CHAPTER 2

Let's Make Some Graphs, Already!

Let's work through some examples that will demonstrate some of the capabilities of D3, and enable you to begin solving some real-world problems immediately—if only by adapting the examples. The first two examples in this chapter will show you how to create standard *scatter* and *xy-plots*, complete with axes, from data files. The plots won't be pretty, but will be fully functional, and it should be easy to tidy up their looks and to apply the concepts to other data sets. The third example is less complete. It is mostly a demo to convince you how easy it is to include event handling and animation in your documents.

A First Example: A Single Data Set

To begin our exploration of D3, consider the small, straightforward data set in Example 2-1. Plotting this simple data set using D3 will already bring us in contact with many essential concepts.

Example 2-1. A simple data set (examples-simple.tsv)

```
x       y
100     50
200     100
300     150
400     200
500     250
```

As I already pointed out in Chapter 1, D3 is a JavaScript library for manipulating the DOM tree in order to represent information visu-

ally. This suggests that any D3 graph has at least *two* or *three* moving pieces:

- An HTML file or document, containing a DOM tree
- A JavaScript file or script, defining the commands that manipulate the DOM tree
- Frequently, a file or other resource, containing the data set

Example 2-2 shows the HTML file in its entirety.

Example 2-2. An HTML page defining an SVG element

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script src="d3.js"></script>                                    ❶
  <script src="examples-demo1.js"></script>                        ❷
</head>

<body onload="makeDemo1()">                                        ❸
  <svg id="demo1" width="600" height="300"
       style="background: lightgrey" />                            ❹
</body>
</html>
```

Yes, that's right. The HTML file is basically empty! All the action takes place in JavaScript. Let's quickly step through the few things that actually do happen in this document:

❶ First, the document loads the D3 library *d3.js*.

❷ Then, the document loads our own JavaScript file. This file contains all the commands that define the graph we are preparing.

❸ The `<body>` tag defines an `onload` event handler, which will be triggered when the `<body> element` has been completely loaded by the browser. The `makeDemo1()` event-handler function is defined in our JavaScript file *examples-demos1.js*.[1]

1 Defining an event handler via the `onload` tag is sometimes frowned upon because of the way it embeds JavaScript code in HTML. See Appendices A and C for modern alternatives.

❹ Finally, the document contains an SVG element of 600×300 pixels. The SVG element has a light-gray background to make it visible, but is otherwise empty.

The third and last piece is the JavaScript file, shown in Example 2-3.

Example 2-3. Commands for Figure 2-1

```
function makeDemo1() {                                          ❶
    d3.tsv( "examples-simple.tsv" )                             ❷
        .then( function( data ) {                               ❸ ❹
            d3.select( "svg" )                                  ❺
                .selectAll( "circle" )                          ❻
                .data( data )                                   ❼
                .enter()                                        ❽
                .append( "circle" )                             ❾
                .attr( "r", 5 ).attr( "fill", "red" )           ❿
                .attr( "cx", function(d) { return d["x"] } )    ⓫
                .attr( "cy", function(d) { return d["y"] } );
        } );
}
```

If you put these three files (the data file, the HTML page, and the JavaScript file), together with the library file *d3.js* into a common directory, and then load the page into a browser, the browser should render a graph equivalent to Figure 2-1.[2]

Let's step through the JavaScript commands and discuss them:

❶ The script defines only a single function, the `makeDemo1()` callback, to be invoked when the HTML page is fully loaded.

❷ The function loads (or "fetches") the data file, using the `tsv()` function. D3 defines several functions to read delimiter-separated-value file formats. The `tsv()` function is intended for tab-separated files.

❸ The `tsv()` function, like all functions in the JavaScript Fetch API, returns a JavaScript `Promise` object. A `Promise` is an object

2 You should be able to load the page and the associated JavaScript file by pointing the browser to the local directory. But the browser may refuse to load the data file in this way, hence it is usually necessary to run a web server when working with D3. See Appendix A for some advice in this regard.

that packages a result set and a callback, and invokes the callback when the result set is complete and ready for processing. A `Promise` provides the `then()` function to register the desired callback to invoke. (We will have more to say about JavaScript promises in "JavaScript Promises" on page 115.)

❹ The callback to be invoked when the file is loaded is defined as an anonymous function, which receives the content of the data file as argument. The `tsv()` function will return the contents of the tab-separated file as an array of JavaScript objects. Each line in the file results in one object, with property names defined through the header line in the input file.

❺ We select the `<svg>` element as the location in the DOM tree to contain the graph. The `select()` and `selectAll()` functions accept a CSS selector string (see "CSS Selectors" on page 36 in Chapter 3) and return matching nodes: `select()` only the first match, and `selectAll()` a collection of all matching nodes.

❻ Next, we select *all* `<circle>` elements inside the `<svg>` node. This may seem absurd: after all, there aren't any `<circle>` elements inside the SVG! But `selectAll("circle")` simply returns an *empty* collection (of `<circle>` elements), so this is not a problem. The odd-looking call to `selectAll()` fulfills an important function by creating a *placeholder* (the empty collection) which we will subsequently fill. This is a common D3 idiom when populating a graph with new elements.

❼ Next we associate the collection of `<circle>` elements with the data set via the call to `data(data)`. It is essential to realize that the two collections (DOM elements on the one hand, and data points on the other) are not affiliated with each other as collections "in bulk." Instead, D3 attempts to establish a one-to-one correspondence between DOM elements and data points: *each data point is represented through a separate DOM element*, which in turn takes its properties (such as its position, color, and appearance) from the information of the data point. In fact, it is a fundamental feature of D3 to establish and manage such one-to-one correspondences between individual data points and their associated DOM elements. (We will investigate this process in much more detail in Chapter 3.)

❽ The `data()` function returns a collection of elements that have been associated with individual data points. In the current case, D3 cannot associate each data point with a `<circle>` element because there aren't any circle elements (yet), and hence the collection returned is empty. However, D3 also provides access to all surplus data points that could not be matched with DOM elements through the (confusingly named) `enter()` function. The remaining commands will be invoked for each element in this "surplus" collection.

❾ First, a `<circle>` element is appended to the collection of `<circle>` elements inside the SVG that was selected on line 6.

❿ Some fixed (that is, not data-dependent) attributes and styles are set: its radius (the `r` attribute) and the fill color.

⓫ Finally, the position of each circle is chosen based on the value of its affiliated data point. The `cx` and `cy` attributes of each `<circle>` element are specified based on the entries in the data file: instead of providing a fixed value, we supply *accessor functions* that, given an entry (that is, a single-line record) of the data file, return the corresponding value for that data point.

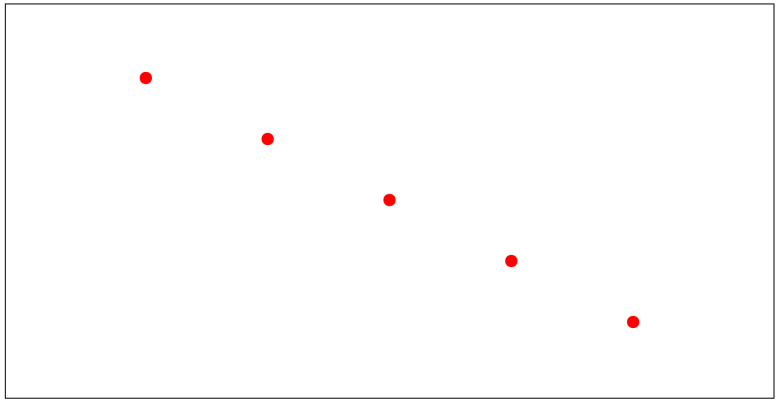

Figure 2-1. A plot of a simple data set (see Example 2-3)

To be honest, that's a remarkably painful process for such a simple graph! Already, you see here what will become ever more apparent: D3 is not a graphics library, much less a plotting tool. Instead, it is a library for manipulating the DOM tree, in order to represent infor-

mation visually. You will find yourself operating on parts of the DOM tree (via selections) all the time. You will also notice that D3 is not exactly parsimonious with keystrokes, requiring the user to manipulate attribute values and to write accessor functions one by one. At the same time, I think it is fair to say that the code, although verbose, is clean and rather straightforward.

If you actually experiment with the examples, you may encounter some additional surprises. For example, the `tsv()` function is picky: columns *must* be tab separated, whitespace is *not* ignored, the header line *must* be present, and so on. Finally, on closer inspection of the data set and the graph you should realize that the figure is actually not correct—it's upside down! That's because SVG uses "graphical coordinates" with the horizontal axis running left to right as usual, but with the vertical axis running top to bottom.

Let's recognize these initial impressions as we continue our exploration with the second example.

A Second Example: Two Data Sets

For our second example, we will use the data set in Example 2-4. It looks almost as innocuous as the previous one, but on closer inspection, it reveals some additional difficulties. Not only does this file contain *two* data sets (in columns `y1` and `y2`), but the numeric ranges of the data require attention. In the previous example, the data values could be used directly as pixel coordinates, but values in the new data set will require scaling to transform them to meaningful screen coordinates. We will have to work a little harder.

Example 2-4. A more complicated data set (examples-multiple.tsv)

```
x       y1      y2
1.0     0.001   0.63
3.0     0.003   0.84
4.0     0.024   0.56
4.5     0.054   0.22
4.6     0.062   0.15
5.0     0.100   0.08
6.0     0.176   0.20
8.0     0.198   0.71
9.0     0.199   0.65
```

Plotting Symbols and Lines

The page we use is essentially the same that we used previously (Example 2-2), except that the line:

```
<script src="examples-demo1.js"></script>
```

must be replaced with:

```
<script src="examples-demo2.js"></script>
```

referencing our new script, and the `onload` event handler must give the new function name:

```
<body onload="makeDemo2()">
```

The script itself is shown in Example 2-5, and the resulting figure is shown in Figure 2-2.

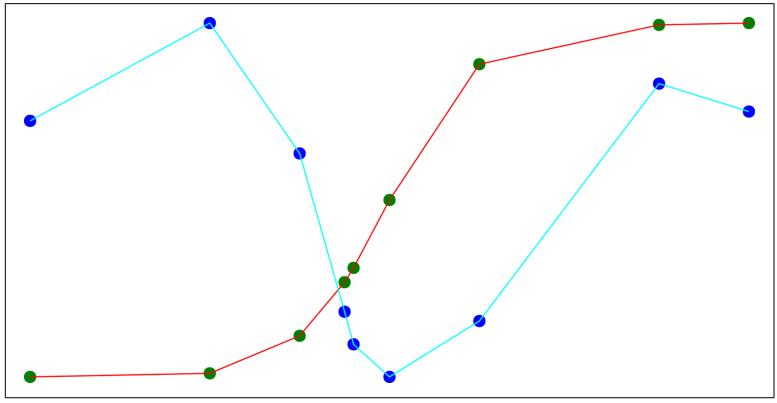

Figure 2-2. A basic plot of the data set in Example 2-4 (also see Example 2-5)

Example 2-5. Commands for Figure 2-2

```
function makeDemo2() {
    d3.tsv( "examples-multiple.tsv" )
        .then( function( data ) {
            var pxX = 600, pxY = 300;                          ❶

            var scX = d3.scaleLinear()                         ❷
                .domain( d3.extent(data, d => d["x"] ) )       ❸
                .range( [0, pxX] );
            var scY1 = d3.scaleLinear()                        ❹
                .domain(d3.extent(data, d => d["y1"] ) )
                .range( [pxY, 0] );                            ❺
            var scY2 = d3.scaleLinear()
```

```
            .domain( d3.extent(data, d => d["y2"] ) )
            .range( [pxY, 0] );

        d3.select( "svg" )                                          ❻
            .append( "g" ).attr( "id", "ds1" )                      ❼
            .selectAll( "circle" )                                  ❽
            .data(data).enter().append("circle")
            .attr( "r", 5 ).attr( "fill", "green" )                 ❾
            .attr( "cx", d => scX(d["x"]) )                         ❿
            .attr( "cy", d => scY1(d["y1"]) );                      ⓫

        d3.select( "svg" )                                          ⓬
            .append( "g" ).attr( "id", "ds2" )
            .attr( "fill", "blue" )                                 ⓭
            .selectAll( "circle" )                                  ⓮
            .data(data).enter().append("circle")
            .attr( "r", 5 )
            .attr( "cx", d => scX(d["x"]) )
            .attr( "cy", d => scY2(d["y2"]) );                      ⓯

        var lineMaker = d3.line()                                   ⓰
            .x( d => scX( d["x"] ) )                                ⓱
            .y( d => scY1( d["y1"] ) );

        d3.select( "#ds1" )                                         ⓲
            .append( "path" )                                       ⓳
            .attr( "fill", "none" ).attr( "stroke", "red" )
            .attr( "d", lineMaker(data) );                          ⓴

        lineMaker.y( d => scY2( d["y2"] ) );                        ㉑

        d3.select( "#ds2" )                                         ㉒
            .append( "path" )
            .attr( "fill", "none" ).attr( "stroke", "cyan" )
            .attr( "d", lineMaker(data) );

//          d3.select( "#ds2" ).attr( "fill", "red" );              ㉓
    } );
}
```

❶ For later reference, assign the size of the embedded SVG area to variables (px for pixel). Of course it is possible to leave the size specification out of the HTML document entirely and instead set it via JavaScript. (Try it.)

❷ D3 provides *scale objects* that map an input *domain* to an output *range*. Here we use linear scales to map the data values from their natural domain to the pixel range of the graph, but the

library also includes logarithmic and power-law scales, and even scales that map numeric ranges to colors for false-color or "heatmap" graphs (see Chapters 7 and 8). Scales are function objects: you call them with a value from the domain and they return the scaled value.

❸ Both domain and range are specified as two-element arrays. The `d3.extent()` function is a convenience function that takes an array (of objects) and returns the greatest and smallest values as a two-element array (see Chapter 10). To extract the desired value from the objects in the input array, we must supply an accessor function (similar to what we did in the final step in the previous example). To save on keystrokes, here (and in most of the following!) we make use of JavaScript's *arrow functions* (or "fat arrow notation"—see Appendix C).

❹ Because the three columns in the data set have different ranges, we need three scale objects, one for each column.

❺ For the vertical axes, we invert the output range in the definition of the scale object to compensate for the upside-down orientation of the SVG coordinate system.

❻ Select the `<svg>` element to add symbols for the first data set.

❼ This is new: before adding any graphical elements, we append a `<g>` element and give it a unique identifier. The final element will look like this:

```
<g id="ds1">...</g>
```

The SVG `<g>` element provides a logical grouping. It will enable us to refer to *all* symbols for the first data set together and to distinguish them from symbols for the second data set (see Appendix B and the sidebar "The <g> Element Is Your Friend" on page 112).

❽ As before, we create an empty placeholder collection using `selectAll( "circles" )`. The `<circle>` elements will be created as children of the `<g>` element just added.

❾ Fixed styles are applied directly to each `<circle>` element.

❿ An accessor functions picks out the appropriate column for the horizontal axis. Note how the scale operator is applied to the data before it is returned!

⓫ An accessor function picks out the appropriate column for the first data set, again scaled properly.

⓬ The familiar procedure is used again to add elements for the second data set—but note the differences!

⓭ For the second data set, the fill color is specified as a `fill` attribute on the `<g>` element; this appearance will be inherited by its children. Defining the appearance on the parent allows us to change the appearance of all children in one fell swoop at a later time.

⓮ Again, an empty collection is created to hold the newly added `<circle>` elements. This is where the `<g>` parent element is more than a convenience: if we would invoke `selectAll( "circle" )` on the `<svg>` element at this point, we would not obtain an empty collection, but instead receive the `<circle>` elements from the *first* data set. Instead of adding new elements, we would modify the existing ones, overwriting the first data set with the second. The `<g>` element allows us to distinguish clearly between elements and their mapping to data sets. (This will become clearer once we will have studied the D3 `Selection` abstraction systematically in Chapter 3.)

⓯ The accessor function now picks out the appropriate column for the second data set.

⓰ To distinguish the two data sets more clearly, we want to connect the symbols belonging to each data set with straight lines. Lines are more complicated than symbols because each line (line segment) depends on *two* consecutive data points. D3 helps us here: the `d3.line()` factory function returns a function object, which, given a data set, produces a string suitable for the `d` attribute of the SVG `<path>` element. (See Appendix B to learn more about the `<path>` element and its syntax.)

⓱ The line generator requires accessor functions to pick out the horizontal and vertical coordinates for each data point.

⓲ Select the `<g>` element for the first data set by the value of its `id` attribute. An ID selector string consists of the hashmark `#`, followed by the attribute value.

⓳ A `<path>` element is added as child of the `<g>` group of the first data set…

⓴ … and its `d` attribute is set by invoking the line generator on the data set.

㉑ Instead of creating a new line generator from scratch, we reuse the existing one by specifying a new accessor function, this time for the second data set.

㉒ A `<path>` element for the second data set is appended at the appropriate place in the SVG tree and populated.

㉓ Because the fill style for the symbols of the second data set was defined on the parent element (not on the individual `<circle>` elements themselves) it is possible to change it in a single operation. Uncomment this line to make all circles for the second data set red. Only appearance options are inherited from the parent: it is not possible to change, for example, the radius of all circles in this way, or to turn circles into rectangles. These kinds of operations require touching every element individually.

At this point, you might begin to get a feel for what working with D3 is like. Sure, it is verbose, but much of it feels like an assembly-type job where you simply snap together premade components. The method chaining, in particular, can resemble the construction of a Unix pipeline (or playing with LEGO, for that matter). The components themselves tend to emphasize mechanism over policy: that makes them reusable across a wide range of intended purposes, but leaves a greater burden on the programmer or designer to create semantically meaningful graphical assemblies. One final aspect that I would like to emphasize is that D3 tends to defer decisions in favor of "late binding": for example, in the way that it is customary to pass accessor functions as arguments, rather than requiring that the

appropriate columns be extracted from the original data set before being passed to the rendering framework.

Adding Graph Elements with Reusable Components

Figure 2-2 is bare-bones: it shows the data but nothing else. In particular, it doesn't show the scales—which is doubly important here, because the two data sets have quite different numerical ranges. Without scales or axes it is not possible to read quantitative information from the graph (or any graph, for that matter). We therefore need to add axes to the existing graph.

Axes are complex graphical elements because they must manage tick marks and tick labels. Thankfully, D3 provides an axis facility that will, given a scale object (which defines domain and range and the mapping between them), generate and draw all the required graphical elements. Because the visual axis component consists of many individual SVG elements (for the tick marks and labels), it should always be created inside its own `<g>` container. Styles and transformations applied to this parent element are inherited by all parts of the axis. This is important because all axes are initially located at the origin (the upper-left corner) and must be moved using the `transform` attribute to their desired location. (Axes will be explained in more detail in Chapter 7.)

Besides adding axes and new D3 functionality for generating curves, Example 2-6 also demonstrates a different style of working with D3. The code in Example 2-5 was very straightforward, but also verbose, and included a great deal of code replication: for example, the code to create the three different scale objects is almost identical. Similarly, the code to create symbols and lines is mostly duplicated for the second data set. The advantage of this style is its simplicity and linear logical flow, at the cost of higher verbosity.

In Example 2-6 there is less redundant code because duplicated instructions have been pulled into local functions. Because these functions are defined inside of `makeDemo3()`, they have access to the variables in that scope. This helps to keep the number of arguments required by the local helper functions small. This example also introduces *components* as units of encapsulation and reuse, and demonstrates "synthetic" function invocation—all important techniques when working with D3.

Example 2-6. Commands for Figure 2-3

```
function makeDemo3() {
    d3.tsv( "examples-multiple.tsv" )
        .then( function( data ) {
            var svg = d3.select( "svg" );                                    ❶

            var pxX = svg.attr( "width" );                                   ❷
            var pxY = svg.attr( "height" );

            var makeScale = function( accessor, range ) {                    ❸
                return d3.scaleLinear()
                    .domain( d3.extent( data, accessor ) )
                    .range( range ).nice();
            }
            var scX  = makeScale( d => d["x"],  [0, pxX] );
            var scY1 = makeScale( d => d["y1"], [pxY, 0] );
            var scY2 = makeScale( d => d["y2"], [pxY, 0] );

            var drawData = function( g, accessor, curve ) {                  ❹
                // draw circles
                g.selectAll( "circle" ).data(data).enter()
                    .append("circle")
                    .attr( "r", 5 )
                    .attr( "cx", d => scX(d["x"]) )
                    .attr( "cy", accessor );

                // draw lines
                var lnMkr = d3.line().curve( curve )                         ❺
                    .x( d=>scX(d["x"]) ).y( accessor );

                g.append( "path" ).attr( "fill", "none" )
                    .attr( "d", lnMkr( data ) );
            }

            var g1 = svg.append( "g" );                                      ❻
            var g2 = svg.append( "g" );

            drawData( g1, d => scY1(d["y1"]), d3.curveStep );                ❼
            drawData( g2, d => scY2(d["y2"]), d3.curveNatural );

            g1.selectAll( "circle" ).attr( "fill", "green" );                ❽
            g1.selectAll( "path" ).attr( "stroke", "cyan" );

            g2.selectAll( "circle" ).attr( "fill", "blue" );
            g2.selectAll( "path" ).attr( "stroke", "red" );

            var axMkr = d3.axisRight( scY1 );                                ❾
            axMkr( svg.append("g") );                                        ❿

            axMkr = d3.axisLeft( scY2 );
```

```
            svg.append( "g" )
                .attr( "transform", "translate(" + pxX + ",0)" )  ⓫
                .call( axMkr );                                   ⓬

            svg.append( "g" ).call( d3.axisTop( scX ) )
                .attr( "transform", "translate(0,"+pxY+")" );     ⓭
        } );
}
```

❶ Select the `<svg>` element to draw on and assign it to a variable so that it can be used later without having to call `select()` again.

❷ Next, query the `<svg>` element for its size. Many D3 functions can work as setters as well as getters: if a second argument is supplied, the named property is set to the specified value, but if the second argument is missing, the current value of the property is returned instead. We use this feature here to obtain the `<svg>` element's size.

❸ The `makeScale()` function is simply a convenient wrapper to cut down on the relative verbosity of the D3 function calls. Scale objects are already familiar from the previous listing (Example 2-5). The `nice()` function on a scale object extends the range to the nearest "round" values.

❹ The `drawData()` function bundles all commands necessary to plot a single data set: it creates both the circles for individual data points as well as the lines connecting them. The first argument to `drawData()` must be a `Selection` instance; typically, a `<g>` element as container for all the graphical elements representing a single data set. Functions that take a `Selection` as their first argument and then add elements to it are known as *components* and are an important mechanism for encapsulation and code reuse in D3. This is the first example we see; the axis facility later in this example is another. (See Chapter 5 for the full discussion.)

❺ The `d3.line()` factory is already familiar from Example 2-5. It can accept an algorithm that defines what kind of curve should be used to connect consecutive points. Straight lines are the default, but D3 defines many other algorithms as well—you can also write your own. (See Chapter 5 to learn how.)

❻ Create the two `<g>` container elements, one for each data set.

❼ Invoke the `drawData()` function while supplying one of the container elements, as well as an accessor describing the data set and the desired curve shape. Just to demonstrate what capabilities are available, the two data sets are drawn using different curves: one with step functions and the other with natural cubic splines. The `drawData()` function will add the necessary `<circle>` and `<path>` elements to the `<g>` container.

❽ For each container, select the desired graphical elements to set their color. Although it would have been easy enough to do so, choosing the color was quite intentionally *not* made part of the `drawData()` function. This reflects a common D3 idiom: creating DOM elements is kept separate from configuring their appearance options!

❾ The axis for the first data set is drawn on the left side of the graph; remember that by default all axes are rendered at the origin. The `axisRight` object draws tick labels on the *right* side of the axis, so that they are outside the graph if the axis is placed on the graph's right side. Here, we use it on the left side and allow for the tick labels to be inside the graph.

❿ The factory function `d3.axisRight( scale )` returns a function object that generates the axis with all its parts. It requires an SVG container (typically a `<g>` element) as argument, and creates all elements of the axis as children of this container element. In other words, it is a *component* as defined earlier. (See Chapter 7 for details.)

⓫ For the axis on the right side of the graph, the container element must be moved to the appropriate location. This is done using the SVG `transform` attribute.

⓬ This is new: instead of calling the `axMkr` function explicitly with the containing `<g>` element as argument, the `axMkr` function is passed as argument to the `call()` function instead. The `call()` function is part of the `Selection` API (see Chapter 3); it is modeled after a similar facility in the JavaScript language. It invokes its argument (which must be a function), while supplying the

current selection as argument. This form of "synthetic" function invocation is quite common in JavaScript, and it is a good idea to get used to it. One advantage of this style of programming is that it supports method chaining, as you can see in the next item…

⓭ … where we add the horizontal axis at the bottom of the graph. Just to show what's possible, the order of function calls has been interchanged: the axis component is invoked first, the transformation is applied second. At this point we have also dispensed with the ancillary `axMkr` variable. This is probably the most idiomatic way to write this code.[3]

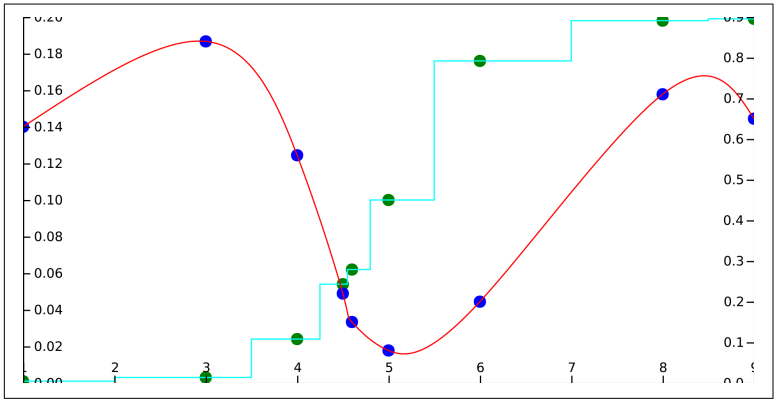

Figure 2-3. An improved plot, including coordinate axes and using different types of curves (compare Figure 2-2 and Example 2-6)

The resulting graph is shown in Figure 2-3. It is not pretty, but fully functional, showing two data sets together with their respective scales. The first step to improve the appearance of the graph would be to introduce some padding inside the SVG area, around the actual data, to make room for the axes and tick labels. (Try it!)

3 In fact, the `drawData()` function would typically be called this way, too: `g1.call( drawData, d => scY1(d["y1"]), d3.curveStep )`

A Third Example: Animating List Items

Our third and last example may appear a bit more whimsical than the other two, but it nevertheless demonstrates two important points:

- D3 is not limited to generating SVG. On the contrary, it can manipulate *any* part of the DOM tree. In this example, we will use D3 to manipulate ordinary HTML list elements.
- D3 makes it easy to create responsive and animated documents; that is, documents that can respond to user events (such as mouse clicks) and that change their appearance over time. Here, we will only give the briefest preview of the amazing possibilities; we will revisit the topic in much more detail in Chapter 4.

Creating HTML Elements with D3

For a change, and because the actual script is very short, the JavaScript commands are included directly in the page and are not kept in a separate file. The entire page, including the D3 commands, for the current example is shown in Example 2-7 (also see Figure 2-4).

Example 2-7. Using D3 for nongraphical HTML manipulation (see Figure 2-4)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script src="d3.js"></script>
  <script>
function makeList() {
    var vs = [ "From East", "to West,", "at Home", "is Best" ];    ❶

    d3.select( "body" )                                           ❷
        .append( "ul" ).selectAll( "li" )                         ❸
        .data(vs).enter()                                         ❹
        .append("li").text( d => d );                             ❺
}
  </script>
</head>

<body onload="makeList()" />
</html>
```

Structurally, this example is almost equivalent to the example that opened this chapter (in particular Example 2-3), but with a few differences in detail:

❶ The data set is not loaded from an external file but is defined inside the code itself.

❷ The HTML `<body>` element is selected as the outermost container of interest.

❸ The code appends an `<ul>` element and then creates an empty placeholder for the list items (using `selectAll( "li" )`).

❹ As before, the data set is bound to the selection, and the collection of data points without matching DOM elements is retrieved.

❺ Finally, a list item is appended for each data point, and its contents (which, in this example, is the text for the list item) are populated with values from the data set.

All of this is quite analogous to what we have done before, except that the resulting page is plain, textual HTML. D3 turns out to be a perfectly good tool for generic (that is, nongraphical) DOM manipulations.

Figure 2-4. A bulleted list in HTML (see Examples 2-7 and 2-8)

Creating a Simple Animation

It does not take much to let this document respond to user events. Replace the `makeList()` function in Example 2-7 with the one in Example 2-8. You can now toggle the color of the text from black to red and back by clicking on a list item. Moreover, the change won't take effect immediately; instead, the color of the text will change continuously over a couple of seconds.

Example 2-8. Animation in response to user events

```
function makeList() {
    var vs = [ "From East", "to West,", "at Home", "is Best" ];

    d3.select( "body" )
        .append( "ul" ).selectAll( "li" )
        .data(vs).enter()
        .append("li").text( d => d )                                    ❶
        .on( "click", function () {                                     ❷
            this.toggleState = !this.toggleState;                       ❸ ❹
            d3.select( this )                                           ❺
                .transition().duration(2000)                            ❻
                .style("color", this.toggleState?"red":"black");        ❼
        } );
}
```

❶ Up to this point, the function is identical to the one from Example 2-7.

❷ The `on()` function registers a callback for the named event type (`"click"` in this case), with the current element as the DOM `EventTarget`. Each list item can now receive click events and will pass them to the supplied callback.

❸ We need to keep track of the current toggle state separately for each list item. Where else to keep this information than on the element itself? JavaScript's permissive nature makes this supremely easy: simply add a new member to the element! D3 assigns the active DOM element to `this` before invoking the callback, and so provides access to the current DOM element.

❹ This line makes use of JavaScript's permissive nature in another way as well. The first time the callback is invoked (for each list item), the `toggleState` has not been assigned yet. It therefore has the special value `undefined`, which evaluates to `false` in a Boolean context, making it unnecessary to initialize the variable explicitly.

❺ In order to operate on it using method chaining, the current node needs to be wrapped in a selection.

❻ The `transition()` method interpolates smoothly between the current state of the selected elements (the current list item, in

this case) and its desired final appearance. The transition interval is specified as 2000 milliseconds. D3 can interpolate between colors (via their numerical representation) and many other quantities. (Interpolations will be discussed in Chapter 7.)

❼ Finally, the new text color is selected based on the current state of the status variable.