SSI - RDW - JavaScript Visualisation
28th July 2016

Introduction to D3

Bring on the JavaScript

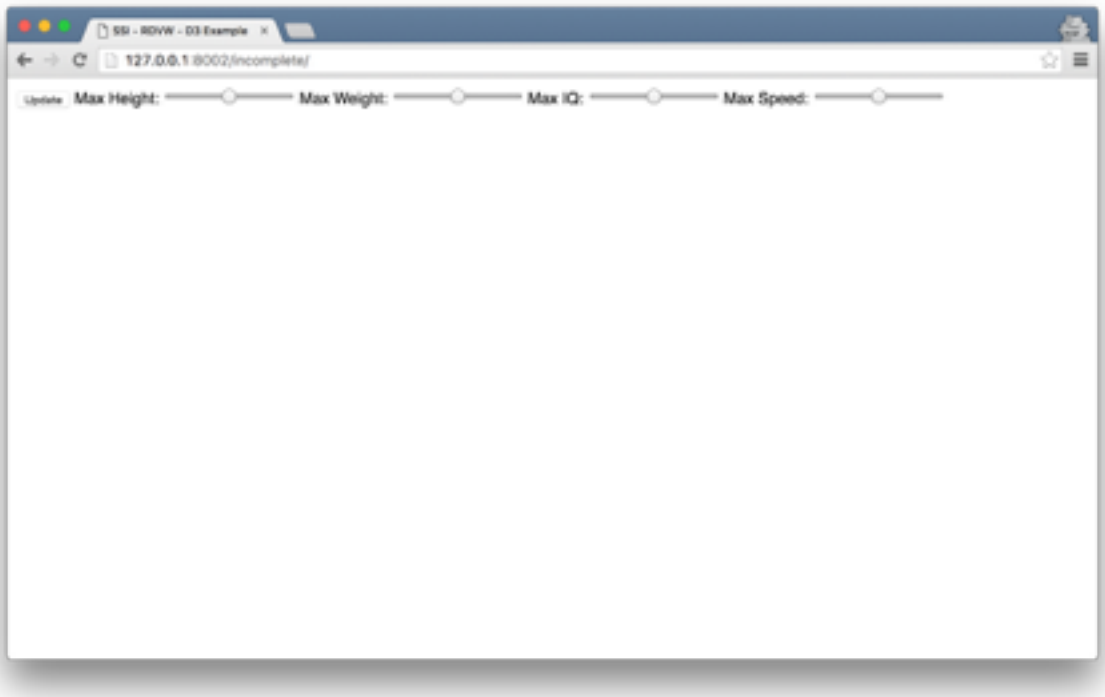
In this session, we will start using the D3 library to visualise data. We will examine the typical D3 process of binding data to elements, and changing the attributes of those elements according to the data values.

This session will walk you through creating a scatterplot visualisation of some random data. You can see a completed version of the visualisation here. Adding lines of code to a single file can get complicated, so if at any point you get lost, each code block to be inserted has a link next to it that shows you what the state of the code should be at that point. At any time you can compare your code to the sample online and make sure you're on the right track.

Getting Started

We can start a simple Python server using the command '`python3 -m http.server`'. This Python server will serve the current directory at the IP address `127.0.0.1:8000`.

If you open a web browser to the address `127.0.0.1:8000` you should see a page looking like the below example, containing some input elements and not much else:



`starting code`
`commit: 4e7ea70`

Using D3

If you look at the source code for the page in a text editor, you'll see we've included version 4 of the D3 library on line 7:

```
<script src="//d3js.org/d3.v4.min.js"></script>
```

You'll see that we have `createRandomData()` and `update()` functions, starting on line 89 and 69 respectively. You won't need to change `createRandomData()`, it is a helper function that creates some random data objects for us to visualise. The `update()` function is where we'll put the code responsible for drawing our data to the screen. Outside of these two functions, there is a block marked out with comments for declaring the variables and other functions we'll need for our visualisation. You'll see here that we've defined the

width and height of our visualisation on lines 47 to 53, and some margin to add space around the edges, and then on line 58 we have our first bit of d3 code:

```
var svg = d3.select('#vis')
 .append('svg')
  .attr('height', height + margin.top + margin.bottom)
  .attr('width', width + margin.left + margin.right)
 .append('g')
  .attr('transform', 'translate(' + margin.left + ',' + margin.top + ')');
```

This demonstrates how d3 employs 'method chaining'. We call a number of methods sequentially, each one acting on the object returned from the previous method. On line 58, we use the '`.select()`' function[1] to select the element in the page with the id "`vis`". This select method returns a variable to us holding the DOM element. We then immediately call `.append()` on this variable to append an '`<svg>`' element to it. The append method returns us the new '`<svg>`' element, which we then call the `.attr()` method on, in order to set the width, and then again to set the height.

We then call the `.append()` function again, to add a '`<g>`' element within the `<svg>`. This is a group element, to which we'll add our visualisation elements. We move this element so its top left corner is at the edge of our margin, which means any elements we add to the group will automatically be offset within our parent `<svg>`. This ensures our margin is respected at all times.

We could accomplish exactly the same thing with the below code:

```
var vis = d3.select('#vis');
var svg = vis.append('svg');
svg.attr('height', height + margin.top + margin.bottom)
svg.attr('width', width + margin.left + margin.right)
var g = svg.append('g')
g.attr('transform', 'translate(' + margin.left + ',' + margin.top + ')');
```

But that doesn't look as nice as the method chaining…

[1] As with jquery, D3 contains methods for selecting DOM elements: 'select' and 'selectAll'

Binding data

The first thing we need to do to visualise our data with D3 is to *bind* the data to the elements that we will use to represent the data. This is where D3 plays its first confusing trick[2] on us. To bind data to elements, we select the elements we want to attach data to:

```
var exampleData = createRandomData(…);

var dots = svg.selectAll(".dot")
     .data(exampleData);
```

Here, we use the `.selectAll()` method to select a reference to all elements with the class of '`dot`'. We then use the '`.data()`' method to *bind* each of our data items to one of the elements in the selection. The only issue is that to begin with there are no elements in our page with a class of '`dot`'.

HOLD UP. If there are no matching elements, what are we attaching our data to?

Really, we're not attaching them to anything. Instead, we're telling D3 that we'd *like* to attach them to elements with a class of '`dot`'. D3 looks at the selection we've chosen, and it counts our data items. If there aren't enough elements in the selection, it works out how many we'd need to create, and returns placeholder elements to us in a function called `.enter()`. We can then call this `.enter()` function, and tell D3 exactly how to create the elements we're looking for.

```
dots
     .enter()
     .append('circle')
     .attr('class', 'dot');
```

D3 will run any code chained to the `.enter()` function as many times as there are elements missing. This will ensure that our SVG ends up containing at least enough of the right elements for our data items.

---

[2] first, assuming we're totally fine with the method chaining structure. Second if that's already got you confused

`.enter()`, update, `.exit()`

Whenever we use `.data()` in combination with a selection, we end up with three things:

1. a reference to all the existing elements in the selection, with their new data items attached. This is the '`update`' selection.
2. `.enter()` - a function we can call to get a reference to all the elements that need to be created in order to make sure there are enough elements for our data items
3. `.exit()` - a function we can call to get a reference to all the elements that exist in our selection but which have no data items bound to them

In combination, these three methods allow us to select elements in our page, add new data to them, and either create new elements (if there aren't enough in our selection), or remove surplus elements (if we have fewer data items than elements). We can then update the properties of all our elements that need updating. We can use the '`merge()`' function to combine the elements that have been 'entered' together with the 'update' selection. This way we can update all new and existing elements at the same time.

`function(d, i){ … }`

There is one last bit of magic[3] that D3 uses that we should know about to allow us to work with data. We have talked about binding data to an element, and creating and removing elements as necessary. But how do we get access to the bound data of an element? D3 allows us to use special anonymous functions within our code to access data items. These functions have a couple of parameters (usually we use just '`d`', sometimes we use both '`d`' and '`i`'), which give us access to the individual data item bound to an element ('`d`') or the data

[3] confusion

item and its index (‘d’ and ‘i’). This will (hopefully) become clearer as we move on through the example.

Back to the example…

Now we know how to bind data to elements, create elements, update elements, and access their bound data, we can create our D3 scatter plot chart! Our update function currently looks something like this:

```
69        function update() {
…
78            var exampleData = createRandomData(…);
79
80            var dots = svg.selectAll(‘.dot’)
81                 .data(exampleData);
82
83        }
```

To create some dots, we first need to select some elements to bind the data. This is what is happening on line 80. On line 81, we actually bind the data to the element selection.

D3 looks at the existing page and finds any elements with a class of ‘`.dot`’. Every element it finds gets assigned one data item from our `exampleData` array. Any elements that do not have a data item assigned are noted for use by the `.exit()` function. Any elements that will need to be created are noted for use by the `enter()` function, which stores ‘placeholder’ elements for these items, that will (it is assumed) be created later.

The first time this page is loaded, there shouldn’t be any elements with a class of ‘`.dot`’ in the page. The update selection (stored in ‘dots’) should therefore be empty, while the `.enter()` selection will contain enough placeholder elements for the entire dataset. The next time `.update()` is run, the number of data

items may change. There may be more data items than exist in the page, in which case the `.enter()` selection will again contain new placeholder items, or there may be fewer data items than existing '`.dot`' elements, in which case the `.exit()` selection will contain the elements to be removed.

Lets get rid of any unnecessary elements first:

step 1
commit: 014f586

```
dots
  .exit()
  .remove();
```

We can then create any elements we need, by calling the enter() function to get the selection of placeholders for any elements that need to be created:

```
var new_dots = dots
      .enter()
      .append('circle')
      .attr('class','dot');
```

All the dots will share some common properties, so we can set these at the same time as we create the elements. For now, we'll just be adjusting the x and y position of the dots based on the data item's `Height` and `Weight` properties. We'll set their size and colour to be static:

step 2
commit: 8e7f441

```
var new_dots = dots
      .enter()
      .append('circle')
      .attr('class','dot')
      .attr('r', 5)
      .attr('fill', 'red')
      .attr('stroke','black')
      .attr('stroke-width', '0.5px');
```

Once we have our new elements and our existing elements to be updated, we can then update them to be at the correct position based on their data values.

First, we merge our selection of `new_dots` with any existing dots using the `merge()` function. We can then use the `function(d){…}` pattern to get access to the data bound to each dot, and update their properties accordingly. As we said, to begin with, we'll just look at the `x` and `y` position of each dot. So how do we know how to translate the `Weight` and `Height` properties of our data items into `x` and `y` coordinates for our visualisation? The simplest solution is just to manually convert them. The data values for both `Weight` and `Height` will be somewhere between 1 and 100. Our `<svg>` is 500 by 500 pixels, so multiplying the data value by 5 will give us a position within the `<svg>`:

```
step 3
commit: c9c6a20
```

```
new_dots.merge(dots)
      .attr("cy", function(d){ return d.Weight * 5; })
      .attr("cx", function(d){ return d.Height * 5; });
```

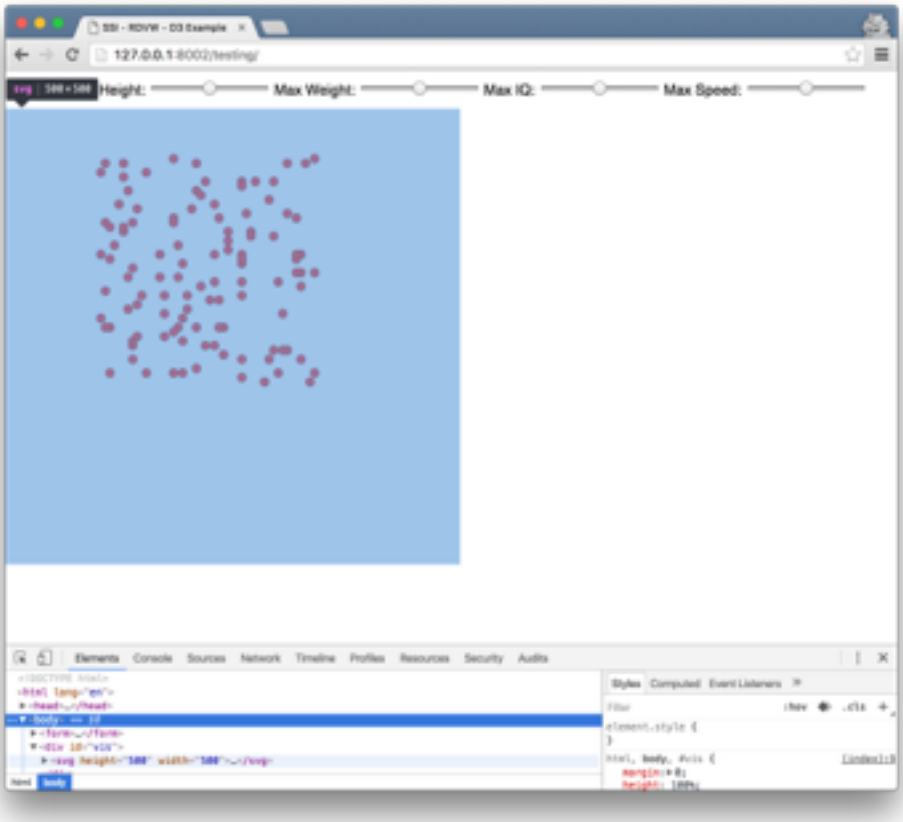
And that's it; all we need to create a dynamically updating visualisation:

Task

Experiment with adjusting the MaxHeight and MaxWeight input sliders and updating the visualisation. What happens to the spread of the dots? Why is that happening?

Introducing scales…

We've coded the visualisation to deal with the maximum possible values of our data. However, at any moment, the input sliders may restrict these values to much less than the maximum. Because we have hard wired the calculation that translates our data values to the x and y positions, this means that at any time, we may not be using the full width or height of the `<svg>` element. In fact, if we inspect the `<svg>` element, we can see that because we've added a margin which has reduced the size of the `<svg>` available to us for drawing, we're actually using completely the wrong values for displaying our data:



This is clearly stupid. What we need to do is find a way of mapping between the *domain* of input values we have, and an output *range* representing the size of the `<svg>`. Fortunately d3 includes scale functions that help us to do exactly that.

We will declare two linear scales for mapping from an input domain to an output range. Add the declarations at the end of the declaration block:

step 4
commit: 8eda1bf

```
var x_scale = d3.scaleLinear()
          .rangeRound([0, width]);

var y_scale = d3.scaleLinear()
          .rangeRound([height, 0]);
```

We only set the output range at the moment, as the input domain is not yet known. We only know the possible input data values once the random data is created in the `update()` function:

step 5
commit: 24e0fbc

```
var example_data = createRandomData(. . .);

x_scale.domain([0, d3.max(exampleData, function(d){ return d.Height; })]);
y_scale.domain([0, d3.max(exampleData, function(d){ return d.Weight; })]);
```

There's a lot going on here, so lets take a look. The '`d3.max()`' function does exactly as it suggests: it finds the maximum value from a set of data. As you can see, it accepts an optional function that allows us to specify which particular property of each piece of data to access. We can then use this max as part of an array setting the bounds of the input domain for the scale.

This scale can then be used to map from any possible input value between 0 and max to a possible output coordinate within our `<svg>`. To do this, we simply call the scale function and pass in the data value to be converted. So, we can update our dots like so:

step 6
commit: 72d5001

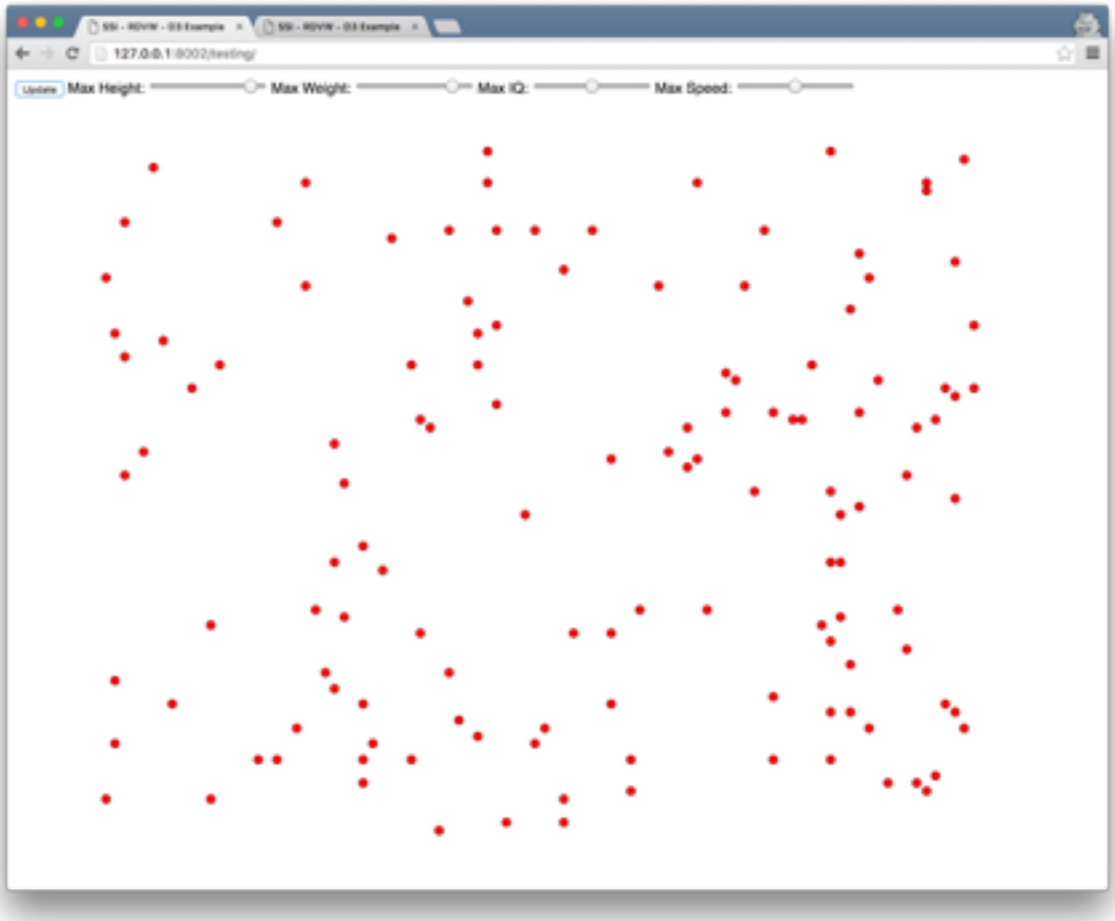
```
96    new_dots.merge(dots)
97        .attr('cx', function(d){ return x_scale(d.Height); })
98        .attr('cy', function(d){ return y_scale(d.Weight); });
```

Our visualisation will then use the full space available within the SVG to display the dots, no matter what the range of input values used. This has an additional benefit - we are not reliant on a fixed size for our visualisation. We can therefore update the width and height of the SVG to be dynamic based upon the size of the browser window at the beginning of the declaration block:

```
step 7
commit: 9977c4c
```

```
39   var view_width = document.getElementById("vis").clientWidth;
40   var view_height = document.getElementById("vis").clientHeight;
```

Our visualisation will now expand to fit our browser window:



Task

Why does the output range of the x_scale run up from '0' to 'width', but the output range of the y_scale run the opposite way down from 'height' to '0'?

Adding axes…

The look of our visualisation can easily be improved by adding axes. We would like to add a y-axis to the left of the visualisation, and an x-axis on the bottom. In our declaration block, we can create these axes like so, passing in the scale that each axis should use:

step 8
commit: d72268f

```
var y_axis = d3.axisLeft(y_scale);
var x_axis = d3.axisBottom(x_scale);
```

We then need to create some `<svg>` group elements to hold the axes once they're drawn. The x axis needs to be translated down to the bottom of the visualisation, to make sure it's in the right place:

step 9
commit: b2001e7

```
svg.append("g")
  .attr("class", "y axis")

svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + (height) + ")");
```

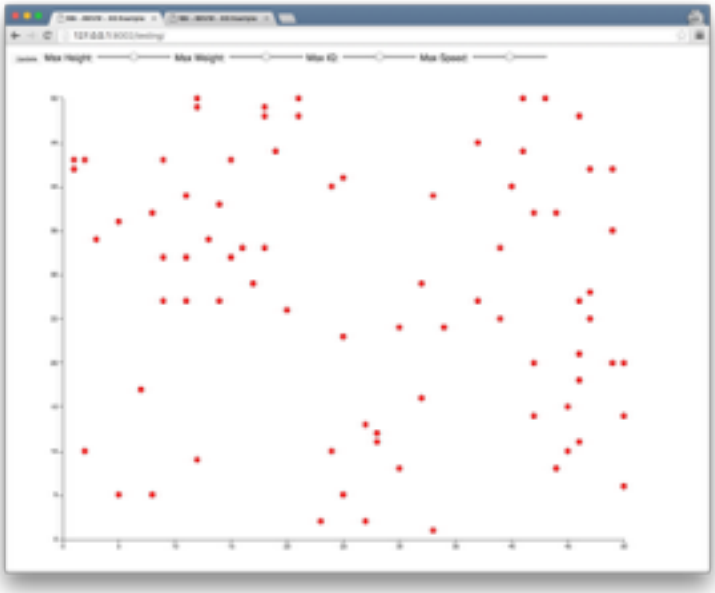
If you refresh the page now, you still won't see any axes. That's because although we've defined some functions for creating the axes, and created some elements for them to be drawn in, we haven't actually drawn them. We will do this at the end of the update function:

step 10
commit: eea111e

```
d3.select('.x.axis')
      .call(x_axis);

d3.select('.y.axis')
      .call(y_axis);
```

Here we select the '`<g>`' elements we created earlier, and call the axis functions on them. This causes d3 to draw the axes as needed:



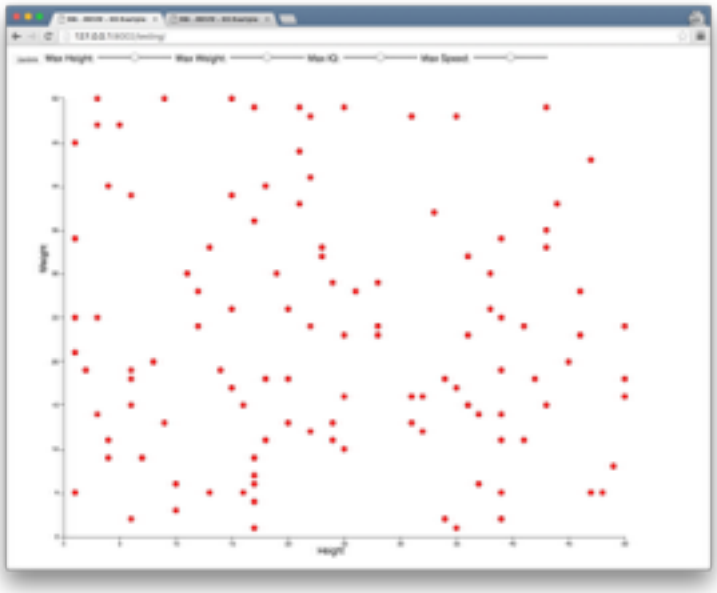
We can improve this further by adding labels to the axes using the SVG text element. In the declaration block where we defined our axes:

step 11

commit: 4c6d2a3

```
svg.append("text")
  .attr("transform", "translate(-30," + height/3 + ")rotate(-90)")
  .attr("text-anchor", “end")
  .text(“Weight");

svg.append(“text")
  .attr("transform", "translate(" + width/2 +  "," + (height+30) + ")")
  .attr("text-anchor", "end")
  .text(“Height");
```



Adding animation…

Our visualisation is starting to look pretty good! However, whenever we update the data, the changes happen instantaneously. We can slow this down, and add some animation to the visualisation.

Animations in d3 use 'transitions'. We can set up a transition with the properties we require, then tell d3 to use this transition when updating or changing properties.

At the beginning of our update function, we can declare our transition:

step 12
commit: 41332af

```
var t = d3.transition()
    .duration(2000);
```

And then we can use the transition to add animation when changing properties of elements. So, in our update code when the dot elements are updated:

step 13
commit: 7dedc2b

```
new_dots.merge(dots)
    .transition(t)
    .attr('cx', function(d){ return x_scale(d.Height); })
    .attr('cy', function(d){ return y_scale(d.Weight); });
```

Our dots will now animate as they enter the visualisation, and will animate as they change position when the data is updated. However, the exiting dots still just disappear. It would be nicer to animate their exit:

step 14
commit: e180910

```
dots
    .exit()
    .transition(t)
    .attr('cx', 0)
    .attr('cy', height)
    .attr('r', 0)
    .remove();
```

Now, before the elements are removed, their position is changed to the bottom left corner of the visualisation. The transition causes this change to be animated.

Finally, lets add the transition to the axes drawing too, so that the updates to these happen smoothly:

step 15
commit: 64eae19

```
d3.select('.x.axis')
    .transition(t)
    .call(x_axis);

d3.select('.y.axis')
    .transition(t)
    .call(y_axis);
```

We now have a dynamic interactive visualisation that animates as it updates. However, we've got some data properties we haven't used…

Adding more dimensions…

Let's add the IQ and speed data items to our visualisation. Firstly the IQ - we'll represent this by adjusting the size of the dots in our visualisation. We'll need a scale to map between the IQ values and the area of the circles. In our declaration block, where we declared the other scales:

step 16
commit: dd63950

```
var area_scale = d3.scaleLinear()
            .range([0, maxArea]);
```

We have set the output *range* of this scale to map between 0 and 800, representing the area of the dots. Again we will set the input *domain* of the scale in the update function, here we use the extent function, which gives us the min and max of the values we're considering.:

step 17
commit: bf4b4a1

```
area_scale.domain(d3.extent(exampleData, function(d){ return d.IQ; }));
```

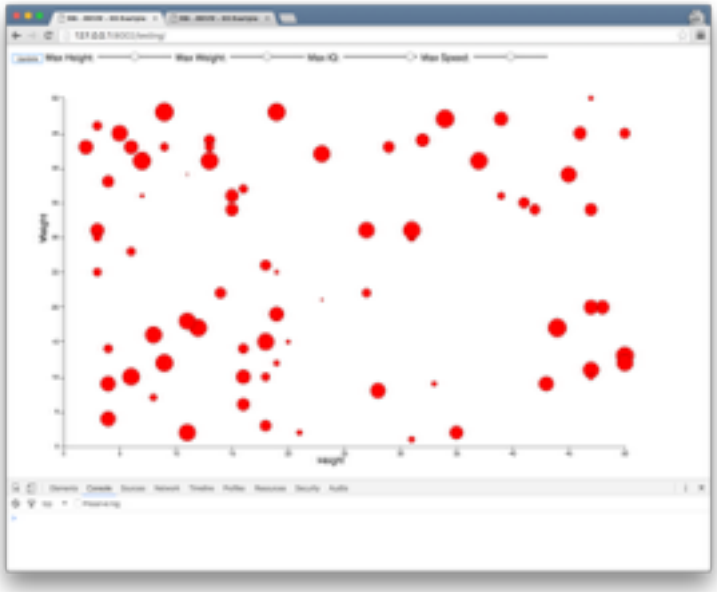
Finally, we use the scale to set the radius of the circles based upon the area we require:

step 18

commit: 74d5883

```
new_dots
    .merge(dots)
        .transition(t)
        .attr('cx', function(d){ return x_scale(d.Height); })
        .attr('cy', function(d){ return y_scale(d.Weight); })
        .attr('r', function(d){
            var area = area_scale(+d.IQ);
            var radius = Math.sqrt(area / Math.PI);
            return radius;
        });
```

Our dots are now scaled in size according to their IQ property:



Finally, we can add the Speed data property to the visualisation. We'll do this using the colour of the dots. Again, we'll need a scale to map between input values and an output range. However, this time rather than a linear scale (mapping between continuous input domain and continuous output range), we want to map between a continuous input domain and a set of categories as the output range. In this case, each category will be a different colour:

step 19
commit: 449b44d

```
var colour_scale = d3.scaleQuantile()
                .range(colorbrewer.Reds[9]);
```

Again, we set the domain in the `update()` function:

step 20
commit: 30f8141

```
colour_scale.domain(d3.extent(data, function(d){ return d.Speed; }));
```

And then finally we set the fill colour of the dots in our update code:

step 21
commit: 22061ac

```
new_dots.merge(dots)
    .transition(t)
    .attr('cx', function(d){ return x_scale(d.Height); })
    .attr('cy', function(d){ return y_scale(d.Weight); })
    .attr('r', function(d){
        var area = area_scale(+d.IQ);
        var radius = Math.sqrt(area / Math.PI);
        return radius;
    })
    .attr('fill', function(d){
        return colour_scale(d.Speed);
    });
```

Our dots are now coloured according to their Speed data property: