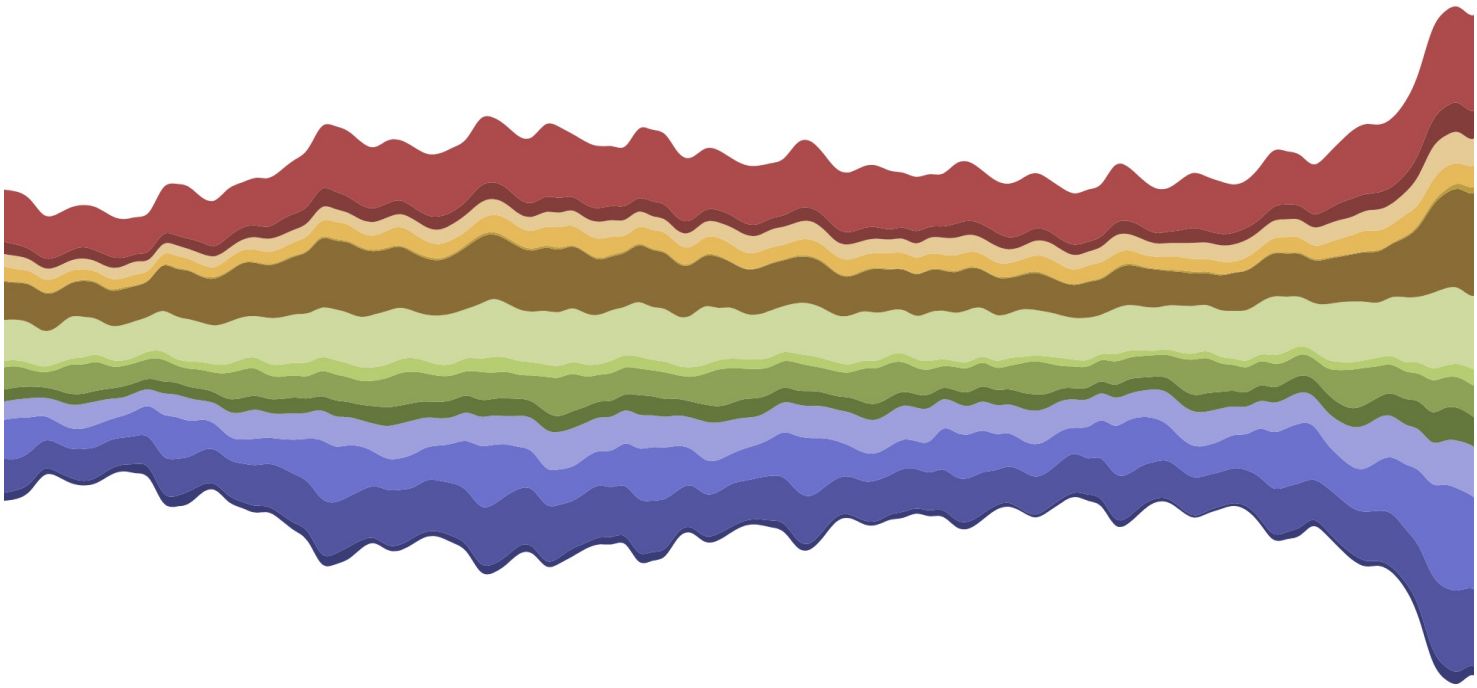


Assignment 6: D3 lab

D3.js is a powerful data visualization library. Rather than being based around existing plot types, d3 allows you to bind any html element. This power is a double edged sword. It allows you to build completely custom data visualizations, but because you're dealing with individual html elements, you're forced to interact with your visualizations on a very low level.

For this lab, we'll be building our way up to making the stream graph above, which shows the number of unemployed people by industry over time.

Submission: Feel free to upload this to github pages, but since this is a self contained directory, you can just edit the individual js files, zip it and submit to blackboard.

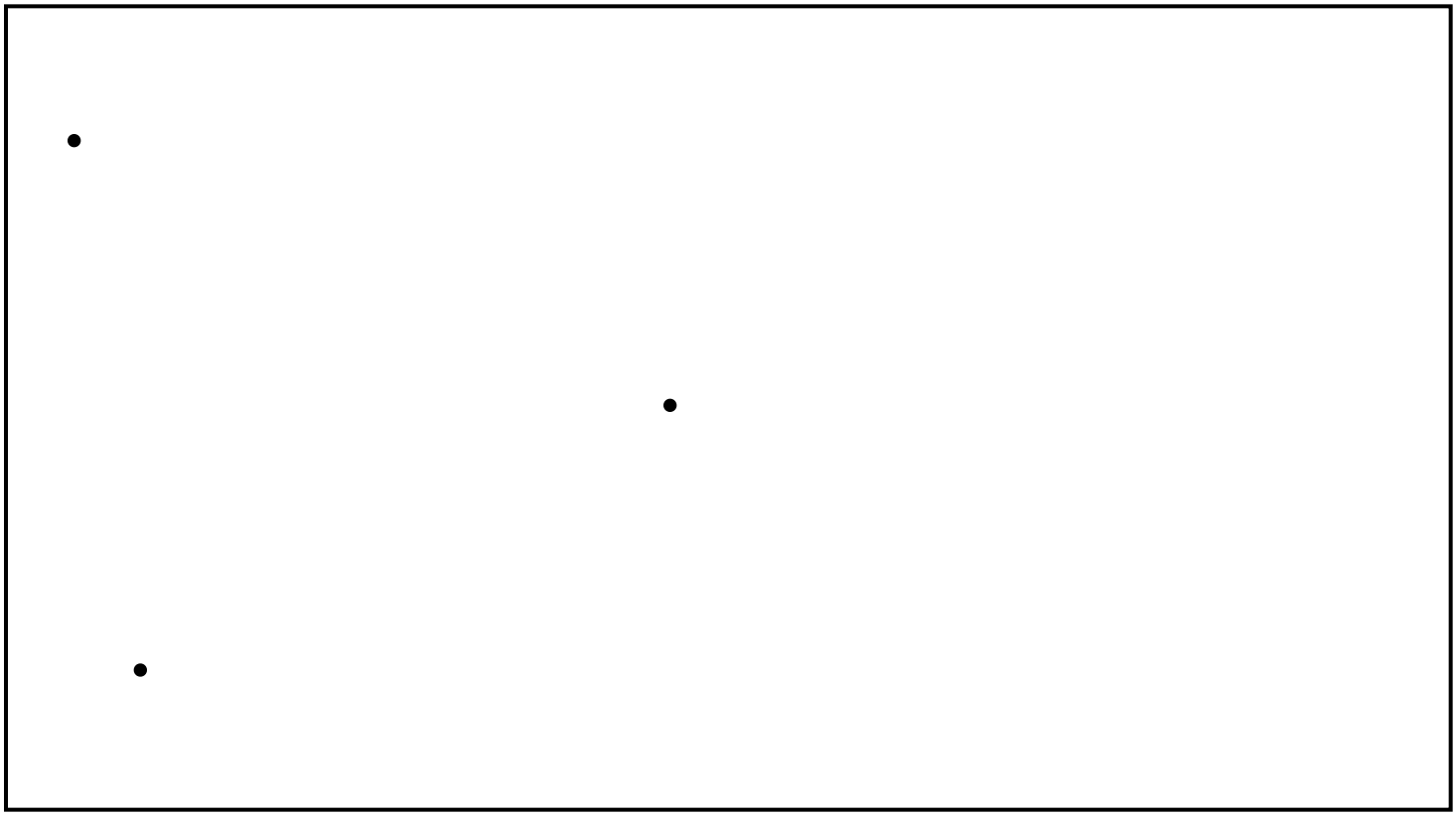
Step 1: Drawing Points

Lets use a simple graph type as an example. We think of scatter plots as points tied x and y axes. Points in d3 are drawn using the svg `circle` object.

Within a larger svg tag, a circle is defined by its center (x and y in pixels) and its radius.

The below code starts with three datapoints containing x and y values. It uses these values to assign centers to three points.

```
dataset = [
    {'x': 50, 'y': 100},
    {'x': 100, 'y': 500},
    {'x': 500, 'y': 300}
]

d3.select('#part1')
    .selectAll('circle')
    .data(dataset)
    .enter()
    .append('circle')
    .attr('r', d => 5)
    .attr('cx', d => d.x)
    .attr('cy', d => d.y)
```



If you look at the above `svg` in your browser's dev tools, you'll see one `circle` element for each point.

Note the pattern that creates this in d3:

- `d3.select` selects the svg element under which the circles will be placed. The octothorpe allows us to select an element by id.
- `selectAll` selects all circles which will be nested under the `svg`
- `data` binds each datapoint to the elements to be created.
- `enter` is needed because the `circle` elements don't exist yet. This tells d3 what to do when the number of datapoints is greater than the existing html elements (in this case, there are zero existing circles)

Also note: the above code includes the passing of anonymous functions as arguments:

```
someFunction('argument', function(d){ return d+5; });
```

Above is one way to pass a function that takes in `d` and returns `d+5`

As of the latest javascript update (ES6), there is a more concise way to define these sorts of functions:

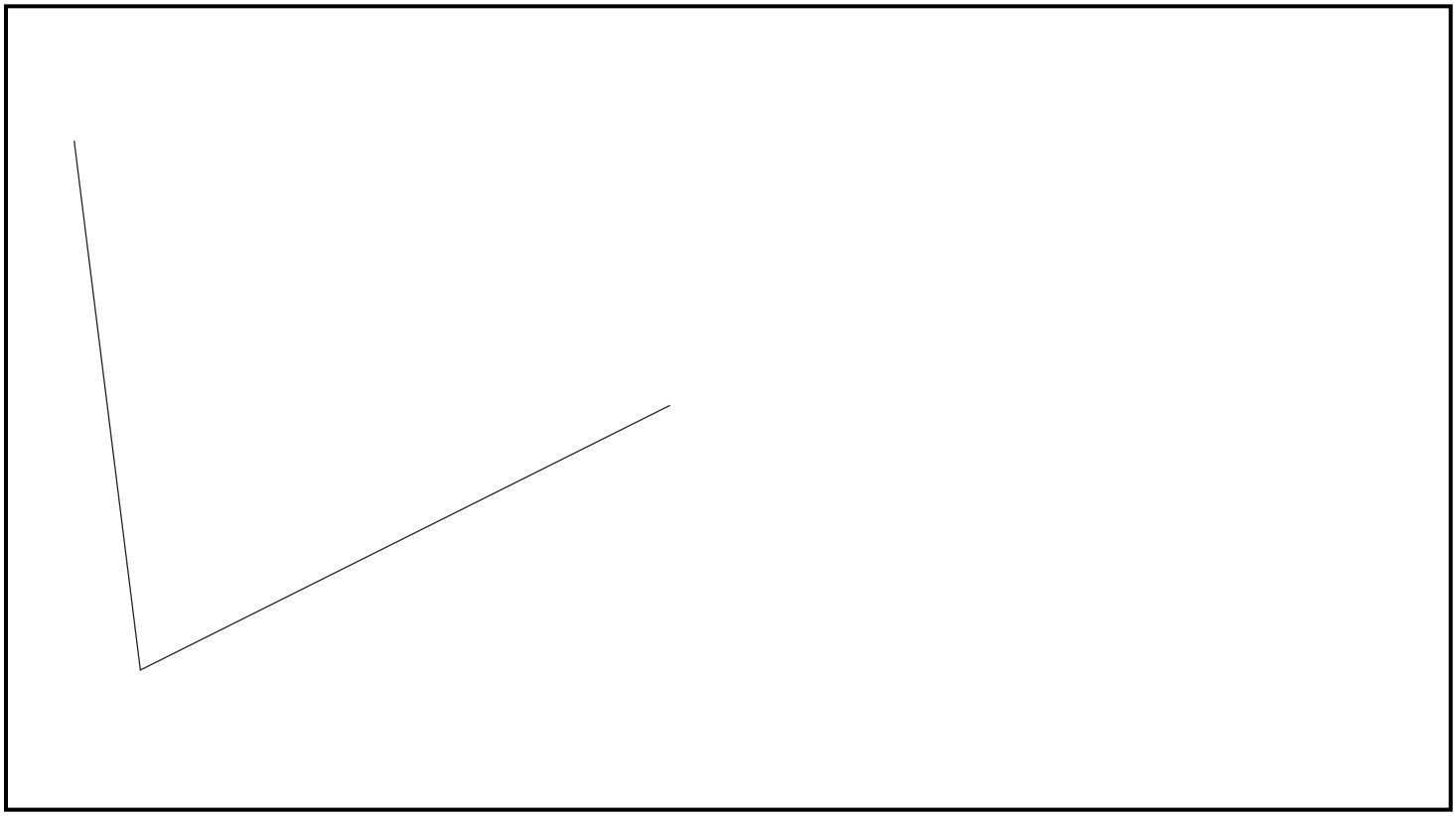
```
someFunction('argument', d => d+5)
```

This "arrow function" is equivalent to the function definition above.

Step 2: Drawing Lines

Lines are drawn with the `path` svg tag. Paths are defined by the `d` attribute. If I wanted to draw a line through the same points shown above, I would use the below path:

```
<path d="M50,100L100,500L500,300" stroke="#2e2928"></path>
```



A path is a single html element however, and we cannot bind each datapoint to individual element.

In order to bind this to our data, we use a `generator` function.

In this case, `d3.line` can be used to generate a path taking the entire dataset as its argument

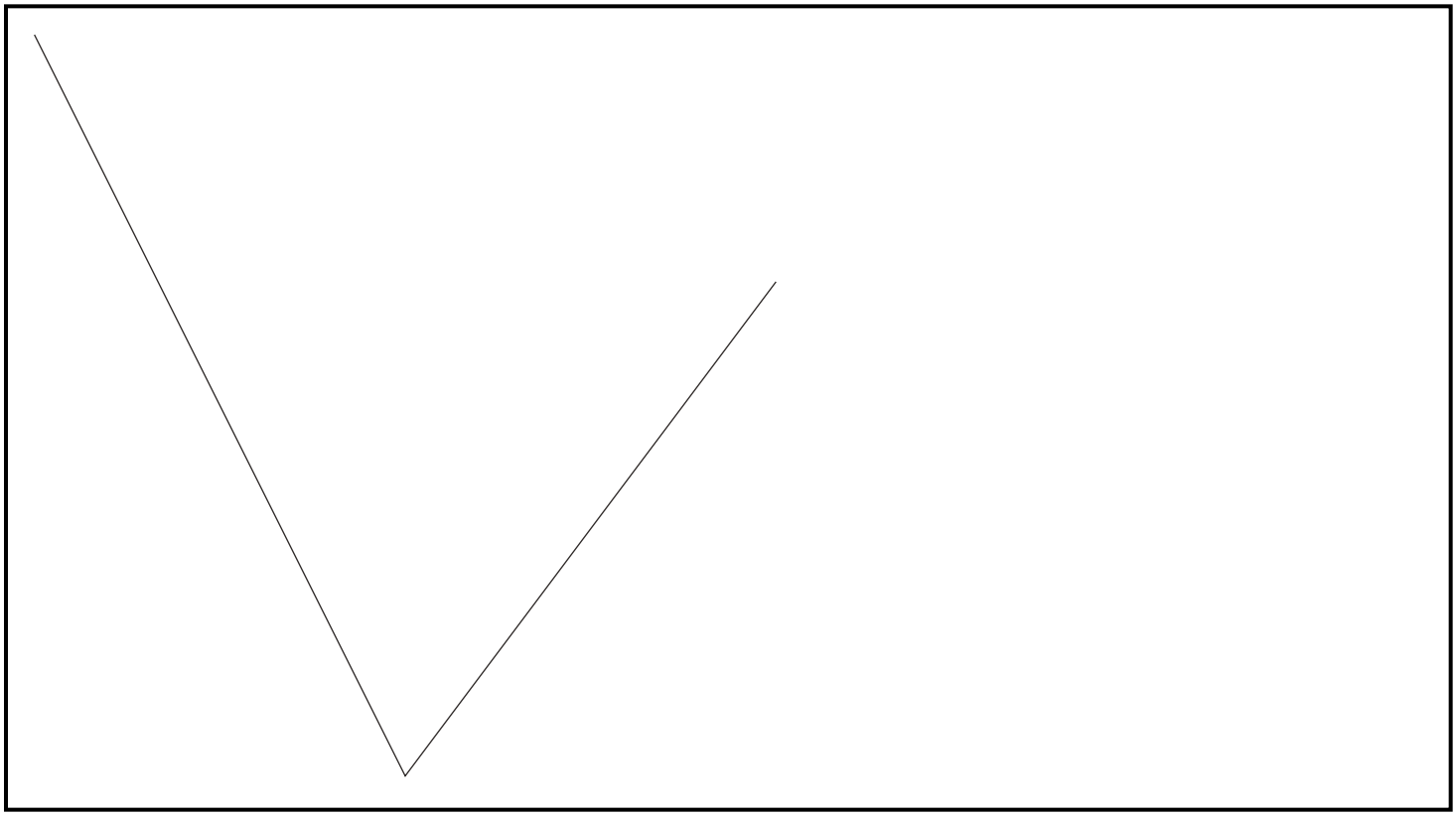
```
dataset = [
    {'x': 50, 'y': 100},
    {'x': 100, 'y': 500},
    {'x': 500, 'y': 300}
]

let line2 = d3.line()
    .x(d => d.x)
    .y(d => d.y)

d3.select('#part2')
    .append('path')
    .attr('d', line2(dataset))
    .attr('stroke', '#2e2928')
```

Step 3: Scales

So far, we've been mapping points directly to pixels within the `svg` tag. Scales map the `domain` of our dataset to the `range` of the `svg` container.

Below is a path scaled to a custom dataset rather than to the pixels of the `svg` container

```
let dataset = [
    {'x': 1, 'y': 1},
    {'x': 2, 'y': 4},
    {'x': 3, 'y': 2}

]

let xScale = d3.scaleLinear().domain([1,3]).range([20, 580]);
let yScale = d3.scaleLinear().domain([1,4]).range([20, 580]);

let line3 = d3.line()
    .x(d => xScale(d.x))
    .y(d => yScale(d.y));

d3.select('#part3')
    .append('path')
    .attr('d', line3(dataset))
    .attr('stroke', '#2e2928')
```



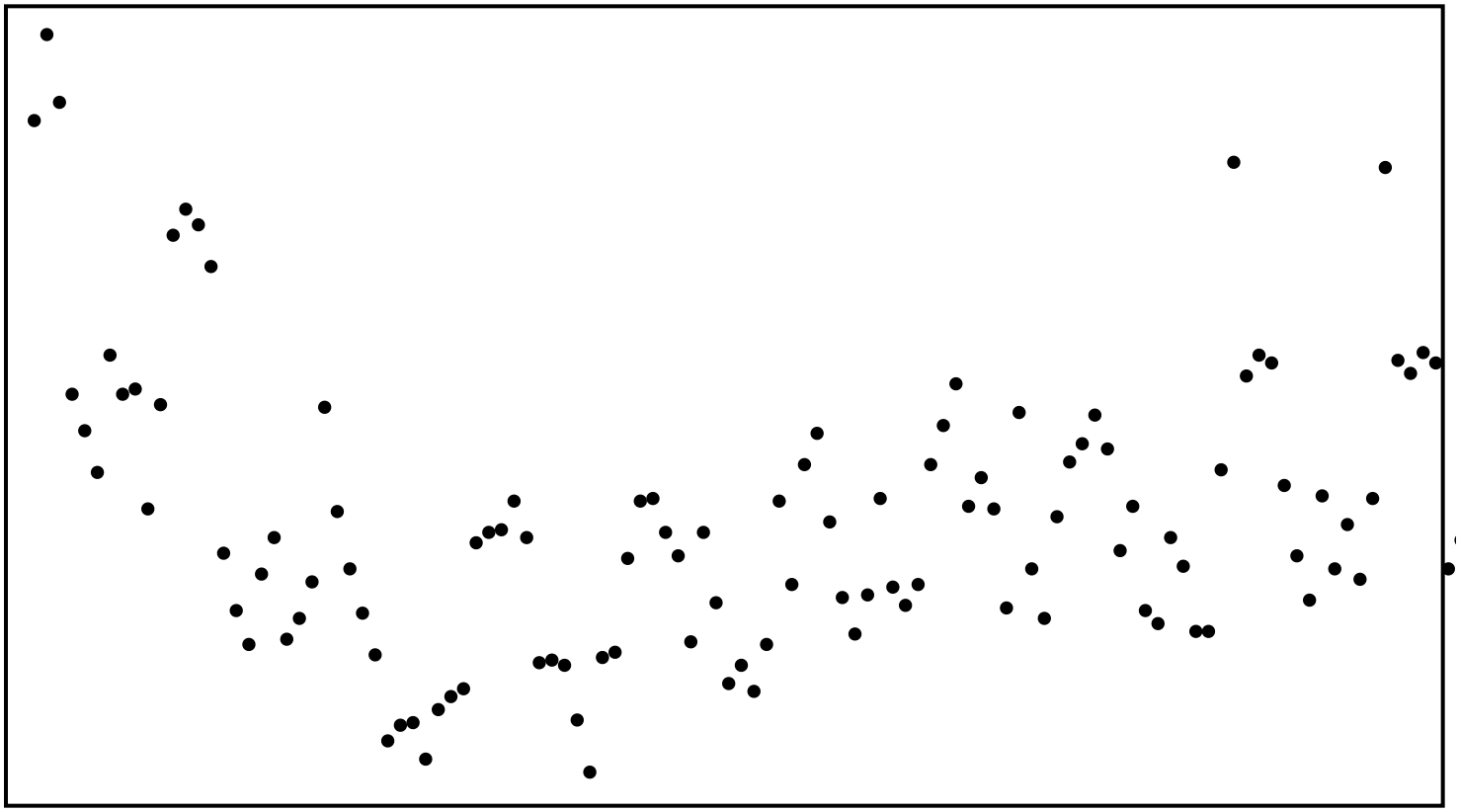
Step 4: Binding Data

So far we've just been using toy datasets, but d3 comes with functions that allow you to bring in external datasets. `d3.csv` lets you read in an external csv.

I have prepared some employment data by industry. Take a look at the file to get an idea of the structure of the data. For this first part, we'll be looking at Agricultural employment.

The below code will bind to agricultural employment data, scale it to the `svg` container, and display it.

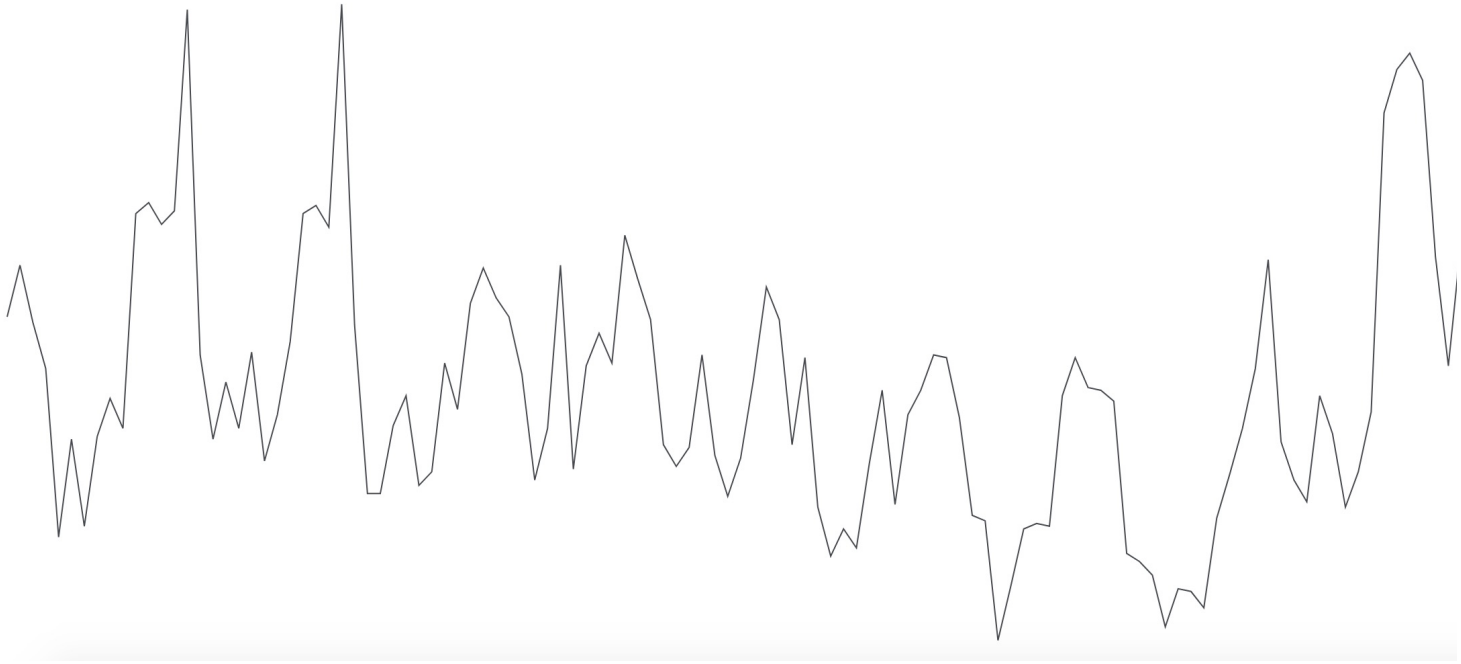
```
d3.csv('ue_industry.csv', data => {

    console.log(data);

    const xScale = d3.scaleLinear()
        .domain(d3.extent(data, d => +d.index))
        .range([1180, 20]);

    const yScale = d3.scaleLinear()
        .domain(d3.extent(data, d => +d.Agriculture))
        .range([580, 20]);

    d3.select('#part4')
        .selectAll('circle')
        .data(data)
        .enter()
        .append('circle')
        .attr('r', d => 5)
        .attr('cx', d => xScale(d.index))
        .attr('cy', d => yScale(d.Agriculture));

});
```

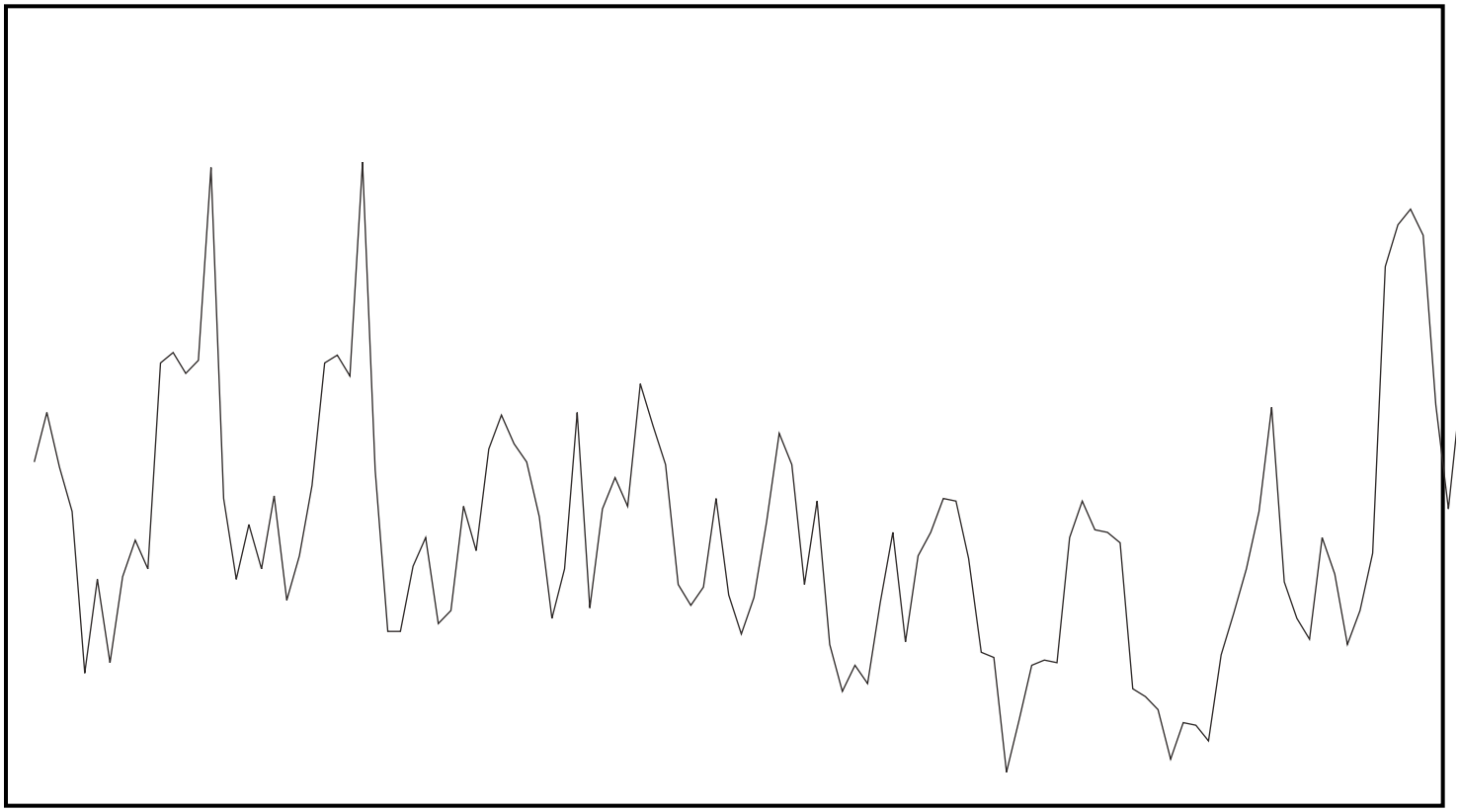


Question 1

Edit the attached file `01answer.js` to display a line graph of Agricultural employment data.

I set the d3 function to select the `svg` tag below. Your graph should look like this image:

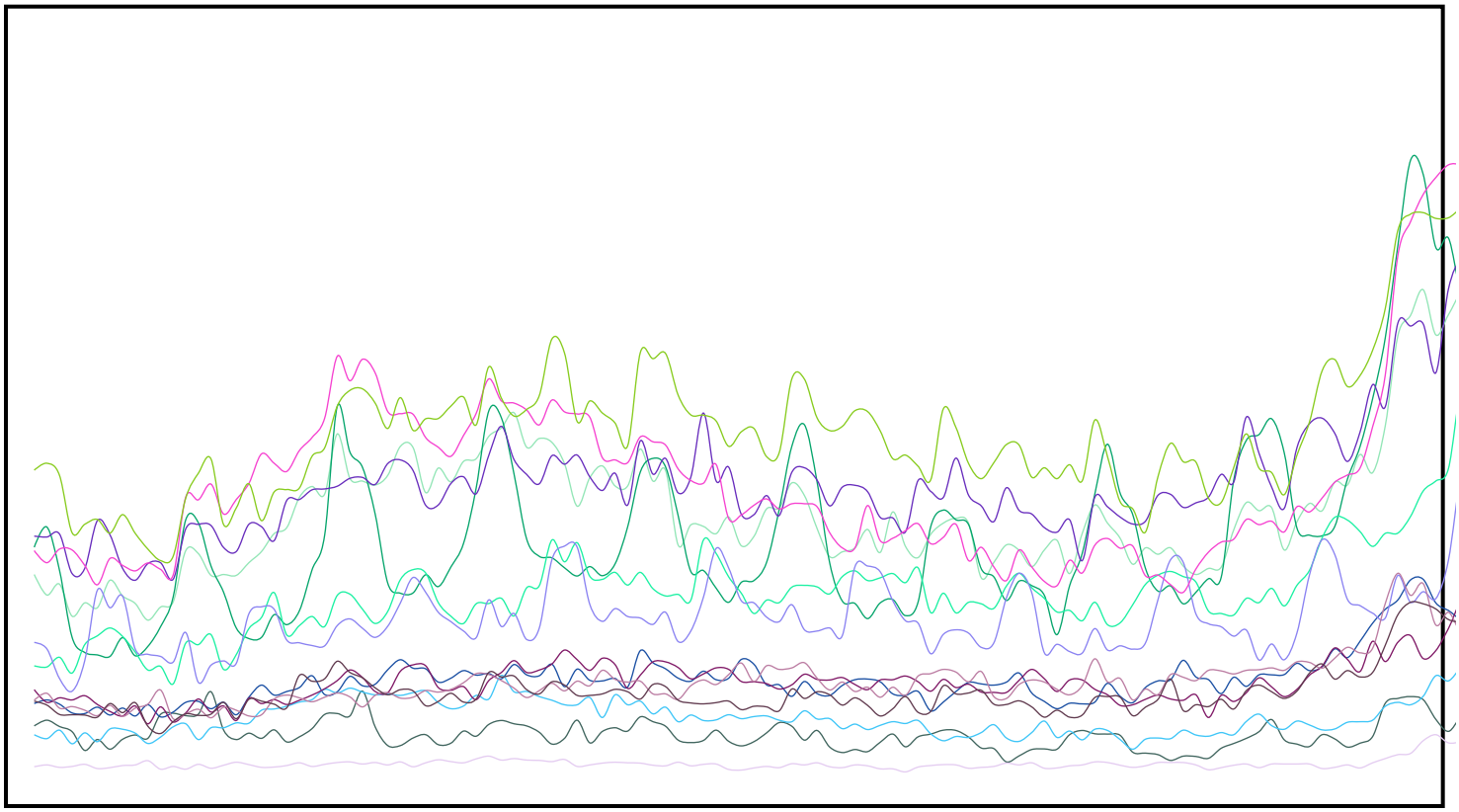




Step 5: Multiple Lines

We have employment numbers for multiple industries in our CSV, not just Agricultural employment.

In order to display all industries on the same graph, we could loop through each industry and get a line for each.



```
d3.csv('ue_industry.csv', data => {

    const industries = ['Agriculture','Business services','Construction','Education and Health',
        'Finance','Government','Information','Leisure and hospitality','Manufacturing',
        'Mining and Extraction','Other','Self-employed','Transportation and Utilities',
        'Wholesale and Retail Trade'];
```

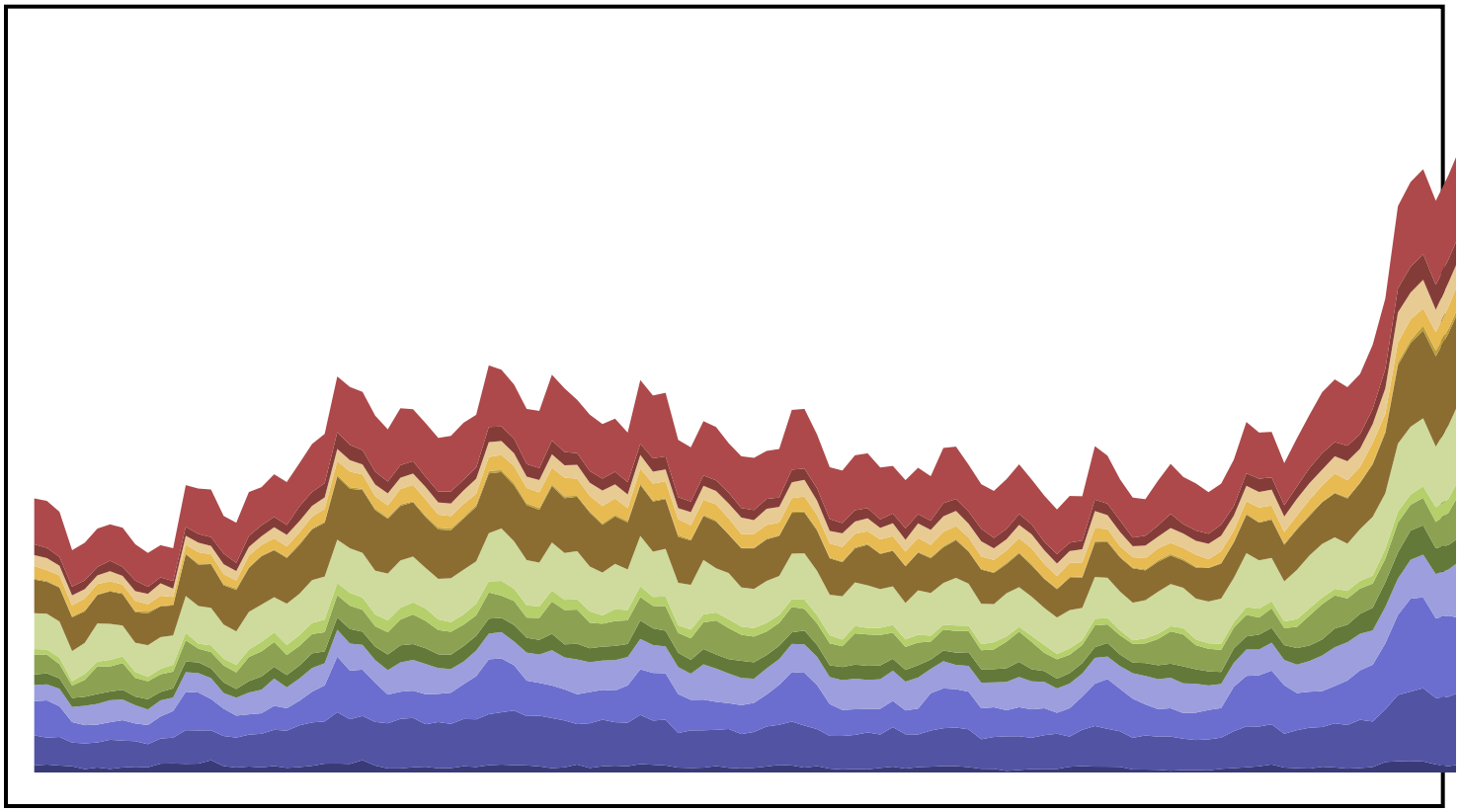
```
        const colors = ['#40655e', '#93e6b7', '#06a56c', '#1cf1a3', '#1a4fa3', '#8b83f2', '#3fc6f8',
            '#682dbd', '#f642d0', '#e4ccf1', '#801967', '#bc7da3', '#613b4f', '#88cc1f'];

        const totalYmax = d3.max(
            industries.map(
                d => d3.max(data, e => +e[d])
            )
        );

        const xScale = d3.scaleLinear()
            .domain(d3.extent(data, d => +d.index))
            .range([20, 1180]);

        const yScale = d3.scaleLinear()
            .domain([0, totalYmax])
            .range([580, 20]);

        const fillScale = d3.scaleOrdinal()
            .domain(industries)
            .range(colors);

        Object.keys(data[0]).forEach(key => {
            if (key != 'index') {

                var line = d3.line()
                    .x(d => xScale(+d.index))
                    .y(d => yScale(+d[key]))
                    .curve(d3.curveCardinal);

                d3.select('#part5')
                    .append('path')
                    .attr('d', line(data))
                    .attr('stroke', fillScale(key))

            }

        });

    });
```

We're doing a few things here:

- `Object.keys` is allowing us to iterate across each key. For each key we're appending a separate path.
- `.curve(d3.curveCardinal` is added to the line function. This adds a smoother interpolation between points, rather than drawing straight line segments between points.

Step 6: Filled Area Plot

Our next step will be to create a filled area plot out of this data.



```
d3.csv('ue_industry.csv', data => {

    const industries = ['Agriculture','Business services','Construction','Education and Health',
        'Finance','Government','Information','Leisure and hospitality','Manufacturing',
        'Mining and Extraction','Other','Self-employed','Transportation and Utilities',
        'Wholesale and Retail Trade'];

    const colors = ['#393b79', '#5253a3', '#6b6ecf', '#9c9ede', '#637939', '#8ca252', '#b5cf6b',
        '#cedb9c', '#8b6d31', '#bd9e38', '#e7ba52', '#e7cb93', '#843c39', '#ad494a'];

    const totalYmax = d3.sum(
        industries.map(
            d => d3.max(data, e => +e[d])
        )
    );

    const xScale = d3.scaleLinear()
        .domain(d3.extent(data, d => +d.index))
        .range([20, 1180]);

    const yScale = d3.scaleLinear()
        .domain([0, totalYmax])
        .range([580, 20]);

    const fillScale = d3.scaleOrdinal()
        .domain(industries)
        .range(colors);

    const stackLayout = d3.stack()
        .keys(industries);

    const stackArea = d3.area()
        .x((d, i) => xScale(i))
        .y0(d => yScale(d[0]))
        .y1(d => yScale(d[1]));

    d3.select('#part6')
        .selectAll('path')
        .data(stackLayout(data))
        .enter().append('path')
        .attr('d', d => stackArea(d))
        .style('fill', d => fillScale(d.key))

});
```

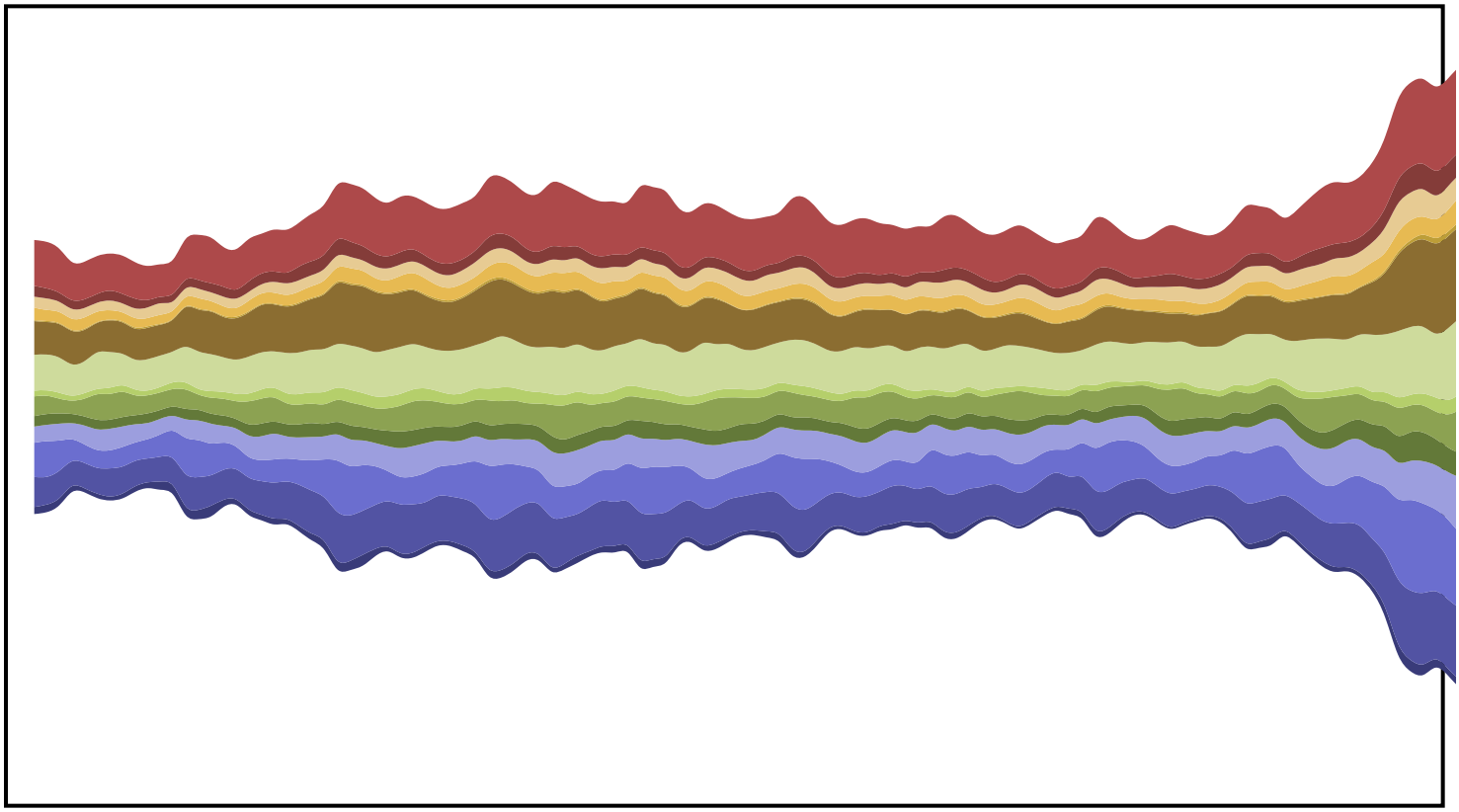
This time, we're using the `d3.stack` layout function to easily return a stacked version of the data.

The `keys` function takes in the industries as keys.

Notice we're performing the layout function in the `data` step. `d3.area` is a more traditional generator function, and is used to create our `path`.

Step 7: Streamgraph

For our next step, we'll be creating a streamgraph. Using d3 parts, a stream graph is simply a special type of filled area plot: one that is symmetric rather than starting at 0



It almost looks like our image, however there's still something off.

Question 2

It seems like our our stream graph is being cut off at the bottom. Try to find out what is needed to display the graph centered in the div and edit `07offset.js` to correct this.

You may want to inspect what the data looks like. Try using `console.log` at any point in your code to print out the data. Then, enter developer mode, and see what prints out in the console.