In this two-article series, learn how to use Scalable Vector Graphics (SVG) with the open source D3 JavaScript library to create data visualizations. Shapes, colors, and layouts can be of great help in making business sense out of data volumes. An example scenario demonstrates the use of SVG and D3 for creating informative graphics out of browsing metrics for social media.

View more content in this series

This article is the first in a two-article series that demonstrates visualization techniques that can help you extract information of business value from data. You'll see how to use Scalable Vector Graphics (SVG) and the open source D3 JavaScript library to create browser-viewable visualizations that convey information through shapes and colors. I'll demonstrate the techniques with examples that visualize browsing metrics that are related to social media usage. Here in Part 1, I present an overview of how SVG and D3 work together, along with some basic examples. Part 2 delves more deeply into the visualization capabilities of this powerful combination of open-standards technologies.

Analyzing social media metrics

The social media big data challenge

Social media sites such as Twitter, Facebook, and YouTube provide comprehensive web service interfaces that expose their features. For example, the YouTube Data API enables applications to upload a video to YouTube or play an existing YouTube video on a website. Now these sites are developing analytic APIs too. The YouTube Analytics API, for example, provides statistics such as number of views and number of likes to programmatic clients. The upshot is that more business applications can interact with social media through visual and programmatic interfaces. The natural next challenge for companies of all sizes is to make the best business use of large social data volumes through big data analytics. Data visualization — one component of the whole analytics scene — is the focus of this article series. Learn about the comprehensive analytics capabilities of the IBM big data platform's IBM InfoSphere Streams and IBM InfoSphere BigInsights products.
A creative way for a company to understand customer behavior is to present ideas through social media and engage potential clients in an interactive discussion. Interaction on social media mirrors two-way human interaction: To understand people's likes and dislikes, you must listen to them, just as you do with anyone you want to interact with in a positive manner.

Take the hypothetical scenario of a home decor company that publishes open content in the form of blogs, videos, Facebook pages, and discussion forums. This content presents the company's ideas through social media resources and tries to initiate discussion and other forms of user interaction. The content caters to the tastes and preferences of individual customers and accordingly helps them navigate from one social resource to another. To judge changing customer trends and come up with new approaches and new designs, the company wants to analyze browsing data in three dimensions:

- Popularity, as indicated by the number of views of each social resource
- The number of users that are engaged in interaction on the resource
- The direction of users' navigation from one resource to another

Tables 1, 2 and 3 display sample data that shows the number of user views, user interaction, and navigation, respectively, that occurred over three weeks. Notice that the tables use color names to represent the types of social media resources (such as blogs and Facebook pages) that the company uses.

Table 1 shows the number of user views for each resource:

**Table 1. Number of user views for each social resource**

<table>
<thead>
<tr>
<th>Social resource</th>
<th>Blue</th>
<th>Gold</th>
<th>Green</th>
<th>Red</th>
<th>Maroon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>7057</td>
<td>7483</td>
<td>3749</td>
<td>3846</td>
<td>4598</td>
</tr>
<tr>
<td>Week 2</td>
<td>2371</td>
<td>7397</td>
<td>4589</td>
<td>2861</td>
<td>8249</td>
</tr>
<tr>
<td>Week 3</td>
<td>5972</td>
<td>5672</td>
<td>9152</td>
<td>9725</td>
<td>8983</td>
</tr>
</tbody>
</table>

You can see in Table 1 that the blue resource was viewed 7,057 times in week 1, and that the gold resource was the most-viewed resource during that week.

Table 2 shows the user interaction data:

**Table 2. Number of users that interact on each social resource**

<table>
<thead>
<tr>
<th>Social resource</th>
<th>Blue</th>
<th>Gold</th>
<th>Green</th>
<th>Red</th>
<th>Maroon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>2052</td>
<td>2089</td>
<td>1586</td>
<td>1426</td>
<td>2632</td>
</tr>
<tr>
<td>Week 2</td>
<td>2071</td>
<td>2190</td>
<td>7214</td>
<td>3782</td>
<td>2721</td>
</tr>
<tr>
<td>Week 3</td>
<td>3076</td>
<td>3190</td>
<td>4532</td>
<td>3825</td>
<td>4831</td>
</tr>
</tbody>
</table>

You can see from Table 2 that 2,052 users interacted on the blue resource in week 1, and that user interaction was highest on the maroon resource during the first week.

Table 3 shows the number of users that navigated from the blue resource to other resources:
Table 3. Navigation data for the blue social resource

<table>
<thead>
<tr>
<th>Social resource</th>
<th>Blue-to-Gold</th>
<th>Blue-to-Green</th>
<th>Blue-to-Red</th>
<th>Blue-to-Maroon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>3057</td>
<td>3483</td>
<td>8749</td>
<td>8456</td>
</tr>
<tr>
<td>Week 2</td>
<td>2371</td>
<td>7397</td>
<td>4589</td>
<td>2861</td>
</tr>
<tr>
<td>Week 3</td>
<td>5972</td>
<td>5672</td>
<td>9152</td>
<td>9725</td>
</tr>
</tbody>
</table>

You can see from Table 3 that 3,057 users navigated to the gold resource after they visited the blue resource in week 1, and that the red resource received the most audience from the blue resource.

Visualizations of browsing data

Visual representation provides an easier and quicker way to interpret large data volumes than tables of numbers. It's possible to represent the data in Tables 1, 2 and 3 graphically in many ways. Figure 1, for example, is a simple way to show the week 1 data in Table 1:

**Figure 1. Number of week 1 views for each social resource, represented by circles**

![Figure 1](image)

*Figure 1* represents the number of views of each resource as a circle. The circles' relative sizes are proportionate to the numbers they represent, so the most-viewed resource (gold) is represented by the largest circle. Each circle also shows the actual number of views that each resource received in week 1.

Figure 2 is a slight variant of *Figure 1* that uses a different layout of circles:
Figure 2. Week 1 views that are shown in a slightly different layout of circles

The popularity of each resource is just one dimension of data that the company wants to analyze. Figure 3 shows both the number of views of each resource and the number of users who interact on the resource in week 1 — two dimensions in one visual representation:

Figure 3. Circles within circles that show the number of user views and the number of users who interact in week 1

In Figure 3, the outer circle represents the total number of views of the resource, and the inner circle represents the number of users who interact on the resource. I have also put numbers in each circle to show the actual viewing and interaction data.

Figures 1, 2, and 3 show simple ways to combine colors, shapes, and actual figures to represent the data. By looking at Figure 3, you can see that the gold resource attracted the most user views in week 1, and the maroon resource the most interaction.
Using open technologies for data visualization

Dynamically generated JavaScript for visualization

Together, D3, DVG, and JavaScript form a complete browser-based data-visualization suite. To demonstrate their combined use in this series, I hardcoded example data in the sample code. Most JavaScript code in real-world projects is put together dynamically by server-side components. In big data analytic applications, such components might include clusters that are based on Apache Hadoop, web and application servers that host a variety of server-side modules, and database servers. Use of D3 to generate SVG code with the JavaScript coming from such sources is an elegant fit with modern business applications — such as IBM Business Process Manager (see Related topics) — that support JavaScript.

Present-day open standards and open source tools are powerful enough to support graphical representation of data. SVG, an open World Wide Web Consortium (W3C) standard, defines an XML-based format for drawing two-dimensional graphical objects. (See Related topics for a link to an introductory article about SVG.) SVG is supported by several browsers. I tested all of this article’s SVG and JavaScript code on Google Chrome.

The D3 library offers a bit of magic if you are working with data and SVG. It takes your data along with drawing instructions, relates your data to required SVG tags, and generates SVG code immediately that you can view in a browser. As D3 draws complex graphical objects, it keeps your data handy so you can go deeper and deeper into your drawing, one step at a time, and work with just the bit of data that is required to draw each small part of your drawing.

For the examples in this article and in Part 2, I progress from simple to complex drawings, explore the features of D3, and provide SVG and JavaScript code for each drawing. I start with the simplest case: SVG code that generates the circles that are shown in Figure 1.

SVG data representation

Listing 1 is the SVG code for drawing the circles in Figure 1:

Listing 1. SVG code that draws the circles in Figure 1

```xml
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/graphics/svg/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
width="1000" height="1000">
  <g transform="translate(70,143)">
    <circle r="70.57" style="fill: #000fff;"></circle>
    <text x="-10" fill="grey">7057</text>
  </g>
  <g transform="translate(216,143)">
    <circle r="74.83" style="fill: #fff555;"></circle>
    <text x="-10" fill="grey">7483</text>
  </g>
  <g transform="translate(329,143)">
    <circle r="37.49" style="fill: #aaf000;">
    </circle>
    <text x="-10" fill="grey">3749</text>
  </g>
</svg>
```
You can see that the root tag in Listing 1 is <svg>. The <svg> tag creates the canvas on which you use other SVG tags for drawing, also serving as a wrapper for those tags. The <svg> tag's width and height attributes specify the dimensions of the SVG canvas. The canvas that I've set with Listing 1 is 1,000 by 1,000 units.

The root <svg> tag has five <g> child tags, one for each circle in Figure 1. The <g> tag serves as a wrapper for the drawing that is done to create each circle and its accompanying text. Each <g> tag has a single attribute, named transform, whose value is translate (X, Y). The translate (X, Y) value determines the point around which the circle is drawn. For example, the transform attribute of the first <g> tag in Listing 1 puts the first circle's center at position 70, 143.

Each of the five <g> tags has a different X value for the transform attribute, and they all have the same Y value. As a result, the five circles are arrayed along a horizontal line. The <g> tags' different X values cause each circle to be drawn next to the preceding one, with no visible white space in between. In a moment, I'll show you the simple JavaScript for generating these values.

Further into Listing 1, notice that each of the <g> tags has two children — a <circle> tag and a <text> tag — which draw the circle and its accompanying text, respectively. Each <circle> tag has an r attribute, which defines the circle's radius. Notice that value of the r attribute for each <circle> tag is the number of views that is scaled down by a factor of 100. Each <circle> tag also has a style attribute, which specifies the color that fills the circle. I use six-digit Hex format to represent RGB (red green blue) codes for colors. (See Related topics more details on using colors.) The <text> tag wraps the text to be displayed inside each circle. The x="-10" attribute that I include with each <text> tag shifts the text a bit for better positioning. The <text> tag's fill attribute specifies the text color.

You can download all of this article's code examples and open each of them in a browser window. Try adding more circles or text to the SVG code in Listing 1 and see how the browser displays your SVG.

Using JavaScript and D3 to generate SVG

The JavaScript code in Listing 2 uses D3 to generate the SVG code in Listing 1:

Listing 2. Using D3 to generate the SVG for Figure 1

```html
<!DOCTYPE html>
<meta charset="utf-8">
<body>
```
I included comments to indicate the steps in Listing 2:

**Step 1:** The first step is to store user-views data in an array. I also set some variables to store the dimensions of the canvas and the colors that I want to use in my SVG drawing.
Step 2: I start using D3 with `d3.select("body").append("svg")`, which adds an `<svg>` tag to the body of an HTML page. Here I also set the width and height of the SVG canvas by calling the `.attr()` function.

Step 3: The three function calls `.selectAll().data().enter()` together form the strength and ease of use that comes with using D3:

- The `.selectAll("g")` call selects all the `<g>` child tags of the root SVG tag. Although it cannot select any now (because none exists), it knows what I am looking for.
- The `.data(views[week])` call gives one week of user-views data for all five social resources to D3.
- The `.enter()` call relates individual `<g>` tags with individual records of user-views data.

Step 4: The `.append("g")` call tells D3 to add enough `<g>` tags to provide one `<g>` tag for every record in the data. Because no `<g>` tags existed in the beginning, and every week in the data array has five records (one for each social resource), D3 adds five `<g>` tags to the root `<svg>` tag. The first `<g>` tag is internally associated with the first data record (that is, 7057 in the views-data array), the second `<g>` tag with the second record, and so on. D3 internally associates data with SVG tags, so I only need to pass on the data array once while authoring and needn't worry about associating the correct data with each individual SVG tag.

Step 5: I also must add a `transform` attribute for each of the five `<g>` tags, so I call the `.attr()` function. I need to call it only once, and D3 internally handles the loop to ensure that the `transform` attribute is added to each of the five `<g>` tags.

The `.attr()` function takes the name of the attribute that you are authoring (in this case, `transform`) as its first parameter. Recall from the discussion of Listing 1 that the `transform` attribute determines the circle's position, so its value must be calculated in a function. For this reason, the second parameter to the `.attr()` function is another function call, which itself takes two parameters: `d` and `i`.

D3 gives you this opportunity for a function call because you might need data that's associated with the `<g>` tag while authoring the attribute value. The `d` parameter holds the piece of data for the particular `<g>` tag for which you are currently authoring the `transform` attribute value (for example, 7057 for the first `<g>` tag that is associated with the record that represents user views for the blue social resource). The magic of D3 is that it internally handles correct data association with individual tags, so you don't need to worry about picking the right piece of data. The magic of D3 is that it internally handles correct data association with individual tags, so you don't need to worry about picking the right piece of data. You can focus on data processing while authoring your SVG.

The `i` parameter is a zero-based index into the data (for example, `i = 0` for the first `<g>` tag that is associated with the first record). You can see in this step that I use the `i` parameter to calculate an appropriate position for the `<g>` tag, thereby drawing the circle for the first (blue) social resource at the far left, then the second circle that's adjacent to the first, and so on.
If you try running the code for just steps 1 - 5, you see five `<g>` child tags along with `transform` attributes, as shown in the SVG code in Listing 3. (These `<g>` tags will not show a drawing in the browser window yet.)

**Listing 3. SVG code with just the five `<g>` tags**

```xml
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
width="1000" height="1000">
  <g transform="translate(70,143)">
  </g>
  <g transform="translate(216,143)">
  </g>
  <g transform="translate(329,143)">
  </g>
  <g transform="translate(404,143)">
  </g>
  <g transform="translate(489,143)">
  </g>
</svg>
```

**Step 6:** The remaining tasks are to add a `<circle>` tag along with a `<text>` tag to each of the five `<g>` tags. For this purpose, you call the `.append()` function twice (once for `<circle>` and once for the `<text>` tag).

Notice that I use `.attr("r", function(d){return d/100})` to author a circle's `r` attribute (radius). In this case, I must know only the number of views of a particular resource to draw a circle of proportionate size. That's why I need only the `d` parameter and not the `i` parameter, so I omit `i`. The `d` parameter automatically contains the correct data (number of views), so I divide value of `d` by 100 to scale the size of the circle.

The D3-based JavaScript code is ready to draw the circles of Figure 1. If you open Listing2.html from the sample-code download in a browser window, you see that it generates the SVG of Listing 1 and displays the circles of Figure 1.

**A slightly different layout of circles**
You can slightly modify the code in Listing 2 to draw the layout of circles that's shown in Figure 2. Listing 4 shows the modified JavaScript:

**Listing 4. Modified code for drawing the layout of Figure 2**

```html
<!DOCTYPE html>
<meta charset="utf-8">
<body>
<script src="http://d3js.org/d3.v3.min.js"></script>
```

By comparing Listing 2 with Listing 4, you can see that the modification lies in the calculation of the `transform` attribute value, which is shown with bold highlighting in Listing 4.
Circle within a circle

Now look at Listing 5, which is the SVG code for drawing the circles of Figure 3 (circles-within-circles that represent views and interaction data simultaneously):

Listing 5. SVG code for Figure 3

```xml
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
width="1000" height="1000">
  <g transform="translate(100,170.57)">
    <circle r="70.57" style="fill: #0000ff;">
    </circle>
    <circle r="20.52" style="fill: #add8e6;">
    </circle>
    <text x="-15" y="35.52" fill="white">7057</text>
    <text x="-15" y="5">2052</text>
  </g>
  <g transform="translate(245.4,170.57)">
    <circle r="74.83" style="fill: #ffd700;">
    </circle>
    <circle r="20.89" style="fill: #ffff00;">
    </circle>
    <text x="-15" y="35.89" fill="white">7483</text>
    <text x="-15" y="5">2089</text>
  </g>
  <g transform="translate(245.4,282.89)">
    <circle r="37.49" style="fill: #008000;"></circle>
    <circle r="15.86" style="fill: #90ee90;"></circle>
    <text x="-15" y="30.86" fill="white">3749</text>
    <text x="-15" y="5">1586</text>
  </g>
  <g transform="translate(321.35,282.89)">
    <circle r="38.46" style="fill: #ff0000;"></circle>
    <circle r="14.26" style="fill: #f08080;"></circle>
    <text x="-15" y="29.26" fill="white">3846</text>
    <text x="-15" y="5">1426</text>
  </g>
  <g transform="translate(321.35,367.33)">
    <circle r="45.98" style="fill: #800000;"></circle>
    <circle r="26.32" style="fill: #cd5c5c;"></circle>
    <text x="-15" y="41.32" fill="white">4598</text>
    <text x="-15" y="5">2632</text>
  </g>
</svg>
```

You can see that Listing 5 has two `<circle>` and two `<text>` tags within each `<g>` tag, whereas Listing 1 has only one pair of `<circle>` and `<text>` tags. That's easily understandable because Figure 3 must draw two circles with the same center to give the view of a circle-within-a-circle, and each of the two circles has a number that is associated with it.

Listing 6 shows the D3-based JavaScript code for generating Figure 3:
Listing 6. JavaScript for drawing Figure 3

```html
<!DOCTYPE html>
<meta charset="utf-8">
<body>
<script src="http://d3js.org/d3.v3.min.js"></script>

<script>
var viewsAndInteraction = [
    [[7057, 2052], [7483, 2089], [3749, 1586], [3846, 1426], [4598, 2632]],
    [[5972, 2071], [5672, 2190], [9152, 7214], [9725, 3782], [8983, 2721]],
    [[8749, 3076], [4768, 3190], [6738, 4532], [9546, 3825], [6983, 4831]]
];
var width = 1000, height = 1000;
var viewColors = [
    "blue",
    "gold",
    "green",
    "red",
    "maroon"
];
var interactionColors = [
    "lightblue",
    "yellow",
    "lightgreen",
    "lightcoral",
    "indianred"
];
var week = 0, scale = 100;
var svg = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height)
    .selectAll("g")
    .data(viewsAndInteraction[week])
    .enter();
var g = svg.append("g")
    .attr("transform", function(d,i){
        var x = 10000, y = 10000;
        if (!(i%2)){
            for (var count=0; count<i; count++)
                x+=viewsAndInteraction[week][count][0];
        } else {
            for (var count=0; count<i; count++)
                y+=viewsAndInteraction[week][count][0];
        }
        return "translate(" + x/scale + "," + y/scale + ")"});
</script>
```
g.append("circle")
  .attr("r", function(d){return d[0]/scale})
  .style("fill", function(d, i){return viewColors[i];});

g.append("circle")
  .attr("r", function(d){return d[1]/scale})
  .style("fill", function(d, i){return interactionColors[i];});

g.append("text")
  .attr("x", -15)
  .attr("y", function(d){return (15 + d[1]/scale);})
  .attr("fill", "white")
  .text(function(d){return d[0];});

g.append("text")
  .attr("x", -15)
  .attr("y", 5)
  .text(function(d){return d[1];});

</script>
</body>

If you compare the JavaScript in Listing 2 with the code in Listing 6, you can see that almost everything is the same. The differences are that in Listing 6:

- You must work with views and interaction data, so a viewsAndInteraction array holds both views and interaction data.
- There are two pairs of g.append("circle") and g.append("text") function calls to draw two circles, each along with text at the same center.

You've now seen the SVG and JavaScript for all three of the figures. The next example visualizes navigation data.

**Visual representation of navigation data**

Figure 4 uses circular arcs and chords of different colors to show some navigation data from the first week: the number of users that navigate from the blue resource to the other resources:
The colored arcs in Figure 4 represent the different social resources. The blue resource connects to all the other resources through chords. Each chord represents navigation.

The blue chord starts and ends at the blue resource, representing the number of users who start browsing at the blue resource and don't navigate further. The gold chord, which starts at the blue resource and ends at the gold resource, represents the number of users that navigate from the blue resource to the gold resource. Similarly, Figure 4 depicts navigation from the blue resource to the remaining resources.

Notice that chords (starting with the blue one) are in ascending order, with lower numbers of users represented first, by thinner chords. So you don't need numbers to see which resources garner more users.

Using SVG to draw arcs and chords

Listing 7 shows partial SVG code for drawing Figure 4:

Listing 7. Partial SVG code for drawing Figure 4

```xml
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
width="500" height="500">
  <g transform="translate(250,250)">
    <path style="fill: #0000ff;"
      d="M1.1633760361312584e-14,-190A190,190 0 0,1
177.80498922385382,-66.97302298019123L140.372
9.18454765366783e-15,-150Z"></path>
  </g>
</svg>
```
You can see that the root `<svg>` tag in Listing 7 has one `<g>` child, which in turn has five `<path>` tags and another inner `<g>` child tag. The five `<path>` tags draw the five colored arcs. The inner `<g>` tag wraps SVG code for drawing chords, which I will explain in a minute. First, I explain how the `<path>` tags draw the five arcs.

### Drawing a set of colored arcs

Drawing a set of arcs with appropriate colors is simple with the SVG `<path>` tag, whose purpose is to define a drawing path, just like drawing on a paper with a pen. The path can be closed (for example, a triangle) or open (for example, a set of interconnected lines). You need a closed path for drawing an arc, as shown in Figure 5:

**Figure 5. Drawing an arc with a `<path>` tag**

![Figure 5. Drawing an arc with a `<path>` tag](image)

In Figure 5, the perimeter of the arc's closed path is drawn in black, and the closed path is filled with yellow. If you drew this arc with a pen, you'd draw the two curved and two straight black perimeter lines and then fill the interior with yellow.

Each of the five arcs in Figure 4 uses the same color for its perimeter and for filling it closed path. So you need five `<path>` tags, one for each arc of the circle.
In Listing 7, you can see that each of the five `<path>` tags has two attributes: `style` and `d`. The `style` attribute defines the color to be used for drawing, and the `d` attribute defines the drawing path.

The values of the `d` attributes (M8.572244476756641e-15, -140A140, 140 0 0, 1 93.95877067680189, 103.78703875197591L67.11340762628707, 74.13359910855422A100, 100 0 0, 0 6.1230317691118866e-15, -100Z, for example) seem long and complex. This format is required by the official SVG specification for this value to define the exact drawing path.

The format consists of an `M` followed by two comma-separated numbers, then an `A` followed by several numbers, then an `L` followed by two comma-separated numbers, then another `A` followed by several numbers, and finally a `Z`. This sequence means: **Move to the point defined by the numbers after `M`, then draw an Arc defined by numbers after `A`, then draw a Line to the point defined by numbers after `L`, then draw another Arc (the second arc). The `Z` at the end means go back to the initial position, thus forming a closed path.**

The good news is that you don't need to know any further details of how this sequence accurately draws the required arcs — because all these values were calculated using the D3 library, which contains easy-to-use functionality for drawing complex figures. In a moment, I will show you the D3 JavaScript code that performs all drawing calculations and generated the complete SVG code for Figure 4 dynamically.

The five `<path>` tags for each circle are all that you need to draw arcs. Open the Listing7.svg file (see Download) in a browser to see colored arcs without any chords, as shown in Figure 6:

**Figure 6. Colored arcs**
Drawing the chords

Like the arcs, chords are drawn with a `<g>` tag that's a wrapper for five `<path>` tags, one for each chord, as shown in Listing 8:

Listing 8. SVG code for drawing the set of five chords

```xml
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
     width="500" height="500">
  <g transform="translate(250,250)"

     <!--Drawing SVG code for arcs goes here. -->

     <g>
         <path d="M9.18454765366783e-15,-150A150,150 0 0,1
               19.52349842170555,-148.72401624948697Q 0,0
               9.18454765366783e-15,-150Z"
               style="fill: #0000ff;">
         </path>
         <path d="M19.52349842170555,-148.72401624948697A150,150
               0 0,1 41.344235247425466,-144.1896466729066Q 0,0
               144.99722485611238,-38.41620470616511Q 0,0
               19.52349842170555,-148.72401624948697Z"
               style="fill: #ffd700;">
         </path>
         <path d="M111.39598065576133,-100.4536484839712A150,150
               0 0,1 140.3723599135688,-52.873439194887816A150,150
               0 0,1 84.8772383898757,123.6764412000974A150,150
               0 0,1 50.93706344958661,141.08655345968583Q 0,0
               111.39598065576133,-100.4536484839712Z"
               style="fill: #008000;">
         </path>
         <path d="M-149.71409635840777,9.256854302914952A150,150
               0 0,1 -140.64142529945457,-52.1535194887816Q 0,0
               68.6764412000974,-133.3549640024362A150,150
               0 0,1 111.39598065576133,-100.4536484839712Q 0,0
               -149.71409635840777,9.256854302914952Z"
               style="fill: #ff0000;">
         </path>
         <path d="M-59.58349836433014,-137.65829696268898A150,150
               0 0,1 -1.6078040591602227e-13,-150Q 0,0
               41.344235247425466,-144.1896466729066A150,150
               0 0,1 68.6764412000974,-133.3549640024362Q 0,0
               -59.58349836433014,-137.65829696268898Z"
               style="fill: #800000;">
         </path>
     </g>
 </g>
</svg>

You can see that `<path>` tags for the chords in Listing 8 all (like the ones in Listing 7) have a long, complex value for the `d` attribute to be calculated by D3.

Listing 8 omits the `<path>` tags for the arcs. If you view the SVG code in Listing 8 in a browser, you'll see chords without arcs (and thus without the perimeter of the circle), as shown in Figure 7:
Now I'll show you the D3-based JavaScript code for generating Figure 4's SVG code.

**Using D3 to draw circles with chords**

You can see five steps in the JavaScript code in Listing 9, which uses D3 to author the SVG for Figure 4:

**Listing 9. Five steps for generating the SVG code in Figure 4**

```html
<!DOCTYPE html>
<meta charset="utf-8">
<body>
<script src="http://d3js.org/d3.v3.min.js"></script>
<script>

//Step 1:
var navigation = [
    [3057, 3483, 8749, 8465, 4598],
    [2371, 7397, 4589, 2861, 8249],
    [5972, 5672, 9152, 9725, 8983],
    [9763, 8462, 9782, 1953, 5182],
    [9567, 1571, 2895, 2783, 1874]
];

var navigationChord = d3.layout.chord()
    .matrix(navigation);

var chordCalculations = navigationChord.chords;

var colors = [
    "blue",
    "gold",
    "green",
    "red",
    "maroon"
];

var width = 500, height = 500, radius = 150, arcStroke = 40;

</script>
</body>
</html>
```
Step 1 stores the data and sets the variables. I store the navigation data in an array named navigation and then pass it on to d3.layout, which internally handles layout. I store calculations that are done by d3.layout in a variable named chordCalculations, which is used in step 5 to draw chords.

Step 2 is to author the <svg> wrapper and set its dimensions.

Step 3 is to add the <g> tag that serves as a wrapper for all the drawing tags.

Step 4 authors the five <path> tags for the arcs whose SVG code you saw in Listing 7. Use d3.svg.arc(), which internally makes all the calculations to author the values of the d attributes for all five <path> tags for the colored arcs.

Step 5 authors the <path> tags for the chords. I first put the chords in ascending order and then pass on chord calculations to D3's four magical steps: .selectAll().data().enter().append(). Finally, d3.svg.chord() authors the values of the d attributes of the <path> tags that draw the five chords.
Next time

In this article, I set the scene for discovering the full graphical power of SVG and D3 working together. Part 2 demonstrates the strength of D3 layouts, which can perform graphical calculations to draw multiple copies of a shape on a single SVG canvas. It also shows how to draw chart representations of the popularity, user-interaction, and navigation data. Finally, I show you how to put together the circles, arcs, chords, charts, and layouts in a single SVG drawing.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code</td>
<td>source-code.zip</td>
<td>10KB</td>
</tr>
</tbody>
</table>
Related topics

- **Scalable Vector Graphics**: Visit the official SVG site to find the SVG specification (including the list of officially recognized colors), the SVG Wiki, and other resources.
- **SVG on Wikipedia**: Read more about SVG.
- **Yellowpipe color converter**: This website offers a color format converter, where you can learn more about RGB and Hex representations of colors.
- "Scalable Vector Graphics in HTML5" (Jeremy Wischusen, developerWorks, January 2012): This article is a good introduction to SVG.
- **D3**: Visit the official D3 website and check out the eye-catching graphics in the examples gallery.
- **IBM ManyEyes**: Explore thousands of data visualizations or create your own. This experimental site is set up to allow the entire Internet community to upload data, visualize it, and talk about their discoveries with others.
- **D3**: Download the D3 JavaScript library.
- **IBM Business Process Manager**: Learn more about the IBM Business Process Manager family of products.
- **Faster than Real Time: Understanding shifts in digital consumer behavior**: This slide presentation shares ideas about learning customer behavior by analyzing big data.
- "Integrate social networks into BPM, Part 1: Collect Twitter data" (Bilal Siddiqui, developerWorks, May 2011) and "Integrate social networks into BPM, Part 2: Add Twitter data to CRM" (Bilal Siddiqui, developerWorks, June 2011): These articles discuss the integration of social media with BPM applications.
- **InfoSphere BigInsights**: Get a trial version of IBM InfoSphere BigInsights and manage and analyze massive volumes of structured and unstructured data at rest.
- **InfoSphere Streams**: Get a trial version of IBM InfoSphere Streams and build applications that rapidly ingest, analyze, and correlate information as it arrives from thousands of real-time sources.

© Copyright IBM Corporation 2013

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)