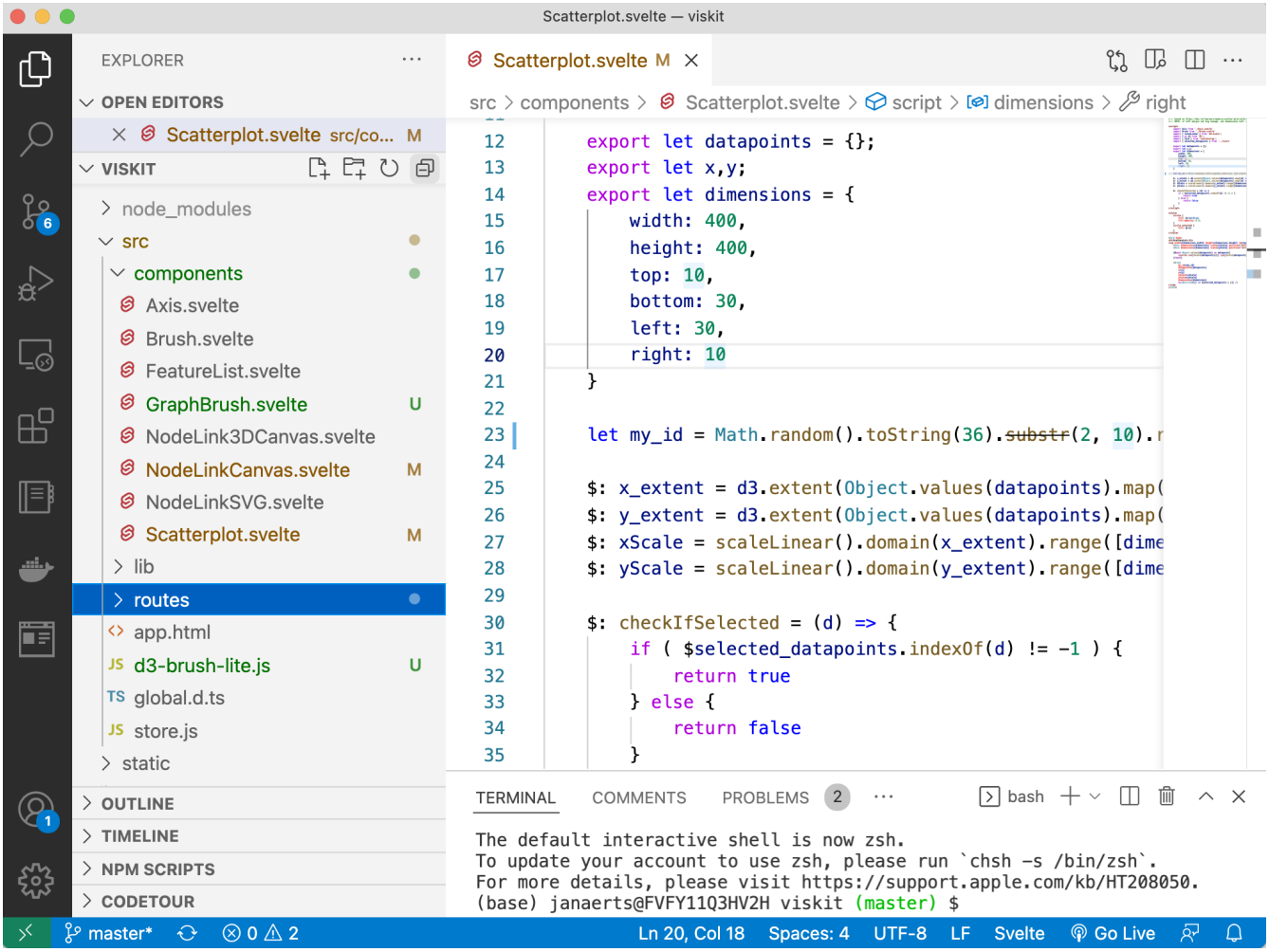
If you want to set up a development environment on your own machine, you will need node and Visual Studio Code.

1.2.1. Node

We'll need to install node, a javascript runtime. You can get it here.

1.2.2. Visual Studio Code

We'll use Visual Studio Code ("vscode" for short) as a code editor. You can get it here.



You will also want to add some *extensions* to make programming for visualisation a bit easier. You can do this by clicking on the "Extensions" icon in the left bar:

The extensions that we recommend, are:

- Svelte for VS Code (by Svelte)
- Svelte Intellisense (by ardenivanov)

1.3. Exercises

We have set up a group of exercises that will guide you through the different facets of creating visualisations using web technologies and sveltekit. Below are the instructions on how to get these.

Svelte is a language and compiler for building web applications. It allows you to create reusable components. **SvelteKit** is a full-stack web application framework built *on top of* Svelte.

1.3.1. Github

To be able to log into stackblitz, you'll have to create an account on github first. This will make it easier to get everything organised.

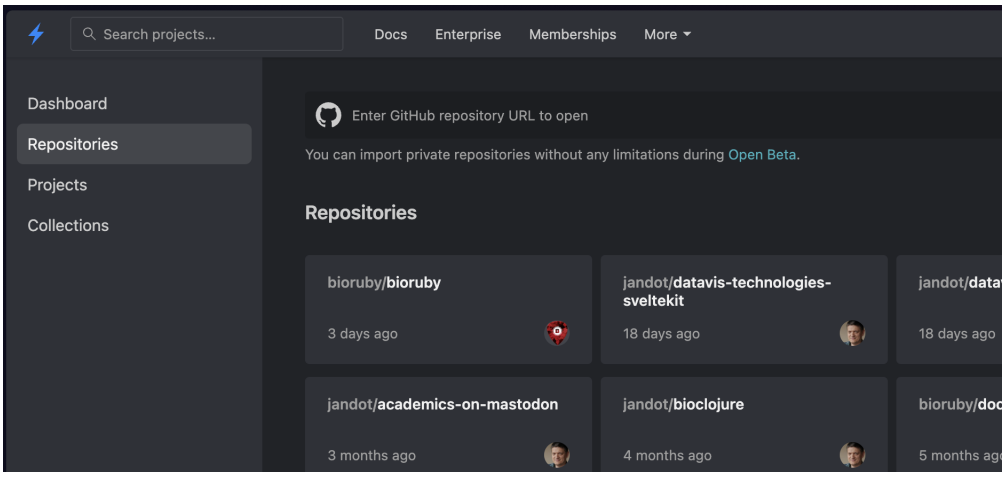
We have prepared a project on github so that the right structures are set up and the necessary boilerplate code is ready to go. To get set up:

On github

- Log into github
- Go to http://github.com/vda-lab/datavis-technologies-handson
- Click on "Fork"

On stackblitz

- Log in using your github account
- Go to https://stackblitz.com/codeflow and click on "Join the Codeflow Beta"
- Click on "repositories" in the left and then on the "datavis-technologies-handson"



Chapter 2. HTML, CSS, and javascript

Back in the 1990s, java was the programming language of choice if you wanted something visual and interactive on your screen. Over the years, this has shifted towards standard web technologies HTML, CSS and javascript.

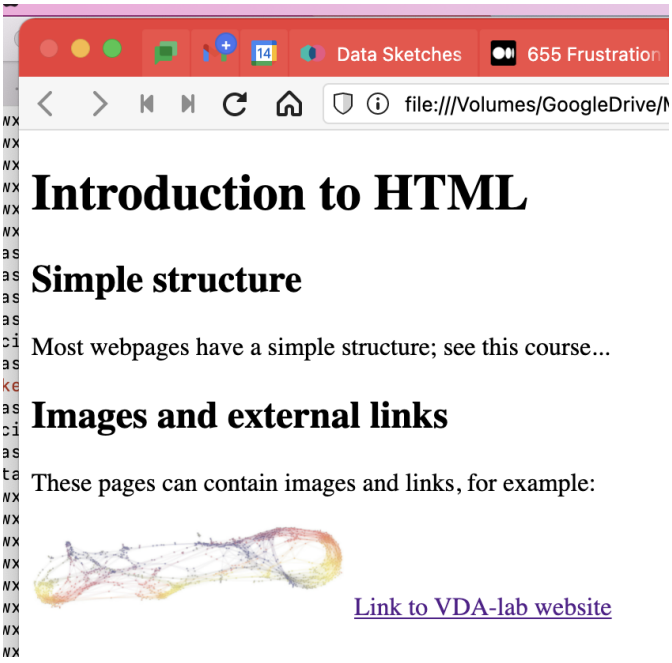
2.1. General HTML file structure

Any website you visit basically consists of the same components. The whole page is contained within a `html` section, which will contain a `body`, and possibly a `head` and `script`. Below is a minimal html page.

```
<!DOCTYPE html>
<html>
    <head>
    </head>
    <body>
        The actual content is here...
    </body>
</html>
```

A page containing some actual content:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My webpage</title>
    </head>
    <body>
        <h1>Introduction to HTML</h1>
        <h2>Simple structure</h2>
        <p>Most webpages have a simple structure; see this course...</p>

        <h2>Images and external links</h2>
        <p>These pages can contain images and links, for example:</p>
        <img src="http://vda-lab.github.io/assets/stad_airquality.png" width=200 />
        <a href="http://vda-lab.github.io">Link to VDA-lab website</a>
    </body>
</html>
```



2.2. Create your first page

At this point, create a file called with an `.html` extension with the contents shown above (e.g. using vscode), and open it in any webbrowser. At this point, we suggest creating a separate file for different exercises (e.g. `exercise1.html`, `exercise2.html`, etc).

2.3. The Document Object Model (DOM)

To give some structure to HTML we use *tags* to indicate *elements*. The collection of all these (nested) elements is called the Document Object Model or *DOM*.

Tags exist to denote headers, lists, paragraphs, etc.

Most tags need to be *opened* and then *closed*. For example:

- `<li>` ... `</li>`
- `<h3>` ... `</h3>`

There are some exceptions, such as when you create a horizontal line (`<hr/>`) or want to add a linebreak (`<br/>`).

Here's a list of the most relevant attributes:

- `<h1>`, `<h2>` ...: heading of level 1, 2, ...
- `<ul>`: unordered list
- `<ol>`: ordered list
- `<li>`: list item within unordered or ordered list
- `<p>`: paragraph
- `<g>`: group other elements

- ...

`<div>` is a generic container that will be useful later in the tutorial.

You can find a complete list of HTML elements at https://developer.mozilla.org/en-US/docs/Web/HTML/Element.

2.3.1. Attributes

Tags can have attributes, for example:

- `<a href="www.google.com">` defines the destination of a link, for example like this.
- `<p style="color: blue">` sets the colour of that paragraph to blue
- ...

`id` and `class`

We will use the special attributes `id` and `class` a lot later in this session.

The `id` attribute gives an element a certain id (obviously), and that id needs to be unique within the document.

The `class` attribute assigns that element to a certain class. We can use this to easily select different elements together. See for example the `unimportant` class that we add to two of the shopping list items below. A single HTML element can have multiple classes. We can do this by adding them in all separate with a space. For example: `<li class="datapoint selected">`.

Some other examples

```
<p>Shopping list:</p>
<ul>
<li class="unimportant">Eggs</li>
<li class="unimportant">Milk</li>
<li>Toothpaste</li>
</ul>
```

A button that does nothing, and a generic `div` element. But note that the generic `div` has an `id`.

```
<button type="submit">Submit</button>
<div id="my_new_div">... and a generic HTML block with an id</div>
```

2.4. Styling elements using CSS

A typical website will not only contain HTML, but also CSS (cascading style sheets) and javascript.

HTML
: provides the basic structure of the page, which can be enhanced using CSS and javascript

CSS
: is used to control presentation, formatting and layout: what it *looks* like.

javascript
: is used to control the *behaviour* of different elements and the page as a whole.

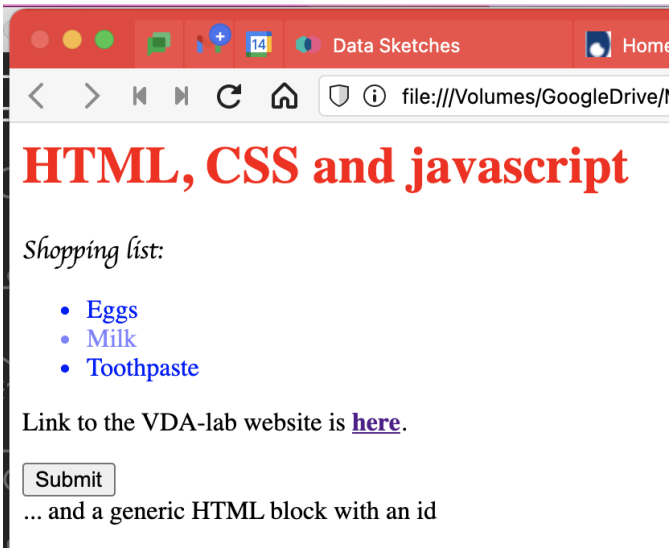
There are different ways to add CSS to your webpage.

2.4.1. Inline CSS

We can add CSS directives using the `style` attribute on HTML elements. For example:

```
<html>
    <head>
        <title>My HTML file</title>
    </head>
    <body>
        <h1 style="color: red;">HTML, CSS and javascript</h1>
        <p style="font-family: cursive;">Shopping list:</p>
        <ul>
            <li style="color: blue;">Eggs</li>
            <li style="color: blue; opacity: 0.5;">Milk</li>
            <li style="color: blue;">Toothpaste</li>
        </ul>
        <p>Link to the VDA-lab website is <a style="font-weight: bold;"
href="http://vda-lab.github.io">here</a>.</p>
        <button type="submit">Submit</button>
        <div id="my_new_div">... and a generic HTML block with an id</div>
    </body>
</html>
```

What it looks like:

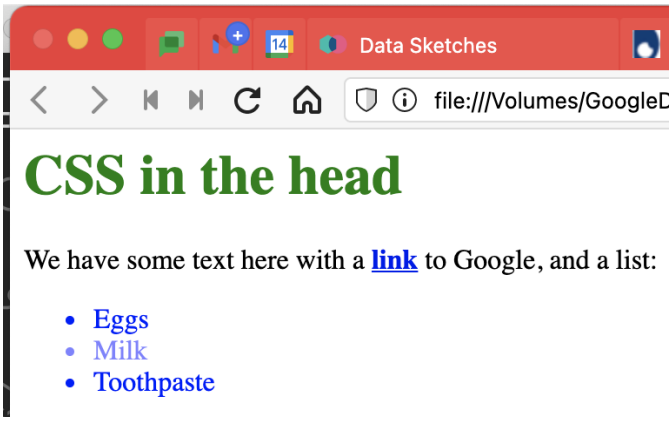


2.4.2. CSS in the head

When we use CSS inline as above, we can set the style for individual elements. But if we want to make all headers bold, we'd need to add `font-weight: bold` to *every* single `<h1>`. To set the style for whole groups of elements, we better define that once instead of for each element. We can do this in the `<head>` using the `<style>` element. In the example below, we make all headers of level 1 green, each list item should be blue, and links should be bold.

```
<html>
    <head>
        <title>My HTML file</title>
        <style>
            li { color: blue; }
            h1 { color: green; }
            a { font-weight: bold;}
        </style>
    </head>
    <body>
        <h1>CSS in the head</h1>
        <p>We have some text here with a <a href="www.google.com">link</a> to Google,
and a list:</p>
        <ul>
            <li>Eggs</li>
            <li style="opacity: 0.5;">Milk</li>
            <li>Toothpaste</li>
        </ul>
    </body>
</html>
```

The result:



As you can see in this example, you can combine CSS in the head with inline CSS in which case the inline CSS takes precedence: the second list item is partly transparent.

2.4.3. Using CSS selectors

In the last example, we set the color for every list item to blue, but wanted to have one specific item transparent as well. We did this by combining CSS directives at two different places. A better

approach is to separate the HTML and CSS even more, instead of hard-coding this transparency. We can do this using **CSS selectors**. These are a very powerful way to *select* different elements. The `id` and `class` attributes we mentioned above are crucial.

- To select all elements of a certain *type* (e.g. `<h1>` or `<p>`), use the name of that element.
- To select all elements with a certain *class* (e.g. `<li class="important">`), prepend that class with a period `.`.
- To select the element with a given *id* (e.g. `<p id="paragraph_5>`), prepend that id with a hash `#`.

Your CSS can look like this:

```
h1 { color: red; }
a { font-weight: bold;}
p {
    max-width: 400px;
}
.unimportant {
    opacity: 0.5
}
#paragraph_5 { font-family: cursive; }

ul li { color: blue; }
p:hover {
    background-color: aqua;
}
p::first-letter {
    font-size: larger;
    font-weight: bolder;
}
```

In the above, we

- make all `<h1>` red
- make all links `<a>` bold
- set the maximum with of all paragraphs (`<p>`) to 400 pixels
- give all elements that have `unimportant` as a class an opacity of 50%
- set the element with `id` of `paragraph_5` in italics

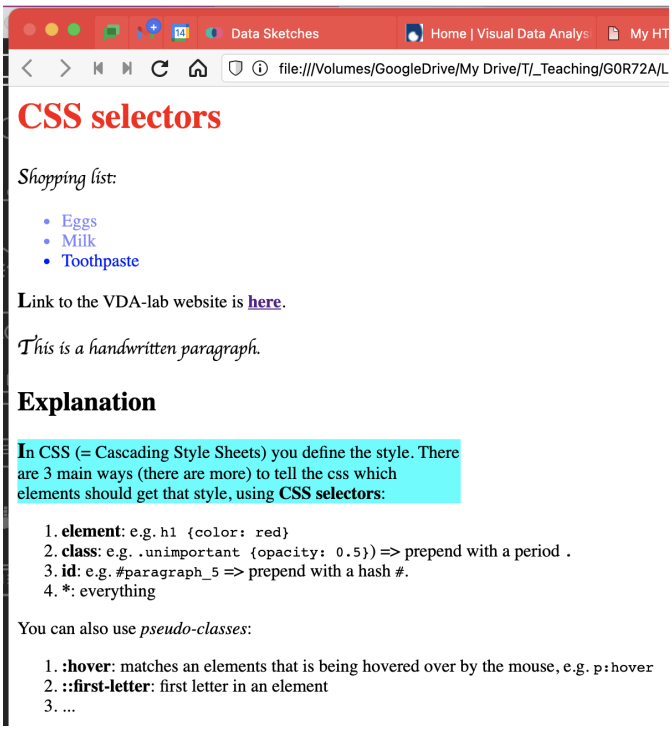
Those are the most basic selectors. But these can be combined and make much more complex. In the following line, we set the colour of a list item `<li>` to blue, but only if it's preceded by a `<ul>`. This means that items in an *ordered* list (`<ol>`) will *not* be blue. We can also use *pseudo-classes*, like `:hover` (which matches the element your mouse is hovering on), or `::first-letter` (which speaks for itself).

For the full reference, see https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors.

```
<html>
    <head>
        <title>My HTML file</title>
        <style>
            ul li { color: blue; }
            h1 { color: red; }
            a { font-weight: bold;}
            .unimportant {
                opacity: 0.5
            }
            p {
                max-width: 400px;
            }
            #paragraph_5 { font-family: cursive; }
            p:hover {
                background-color: aqua;
            }
            p::first-letter {
                font-size: larger;
                font-weight: bolder;
            }
        </style>
    </head>
    <body>
        <h1>CSS selectors</h1>
        <p style="font-family: cursive;">Shopping list:</p>
        <ul>
            <li class="unimportant">Eggs</li>
            <li class="unimportant">Milk</li>
            <li>Toothpaste</li>
        </ul>
        <p>Link to the VDA-lab website is <a href="http://vda-
lab.github.io">here</a>.</p>
        <p id="paragraph_5">This is a handwritten paragraph.</p>
        <h2>Explanation</h2>
        <p>In CSS (= Cascading Style Sheets) you define the style. There are 3 main
ways (there are more)
            to tell the css which elements should get that style, using <b>CSS
selectors</b>:
        </p>
        <ol>
            <li><b>element</b>: e.g. <kbd>h1 {color: red}</kbd></li>
            <li><b>class</b>: e.g. <kbd>.unimportant {opacity: 0.5}</kbd>)
                => prepend with a period <kbd>.</kbd></li>
            <li><b>id</b>: e.g. <kbd>#paragraph_5</kbd> => prepend with a
                hash <kbd>#</kbd>.</li>
            <li><b>*</b>: everything</li>
        </ol>
        You can also use <em>pseudo-classes</em>:
        <ol>
```

```
            <li><b>:hover</b>: matches an elements that is being hovered over by the
mouse,
                e.g. <kbd>p:hover</kbd></li>
            <li><b>::first-letter</b>: first letter in an element</li>
            <li>...</li>
        </ol>
    </body>
</html>
```

has this result:



Notice the brightly coloured paragraph...

2.4.4. CSS in a separate file

Instead of adding the complete css inline or in the head of a file, we can also put it in a separate file. We load this file in the head, using `<link rel="stylesheet" href="my_css_file.css">`.

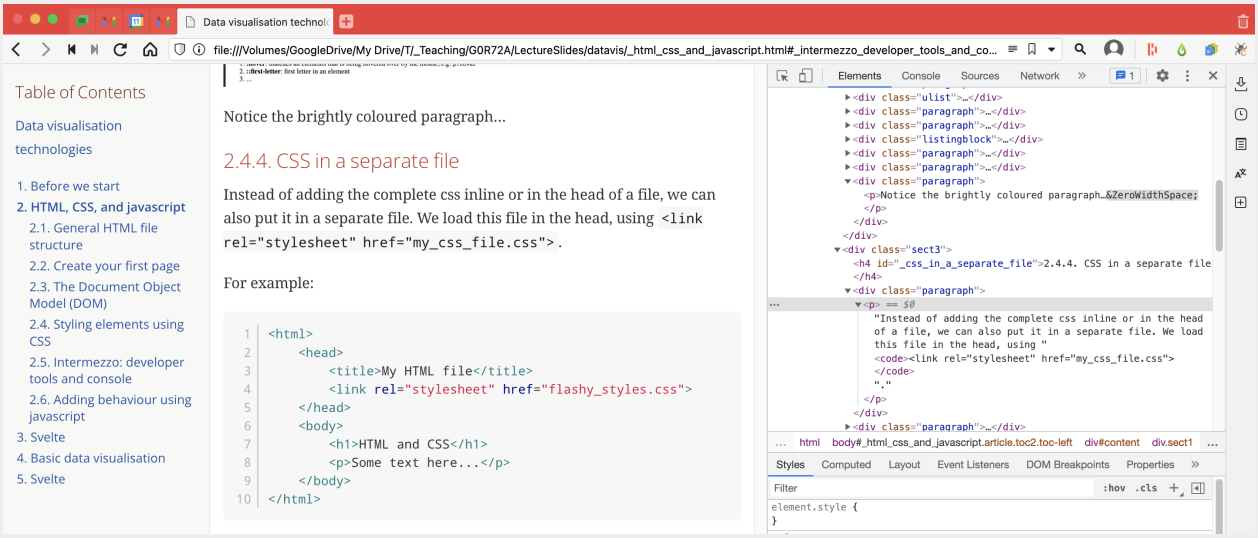
For example:

```
<html>
    <head>
        <title>My HTML file</title>
        <link rel="stylesheet" href="flashy_styles.css">
    </head>
    <body>
        <h1>HTML and CSS</h1>
        <p>Some text here...</p>
    </body>
</html>
```

Developer tools and console

Whatever you write in the `.html` file, it gets *rendered* in the browser so you do not see the tags anymore. However, each browser has *developer tools* that allow you to get to the underlying HTML code. You can activate them here:

- Chrome: View > Developer > Developer Tools
- Firefox: Tools > Browser Tools > Web Developer Tools
- Safari: Develop > Show Web Inspector
- Vivaldi: Tools > Developer Tools



In the image above you see the rendered version of the page on the left, and the underlying source on the right.

You will also see a tab named `console` in the developer tools.

TIP To know what's going on in your javascript code, you can add `console.log(your_variable)`. The value of that variable will then be visible in the console.

2.5. Adding behaviour using javascript

We can use javascript to add additional functionality to a site.

2.5.1. Inline javascript

We can add the javascript within the `body` between `script` tags. If we want to reference existing elements (such as the `demo` element in our example below, we have to put the `script` at the *end* of the `body`. In this example below, a counter is updated every time we push the button.

```
<html>
    <head>
```

```
        <title>My HTML file</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <p>We can use javascript to add additional functionality to a site.</p>
        <button onclick="handleClick()">Increment the counter</button>

        <p id="demo"></p>

        <script>
            let count = 0;
            function handleClick() {
                count += 1
                document.getElementById("demo").innerHTML = "You clicked the button "
+ count + " times."
            }
        </script>
    </body>
</html>
```

Let's walk through this code:

- In line 8, we create a `button` to which we attach an action whenever we click (`onclick`) on it. The action is called `handleClick` (but this might be any name that we give it).
- Lines 343 to 349 define the javascript.
  - We set a counter to zero in line 344. This happens the moment that the page is opened or refreshed.
  - The actual function is defined in lines 345 to 348. Whenever we click the button, these lines are executed.
  - First we increment the counter with 1.
  - Then we find the element with `id` of `demo` (`document.getElementById("demo")`) and set its contents (`innerHTML`) to the text "You clicked..."

The `document.getElementById()` might be overwhelming, but we will find easier ways to do this later.

2.5.2. Loading javascript from a file

If the code becomes a bit more involved, we can put it in a separate file, and reference it in the `<head>` like so, in line 5 below:

```
<html>
    <head>
        <title>My HTML file</title>
        <link rel="stylesheet" href="styles.css">
        <script src="script1.js"></script>
    </head>
    <body>
```

```
        <p>We can use javascript to add additional functionality to a site.</p>
        <button onclick="handleClick()">Increment the counter</button>

        <p id="demo"></p>
    </body>
</html>
```

Intermezzo - The JSON format

From the next section onwards we will work with variables and data. At this point it is important to describe the JSON data format.

The JSON ("JavaScript Object Notation") data representation format follows the same principle as XML, in that it describes the data in the object itself. This format is used in many Application Programming Interfaces (APIs) such as https://dummyjson.com/products, as well as the internal representation of complex data in javascript code.

An example JSON object containing information about the BRCA2 gene, involved in breast cancer:

```
{ id: 12345,
  common_name: "BRCA2",
  names: ["BRCA2", "ENST00000544455.6", "FACD", "FANCD1"],
  description: "Breast cancer type 2 susceptibility protein",
  location : {
    genome_build: "hg38",
    chromosome: "13",
    start: 32315086,
    end: 32400268,
    strand: "+"
  },
  diseases: [{
    name: "breast cancer",
    description: "Defects in BRCA2 are a cause of susceptibility to breast cancer
(BC)"
  }, {
    name: "BROVCA2",
    description: "Defects in BRCA2 are involved in familial breast-ovarian cancer
type 2"
  }, {
    name: "FANCD1",
    description: "Defects in BRCA2 are the cause of Fanconi anemia
complementation group D type 1"
  }]
}
```

Data in JSON format (or javascript objects) are presented in key/value pairs. To be completely JSON-compliant the key should be put in quotes, although those are often omitted for brevity

(as in the example above). Different key/value pairs are separated by a comma. JSON values can be of different types. They can be:

- **Scalars**: strings (in quotes; see e.g. `common_name` above), numbers (without quotes; see e.g. `start`), booleans (`true`/`false`), or `null`.
- **Arrays** containing zero or more elements, surrounded by square brackets (`[]`). For example, see `names` in the example above.
- Other **objects**, surrounded by curley brackets (`{}`; see e.g. `location`).

The values of arrays and objects can again be scalars, arrays or other objects. For example, the value for the `diseases` key is an array of objects.

The top-level element in a JSON-formatted file can be of any of the above types. In the example, this is an object.

2.5.3. Javascript basics

Variables

Variables in javascript are defined using the `var`, `let`, or `const` commands. For all purposes, `var` and `let` are exactly the same. There are slight differences in their scopes, but `let` was created because `var` 's scope was an important source for bugs in javascript. Ergo: use `let`.

Declaring and *assigning* variables are two different things: a variable *declaration* means that you create the variable and give it a name; with variable *assignment* you give it an actual value.

For example:

```
var first_variable;           // declaration of first_variable
first_variable = "a";         // assignment of value to first_variable
let second_variable;          // declaration of second_variable
second_variable = "b";        // assignment of value to second_variable
let third_variable = "c";     // declaration and assignment done in one go
console.log(third_variable);  // value of third_value is printed to the console
third_variable = "d";         // third_variable gets a new value
console.log(third_variable);  // value of third_value is printed to the console
```

In contrast to `var` and `let`, a variable that is declared using `const` can *not* be assigned a new value.

Everything on a line after a double forward slash (`//`) in javascript is seen as a comment and not parsed.

NOTE	We'll see later that in Svelte, you can also declare variables using `$:` instead of `let` or `const`.

Functions

There are several ways of creating functions in javascript. Below, we show three statements that do the same thing: take 2 arguments, add them, return the result.

```
// First version
function function_1(a,b) { return a + b }

// Second version
let function_2 = function(a,b) { return a + b }

// Third version
let function_3 = (a,b) => { return a + b }

console.log(function_1(1,2))
console.log(function_2(1,2))
console.log(function_3(1,2))
```

For the second and third version, it might sometimes be useful to use `const` (or `$:`) depending on the use case. The difference between the first version on the one hand, and the second and third on the other is not relevant in the cases that we will use them.

2.6. Exercises

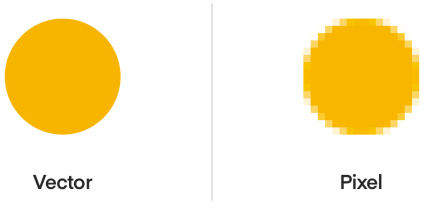
Here are some exercises related to this chapter:

- Headers: https://svelte.dev/repl/1d295420874d42818954da8fcb50ad7d?version=3.59.1
- Lists: https://svelte.dev/repl/d4a024f5e6794abfa2ccc213d41c7e18?version=3.59.1
- Text tags: https://svelte.dev/repl/f1a9ed0ed9444c12a1684a00166619e3?version=3.59.1
- Styles (inline): https://svelte.dev/repl/e1f0c316e39c45cb97fbf1fe18126e4c?version=3.59.1
- CSS classes: https://svelte.dev/repl/987e9e43eab74967912e83b302401291?version=3.59.1
- Printing: https://svelte.dev/repl/f7fe97bc39ba4ecf8b9077bf5e497450?version=3.59.1
- Variables: https://svelte.dev/repl/5b50bf2c9bea4811ad62ef82ca38f18e?version=3.59.1
- Arrays: https://svelte.dev/repl/b7f4e0f2a9034b16972f82f86df39ee1?version=3.59.1
- Objects: https://svelte.dev/repl/f5fe639471b04e588048059076e25ad7?version=3.59.1
- Functions: https://svelte.dev/repl/b1e70f4349cc4a4ca8169f808794a099?version=3.59.1
- Scope: https://svelte.dev/repl/7c44e0ca379c4e869aa37d6dceb1429d?version=3.59.1
- Importing: https://svelte.dev/repl/424ccb07f0bf4a23a82ea4789f8ca25d?version=3.59.1

Chapter 3. Basic data visualisation

3.1. Scalable vector graphics

Knowing how HTML, CSS and javascript work, we can start making our visualisations. There are two main approaches for making visualisations in a webbrowser. You can either tell the computer which pixels need to have which colour (*pixel-based*), or you can tell it at a higher level that you want a circle at a certain position on the screen (*vector-based*). Although the pixel-based approach gives more flexibility and control, it can be much harder to create these visuals and interact with them. In addition, vector-based visuals are *scalable*. The image below shows the difference between vector- and pixel-based circle after zooming:



We will be using SVG (scalable vector graphics) for this tutorial. A plot is a collection of `<circle>`, `<rect>`, etc elements that are contained within a single `<svg>` element. They are basically like any other HTML element. Here is a simple plot of 300x300 pixels, containing a circle and a rectangle.

```
<html>
<body>
    <svg width="300" height="300">
        <circle cx="20" cy="50" r="30"/>
        <rect x="100" y="200" width="70" height="30" />
    </svg>
</body>
```

IMPORTANT

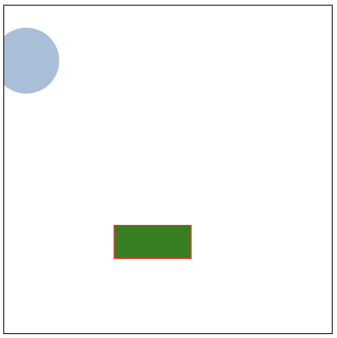
The **origin** `(0,0)` of an SVG image is at the **top left**, not the bottom left. This means that higher values for `y` will give you marks that are lower in the graphic, as you can see with the circle and the rectangle below. When plotting actual data, make sure that you flip that orientation.

As these are regular HTML elements, we can style them using CSS as well:

```
<html>
<head>
    <style>
        svg {
            border: 1px;
            border-style: solid;
        }
        circle {
```

```
            fill: steelblue;
            opacity: 0.5;
        }
        circle:hover {
            fill: red;
            opacity: 1;
        }
        rect {
            fill: green;
            stroke: red;
        }
    </style>
</head>
<body>
    <svg width="300" height="300">
        <circle cx="20" cy="50" r="30"/>
        <rect x="100" y="200" width="70" height="30" />
    </svg>
</body>
```

We give the SVG itself a border. Any circle should be blue and 50% transparent, except when we hover over it when it should become red and fully opaque. Rectangles should be green with a red outline. This is the resulting graphic (see what happens when you hover over the circle):



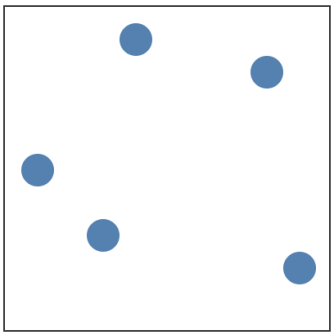
There are a large number of elements that can be used within an SVG element. These include `circle`, `rect`, `line`, `path`, etc. Each has their own list of possible attributes. For a full overview, see https://developer.mozilla.org/en-US/docs/Web/SVG/Element/svg. Definitely check out that resource.

3.1.1. A simple scatterplot

Let's say we have 5 datapoints that we want ot put into a scatterplot: `[[20,100],[60,140],[80,20],[160,40],[180,160]]`. We can make a plot like this:

```
<html>
<head>
    <style>
        svg {
            border: 1px;
            border-style: solid;
        }
```

```
        circle {
            fill: steelblue;
        }
    </style>
</head>
<body>
    <svg width="200" height="200">
        <circle cx="20" cy="100" r="10"/>
        <circle cx="60" cy="140" r="10"/>
        <circle cx="80" cy="20" r="10"/>
        <circle cx="160" cy="40" r="10"/>
        <circle cx="180" cy="160" r="10"/>
    </svg>
</body>
</html>
```



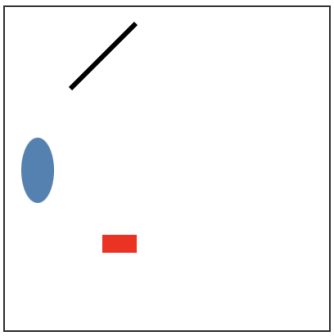
3.1.2. Rectangle, ellipse, line

Other straightforward marks available to you in SVG are rectangles, ellipses, and lines.

TIP Make sure to bookmark the Mozilla MDN Docs at https://developer.mozilla.org/en-US/docs/Web/SVG. You can find information there on what parameters are available for which shapes, how to draw text, transformations and coordinate systems, etc.

```
<style>
    line {
        stroke: black;
        stroke-width: 3;
    }
    ellipse {
        fill: steelblue;
    }
    rect {
        fill: red;
    }
</style>

<svg width="200" height="200">
    <ellipse cx="20" cy="100" rx="10" ry="20" />
    <rect x="60" y="140" width="20" height="10" />
```

```
    <line x1="80" y1="10" x2="40" y2="50" />
</svg>
```



3.1.3. Polylines and polygons

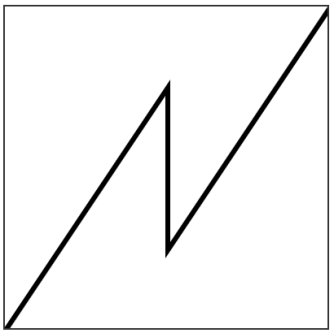
If we want to create a jagged line, we can add multiple lines together when one starts where another ends, like so:

```
<svg width="200" height="200">
    <line x1="0" y1="200" x2="100" y2="50" />
    <line                 x1="100" y1="50" x2="100" y2="150" />
    <line                                  x1="100" y1="150" x2="200" y2="0" />
</svg>
```

(I've added whitespace here to indicate that the startpoints of each line are the same as the endpoints of the previous line.)

Of course it's nicer to do that in one go, like so:

```
<style>
    polyline {
        fill: none;
        stroke: black;
        stroke-width: 3px;
    }
</style>

<svg width="200" height="200">
    <polyline points="0,200 100,50 100,150 200,0" />
</svg>
```

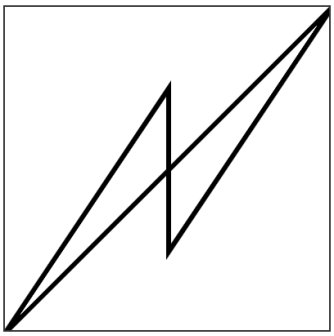


Note that we defined `fill` as `none`. If we give the element a fill colour, it act as if the start and end point are connected. For example:

```
<style>
    polyline {
        fill: green;
        stroke: black;
        stroke-width: 3px;
    }
</style>

<svg width="200" height="200">
    <polyline points="0,200 100,50 100,150 200,0" />
</svg>
```

A `polygon` is almost exactly the same as a `polyline`, but it connects the start and end points.

```
<style>
    polygon {
        fill: none;
        stroke: black;
        stroke-width: 3px;
    }
</style>

<svg width="200" height="200">
    <polygon points="0,200 100,50 100,150 200,0" />
</svg>
```

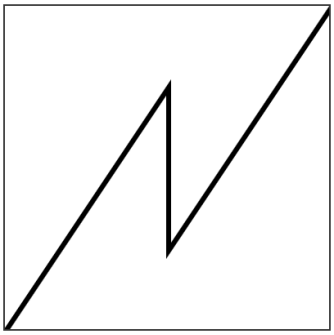


3.1.4. Paths

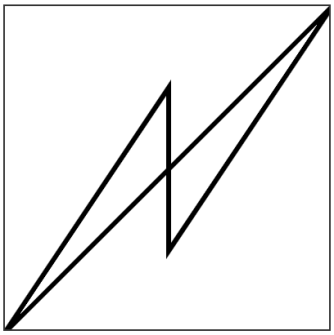
Any of the marks shown above can also be created using a `path`. The Mozilla MDN Docs have pretty good tutorial on them.

A path has one argument, `d` which is a string that describes the path. At its most basic, we can move (`M` or `m`) the mouse (without drawing), and draw a lines (`L` or `l`). We can recreate the polyline from above like so:

```
<style>
    path {
        fill: none;
        stroke: black;
        stroke-width: 3px;
    }
</style>

<svg width="200" height="200">
    <path d="M 0,200 L 100,50 L 100,150 L 200,0" />
</svg>
```

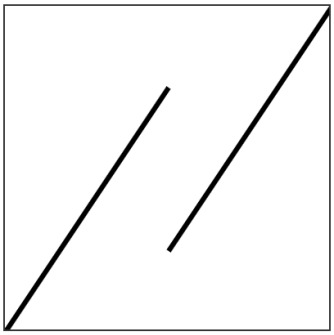


We can close the path (and change the polyline to a polygon) by adding a `Z` at the end of the string: `<path d="M 0,200 L 100,50 L 100,150 L 200,0 Z" />`.



So `M` moves the pen across the paper without touching the paper; `L` does so with touching the paper. We can illustrate this by replacing one of the `L` s above into an `M`.

```
<style>
    path {
        fill: none;
        stroke: black;
        stroke-width: 3px;
    }
</style>

<svg width="200" height="200">
    <path d="M 0,200 L 100,50 M 100,150 L 200,0" />
</svg>
```



Relative positions

The coordinates given for `M` and `L` are the absolute coordinates within the SVG element. We can also give positions that are *relative* to the current position. We do this using the lowercase `m` and `l`. This means that `M 50,50 L 50,100 L 100,50` is the same as `M 50,50 l 0,50 l 50,-50`.

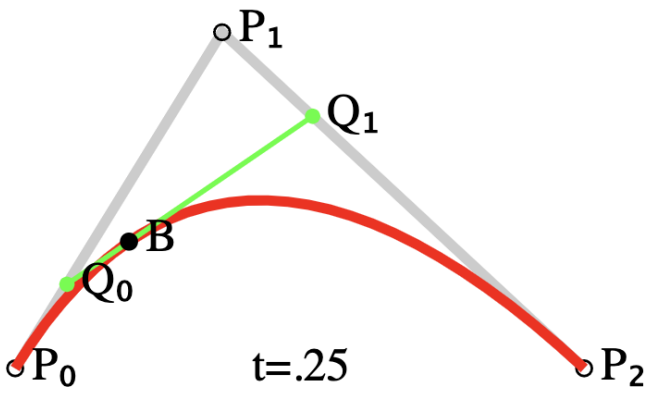
3.1.5. Curves

The `path` element can also be used to draw curves like bezier curves and arcs.

Bezier curves

In SVG we can draw quadratic (`Q`) and cubic (`C`) bezier curves. A **quadratic bezier curve** is an interpolation between two linear interpolations. In the image below, \(P_0\) and \(P_2\) are the *anchor points* and \(P_1\) is a *control point*. We interpolate between \(P_0\) and \(P_1\) on one hand,

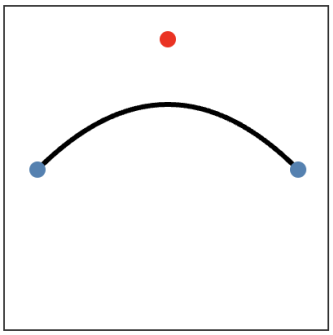
and \(P_1\) and \(P_2\) on the other. We connect those interpolations with a (green) line. Next, we interpolate across that green line to get the bezier curve itself.



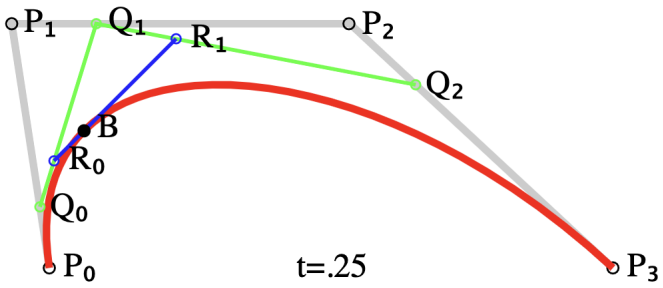
Source: https://en.wikipedia.org/wiki/Bézier_curve

In an SVG `path`, the \(P_0\) is the last point in the path, followed by the control point \(P_1\), and second anchor point \(P_2\). In the example below, \(P_0\) is `20,100`, \(P_1\) is `100,20`, and \(P_2\) is `180,100`. We also draw the points themselves as a reference (control point is in red).

```
<style>
    path {
        stroke: black;
        stroke-width: 3px;
        fill: none;
    }
</style>

<svg width="200" height="200">
    <path d="M 20,100 Q 100,20 180,100" />

    <circle cx="20" cy="100" r="5" style="fill: steelblue" />
    <circle cx="100" cy="20" r="5" style="fill: red" />
    <circle cx="180" cy="100" r="5" style="fill: steelblue" />
</svg>
```



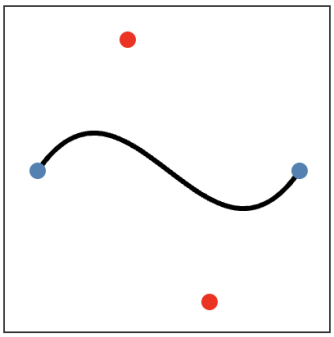
A **cubic bezier curve** takes this one level higher, and therefore uses two control points.



Source: https://en.wikipedia.org/wiki/Bézier_curve

An example:

```
<style>
    path {
        stroke: black;
        stroke-width: 3px;
        fill: none;
    }
</style>

<svg width="200" height="200">
    <path d="M 20,100 C 75,20 125,180 180,100" />
    <circle cx="20" cy="100" r="5" style="fill: steelblue;" />
    <circle cx="75" cy="20" r="5" style="fill: red;" />
    <circle cx="125" cy="180" r="5" style="fill: red;" />
    <circle cx="180" cy="100" r="5" style="fill: steelblue;" />
</svg>
```



Arcs

Arcs are tricky in SVG, so definitely check out the `path` tutorial at https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Paths if you need them.

As with the bezier curves, a curve starts from a certain point that is already defined in the `path`. So you will need for example a `M` or `L` directive before the arc itself. The arc is drawing using the `A` directive, which takes the following arguments:

- radius in x-direction `rx`
- radius in y-direction `ry`

- rotation around x axis
- `large-arc-flag`
- `sweep-flag`
- x coordinate of end point
- y coordinate of end point

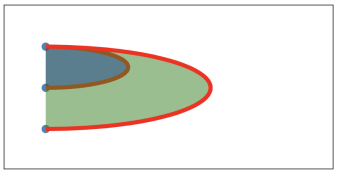
For example:

```
<style>
    path {
        fill: orange;
        stroke: black;
        stroke-width: 3px;
    }
</style>

<svg width="200" height="200">
    <path d="M 50 50 A 2 1 -45 0 1 50 150" />
</svg>
```

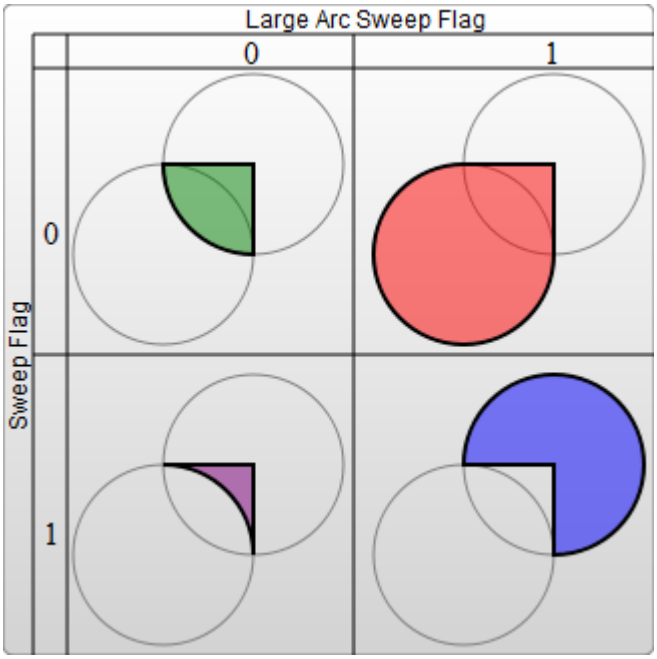
Notice that the radii `rx` and `ry` are scaled so that the arc can reach the end point.

```
<style>
    path {
        stroke: red;
        stroke-width: 5;
        fill-opacity: 0.5
    }
    path.smallarc {
        fill: blue;
    }
    path.largearc {
        fill: green;
    }
    circle {
        fill: steelblue;
    }
    svg {
```

```
        border: 1px;
        border-style: solid;
    }
</style>

<svg width="400" height="200">
    <circle cx="50" cy="50" r="5" />
    <circle cx="50" cy="150" r="5" />
    <circle cx="50" cy="100" r="5" />
    <path d="M 50 50 A 4 1 0 0 1 50 100" class="smallarc"/>
    <path d="M 50 50 A 4 1 0 0 1 50 150" class="largearc"/>
</svg>
```



For an explanation of the `large-arc-flag` and `sweep-flag`, please see the tutorial mentioned above. The image below shows their effect on an arc:

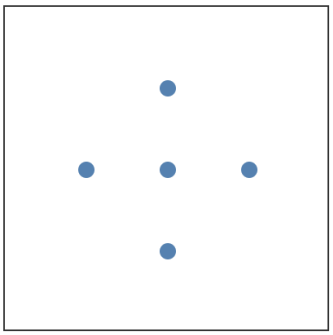


3.2. Groups and transformations

When putting elements (like circles) on the screen, we can define their positions in `cx` and `cy` using the coordinate system that they end up with. Say we want to have 5 circles in a `+` pattern, like so:

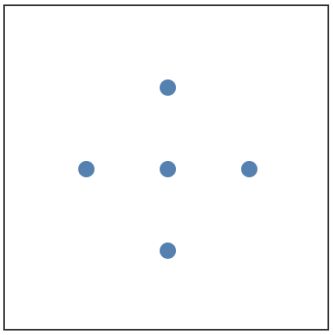
```
<style>
    circle {
        fill: steelblue;
    }
</style>
```

```
<svg width="200" height="200">
    <circle cx="100" cy="100" r="5" />
    <circle cx="50" cy="100" r="5" />
    <circle cx="100" cy="50" r="5" />
    <circle cx="150" cy="100" r="5" />
    <circle cx="100" cy="150" r="5" />
</svg>
```



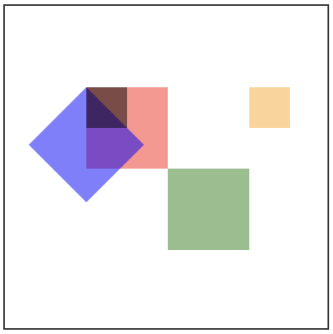
Instead of setting the center of the `+` at `(100,100)` it might be easier to set it at `(0,0)` and define the other points relative to that position, and then shift (translate) everything to afterwards. We can do this using the `transform` directive. `transform` can be applied to almost any HTML element, but here we will first create a **group** around the circles so that we only need to transform the group instead of every circle separately.

```
<style>
    circle {
        fill: steelblue;
    }
</style>

<svg width="200" height="200">
    <g transform="translate(100,100)">
        <circle cx="0"   cy="0"   r="5" />
        <circle cx="-50" cy="0"   r="5" />
        <circle cx="0"   cy="50"  r="5" />
        <circle cx="50"  cy="0"   r="5" />
        <circle cx="0"   cy="-50" r="5" />
    </g>
</svg>
```



Apart from translation, `transform` let's us also scale, and rotate elements, as well as combine any of these. In the example below, the red square gets translated (green), rotated (blue), scaled (black) and scaled after rotation (orange). Note that in the last case, it *does* matter in which order you perform the transformations: scaling after rotation is not the same as rotation after scaling.

```
<style>
    rect {
        fill-opacity: 0.5;
        stroke: none;
    }
    rect.base {
        fill: red;
    }
    rect.translated {
        fill: green;
    }
    rect.rotated {
        fill: blue;
    }
    rect.scaled {
        fill: black;
    }
    rect.combined {
        fill: orange;
    }
</style>
<svg width="200" height="200">
    <g transform="translate(50,50)">
        <rect x="0" y="0" width="50" height="50" class="base"/>
        <rect x="0" y="0" width="50" height="50" class="translated"
                    transform="translate(50,50)"/>
        <rect x="0" y="0" width="50" height="50" class="rotated"
                    transform="rotate(45)"/>
        <rect x="0" y="0" width="50" height="50" class="scaled"
                    transform="scale(0.5)"/>
        <rect x="0" y="0" width="50" height="50" class="combined"
                    transform="translate(100,0) scale(0.5)"/>
    </g>
</svg>
```



3.3. Creating SVG using javascript

Up to now we have only shown a few datapoints, and hard-coded their positions. Obviously, this is not the approach to use when we work with more than these few datapoints. This is where javascript comes in again. Let's create a scatterplot with 10 random points. We create a `scatterplot.js` file with the following content:

```
var my_plot = document.getElementById('my_plot')
for ( var i = 0; i < 10; i++ ) {
    let newElement = document.createElementNS('http://www.w3.org/2000/svg','circle')
    newElement.setAttribute('cx', Math.floor(Math.random()*300));
    newElement.setAttribute('cy', Math.floor(Math.random()*300));
    newElement.setAttribute('r','20');
    my_plot.appendChild(newElement);
}
```

and the following HTML file:

```
<html>
<head>
    <style>
        svg {
            border: 1px;
            border-style: solid;
        }
        circle {
            fill: steelblue;
            opacity: 0.5;
        }
        circle:hover {
            fill: red;
        }
    </style>
</head>
<body>
    <svg id="my_plot" width="300px" height="300px">
        <!-- This is empty! -->
    </svg>
    <script src="scatterplot.js"></script>
</body>
</html>
```

What's happening?

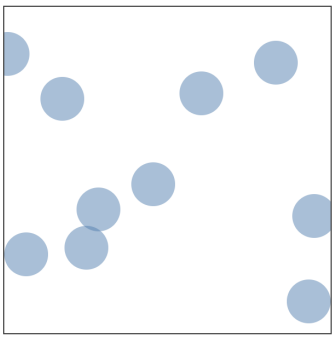
In `scatterplot.js`:

- In line 1, we extract the element in the HTML file with id `my_plot`.

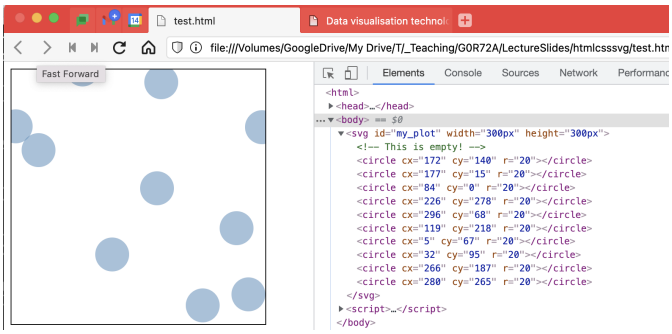
IMPORTANT	We need to put the reference to the external script at the *bottom* of the HTML code, because the element with that ID needs to exist before you try to access it.

- We loop 10 times, where each time we
  - create a new `circle` element
  - give it a random `x` and `y` position
  - give it a radius `r`
  - append that new element to `my_plot`

The result should look like this (hover over the points for interactivity):



So even though the `index.html` file has an *empty* `svg` element, we *do* see the circles on the screen. Check the source code in the developer tools: you will see 10 `circle` elements nested within the `svg`.



This looks complicated, but will become much simpler once we start using svelte.

3.4. Exercises

Here are some exercises related to this chapter:

- SVG: https://svelte.dev/repl/bbe83abbe89f4e00a1a9d0b87f55b555?version=3.59.1

Chapter 4. Basic data visualisation with svelte

NOTE	This section contains several interactive visuals. These are available in the html-version, but not in the PDF version.

The code in the previous section with `getElementById`, `setAttribute`, `appendChild`, etc gets the job done, but it's very verbose. But there are easier ways of doing this. In this tutorial, we will use svelte and sveltekit as our main approach. Sveltekit is a programming framework like React or Vue that provides us with some tools to build websites (and therefore visualisations) more easily. Svelte is a preprocessor that converts code that we write into vanilla javascript. You can compare sveltekit to a restaurant, and svelte to its kitchen. The magic happens in the kitchen, but customers interact with it through the restaurant. **Svelte** is a language and compiler that allows you to create reusable components; **SvelteKit** is a full-stack web application framework built *on top of* Svelte.

> Svelte is a radical new approach to building user interfaces. Whereas traditional frameworks like React and Vue do the bulk of their work in the browser, Svelte shifts that work into a compile step that happens when you build your app. Instead of using techniques like virtual DOM diffing, Svelte writes code that surgically updates the DOM when the state of your app changes.
>
> — svelte.dev

Below, we'll first go over the basics of svelte, and then integrate that into sveltekit.

4.1. HTML, CSS and javascript in svelte

In the previous sections, we generally kept HTML, CSS and javascript in separate files. In svelte, we do not do this. A svelte file therefore consists of 3 parts:

1. a javascript section (`<script></script>`)
2. a CSS section (`<style></style>`)
3. an HTML section (the rest)

Any of these can be omitted if you don't need them.

IMPORTANT	The javascript and CSS only apply to the HTML written in this specific file. This means that different *components* (i.e. files) can be styled independently.

4.2. Using the svelte REPL

To get a quick feel of what Svelte looks like, go to the online REPL (Read-Eval-Print-Loop) at http://svelte.dev/repl.

You can write regular HTML in the "App.svelte" tab, but don't add the `<html>`, `<head>` and `<body>` tags.

A svelte file can have three parts:

- `<script>`
- `<style>`
- the rest is HTML

Change the code in the editor with the code below, and you should see 2 circles and 1 rectangle:

```
<svg width=400 height=400>
  <circle cx=100 cy=100 r=15 />
  <circle cx=150 cy=75 r=20 />
  <rect x=250 y=300 width=30 height=20 />
</svg>
```

4.3. Basics of svelte

The svelte website has a very good tutorial at http://svelte.dev/tutorial. You should definitely go over it and refer back to it when you have questions. We'll highlight loops, conditionals and reactivity in this document, but these are only a small part of svelte's strengths.

4.3.1. Looping over datapoints: {#each}

Svelte helps us to loop over lists in a declarative way. The following code in html gives a bulleted list:

```
<ul>
  <li>John</li>
  <li>Jane</li>
  <li>Joe</li>
```

```
</ul>
```

- John
- Jane
- Joe

In svelte, we can create an array in the `script` section, and use the `#each` pragma to loop over all items. First, we'll create an array called `names` (denoted with the square brackets `[]`) in the `script` section. In the HTML itself, we can loop over them, using the `{#each}` directive (which is closed using `{/each}`). In that loop, each value is put in the temporary variable `name`:

```
<script>
  let names = ["John","Jane","Joe"];
</script>

<ul>
  {#each names as name}
    <li>{name}</li>
  {/each}
</ul>
```

NOTE	You can refer to javascript variables that were defined in the `<script>` section or in the `#each` pragma by putting between curly brackets, e.g. `{name}`.

IMPORTANT	The `{#each}` syntax works only in the HTML part of a svelte file, not for the script part which is regular javascript.

Similarly, instead of hard-coding the datapoints in the SVG, or using the `getElementById`, `appendChild` etc as in the previous section, we have an easier way of looping over datapoints in svelte. First, we'll create an array called `datapoints`, each containing an `x` and `y` value in the `script` section. In the HTML itself, we can loop over them, using the `{#each}` directive (which is closed using `{/each}`).

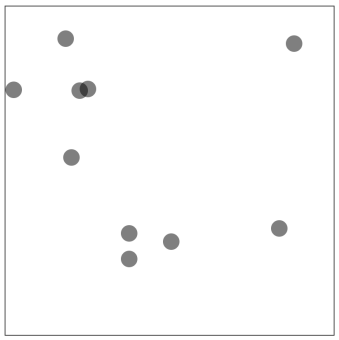
```
<script>
  let datapoints = [{x: 100, y: 100},
                    {x: 150, y: 275},
                    {x: 10, y: 101},
                    {x: 80, y: 183},
                    {x: 350, y: 45},
                    {x: 201, y: 285},
                    {x: 150, y: 306},
                    {x: 90, y: 102},
                    {x: 73, y: 39},
                    {x: 332, y: 269}]
</script>

<style>
```

```
    circle {
        fill-opacity: 0.5;
    }
</style>

<svg width=400 height=400>
  {#each datapoints as dp}
    <circle cx={dp.x} cy={dp.y} r=10 />
  {/each}
</svg>
```

In line 21-23, we loop over the `datapoints` array, each time putting a single element in a local `dp` variable. We can refer to the `x` and `y` properties like we do on line 22.

NOTE	As with regular arrays, you can refer to javascript *objects* that were defined in the `<script>` section or in the `#each` pragma by putting between curly brackets, e.g. `{dp.x}`, and adding a period followed by the property.

The result:



4.3.2. Conditionals: {#if}

Similarly, the `{#if}` directive (in full: `{#if} ⋯ {:else} ⋯ {/if}`) allows you to put conditions in your html. For example, let's create an array of individuals as objects, that contain both a name and a gender.

```
<script>
  let individuals = [
    {"name":"Julia","gender":"F"},
    {"name":"John","gender":"M"},
    {"name":"Joe","gender":"M"},
    {"name":"Jane","gender":"F"}];
</script>

<ul>
    {#each individuals as individual}
        {#if individual.gender == "F"}
            <li>{individual.name} ({individual.gender})</li>
        {/if}
    {/each}
```

```
</ul>
```

This will return:

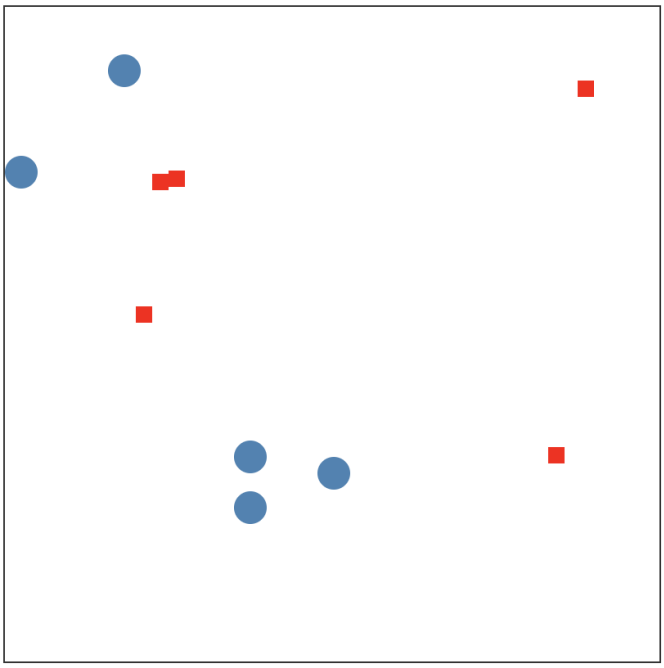
- Julia (F)
- Jane (F)

TIP	Go to the svelte tutorial at http://svelte.dev/tutorial and go through the following sections: "Introduction" and "Logic"

For our scatterplot, let's add a value to all these datapoints, and draw either a blue circle or a red rectangle based on that value.

```
<script>
  let datapoints = [{x: 100, y: 100, value: 9},
                    {x: 150, y: 275, value: 11},
                    {x: 10, y: 101, value: 72},
                    {x: 80, y: 183, value: 2},
                    {x: 350, y: 45, value: 10},
                    {x: 201, y: 285, value: 109},
                    {x: 150, y: 306, value: 24},
                    {x: 90, y: 102, value: -4},
                    {x: 73, y: 39, value: 88},
                    {x: 332, y: 269, value: 8}]
</script>

<style>
  svg {
    background-color: whitesmoke;
  }
  circle {
    fill: steelblue;
  }
  rect {
    fill: red;
  }
</style>

<svg width=400 height=400>
  {#each datapoints as datapoint}
    {#if datapoint.value > 10}
      <circle cx={datapoint.x} cy={datapoint.y} r=10 />
    {:else}
      <rect x={datapoint.x} y={datapoint.y} width=10 height=10 />
    {/if}
  {/each}
</svg>
```

The result:



4.3.3. Reactivity

This is one of the major features of svelte which has an immense effect on programming experience, its *reactivity*. Reactivity means that when some variable `a` depends on a variable `b`, and `b` is changed, that the value of `a` is automatically updated as well. This is what makes a tool like Excel so strong: if you have a cell in a spreadsheet with a formula `=A1*2`, it will have the value of cell A1 multiplied by 2. If you change the value of A1, the value in the derived cell is *automatically* updated as well. Most programming languages do not have this baked in, but with svelte you do have that functionality.

We do this using the `$:` pragma. For example:

```
<script>
  let slider_value = 50;

  $: multiplied_value = slider_value * 2
</script>

<input type="range" min="0" max="100" bind:value={slider_value} />
<p>The value {slider_value} multiplied by 2 is {multiplied_value}.</p>
```

We've seen before that we can use curly brackets `{}` to pass in a value. Here we also need to work in the other direction: when the value of the slider changes, it should be passed through to the script above. We do that using `bind:value`. Sliding left and right will now update the multiplied value as well. You can try it below.

4.4. About sveltekit

4.4.1. Local installation

Although it is extremely useful for quickly checking things, we can't use the REPL for *real* work. Still, you might go back to it regularly to test something out.

Instead, we can develop sveltekit applications (i.c. visualisations) locally, on our own machine. See the sveltekit website on how to get set up. These are the commands you need:

```
npm create svelte@latest my-app
cd my-app
npm install
npm run dev -- --open
```

The first step will create a new directory (called `my-app`) with your application. It will ask you for some information like if you'd want to have an empty (skeleton) setup, or already have demo code included. The `npm run install` installs all dependencies (which are listed in the `package.json` file). Finally, `npm run dev` will start a local webserver so that you can access your application. The output will list which port the application is running on. This will most probably be port 5173, so you should open the website `http://localhost:5173`. If you use `npm run dev -- --open` it will automatically open that website for you.

NOTE	Using `npm run dev` without the `-- --open` works as well. You will however need to open the webpage yourself. This is often the better option if you want to restart the server.

4.4.2. Directory structure and routing

To understand how data can be loaded in sveltekit, we need to understand how routing works. *Routing* maps a file to a URL and vice versa. The directory structure in sveltekit is important: each URL points to a subdirectory of `src/routes`. For example:

- http://localhost:5173 points to the `src/routes/` directory
- http://localhost:5173/about points to the `src/routes/about/` directory
- http://localhost:5173/contact points to the `src/routes/contact/` directory
- ...

Each of these directories should have a `page.svelte` (Notice the `-sign!) file, which contains the actual content of that page.

```
...
|
+- src/
|  +- routes/
|      +- +page.svelte
```

```
|      +- about/
|      |   +- +page.svelte
|      +- contact/
|          +- +page.svelte
|
...
```

See https://kit.svelte.dev/docs/routing for more information.

4.5. Loading data in sveltekit

As we have a `<script>` section in a `.svelte` file, we can define variables and data there, like this:

```
<script>
  let values = [1,2,3,"a string"]
</script>

JSON.stringify(values)
```

In this example, we create a value and show a "stringified" version in the browser.

This works great, except when we have large datasetes. We'll need to load those in a different way. Enter `+page.js`.

If we need to load data before a page (as defined in `+page.svelte`) is rendered, we add a `+page.js` file in the same directory. For example, if the root `index.html` and `contact` need data:

```
...
|
+- src/
|  +- routes/
|      +- +page.svelte
|      +- +page.js            ①
|      +- about/
|      |   +- +page.svelte
|      +- contact/
|          +- +page.svelte
|          +- +page.js        ①
|
...
```

4.5.1. Hard-coding our data

Let's start with the proof-of-principle setup: we hard-code the data to be loaded.

Make the `src/routes/+page.js` look like this:

```
export const load = () => {
  return {
    values: [1,2,3,"a string"]
  }
}
```

And the `src/routes/+page.svelte` look like this:

```
<script>
    export let data;
</script>

{JSON.stringify(data)}
```

You should see `values: [1,2,3,"a string"]` in your web browser at http://localhost:5173.

4.5.2. From an online JSON file

Imagine we need to load the iris dataset, available from a public url (https://raw.githubusercontent.com/domoritz/maps/master/data/iris.json). The data file looks like this:

```
[
  {"sepalLength": 5.1, "sepalWidth": 3.5, "petalLength": 1.4, "petalWidth": 0.2,
"species": "setosa"},
  {"sepalLength": 4.9, "sepalWidth": 3.0, "petalLength": 1.4, "petalWidth": 0.2,
"species": "setosa"},
  {"sepalLength": 4.7, "sepalWidth": 3.2, "petalLength": 1.3, "petalWidth": 0.2,
"species": "setosa"},
  {"sepalLength": 4.6, "sepalWidth": 3.1, "petalLength": 1.5, "petalWidth": 0.2,
"species": "setosa"},
  {"sepalLength": 5.0, "sepalWidth": 3.6, "petalLength": 1.4, "petalWidth": 0.2,
"species": "setosa"},
  {"sepalLength": 5.4, "sepalWidth": 3.9, "petalLength": 1.7, "petalWidth": 0.4,
"species": "setosa"},
  ...
]
```

To load that data, we'd write the following `+page.js`:

```
export const load = ({ fetch }) => {
  const fetchFlowers = async () => { (1)
    const res = await
fetch('https://raw.githubusercontent.com/domoritz/maps/master/data/iris.json') (2)
    const data = await res.json() (3)
    return data
```

```
  }

  return {
    flowers: fetchFlowers() (4)
  }
}
```

What happens here?

- **(1)**: We create an asynchronous function `fetchFlowers`...
- **(2)**: ...that captures the HTTP response into a variable `res`...
- **(3)**: ...from which we extract the `json` part which actually contains the data.

It's interesting to add a `console.log(res)` after line 3 to see what that `res` looks like.

In **(4)** we create the actual return value of the `+page.js` file: it is a *map* with a single key `flowers` and its value coming from `fetchFlowers()`.

To use that data in the `+page.svelte` file, we need to define a `data` variable and get the flowers from it. A simple page showing the sepal length of al flowers would therefore look like this:

```
<script>
  export let data;
</script>

<ul>
  {#each data.flowers as flower}
  <li>{flower.sepalLength}</li>
  {/each}
</ul>
```

NOTE The variable *must* to be called `data`.

Synchronous vs asynchronous programming

In contrast to other languages that you may know (e.g. python and R), javascript is an *asynchronous* language. When you write a program in a synchronous programming language, the program executes instructions in series. This means that each instruction must be completed before moving on to the next one. In contrast, in an asynchronous programming language like javascript, the program can start executing a new instruction before completing the previous one. In some cases we do not want that and actually need to wait until the previous command has finished. To get around it, we can use promises with the `async`/`await` combination.

The `+page.js` example above is a minimal one: you can add additional data transformations. For example, the iris dataset has the following form:

```
[
  {"sepalLength": 5.1, "sepalWidth": 3.5, "petalLength": 1.4, "petalWidth": 0.2,
"species": "setosa"},
  {"sepalLength": 4.9, "sepalWidth": 3.0, "petalLength": 1.4, "petalWidth": 0.2,
"species": "setosa"},
  {"sepalLength": 4.7, "sepalWidth": 3.2, "petalLength": 1.3, "petalWidth": 0.2,
"species": "setosa"},
  {"sepalLength": 4.6, "sepalWidth": 3.1, "petalLength": 1.5, "petalWidth": 0.2,
"species": "setosa"},
  {"sepalLength": 5.0, "sepalWidth": 3.6, "petalLength": 1.4, "petalWidth": 0.2,
"species": "setosa"},
  {"sepalLength": 5.4, "sepalWidth": 3.9, "petalLength": 1.7, "petalWidth": 0.4,
"species": "setosa"},
  ...
]
```

but we would like to add a unique ID to each of these records. Also, we'd like to have the full species name, e.g. "Iris setosa" instead of just "setosa". We can adapt the script above like this:

```
export const load = ({ fetch }) => {
    const fetchFlowers = async () => {
      const res = await
fetch('https://raw.githubusercontent.com/domoritz/maps/master/data/iris.json')
      const data = await res.json()
      data.forEach((d,i) => { d.id = i, d.species = "Iris " + d.species })
      return data
    }

    return {
      flowers: fetchFlowers()
    }
  }
```

The data that is now passed to `+page.svelte` looks like this:

```
[
  {"id": 0, "sepalLength": 5.1, "sepalWidth": 3.5, "petalLength": 1.4, "petalWidth":
0.2, "species": "Iris setosa"},
  {"id": 1, "sepalLength": 4.9, "sepalWidth": 3.0, "petalLength": 1.4, "petalWidth":
0.2, "species": "Iris setosa"},
  {"id": 2, "sepalLength": 4.7, "sepalWidth": 3.2, "petalLength": 1.3, "petalWidth":
0.2, "species": "Iris setosa"},
  {"id": 3, "sepalLength": 4.6, "sepalWidth": 3.1, "petalLength": 1.5, "petalWidth":
0.2, "species": "Iris setosa"},
  {"id": 4, "sepalLength": 5.0, "sepalWidth": 3.6, "petalLength": 1.4, "petalWidth":
0.2, "species": "Iris setosa"},
  {"id": 5, "sepalLength": 5.4, "sepalWidth": 3.9, "petalLength": 1.7, "petalWidth":
0.4, "species": "Iris setosa"},
```

```
  ...
]
```

4.5.3. From an online CSV file

In contrast to JSON, `fetch` is not able to automatically parse a CSV file. We'll have to do that ourselves. We have to install the PapaParse npm package. To do so:

- Stop the `npm run dev` server.
- Run `npm install papaparse` in the root folder of your svelte application.
- Restart `npm run dev`.

Here's a working example using data about flights. The file looks like this:

```
from_airport,from_city,from_country,from_long,from_lat,to_airport,to_city,to_country,to_long,to_lat,airline,airline_country,distance
Balandino,Chelyabinsk,Russia,61.838,55.509,Domododevo,Moscow,Russia,38.51,55.681,Aerocondor,Portugal,1458
Balandino,Chelyabinsk,Russia,61.838,55.509,Kazan,Kazan,Russia,49.464,56.01,Aerocondor,Portugal,775
Balandino,Chelyabinsk,Russia,61.838,55.509,Tolmachevo,Novosibirsk,Russia,83.084,55.021,Aerocondor,Portugal,1341
Domododevo,Moscow,Russia,38.51,55.681,Balandino,Chelyabinsk,Russia,61.838,55.509,Aerocondor,Portugal,1458
...
```

```
import Papa from 'papaparse'

export const load = ({ fetch }) => {
  const fetchFlights = async () => {
    const res = await fetch('https://vda-lab.gitlab.io/datavis-technologies/assets/flights_part.csv', {
      headers: {
        'Content-Type': 'text/csv'
    }})
    let csv_data = await res.text()
    let csv_parsed = Papa.parse(csv_data, {header: true})

    return csv_parsed.data
  }

  return {
    flights: fetchFlights()
  }
}
```

If you get an error in the console that mentions the CORS policy (`No 'Access-Control-Allow-Origin'`

`header is present`), try using `https` instead of `http` in the URL for the dataset.

Let's walk over this code:

- line 1: import the PapaParse package
- line 6-8: we have to add the `'Content-Type': 'text/csv'` to what is returned because your browser would otherwise try to download the file instead of using it in our application
- line 9: In the JSON example before, we got the actual data through `res.json()`. Here we need it as a text: `res.text()`
- line 10: Finally, we need to *parse* the text to actual values. This will return an object where those values are available under the `data` key, which we extract on line 12.

You can always add a `console.log()` for `csv_data` or `csv_parsed` to see what those variables look like.

We get the following output:

```
Balandino (Chelyabinsk)
Balandino (Chelyabinsk)
Domododevo (Moscow)
Domododevo (Moscow)
Domododevo (Moscow)
Domododevo (Moscow)
Domododevo (Moscow)
Domododevo (Moscow)
Heydar Aliyev (Baku)
Khrabrovo (Kaliningrad)
Kazan (Kazan)
Kazan (Kazan)
Kazan (Kazan)
Kazan (Kazan)
Kazan (Kazan)
Pulkovo (St. Petersburg)
Pulkovo (St. Petersburg)
Pulkovo (St. Petersburg)
Franz Josef Strauss (Munich)
```

4.5.4. From an local JSON or CSV file

The above CSV and JSON files are on a remote server. But what if we have the data on our own machine? Actually, this is very simple as we are running our own server. If you put the data file in the `static` directory of your svelte project, you can access it directly, e.g. with

- `Papa.parse('http://localhost:5173/airports.csv', { ⋯ })`, or
- `fetch('http://localhost:5173/airports.json')`

4.5.5. From an SQL database

Data stored in local SQLite3 database

Let's say we have a small database with employee data. It only has one table, `employees`, with the following columns: `name` and `firstname`.

To load data from this SQLite3 database, we'll use the knex.js query builder. It'll make it easier to switch between different types of SQL databases later (mysql, postgresql, etc). To get it to work with a local sqlite3 database, install `knex` and the necessary sqlite driver:

```
npm install knex --save
npm install better-sqlite3 --save
```

Create a new file `src/lib/db.js` with the following contents:

```
import knex from 'knex'

export default knex({
    client: 'better-sqlite3',
    connection: {
        filename: "./static/test.db"
    },
  })
```

The path is relative to the main directory in sveltekit.

Above, we have used the `+page.js` file to load our data from CSV sources. For loading SQL data, we will however need to name our file `+page.server.js` (see https://kit.svelte.dev/docs/routing#page-page-server-js for details).

Make the `+page.server.js` file look like this:

```
import db from '$lib/db';

export const load = () => {
    const fetchData = async () => {
        let employees = await db.select('*')
                                .from('employees')
                                .limit(2)
        return employees
    }

    return {
        employees: fetchData()
    }
}
```

We can then access these in our `+page.svelte`:

```
<script>
    export let data = [];
</script>

<h1>Employees</h1>

<ul>
{#each data.employees as employee}
    <li>{employee.firstname} {employee.name}</li>
{/each}
</ul>
```

Data stored in mysql database

Loading data from a mysql database is very similar, although we will run into a small bump here.

Say we want to access data from `knownGene` table of the UCSC Genome database (http://genome-euro.ucsc.edu/).

Replace the `lib/db.js` file with the following.

```
import knex from 'knex'

export default knex({
  client: 'mysql',
  version: '5.7',
  connection: {
    host: 'genome-euro-mysql.soe.ucsc.edu',
    port: 3306,
    user: 'genome',
    password: '',
    database: 'hg38'
  },
})
```

Based on our experience with sqlite3, the `+page.server.js` file would look like this:

```
import db from '$lib/db';

export const load = () => {
    const fetchData = async () => {
        let genes = await db.select('*')
                            .from('knownGene')
                            .limit(20)
        return genes
    }
```

```
    return {
        genes: fetchData()
    }
}
```

Unfortunately, we get an error `Error: Data returned from load while rendering / is not serializable: Cannot stringify arbitrary non-POJOs (data.genes[0])`. If we add a `console.log(genes)` in our `+page.server.js` file, we see that what is returned from the server is the following:

```
[
  RowDataPacket {
    name: 'ENST00000456328.2',
    chrom: 'chr1',
    strand: '+',
    ...
  },
  RowDataPacket {
    name: 'ENST00000619216.1',
    chrom: 'chr1',
    strand: '-',
    ...
  }
]
```

However, we expected the output to look like this:

```
[
  {
    name: 'ENST00000456328.2',
    chrom: 'chr1',
    strand: '+',
    ...
  },
  {
    name: 'ENST00000619216.1',
    chrom: 'chr1',
    strand: '-',
    ...
  }
]
```

We can accomplish this to parse the output first before we return it, using `return JSON.parse(JSON.stringify(genes))` instead of just `return genes` (source: https://stackoverflow.com/questions/31221980/how-to-access-a-rowdatapacket-object):

```
import db from '$lib/db';

export const load = () => {
    const fetchData = async () => {
        let genes = []
        genes = await db.select('*')
                                .from('knownGene')
                                .limit(20)
        return JSON.parse(JSON.stringify(genes))
    }

    return {
        genes: fetchData()
    }
}
```

4.5.6. Loading multiple datasets

Above we only loaded a single dataset, but obviously we will sometimes need multiple datasets in our application. To do this we just add an additional function to the `load` function in our `+page.js`. For example:

```
export const load = ({ fetch }) => {
  const fetchGenes = async () => {
    const res = await fetch("https://some-url/genes.json")
    const data = await res.json()
    return data
  }

  const fetchProteins = async () => {
    const res = await fetch("https://some-other-url/proteins.json")
    const data = await res.json()
    return data
  }

  return {
    genes: fetchGenes(),
    proteins: fetchProteins()
  }
```

As before, the `data` variable in `+page.svelte` is the object that is returned by `+page.js` (i.c. `{genes: [⋯,⋯,⋯], proteins: [⋯,⋯,⋯]}`)

4.6. Data subpages and slugs

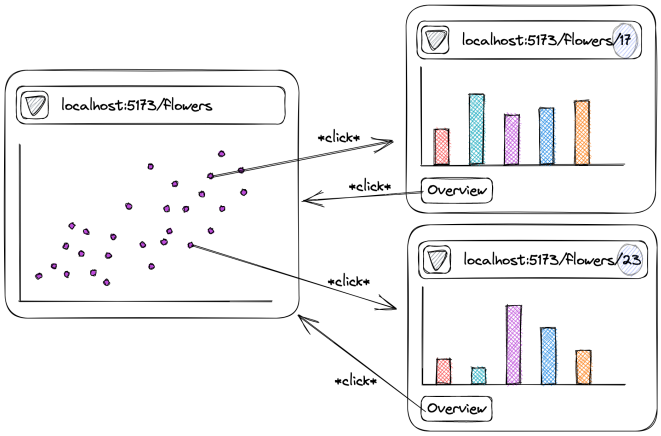
In many cases, we might want to have a subpage for each datapoint. Imagine a blog with many posts, where those posts are stored as a large JSON structure. From what we've seen above, we are

able to create a page that lists the title of all blog posts. But we're still unable to show a single blog *post*. Similarly, we might have a site that shows a list of genes, but you cannot go to the information of a *single* gene.

Let's say we want to create a page for every single flower, using its ID in the URL. For example, to get the information (or a visualisation) for flower 5, we would use the URL http://localhost:5173/flowers/5. This is the same URL as before for the list, but adding the flower ID. To get this to work, we use `[slug]`: create a new directory under the flowers folder:

```
...
|
+- src/
|  +- routes/
|      +- +page.svelte
|      +- flowers/
|          +- +page.svelte
|          +- +page.js
|          +- [slug]
|              +- +page.svelte
|              +- +page.js
|
...
```

IMPORTANT The directory name must be `[slug]`, including the square brackets!



What this `[slug]` allows you to do, is add *parameters* to the URL. In our case, this could for example be the ID of a flower. The `+page.js` in the `[slug]` directory can be very similar to the one in the `flowers` directory, but there are some important differences:

```
export const load = ({ fetch, params }) => { (1)
  const fetchFlower = async () => {
    const res = await
fetch('https://raw.githubusercontent.com/domoritz/maps/master/data/iris.json')
    const data = await res.json()
    data.forEach((d,i) => { d.id = i })
    let data_for_slug = data.filter((d) => { return d.id == params.slug})[0] (2)
```

```
    return data_for_slug
  }

  return {
    flower: fetchFlower() (4)
  }
}
```

In this code, we

- **(1)** call `fetch` *and* `params` in the loading function instead of only `fetch`
- **(2)** filter the returned data based on the id of the flower `d.id == params.slug`

In the `+page.svelte` file we can then display or visualise only the information for that single flower.

NOTE	See https://kit.svelte.dev/docs/load for the full documentation on how to load data in sveltekit.

4.7. Our first real scatterplot

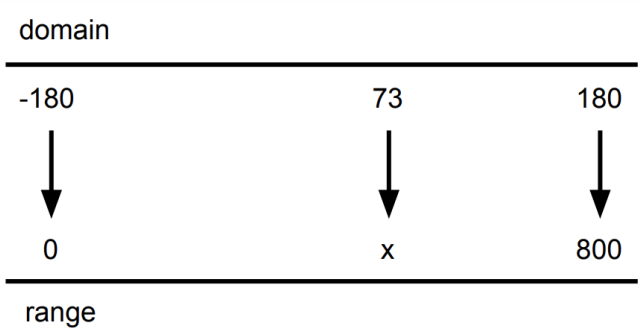
Now - finally - we can start working on the real thing and create a data visualisation. Let's plot the longitude and latitude (present as `long` and `lat` in the datafile) of all departure airports in http://vda-lab.gitlab.io/datavis-technologies/assets/flights_part.json. If we do this, we should get something that looks like a map of the world.

Load the data using `+page.js` as described above. Below we show the contents of the `+page.svelte` file.

```
<script>
  export let data = [];
</script>

<style>
  svg {
      border: 1px;
      border-style: solid;
  }
  circle {
      fill: steelblue;
      fill-opacity: 0.5;
  }
</style>

<svg width="800" height="400">
  {#each data.flights as datapoint}
      <circle cx={datapoint.from_long} cy={datapoint.from_lat} r="2" />
  {/each}
</svg>
```

The resulting image looks like this:

This is clearly *not* what we expected. The reason is simple: the longitude in the data file ranges from -180 until 180 and the latitude is between -90 and 90. If we plot these directly as circles than 3/4 of all datapoints will be outside of the SVG (because they have either a negative longitude or latitude). Instead of using `cx={datapoint.from_long}` we have to rescale that longitude from its original range (called its *domain*) to a new *range*.



The formula to do this is:

$$\frac{(range_{max} - range_{min}) * (x - domain_{min})}{domain_{max} - domain_{min}} + range_{min}$$

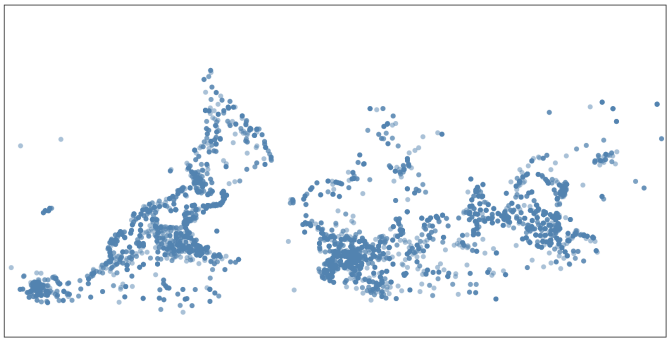
Let's put that in a function that we can use. Add the `rescale` function to the `script` section of your svelte file, and call it where we need to set `cx` and `cy`.

```
<script>
  export let data = [];

  const rescale = function(x, domain_min, domain_max, range_min, range_max) {
      return ((range_max - range_min)*(x-domain_min))/(domain_max-domain_min) +
range_min
  }
</script>

<style>
  circle {
      fill: steelblue;
      fill-opacity: 0.5;
  }
</style>
```

```
<svg width="800" height="400">
  {#each data.flights as datapoint}
      <circle cx={rescale(datapoint.from_long, -180, 180, 0, 800)}
              cy={rescale(datapoint.from_lat, -90, 90, 0, 400)}
              r=3 />
  {/each}
</svg>
```

Our rescaling function is defined on lines 7-9 and used on lines 21 and 22.

The result:



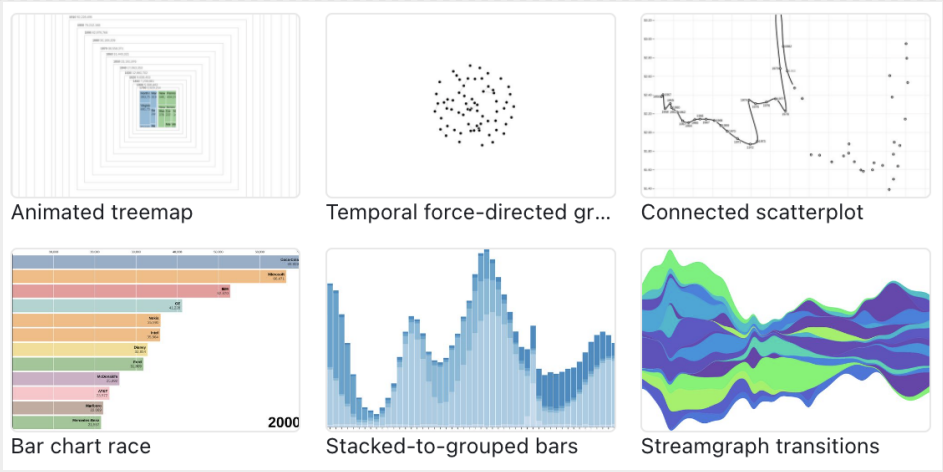
This is better, but the world is upside down. This is because the origin [0,0] in SVG is in the top left, not the bottom left. We therefore have to flip the scale as well, and set the range to `400,0` instead of `0,400` for `cy`. If we do that we'll get the world the right side up.

4.8. D3 scales

In the example above, we have written our own code for rescaling the longitude (-180 to 180) and latitude (-90 to 90) to width (0 to 800) and height (400 to 0). We can however also use the powerful functionality provided by D3.

D3 - Data-Driven Documents

D3 (Data-Driven Documents) has been the go-to library for data visualisation for many years. It allows you to create very complex and interactive visuals like showcased in the D3 gallery.



The functionality of D3 has been split in different modules (see here), that cover for example the creation of hexagonal bins, working with geographic projections, creating force-directed graphs, and scaling.

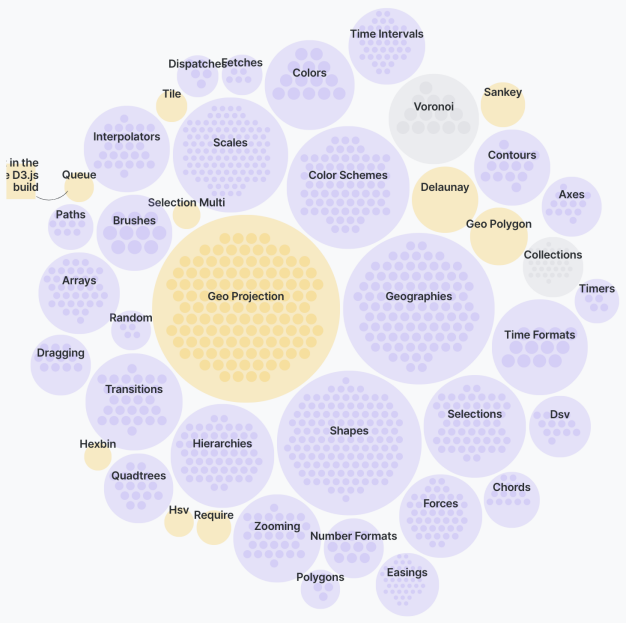
Although very powerful, this library does have a steep learning curve. For example, creating a scatterplot like the one above using D3, you'd write

```
d3.select("#my_svg")
  .append("g")
  .selectAll("circle")
  .data(datapoints)
  .enter()
  .append("circle")
    .attr("cx", function(d) { return rescale(datapoint.from_long, -180, 180, 0,
800) })
    .attr("cy", function(d) { return rescale(datapoint.from_lat, -90, 90, 400, 0)
})
    .attr("r", 3)
```

That is why we focus on using svelte in this tutorial for the main work, and use D3 modules when we need them for a specific tasks (e.g. scaling). For example, this blog post by Connor Rothshield also goes into why he switched from D3 to svelte+D3 for data visualisation: https://www.connorrothschild.com/post/svelte-and-d3

D3 is organised as a group of modules (see https://github.com/d3/d3/blob/main/API.md), so we can choose to load only those functions that have to do with scaling (`d3-scale`), colour (`d3-color`), etc.

Here is an overview of the different modules:



(Source: https://wattenberger.com/blog/d3) For an interactive version, see https://wattenberger.com/blog/d3.

Let's replace our own rescaling function with a linear scale provided by D3. We will load the `scaleLinear` function from `d3-scale`.

IMPORTANT	We have to install the `d3-scale` module first. Do this by running `npm install d3-scale` on the command line.

```
<script>
  import { scaleLinear } from 'd3-scale'; ①

  export let data = [];

  const scaleX = scaleLinear().domain([-180,180]).range([0,800]) ②
  const scaleY = scaleLinear().domain([-90,90]).range([400,0]) ②
</script>

<style>
  circle {
      fill: steelblue;
      fill-opacity: 0.5;
  }
</style>

<svg width="800" height="400">
  {#each data.flights as datapoint}
      <circle cx={scaleX(datapoint.from_long)} ③
              cy={scaleY(datapoint.from_lat)}
              r=3 />
  {/each}
```

```
</svg>
```

In **(1)** we load `scaleLinear` and make the function available in our code. We define a `scaleX` and a `scaleY` function in **(2)**. The `domain` refers to the actual data, and `range` to the projection (in this case: pixel position). In **(3)** we use `scaleX` and `scaleY`.

Note that the range does not have to be numeric: we can also use colours here. D3 is clever enough to interpolate colours across the range. In the example below, we let the colour of the points go from red to green along with the longitude. (If we had information on the altitude of the airports, this would be more useful.)

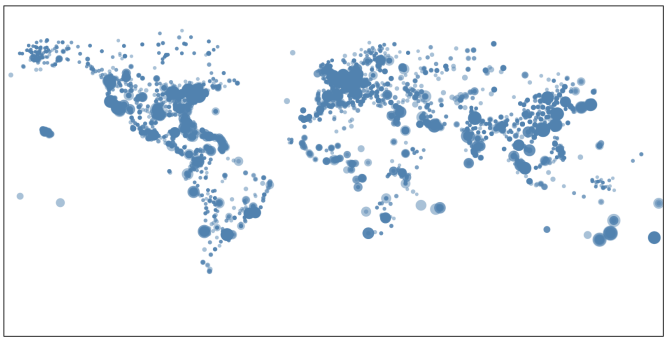
```
<script>
  import { scaleLinear } from 'd3-scale';

  export let data = [];

  const scaleX = scaleLinear().domain([-180,180]).range([0,800])
  const scaleY = scaleLinear().domain([-90,90]).range([400,0])
  const scaleColour = scaleLinear().domain([-180,180]).range(["red","green"]) ①
</script>

<style>
  circle { ②
      fill-opacity: 0.5;
  }
</style>

<svg width="800" height="400">
  {#each data.flights as datapoint}
      <circle cx={scaleX(datapoint.from_long)}
              cy={scaleY(datapoint.from_lat)}
              r=3
              style={"fill:" + scaleColour(datapoint.from_long)} /> ③
  {/each}
</svg>
```

We added a scale with colours as the range in **(1)**, remove the default colour for a circle in **(2)**, and set the CSS colour dynamically in **(3)** using.

D3 provides a lot of other scales as well, including logarithmic, time, radial etc. Check out https://github.com/d3/d3-scale for more information.

Let's add another scale: we can let the size of the point be dependent of the `distance` in the csv file

```
<script>
  import { scaleLinear } from 'd3-scale';

  export let data = [];

  const scaleX = scaleLinear().domain([-180,180]).range([0,800])
  const scaleY = scaleLinear().domain([-90,90]).range([400,0])
  const scaleRadius = scaleLinear().domain([1,15406]).range([2,10])
  </script>

<style>
  svg {
    border: 1px;
    border-style: solid;
  }
  circle {
    fill: steelblue;
    fill-opacity: 0.5;
  }
</style>

<svg width="800" height="400">
  {#each data.flights as datapoint}
    <circle cx={scaleX(datapoint.from_long)}
        cy={scaleY(datapoint.from_lat)}
        r={scaleRadius(datapoint.distance)} />
  {/each}
</svg>
```



4.9. Classes

Above we have used a colour scale that ranges from red to green according to longitude. If we want to handle categorical aspects in the data (e.g. if a flight is international or domestic), we can actually do this easier, using HTML classes. For an overview, see the "HTML, CSS and javascript" section of

this tutorial. (This will become very important once we start looking into brushing and linking in the next section.)

If we change the code above by (a) adding a `circle.international` in the CSS that sets the fill colour to red, and (b) add a `class="international"` as a property of the `circle` element, all the airports will be red. But can we actually make this dependent on the actual data?

We give an HTML element one or more classes like so:

```
<circle class="first_class second_class third_class" />
```

Using svelte, we can do this dynamically. To know if a flight is international, we can check if its `from_country` is different from its `to_country`.

```
<circle class={datapoint.from_country != datapoint.to_country ? 'international' :
'domestic' } />
```

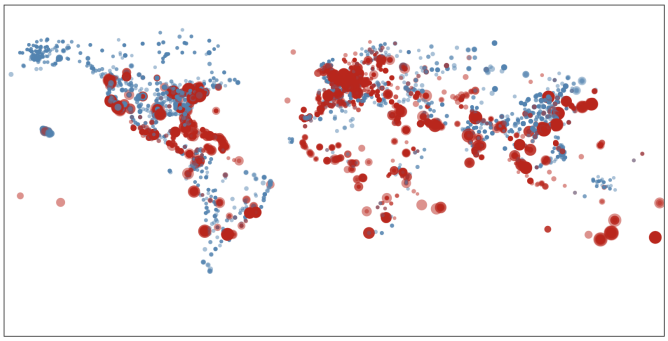
Or in its shorthand version:

```
<circle class:international={datapoint.from_country != datapoint.to_country} />
```

```
<script>
  import { scaleLinear } from 'd3-scale';

  export let data = [];

  const scaleX = scaleLinear().domain([-180,180]).range([0,800])
  const scaleY = scaleLinear().domain([-90,90]).range([400,0])
  const scaleRadius = scaleLinear().domain([1,15406]).range([2,10])
  </script>

<style>
  svg {
    border: 1px;
    border-style: solid;
  }
  circle {
    fill: steelblue;
    fill-opacity: 0.5;
  }
  circle.international { ①
    fill: red;
  }
</style>

<svg width="800" height="400">
  {#each data.flights as datapoint}
```

```
    <circle cx={scaleX(datapoint.from_long)}
        cy={scaleY(datapoint.from_lat)}
        r={scaleRadius(datapoint.distance)}
        class:international={datapoint.from_country != datapoint.to_country}/> ②
  {/each}
</svg>
```

(1) is where we set the colour of international flights; **(2)** is where we apply it.



4.10. Exercises

Here are some exercises related to this chapter:

- Svelte markup: https://svelte.dev/repl/724a8216d6c84491b7b04951718f0b0d?version=3.59.1
- Foreach: https://svelte.dev/repl/161b76456d00443db6100f7d40e546b1?version=3.59.1
- If-else-then: https://svelte.dev/repl/a603bddd25ed441e9680e2c93a1a1966?version=3.59.1
- Scales: https://svelte.dev/repl/f2cdcb8400f2430f9134206798596a97?version=3.59.1
- Colour scales: https://svelte.dev/repl/f555af12649a4d918db9f46c88ea72a0?version=3.59.1
- Axes: https://svelte.dev/repl/a06224b0183c4f44b6de3f7a734e812e?version=3.59.1
- Paths using d3.line: https://svelte.dev/repl/ec5b4a3992e7459086668fe4e03c011a?version=3.59.1
- Hover: https://svelte.dev/repl/c602355ed2cf4398921f50f7746079ff?version=3.59.1
- Working with objects: https://svelte.dev/repl/08e9e88a020244d5a7cd80b2e0befa3b?version=3.59.1
- Extent: https://svelte.dev/repl/b31541e1dcfb439e818c639427e6db68?version=3.59.1
- Make scales: https://svelte.dev/repl/c6eb38ca440a4dc5b8efa487f92b771d?version=3.59.1
- Make scatterplot: https://svelte.dev/repl/4c447e06f79e4c478682b8476bf1833a?version=3.59.1
- Add axes: https://svelte.dev/repl/3e6b6e947bdb480687d29392d944e15c?version=3.59.1
- Set circle colour: https://svelte.dev/repl/87536213d7044ddc9ed2dfacb208086c?version=3.59.1

Chapter 5. Custom components

NOTE	This section contains several interactive visuals. These are available in the html-version, but not in the PDF version.

What if we want to create two scatterplots instead of one, e.g. one for departure airports and one for arrival airports? We could duplicate the code for the SVG, like so:

```
<svg width="800" height="400">
  {#each datapoints as datapoint}
    <circle cx={scaleX(datapoint.from_long)}
        cy={scaleY(datapoint.from_lat)}
        r={scaleRadius(datapoint.distance)}
        class:international={datapoint.from_country != datapoint.to_country}/> (2)
  {/each}
</svg>

<svg width="800" height="400">
  {#each datapoints as datapoint}
    <circle cx={scaleX(datapoint.to_long)}
        cy={scaleY(datapoint.to_lat)}
        r={scaleRadius(datapoint.distance)}
        class:international={datapoint.from_country != datapoint.to_country}/> (2)
  {/each}
</svg>
```

It would be nicer if we could something like this instead:

```
<Scatterplot which="from" />
<Scatterplot which="to" />
```

This is where we really start to get into the strengths of svelte, because it is a framework to create *custom HTML components*; in this case, a `Scatterplot` component.

5.1. Proof of principle using the Svelte REPL

As a proof of principle, we'll create a new component by clicking the big `+` sign next to `App.svelte`. Give this component the name `MyComponent.svelte` (check the capitalisation!).

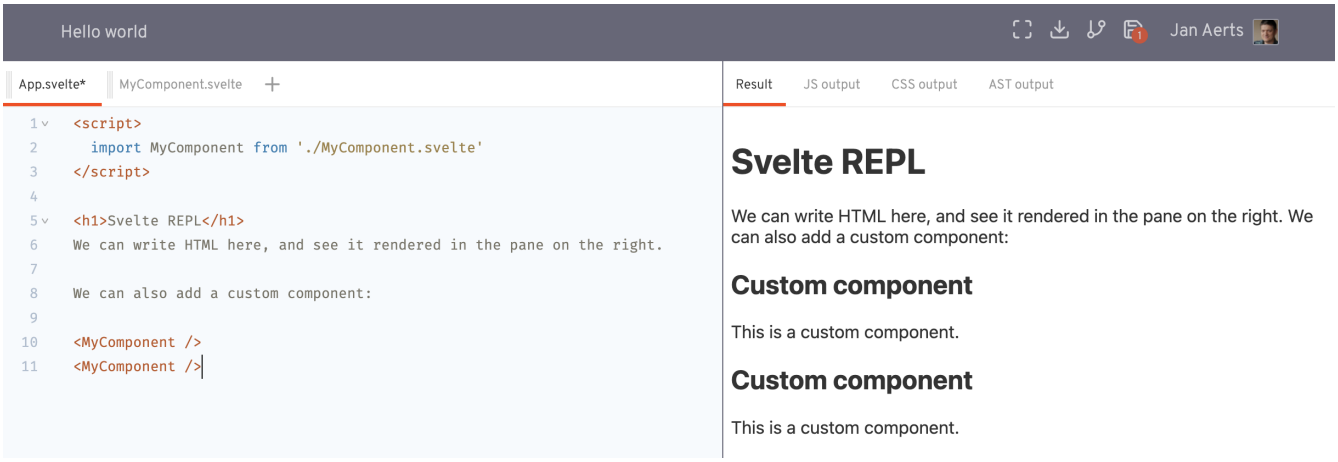
In this new component, add the following code:

```
<h2>Custom component</h2>
This is a custom component.
```

In the `App.svelte`, write

```
<script>
    import MyComponent from './MyComponent.svelte'
</script>

<h1>Svelte REPL</h1>
We can write HTML here, and see it rendered in the pane on the right.

We can also add a custom components:

<MyComponent />
<MyComponent />
```

Your output will look like this:



In the `<script>` section, we load the custom `MyComponent` element, and we use that in the HTML part of `App.svelte`. As you can see, we can call this element more than once.

5.2. Converting our airports map to a component

Create a new subfolder in `src` named `components`, and create a new file named `Scatterplot.svelte`. We'll move everything that is relevant to the scatterplot itself to this file:

```
<script>
  export let datapoints = []; ①

  const rescale = function(x, domain_min, domain_max, range_min, range_max) {
    return ((range_max - range_min)*(x-domain_min))/(domain_max-domain_min) +
range_min
  }
</script>

<style>
  circle {
    opacity: 0.5;
    fill: blue;
  }
```

```
  circle.international {
    fill: red;
  }
</style>

<svg width=800 height=400>
  {#each datapoints as datapoint}
    <circle cx={rescale(datapoint.from_long, -180, 180, 0, 800)}
            cy={rescale(datapoint.from_lat, -90, 90, 400, 0)}
            r={rescale(datapoint.distance, 1, 15406, 2,10)}
            class:international="{datapoint.from_country != datapoint.to_country}" />
  {/each}
</svg>
```

At **(1)**, this component defines a `datapoints` variable. Because of the `export let` instead of just `let` we can access this variable from outside. Now how do we do that? We have moved all scatterplot specific code from `src/routes/+page.svelte` into this new component. We rewrite `src/routes/+page.svelte` to look like this:

```
<script>
    import Scatterplot from './Scatterplot.svelte'; ①
    let datapoints_from_app = [] ②
    fetch("https://vda-lab.gitlab.io/datavis-technologies/assets/flights_part.json")
        .then(res => res.json())
        .then(data => datapoints_from_app = data)
</script>

<h1>Airports</h1>
<Scatterplot datapoints={datapoints_from_app}/>  ③
```

We first import this new `Scatterplot` element that we created **(1)**. We still have to load the data using the fetch API, but now we use the `Scatterplot` element instead of the original SVG **(3)**. Just to make things a bit more clear, we have renamed the original `datapoints` to `datapoints_from_app` **(2)**. The `Scatterplot` element takes a `datapoints` attribute. This attribute exists because we defined the `export let datapoints` in the component. The value of that `datapoints` variable is what comes from `datapoints_from_app`.

CAUTION	Components that we create ourselves (like `Scatterplot`) must be capitalised: `scatterplot` will not work. This in contrast to the regular HTML elements (`h1`, `div`,...).

In our new `Scatterplot` component, the fact that we use the departure airports (`from_long` and `from_lat`) is hard-coded. We can change our code so that we can pass this as an argument. In the code below, we `export` two new variables (`long` and `lat`) that we default to the departure airport.

`Scatterplot.svelte`:

```
<script>
```

```
  export let datapoints = [];
  export let long = 'from_long'; ①
  export let lat = 'from_lat';

  const rescale = function(x, domain_min, domain_max, range_min, range_max) {
    return ((range_max - range_min)*(x-domain_min))/(domain_max-domain_min) +
range_min
  }
</script>

<style>
  circle {
    opacity: 0.5;
    fill: blue;
  }
  circle.international {
    fill: red;
  }
</style>

<svg width=1000 height=500>
  {#each datapoints as datapoint}
    <circle cx={rescale(datapoint[long], -180, 180, 0, 800)} ②
            cy={rescale(datapoint[lat], -90, 90, 400, 0)}
            r={rescale(datapoint.distance, 1, 15406, 2,10)}
            class:international="{datapoint.from_country != datapoint.to_country}" />
  {/each}
</svg>
```

src/routes/+page.svelte:

```
<script>
    import Scatterplot from './Scatterplot.svelte'
    let datapoints_from_app = []
    fetch("https://vda-lab.github.io/assets/svelte-flights.json")
        .then(res => res.json())
        .then(data => datapoints_from_app = data.slice(1,5000))
</script>

<h1>Airports</h1>
<Scatterplot datapoints={datapoints_from_app} long="from_long" lat="from_lat" />  ③
<Scatterplot datapoints={datapoints_from_app} long="to_long" lat="to_lat" />
```

In the Scatterplot.svelte component, we add the new variables **(1)**, and change the hard-coded call to datapoint.from_long to one where we use these variables (datapoint[long]) **(2)**. Finally, we use this by adding the new argument when create a scatterplot in src/routes/+page.svelte **(3)**.

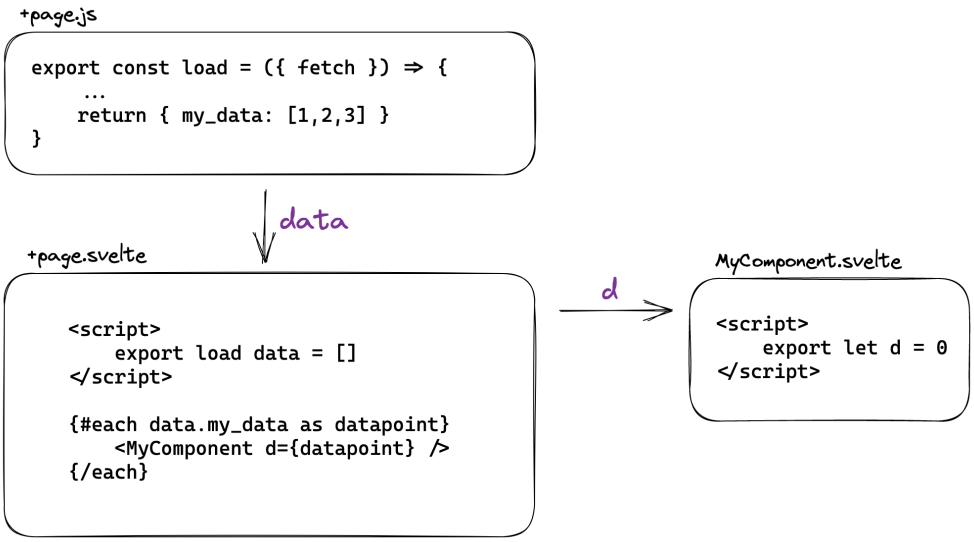
As an exercise, you can create a more generic scatterplot component that is not specific for plotting longitudes and latitudes.

5.3. Passing data around

In the example above, we passed data from `src/routes/+page.svelte` to `Scatterplot.svelte`. There are different ways of making data available between components (i.e. different `+page.svelte` pages). Mainly, these are:

- to use a svelte *store*, but we won't go into this here.
- to pass them as arguments between components.

It's the second solution that we are using. Here's an overview:



5.4. Custom visuals

We can take this further and create real custom visuals. Here, we'll use the well-known iris dataset, listing the size of sepals and petals as well as the species for 150 flowers. These flowers have quite distinct sepals and petals, as you can see in this picture:

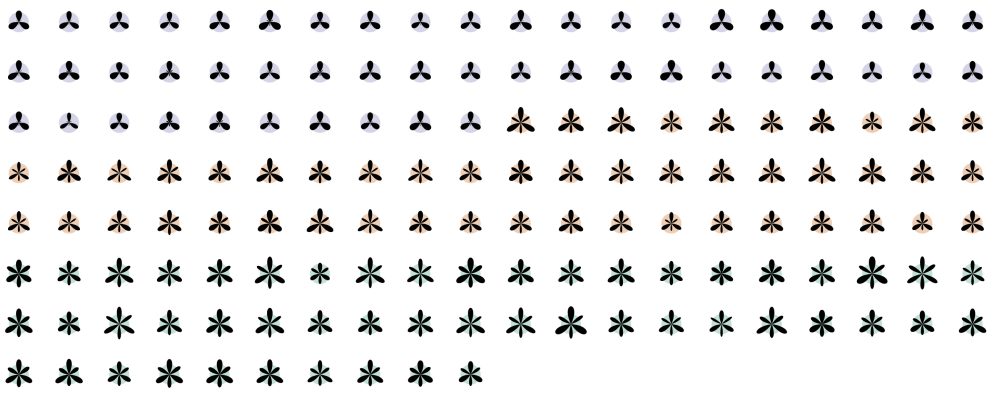
Source: www.plant-world-seeds.com

The data is available from https://vda-lab.github.io/assets/iris.csv and looks like this:

```
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,setosa
4.9,3.0,1.4,0.2,setosa
4.7,3.2,1.3,0.2,setosa
```

```
4.6,3.1,1.5,0.2,setosa
5.0,3.6,1.4,0.2,setosa
5.4,3.9,1.7,0.4,setosa
4.6,3.4,1.4,0.3,setosa
5.0,3.4,1.5,0.2,setosa
4.4,2.9,1.4,0.2,setosa
...
```

It should be simple for you now to create a scatterplot showing either length and width of the sepals, or those of the petals. We'll go further here, and create a small multiple view like the one below:



A single flower looks like this:

The sepals and petals are the big and small parts of the flower, respectively. We also add a circle behind it with a colour that corresponds to the species.

Let's first create the `Flower.svelte` component itself:

```
<script>
  // A datapoint looks like this:
  // { petal_length: 1.4,
  //   petal_width: 0.2,
  //   sepal_length: 5.1,
  //   sepal_width: 3.5,
  //   species: "setosa" }

  export let datapoint = {}
```

```
  let scale = 3;                                                    ①
  $: sl = scale*datapoint.sepal_length
  $: sw = scale*datapoint.sepal_width
  $: pl = scale*datapoint.petal_length
  $: pw = scale*datapoint.petal_width
  $: sepal_path = "M 0,0 " +
                  "C " + sl + ",-" + sw +
                   " " + sl + "," + sw +
                   " 0,0 Z"                                         ②
  $: petal_path = "M 0,0 " +
                  "C " + pl + ",-" + pw +
                  " " + pl + "," + pw +
                  " 0,0 Z"
</script>

<style>
  circle.setosa {
    fill: #7570b3;
    fill-opacity: 0.3;
  }
  circle.virginica {
    fill: #1b9e77;
    fill-opacity: 0.3;
  }
  circle.versicolor {
    fill: #d95f02;
    fill-opacity: 0.3;
  }
</style>

<g>                                                                 ③
  <circle cx=0 cy=0 r=10 class={datapoint.species} />               ④
  <path style="transform: rotate(270deg)" d={sepal_path} />         ⑤
  <path style="transform: rotate(30deg)" d={sepal_path} />
  <path style="transform: rotate(150deg)" d={sepal_path} />
  <path style="transform: rotate(325deg)" d={petal_path} />
  <path style="transform: rotate(90deg)" d={petal_path} />
  <path style="transform: rotate(210deg)" d={petal_path} />
</g>
```

The following things are noteworthy:

- **(1)** The scale we define here is just a hack-y way for getting images that I find good in size. As an exercise, replace this hard-coded scale with a slider.
- **(2)** Here we define the `path` to draw a single petal or sepal. See Curves for how these are defined.
- **(3)** Because none of these petals or sepals stand on their own but are part of a single glyph, it is good practive to group them in a `<g>` element.
- **(4)** The circle that is at the back of each flower and is coloured according to species.

- **(5)** Each of the parts of the flower is drawn in the same way (see **(2)**): we start the path on position `(0,0)` and draw to the right. For each of the parts we then rotate it. You can check the effect by removing the style attribute.

In our `src/routes/+page.svelte` we will load this data.

```
<script>
    import Papa from 'papaparse';
    import Flower from './Flower.svelte';

    let datapoints = []

    Papa.parse("https://vda-lab.github.io/assets/iris.csv", {
        header: true,
        download: true,
        complete: function(results) {
            datapoints = results.data
        }
    })

    const get_xy = function(idx) {                                    ①
        let y = 25 + (Math.floor(idx / 20) * 50)
        let x = 25 + ((idx % 20) * 50)
        return [x,y]
    }
    $: console.log(datapoints)                                        ②
</script>

<svg width=1000 height=1000>
    {#each datapoints as datapoint,idx}                               ③
        <g transform="translate({get_xy(idx)[0]}, {get_xy(idx)[1]})"> ④
            <Flower datapoint={datapoint} />
        </g>
    {/each}
</svg>
```

At **(1)** we create a function that returns an x and y offset for a given index (see **(3)**). There are better ways of doing this (e.g. using Bootstrap), but we just code our own here, creating rows of 20 flowers. **(2)** shows how we can use `console.log` to make sure that the datapoints we load are actually what we expected. This is a typical way in svelte to check what's going on: because we start the command with a `$:`, this will run every time the value for `datapoints` changes. In **(3)** we loop over all datapoints, but we return both the single datapoint *and* its index (which we call `idx`). This index is what will be used by `get_xy` to calculate the position on the screen. Finally, we transform the flower that is drawn in **(4)** to put it in the correct position.

5.5. Exercises

Here are some exercises related to this chapter:

- Components: https://svelte.dev/repl/fce70af77dd44da7a568576e1d02eb40?version=3.59.1
- Communication between components: https://svelte.dev/repl/2ccb36183e504526b3c3c8df5a3310f6?version=3.59.1

Chapter 6. Advanced visualisation using svelte

NOTE	This section contains several interactive visuals. These are available in the html-version, but not in the PDF version.

6.1. Slider

Let's add a slider to the iris visualisation from the previous section that allows us to change the number of flowers displayed in a row instead of the hard-coded 20.

```
<script>
    import Papa from 'papaparse';
    import Flower from './Flower.svelte';

    let datapoints = []

    Papa.parse("https://vda-lab.github.io/assets/iris.csv", {
        header: true,
        download: true,
        complete: function(results) {
            datapoints = results.data
        }
    })

    let slider_value = 20;                                          ①
    $: get_xy = function(idx) {                                     ②
        let y = 25 + (Math.floor(idx / slider_value) * 50)          ③
        let x = 25 + ((idx % slider_value) * 50)                    ③
        return [x,y]
    }
</script>

<input type="range" min="10" max="20" bind:value={slider_value} /><br/> ④
<svg width=1000 height=1000>
    {#each datapoints as datapoint,idx}
        <g transform="translate({get_xy(idx)[0]}, {get_xy(idx)[1]})">
            <Flower datapoint={datapoint} />
        </g>
    {/each}
</svg>
```

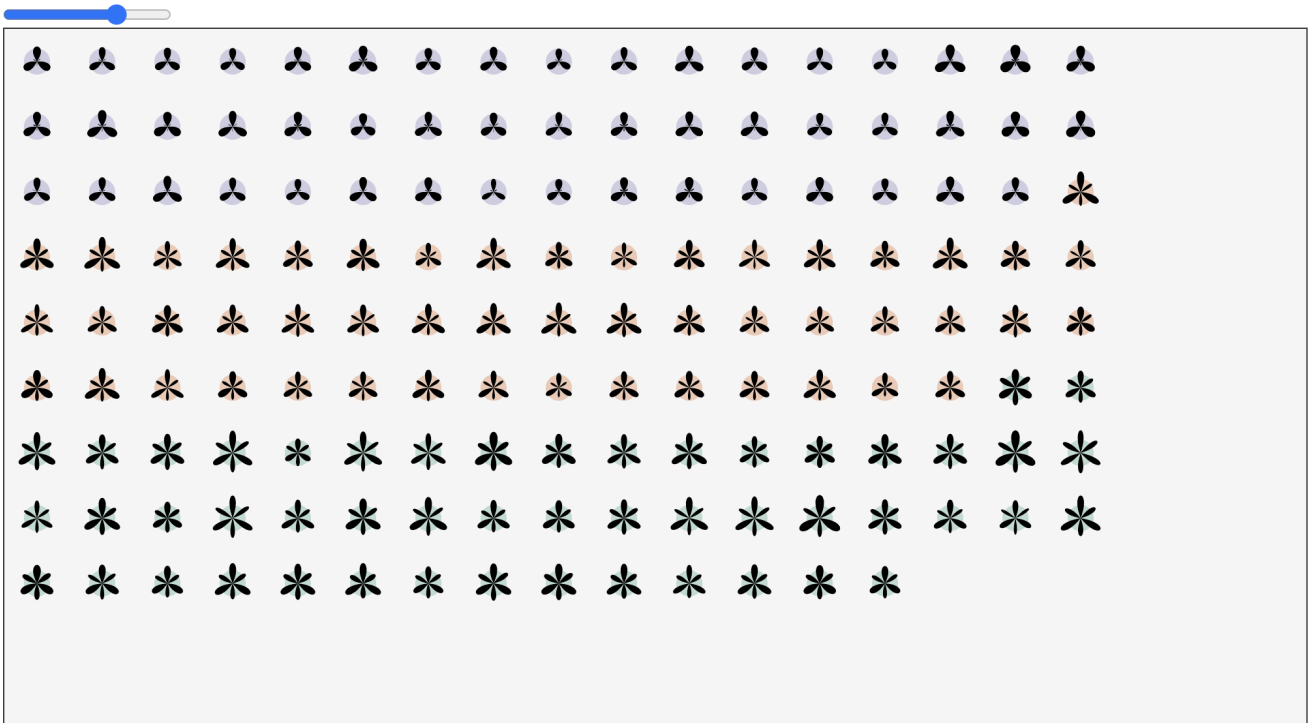
We combined what we saw earlier in this section on adding sliders. What changed in our code?

- **(1)**: We create a new variable that holds the value for the slider and set its default to 20.
- **(2)**: Our function itself will now change as we change the slider. This means that we need to

make the function reactive, and replace `const` with `$:`.

- **(3)**: We replace the hard-coded value of `20` with the value from the slider.
- **(4)**: And we add the actual slider.

The result: ***INTERACTIVE***

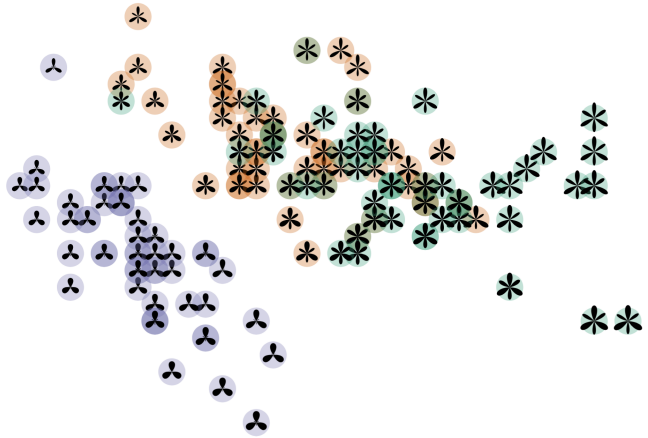


Practice

Add a second slider that allows you to change the size of the flowers themselves.

Practice

Change this example so that, instead of a grid of flowers, these flowers are presented as a scatterplot based on the length and width of their sepals, like so:

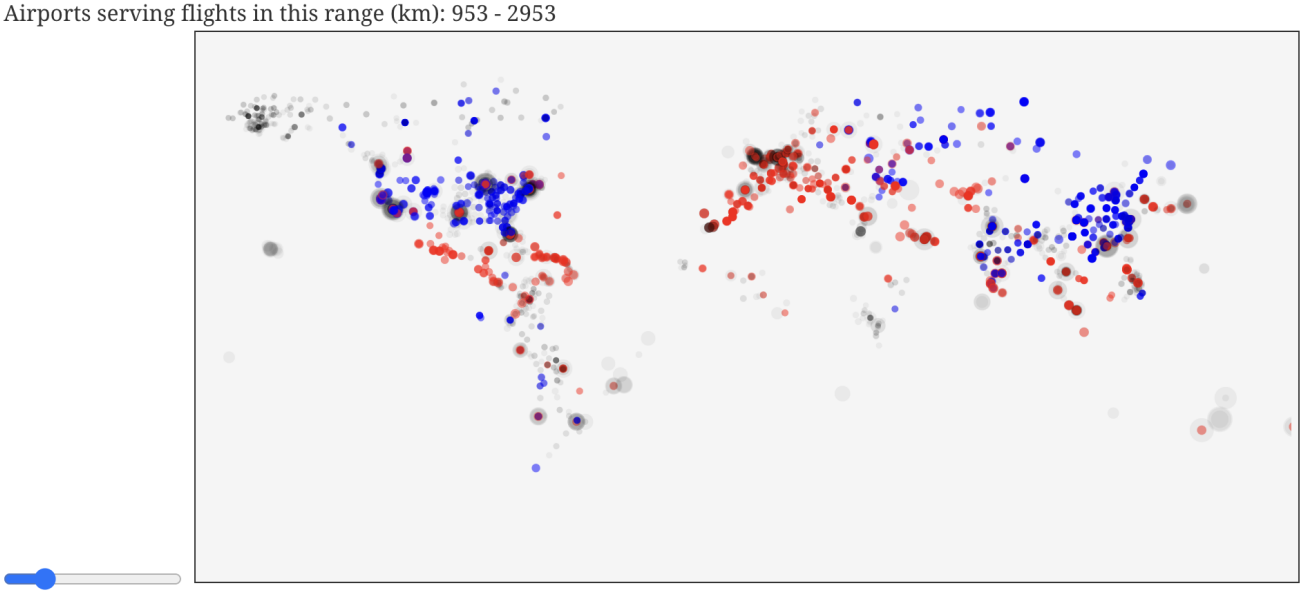


Getting back to our airport data, let's create a version with a slider that highlights only those airports serving flights with a given distance:

Airport flights data

Note: We're only loading a *random part* of the data to make sure that everything is responsive on this site.

Airports serving flights in this range (km): 953 - 2953



The code:

```
<script>
  let slider_value = 5000;

  let datapoints = []
  fetch("https://vda-lab.gitlab.io/datavis-technologies/assets/flights_part.json")
    .then(res => res.json())
    .then(data => datapoints = data.slice(1,5000))

  const rescale = function(x, domain_min, domain_max, range_min, range_max) {
    return ((range_max - range_min)*(x-domain_min))/(domain_max-domain_min) +
range_min
  }
</script>

<style>
  circle {
    opacity: 0.5;
    fill: blue;
  }
  circle.international {
    fill: red;
  }
  circle.hidden {
    opacity: 0.05;
  }
</style>

<h1>Airport flights data</h1>
Airports serving flights in this range (km): {slider_value - 1000} - {slider_value +
```

```
1000} <br/>
<input type="range" min="1" max="15406" bind:value={slider_value} class="slider"
id="myRange" />
<svg width=1000 height=500>
  {#each datapoints as datapoint}
    <circle cx={rescale(datapoint.from_long, -180, 180, 0, 800)}
            cy={rescale(datapoint.from_lat, -90, 90, 400, 0)}
            r={rescale(datapoint.distance, 1, 15406, 2,10)}
            class:international="{datapoint.from_country != datapoint.to_country}"
            class:hidden="{Math.abs(datapoint.distance - slider_value) > 1000}">
      <title>{datapoint.from_airport}</title>
    </circle>
  {/each}
</svg>
```

6.2. Tooltips

Following the "overview first, zoom and filter, and details on demand" mantra, we want to be able to show details when we hover over a datapoint.

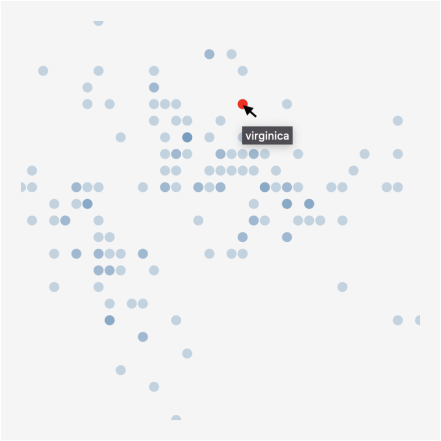
A quick and dirty way to do this, is by using a `title` element embedded within the visual element. For example: instead of

```
<circle cx=50 cy=50 r=10 />
```

we can write

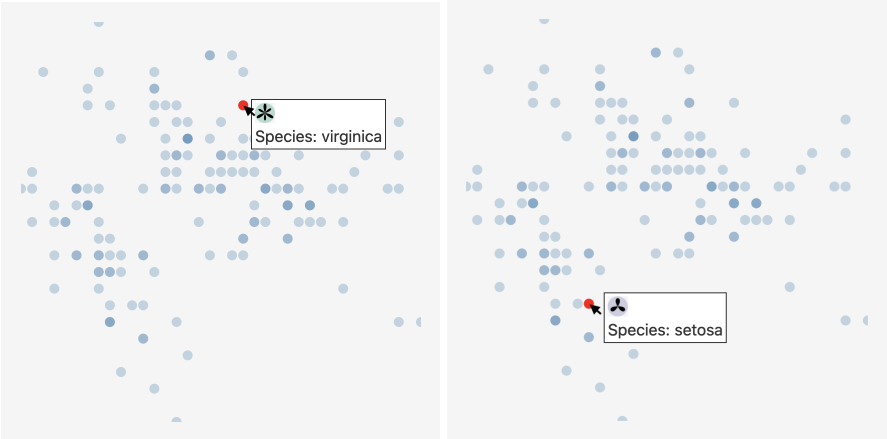
```
<circle cx=50 cy=50 r=10>
  <title>My tooltip</title>
</circle>
```

When we hover over that circle, the text "My tooltip" will be shown after a couple of seconds. You can replace that hard-coded text with, for example, `{datapoint.species}`. The result:



But we can take this much further, and show the glyph of the flower and additional information as

the tooltip, like in the examples below.



To make this happen, we create a `div` that is only shown if `selected_datapoint` is defined.

Our `src/routes/+page.svelte` and `Flower.svelte` are the same as we had above. We adapt `Scatterplot.svelte` like so:

```
<script>
  import { scaleLinear } from 'd3-scale';
  import { extent } from 'd3-array';
  import Flower from './Flower.svelte'

  export let datapoints = []
  export let x;
  export let y;

  export let selected_datapoint = undefined;

  $: xScale = scaleLinear().domain(extent(datapoints.map((d) => { return
d[x]}))).range([0,400])
  $: yScale = scaleLinear().domain(extent(datapoints.map((d) => { return
d[y]}))).range([0,400])

  let mouse_x, mouse_y;                                    ①
  const setMousePosition = function(event) {               ②
    mouse_x = event.clientX;
    mouse_y = event.clientY;
  }
</script>

<style>
  svg {
    background-color: whitesmoke;
    margin: 5px;
    padding: 20px;
  }
  circle {
    fill: steelblue;
```

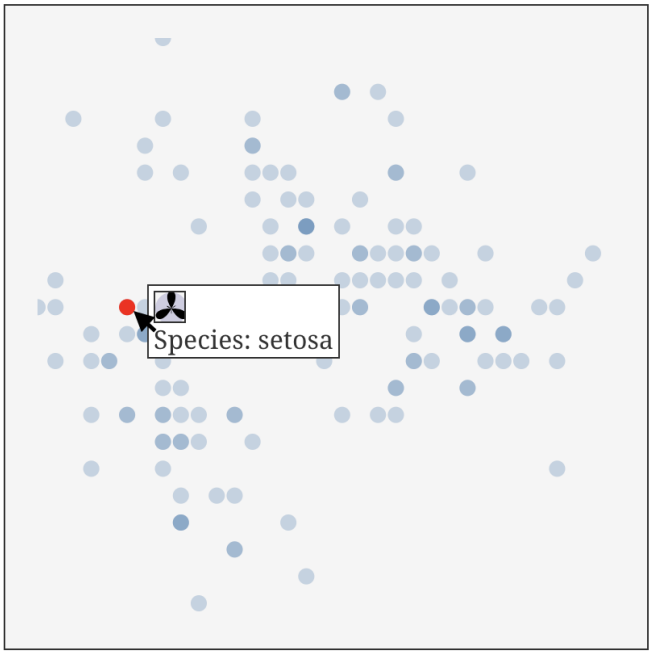
```
    fill-opacity: 0.3;
  }
  circle.selected {
    fill: red;
    fill-opacity: 1;
  }
  #tooltip {                                          ③
    position: fixed;
    background-color: white;
    padding: 3px;
    border: solid 1px;
  }
  svg.tooltip {                                       ④
    margin: 0px;
    padding: 0px;
  }
</style>

<p>{x} by {y}</p>
<svg width=400 height=400>
  {#each datapoints as datapoint}
    <circle cx={xScale(datapoint[x])} cy={yScale(datapoint[y])}
            r=5
            class:selected="{selected_datapoint && datapoint.id ==
selected_datapoint.id}"
            on:mouseover={function(event) {selected_datapoint = datapoint;
setMousePosition(event)}}  ⑤
            on:mouseout={function() {selected_datapoint = undefined}}/>
  {/each}
</svg>

{#if selected_datapoint != undefined}               ⑥
<div id="tooltip" style="left: {mouse_x + 10}px; top: {mouse_y - 10}px">
<svg class="tooltip" width=20 height=20>
  <g transform="translate(10,10)">
  <Flower datapoint={selected_datapoint} />
  </g>
</svg><br/>
Species: {selected_datapoint.species}
</div>
{/if}
```

The main magic happens in **(6)**, where we only show this `div` when `selected_datapoint` is defined. The `div` gets an id of `tooltip` and we set its position next to the location of the mouse (more on that below). The div itself contains an SVG with the flower, and a line of text stating the species. If `Flower` would be an SVG itself, we wouldn't have to define the `<svg>` here, but looking at the `Flower.svelte` file, it returns a `<g>` which should be *part* of an SVG.

The `div` gets a position that depends on `mouse_x` and `mouse_y`. These are set in **(5)** when we hover over a circle, using the `setMousePosition(event)` defined in **(2)**. Such mouse event are automatically

passed the event that triggers them. Check what is in these events by adding a `console.log(event)` in the `setMousePosition` function.

Normally a new `div` is positioned *after* the previous one. This means that the tooltip would be displayed to the right of or below the scatterplot, whether or not we define that `style` attribute in **(6)**. To fix this, we need to set `position: fixed;` in the CSS. Finally, as we have *two* SVGs (one for the scatterplot, and one contained within the tooltip), we might have to give each a different style. In our case, we set the padding and margin to 0 for the SVG in the tooltip.



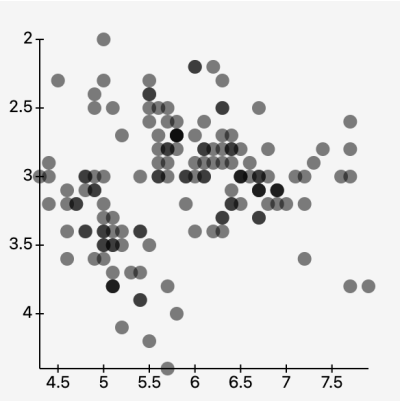
6.3. Axes

There are different ways to draw axes on plots. Looking again at the iris dataset, we can follow the do-it-yourself approach, or use the `d3-axis` library. In the first option, we draw a long line with small lines and text for each tick. As this approach is very simple it is often used.

```
<script>
import { scaleLinear, extent } from 'd3';
let datapoints = [];
fetch("https://raw.githubusercontent.com/domoritz/maps/master/data/iris.json")
  .then(res => res.json())
  .then(data => datapoints = data)
$: console.log(datapoints)
let margins = {"left": 30, "top": 30, "bottom": 30, "right": 30}

$: xDomain = extent(datapoints, (d) => d.sepalLength)
$: yDomain = extent(datapoints, (d) => d.sepalWidth)
$: xScale = scaleLinear().domain(xDomain).range([margins.left,300-margins.right])
$: yScale = scaleLinear().domain(yDomain).range([margins.top,300-margins.bottom])

$: console.log(xDomain)
$: xTicks = [4.5,5,5.5,6,6.5,7,7.5]
$: yTicks = [2,2.5,3,3.5,4]
```

```
</script>

<style>
svg { background-color: whitesmoke }
circle { opacity: 0.5; }
line { stroke: black; }
text { font-size: 12px; }
text.x { text-anchor: middle; }
text.y { text-anchor: end; }
</style>

<svg width=300 height=300>
{#each datapoints as datapoint}
  <circle cx={xScale(datapoint.sepalLength)} cy={yScale(datapoint.sepalWidth)} r=5 />
{/each}

<!-- x axis -->
<line x1={margins.left} y1={300-margins.bottom} x2={300-margins.right} y2={300-
margins.bottom} />
{#each xTicks as tick}
  <line x1={xScale(tick)} y1={300-margins.bottom-3} x2={xScale(tick)} y2={300-
margins.bottom+3} />
  <text class="x" alignment-baseline="hanging" x={xScale(tick)} y={300-
margins.bottom+5}>{tick}</text>
{/each}

<!-- y axis -->
<line x1={margins.left} y1={margins.top} x2={margins.left} y2={300-margins.bottom} />
{#each yTicks as tick}
  <line x1={margins.left-3} y1={yScale(tick)} x2={margins.left+3} y2={yScale(tick)} />
  <text class="y" alignment-baseline="middle" x={margins.left-5}
y={yScale(tick)}>{tick}</text>
{/each}
</svg>
```

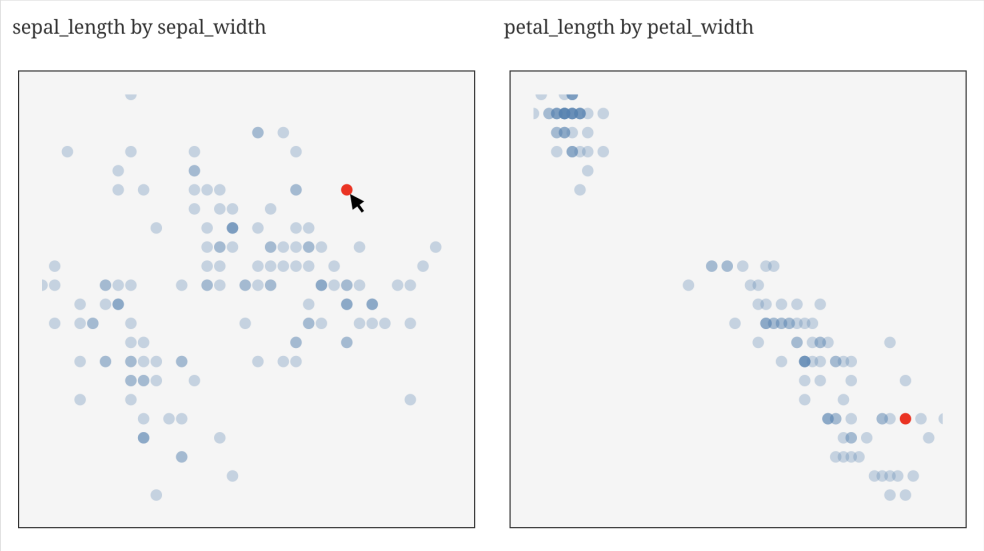
The result:



6.4. Brush

6.4.1. Using hover

Being able to link different visuals together can have a *big* impact on how much insight you can gain from them. Below, we will look into how to make this happen. We'll create two scatterplots on the iris data, and link these together. The final result will be as below. Notice that when you hover over a point, there will also be a point in the other scatterplot that becomes red.



Note that there are multiple ways of achieving this, and definitely look into "svelte stores" as well. Here's we'll go bare bones and do the minimum possible.

`Scatterplot.svelte`:

```
<script>
  import { scaleLinear } from 'd3-scale';
  import { extent } from 'd3-array';

  export let datapoints = []
  export let x;
  export let y;

  export let selected_datapoint = undefined; ①

  $: xScale = scaleLinear()
                .domain(extent(datapoints.map((d) => { return d[x]})))
                .range([0,400])
  $: yScale = scaleLinear()
                .domain(extent(datapoints.map((d) => { return d[y]})))
                .range([0,400])
</script>

<style>
  svg {
```

```
    background-color: whitesmoke;
    margin: 5px;
    padding: 20px;
  }
  circle {
    fill: steelblue;
    fill-opacity: 0.3;
  }
  circle.selected {                          ②
    fill: red;
    fill-opacity: 1;
  }
</style>

<p>{x} by {y}</p>
<svg width=400 height=400>
  {#each datapoints as datapoint}
    <circle cx={xScale(datapoint[x])} cy={yScale(datapoint[y])}
            r=5
            on:mouseover={function() {selected_datapoint = datapoint}}
③
            on:mouseout={function() {selected_datapoint = undefined}}
④
            class:selected="{selected_datapoint && datapoint.id ==
selected_datapoint.id}" ⑤
    />
  {/each}
</svg>
```

What happens here?

- **(1)** We'll need to have a `selected_datapoint` variable to keep track of which datapoint is the selected one.
- **(2)** Instead of using `circle:hover`, we will set the class of our datapoint to `selected` and apply styles like that.
- **(3)** Using `on:mouseover` we can set the `selected_datapoint`...
- **(4)** ... which is unset on `on:mouseout`.
- **(5)** Finally, we can set the class of our circle to `selected` if `selected_datapoint` is defined and the id of our datapoint is the same as the selected datapoint.

`src/routes/+page.svelte`:

```
<script>
  import Papa from 'papaparse';
  import Scatterplot from './Scatterplot.svelte'

  let datapoints = []
  let selected_datapoint = undefined                                    ①
```

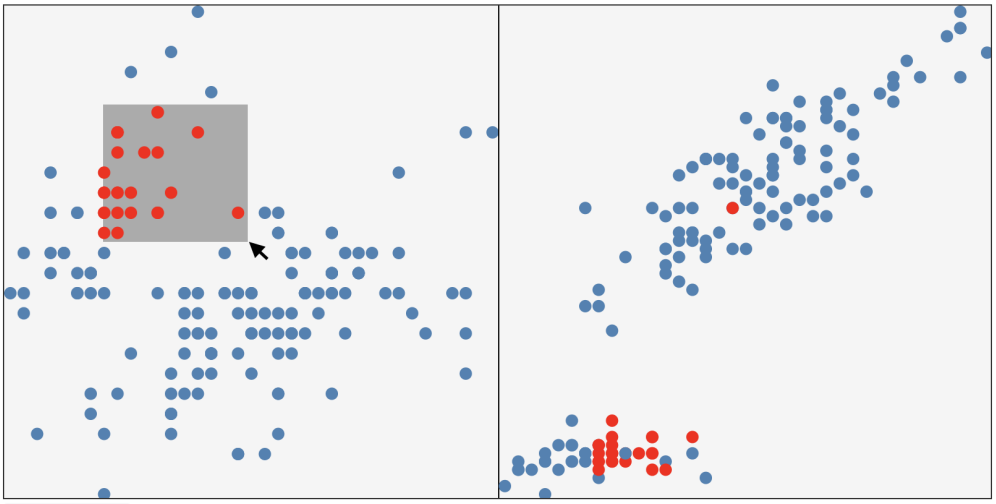
```
  Papa.parse("https://vda-lab.github.io/assets/iris.csv", {
      header: true,
      download: true,
      complete: function(results) {
          let counter = 0                                          ②
          datapoints = results.data.slice(0,-1)
          datapoints.map((d) => d.sepal_length = +d.sepal_length)
          datapoints.map((d) => d.sepal_width = +d.sepal_width)
          datapoints.map((d) => d.petal_length = +d.petal_length)
          datapoints.map((d) => d.petal_width = +d.petal_width)
          datapoints.forEach((d) => {                              ②
              d["id"] = counter
              counter++;
          })
      }
  })
</script>

<table>
<tr>
  <td><Scatterplot bind:selected_datapoint={selected_datapoint}    ③
                   datapoints={datapoints}
                   x="sepal_length"
                   y="sepal_width" /></td>
  <td><Scatterplot bind:selected_datapoint={selected_datapoint}
                   datapoints={datapoints}
                   x="petal_length"
                   y="petal_width" /></td>
</tr>
</table>
```

We define a variable `selected_datapoint` **(1)** that will contain a copy of any datapoint that we hover over in any of the scatterplots. Next, we way of checking if a circle is selected. We can do this by adding a unique ID to all datapoints **(2)**. Finally, we pass the selected datapoint to the scatterplots themselves, but do this using the `bind` operator, so that these scatterplots can pass that info back into the main code (from where it then can be passed to the other scatterplot).

6.4.2. Using a brush

The above is a poor-man's version, and we'd like to have a more useful brush where you can select a *region* of the plot. D3 allows you to do this, but again: it does some things that are unclear to the beginning javascript programmer. In the example below, we only create a proof-of-concept for brushing; in a real setting you would create separate components which would solve some of the issue of the brush going outside the plot, etc.

The plots below show the iris dataset: the left part shows sepal length (x-axis) vs sepal width (y-axis); the right part sepal length (x-axis) vs petal length (y-axis). Brushing on the left part selects flowers that are then highlighted on the right one.



src/routes/+page.svelte:

```
<script>
    import { scaleLinear } from 'd3-scale';
    import Papa from 'papaparse';
    import { onMount } from 'svelte';
    import { extent } from 'd3-array'

    let w = 400;
    let h = 400;

    let datapoints = []
    onMount(() => {
        Papa.parse("https://vda-lab.github.io/assets/iris.csv", {
            header: true,
            download: true,
            complete: function(results) {
                let counter = 0
                datapoints = results.data.slice(0,-1)
                datapoints.map((d) => d.sepal_length = +d.sepal_length)
                datapoints.map((d) => d.sepal_width = +d.sepal_width)
                datapoints.map((d) => d.petal_length = +d.petal_length)
                datapoints.map((d) => d.petal_width = +d.petal_width)
                datapoints.forEach((d) => {
                    d["id"] = counter
                    counter++;
                })
            }
        })
    })

    // slScale: sepal_length, swScale: sepal_width, plScale: petal_length
    $: slScale = scaleLinear().domain(extent(datapoints.map((d) => { return
d.sepal_length}))).range([5,w-5])
    $: swScale = scaleLinear().domain(extent(datapoints.map((d) => { return
d.sepal_width}))).range([h-5,5])
    $: plScale = scaleLinear().domain(extent(datapoints.map((d) => { return
```

```
d.petal_length}))).range([h-5,5])

    let selectedDatapoints = [];
    // $: console.log(datapoints.filter((d) => { return
selectedDatapoints.includes(d.id) }))

    let dragging = false;
    let startX = 0; let startY = 0;
    let startDataX = 0; let startDataY = 0;
    let brushWidth = 0; let brushHeight = 0;
    let mouseX = 0; let mouseY = 0;
    let mouseDataX = 0; let mouseDataY = 0;

    const startBrush = (e) => {
        startX = e.offsetX;
        startY = e.offsetY;

        startDataX = slScale.invert(startX)
        startDataY = swScale.invert(startY)
        mouseX = startX;
        mouseY = startY;
        dragging = true;
    };
    const updateBrush = (e) => {
        mouseX = e.offsetX;
        mouseY = e.offsetY;

        mouseDataX = slScale.invert(mouseX)
        mouseDataY = swScale.invert(mouseY)

        brushWidth = mouseX - startX;
        brushHeight = mouseY - startY;
        if (brushWidth < 0) { brushWidth = -brushWidth; }
        if (brushHeight < 0) { brushHeight = -brushHeight; }
    };

    const getBrushedDatapoints = () => {
        selectedDatapoints = datapoints
            .filter((d) => {
                return (
                    Math.min(startDataX, mouseDataX) < d.sepal_length &&
d.sepal_length < Math.max(startDataX, mouseDataX) &&
                    Math.min(startDataY, mouseDataY) < d.sepal_width && d.sepal_width
< Math.max(startDataY, mouseDataY)
                )})
            .map((d) => d.id);
    };
</script>

<svg width={w*2} height={h}
    on:mousedown={(e) => { startBrush(e); }}
```

```
    on:mousemove={(e) => { if (dragging) { updateBrush(e); getBrushedDatapoints(); }
}}
    on:mouseup={() => { dragging = false; }}
    on:dblclick={() => { brushWidth = 0; brushHeight = 0; }}
>
    <rect class="brush" x={Math.min(startX, mouseX)} y={Math.min(startY, mouseY)}
                        width={brushWidth} height={brushHeight} />
    <g>
    {#each datapoints as datapoint}
        <circle cx={slScale(datapoint.sepal_length)}
cy={swScale(datapoint.sepal_width)} r="5"
            class:selected={selectedDatapoints.includes(datapoint.id)} />
    {/each}
    </g>

    <line x1={w} x2={w} y1=0 y2={h} />

    <g class="plot" transform="translate({w},0)">
    {#each datapoints as datapoint}
        <circle cx={slScale(datapoint.sepal_length)}
cy={plScale(datapoint.petal_length)} r="5"
            class:selected={selectedDatapoints.includes(datapoint.id)} />
    {/each}
    </g>
</svg>

<style>
    line { stroke: black}
    circle { fill: steelblue; }
    circle.selected { fill: red; }
    rect.brush { fill: black; fill-opacity: 0.3; }
</style>
```

6.5. Specific visuals

Below, we will just post example code that can be used as a starting point for more complex visuals. We won't go in depth into explaining this code, though.

6.5.1. Map

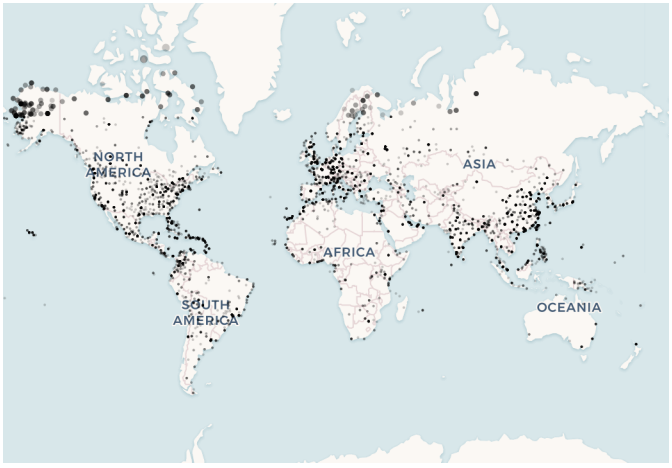
We have shown airports by just plotting their longitude and latitude as a scatterplot, but it'd be nice to plot them on top of an actual map. There are different libraries for doing this, including D3 (see https://www.pluralsight.com/guides/maps-made-easy-with-d3) and leaflet.js, developed by Vladimir Agafonkin.

You'll first have to install the leaflet library with `npm install leaflet`.

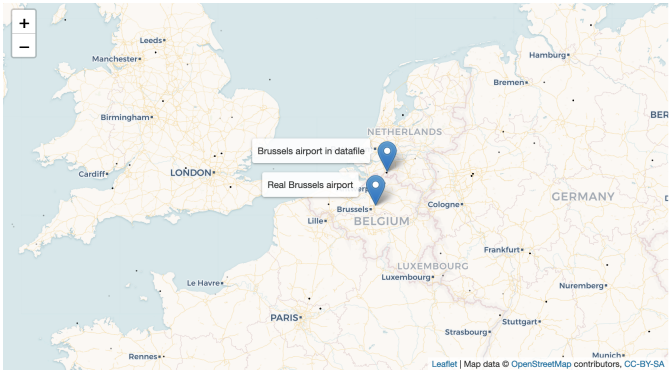
A minimal, basic map:

```
<script>
  import { onMount } from "svelte";
  import L from 'leaflet';

  let datapoints = []
  fetch("https://vda-lab.gitlab.io/datavis-technologies/assets/flights_part.json")
    .then(res => res.json())
    .then(data => datapoints = data)

  $: console.log(datapoints)

  let map;

  onMount(async () => {
    map = L.map("map", {preferCanvas: true }).setView([50.8476,4.3572], 2);

L.tileLayer('https://{s}.basemaps.cartocdn.com/rastertiles/voyager/{z}/{x}/{y}{r}.png'
, {
      attribution:
        'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a>
contributors, <a href="https://creativecommons.org/licenses/by-sa/2.0/">CC-BY-SA</a>',
      maxZoom: 18
    }).addTo(map);
    let real_brussels = L.marker([50.9010, 4.4856]).addTo(map);
    real_brussels.bindTooltip("Real Brussels airport").openTooltip();
    let brussels_in_datafile = L.marker([51.502,4.807],{markerColor:
'red'}).addTo(map);
    brussels_in_datafile.bindTooltip("Brussels airport in datafile").openTooltip();
  });

  $: {
    datapoints.forEach((d) => {
      L.circle([d.from_lat,d.from_long],{
                  stroke: false,
                  color: 'black',
                  radius: 50000} // is radius in meters
                ).addTo(map);
    })
  }
</script>

<style>
  #map {
    height: 480px;
  }
</style>

<svelte:head>
  <link
    rel="stylesheet"
```

```
    href="https://unpkg.com/leaflet@1.6.0/dist/leaflet.css"
    crossorigin="" />

  <script
    src="https://unpkg.com/leaflet@1.6.0/dist/leaflet.js"
    crossorigin="">

  </script>
</svelte:head>

<div id="map" />
```



You might see that some of the airports are in the ocean. Indeed, if we look at Brussels airport, the latitude and longitude in the input file are not exactly the same as the real ones.



6.5.2. Force-directed graph

D3 has a very solid library for drawing node-link diagrams, available at https://github.com/d3/d3-force. Data for a network consists of nodes and links, and should be formatted like this:

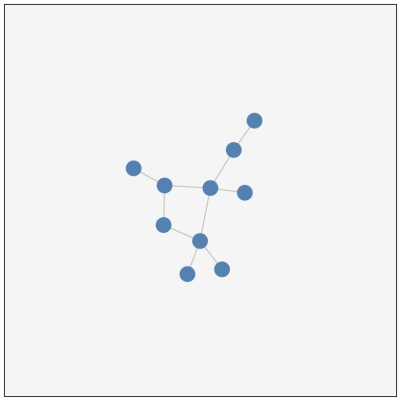
- Nodes must have an ID, e.g. `{"id": 1,"name": "A"}`.
- Links must have a `source` and a `target`, e.g. `{"source": 1,"target": 2}`.

Again, first install the necessary library: `npm install d3-force`.

A simple node-link diagram which allows for dragging nodes around:

```
<script>
  import { onMount } from 'svelte';
  import { forceSimulation, forceLink, forceManyBody, forceCenter } from 'd3-force'

  let nodes = [
    {"id": 1,"name": "A"}, {"id": 2,"name": "B"},
    {"id": 3,"name": "C"}, {"id": 4,"name": "D"},
    {"id": 5,"name": "E"}, {"id": 6,"name": "F"},
    {"id": 7,"name": "G"}, {"id": 8,"name": "H"},
    {"id": 9,"name": "I"}, {"id": 10,"name": "J"}
  ]
  let links = [
    {"source": 1,"target": 2}, {"source": 1,"target": 5},
    {"source": 2,"target": 6}, {"source": 2,"target": 4},
    {"source": 2,"target": 7}, {"source": 3,"target": 4},
    {"source": 8,"target": 3}, {"source": 4,"target": 5},
    {"source": 4,"target": 9}, {"source": 5,"target": 10}
  ]

  let draggedNode = null;
  let simulation;

  function dragNode(event) {
    if ( draggedNode ) {
      draggedNode.x = event.offsetX;
      draggedNode.y = event.offsetY;
      draggedNode.cx = draggedNode.x
      draggedNode.cy = draggedNode.y
      ticked()
    }
  }

  onMount(runSimulation);

  function ticked() {
      nodes = nodes
      links = links
  }

  function runSimulation() {
      simulation = forceSimulation(nodes)
          .force("link", forceLink(links).id(d => d.id))
          .force("charge", forceManyBody().strength(-50))
          .force("center", forceCenter(200,200))
          .on('tick', ticked)
  }
</script>

<style>
  circle {
```

```
  fill: steelblue;
      visibility:visible;
  }
  circle:hover {
      fill: red;
  }
  line {
      stroke: #999;
      stroke-opacity: 0.6;
  }
  .selected {
      fill: red;
      r: 7;
  }
</style>

<svg
  width="400"
  height="400"
  on:mousemove={dragNode}
  on:mouseup={() => { if ( draggedNode ) { runSimulation() }; draggedNode = null; }}>
  {#each links as link}
  <line x1='{link.source.x}' y1='{link.source.y}'
        x2='{link.target.x}' y2='{link.target.y}' >
  </line>
  {/each}

  {#each nodes as point}
      <circle
          class:selected={point.selected}
          cx={point.x}
          cy={point.y}
          r="8"
          on:mousedown={() => { draggedNode = point ; runSimulation() } }
          >
        <title>{point.id}</title>
      </circle>
  {/each}
</svg>
```



Practice

Create a network graph where the nodes are not just circles, but a glyph like we did for the iris flowers.

Chapter 7. Resources

- Svelte homepage
- SvelteKit homepage
- Svelte tutorial
- Svelte/Sveltekit tutorial
- Svelte documentation
- SvelteKit documentation
- Overview of all available CSS selectors
- Overview of all available SVG elements with arguments
- Javascript guide
- Javascript reference