*Journal of Statistical Software*

*January 2012, Volume 46, Issue 1.* *http://www.jstatsoft.org/*

# Interactive and Animated Scalable Vector Graphics and R Data Displays

**Deborah Nolan**
University of California, Berkeley

**Duncan Temple Lang**
University of California,Davis

**Abstract**

We describe an approach to creating interactive and animated graphical displays using R's graphics engine and Scalable Vector Graphics, an XML vocabulary for describing two-dimensional graphical displays. We use the `svg()` graphics device inR and then post-process the resulting XML documents. The post-processing identifies the elements in the SVG that correspond to the different components of the graphical display, e.g., points, axes, labels, lines. One can then annotate these elements to add interactivity and animation effects. One can also use JavaScript to provide dynamic interactive effects to the plot, enabling rich user interactions and compelling visualizations. The resulting SVG documents can be embedded within HTML documents and can involve JavaScript code that integrates the SVG and HTML objects. The functionality is provided via the **SVGAnnotation** package and makes static plots generated viaR graphics functions available as stand-alone, interactive and animated plots for the Web and other venues.

*Keywords*: R graphics, interactive, animation, vector graphics, JavaScript.

## 1. Introduction

The way we view graphical representations of data has significantly changed in recent years. For example, viewers expect to interact with a graphic on the Web by clicking on it to get more information, to produce a different view, or control an animation. These capabilities are available in Scalable Vector Graphics (SVG, Eisenberg 2002), and in this article, we describe an approach within R (R Development Core Team 2011) to creating stand-alone interactive graphics and animations using SVG. SVG offers these capabilities through a rich set of elements that allow the graphics objects to be grouped, styled, transformed, composed with other rendered objects, clipped, masked, and filtered. SVG may not be the most ideal approach to interactive graphics, but it has advantages that come from its simplicity and increasing use on the Web and in publishing generally. The approach presented here uses

R's high-level plotting functions, R's SVG graphics device engine and the third-party cairo rendering engine (Cairo Graphics 2010) to create high-quality plots in SVG. Our tools allow users to post-process SVG documents/plots to provide the additional annotations to make the document interactive with, e.g., hyperlinks, tool tips, sliders, buttons, and animation to create potentially rich, compelling graphics that can be viewed outside of the R environment.

The **SVGAnnotation** package (Nolan and Temple Lang 2011) contains a suite of tools to facilitate the programmer in this process. The package provides "high-level" facilities for augmenting the SVG output from the standard plotting functions in R, including **lattice** (Sarkar 2011) and the traditional plotting functions. In addition to providing useful higher-levels facilities for modifying the more common R plots, the **SVGAnnotation** package also provides many utilities to handle other types of plots, e.g., maps and graphs. Of course, it is possible to generate interactive graphics from "scratch" in SVG by taking charge of all drawing, e.g., determining the coordinate system and drawing the axes, tick marks, labels, etc. However for complex data plots, we are much better off using this post-processing approach and leveraging R's excellent existing graphics facilities.

We do not want to suggest that the approach presented here is the definitive or dominant solution to the quest for general interactive graphics for R or statistics generally. Our focus is on making graphical displays created in R available in new ways to different audiences outside of R and within modern multi-media environments. This approach uses the XML (van Vugt 2007) parsing and writing facilities in the **XML** package (Temple Lang 2011b), and the availability of XML tools make it an alternative that we believe is worth exploring. While the popularity or uptake of this approach may be debatable due to prospects of SVG or implementations of the complete SVG specification in widely used browsers, what we have implemented can be readily adapated to other formats such as Flash (Adobe Systems Incorporated 2011) or the JavaScript canvas (Flanagan 2006). Hence this paper also provides ideas for future work with other similar graphical formats such as Flash, Flex MXML (Kazoun and Lott 2008), and the JavaScript canvas. The essential idea is to post-process regular R graphics output and identify and combine the low-level graphical objects (e.g., lines and text) into higher-level components within the plot (e.g., axes and labels). Given this association, we can then annotate components to create interactive, dynamic and animated plots in various formats.

The remainder of the paper is organized as follows. We start by explaining why SVG is a valuable format. We then give a brief introduction to several examples that illustrate different features of SVG and how they might be used for displaying statistical results. We move on to explore general facilities that one may use to add to an SVG plot generated within R. The examples serve as the context for the discussion of the functions in **SVGAnnotation** and the general approach used in the package. The facilities include adding tool tips and hyperlinks to elements of a display. Next, we introduce the common elements of SVG and examine the typical SVG document produced when plotting in R. This lays the ground work for using more advanced features in **SVGAnnotation**, such as how to handle non-standard R displays, animation, GUI components in SVG, and HTML forms. Additionally, we illustrate aspects of integrating JavaScript and SVG. We conclude the paper by discussing different directions to explore in this general area of stand-alone, interactive, animated graphics that can be displayed on the Web.

Several technologies in this paper (XML, SVG, and JavaScript) may be new to readers. For this reason, we provide an introduction to XML and also to JavaScript as appendices. It may

be useful for some readers to review these before reading the rest of this paper.

We also note that this paper and the figures can be viewed as an HTML document (provided in the supplements and at http://www.omegahat.org/SVGAnnotation/). The SVG displays within the HTML version are "live", i.e., interactive and/or animated. The reader may choose to switch to reading this version of the paper as it is a better medium for understanding the material.

## 2. Why SVG?

Scalable Vector Graphics (SVG) is an XML format for describing two dimensional (2-D) graphical displays that also supports interactivity, animation, and filters for special effects on elements within the display. SVG is a vector-based system that describes an image as a series of geometric shapes. This is in contrast to a raster representation that uses a rectangular array of pixels (picture elements) to represent what appears at each location in the display. An SVG document includes the commands to draw shapes at specific sets of coordinates. These shapes are infinitely scalable because they are vector descriptions, i.e., the viewer can adjust and change the display (for example, to zoom in and refocus) and maintain a clear picture. SVG is a graphics format similar to PNG, JPEG, and PDF. Many commonly used Web browsers directly support SVG (Firefox, Safari, Opera), and there is a plug-in for Internet Explorer. For example, Firefox can act as a rendering engine that interprets the vector description and draws the objects in the display within the page on the screen. There are also other non-Web-browser viewers for SVG such as **Inkscape** (Bah 2007) and **Squiggle** (Apache Software Foundation 2009), based on Apache's **batik** (Apache Software Foundation 2008). Similar to JPEG and PNG files, SVG documents can be included in HTML documents (they can also be in-lined within HTML content). However, quite differently from other image formats, SVG graphics, and their sub-elements, remain interactive when displayed within an HTML document and can participate as components in rich applications that interact with other graphics and HTML components. SVG graphics can also be included in PDF documents via an XML-based page description language named Formatting Objects (FO, Pawson 2002), which is used for high-quality typesetting of XML documents. SVG is also capable of fine-grained scaling. For these reasons, SVG is a rich and viable alternative to the ubiquitous PNG and JPEG formats.

The size of SVG files can be both significantly smaller and larger than the corresponding raster displays, e.g., PNG and JPEG. This is very similar to PDF and Postscript formats. For simple displays with few elements, an SVG document will have entries for just those elements and so be quite small. Bitmap formats however will have the same size regardless of the content as it is the dimensions of the canvas that determines the content of the file. As a result, for complex displays with many elements, an SVG file may be much larger than a bitmap format. Furthermore, SVG is an XML vocabulary and so suffers from the verbosity of that format. However, SVG is also a regular text format and so can be greatly compressed using standard compression algorithms (e.g., GNU zip). This means we cannot simply compare the size of compressed SVG files to uncompressed bitmap formats as we should compare sizes when both formats are compressed. JPEG files, for example, are already compressed and so direct comparisons are appropriate. Most importantly, we should compare the size of files that are directly usable. SVG viewer applications (e.g., Web browsers, **Inkscape**) can read compressed SVG (.svgz) files, but they typically do not read compressed bitmap formats. As

a result, compressed SVG files can be significantly smaller than comparable bitmap graphics, and so utilize less bandwidth and are faster to download.

As just mentioned, an SVG file is a plain-text document. Since it is a grammar of XML, it is also highly structured. Being plain text means that it is relatively easy to create, view and edit, and the highly structured nature of SVG makes it easy to modify programmatically. These features are essential to the approach described here: R is used to create the initial plot; we then programmatically identify the elements in the SVG document that correspond to the components of the plot, and augment the SVG document with additional information to create the interactivity and/or animation.

As a grammar of XML, SVG separates content and structure from presentation, e.g., the physical appearance of components in the presentation can be modified with a Cascading StyleSheet (Meyer 2004) without the need to change or re-plot the graphic. For example, someone who does not know R or even have the original data can change the contents of the CSS file that the SVG document uses, or substitute a different CSS file in order to, for example, change the color of points in the data region or the size of the line strokes for the axes labels. While this is non-trivial, it is feasible and certain annotations on the SVG content added by the **SVGAnnotation** package make this simpler. An additional feature of SVG is that elements of a plot can be grouped together and operated on as a single object. This makes it relatively easy to reuse visual objects and also to apply operations to them both programmatically and interactively. This is quite different from the style of programming available in traditional R graphics which draws on the canvas and does not work with graphical objects.

The SVG vocabulary provides support for adding interaction and animation declarations that are used when the SVG is displayed. However, one of the powerful aspects of SVG is that we can also combine SVG with JavaScript (also known as ECMAScript, Flanagan 2006). This allows us to provide interaction and animation programmatically during the rendering of the SVG rather than declaratively through SVG elements. Both approaches work well in different circumstances. Several JavaScript functions are provided in the **SVGAnnotation** package to handle common cases of interaction and animation. For more specialized graphical displays, the creator of the annotated SVG document may also need to write JavaScript code, which means working in a second language. For some users, this will be problematic, but this package and the **JRSONIO** package do provide some facilities to aid in this step, e.g., making objects in R available to the JavaScript code. (See `addECMAScripts()` and examples Example 7 and Example 8, amongst others.) While switching between languages can be difficult, we should recognize that the SVG plots are being displayed in a very different medium and use a language (JavaScript) that is very widely used for Web content.

The interactivity and animation described here are very different from what is available in more commonly used statistical graphics systems such as **GGobi** (Swayne *et al.* 2010), **iplots** (Urbanek and Wichtrey 2011), or **Mondrian** (Theus 2002). Each of these provide an interactive environment for the purpose of exploratory visualization and are intended primarily for use in the data analysis phase. While one could in theory build similar systems in SVG, this would be quite unnatural. Instead, SVG is mostly used for creating presentation graphics that typically come at the end of the data analysis stage. Rather than providing general interactive features for data exploration, we use SVG to create application-specific interactivity, including graphical interfaces for a display.

## 3. Examples of interactivity with SVG

To get a sense of the possibilities for interactivity with SVG, we first present a relatively comprehensive set of examples. Later sections give the details on how these are created. We begin with very simple examples that use high-level facilities in the **SVGAnnotation** package and proceed to more complex cases built using the utility functions also in the package. Some of these more advanced displays require a deeper understanding of how particular plots created in R are represented in SVG and an ability to write JavaScript.

The examples are intended to introduce the reader to the capabilities of SVG. They are also arranged to gradually move from high-level facilities to more technical details, i.e., they increase in complexity. The aim is to provide readers with sufficient concrete examples and accompanying code to develop SVG-based interactive displays themselves. When readers have

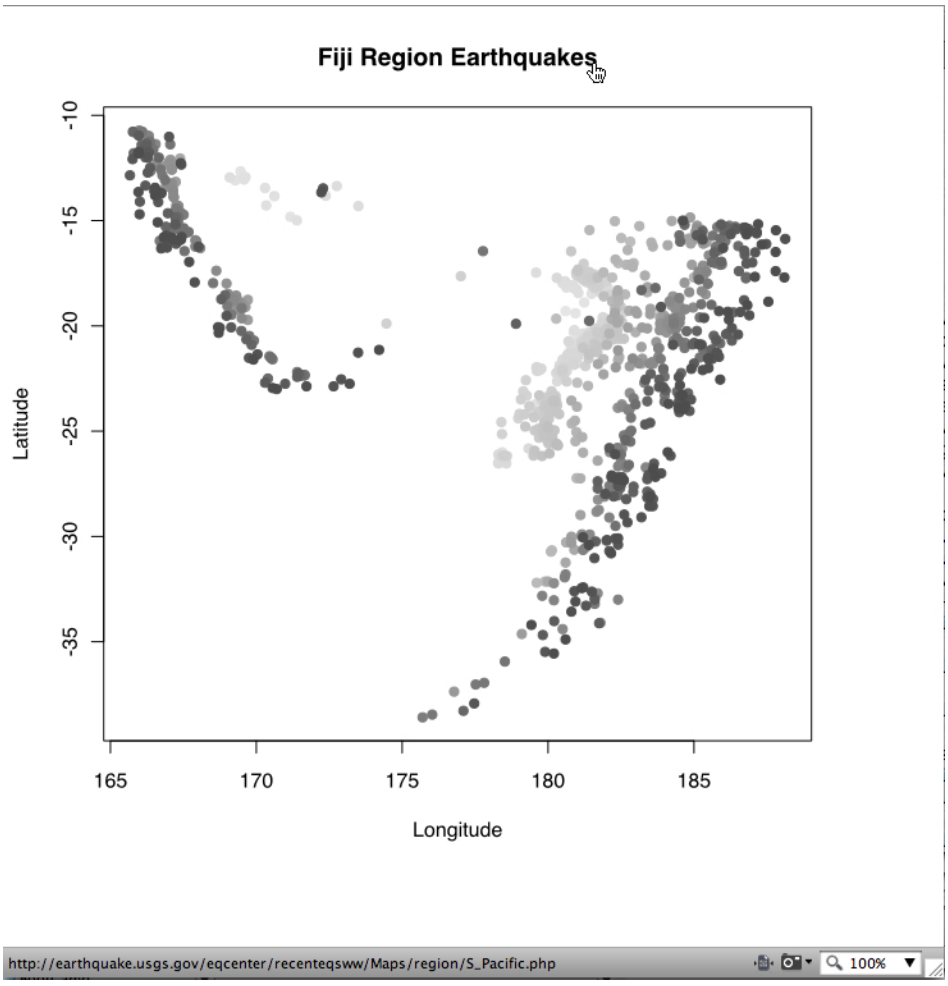


Figure 1: This scatter plot was produced by a call to the `plot()` function in R. After the plot was created, a hyperlink was added to the title of the plot using the `addAxesLinks()` function in **SVGAnnotation**. When viewed in a browser, a mouse click on the title will make the browser open up the url shown in the status bar of the screen shot. (This plot is adapted from Sarkar 2008.)

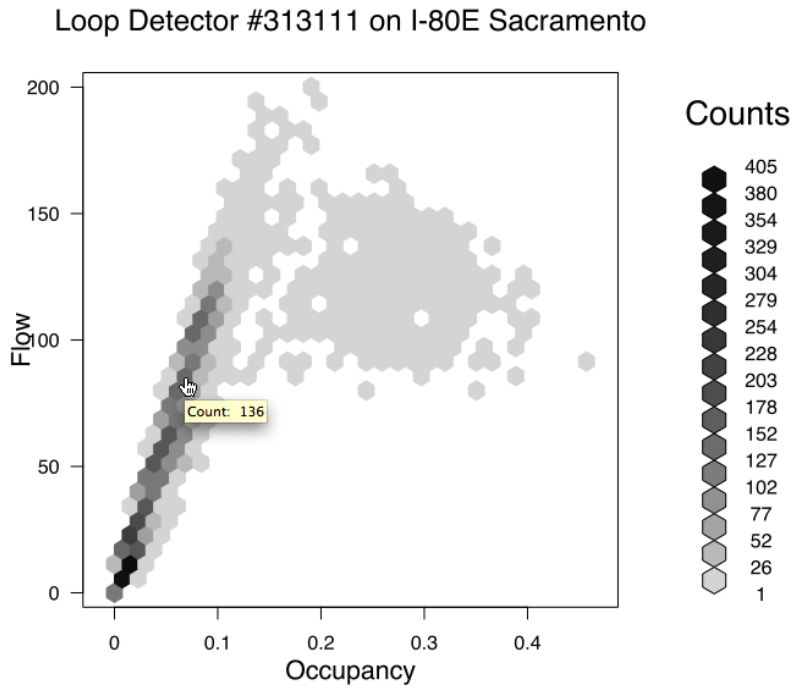


Figure 2: The hexagonal-bin plot shown here was generated by a call to the `hexbin()` function. After the plot was created, the SVG file was then post-processed with `addToolTips()` to add a tool tip to each of the shaded hexagonal regions. When the mouse hovers over a hexagon, the number of observations in that bin appears as a tool tip.

worked through these examples, they will hopefully have the facilities to create SVG and their own customized displays. The code for these and other examples appear in subsequent sections of the paper.

Tool tips and hyperlinks are very simple, but effective, forms of interactivity. With hyperlinks, the user interacts with a plot by clicking on an active region, say an axis label or a point, and in response, the browser opens the referenced Web page. An example of this is shown in the scatter plot in Figure 1. When the mouse moves over the title of the plot, the status bar at the bottom of the screen shows the URL of the USGS map of the South Pacific that will be loaded once the mouse is clicked.

With tool tips, the user interacts with a plot by pausing the pointer over a portion of the plot, say an axis label or a point. This "mouse-over" action causes a small window to pop up with additional information. Figure 2 shows an example where the mouse has been placed on an hexagonal bin in the plot and a tool tip has consequently appeared to provide a count of the number of observations in that bin. Note that these forms of interactivity do not require R or JavaScript within the browser; they merely require an SVG-compliant browser.

It is also possible to link observations across sub-plots. For example, Figure 3 shows the mouse hovering over a point in one plot. This mouse-over action causes that point and the corresponding point in a second plot to be colored red. When the mouse moves off the

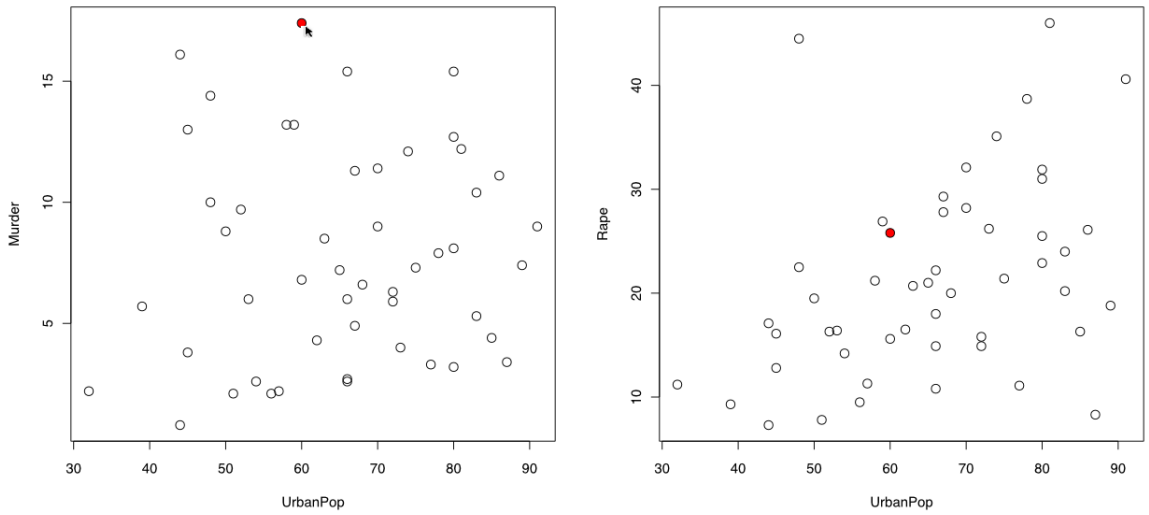


Figure 3: The points in the two scatter plots shown here are linked. When the mouse covers a point in one plot, that point changes color as does the point in the other plot that corresponds to the same observation. When the mouse moves off the point, then the linked points return to their original color(s). The `linkPlots()` function in **SVGAnnotation** takes care of embedding the necessary annotations and JavaScript code in the SVG file to change the color of the points.

point, both points return to their original color(s). Another example found later in this paper demonstrates how to link across conditional plots, where the mouse-over action on a plot legend causes the corresponding group of points to be highlighted in the panels (Figure 11). This linking of points across plots uses reasonably generic JavaScript code that is provided in the **SVGAnnotation** package.

SVG provides basic animation capabilities that can be used to animate R graphics. For example, it is possible to create animations similar in spirit to the well-known Gapminder animation (Rosling 2008), where points move across a canvas according to a time variable. See Figure 4 as an example. This animation is created by simply adding SVG elements/commands to the SVG content generated by R. In other words, no JavaScript code is needed to create the animation effects.

The Carto:Net project (Berger *et al.* 2010) has made available a graphical user interface (GUI) library for SVG. For convenience, this library is distributed as part of the **SVGAnnotation** package. It can be used to add controls such as sliders, radio boxes, and choice menus to an SVG plot. These controls provide an interface for the viewer to have more complex interactions with the graphic. For example, in Figure 5 the location of the slider thumb specifies the bandwidth used to smooth data. When the viewer moves the slider thumb, the corresponding smoothed curve is displayed in the left-hand plot, and the right-hand plot is updated to display the residuals from the newly fitted curve. The interactivity of the GUI controls are provided via JavaScript functionality in Carto:Net. Additional JavaScript is required to perform the plot-specific functionality, i.e., to display the correct curve and

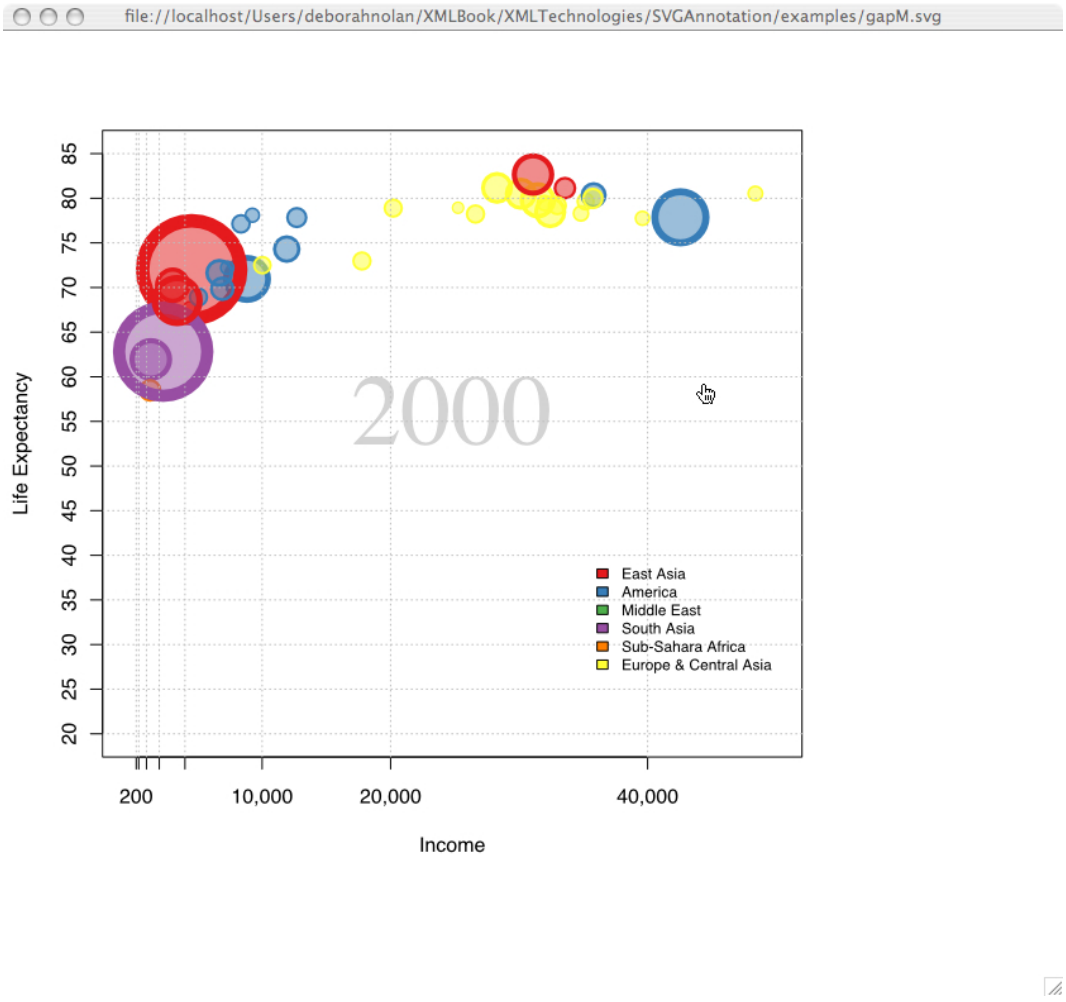


Figure 4: This screen shot shows a scatter plot at the end of an animation. Each circle represents one country. The location of the circle's center corresponds to (income, life expectancy) and the area is proportional to the size of the population. Time is represented through the movement of the circle center from the (x,y) value for one decade to the next. The `animate()` function in **SVGAnnotation** provides the basic functionality to create this scatter plot animation. Reloading the page will restart the animation. At the time of writing, this animation is only visible in the Opera browser (Opera Software ASA 2011).

residuals in the figure. We note that R is not involved when the plot is being viewed and the different curves are being displayed. This is done entirely via SVG and JavaScript because all fitted values are precomputed within R and serialized to the JavaScript code.

An alternative to SVG GUI controls is to embed the SVG graphic in an (X)HTML page and use controls provided by an HTML form to control the plot. Again, JavaScript is required to respond to the user input via the HTML controls. The basic set of HTML UI controls is quite limited, so the author must either use JavaScript to provide a slider (e.g., using the **YUI** toolkit from Yahoo! Inc. 2011) or change the interface to use a simpler control.

A complication from embedding an SVG document within an (X)HTML document stems from

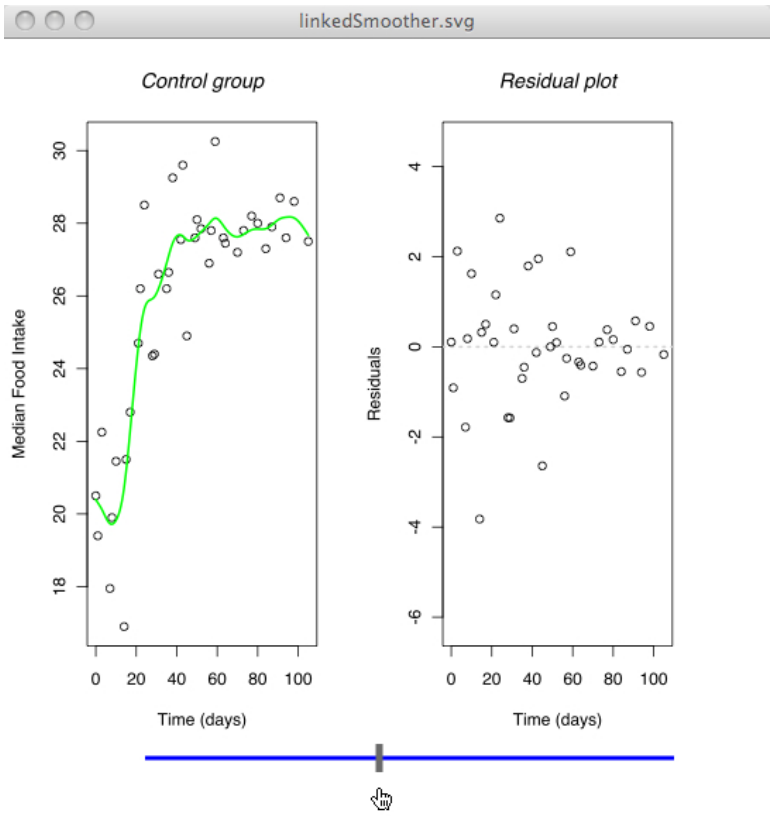


Figure 5: This pair of plots were made in R. Hidden within the plot on the left are curves from the fit for many different values of the smoothing parameter, and hidden in the right-hand plot are the residuals for each of those fits. The slider displayed across the bottom of the image is provided by Carto:Net (Berger *et al.* 2010). It is added to the SVG display using `addSlider()` in **SVGAnnotation**. When the viewer changes the location of the slider thumb, the corresponding curve and residuals are displayed and the previously displayed curve and residuals are hidden. JavaScript added to the SVG document responds to the change in the position of the slider thumb and takes care of hiding and showing the various parts of the two plots.

the JavaScript code within the HTML code being separate from the code and objects within the SVG document. In this case, a simple extra step is necessary to access the SVG elements from the JavaScript code within the HTML document. An example is shown in Figure 6. The map of the state-level outcome of the United States presidential election is embedded in an HTML `<div>` tag. When the viewer clicks on a state in the map, JavaScript code located in the HTML document pulls up the state's summary information and dynamically places it in a table in the region above the map.

These six figures show the variety of interactivity possible with SVG. In the following sections we introduce the functionality available within **SVGAnnotation** to create these and other interactive graphics. The examples are chosen to demonstrate our philosophy behind

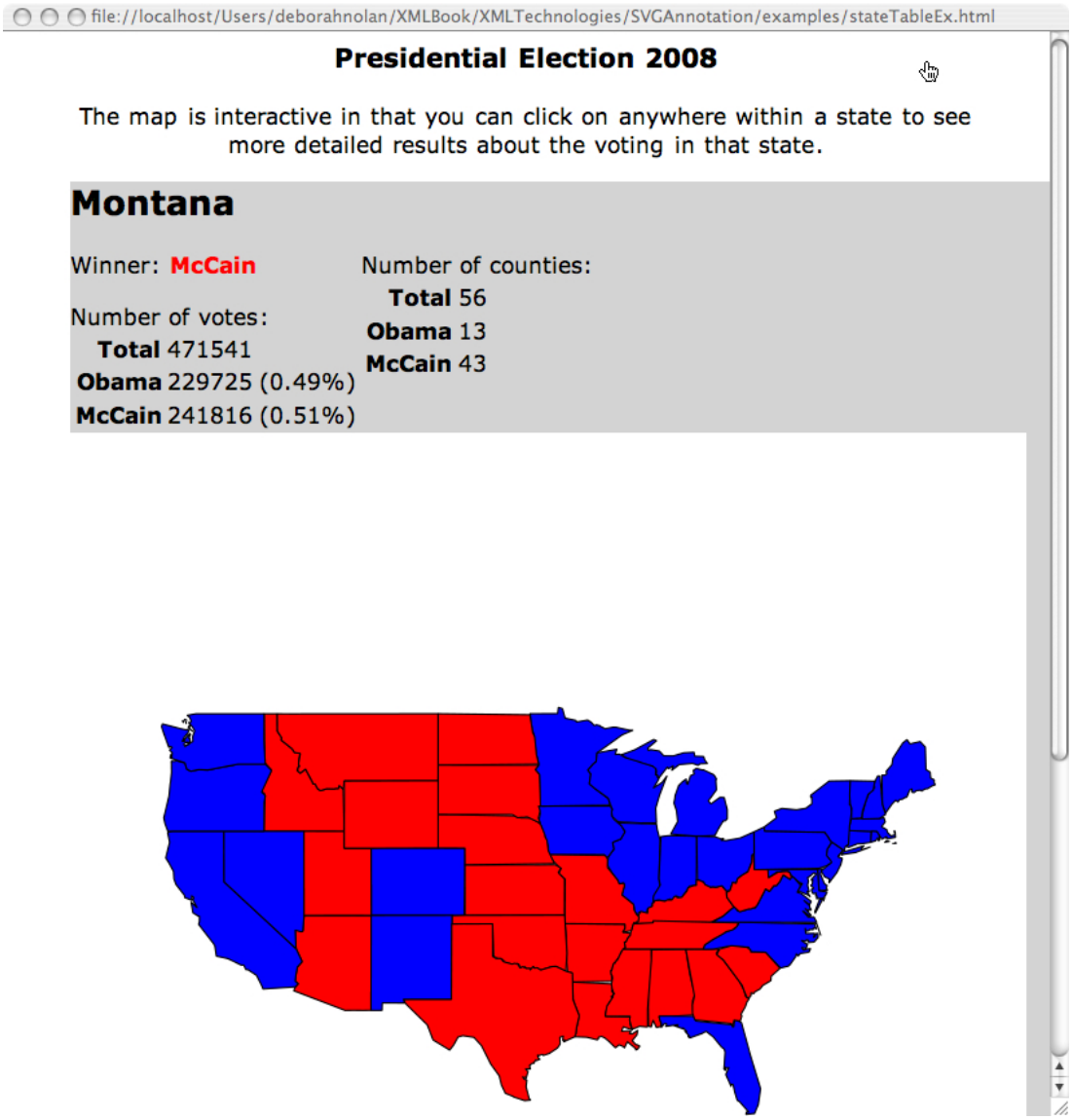


Figure 6: This screen shot shows a canonical red-blue map of the results of US presidential election embedded within a `<div>` tag in an HTML page. Interactivity within the map is controlled via JavaScript located in the HTML page. When the viewer clicks on a state, the table displaying the summary of votes in the state is rendered within the light-grey region above the map. Also, the county-level results are available in another HTML table (not shown in this screen shot).

creating interactivity with R and SVG. The package offers the R programmer a new mode of displaying R graphics in an interactive, dynamic manner on the Web. Tables 1 and 2 contain a comprehensive list of the 14 examples in this paper. The table includes a brief summary of the type of interactivity in the graphical display and the functions used to make the SVG interactive. These examples can be found in the XMLExamples directory in **SVGAnnotation**, and they can be run with `xmlSource()`, a function in the **XML** package, e.g., `xmlSource("exToolTipsAndLinks.xml")`.

| Ex. | Interactive features |
|---|---|
| 1 | Tool tips on points are added to a scatter plot using the high-level function `addToolTips()`. In addition, `addToolTips()` is used to add tool tips to the axis labels of the scatter plot. Source: `exToolTipsAndLinks.xml` |
| 2 | A hyperlink is added to the title in a scatter plot using the high-level function `addAxesLinks()`. Source: `exToolTipsAndLinks.xml` |
| 3 | Hyperlinks are added to regions in a map. This is accomplished by applying the `addLink()` function to the return value of the intermediate-level helper function `getPlotPoints()`. Source: `exLinkMap.xml` |
| 4 | Tool tips are added to the hexagonal bins in a hexbin plot. To do this, we use the helper function `getPlotPoints()` to locate the bins in the SVG document and then apply `addToolTips()` to the hexagonal bin elements in the SVG. Source: `exHexbinToolTips.xml` |
| 5 | The high-level `linkPlots()` function joins points in multiple scatter plots. This activity uses JavaScript to respond to the mouse-over action. The JavaScript code is supplied by `linkPlots()`. Source: `exLinkPlots.xml` |
| 6 | Points in **lattice** panels are linked to a legend for the plot. This customized plot is created using the `getPlotRegionNodes()` to access the panels in the plot, `getLatticeLegendNodes()` to access the legend, `addAttributes()` to add JavaScript calls in response to mouse-over events, and finally `addECMAScripts()` places the JavaScript code in the SVG document. Source: `exLegendLatticeLink.xml` |
| 7 | This plot requires greater understanding of the internal workings of the SVG document. It takes the approach of dynamically constructing and adding line segments to a scatter plot at the time of viewing. All of the computations of nearest neighbors within a set of observations are computed in R and stored as JavaScript variables in the SVG document (using `addECMAScripts()`). These JavaScript functions respond to mouse-over events at viewing time and create line segments connecting points in the scatter plot while the image is being viewed. The points have been annotated with unique identifiers using `getPlotPoints()` and `addAttributes()`. Source: `exKNN.xml` |
| 8 | In this example, a graph is annotated to respond to mouse-over events on the nodes of the graph. The edges and connecting nodes are brought to the forefront by a change in color when the mouse moves over the node. This interactivity is created in a similar manner as the nearest neighbor display. Source: `exGraphviz.xml` |
| 9 | The high-level function `animate()` adds to a scatter plot the capability of moving a point to different locations according to the change in the (x,y) values of the corresponding observation over different time intervals. The `animate()` function uses the animation features available in SVG, i.e., no JavaScript is needed to create the animation. Source: `exWorldAnimation.xml` |
| 10 | This example demonstrates how to use JavaScript facilities to animate a map. This hands-on approach uses a JavaScript timer to change the colors at regular intervals of states in a map of the US. Several helper functions are employed, including `getPlotPoints()`, `addECMAScripts()`, `addToolTips()` and `addAttributes()`. Source: `exJSAnimateElectionMapPaper.xml` |

Table 1: Examples of interactive SVG plots.

| Ex. | Interactive features |
|---|---|
| 11 | A graphical user interface (GUI) control is added to the canvas. The GUI is a slider that controls the bandwidth parameter for fitting a curve to the data. The slider is provided by Carto:Net, and is easily added to the display with a call to `addSlider()`. The connection between the slider and the plots is made via application specific JavaScript functions. Source: `exLinkedSmoother.xml` |
| 12 | Similar to the smoother example, SVG checkboxes are added to a time series plot. These checkboxes are also supplied by Carto:Net. The function `radioShowHide()` takes care of the details, and display-specific JavaScript is used to show or hide a particular time series curve in the plot. Source: `exEUseries.xml` |
| 13 | An earlier example (Example 6) is recreated using an HTML form to control the highlighting of points in panels of a **lattice** plot. The JavaScript functions that earlier responded to mouse events on the **lattice** legend, now are called when the viewer changes the selection in an HTML choice menu. The SVG image is embedded within the HTML document along with the form to control it. The JavaScript sits within the HTML document, rather than the SVG document as with the earlier example. Source: `exLatticeChoiceHTML.xml` |
| 14 | This example continues the ideas from Example 13 and uses mouse events on regions in a map (drawn in R) to display alternative HTML tables. Source: `exStateElectionTable.xml` |

Table 2: Examples of interactive SVG plots (continued).

## 4. Simple annotations

In this section, we turn our attention to how a user can create the simplest displays shown in Section 3. We introduce several high-level functions from the **SVGAnnotation** package that make it easy to add interactivity to SVG plots; these are briefly described in Table 3. We also explain the basic approach we take to create these graphical displays.

In order to produce SVG graphics in R (using the built-in graphics device), libcairo (Cairo Graphics 2010) must be installed when R is built. You can determine whether an R installation supports SVG with the expression `capabilities()["cairo"]`. If this yields `TRUE`, then the libcairo-based SVG support is present. Assuming this is the case, we create/open an SVG graphics device with a call to the `svg()` function and then issue R plotting commands to the SVG device as with any other graphics device. We must remember to close the device with a call to `dev.off()` when the commands to generate the display are complete. For example,

```
R> svg("foo.svg")
R> plot(density(rnorm(100)), type = "l")
R> abline(v = 0, lty = 2, col = "red")
R> curve(dnorm(x), -3, 3, add = TRUE, col = "blue")
R> dev.off()
```

creates the file named `foo.svg`, which contains the SVG that renders a smoothed density of 100 random normal observations.

The **SVGAnnotation** package provides a convenience layer to this process. The `svgPlot()` function opens the device, evaluates the code to create the plot(s), and then closes the device,

all in a single function call. For example,

```
R> svgPlot({
+    plot(density(rnorm(100)), type = "l")
+    abline(v = 0, lty = 2, col = "red")
+    curve(dnorm(x), -3, 3, add = TRUE, col = "blue")
+  }, "foo.svg")
```

performs the same function calls as in the above example. We recommend using `svgPlot()` because it inserts the R code that generated the plot as meta-data into the SVG file and provides provenance and reflection information. This can be convenient for the programmer who is post-processing the resulting SVG. Also important is the information `svgPlot()` adds to the SVG for **lattice** plots, such as the number of panels, strips, conditioning variables, levels, and details about the legend. This extra information allow us to more easily and reliably identify the SVG elements that correspond to the components of the R plot(s) we want to annotate.

An additional reason for using the `svgPlot()` function is that it can hide the use of a file. As shown below, often we want the SVG document generated by R's graphics commands as an XML tree and not written to a file. The return value of `svgPlot()` is a tree structure, if no file name is specified by the caller, e.g.,

```
R> doc <- svgPlot({
+    plot(density(rnorm(100)), type = "l")
+    abline(v = 0, lty = 2, col = "red")
+    curve(dnorm(x), -3, 3, add = TRUE, col = "blue")
+  })
```

The higher-level facilities in **SVGAnnotation** make it quite easy to add simple forms of interactivity to an SVG display. To add this interactivity, we follow a three-step process.

i *Plotting Stage:* First, we create the base SVG document. That is, we create an SVG graphics device and plot to it using R plotting tools. Again, we recommend using `svgPlot()` because it adds the plotting commands and other information to the SVG document.

ii *Annotation Stage:* Here, the SVG file created in the Plotting Stage is modified. That is, SVG content (and possibly JavaScript code) is added to the document to make the display interactive or animated. Depending on the type of plot to be annotated and the kind of annotations desired, the high-level functions described in this section may be able to handle all aspects of these annotations. Some cases, however, may require the intermediate functions in **SVGAnnotation** to, for example, add tool tips to a non-standard plot (see Section 8). Also, the annotation stage may possibly require working directly with the SVG document to, for example, add GUI controls to the plot (Section 12). All of these approaches use the **XML** package to parse and manipulate the SVG document from within R. We'll see examples of how this is used in later sections when we directly manipulate the SVG for more specialized annotations.

Once the annotation is completed, we save the SVG document to a file.

iii *Viewing Stage:* Now the enhanced SVG document is loaded into an SVG viewer, such as Opera, Firefox, Safari, or Squiggle, and the reader views and interacts with the image. In this stage, R is not available. The interactivity is generated by SVG elements themselves and/or JavaScript code that has been added in the annotation stage.

## 4.1. Simple interactivity: Tool tips and links

In this section we provide simple examples of how to use the high-level functions to add tool tips (Example 1) and hyperlinks (Example 2) to scatter plots. The R user does not need to understand much about SVG to add these simple features to plots.

**Example 1.** Adding tool tips to points and labels.

In this example, we demonstrate how to add tool tips to the SVG plot shown in Figure 7. The screen shot shows one of the effects that we are attempting to create: when the mouse is over a point, then a tool tip with additional information about that point appears.

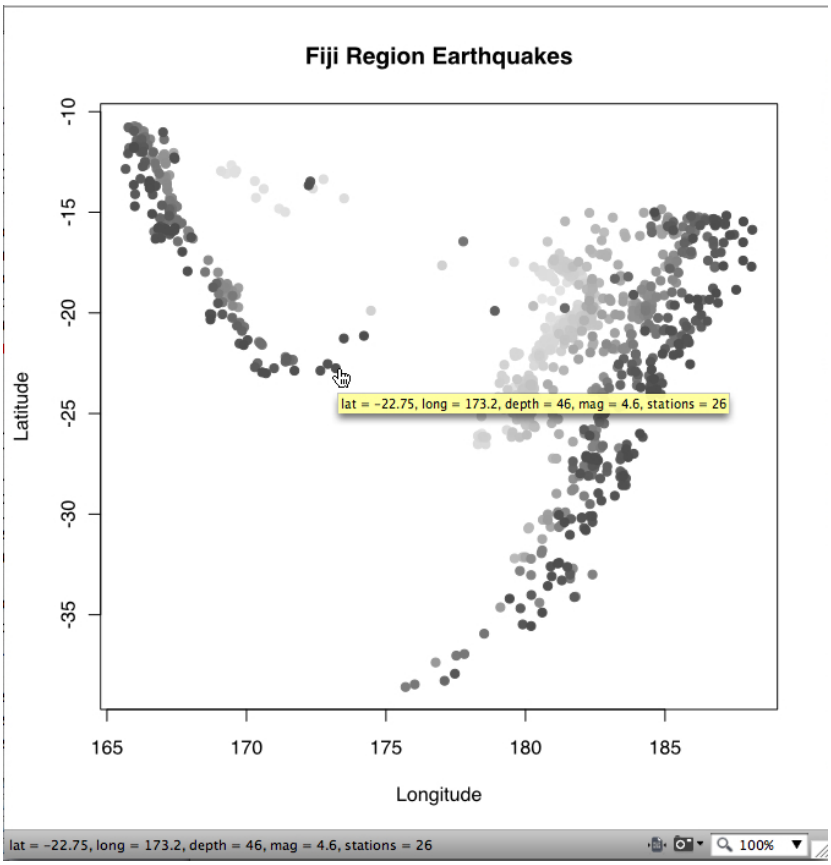


Figure 7: This screen shot shows a scatter plot where each point has a tool tip on it. The tool tips were added using the `addToolTips()` function in **SVGAnnotation**. When viewed in a browser, as the mouse passes over the point, information about the values of each variable for that observation is provided in a pop-up window. (Another screen shot of this plot is shown in Figure 1.)

The first step is to make the plot with the SVG graphics device.

```
R> depth.col <- gray.colors(100)[cut(quakes$depth, 100, label=FALSE)]
R> depth.ord <- rev(order(quakes$depth))
R> doc <- svgPlot(
+    plot(lat ~ long, data = quakes[depth.ord, ], pch = 19,
+    col = depth.col[depth.ord], xlab = "Longitude", ylab = "Latitude",
+    main = "Fiji Region Earthquakes")
+  )
```

As noted earlier, the function `svgPlot()` is a wrapper for R's own `svg()` function. We did not specify a value for the `file` parameter for `svgPlot()` and as a result, the function returns the parsed XML/SVG tree, which we can then post-process and enhance. (The code for this plot of `quakes` is adapted from Sarkar 2008.)

The default operation for `addToolTips()` is to add tool tips on each of the points in a scatter plot. We simply pass the SVG document to `addToolTips()` along with the text to be displayed. For example, adding the row names from the data frame as tool tips for the points is as simple as

```
R> addToolTips(doc, rownames(quakes[depth.ord, ]))
```

If we wanted the tool tip to provide information about the value of each of the variables for that observation, we could do this with

```
R> addToolTips(doc, apply(quakes[depth.ord, ], 1, function(x)
+    paste(names(quakes), x, sep = " = ", collapse = ", ")))
```

We can also use `addToolTips()` to provide tool tips on the axes labels. This time, since it is not the default operation, we need to pass the particular axes label nodes to be annotated rather than the entire SVG document. We find these axes label nodes using another function in **SVGAnnotation**, `getAxesLabelNodes()`. This function locates the nodes in the SVG document that correspond to the title and axes labels:

```
R> ax <- getAxesLabelNodes(doc)
```

The first element returned is the title. We discard the title node, e.g., `ax[ -1]`, and call `addToolTips()` to place tool tips on just the axis nodes:

```
R> addToolTips(ax[-1], c("Degrees east of the prime meridean",
+    "Degrees south of the equator"), addArea = TRUE)
```

The `addToolTips()` function will operate on any SVG element passed to it via the function's first argument. The `addArea` parameter is set to `TRUE` to indicate that the rectangle surrounding the characters in the label should be made active, i.e., when the mose moves into this region the tool tip will pop up. When `FALSE`, only the characters in the label will be responsive to the mouse movement.

The `addCSS` parameter of `addToolTips()` controls the addition to the document of a cascading style sheet for controlling the appearance of the tooltip rectangle. If `TRUE`, then the default

CSS is added. The default value for `addCSS` is `NA`, and in this case, the function determines whether or not the CSS is needed, e.g., if a tool tip is to be placed on a rectangular area surrounding the text in an axis label then a CSS is added. The code adds the specified default CSS file to the document only once. However, a warning message is issued if there are multiple requests to add the CSS to the document. The programmer also can add a specific CSS file via a call to `addCSS()`.

Now that the SVG has been annotated, we save the modified document:

```
R> saveXML(doc, "quakes_tips.svg")
```

This document can then be opened in an SVG viewer (e.g., a Web browser) and the user can interact with it by mousing over the points and axes labels to read the tool tips that we have provided.

**Example 2.** Adding hyperlinks to a plot title.

We sometimes want to click on a phrase or an individual point in a plot and have the browser jump to a different view or a Web page. SVG supports hyperlinks on elements so we can readily add this feature to R plots.

We continue with the plot that we annotated in Example 1 and add a hyper-link to the title. We add a feature to the plot that will allow viewers to click on the phrase "Fiji Region Earthquakes" and have their Web browser display the USGS web page containing a map of recent earthquakes in the South Pacific. We have already seen that the title of the plot is in the first element of the return value from the call to `getAxesLabelNodes()`, which was saved in the R variable `ax` (see note below). We simply associate the target URL with this element via a call to `addAxesLinks()` as follows:

```
R> usgs <- "http://earthquake.usgs.gov/eqcenter/recenteqsww/"
R> region <- "Maps/region/S_Pacific.php"
R> addAxesLinks(ax[[1]], paste(usgs, region, sep = ""))
```

Now, when the viewer clicks on the rectangular region that surrounds the title, the browser will open the USGS website.

*Note:* The parsed SVG document is a C-level structure, and the return value from `svgPlot()` is a reference to this structure. Further, the functions, such as `getAxesLabelNodes()`, return pointers to the nodes in this C-level structure. The consequence of this is that when we assign the return value from, say, `getAxesLabelNodes()` to an R object, we are not getting a new copy of the nodes. Hence, any modification to the returned value modifies the original parsed document. For example, in Example 2 an assignment to the R variable `ax` modifies the axes label nodes in the C-level structure. This is an important distinction from the usual way R handles assignments. The call to `saveXML()` will save the modified, parsed document as an XML file.

*Note:* Currently, many, but not all, common plots in R are handled at the high level described in this section. Some may require low-level manipulation of the XML in the same manner we have illustrated and implemented within the examples in the later sections of this paper and in the **SVGAnnotation** package.

**Example 3.** Adding hyperlinks to polygons.

In this example, we create a popular map of the USA where we color the states red or blue according to whether McCain or Obama, respectively, received the majority of votes cast in that state in the 2008 presidential election (see Figure 8). These data were "scraped" from the New York Times Web site http://elections.nytimes.com/2008/results/president/map.html. The data are total vote counts at the county level for each presidential candidate. We first aggregate the counts to the state level and determine the winner as follows:

```
R> stateO <- sapply(states, function(x) sum(x$Obama))
R> stateM <- sapply(states, function(x) sum(x$McCain))
R> winner <- 1 + (stateO > stateM)
```

We use `map()` in the **maps** package to draw the map, and `match.map()` to identify the polygonal regions so that we can color them correctly.

```
R> regions <- gsub("-", " ", names(winner))
R> stateInd <- match.map("state", regions)
R> polyWinners <- winner[stateInd]
R> stateColors <- c("#E41A1C", "#377EB8")[polyWinners]
R> doc = svgPlot({
+    map('state', fill = TRUE, col = stateColors)
+    title("Election 2008")
+  })
```

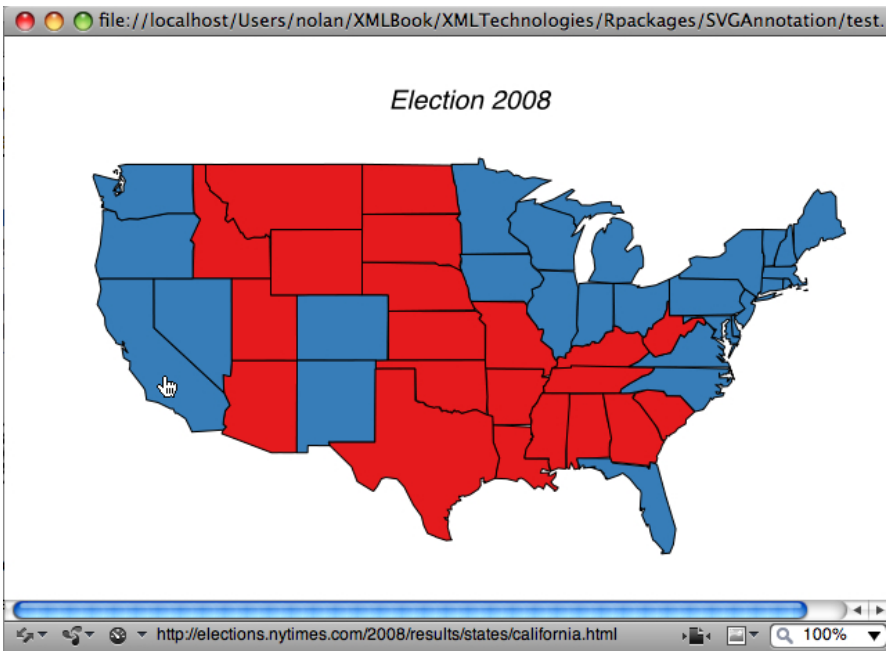


Figure 8: The map shown here was produced by the `map()` function in the **maps** package (Becker *et al.* 2011). The SVG file was then post-processed using the `addLink()` function in **SVGAnnotation** to add hyperlinks to the state regions. When the user clicks on a state, the browser links to the corresponding state's Web page of election results on the New York Times site, e.g., http://elections.nytimes.com/2008/results/states/california.

We then use `getPlotPoints()`, one of the facilities in the **SVGAnnotation** for handling SVG elements, to find the polygons regions for the states:

```
R> polygonPaths <- getPlotPoints(doc)
R> length(polygonPaths)

[1] 63

R> length(stateInd)

[1] 63
```

Note that there are more than 50 polygons because some states are drawn using multiple polygons, e.g., Long Island and Manhattan are drawn as separate polygons and belong to New York. Notice that the number of regions found matches the length of `stateInd`.

Now that we have `polygonPaths`, we can easily add links to the polygons in the SVG with `addLink()`. We simply provide a vector of target URLs to the function. These are created by pasting the state names to the New York Times base URL, as follows:

```
R> urls <- paste("http://elections.nytimes.com/2008/results/states/",
+    names(winner)[stateInd], ".html", sep = "")
```

Then we add the links:

```
R> addLink(polygonPaths, urls, css = character())
```

Note that the order of the polygon paths in the SVG document corresponds to the order in which the polygons were plotted in R.

When the saved document is viewed in a Web browser, or a dedicated SVG viewer such as **batik** (Apache Software Foundation 2008), a mouse click on a state will display the state's page on the New York Times Website.

Finally, we should note that if we had not filled the polygons with color, then the interiors would not be explicitly drawn and therefore, only the boundaries (i.e., the paths) would be active. This means that the interiors of the states would not respond to a mouse click.

We next annotate another slightly non-standard plot, the hexagonal bin plot. As in the previous example, `getPlotPoints()` finds the elements in the SVG document that correspond to the "points" in the plot. In general, the functions in **SVGAnnotation** are intelligent enough to handle many diverse types of plots and should be used in preference to low-level processing of nodes with XPath and `getNodeSet()`.

**Example 4.** Interactive hexagonal bin plots.

The goal of this example is to add tool tips to a hexagonal bin plot (Carr *et al.* 2009) such that mouse movement over a shaded hexagonal region results in the display of information about that bin, e.g., the number of points included in the bin (see Figure 2).

The data have been extracted from the Performance Measurement System (PeMS) Web site `http://pems.dot.ca.gov/`. These data provide the occupancy (percent) and flow (count)

of vehicles (measured in 5 minute intervals for one week) over one loop detector embedded beneath the surface of Interstate 80 in California. We begin by collapsing the measurements for the three lanes of traffic into one set of measurements.

```
R> library("SVGAnnotation")
R> data("traffic")
R> Occupancy <- unlist(traffic[ c("Occ1", "Occ2", "Occ3")])
R> Flow <- unlist(traffic[c("Flow1", "Flow2", "Flow3")])
```

We proceed to make the hexagonal bin plot and save the return value from the call to `hexbin()`, which is an S4 object.

```
R> library("hexbin")
R> hbin <- hexbin(Occupancy, Flow)
R> doc = svgPlot(
+    plot(hbin, main = "Loop Detector #313111 on I-80E Sacramento"))
```

The `count` slot in the `hbin` object is of interest to us because it contains a vector of cell counts for all of the bins, which we will use as the tool tips for the bins.

A call to `getPlotPoints()` locates the hexbins within the SVG.

```
R> ptz <- getPlotPoints(doc)
R> length(ptz)

[1] 276

R> length(hbin@count)

[1] 276
```

The length of `hbin@count` shows that there are 276 bins in the plot. This figure matches the number of "points" found in the SVG, which confirms that the bins have been correctly located by `getPlotPoints()`.

Now we can easily add tool tips to these regions in the plot as follows:

```
R> tips <- paste("Count: ", hbin@count)
R> addToolTips(ptz, tips, addArea = TRUE)
```

Additional annotations that might be included in the tool tip are the `xcm` and `ycm` slots in `hbin`, which hold the x and y coordinates (in data rather than SVG coordinates) for the center of mass of the cell.

### 4.2. Mouse events that change the style of graphics elements

Our examples so far in this section have added SVG in the post-processing stage to create interactive effects with tooltips and hyperlinks. Here, with the help of JavaScript, we create plots with more complex interactivity. We us `linkPlots()` to add JavaScript code to an SVG

display so that the style of an element can be programmatically changed while viewing it. Specifically, points in a scatter plot change color in response to a mouse event. In general, whether it is a point's color, visibility, location, or size, mouse actions can initiate the changing of SVG "on the fly" thus enabling a rich set of user interactions with the plot.

**Example 5.** Point-wise linking across plots.

In this example, we show how to use the high-level `linkPlots()` function to link points across plots (see Figure 3). We start by creating the collection of plots. We might use a simple call to the `pairs()` function to create a draftsman's display of pairwise-scatter plots. Alternatively, one can create the plots with individual R commands and arrange them in arbitrary layouts with `par()` or `layout()`. We'll use the latter approach to create two plots:

```
R> doc <- svgPlot({
+    par(mfrow = c(1,2))
+    plot(Murder ~ UrbanPop, USArrests, main="", cex = 1.4)
+    plot(Rape ~ UrbanPop, USArrests, main = "", cex = 1.4)
+  }, width = 14, height = 7)
```

The high-level function `linkPlots()` does the hard work for us:

```
R> linkPlots(doc)
R> saveXML(doc, "USArrests_linked.svg")
```

We can then view this and mouse over points in either plot to see the linking. The default color to change a point is red; alternative colors can be specified with `col`.

## 5. Dependency on the svg() function and libcairo

One of the features of the approach we describe is that we can post-process the output of an existing R graphics device to provide interactivity and animation. We do not have to replace the device with our own to intercept R graphics operations and assemble the results within the device. This is made possible and relatively easy because of the XML structure of the generated SVG. It is not nearly as simple to provide interactivity with binary formats such as PDF or rasterized formats such as PNG and JPEG.

The post-processing approach does however give rise to a potential problem. Since we post-process the output from the `svg()` function and associated device, we are exploiting a format and structure that may change. There are two layers of software, each of which may change. The first is the implementation of the SVG device in R. The second is the third-party C-level libcairo library on which the R graphics device is based. Since the generic R graphics engine and also the device structure are well-defined and very stable, it is unlikely that there will be significant changes to the format of the SVG generated by R-related code. Changes to libcairo are more likely to cause changes to the SVG. Very old versions of libcairo (e.g., 1.2.4) do yield SVG documents that are non-trivially different from more recent versions (e.g., version 1.10.0) and have caused errors in the **SVGAnnotation** package. Specifically, `<rect>` elements are used for describing rectangles rather than the generic `<path>`. Future versions of libcairo *may* introduce new changes to the format and cause issues for **SVGAnnotation**. We do not however expect significant changes.

The problem of relying on a particular format generated by libcairo is similar to interfacing to a particular application programming interface (API) of a C/C++ library. Any changes in that API will cause the interface to break. Ideally, the API is fixed. However, new major versions can lead to such breaks. The potential for the interface to break across new versions of the software do not make the interface useless. The situation here with libcairo is similar although somewhat more complex. We have to identify a change in the format and this might be subtle.

In many regards, the approach we have taken in designing **SVGAnnotation** is intended to cause minimal disruption to the existing tool chain. We do not require the user to deploy a different graphics device. We do not modify the code of the existing graphics device. Instead, we work to make sense of the output (e.g., identify shapes) rather than knowing the graphical operations, e.g., circle, polygon, rectangle, text. This introduces the perils of a change in the output structure, but is a worthwhile goal of direct software reuse. It synchronizes the annotation facilities to the primary SVG graphics device in use within R. The alternative is to provide our own device and generate the SVG content ourselves and control its format. This would protect us from changes in the format. However, we would not benefit from passively incorporating enhancements to the R graphics device or libcairo. To address this issue, we could use a "proxy" device that acts as a "front" for the regular SVG device. This device would identify the different R-level components and then forward the device calls to the existing SVG libcairo-based graphics device. This would help us to map the high-level R components of the graphical displays to the lower-level SVG content. This would be of marginal benefit and would require direct knowledge of the SVG graphics device C code. In many regards this would be more of a problem than the problem we are trying to address, i.e., potential changes to the SVG format.

This reliance on the format generated by the `svg()` function is an issue of which users should be aware. Changes to this format might make functions in the **SVGAnnotation** package either fail or create erroneous annotations. Such changes are likely to be firstly rare and secondly, relatively minor. The required changes to the **SVGAnnotation** package should be relatively easy to implement. Importantly, we are not expecting or claiming that the **SVGAnnotation** package will work for all plots generated in R. Instead, we are reporting an approach of post-processing the SVG content generated from R graphics commands. The **SVGAnnotation** package provides high-level functions for many types of plots, but not all. Users may have to deal with the SVG directly or be able to use some of the medium- and low-level functions to manipulate the content. The contribution of this work is the approach of post-processing arbitrary R plots and the scheme for mapping low-level graphical primitive operations/elements to higher level graphical components such as axes, data points, legends, etc.

## 6. The SVG grammar

In this section we provide a brief overview of the commonly used XML elements/nodes in SVG and the basics of the drawing model. For more detailed information on SVG, readers should consult (Eisenberg 2002), and readers unfamiliar with XML should read Appendix B or Harold and Means (2004). We also explore the basic layout of an SVG document created by the SVG device in R. In particular, we examine the SVG document produced from a simple call to `plot()` with two numeric vectors. If we wish to annotate other types of plots, i.e.,

one that is not covered by the high-level functions in **SVGAnnotation**, such as a ternary plot, then we will need to understand the structure of documents produced by the SVG device.

We include here a sample SVG file that will make our description of the SVG elements more concrete. This document was created manually (not with R) and is rendered in Figure 9.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns = "http://www.w3.org/2000/svg"
     xmlns:xlink = "http://www.w3.org/1999/xlink"
     width = "300pt" height = "300pt"
     viewBox = "0 0 300 300" version = "1.1">
 <defs>
  <g id="circles">
    <circle id = "greencirc" cx = "15" cy="15" r = "15" fill = "lightgreen"/>
    <circle id = "pinkcirc" cx= "50" cy="50"  r = "15" fill = "pink"/>
  </g>
  <style type="text/css">
   < ![CDATA[
   .recs {fill: rgb(50%, 50%, 50%); fill-opacity: 1; stroke: red;}
   ]] >
  </style>
 </defs>
 <g id = "main">
    <rect x = "10" y = "20" width = "50" height = "100" class = "recs"/>
    <use  x = "100" y = "100" xlink:href = "#circles"/>
    <use  x = "200" y = "50" xlink:href = "#circles" />
    <image xlink:href = "examples/pointer.jpg"
           x = "70" y = "50" width = "10" height = "10"/>
    <path style = "fill: rgb(100, 149, 237);
            fill-opacity: 0.5;stroke-width: 0.75;
            stroke-linecap: round; stroke-linejoin: round;
            stroke: rgb(0%,0%,0%); stroke-opacity: 1;"
        d = "M 102.8 174.6 L 102.8 174.6
             L 106.1 178.0 L 100.9 189.3 L 102.8 191.2
             L 102.1 195.1 L 102.1 209.9 L 53.9 209.9
             L 51.5 210.0 L 50.0 206.0 L 49.3 202.9 L 52.0 191.5
             L 52.7 185.0 L 52.9 184.1 L 53.4 179.0 L 53.3 173.0
             L 54.5 173.1 L 55.1 173.1 L 56.0 172.8 L 60.5 174.0
             L 61.4 177.2 L 63.5 178.4 L 67.7 177.5 L 72.5 177.7
             L 88.5 174.6 L 102.8 174.6
             Z"
          id = "oregon"/>
    <text x = "110" y = "200" fill = "navy" font-size = "15">
        Oregon
    </text>
 </g>
</svg>
```

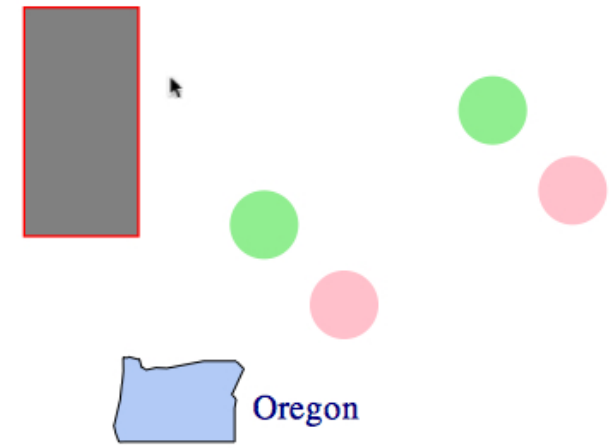


Figure 9: This simple SVG image is composed of a gray rectangle with a red border, two pairs of green and pink circles, a jpeg image of a pointer, a path that draws the perimeter of Oregon and fills the resulting polygon with blue, and the text "Oregon".

The image uses the most common SVG tags, which we describe below.

- SVG documents begin with the root tag `<svg>`. Possible elements it can have are: a `<title>` tag that contains the text to be displayed in the title bar of the SVG viewer; a `<desc>` tag, which holds a description of the document; and other tags for grouping and drawing elements.

- Instructions to draw basic shapes are provided via the `<line>`, `<rect>`, `<circle>`, `<ellipse>`, and `<polygon>` tags. In our sample document,

  ```
  <rect x = "10" y = "20" width = "50" height = "100" class = "recs" />
  ```

  is an instruction to draw a rectangle with upper left corner at (10, 20), a width of 50, and height of 100. (The `class` attribute contains style information about the color of the interior of the rectangle and its border.) These shape elements are specific families of shapes that can also be rendered with the more general `<path>` element. Note the size of the rectangle is relative to the size of the `viewBox` in the canvas, which in our example is 300 by 300.

- The `<path>` tag provides the information needed to draw a "curve". It contains instructions for the placement of a pen on a canvas and the movement of the pen from one point to the next in a connect-the-dot manner. The instructions for drawing the path are specified via a character string that is provided in the `d` attribute of `<path>`. For example, the path for the Oregon border in Figure 9 is as follows

  ```
  d = "M 102.8 174.6 L 102.8 174.6
       L 106.1 178.0 L 100.9 189.3 L 102.8 191.2 L 102.1 195.1
       L 102.1 209.9 L 53.9 209.9 L 51.5 210.0 L 50.0 206.0
  ```

```
L 49.3 202.9 L 52.0 191.5 L 52.7 185.0 L 52.9 184.1
L 53.4 179.0 L 53.3 173.0 L 54.5 173.1 L 55.1 173.1
L 56.0 172.8 L 60.5 174.0 L 61.4 177.2 L 63.5 178.4
L 67.7 177.5 L 72.5 177.7 L 88.5 174.6 L 102.8 174.6
Z"
```

The drawing of the Oregon border begins by picking up the pen and moving it to the starting position (102.8, 174.6). This position is given either as an absolute position, i.e., "M x,y", or a relative position, i.e., "m x,y". Note that the capitalization of the letter determines whether the position is relative (m) or absolute (M). Either a comma or blank space can be used to separate the x and y coordinates. From the starting point, the pen draws line segments from one point to the next. The segment may be a straight line (L or l) or a quadratic (Q or q) or cubic (C or c) Bezier curve. The Z command closes a path by drawing a line segment from the pen's current position back to the starting point. These paths provide very succinct notation for drawing curves.

The libcairo rendering engine (and hence the `svg()` device in R) uses `<path>` for rendering all shapes including characters and text. One benefit to this approach of using a `<path>` command to draw each letter is that there is no reliance on fonts when the SVG document is viewed. It also means that scaling the SVG preserves the shape of the letters with very high accuracy. However, when you add text and shapes to an SVG document, you may want to use the simpler higher-level short-cut tags, e.g., `<text>` and `<ellipse>`.

- Elements can be grouped using the `<g>` tag. This is helpful when for example, you want to treat the collection of objects as a single object in order to transform it as a unit, or place the same appearance characteristics on a collection of elements. The style placed on `<g>` will apply to all of its sub-elements. Grouped elements can also be defined and then inserted multiple times in the document (see the description of the `<defs>` element below). It is also possible to nest other `<svg>` elements within a `<g>` element. This allows us to create compositions reusing previously and separately created displays.

- The `<defs>` node is a container for SVG elements that are defined and given a label, but not immediately put in the SVG display. These definitions can be augmented, displayed and reused within the SVG display through a reference to the element's unique identifier. The `<defs>` element acts as a dictionary of template elements.

  For example, in the sample SVG code we defined a pair of circles, one green and the other pink. These are grouped together into a single unit and identified by the `id` of "circles" as shown here:

```
<g id="circles">
 <circle id = "greencirc" cx = "15" cy="15" r = "15"
   fill = "lightgreen"/>
 <circle id = "pinkcirc" cx= "50" cy="50" r = "15" fill = "pink"/>
</g>
```

  This pair of circles is defined in the `<defs>` element, and rendered via the `<use>` tag. The pair is rendered twice, at two different locations on the canvas as follows,

```
<use  x = "100" y = "100" xlink:href = "#circles"/>
<use  x = "200" y = "50" xlink:href = "#circles" />
```

As with HTML, we specify the reference to an internal element (or "anchor") by prefixing the name/id of the desired element with a '#', i.e., "#circles". This suggests that we can link to elements in other files, and indeed we can. In fact, we have the full power of another XML technology, named XLink (Simpson 2002), available in SVG.

- Many times, we want to create style definitions to be used by multiple SVG elements. The style of an element can be specified in four ways:

  1. *In-line styles.* One approach is to place the style information directly in the element (e.g., `<circle>`, `<path>`) by setting the value of a `style` attribute. For example, the style of the Oregon polygon,

     ```
     <path style = "fill: rgb(100, 149, 237);
              fill-opacity: 0.5;stroke-width: 0.75;
              stroke-linecap: round; stroke-linejoin: round;
              stroke: rgb(0%,0%,0%); stroke-opacity: 1;"
     ...
     />
     ```

     provides the color (cornflower blue) and opacity for filling the polygon, the color of the border (black), and details about the border, such as the thickness of the line. With this approach, the `style` attribute value holds a string of Cascading Style Sheet (CSS) properties. Note that, colors in SVG can be represented by text name, e.g., "cornflowerblue", the red-green-blue triple rgb(100, 149, 237), or the hexadecimal representation of the triple, e.g., "#6495ED".

  2. *Internal stylesheet.* The style information can be placed in a stylesheet that is stored within the file in the `<defs>` node of the document. As an example, the rectangle in our Figure 9 uses the "recs" class within the internal stylesheet. This connection is specified via the `class` attribute on the `<rect>` element:

     ```
     <rect x = "10" y = "20" width = "50" height = "100" class = "recs"/>
     ```

     The CSS style sheet and its classes are found within the `<defs>` portion of the file, in the `<style>` node:

     ```
     <style type="text/css">
      < ![CDATA[
        .recs {fill: rgb(50%,50%,50%); fill-opacity: 1; stroke: red;}
       ] ]>
     </style>
     ```

     For more information about Cascading StyleSheets see Meyer (2004).

  3. *External stylesheet.* The stylesheets may also be located in an external file. It can be included via the xml-stylesheet processing instruction such as

     ```
     <?xml-stylesheet type="text/css" href="RSVGPlot.css" ?>
     ```

  4. *Presentation attributes.* An alternative to using a stylesheet, whether in-line, internal, or external, is to provide individual presentation attributes directly in the SVG element. As an example, the pink circle's color in Figure 9 is specified through a `fill` attribute in the `<circle>` tag as follows,

```
<circle id = "pinkcirc" cx= "50" cy="50"  r = "15" fill = "pink"/>
```

The presentation attributes are very straightforward and easy to use. We can just add a simple attribute, e.g., `fill`, on an element and avoid the extra layer of indirectness. This approach allows us to easily modify the presentation of an element in response to a user action. However, the downside of this approach is that presentation is mixed with content. For this reason, the in-line, internal and external cascading style sheets are preferable to presentation attributes. These various approaches can be mixed; that is, style information can be provided from a combination of in-line, internal, and external stylesheets, as well as presentation attributes.

## 7. The SVG display for an R plot

We next examine a typical document produced from the SVG graphics device in R. The SVG produced in R is highly structured, and we use this structure to locate particular elements and enhance them with additional attributes, parent them with new elements, and insert sibling elements in order to create various forms of interactivity and animation. We examine the SVG generated for the following call to `plot()` that was used to make the scatter plot in Example 1.

```
R> depth.col <- gray.colors(100)[cut(quakes$depth, 100, label = FALSE)]
R> depth.ord <- rev(order(quakes$depth))
R> doc <- svgPlot(
+    plot(lat ~ long, data = quakes[depth.ord, ], pch = 19,
+      col = depth.col[depth.ord], xlab = "Longitude", ylab = "Latitude",
+      main = "Fiji Region Earthquakes")
+  )
```

We can explore the contents of the resulting XML/SVG document programmatically by using the tools available in the **XML** package. The tree in Figure 10 provides a conceptual image of the hierarchy of the SVG nodes for the plot and its annotations.

The **XML** package provides several tools that aid us in examining the structure and content of an SVG document. We demonstrate some of these as we explore the resulting SVG document in `doc`. We begin by retrieving the top node and assigning it to `root`,

```
R> root <- xmlRoot(doc)
```

We can use other functions such as `xmlName()`, `xmlSize()`, and `xmlValue()` to query the name, number of children, and the text content of an element, respectively. With them, we determine that the root has three children, the first contains the R code that is in the call to `svgPlot()`, and the following two are the `<defs>` and the main `<g>` tag that contains the plotting elements.

```
R> xmlSize(root)

[1] 3
```

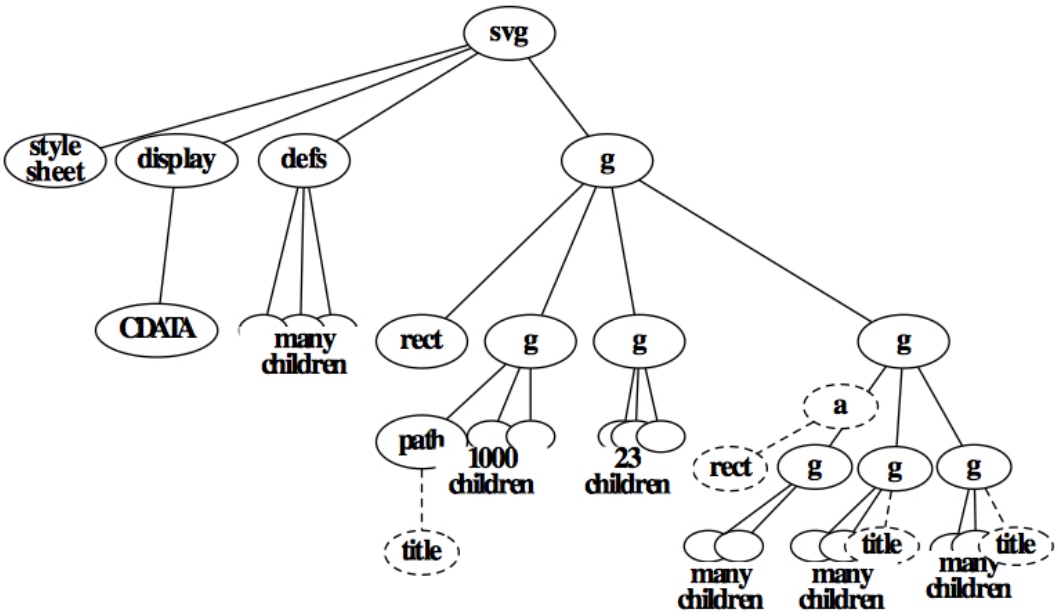


Figure 10: This tree provides a visual representation of the organization and structure of the SVG document produced by the call to `plot()` in Example 1. SVG elements are shown as nodes in the tree. The style-sheet and the `<display>` element on the left of the tree are added by `svgPlot()`. The `<CDATA>` child of `<display>` contains the R code passed in the call to `svgPlot()`. The nodes with dotted lines are those that have been added to the SVG document by the `addLink()` and `addToolTips()` functions. For readability, not all nodes are displayed, and in some cases the number of nodes is provided to make it clear to which part of the plot these elements correspond. For example, the "1000 children" refers to the elements that plot the points in the scatter plot in Figure 7; they correspond to the 1000 observations in the `quakes` data frame.

```
R> xmlApply(root, xmlName)

$display
[1] "display"

$defs
[1] "defs"

$g
[1] "g"
```

or more simply

```
R> names(root)

  display      defs         g
"display"    "defs"       "g"
```

```
R> xmlValue(root[[1]])

[1] "plot(lat ~ long, data = quakes[depth.ord, ], pch = 19,
 col = depth.col[depth.ord], \n    xlab = \"Longitude\",
 ylab = \"Latitude\", main = \"Fiji Region Earthquakes\")"
```

To examine the `<rect>` element, we can also use either of the following approaches,

```
R> root[[3]][[1]]

<rect x="0" y="0" width="504" height="504"
      style="fill:rgb(100%,100%,100%); fill-opacity: 1;
      stroke: none;"/>

R> root[["g"]][["rect"]]

<rect x="0" y="0" width="504" height="504"
      style="fill:rgb(100%,100%,100%); fill-opacity: 1;
      stroke: none;"/>
```

Also, we can use the `xmlChildren()` function to extract each child node into a list of regular XML nodes, which can make it easier to explore and manipulate them.

```
R> kids <- xmlChildren(root[[3]])
R> length(kids)

[1] 4

R> kids[[1]]

<rect x="0" y="0" width="504" height="504"
      style="fill:rgb(100%,100%,100%); fill-opacity: 1;
      stroke: none;"/>
```

Alternatively, the function `getNodeSet()` extracts elements from the XML tree, `doc`, and is a very general mechanism for querying the entire tree or sub-trees. It requires an XPath expression that specifies how to locate nodes. XPath (Simpson 2002) is an extraction tool for locating content in an XML document. It uses the hierarchy of a well-formed XML document to specify the desired elements to extract. XPath is not an XML vocabulary; it has a syntax that is similar to the way files are located in a hierarchy of directories in a computer file system, but it is much more flexible and general. Rather than locating a single node in a tree, XPath extracts node-sets, which are collections of nodes that meet the criteria in the XPath expression. The node-set may be empty when no nodes satisfy the XPath expression. Likewise, when multiple nodes match the expression, a collection of nodes make up the node-set. For example,

```
"/x:svg/x:g/*"
```

locates all grandchildren of the root <svg> that have a <g> parent. Note that we must specify the name space for the <svg> tag. Since the SVG elements use the default name space in this document, getNodeSet() allows us to use any name space abbreviation without defining it or matching it to the name space prefix in the document. In this case we simply chose "x". We use getNodeSet() to extract these elements, and then request their names via a call to sapply(),

```
R> kids <- getNodeSet(doc, "/x:svg/x:g/*", "x")
R> sapply(kids, xmlName)
```

```
[1] "rect" "g"    "g"    "g"
```

For more details on how to use XPath to retrieve nodes from an XML document see Simpson (2002). The return value from getNodeSet() is a list, and we access the first element by the standard indexing methods for a list:

```
R> kids[[1]]
```

```
<rect x="0" y="0" width="504" height="504"
  style="fill: rgb(100%,100%,100%); fill-opacity: 1;
         stroke: none;"/>
```

Of the following three approaches,

```
R> root[[3]][[1]]
R> root[["g"]][["rect"]]
R> getNodeSet(doc, "/x:svg/x:g/x:rect", "x")
```

the first two return an object of class XMLInternalNode, whereas the call to getNodeSet() returns a list where each element is an 'XMLInternalNode'. R's plotting functions are very regular and predictable so we can determine which nodes corresponds to which graphics objects quite easily. The approach we use here capitalizes on understanding the default operation of the plotting functions. To see how, notice that the first <g> sibling of <rect> has 1000 children.

```
R> sapply(kids, xmlSize)
```

```
[1]    0 1000   25    3
```

This number exactly matches the number of points plotted.

```
R> dim(quakes)
```

```
[1] 1000  5
```

The element with 25 children corresponds to the axes of the data region of the plot and their tick marks. The element with just 3 children contains the title and the two axes labels.

The high-level functions described in Section 4 make use of these default locations in the SVG output for the most common plots. If you need to annotate less common plots, then you may need to use the other functions in **SVGAnnotation**, or directly handle the XML nodes yourself with the functionality available in the **XML** package.

Next, let's examine the first of these 1000 nodes. We see that it is a `<path>` element.

```
R> kids[[2]][[1]]
```

```
<path style="fill-rule: nonzero;
   fill: rgb(90.196078%,90.196078%,90.196078%);
   fill-opacity: 1;stroke-width: 0.75; stroke-linecap: round;
   stroke-linejoin: round;
   stroke: rgb(90.196078%,90.196078%,90.196078%);
   stroke-opacity: 1;stroke-miterlimit: 10; "
   d="M 337.144531 191.292969 C 337.144531 194.894531
        331.746094 194.894531 ...  " />
```

Although the symbol used to represent a point in the plot is a circle, the SVG graphics device in R (via libcairo) does not use the `<circle>` element to draw it. Instead, the `<path>` node provides instructions for drawing the circle using Bezier curves to connect the points supplied in the `d` attribute. (The letter C between the (x,y) pairs means the points are to be connected by a cubic Bezier).

The x, y coordinates used to specify the path are in the coordinate system of the SVG canvas, not the coordinate system of the data. This system places the smallest values at the upper left corner of the canvas and the maximum values for x and y at the lower right corner, i.e., y increases as you move down the canvas and x increases as you move right. As a result, we cannot directly use the values in our data to directly identify SVG elements in the document; the data values first need to be converted into this alternative coordinate system. The SVG coordinate system supports various units of measurement, but the device in R uses only points (abbreviated as 'pt' or 'pts'). A point is approximately 1/72 of an inch. The size of the canvas is provided via the `width` and `height` attributes on the `<svg>` root node of the document.

The libcairo engine used by R generates all shapes exclusively with the `<path>` tag. This holds true as well for the text in axes and plot labels; that is, the cairo rendering engine in R creates the text by explicitly drawing the letters via SVG paths. The resulting letters scale extremely well and do not rely on special fonts which may not be available at the time of rendering. More specifically, the path for the glyphs that correspond to the text are created and placed in a `<defs>` element, and a `<use>` element brings in the glyph at the proper location in the plot. This representation introduces some difficulties for us because the text for legends and axes labels do not appear as plain text in the SVG document and so are not easily located for post-processing. The placement of the glyphs in the `<defs>` means that there is one additional level of indirection that needs to be handled when annotating text.

To make this concrete, let's consider our scatter plot example again. The last child of the main graphing node contains the information for drawing the title and axes labels. It has three children, one each for the title, y axis, and x axis, respectively. We identify the x axis with the following XPath expression,

```
R> getNodeSet(doc, "/x:svg/x:g/x:g[3]/*[last()]", "x")

[[1]]
<g style="fill: rgb(0%,0%,0%); fill-opacity: 1;" type="axis-label">
  <use xlink:href="#glyph1-7" x="14.398438" y="266.152344"/>
  <use xlink:href="#glyph1-8" x="14.398438" y="259.478516"/>
  <use xlink:href="#glyph1-9" x="14.398438" y="252.804688"/>
  <use xlink:href="#glyph1-10" x="14.398438" y="249.470703"/>
  <use xlink:href="#glyph1-9" x="14.398438" y="246.804688"/>
  <use xlink:href="#glyph1-11" x="14.398438" y="243.470703"/>
  <use xlink:href="#glyph1-12" x="14.398438" y="236.796875"/>
  <use xlink:href="#glyph1-13" x="14.398438" y="230.123047"/>
</g>

attr(,"class")
[1] "XMLNodeSet"
```

This XPath expression starts at the root node, proceeds down one step to the root's `<g>` child, then down another level in the tree to select the third `<g>` element, and finally, one more level to the last child of the third `<g>`. Notice that we use the XPath predicate `[3]` to select the third `<g>` and the XPath function `last()` to get the last child element. This particular `<g>` element contains the instructions for drawing the axes label "Latitude". It references eight glyphs that are located in the `<defs>` node of the document. The references are via the `href` attribute. Note that there is one glyph per letter so, for example, the letter 'a' appears in the document as "glyph1-8".

```
R> getNodeSet(doc, "/x:svg/x:defs/x:g/x:symbol[@id ='glyph1-8']", "x")

[[1]]
<symbol overflow="visible" id="glyph1-8">
  <path style="stroke: none;" d="M -1.671875 -1.578125
    C -1.367188 -1.578125 -1.128906 -1.6875 -0.953125 -1.90625
    C -0.773438 -2.132812 -0.6875 -2.398438 -0.6875 -2.703125
    ...
    Z M -6.421875 -3.265625 "/>
</symbol>

attr(,"class")
[1] "XMLNodeSet"
```

The high-level functions in **SVGAnnotation** add elements and/or attributes to the SVG produced by the graphics device. For example, `addToolTips()` adds a `type` attribute with value of "plot-point" so the `<path>` element can be more easily identified as instructions to draw a point in a plot. This addition makes it easier for the programmer to extract and annotate elements.

```
R> addToolTips(doc, apply(quakes[depth.ord, ], 1, function(x)
+    paste(names(quakes), x, sep = " = ", collapse = ", ")), addArea = TRUE)
```

```
R> kids[[2]][[1]]

<path style="fill-rule: nonzero;
             fill: rgb(90.196078%,90.196078%,90.196078%);
             fill-opacity: 1;stroke-width: 0.75;
             stroke-linecap: round;
             stroke-linejoin: round; stroke:
             rgb(90.196078%,90.196078%,90.196078%);
             stroke-opacity: 1;stroke-miterlimit: 10; "
      d="M 337.144531 191.292969 C 337.144531 194.894531
        331.746094 194.894531 ... "
      type="plot-point"
      xlink:title="lat = -20.32, long = 180.88, depth = 680,
                   mag = 4.2, stations = 22">
  <title>lat = -20.32, long = 180.88, depth = 680, mag = 4.2,
            stations = 22
  </title>
</path>
```

In the call to `addToolTips()`, the child element, `<title>`, was added to `<path>`; it contains the information that will be displayed when the mouse hovers the point. Similarly, adding tool tips to the axes and a hyperlink to the plot title further modifies the SVG. The addition of the hyperlink results in the insertion of an `<a>` tag as the parent of the `<g>` element that contains the title information and a `<rect>` element as a sibling to this `<g>`. All elements added after the creation of the SVG are denoted by dashed lines in Figure 10.

## 8. Tools for customizing interactivity

We have seen that **SVGAnnotation** provides facilities for identifying elements of the SVG document that correspond to particular parts of plots, e.g., in the Example 4 the function `getPlotPoints()` locates the SVG elements that correspond to the hexagonal bins in the R display. These functions can assist developers in creating new high-level functions for annotating the output from "non-standard" plotting functions in R. In this section, we demonstrate the lower-level details of how a developer might post-process SVG to add annotations and JavaScript in order to enable interactivity. Table 4 lists additional functions available for working with various parts of the SVG document.

To begin, we explain how `linkPlots()`, which was demonstrated in Example 5, modifies the SVG to perform the linking action. The key idea is that the SVG is annotated to add a unique identifier to each SVG element that represents a point in a plot. This identifier includes information as to which plot region and observation the point belongs. The `linkPlots()` function retrieves the plot regions by using `getPlotRegionNodes()`. Whether we use `pairs()` or `par(mfrow = ...)`, the points within each plotting region appear in a regular order; that is, the first element in each plotting region corresponds to the first row in the data frame, the second element to the second row, and so on. Thus elements across plots can be matched by simply using the order in which they appear in their respective plotting regions, assuming that they come from the same data frame or have a correspondence by row. The identifiers added

to the points are of the form "plotI-J" where "I" is the plot index and "J" is the observation index within the plot. This unique identifier is added as an `id` attribute on each of the SVG elements, which makes it easy for the JavaScript function to find the elements to be linked.

In addition, the `linkPlots()` function in **SVGAnnotation** adds `onmouseover` and `onmouseout` attributes to each point element in the SVG document. The value of each of these attributes is a JavaScript function call to perform the linking and unlinking action, e.g., to change the color of the linked points. Below is the augmented SVG node that corresponds to the 10th point in the first plot. This is the point colored red in the left-hand plot in the screen shot in Figure 3.

```
<path style="stroke-width:0.75; ... "
      d="M 260.417969 72.800781 C 260.417969 77.839844 252.855 ..."
      id="plot1-10"
      onmouseover="color_point(evt, 10 , 2 , 'red' )"
      onmouseout="reset_color(evt, 10 , 2 )"
      fill="none" originalFill="none"/>
```

The `linkPlots()` function adds to the SVG document the JavaScript code for the two functions `color_point()` and `reset_color()`. The code appears in a `<script>` node within the SVG document. Note that these JavaScript functions are relatively general and available in the package for making customized plots. (For more information about JavaScript, see Appendix D.)

The principles we established for annotating the SVG to link points across plots carry over to more general settings. The basic ideas used in `linkPlots()` are to: post-process the document so that elements in the SVG are easily identified and changed at the time of viewing; create JavaScript functions, which are not dependent on a particular set of data to respond to mouse events and make the requested changes to the display. This approach can be summarized by the following set of tasks performed in the annotation stage:

- Within R, add unique identifiers to the relevant graphics elements in the SVG document so they can be retrieved easily in the viewing stage (via the JavaScript function `getElementById()`).
- Create special attributes to store the default values for settings that are subject to change, e.g., `original-style`. These attribute names are made up by the developer, and should not conflict with SVG attribute names.
- For the "action", set up event attributes, such as `onmouseover` and `onmouseout`, on the appropriate elements. The attribute values are JavaScript function calls. Our philosophy is to pass all element-specific information needed to respond to a request/mouse-event in the call. This way, the JavaScript functions can be used in other situations, e.g., with other data and other types of plots, and not rely on auxiliary global variables.
- Embed in the document the JavaScript functions that modify and reset the element attributes.

We demonstrate how to use the basic approach just outlined to create a customized event handler for **lattice** plots. The interactivity in this example extends the notion of linked plots

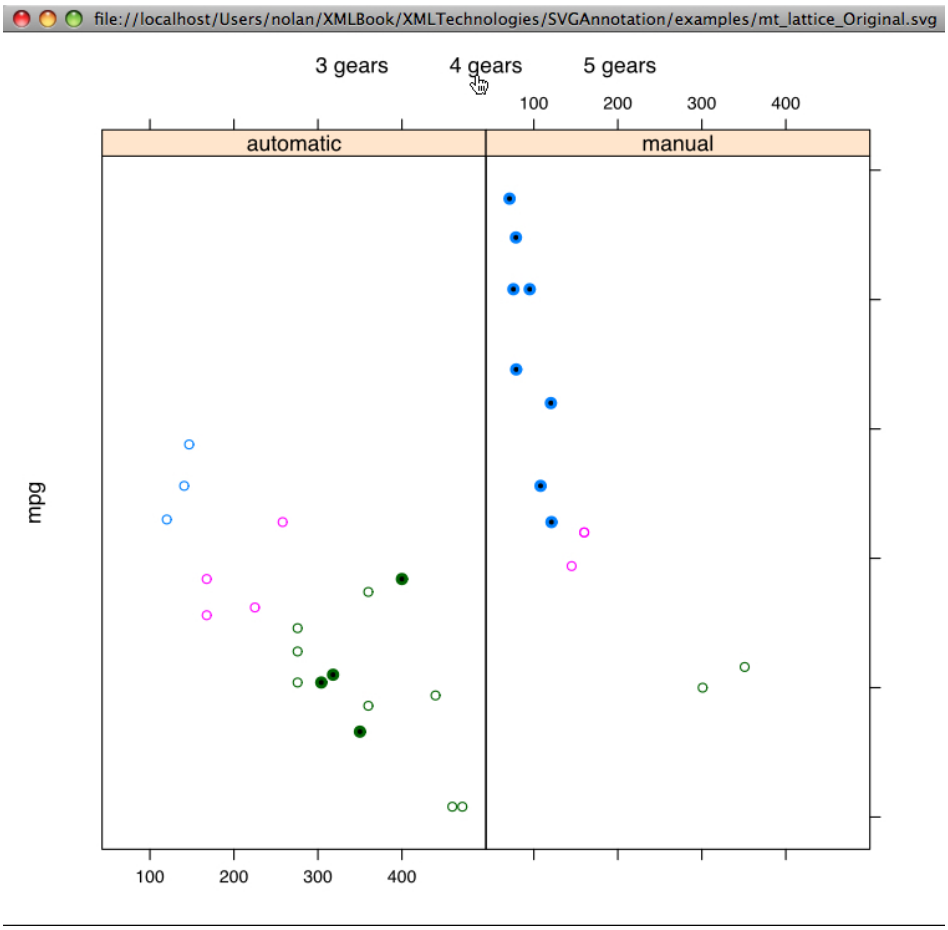


Figure 11: The scatter plots show the relationship between horsepower and miles per gallon for automatic and manual transmission cars. The points within each panel are colored according to the number of cylinders the engine has (4 cyan, 6 pink, and 8 green). The legend above the plots lists the possible number of gears, 3, 4, and 5. When for example, the mouse hovers over the 4-gear group, the points belonging to that group are highlighted. This interactivity is accomplished by changing the `style` attribute of the points via JavaScript. When the mouse moves off the legend, the original styles for these points are restored.

to **lattice** plots where points are linked via an interactive legend.

**Example 6.** Linking across **lattice** plots.

In this example, we follow the steps listed above to customize a JavaScript event handler. In particular, we show how you might extend the linking type of interactivity in Example 5 to **lattice** plots. The data used are the familar `mtcars`, and the **lattice** plot is constructed with the formula `mpg ~ disp | am` with `cyl` as the `groups` argument. We add a legend to a **lattice** plot for a fifth variable (`gear`). A mouse-over event on the legend results in the highlighting of the observations in each panel that belong to the corresponding level of `gear`. In this way, information from an additional variable is added to the **lattice** plot.

We begin by making the plot. We convert `cyl` and `am` into factors with more meaningful labels for the plot, and we use `simpleKey()` to add a legend to the **lattice** plot:

```
R> library("lattice")
R> mtcars$cyl <- factor(mtcars$cyl,
+    labels = paste(c("four", "six", "eight"),  "cylinder"))
R> mtcars$am <- factor(mtcars$am, labels = c("automatic", "manual"))
R> gearGroups <- sort(unique(mtcars$gear))
R> gearLabels <- paste(gearGroups, "gears", sep = " ")
R> colors <- trellis.par.get("superpose.symbol")$col[
+    seq(along = levels(mtcars$cyl))]
R> topArgs <- list(fun = draw.key, args = list(key = list(
+    text = list(gearLabels), columns = 3)))
R> botArgs <- list(fun = draw.key, args = list(
+    key = list(text = list(levels(mtcars$cyl)),
+    points = list(pch = 21, col = colors), columns = 3)))
R> doc <- svgPlot(xyplot(mpg ~ disp| am, groups = cyl, data = mtcars,
+    legend = list(top = topArgs, bottom = botArgs)))
```

There are 2 plotting regions in the SVG document, one for each panel. Each plotting-region element contains its points as child elements and we locate the entire collection of the corresponding SVG elements with

```
R> panels <- getPlotRegionNodes(doc)
R> points <- unlist(lapply(panels, xmlChildren), recursive = FALSE)
```

Now that we have the point elements, we can proceed with the first step in creating our own event handler, and augment these elements with unique identifiers. We construct identifiers based on the point's index within its gear-group and panel, e.g., `id="2-4-1"` is the identifier for the first observation in the four-gear group in the second panel (the manual transmission panel). We use `addAttributes()`, a function in the **XML** package, to add the `id` attribute to the point elements as follows:

```
R> ids = by(mtcars, list(mtcars$gear, mtcars$am), function(x)
+    paste(as.numeric(x$am), x$gear, 1:nrow(x), sep = "-"))
R> uids = unlist(ids)
R> mapply(function(node, id) addAttributes(node, id = id), points, uids)
```

We are ready for the third step in the process: augmentation of the legend labels with JavaScript calls for mouse events. We skipped the second step, that of preserving the default style values, because we will take care of it at viewing time with our JavaScript function `highlight()`. That is, `highlight()` will highlight or un-highlight the relevant points, and it will save and restore the default styles of the points. `highlight()` is called with 3 arguments: the first argument indicates the gear-group; the second is an array giving the number of elements within that group in each panel; and, the third indicates whether the event is a mouse over (true) or a mouse out (false). Thus for the third step, we need to add `onmouseover` and `onmouseout` attributes that have JavaScript function calls such as `highlight(4, [4, 8], true)`. Notice that JavaScript uses square brackets to delimit arrays (e.g., `[1, 2]`). We construct these simple arrays with direct calls to `paste()`. For more complex R objects, we would be advised to use the **RJSONIO** package to serialize R objects to JavaScript object notation (JSON).

To generate the values for these arrays, we need to know how many points are in each group within each panel. We get a frequency table with this information via

```
R> counts = table(mtcars$am, mtcars$gear)
R> counts

            3  4  5
  automatic 15  4  0
  manual     0  8  5
```

We use these counts to construct the JavaScript calls as follows:

```
R> nodes <- getLatticeLegendNodes(doc, panels, 1)
R> sapply(seq(along = 1:length(gearGroups)), function(i) {
+    cts <- paste("[", paste(counts[,i], collapse = ", "), "]", sep = "")
+    addAttributes(nodes[[i]],
+      onmouseover = paste("highlight(", gearGroups[i], ",", cts, ", true)"),
+      onmouseout = paste("highlight(", gearGroups[i], ",", cts, ", false)"))
+  })
```

Note that `getLatticeLegendNodes()` locates the elements corresponding to the legend in the SVG.

For the fourth and last step, we add to the SVG document the JavaScript function definition for `highlight()` and its helper function `highlightPoint()`. The `addECMAScripts()` function takes care of this for us:

```
R> jscript = list.files(system.file("examples", "Javascript",
+    package = "SVGAnnotation"), full.names = TRUE, pattern = "multiLegend")
R> addECMAScripts(doc, jscript)
R> saveXML(doc, "mt_lattice.svg")
```

For completeness, we examine the JavaScript functions `highlight()` and `highlightPoint()` to see how they handle the interactivity. Recall that `highlight()` is called with the index of the group to be highlighted/un-highlighted and the array holding the counts of the number of points in that group within each panel. The function, shown below, iterates over each panel and each point within the desired group in that panel and constructs the corresponding `id` value for the affected points. It then retrieves the SVG element by identifier, using the Javascript method `getElementById()`. This method is very convenient because it retrieves a node in the SVG document by specifying the value of its `id` attribute. Our `highlight()` function is defined as

```
function highlight(group, pointCounts, status)
{
    var el;
    var i, numPanels = pointCounts.length;
       /*  we want the group */
     for(panel = 1; panel <= numPanels; panel++) {
```

```
      for(i = 1; i <= pointCounts[panel-1]; i++) {
        var id = panel + "-"+ group + "-" + i;
        el = document.getElementById(id);
        if(el == null) {
          alert("can't find element " + id)
          return(1);
        }
        highlightPoint(el, status);
      }
    }
}
```

Once an element is retrieved, `highlightPoint()` is called to change its appearance.

The `highlightPoint()` function modifies the value of the `style` attribute for an element. We could have handled the style change by adding a specific presentation attribute to the element, e.g., adding a `fill` attribute to change the fill color. Here we use regular expressions (in Perl format) to process the value of the `style` attribute. We break the character string into individual components, change the fill component, and create the new in-line style string. At the same time, we make sure to save the original style information in order to restore it as needed. It is saved in an attribute we made especially for this purpose, dubbed `original-style`. The `highlightPoint()` function is defined as follows.

```
function highlightPoint(el, status)
{
  var old = el.getAttribute('original-style');
  if(status && old == null)
    el.setAttribute('original-style', el.getAttribute('style'));
  if(status) {
    /* Have to set the attribute within the style attribute,
       i.e. a sub-attribute which makes things more complex. */
    var cur = el.getAttribute('style');
    var tmp = cur.replace(/fill: [^;]+/, "fill: black");
    var tmp = tmp.replace(/stroke-width: [^;]+/,
                          "stroke-width: 2");
    el.setAttribute('style', tmp);
  }
  else
    el.setAttribute('style', old);
}
```

We conclude the example, with a brief discussion as to how we determined the ordering of the points in the SVG for `xyplot()`. We programmatically changed the color of a point in the SVG, and viewed the modified document to determine that the points for each group appear sequentially within the document. Further exploration of the document confirmed that all of the elements of the plot's legend are sibling nodes of the plotting region. A simple XPath expression was used to locate them:

```
R> other <- getNodeSet(panels[[length(panels)]], "./following-sibling::*")
R> nodesAlt <- other[-(1:(length(other) - length(gearGroups)))]
```

This basic approach was then incorporated into `getLatticeLegendNodes()`, which does this more robustly by matching against the `type` of node. (`svgPlot()` uses the plotting calls to annotate the SVG elements with this additional information.)

## 9. Complex/non-standard examples

In this section, we use a different approach to creating interactivity with JavaScript. In Section 4.2, we used JavaScript to simply alter the attributes of existing elements. That is, all of the computations on the data were done in advance, in R, in either the plotting stage or the annotation stage, and the JavaScript functions simply changed and reset attributes. Here, in the annotation stage, we place R objects in the SVG document as JavaScript variables, and in the viewing stage, we use JavaScript and these variables to create new shapes in the display. We provide two examples. In the first, we draw line segments on a scatter plot to connect a point to its nearest neighbors. The information as to which points are nearest others is calculated in R and placed in the SVG document as JavaScript variables. These examples demonstrate that drawing can be done in two places: in R via the plotting routines and in JavaScript at the time the document is viewed (and R is no longer available). If we had made this plot using the earlier approach in Section 8, we would have drawn all possible line segments in R, hidden them within the plot, and the JavaScript code would respond to mouse-over and mouse-out events by modifying element attributes in order to show and hide the line segments. In the approach here, we draw new lines in response to a mouse-over event, and then we discard them when the mouse moves off the element.

In Section 6, we determined the structure of an SVG document generated via libcairo. To do this, we used information about the data being plotted. For example, knowing the number of observations helped us find the SVG nodes corresponding to the points. We matched the number of rows in the data frame against the number of elements in various parts of the SVG document. The examples in this section continue this approach of determining which elements in the SVG document correspond to particular components of a plot. In one case, in order to uncover the structure in the SVG image, we compare the graphics object returned from the call to the R plotting function with the graph object returned from `randomGraph()`. This is a deterministic process that need only be done once because the SVG output for a particular type of plot has a very regular structure. Although regular, the structure is not a formal one that allows us to immediately identify the SVG elements corresponding to R components. The **SVGAnnotation** package does most of this automatically. We describe it here to illustrate the underlying mechanism and approach so it is clearly understood and can be adapted for new types of plots.

**Example 7.** Interactive nearest neighbors.

In this example, we use JavaScript to dynamically (i.e., at viewing time) augment a scatter plot so that it displays the four nearest-neighbors to a point. That is, when the mouse moves over a point, new line segments are drawn from that point to its four nearest neighbors (see Figure 12). Also, when the mouse moves off the point, the lines are removed from the display. To do this, we use JavaScript to lookup the elements in the SVG display that correspond to

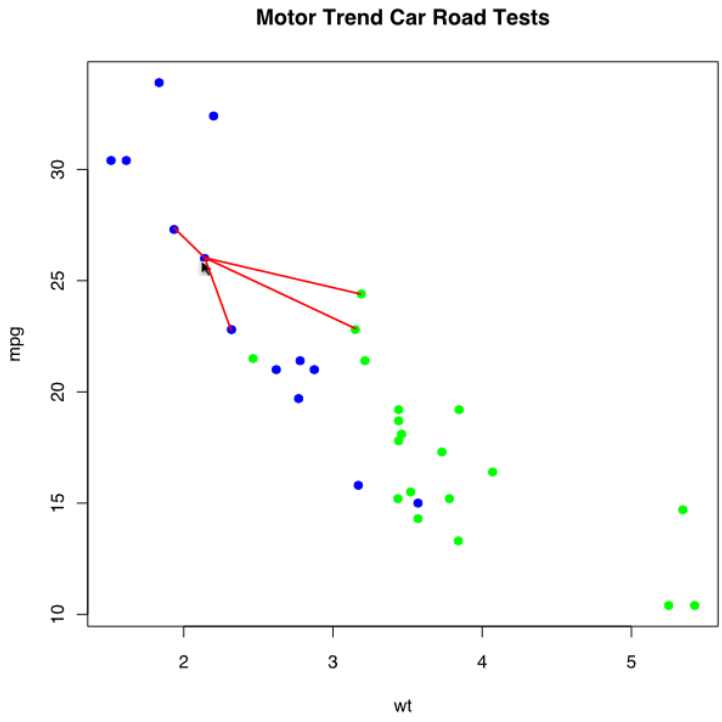


Figure 12: This interactive scatter plot can display the four-closest neighbors of any point. When the mouse moves over a point, a JavaScript function is called to determine the coordinates of the 4 nearest neighbors and draw line segments from the active point to its neighbors. The SVG document contains JavaScript variables that hold the indices of the neighbors for each point, in order, from closest to farthest. The nearest neighbors are defined by Euclidean distance in the `mpg` and `wt` dimensions. Notice that the scales on the two axes are not the same and so the aspect ratio for the plot is not 1. This can yield a somewhat misleading display of the nearest neighbors.

the nearest points, and we add new `<line>` elements from the active point to its neighbors. Note that we are not adding XML content to the original SVG document, but rather adding line objects to the JavaScript rendering of the SVG display.

We begin by creating the basic scatter plot:

```
R> doc <- svgPlot(plot(mpg ~ wt, mtcars, main = "Motor Trend Car Road Tests",
+    pch = 19, col = c("green", "blue")[(am + 1)]))
```

In the annotation stage, we add unique identifiers to the points in the plot. Specifically, we add an `id` attribute to each point that simply has the index of the observation in the data frame. Since JavaScript uses 0-based indexing, we find it easier to start the indexing at 0, e.g.,

```
R> ptz = getPlotPoints(doc, simplify = FALSE)[[1]]
R> sapply(seq(along = ptz), function(i) addAttributes(ptz[[i]], id = i - 1))
```

We also add calls to the JavaScript functions (`showNeighbors()` and `hideLines()`) to handle the mouse-over and mouse-out events on each point:

```
R> sapply(seq(along = ptz), function(i) {
+    addAttributes(ptz[[i]],
+      onmouseover = "showNeighbors(evt, k, neighbors)",
+      onmouseout = "hideLines(evt)")
+  })
```

Also in the annotation stage, we calculate the distances between points, and use these distances to identify the nearest neighbors as follows:

```
R> DD <- as.matrix(dist(mtcars[, c("mpg", "wt")]))
R> D <- t(apply(DD, 1, order)) - 1
```

Notice that we use the order of the observation in the data frame so that it will match the identifier that we added to the point in the plot.

We make this information about a point's neighbors available to the JavaScript code as a two-dimensional JavaScript array, serialized from R. The following call to `addECMAScripts()` illustrates how we serialize the R matrix `D` to JavaScript as the variable `neighbors`, in addition to adding the JavaScript code from several files:

```
R> dimnames(D) <- list(NULL, NULL)
R> jscript <- list.files(path = system.file("examples", "Javascript",
+    package = "SVGAnnotation"), full.names = TRUE, pattern = "knn")
R> addECMAScripts(doc, jscript, TRUE, neighbors = D)
```

The `addECMAScripts()` function has a `...` argument that accepts R objects in the form `name = value`. These objects are added to the SVG document as JavaScript variables with the argument name used as the name for the JavaScript variable. The `TRUE` value for the third argument indicates that we want the contents of the JavaScript file to be copied into the SVG rather than supplied via a link. We choose to copy the contents to make the resulting SVG file independent of auxiliary files.

The JavaScript function `showNeighbors()` has three arguments: the event object, the number of neighbors of interest (`k`), and a two-dimensional array identifying the nearest neighbors for all points (`neighbors`). The array is a JavaScript variable that contains the contents of the R matrix `D`. Notice that the data are separated from the function and explicitly passed as arguments in the function call so that the function can be used with other data in other contexts. With its helper functions, `showNeighbors()` retrieves the index of each of the `k` neighboring points from the `neighbors` array; extracts the corresponding JavaScript SVG object and gets its coordinates in the SVG canvas; and then creates a new line segment between the neighboring point and the active point. The line segments run from the center of the active point's symbol to the center of each of its neighbors. The function puts these line segments inside a new group element (corresponding to a `<g>` node); this facilitates removing the set of lines when the mouse moves off the point.

We provide here the complete code for one of the helper functions, `addLines()`. We include it to show how JavaScript methods are used to construct new elements in the display:

```
function addLines(obj, neighbors, numNeighbors)
{
    var x, y, x1, y1;
    var tmp = obj.getBBox();
```

```
    x = tmp.x + tmp.width/2;
    y = tmp.y + tmp.height/2;

    lineGroup = document.createElementNS(svgNS, "g");
    obj.parentNode.appendChild(lineGroup);
    var ids = obj.getAttribute('id') + ": ";
    for(var i = 1; i <= numNeighbors ; i++) {
      var target;
      target = document.getElementById(neighbors[i]);
      ids = ids + " " + neighbors[i];

      tmp = target.getBBox();
      x1 = tmp.x + tmp.width/2;
      y1 = tmp.y + tmp.height/2;

      var line = document.createElementNS(svgNS, "line");
      line.setAttribute('x1', x);
      line.setAttribute('y1', y);
      line.setAttribute('x2', x1);
      line.setAttribute('y2', y1);
      line.setAttribute('style', "fill: red; stroke: red;");

      line.setAttribute('class', "neighborLine");
      lineGroup.appendChild(line);
    }
    window.status = ids;
}
```

Note that although libcairo uses only `<path>` elements to draw objects, we can use higher-level elements such as `<line>` in our JavaScript drawing. Also note that when we create the new nodes, we must include the name space or otherwise the node is not recognized as SVG. The line segments are put into a group that is stored as a global variable in JavaScript. This makes it easy to remove the lines in one operation when the mouse moves off the point. This is illustrated by the definition of the `hideLines()` function which simply checks the value of the variable `lineGroup` and removes its children if it is an existing node:

```
function hideLines()
{
    if(typeof lineGroup != "undefined") {
      lineGroup.parentNode.removeChild(lineGroup);
      lineGroup = document.createElementNS(svgNS, "g");
    }
}
```

This example has demonstrated how to use JavaScript to dynamically manipulate an SVG display using data created in R when the plot was originally created. Again, in the viewing stage, when the JavaScript is running, the original data in R are not available. The matrix

of ordered neighbors for each point that was computed in R is available within the JavaScript code. If the script needs any of the additional data, then they must be placed in the document in the annotation stage.

An alternative approach to the above example is to add, in the annotation stage, all of the nearest neighbor lines for all of the points. We would set the `visibility` style attribute to "hidden" so these line segments would not be displayed when the document is loaded by the viewer. Then, a point's `onmouseover` function call would identify the appropriate line segments emanating from the point to its nearest neighbors and change their visibility attribute to "visible". Similarly, the `onmouseout` event would change them back to "hidden". This alternative approach is similar in spirit to those examples in Section 4.2, i.e., JavaScript functions do little other than change attribute values on nodes in the document. There are run-time benefits to this approach because we do not have to create the lines each time the user moves over a point. However, there are many more lines in the display, and the loading of the SVG file may be slower. This trade-off involves the issue of where the computations are performed. The approach that leaves the line creation to JavaScript also generalizes to allow dynamic specification of the number of nearest neighbors, e.g., with a slider or HTML spin box.

**Example 8.** Highlighting nodes and edges in a graph.

In this example, we explore how to annotate an SVG document created via **Rgraphviz** (Gentry *et al.* 2011) to make the graph interactive. Specifically, when the mouse moves over a node in the graph, the node and its connections to other nodes are highlighted. In addition, the other connections between nodes in the graph recede in appearance (see Figure 13).

This example is adapted from Gentry *et al.* (2010). We follow their example and make a simple graph with 10 nodes using the **graph** package.

```
R> library("Rgraphviz")
R> library("RJSONIO")
R> set.seed(123)
R> V <- letters[1:10]
R> M <- 1:4
R> g1 <- randomGraph(V, M, 0.8)
```

We use a circular `"twopi"` layout of the nodes with

```
R> doc <- svgPlot(plot(g1, "twopi",
+    attrs = list(node = list(fillcolor = "white"))))
```

We then examine the resulting SVG content

```
R> top <- xmlRoot(doc)[["g"]][["g"]]
R> table(names(top))

   g path
  10   55
```

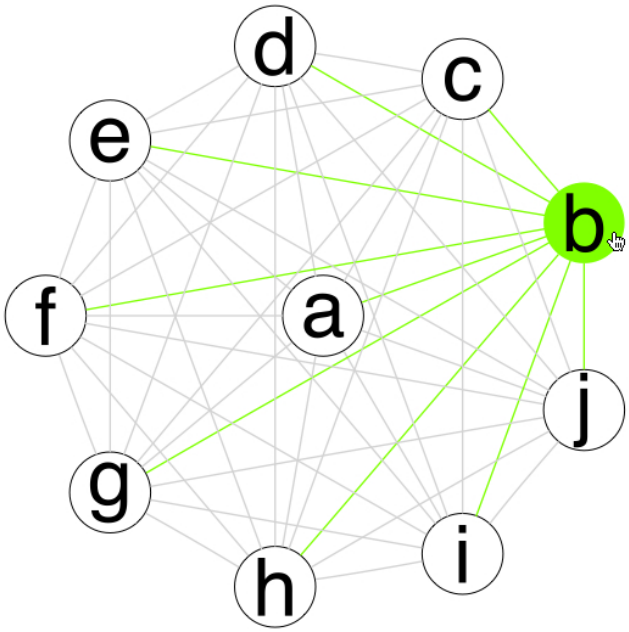


Figure 13: This graph is made using the **Rgraphviz** (Gentry *et al.* 2011) package. The graph has been annotated to make it interactive, where a mouse-over event on a node brings that node and its edges to the forefront, and the remaining edges recede into the background. This interactivity relies on JavaScript. The mouse-over event triggers a call to a JavaScript function that finds the edges and nodes that need to be modified and changes their color by modifying the corresponding SVG elements. The information identifying the connections between nodes is supplied via JavaScript variables whose values were computed in R and stored in the SVG document as JavaScript variables. This particular graph is not of great interest, being a simple random graph. However, the ideas apply to more interesting data such as package dependencies and call graphs illustrating which functions call which other functions.

We see that we have 65 nodes; 10 are `<g>` nodes and the remainder are `<path>` elements. In this layout, the first 10 SVG elements in the graph specify the text labels for the graph's nodes. The remaining 55 correspond to the 10 circles, one for each node, and the 45 edges corresponding to the 10 choose 2 undirected connections.

We can find out more about the structure of the graph with `agopen()`.

```
R> layout2pi <- agopen(g1, layoutType = "twopi", name = "bob")
```

The `layout2pi` variable contains all the information about the positions of the nodes and edges. `renderGraph()` uses this to draw the graph using R's graphics primitives (circle, line, text, etc.). So rather than using R's graphics and post-processing the resulting SVG document, we might consider creating the SVG directly from the layout information returned by `agopen()`. We wouldn't have to identify the node and edge elements in the SVG tree as we would explicitly create them. We could give them meaningful `id` attributes to make annotating the SVG content easier. This is a reasonable approach, but it requires that we permit the R user to specify the color, font family and size, line types and widths, etc. for

the nodes and edges. Furthermore, the R user would not be able to place additional elements on the plot with regular R graphics functions so it is best to work with the SVG document that R creates.

We can however make use of the 'Ragraph' object (returned by `agopen()`) to identify the elements in the SVG. For example, we can look at the edges in the `AgEdge` slot of the layout. This is a list with 45 elements. The order of the elements in the top-level SVG `<g>` element corresponds to the order of the edges in this list.

Now that we have some understanding of the structure of the SVG document and how to identify the nodes and edges, we can proceed with our goal of making the display interactive. We add code to the document that allows the user to mouse-over/click on a node and highlight the associated edges. The action is done via JavaScript. We start by identifying each node by a number, its index in the list of nodes. To highlight the node and edge, we call the function `highlightEdges()`, passing it that index. The `highlightEdges()` function determines the ids of the edges associated with that node and uses these to retrieve the corresponding element in the display. It then sets their color attribute to, e.g., chartreuse. The function also changes the color of the other edges and nodes to make them less visible. When the mouse leaves the node, we set the color of the edges back to their original colors.

The first step is to add an `id` attribute on each of the SVG elements corresponding to the nodes and edges. The function `addGraphIds()` does this for us, e.g.,

```
R> ids <- addGraphIds(doc, layout2pi)
```

This uses the node labels as the node ids. The edge ids are of the form, "edge:sourceId-destinationId", e.g., "edge:a-b", identifying the two end points with their ids and indicating the direction of the edge.

To implement `highlightEdges()`, we need information for each node giving the ids of the associated edges. It is natural to represent this in R as a list with each element being a character vector giving the edge identifiers corresponding to the node. This is relatively straightforward to create in R using `getEdgeInfo()` (in **SVGAnnotation**). However, we need it in JavaScript. The `addECMAScripts()` function will take care of serializing this R object in a suitable form so that it can be used directly in JavaScript code. It uses the `toJSON()` function in the **RJSONIO** package (Temple Lang 2011a) to do this. The following illustrates the form in which it will appear in the JavaScript code:

```
R> cat(toJSON(getEdgeInfo(g1)))
{
 "a": [ "edge:a-c", "edge:a-d", "edge:a-e", "edge:a-g", "edge:a-h",
        "edge:a-i", "edge:a-j", "edge:a-b", "edge:a-f" ],
 "b": [ "edge:b-c", "edge:b-d", "edge:b-e", "edge:b-f", "edge:b-g",
        "edge:b-h", "edge:b-i", "edge:b-j", "edge:a-b" ],
...
 "j": [ "edge:a-j", "edge:c-j", "edge:d-j", "edge:e-j", "edge:g-j",
        "edge:h-j", "edge:i-j", "edge:b-j", "edge:f-j" ]
}
```

Now that we have arranged for the node-edge information to be available to the `highlightEdges()` function, the JavaScript code to manipulate this is, at its very basic, like the following:

```
function highlightEdges(evt, row, color)
{
  var labels = edgeTable[row];
  var reset = false;
  var el;

   /* If no color was specified, we reset the original values. */
  if(typeof color == 'undefined') {
    reset = true;
    color=black;
  } else {
    currentStyle = evt.target.getAttribute('style');
  }

    /* Loop over the edges associated with this node */
  for(var i = 0; i < labels.length; i++) {
      el = document.getElementById(labels[i]);
      setEdgeStyle(el, "stroke: " + color);
  }

  labels = edgeDiff[row];
  var stroke = "lightgray";
  if(reset) stroke = "black";

    /* hide the other edges */
  for(var i = 0 ; i < labels.length; i++) {
    el = document.getElementById(labels[i]);
    setEdgeStyle(el, 'stroke: ' + stroke);
  }

   /* Restore or set the style for the target node. */
  if(reset) evt.target.setAttribute('style', currentStyle);
  else evt.target.setAttribute('style', 'fill: ' + color);
}
```

Given this JavaScript function, we are ready to put all the pieces together to annotate the SVG document. The steps are: (i) put mouse event handlers on the SVG elements corresponding to the nodes in the graph, (ii) compute the information about the edges for each node, (iii) add the code and this edge information data to the SVG document. We do this as

```
R> ids <- addGraphIds(doc, layout2pi)
R> els <- getNodeElements(doc)
R> sapply(seq(along = els), function(i) addAttributes(els[[i]],
+    onmouseover = paste("highlightEdges(evt, ", i- 1, ", "chartreuse");"),
+    onmouseout = paste("highlightEdges(evt, ", i-1, ");")))
R> info <- getEdgeInfo(g1)
R> names(info) <- seq(from = 0, length = length(info))
```

```
R> otherEdges <- lapply(info, function(x) setdiff(ids$edgeIds, x))
R> mapply(addLink, els, ids$nodeIds, MoreArgs = list(silent = TRUE))
R> jscript <- c(system.file("examples", "Javascript", "highlightEdges.js",
+    package = "SVGAnnotation"), system.file("examples", "Javascript",
+    "setEdgeStyle.js", package = "SVGAnnotation"))
R> addECMAScripts(doc, jscript, TRUE, edgeTable = info,
+    edgeDiff = otherEdges)
R> saveXML(doc, "graphviz.svg")
```

Note we pass the edge information to JavaScript via `addECMAScripts()` and create JavaScript variables named `edgeTable` and `edgeDiff`.

## 10. Animation

SVG supports two types of animation: declarative, which solely uses SVG facilities; and scripted, which relies on JavaScript. In this section, the focus is on the declarative approach, however, we present examples of both. The SVG animation facilities make it quite easy to produce animations similar to GapMinder, by simply annotating a scatter plot display with animation elements.

The basic concept of a "pure" SVG animation is that animation elements provide directions on how to move an object, or group of objects, and how to change the shape or appearance/style of an object. These animation elements are added as children of the object that is being animated. The animation tag names are `<animate>`, `<animateMotion>`, `<animateTransform>`, `<animateColor>`, and `<set>`. These tags act on a particular attribute of the parent object. For example, the `<animateTransform>` tag can be used to change the `width` attribute of its parent, which causes the parent object to grow or shrink. As another example, `<animateTransform>` can change the `visibility` attribute in order to make an object become hidden or visible. Furthermore, with `<animateMotion>`, we can specify a new location for the object, which causes the object to move to this new position.

To get an idea as to how this works, the following SVG animation takes two seconds to move a circle from its original location at the time of loading to a new location and simultaneously shrink the circle from a radius of 5.4 to 3. This mini-animation is the building-block for the scatter-plot animation shown in Figure 4. There, each point represents a country, and the x and y locations correspond to the country's average life expectancy and income for a particular decade. As the points move, they grow or shrink according to the change in population size from one decade to the next. The animation is described in more detail in Example 9.

```
<circle x="95.1" y="369.8" r="5.4"
        style="fill-rule: nonzero; fill: rgb(100%,0%,0%); ...">
    <animateMotion id="move1" from="95.1, 369.8" to="66.7, 412.7"
                   fill="freeze" dur="2s" begin="0s"/>
    <animateTransform attributeName="transform" type="scale"
                   fill="freeze" to="3" dur="2s" begin="0s"/>
</circle>
```

The <animateMotion> tag provides the details for moving the circle across the canvas, and the <animateTransform> tag provides information to resize the circle. The attributes from and to on <animateMotion> provide the beginning and ending location for the circle. Whereas to on <animateTransform> indicates the final size of the circle. The type attribute of "scale" indicates that the circle is to be scaled; other possible values for this attribute are rotate, skewX, skewY, and translate. The fill attribute is not to be confused with the fill attribute of the circle. It refers to what is to happen at the end of the animation. The value of "freeze" indicates that the final value should be kept. Without it, the attribute being animated would return to its original value, i.e., the circle would return to its starting position or it would return to its original size. The attributes begin and dur, indicate that the movement of the circle and its change in size are to both start at "0s", i.e., when the page is loaded, and last for 2 seconds.

The movement of the circle and the changing of its size are coordinated by specifying that both animations are to begin at the same time (begin is at 0 seconds) and take the same amount of time (2 seconds). With <animateMotion>, it is also possible to specify the path the circle would take in traveling from one location to the other. Any animation element requires a dur attribute to specify its duration. The value can be provided in seconds, e.g., dur="10s", or minutes, e.g., dur="2min". It is also possible to use clock values, e.g., dur="2:10" for 2 minutes and 10 seconds. The SVG clock starts ticking when the document has completed loading.

In addition to specifying the duration of an animation, it is also possible to specify when the animation is to begin or end. The values of these attributes can be absolute values, as with dur, or they can be relative to the start or end of another animation. For example, if we change the begin value in our <animateTransform> to "move1.end+3s", then the circle would start shrinking 3 seconds after it finished moving to its new location, i.e., 3 seconds after the animation with an id attribute value of "move1" ends. This can be useful when synchronizing parts of an animation.

The <set> element behaves somewhat differently in that it is used to change attribute values such as fill. For example, the color of the circle could be changed when it reaches its new location by embedding the following element as a child of <circle>:

```
<set attributeName="fill" fill = "freeze"
     attributeType = "CSS"  to = "#FFDB00FF"
     begin = "move1.end"/>
```

The attribute called attributeName specifies which attribute of the parent node is to be "animated". (Note that attributeName in <animateTransform> is "transform", which is not an explicit attribute on the circle.) The to attribute indicates the new color of the circle. The change of color begins when the circle has finished moving because begin has a value of "move1.end". As mentioned already, fill refers to what is to happen at the end of the animation; the value of "freeze" indicates that the final color should remain.

One additional point to note is that the attributeType attribute in the <set> element is used to indicate where the attribute being changed can be found. A value of "CSS" means that the fill color is located in the style attribute of <circle>. If the circle had specified the value of its fill color in a presentation attribute, then we would have given attributeType a value of "XML".

### 10.1. Scatter plot animation

The `animate()` function in the **SVGAnnotation** package takes a scatter plot and moves the points around the canvas through a sequence of steps. In each step, a point moves continuously from one position to another; then in the next step, the point moves from there to yet another location. We implement it by adding `<animateMotion>` elements to each of the SVG elements corresponding to points in the plot. Each "point" in the SVG document will have as many `<animateMotion>` elements as there are time steps. In addition, if we want the size of the points to represent the value of an additional variable, which also changes in time, then we can specify the radii of the circles in each stage. In this case, the `animate()` function will add `<animateTransform>` elements to the display to change the circle sizes. Also, if the point is to change color, `<set>` elements are also added. These various options are specified in the function call.

The `animate()` function takes as an argument the SVG display of the initial scatter plot. It could in fact create the plot itself based on the data for the first time step, but the caller will most likely want to create the plot in advance in order to customize its appearance (e.g., title, axes labels, lines, etc.). A second argument provides the data to be animated. The data are provided as a data frame giving the locations of the points in the scatter plot for the different steps. It is structured as a column of x values and a column of y values for each step stacked on top of each other. In addition, the `which` parameter indicates to which step the points belong. We can provide radii for the circles via the `radii` parameter, if we want to change the size of the circles. In addition, we can supply colors for each stage via the `colors` parameter. We demonstrate some of these features and some implementation issues in the next example, Example 9.

**Example 9.** Animating scatter plots through snapshots in time.

In this example, we display 11 decades of United Nations data via a scatter plot animation shown in Figure 4. The following is a snippet of the data we use,

```
R> head(gapM)
```

```
  longevity    income population   yr    country
1     39.70 4708.2323    6584000 1900  Argentina
2     63.22 6740.9394    4278000 1900  Australia
3     42.00 5163.9268    6550000 1900    Austria
4     22.00  794.8383   31786000 1900 Bangladesh
5     51.11 4746.9662    7478000 1900    Belgium
6     32.90  705.4758   21754000 1900     Brazil
```

```
R> tail(gapM)
```

```
    longevity   income population   yr        country
391    80.550 48264.67    4610820 2000         Norway
392    69.906  6875.51   28302603 2000           Peru
393    70.303  3030.88   89468677 2000    Philippines
394    78.920 20149.08   10605870 2000       Portugal
395    78.471 32334.53   60609153 2000 United Kingdom
396    77.890 42445.70  298444215 2000  United States
```

We want to plot the variable `income` along the x axis and `longevity` along the y axis; we use `yr` to indicate the time/step; and `population` determines the radius of the circle.

These data have many missing values, and most of the work for us is in cleaning up that data and formatting it for `animate()`. Below is a snippet of the data preparation that we need to do. Here we are computing the radii for the circles. When the data are missing, we set the radii to a small value.

```
R> rad <- 1 + 10 * sqrt(gapM$population)/max(sqrt(gapM$population))
R> disappear <- is.na(gapM$longevity) | is.na(gapM$income) |
+    is.na(gapM$population)
R> rad[disappear] <- 0.00001
R> radL <- lapply(ctry, function(i) rad[gapM$country == i])
R> names(radL) <- ctry
```

Here `ctry` is a vector of the names of the countries to be included in the animation.

Once the data are prepared, we plot the points for the first decade, which creates the starting frame of the animation. We control much of the layout, placing the tick marks, grid lines, and labels ourselves, as shown here.

```
R> doc = svgPlot({
+    plot(longevity ~ income, subset(gapM, yr == 1900 & country %in% ctry),
+      pch = 21, col = colsB, bg = colsI, xlab = "Income",
+      ylab = "Life Expectancy", axes = FALSE, xlim = c(-400, 50000),
+      ylim = c(20, 85))
+    box()
+    y.at <- seq(20, 85, by = 5)
+    x.at <- c(200, 400, 1000, 2000, 4000, 10000, 20000, 40000)
+    axis(1, at = x.at, labels = formatC(x.at, big.mark = ",", format = "d"))
+    axis(2, at = y.at, labels = formatC(y.at))
+    abline(h = y.at, col = "gray", lty = 3)
+    abline(v = x.at, col = "gray", lty = 3)
+    legend(35000, 40, longCont, col = colB, fill = colB, bty = "n",
+      cex = 0.75)
+  })
```

Here `colB` holds the colors for each circle border, `colI` has the colors for the interior of the circles, and `longCont` has that names of the continents. We would also like to provide tooltips for each of the points that display the particular country's name. We annotate the plot using `addToolTips()` as follows

```
R> addToolTips(doc,
+    as.character(gapM$country[gapM$yr == 1900 & gapM$country %in% ctry]))
```

The initial view or starting point of the animation is now in the variable `doc`. We provide this to `animate()`, along with the subsequent slices of the data for each decade. We specify the duration of the animation via the `interval` parameters (the total duration of the animation can also be specified via `dur`), and the radii of the circle at each time period (e.g., decade) is provided in `radii`. The call to `animate()` is

```
R> animate(doc,
+    gapM[gapM$country %in% ctry, c("income", "longevity")],
+    gapM$yr[gapM$country %in% ctry],
+    dropFirst = TRUE, labels = seq(1900, length = 11, by = 10),
+    begin = 0, interval = 3,  radii = radL[ctry])
```

The `animate()` function also needs to know the range of the horizontal and vertical data used to create the initial plot. If this is not the range of the entire data, the caller must specify these explicitly. Furthermore, background labels and point colors can be changed at each time period; these can be specified via `labels` and `colors`, respectively.

The `animate()` function handles the complication that arises from the coordinate system of the data being different from the SVG coordinate system. The data values must be transformed into the SVG coordinate system before we add an `<animateMotion>` node to move the points to new locations. Below is a snippet of the code in the `animate()` function that does this.

```
R> rect <- getBoundingBox(plotRegion)
R> transformX <- function(vals, xlim, rect) {
+    (vals - xlim[1]) * (rect[2, 1] - rect[1, 1])/(xlim[2] - xlim[1]) +
+    rect[1, 1]
+  }
```

The `getBoundingBox()` function returns the limits of the plotting region in the SVG coordinate system, and these are used to transform the data values (in `vals`). If the data are to be plotted on a log scale, then the data must be transformed as well. That is, the parameter `log` in `plot()` will create a mismatch between the scale of the data and the plotting region. Instead, the caller will need to take logs of the data and handle the specification of the tick mark labels if they are to appear in unlogged units.

A second complication arises due to the SVG element for drawing the plotting symbol being a `<path>` rather than a `<circle>`. The `animate()` function examines the path to determine whether or not the plotting symbol is a circle. If the `radii` parameter is provided and the plotting symbol is a circle, then the `<path>` is replaced by a circle so that it can easily grow and shrink as it moves through the stages of the animation. Currently, all other symbols will move around the canvas and `radii` will be ignored, but a scaling transformation is possible.

## 11. Controlling animation with JavaScript

In this section, we examine how to animate plots using JavaScript explicitly via programmatic changes to the display at viewing time. There are some kinds of animations that are difficult to achieve with the declarative approach of the SVG model; for example, to change a path dynamically (i.e., at run/viewing-time) requires scripting. We provide a simple example to illustrate a few of the ideas in the JavaScript approach.

The `setInterval()` function is key to using JavaScript for animation. This function sets a timer to go off at repeated intervals. When it does, previously specified JavaScript code is evaluated. For example, in the code

```
animationHandle = setInterval("animationStep(polyIds,
                                             stateResultsByYear,
                                             yearLabels,
                                             polyStateNames)",
                              Interval);
```

the `setInterval()` function is called with the values "animationStep(polyIds, stateResultsByYear, yearLabels, polyStateNames)" and Interval. This means that every `Interval` milliseconds, the specified call to `animationStep()` will be made. The return value from the `setInterval()` function is a reference to the timer just established. We assign this to the global variable `animationHandle`. To stop the animation, we can call the JavaScript function `clearInterval` with `animationHandle` as its argument, and the timer will be removed.

Previously, we encountered the use of scripts to interact with an SVG graphic in Section 4.2. There, a command was called as the result of a mouse event, such as a mouse-over and mouse-out. In addition, there are the click, mouse-down, and mouse-up events associated with the mouse button. Another event that can be used to trigger a function call is "load", which occurs when the SVG document has been loaded and rendered in the viewer. Each of these events has a corresponding attribute, e.g., `onmouseover`, `onclick`, and `onload`, and the value of the attribute is one or more JavaScript statements to be executed when the event occurs (see Example 7 and Example 5). Any of these events can also be used to initiate an animation. Below, we place `setInterval()` within an `init()` function, and have it called at the time the SVG document is loaded, i.e., the top of the document appears as

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="504pt" height="504pt" viewBox="0 0 504 504"
     version="1.1" onload="init(evt)">
```

The key attribute is `onload`. The corresponding JavaScript code is

```
var animationHandle;
function init(evt) {
...
  animationHandle = setInterval("animationStep(polyIds,
                                               stateResultsByYear,
                                               yearLabels,
                                               polyStateNames)",
                                Interval);
}
```

The following example demonstrates how to use the timer to create an animation.

**Example 10.** Animation with JavaScript and SVG.

In this example, we animate a political map of the states within the USA. Each state is painted according to the party of the winning presidential candidate, with Republican red, Democrat blue, and Independent green. For any given year, this gives us a visualization of the geographical voting patterns. We obtained this information for 1900 to 2008 from onwards http://electionatlas.org/.

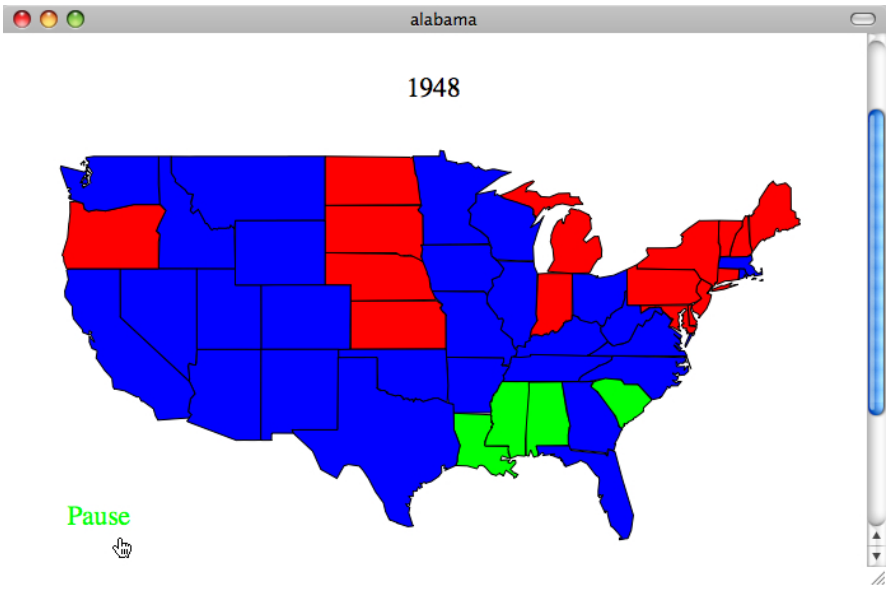


Figure 14: This screen shot was captured in the midst of an animation where the colors of the states (red for Republican, blue for Democrat, and green for Independent) change to reflect the party of the presidential candidate who received the most votes in the state. The animation is initiated with a mouse-click on the term "Start", and once it starts, it can be paused with a mouse-click on "Pause". The animation is controlled by a JavaScript timer, where at regular intervals the SVG is modified. In particular, the state polygons are filled with color for the next election year and the map title is changed to the next date.

The data are in `electionHistory`, a named list with an element for each of the presidential elections. Each such element is a character vector giving the colors for each of the states. The names on this character vector identify each state. The names of the `electionHistory` list identify the election year, e.g., 1900, 1904, ..., 2008.

We start by plotting a grey state map of the US. Also, since we do not want the animation to start when the page is loaded, we add a label titled "Start" which we will have respond to a mouse click to start the animation.

```
R> library("SVGAnnotation")
R> library("maps")
R> data("electionHistory")
R> doc <- svgPlot({
+    m <- map("state", fill = TRUE, col = "grey66")
+    title("Presidential Election Results by State 1900-2008")
+    text(m$range[1] + 3, m$range[3] + 1, "Start", col = "green")
+  })
```

At regular intervals, e.g., 1 second, we will update the title to the next election year and change the color of the polygon(s) for each state. The title element is reasonably easy to find using the function `getAxesLabelNodes()`:

```
R> labels <- getAxesLabelNodes(doc)
```

However, we change the nature of this title node to be a regular text element rather than a path element. The reason for this is that it is significantly simpler to modify the value of the text node than that of a path which represents text:

```
R> title <- asTextNode(labels[[1]][[1]],
+    "Presidential Election Results by State 1900-2008")
```

We put an `id` attribute on the new title element, which is the first element of the `labels` list, so that we more easily can retrieve it with JavaScript code at viewing time.

```
R> xmlAttrs(title) <- c(id = "title")
```

We add an `id` attribute to the "Start" label, and make it respond to `onclick` events. A mouse-click will causes the state of the animation to change from stopped to running, running to paused, or from paused to running, and to also change the text of this label appropriately. We do this with the JavaScript function `toggleAnimation()` and add the call to the label with

```
R> start <- asTextNode(labels[[2]][[1]], "Start")
R> xmlAttrs(start) <- c(id = "Start", onclick = "toggleAnimation(evt)")
```

We'll also add a tooltip to this label which will both provide information to the viewer but also make the entire label (and not just the path) active. So we add this and the CSS file which takes care of the appearance of the underlying rectangle:

```
R> addToolTips(labels[[2]], "Start or pause the animation")
```

The next step is to add an identifier to each polygon on the map. The map contains 63 polygons, as some states such as Washington are drawn using multiple polygons. Each polygon has a unique name, which we add as ids to the elements:

```
R> pts <- getPlotPoints(doc)
R> mapply(function(node, id) xmlAttrs(node) <- c(id = id), pts, m$names)
```

We also put a tooltip on each polygon to identify the state:

```
R> addToolTips(pts, m$names)
```

Finally, we add the necessary variables to the JavaScript code in the SVG document along with the JavaScript code for handling the animation. We will assume the JavaScript code is in a file named `animateElectionMap.js`, and add the code and variables to the SVG document using

```
R> polyStateNames = gsub(":.*$", "", m$names)
R> polyStateNames[ polyStateNames =="district of columbia"] = "d. c."
R> addECMAScripts(doc, system.file("examples", "Javascript",
+      "animateElectionMap.js", package = "SVGAnnotation"),
+    stateResultsByYear = electionHistory,
+    yearLabels = names(electionHistory),
+    polyIds = m$names, polyStateNames = polyStateNames,
+    insertJS = TRUE)
```

Note that we are adding 4 variables: `stateResultsByYear`, which holds the colors for the states; `yearLabels`, the years of the elections; `polyIds`, the unique names for the polygons; and `polyStateNames`, the state names for each of the polygons (so Washington appears multiple times, once for each polygon).

The JavaScript code will update the value of `fill` for each polygon. It is easiest to override the value for fill whene all the style information are in separate attributes on the element. To this end, we will change the way the SVG style attributes are stored for each of the polygon nodes. We use `convertCSSStylesToSVG()` for this:

```
R> convertCSSStylesToSVG(pts)
```

That is all we need to do to our SVG document at this point so we can save it to a file:

```
R> saveXML(doc, "exJSAnimateElectionMap.svg")
```

All that remains in our example is to write the relevant JavaScript code. We need the `toggleAnimation()` function and the code that does the actual changing of the color of each state/polygon. This code also checks to see if the animation is complete and, if so, changes the label from "Pause" to "Start".

We begin by defining some global variables for controlling the animation. These are the identifier for the interval, the current year being displayed and the amount of time between updates in the animation:

```
var animationHandle = null;
var currentYear = 0;
var Interval = 1000;
```

Next we define the function that updates the display. This function increments the value of `currentYear` and checks to see if the animation is done. If the animation has completed, we reset the variables and the display. If not, we update the display with the values for this election year by calling `displayYear()`. The entire `animationStep()` appears here.

```
function animationStep(ids, colors, yearLabels, stateNames)
{
  currentYear++;

  if(currentYear >= yearLabels.length) {
     var el = document.getElementById("Start");
     setTextValue(el, "Start");
     clearInterval(animationHandle);
     animationHandle = null;
     for(var i = 0; i < ids.length; i++) {
        var el = document.getElementById(ids[i]);
        if(el)
          el.setAttribute('style', 'fill: ' + '#A8A8A8');
     }
     var title = document.getElementById('title');
```

```
    setTextValue(title,
           'Presidential Election Results by State 1900-2008');
    return;
  }

  displayYear(currentYear, ids, colors, yearLabels, stateNames);
}
```

The function `displayYear()` updates the colors of the polygons and and changes the year displayed in the title node. The simple `<style>` attribute contains fill information only because the other style details have been provided as indiviual attributes and so will remain in effect. It is defined as,

```
function displayYear(year, ids, colors, yearLabels, stateNames)
{
  for(var i = 0; i < ids.length; i++) {
    var el = document.getElementById(ids[i]);
     /* Lookup the year by name. Then within the year, look up
        the state, using the actual name not the polygon id. */
    var col = colors[yearLabels[year]][ stateNames[ i ] ] ;
    if(el && col)
       el.setAttribute('style', 'fill: ' + col);
  }

  var title = document.getElementById('title');
  setTextValue(title, yearLabels[year]);
}
```

The `toggleAnimation()` function is in charge of starting and pausing the animation. It is called when there is a mouse-click on the Start/Pause "button". The function examines the value of the animation handle and if this is non-null, terminates the animation. Otherwise, it starts the animation and shows the first year. It also updates the value displayed on the "Start" button as shown below.

```
function toggleAnimation(evt)
{
  var label;
  if(animationHandle) {
    clearInterval(animationHandle);
    animationHandle = null;
    label = currentYear > 0 ? "Restart" : "Start";
  } else {
    animationHandle= setInterval("animationStep(polyIds,
                                    stateResultsByYear,
                                    yearLabels, polyStateNames)",
                                 Interval);
    if(currentYear >= yearLabels.length) currentYear = 0;
```

```
    displayYear(currentYear, polyIds, stateResultsByYear,
                yearLabels, polyStateNames);
    label = "Pause";
  }
  var start = document.getElementById('Start');
  setTextValue(start, label);
}
```

The utility function which changes the value/label of a `<text>` node is given by

```
function setTextValue(node, val)
{
 node.firstChild.data = val;
}
```

## 12. Creating GUIs

In Example 10, we added a rectangle, containing the text "Start", to act as a button, where clicking on this region in the plot started an animation. In this section, we extend this idea and demonstrate how to provide interactivity through user controls, such as buttons, check boxes, and sliders that are drawn on the display using SVG & JavaScript rather than R. The Carto:Net community (Berger *et al.* 2010) has developed an open source library of SVG graphical user interface (GUI) controls to facilitate interactivity in SVG documents. The library consists of JavaScript functions to build the controls as SVG elements and respond to user interaction. This library is available from http://carto.net/, and is also distributed with the **SVGAnnotation** package, for convenience. It includes controls such as a check box, radio button, button, slider, text box, and selection menu, which are rendered using native SVG elements and are quite different from their JavaScript equivalents.

By using these SVG GUI components, we can create applications that have rich user interfaces. An alternative approach is to embed the SVG graphic in an HTML document and control it through buttons and boxes in an HTML form. We show how to do this in Section 13. With SVG, there is the potential to design unique, sophisticated GUI elements, such as sliders, dial knobs, and color choosers. Another potentially useful feature of SVG-based GUI components is that they can be rendered with SVG filters and transformations, e.g., at an angle, with different filters (e.g., Gaussian blurring), or even animated. The main disadvantage is complexity. SVG GUI elements are somewhat more complex to use in programs than the "built-in" HTML form elements. Fortunately, the Carto:Net library offers a variety of GUI elements that can be easily embedded in the document.

In this section, we provide examples of how to use two of these GUI controls, a slider and check box. The first example adds a slider to a display in order to give the viewer control of a smoothing parameter. The second example uses check boxes to add and remove groups of data (time series) from a plot.

Although the GUI control is being drawn with JavaScript commands, the basic approach to annotating the SVG remains essentially the same. The annotation stage now typically requires the following additional tasks:

- Enlarge the viewing area so that there is room for the GUI control (i.e., change the `viewBox` attribute on the root element).
- Add an initialization script to the document in order to create the GUI object when the document is loaded (i.e., add a call to a JavaScript initialization function via the `onload` attribute on the `<svg>` element). This script needs to call the JavaScript function provided by Carto:Net that creates the GUI object.
- Populate the `<defs>` node with elements which contain the generic drawing instructions for the control that are provided by Carto:Net. Plus, locate the GUI in the display by adding an empty `<g>` tag that Carto:Net scripts use to draw the graphical representation of the control.
- Add the Carto:Net scripts to the document along with the application specific JavaScript functions that connect GUI events to changes in the display, e.g., a change in the location of the slider's thumb calls a function that reveals a new curve on a plot and hides from view the old curve.

### 12.1. Slider widget

In this section, we consider an example of how to embed a Carto:Net slider in an SVG document. The **SVGAnnotation** package provides a generic `addSlider()` function to add a slider to an SVG document. The `addSlider()` function takes care of setting up the SVG content according to the Carto:Net requirements. It enlarges the "viewBox" to make room for the slider, adds the initialization script to the `<onload>` attribute of the root node, adds the required `<g>` parent node for the slider, and inserts the required JavaScript. In addition, it places the slider thumb drawing instructions in the `<defs>` portion of the document. The declaration of the function is shown below.

```
addSlider <- function(doc, onload, javascript, id = "slider", svg = NULL,
  defaultJavascripts = c("mapApp.js", "helper\_functions.js", "slider.js"),
  side = "bottom", ...) {}
```

The `onload` argument gives the JavaScript code that will be invoked when the document is loaded. Also, `javascript` and `defaultJavascripts` take the file names for the JavaScript code (file names or text content) that contain, respectively, application specific functions and the required Carto:Net code. Finally, any JavaScript variables needed in the document are supplied via the ... argument.

The JavaScript slider can be queried at various times to find the location of the slider thumb, to set its value, to move the slider thumb, and to remove the slider from the display. This functionality is provided via the JavaScript slider object's methods `getValue()`, `setValue()`, `moveTo()`, and `removeSlider()`, respectively. These methods are invoked as, e.g., `slider.getValue()`. The next example, demonstrates how to use the Carto:Net slider.

**Example 11.** Controlling smoothness with a slider.

In this example, we build a slider to interactively control the bandwidth of a kernel smoother (see Figure 5). When the viewer moves the slider thumb to a new position, the corresponding

fitted curve is overlaid on the scatter plot on the left-hand side of the display, and a scatter plot of the residuals for that curve is rendered in the plot on the right-hand side.

Similar to the approach of many of the previous examples, the computations for fitting the curve to the data are performed in R in advance for every possible selectable value of the smoothing parameter. The resulting curves and residuals are plotted in the SVG document. Then in the annotation stage, the SVG elements corresponding to each curve are given unique `id`s so that they can be identified. In this case, we use the identifier, "curve-lambda-N", where N is the parameter value associated with the curve. These nodes are also annotated with a `visibility` attribute so the curves can be exposed or hidden according to the value of the smoothing parameter selected by the viewer via the slider. The residuals are similarly annotated. To do this, we group together the residuals from the fit for a particular smoothing parameter, and place them within a `<g>` node that has a unique `id`, namely "residual-group-N". We also add a `visibility` attribute of "hidden" to each `<g>`.

The final annotation task is to add the slider to the document. A call to `addSlider()` does this for us. The value of the `onload` argument is a call to the `init()` function that creates the slider. When the document is loaded, this function will be called with the number of smoothing parameter values, which is also the number of fitted curves, and the number of positions on the slider. This extra information is used to coordinate the slider value with the curve and residual points to be displayed. We add this `onload` attribute and the supporting JavaScript code to the SVG document with

```
R> svgRoot <- xmlRoot(doc)
R> enlargeSVGViewBox(doc, y = 100, svg = svgRoot)
R> onl <- sprintf("init(evt, %d);", max(lambdas) )
R> jscript <- list.files(
+    path = system.file("examples", "Javascript", package = "SVGAnnotation"),
+    full.names = TRUE, pattern = "linkedSmoother")
R> addSlider(doc, onload = onl, svg = svgRoot, javascript = jscript,
+    id = "slider-lambda")
R> saveXML(doc, "linkedSmoother.svg")
```

The `id` parameter identifies the `id` of the SVG element in the SVG document that corresponds to the slider, i.e., the identifier on the `<g>` element that is added to support the slider.

The `init()` function simply instantiates the slider. It is defined as

```
var myMapApp = new mapApp(false,undefined);
var cur_lambda = 2;
var numPoints;

/*  Create the slider.  */
function init(evt, maxLambda, n) {
   var sliderStyles = {"stroke" : "blue", "stroke-width" : 3};
   var invisSliderWidth = 15;
   new slider("lambda", "slider-lambda", 100, 510, 2, 475, 510,
              maxLambda, 2, sliderStyles, invisSliderWidth,
              "sliderSymbol", setLambda, false);
   numPoints = n;
}
```

A call to `init()` creates the slider via the JavaScript constructor function,

```
var slider = new slider(id, parentNode, x1, y1, value1, x2, y2,
                        value2, startVal, sliderStyles,
                        invisSliderWidth, sliderSymb,
                        functionToCall, mouseMoveBool);
```

In our example, the `id` of "lambda" specifies a unique identifier for the slider that is used internally within Carto:Net; the `parentNode`, "slider-lambda", gives the id for the `<g>` element that will contain the slider; the triple `x1`, `y1`, `value1` provides the (x,y) coordinates for the location of the start of the slider and the starting value of the slider; similarly, `x2`, `y2`, `value2` provide this information for the end of the slider; and `startVal` provides the initial value of the slider. We further customize the slider by specifying presentation attributes via CSS. In our example, we set the `sliderStyles` so the slider will be drawn with a thick blue line. Finally, the events on the slider are handled by the JavaScript callback function, `setLambda()`, which we provide via the `functionToCall` argument, and the final argument, `mouseMoveBool` is "false", indicating not to trigger updates as the mouse drags the slider, i.e., only call `setLambda()` when the mouse is released and the action of dragging the slider thumb has stopped.

The `setLambda()` function has a helper function `setVisibility()`. Both are shown below. They are similar in spirit to JavaScript functions we have used in other examples in that they simply reset the visibility attribute for the appropriate group of points and curve (e.g., Example 6). They are defined as

```
function setLambda(evType, group, val)
{
    /* If it is the same value, do not do anything*/
    if(Math.floor(val) == cur_lambda) return(0);
    setVisibility(cur_lambda, 'hidden', numPoints);
    cur_lambda = Math.floor(val);
    setVisibility(cur_lambda, 'visible', numPoints);
}

function setVisibility(lambda, state, numPoints)
{
     var el;
     lambda = Math.floor(lambda);
     el = document.getElementById("curve-lambda-" + lambda);
     el.setAttribute('visibility', state);

     el = document.getElementById("residual-group-" + lambda);
     if(el) {
          el.setAttribute('visibility', state);
          return(0);
     }

     for(i = 0 ; i < numPoints; i++) {
```

```
        el = document.getElementById("residual-" + lambda +
                                     "-" + (i+1));
        el.setAttribute('visibility', state);
    }
}
```

We have put this code in the `linkedSmootherSet.js` file and so have already included it in the SVG document.

## 12.2. Check boxes for toggling elements in a plot

We can use a similar approach to create interactivity with other SVG GUI controls offered in Carto:Net. For users unfamiliar with event handling, we provide one additional example involving check boxes. The check boxes in Carto:Net support toggle events, i.e., when a box is checked or unchecked then a user-supplied function is called to perform some action. For example, in Example 12, several time series are overlaid on a plot and made visible or invisible according to whether or not a corresponding box is checked. That is, when a check box is toggled, this event then triggers a function call to change the visibility attribute of the associated times series.

The SVG document needs to be modified in the same way we did for the slider so as to include the check box GUI controls. In particular, the document should have an empty group `<g>` that will hold the elements for the check boxes and their labels. This needs to be added with a unique `id` to the appropriate place in the document; the checkBox JavaScript object needs to be initialized via JavaScript in the `onload` attribute on the root element; and JavaScript helper functions need to be included in the document.

The `radioShowHide()` function in R sets up a group of check boxes for toggling on and off curves in a plot. Once the initial SVG plot has been made, `radioShowHide()` post-processes the SVG in the annotation stage. This function handles all of the setup. Similar to the slider, `radioShowHide()` expands the viewing area to make room for drawing the check boxes; sets up the drawing areas for the check boxes; adds to the document the initialization script that creates the JavaScript check box objects when the document is loaded in the browser; and includes the JavaScript functions provided by Carto:Net that respond to the viewer actions and the application specific JavaScript functions that change the graphical display in response to viewers actions.

The function signature/declaration for the `radioShowHide()` function provides the information needed to perform these tasks:

```
radioShowHide <- function(doc, insertScripts = TRUE, within = FALSE,
  group = FALSE, labels = paste("Series", seq(length = numSeries)),
  extraWidth = 15 * (max(nchar(labels)) + 1),
  save = !is(doc, "XMLInternalDocument"),
  id.prefix = "series",
  checkboxCallback = if(is.na(within)) "togglePanel" else "toggle",
  jscripts = c("helper\_functions.js", "timer.js",
    "checkbox\_and\_radiobutton.js", "hideSVGElements.js"),
  numPanels = 1) {}
```

Briefly, `doc` specifies the SVG document to be annotated; `within` indicates whether a regular plot (`FALSE`) or matplot (`TRUE`) call was used to create the plot, which helps in finding the series; `group` allows the series to be matched up across **lattice** panels so that a series can be toggled on and off in all panels; `labels` contains the text labels for the check boxes; and `checkboxCallback` holds the JavaScript for the call back function.

**Example 12.** Showing and hiding currency exchange series with check boxes.

In this example, we create an interactive time series plot, where several times series curves are super-imposed on the same plot and check boxes along side the plot are used to toggle the display of the series on and off (see Figure 15). The check boxes are ordered according to the median value of the time series to make it easier to visually connect the check box with its time series. The text labels could also have their colors coordinated with the series to make this connection clearer. We create the SVG plot with the command

```
R> svgPlot({
+    matplot(eu$Date, as.data.frame(lapply(eu[,-1], log)),
+      type = "l", xlab = "Year", xaxt = "n", ylab = "log(exchange rate)",
+      main = "European Exchange Rate")
+    startYr <- min(eu$Date)
+    endYr <- max(eu$Date)
+    axis.POSIXct(1, at = seq(startYr, endYr, by = "year"), format ="%y")
+    abline(h = 1, col = "gray")
+  }, "euSeries.svg")
```

Note that in this case we are explicitly writing the SVG to a file rather than returning the SVG tree.

A simple call to `radioShowHide()` will do all the post-processing of the SVG document created in the `matplot()` call:

```
R> doc <- radioShowHide("euSeries.svg", within = TRUE,
+    labels = currency.names[match(names(eu)[-1], names(currency.names))])
```

We can also extend this functionality to plots with multiple panels. For example, we might draw the daily time series for the different currencies for each year in a separate panel by conditioning on year. Then toggling the currency check box would show/hide the corresponding curves in each of the panels. Again, the reader should view the example by visiting the package's Web site and viewing it there.

## 13. Interfacing with (X)HTML

The interactivity for the examples considered so far have been entirely contained within the SVG document. That is, the SVG or JavaScript code that responds to user interaction has been located in the SVG file. In this section, we demonstrate that it is also possible to control SVG using JavaScript that is placed within an HTML document, where the SVG display is also embedded in the HTML.

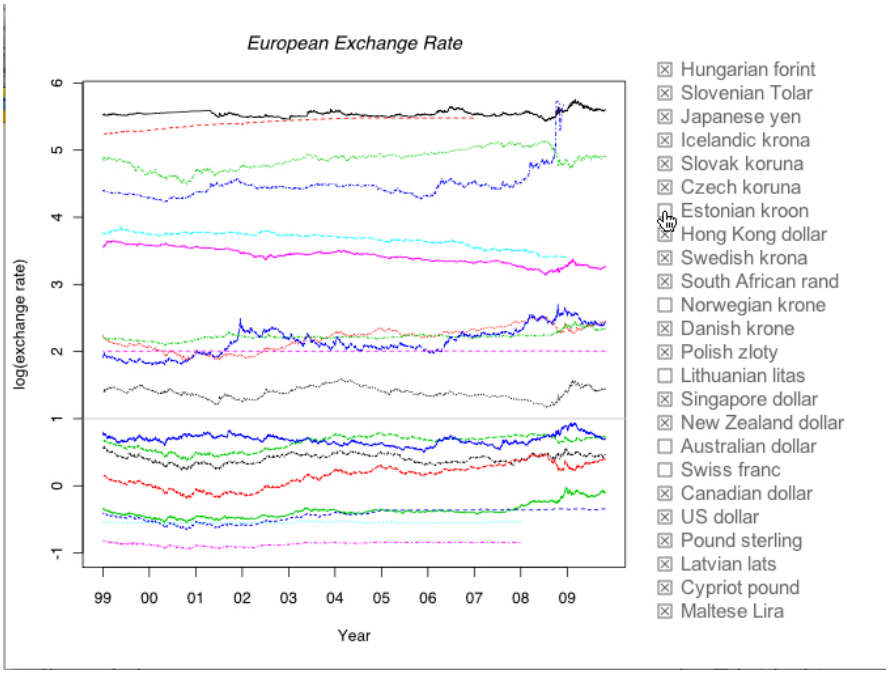


Figure 15: This screen shot shows a time series plot (via `matplot()`) of exchange rates against the Euro for 24 different currencies. The SVG is post processed to add check boxes to the right of the plot. These check boxes are interactive and allow the viewer to toggle on/off the display of the corresponding currency's time series. The SVG GUI controls shown here are provided by Carto:Net.

There are several benefits to this approach. For example, HTML forms can be used to control the interactivity, and these controls can be easily visually arranged using layout facilities in HTML, e.g., lists, and tables. Other advantages are that embedding the SVG document in HTML allows us to control the width and height of the area in which it is displayed; multiple SVG documents can be embedded in one HTML document; and SVG documents can be linked together via JavaScript in the HTML document. There can be separate JavaScript code in the different components, i.e., in the HTML document and in each SVG document. This increased locality and flexibility can also be slightly more complicated.

In this section, we provide two examples of this approach. The first revisits a previous example and creates interactivity through HTML forms rather than with `onmouseover` calls to JavaScript functions. It serves as a comparison between these two approaches. In the second example, we move the process of annotating the SVG entirely into the viewing stage, specifically at the point where the document is loaded into the browser. As such, JavaScript, rather than R, is used to augment the SVG elements with the necessary annotations for interactivity, e.g., adding `onmouseclick` attributes to plot elements.

## 14. (X)HTML forms

HTML forms (Kennedy and Musciano 2006) provide built-in controls, such as a choice menu, check box, radio button, textbox, and button, that receive input from the user. The controls

may not be as rich as those available in Carto:Net, e.g., a slider is only available via JavaScript rather than a built-in HTML form element, however they are simple to deploy in an HTML page and make available to readers.

With forms, we can mix controls in HTML with a plot in SVG. We embed the SVG display in an HTML document using the HTML element `<object>` such as

```
<object id="PlotID" data="plot.svg"
        type="image/svg+xml" width="960" height="800"/>
```

which specifies the name of the SVG file via the `data` attribute, its MIME type, and dimensions. When we want to operate on pieces of the SVG document (e.g., when the viewer clicks on an HTML button), we use JavaScript to retrieve the resulting SVG object. Once we have the SVG object, we can operate on it with JavaScript, just as we have done when we placed the scripts in the SVG document. The JavaScript code below shows how to retrieve the embedded object and treat it as an SVG document.

```
doc = document.getElementById('PlotID');
doc = doc.getSVGDocument();
```

We can then deploy JavaScript callback functions that respond to viewer input via the controls in an HTML form where these functions interact with an SVG document embedded in the HTML document. For example, when a viewer selects an option from a choice menu, a JavaScript function can be called to modify an SVG display in the document. We illustrate this approach in the next example.

**Example 13.** Using forms to highlight points in a **lattice** plot.

This example re-implements Example 6. In that example, we have a legend on a **lattice** plot. The legend displays the levels of the variable `gear`, which is not contained in the display. The display shows `mpg`, `disp`, `am`, and `cyl` via points, color, or location in a panel. When the viewer mouses over an entry in the legend, the observations corresponding to this value of `gear` are highlighted in the different panels of the **lattice** plot. In contrast, the example here removes the legend from the plot, and controls the highlighting action through a choice menu in an HTML form. See Figure 16. That is, the SVG is embedded in an HTML document and this document also contains an HTML form with a choice menu for selecting a gear value.

The HTML document rendered in Figure 16 contains the SVG plot by including it using the following `<object>` tag:

```
<object id="latticePlot"  data="mt\_lattice\_Choice.svg"
        type="text/svg+xml" width="960" height="768" />
```

The SVG document is the same as the one created in the earlier example with two important differences. Firstly, it no longer contains any JavaScript code. Secondly, the legend has been removed from the **lattice** plot as the HTML form is replacing the mouse-over capability. However, all of the plotting elements still have their appropriate identifier attributes (i.e., the `id` attributes indicate to which group they belong). We show the code here:

```
R> mtcars$cyl <- factor(mtcars$cyl,
+    labels = c("four cyl", "six cyl", "eight cyl"))
```

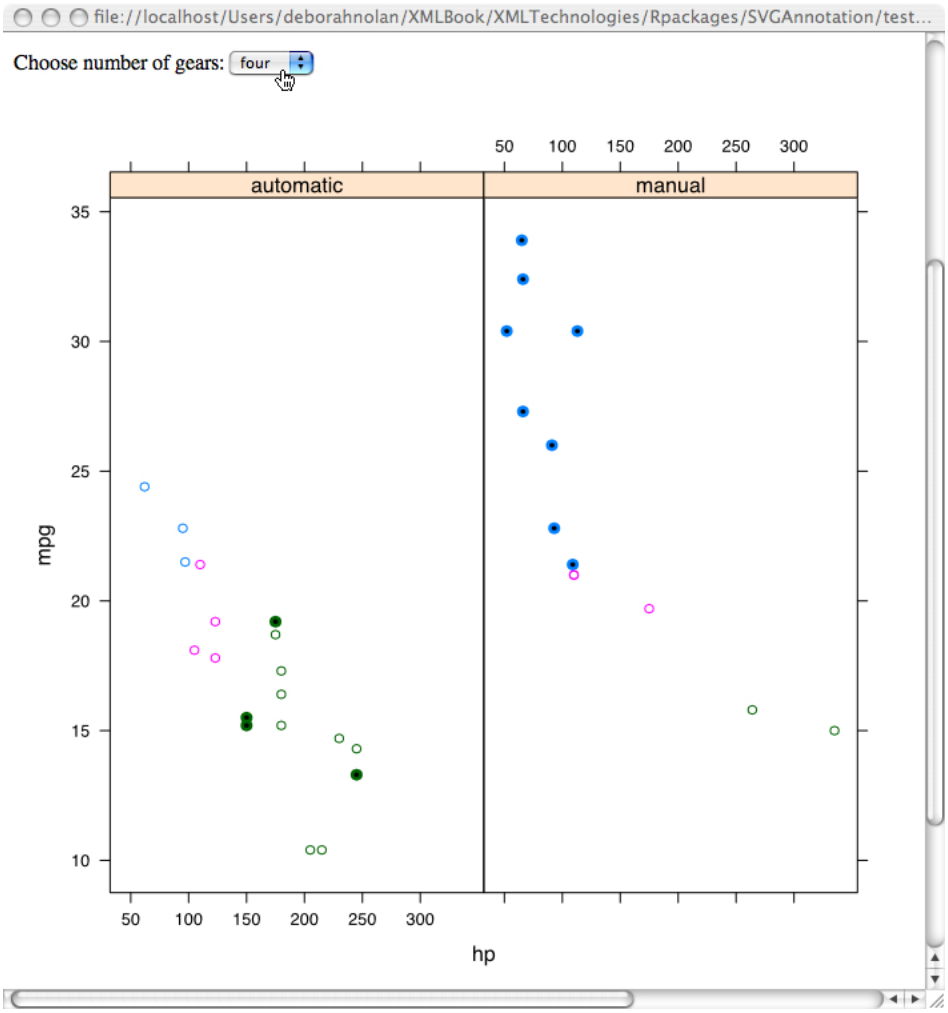


Figure 16: This HTML page contains a form and an SVG **lattice** plot. The SVG is controlled by the choice menu in the form. It offers the same functionality as the SVG document in Figure 11. However, when the viewer changes the choice in the HTML form, a JavaScript function in the HTML document is called to respond to this event, i.e., to highlight the appropriate set of points.

```
R> mtcars$am <- factor(mtcars$am, labels = c("automatic", "manual"))
R> doc <- svgPlot(xyplot(mpg ~ hp | am, groups = cyl, data = mtcars))
R> panels <- getPlotRegionNodes(doc)
R> points <- unlist(lapply(panels, xmlChildren), recursive = FALSE)
R> ids <- by(mtcars, list(mtcars$gear, mtcars$am), function(x)
+    paste(as.integer(x$am), x$gear - 2, 1:nrow(x), sep = "-"))
R> uids <- unlist(ids)
R> mapply(function(node, id) addAttributes(node, id = id), points, uids)
R> saveXML(doc, "mt_lattice_Choice.svg")
```

Along with the SVG display of the scatter plot, the HTML document also contains a `<form>` with a `<select>` menu that has four options as shown below.

```
<form>
Choose number of gears:
 <select name="gear" onchange="showChoice(this)">
   <option value="0" default="true">Select</option>
   <option value="1">three</option>
   <option value="2">four</option>
   <option value="3">five</option>
 </select>
</form>
```

Notice that when the selection is changed, the `showChoice()` function is called.

The `showChoice()` function is passed the JavaScript object corresponding to the choice menu, and the function uses this object to query the value the viewer has selected. The `showChoice()` function then calls `highlight()`, passing it the information it needs to highlight the new group of points. `showChoice()` is

```
var oldgroup = 0;
var group = 0;
var doc;

function showChoice(obj)
{
   doc = document.getElementById('latticePlot');
   doc = doc.getSVGDocument();
   group = Math.floor(obj.value);
   if (group > 0) {
    highlight(group, pointCounts[(group - 1)], true);
   }
   if (oldgroup > 0) {
    highlight(oldgroup, pointCounts[(oldgroup - 1)], false);
   }
   oldgroup = group;
}
```

The `highlight()` function called from `showChoice()` and its helper function `highlightPoint()` are identical to the functions by the same name that were placed in the `<script>` tag within the SVG document in Example 6. They extract the elements in the SVG document corresponding to the points that belong to the selected group, and change their style attributes. These functions rely on the plotting elements having unique `id`s that indicate to which group the points belong. Note that the embedded SVG document is in the global variable `doc`, which must be extracted from the HTML for `highlight()`.

```
function highlight(group, pointCts, status)
{
    var el;
    var i, numPanels = pointCts.length;
```

```
      /*  we want the group */
    for(panel = 1; panel <= numPanels; panel++) {
      for(i = 1; i <= pointCts[panel-1]; i++) {
        var id = panel + "-"+ group + "-" + i;
        el = doc.getElementById(id);
        if(el == null) {
          alert("can't find element " + id)
          return(1);
        }
        highlightPoint(el, status);
      }
    }
 }
```

`highlight()` and `highlightPoint()` now appear in the `<head>` of the HTML document rather than in the SVG. We can programmatically add them to the HTML document in much the same way we did for the SVG document using `addECMAScripts()`.

Note that the SVG document created for this example had no JavaScript in it and no calls to JavaScript functions. The interactive functionality relies on the plot elements in the SVG having unique identifiers. In general, provided the plot elements have appropriate ids, "plain" SVG can be made interactive via JavaScript embedded in HTML rather than in the SVG. The advantage of this approach is that plots made for some general purpose can be made to have interactive capabilities.

## 15. Adding interactivity to SVG embedded in HTML

There is yet another approach to providing interactivity for SVG documents that are embedded within HTML. The idea is that we use JavaScript code in the HTML document to programmatically modify elements in the SVG document at the time it is loaded in the browser. This approach can be useful when the plots have been produced in SVG but the creator has no motivation to add any interactivity. If the JavaScript code can identify the appropriate elements in the SVG document, either by contextual knowledge or using known `id` attribute values, it can add `onmouseover`, `onmouseout` and `onclick` event attributes to these elements and so dynamically (i.e., at viewing time) add the interactivity.

As an example of this approach, we consider the case where we place mouse capabilities on a map so that a mouse click triggers the display of an HTML table.

**Example 14.** Communication between SVG and an HTML page.

In this example, we create an HTML page that displays the canonical red-blue map of the United States, where the states are colored red or blue according to whether the majority of votes in the 2008 presidential election were Republican or Democrat, respectively. The viewer interacts with the map by clicking on a state in the map, and as a result, summary statistics for the selected state are loaded into the web page.

We layout the HTML page using nested `<div>` tags as follows

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

```
<center>
<h3>Presidential Election 2008</h3>

<p>
The map is interactive in that you can click on
anywhere within a state to see more detailed
results about the voting in that state.
</p>
</center>

<div id="main">
 <div id="summaryMap">
    <div id="stateSummary"></div>
    <div id="map">
       <object id="svgMap" data="stateMap.svg"
          type="image/svg+xml" width="700" height="700">
       </object>
    </div>
 </div>

 <div id="countySummary">
  <a id="toggleCounty"
     onclick="toggleCountyView()">+ Click to see county results</a>
  <div id="countySummaryContent" class="hidden">
     Click on a state to see more detailed information about the
     election results for that state.
  </div>
 </div>
</div>
```

Each of these `<div>`s has an associated style which we place in the `<head>`.

The SVG map is embedded in the `<div>` that has an `id` of "map". The plot was created with R using the **maps** package. We add `id` attributes to the polygon elements that correspond to the polygon name used in `map()`. (See Example 3 for an example of how to do this.) Although we place unique identifiers on each polygon, we do not add any JavaScript interactivity in the annotation stage of the plot creation. Recall that the first stage is when we create the graphical display in R; the second stage, the annotation stage, is when we use R to modify the resulting SVG document to add tags, attributes, and JavaScript to support interactivity; and the third stage is the viewing stage, when we are in the browser, R is no longer available, and JavaScript functions respond to events. In this example, we use JavaScript that is located in the HTML page to annotate the SVG with `onclick` attributes on each of the polygon elements. This annotation occurs at the time the page is loaded into the browser. We do this with the following JavaScript code:

```
<script type="text/javascript">
function loaded() {
    show("national");
```

```
    var doc;
    doc = document.getElementById('svgMap');
    doc = doc.getSVGDocument();

    for(var i = 0; i < polyIds.length ; i++) {
        var el = doc.getElementById(polyIds[i]);
        var txt = "parent.show('" + polyStates[i] + "')";
        el.setAttribute("onclick", txt);
    }
}
</script>
```

The mouse clicks occur on elements in the SVG document, and since the JavaScript to handle these events are in the HTML document, not the SVG, we need to call the `show()` method of the parent document using `parent.show`. Notice that the mouse click function call passes the state name, not the polygon name. The JavaScript variables `polyIds` and `polyStates` contain the unique names of the polygons in the map and the names of the states that the polygons belong to, respectively. (Recall that there are 63 polygons in the map because some states are drawn with multiple polygons, e.g., Manhattan in New York).

The `show()` function appears in a `<script>` tag in the `<head>` of the HTML file. We also place the JavaScript variables `polyIds` and `polyStates` there. `show()` is defined as

```
<script type="text/javascript">
function show(stateName)
{
  var val;
  var div;
  val = stateSummaryTables[stateName];
  div = document.getElementById("stateSummary");
  div.innerHTML = val;

  val = countyTables[stateName];
  if(val) {
    div = document.getElementById("countySummaryContent");
    div.innerHTML = val;
  }
}
</script>
```

When `show()` is called, it retrieves a reference to the embedded element in the HTML document named "stateSummary", which is where it will place the table of statistics for the selected state. We modify the contents of this `<div>` element, adding the new HTML table. In addition, the county level information is retrieved and placed in the `<div>` called "countySummaryContent". Finally, additional JavaScript variables are loaded into the document with

```
<script type="text/javascript" src="stateHTMLTables.js"></script>
```

These variables were created in R and contain the HTML table content for the states and counties information.

The main difference between the approach presented here and that found in the earlier examples is that some of the post-plotting annotations have been lifted into the viewing stage. One result of this approach is that the code that responds to user interaction will be more indirect as it is no longer within the SVG document. Also, the post-processing occurs outside of the R environment, at the time the document is loaded. Thus, we can take SVG documents not made for interactivity and modify them with JavaScript. A downside to this approach is that the interactivity may require access from the browser to the data and statistical routines used to generate the plot.

## 16. Related approaches

The **SVGAnnotation** package allows R users to leverage the sophisticated and flexible static graphics frameworks in R to create views of data and models and then display them in new and interesting ways in new emerging media. With **SVGAnnotation**, we can add interactivity and animation to the displays. The mechanism relies on post-processing the output from the R graphics engine and associating elements within the SVG output with the high-level components of the display. This is entirely deterministic based on the nature of the R plot, but is slightly different for each type of plot since there is no simple format or model for representing all plots. The purpose of **SVGAnnotation** is to find the SVG elements corresponding to the high-level graphical elements and allow the R user to easily augment these.

The package provides high-level facilities for "standard" plots in which we can readily identify the R elements. It also allows an R programmer to use intermediate-level facilities to operate on axes, legends, etc. There are also low-level facilities for identifying the shape of visual elements, e.g., polygon, line, vertical line, text. The functions in the package can also add high-level type identifiers to nodes in the SVG document such as identifying data points, axes, labels, titles, frames. These facilities allow us to handle cases from regular R graphics, **lattice** (Sarkar 2011), **ggplot2** (Wickham 2009) and **grid** (Murrell 2006). By leveraging the XML facilities in R, **SVGAnnotation** offers capabilities for creating rich new plots. Most importantly, the approach and the concept it implements is readily extended to other contexts and types of plots. The abstract idea and approach is the main contribution of the paper and package.

Our approach in the **SVGAnnotation** package is to use the high-quality rendering engine provided by libcairo from within R. There are two other SVG graphics devices available for use within R. These are found in the **RSvgDevice** (Luciani 2009) and the derived **RSVGTipsDevice** (Plate 2011) packages. The former generates an SVG document that contains SVG elements corresponding to the graphical primitives the R graphics engine emits, e.g., lines, rectangles, circles, text. However, the libcairo system is more widely used and more robust. It also deals with text in more advanced and sophisticated ways (specifically drawing letters as infinitely scalabe shapes/paths rather than using font sets). This is the primary reason we use the libcairo approach even though the format of the SVG documents that **RSvgDevice** produces is simpler and more direct.

The **RSVGTipsDevice** package builds on the code from **RSvgDevice**. It allows R programmers to add SVG annotations and does so when the SVG is being created, rather than via post-

processing additions. R users can set text for tooltips and URLs for hyperlinks that are applied to the next graphical primitive that the R graphics engine emits. This works well when the R programmer is creating directly in their own code all graphical elements of a display. However, it does not work when calling existing plotting functions that create many graphical elements. The computational model does not provide the appropriate resolution for associating an annotation with a particular element, but just the next one that will be created. As a result, we cannot annotate just the vertical axis label when calling a function that creates an entire plot.

Our post-processing approach is not limited to using libcairo. We could also use **RSVGTipsDevice** to generate the SVG content and then programmatically manipulate that. The documents generated by the two different devices will be somewhat different (e.g., the use of groups of characters for strings, in-line CSS styles versus separate classes, SVG primitives for circles rather than paths). However, because the elements of the graphical display were generated by the R graphics engine with the same R expressions, the order of the primitive elements and the general structure will be very similar.

The package **gridSVG** (Murrell 2011) is an earlier and quite different approach to taking advantage of SVG. As the name suggests, it is focused on R's grid graphics system. For this reason, it cannot work with displays created with the traditional and original graphics model, but it does handle anything based on grid such as all of the plot types in **lattice** and **ggplot2**. The package **gridSVG** takes advantage of the structured self-describing information contained in grid's graphical objects. As one creates the grid display, the information about the locations and shapes are stored as objects in R. These can then be translated to SVG without using the R graphics device system. New graphics primitives have been added to the grid system to add hyperlinks, animations, etc. corresponding to the facilities provided in SVG.

The approach provided by **gridSVG** is quite rich. It removes the need to post-process the graphics that we presented here. Instead, one proactively and explicitly specifies graphical concepts. One limitation that is similar to that of **RSVGTipsDevice** is that we still run into problems with interleaving SVG annotations. While a user can issue grid commands that add SVG facilities to the output, higher-level functions that create plots produce entire sequences/hierarchies of grid objects in a single operation. If these do not add the desired annotations, we have to post-process the grid hierarchy to add them. This post-processing would be very similar to what we are doing in **SVGAnnotation**, however it would be done on R objects.

Of course, **gridSVG** is restricted to the grid graphics system, and higher-level graphics facilities based on grid such as **ggplot2** and **lattice**. Our approach however works with any R graphical output, although it needs to be "trained" to understand the content. The combination of the two approaches: **gridSVG** and **SVGAnnotation** appear to give a great deal of flexibility that allows both pre-processing and post-processing. If all graphics were grid-based, **gridSVG** may well be the most appropriate approach. Since many widely used graphics functionality in R are based on the traditional graphics system, **SVGAnnotation** is necessary.

The **animation** package (Xie 2011) provides functionality to create movies (e.g., animated GIF files or MPEG movies) entirely within R, using some additional external software. The idea is that one draws a sequence of plots in R. These are combined and then displayed at regular intervals as frames of the movie to give the appearance of motion. R users can draw each frame

using R code and without regard for particular graphics device or formats. This simplifies the process for many users. The approach does, however, involve redrawing each frame rather than animating individual objects. It also provides no interactive facilities within the plot. It involves a different programming model where we redraw entire displays rather than working on individual graphical elements. In different circumstances, each model has advantages. So while animated displays are common to both the **animation** and **SVGAnnotation** packages, the goals and infrastructure are very different. Being able to work at the level of graphical objects within the plot is richer and more efficient for many applications.

Finally, the **imagemap** package (Rowlingson 2004) offers another approach to creating interactivity within R graphical displays. The basic concept is that a plot is created in R in the usual manner. Then, the creator specifies different regions within the plot that correspond to areas of interest to viewers, e.g., the axes labels, points in a scatter plot, bars in a histogram. These regions are specified using the coordinate system of the data. These regions define an image map that can be displayed within an HTML document. The creator specifies actions in the form of JavaScript code that are triggered as the viewer moves the mouse over the different regions or clicks within a region. This approach does not map the elements of the plot, such as a title, hexagonal bin, or state polygon, to mouse events. It instead creates an overlay for the image from separately identified regions. In addition, the elements in the image cannot be programmatically changed at viewing time. For example, we cannot change the color of a line, or the size of a point or rectangle in response to a viewer's actions. We also cannot move elements to achieve animation effects.

## 17. Future directions

The motivation behind the **SVGAnnotation** package is the ability to exploit and harness new media such as interactive Web pages. The aim is to enable statisticians to readily create new graphical displays of data and models using existing tools that can be displayed in rich, interactive and animated venues such as Web browsers, Google Earth, Google Maps, GIS applications. The **SVGAnnotation** package focuses on facilitating R users in leveraging existing R graphical functionality by post-processing the output to make it interactive and/or animated. It is a complete computational model in that it enables the author of a display (or a third-party) to modify all elements in that display. This approach can be used for other modern formats.

Another format for graphical displays and applications is Adobe's Flash & Flex (`http://www.adobe.com/products/flash`). This is an alternative to SVG that is a very widely used framework (the ActionScript programming language, collection of run-time libraries, and compiler suite) for creating interactive, animated general interfaces and displays. The range of applications include rich graphical user interfaces and interactive, animated business charts. Flash and Flex are freely available, but not open source. The format and tools are mostly controlled by Adobe. The Adobe chart libraries for Flash are only available in commercial distributions.

There are publicly available open source libraries for creating certain types of common plots, e.g., flare (`http://flare.prefuse.org/`). Alternatively, we can use R to create statistical graphical displays by generating ActionScript code to render the different elements in the display. We have developed a prototype R graphics device (in the **FlashMXML** package) that creates Flash plots within R using the regular graphics framework. We can then post-process

the generated content in much the same way we do with the **SVGAnnotation** package in order to provide interaction and animation, connect the plot to GUI components, etc.,

Rather than considering Flash and JavaScript as competitors to SVG, we think that each has its own strengths. SVG is more straightforward and direct than Flash and JavaScript for creating graphical displays. For one, we have an excellent R graphics device that creates the initial SVG content. We can add GUI components, but Flash is better for this as it provides a much richer collection of GUI components such as a data grid. Drawing displays with JavaScript avoids depending on the presence of support for either SVG or Flash and so can be deployed in more environments.

Another approach is to use the HTML5 canvas element that has been recently introduced in several browsers. We can create JavaScript objects for drawing, e.g., circles, lines, text on a canvas. Again, we have developed a prototype R graphics device that generates such code for an R plot. We can also annotate this to add animation and interaction. We also mention the Vector Markup Language (VML, http://msdn.microsoft.com/en-us/library/bb263898(VS.85).aspx) which is quite similar to SVG. It is used within Microsoft products such as Word, Excel and PowerPoint. However, it is not widely supported by other applications.

A fundamental aspect of what we have described with respect to the **SVGAnnotation** package is that R is used to create the display but is not available at viewing time when the SVG document is rendered. If R were available for the SVG viewer, then the JavaScript/ECMAScript code within an SVG or HTML document could make use of R at run-time. It could invoke R functions and access data to update the display in response to user interaction and animation. Additionally, we could use R code to manipulate the SVG and HTML content and so enable programming in both ECMAScript and R. One approach is to have R available on the server side of a client-server setup via, e.g., **RApache**. Alternatively, R could be plugged into the HTML viewer and perform R computations on the client side. We are developing such an extension for Firefox which embeds R within a user's browser. This modernizes previous work in 2000 on the **SNetscape** package that embedded R and R graphics devices within the Netscape browser. We believe the combination of SVG (or Flash or the HTML5 canvas) with R at viewing-time will prove to be a very rich and flexible environment for creating new types of dynamic, interactive and animated graphics and also allow Google Earth and Google Maps displays to be combined with R plots and computations.

In conclusion, the increasing importance of Web based presentations is a space where statisticians need to be engaged. To accomplish this, we need more tools for creating these presentations. **SVGAnnotation** offers one approach; we are working on others. We ask that the reader think of the package as a rich starting point that enables a new mode of displaying R graphics in an interactive, dynamic manner on the Web. It establishes a foundation on which we and others can build even higher-level facilities for annotating SVG content and providing rich plots in several different media (i.e., SVG, HTML, JavaScript).

## Acknowledgments

We thank the referees for their detailed comments, which led to an improved version of this article. We also thank Gabe Becker for assistance with the examples. This material is based in part upon work supported by the National Science Foundation under Grant Number DUE-0618865.

## References


Adobe Systems Incorporated (2011). "Flash Player Software, Version 10.3." URL http://get.adobe.com/flashplayer/.

Apache Software Foundation (2008). "**batik**: Java SVG Toolkit, Version 1.7." URL http://xmlgraphics.apache.org/batik/.

Apache Software Foundation (2009). "**Squiggle**, the SVG Browswer." URL http://xmlgraphics.apache.org/batik/tools/browser.html.

Bah T (2007). ***Inkscape**: Guide to a Vector Drawing Program*. Prentice Hall Press, Upper Saddle River.

Becker R, Wilks A, Brownrigg R, Minka T (2011). ***maps**: Draw Geographical Maps*. R package version 2.1, URL http://CRAN.R-project.org/package=maps.

Berger A, Pucher A, Medwedeff A, Neumann A, Winter A, Furpass C, Resch C, Chuffart F, Jurgeit F, Held G, Sepesi G, Fibinger I, Forster K, Galanda M, Rogers N, Ueberschar N, Sykora P, Reddy Maddirala SK, Mailander T, Voswinckel T, Bruehlmeier T, Ullrich T, Barth Y (2010). "Carto:Net Software." URL http://www.carto.net/.

Cairo Graphics (2010). "Cairo Software, Version 1.1." URL http://www.cairographics.org/.

Carr D, Lewin-Koh N, Maechler M (2009). ***hexbin**: Hexagonal Binning Routines*. R package version 1.22, URL http://www.bioconductor.org/packages/release/bioc/html/hexbin.html.

Eisenberg J (2002). *SVG Essentials*. O'Reilly Media, Sebastopol.

Flanagan D (2006). *JavaScript: The Definitive Guide*. O'Reilly Media, Sebastopol.

Gentry J, Gentleman R, Huber W (2010). "How to Plot a Graph Using **Rgraphviz**." URL http://www.bioconductor.org/packages/release/bioc/vignettes/Rgraphviz/inst/doc/Rgraphviz.pdf.

Gentry J, Long L, Gentleman R, Falcon S, Hahne F, Sarkar D, Hansen K (2011). ***Rgraphviz**: Provides Plotting Capabilities for R Graph Objects*. R package version 1.30, URL http://www.bioconductor.org/packages/release/bioc/html/Rgraphviz.html.

Harold ER, Means WS (2004). *XML in a Nutshell*. O'Reilly Media, Sebastopol.

Kazoun C, Lott J (2008). *Programming Flex 3: The Comprehensive Guide to Creating Rich Internet Applications with Adobe Flex*. O'Reilly Media, Sebastopol.

Kennedy B, Musciano C (2006). *HTML and XHTML: The Definitive Guide*. O'Reilly Media, Sebastopol.

Luciani J (2009). ***RSvgDevice**: An R SVG Graphics Device*. R package version 0.6.4.1, URL http://CRAN.R-project.org/package=RSvgDevice.

Meyer E (2004). *CSS Pocket Reference*. O'Reilly Media, Sebastopol.

Murrell P (2006). ***gridBase**: Integration of **base** and **grid** Graphics*. R package version 0.4-3, URL http://CRAN.R-project.org/package=gridBase.

Murrell P (2011). ***gridSVG**: Export **grid** Graphics as SVG*. R package version 0.8-1, URL http://CRAN.R-project.org/package=gridSVG.

Nolan D, Temple Lang D (2011). ***SVGAnnotation**: Tools for Post-Processing SVG Plots Created in R*. R package version 0.9, URL http://www.omegahat.org/SVGAnnotation.

Opera Software ASA (2011). "Opera Software, Version 11.51." URL http://www.opera.com/browser/.

Pawson D (2002). *XSL FO: Making XML Look Good in Print*. O'Reilly Media, Sebastopol.

Plate T (2011). ***RSVGTipsDevice**: An R SVG graphics device with dynamic tips and hyperlink*. R package version 1.0-4, URL http://CRAN.R-project.org/package=RSVGTipsDevice.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Rosling H (2008). "Gapminder: World." URL http://www.gapminder.org/world.

Rowlingson B (2004). ***imagemap**: Create HTML Imagemaps*. R package version 0.9, URL http://www.maths.lancs.ac.uk/Software/Imagemap/.

Sarkar D (2008). ***lattice**: Multivariate Data Visualization with R*. Springer-Verlag, New York.

Sarkar D (2011). ***lattice**: Lattice Graphics*. R package version 0.19, URL http://CRAN.R-project.org/package=lattice.

Simpson J (2002). *XPath and XPointer: Locating Content in XML Documents*. O'Reilly Media, Sebastopol.

Swayne D, Cook D, Temple Lang D, Buja A (2010). "**GGobi** Software, Version 2.1." URL http://www.ggobi.org/.

Temple Lang D (2011a). ***RJSONIO**: Serialize R Objects to JSON, JavaScript Object Notation*. R package version 0.95, URL http://www.omegahat.org/RJSONIO.

Temple Lang D (2011b). ***XML**: Tools for Parsing and Generating XML within R and S-PLUS*. R package version 3.4, URL http://www.omegahat.org/RSXML.

Theus M (2002). "Interactive Data Visualization Using **Mondrian**." *Journal of Statistical Software*, **7**(11), 1–9. URL http://www.jstatsoft.org/v07/i11/.

Urbanek S, Wichtrey T (2011). ***iplots**: Interactive Graphics for R*. R package version 1.1-4, URL http://CRAN.R-project.org/package=iplots.

van Vugt W (2007). "Open XML: The Markup Explained." URL http://openxmldeveloper.org/attachment/1970.ashx.

Wickham H (2009). ***ggplot2**: Elegant Graphics for Data Analysis*. Springer-Verlag, New York.

Xie Y (2011). ***animation**: A Gallery of Animations in Statistics and Utilities to Create Animations*. R package version 2.0-6, URL http://CRAN.R-project.org/package=animation.

Yahoo! Inc (2011). "**YUI** Library." URL http://developer.yahoo.com/yui/.

# A. Functions in SVGAnnotation

There are approximately 60 functions available in **SVGAnnotation** for annotating plots. Among these are high-level functions that add interactivity through a single function call. These functions are listed in Table 3. Other intermediate level functions are described in Table 4. These roughly fall into two types of functions: those that find nodes in the SVG that correspond to particular parts of a plot e.g., points, legend, panel; and those that annotate or add a node. The functions that locate the nodes begin with the prefix "get" and those that annotate a node or add a node begin with "add".

We recommend creating the SVG plot with the function `svgPlot()` in **SVGAnnotation**. It opens the SVG device, evaluates the commands to generate the plot, and closes the device. It also adds the R plotting commands to the SVG document and can either return the parsed XML document as a tree-like structure or write it to a file.

There are many more "get" functions available in **SVGAnnotation**. These can make it easier for the developer to build other high-level functions for annotation. Here is a list of all the get functions currently in the package:

```
R> objects(2)[substr(objects(2), 1, 3) == "get"]
```

```
 [1] "getAxesLabelNodes"          "getAxesLabelNodes.mosaic"
 [3] "getBoundingBox"             "getCategoryLabelNodes.mosaic"
 [5] "getCSS"                     "getECMAScript"
 [7] "getEdgeElements"            "getEdgeInfo"
 [9] "getGeneralPath"             "getJavaScript"
[11] "getLatticeLegendNodes"      "getLatticeObject"
[13] "getMatplotSeries"           "getNodeElements"
[15] "getPanelCoordinates"        "getPanelDataNodes"
[17] "getPlotPoints"              "getPlotRegion"
[19] "getPlotRegionNodes"         "getRCommand"
[21] "getRect"                    "getShape"
[23] "getStripNodes"              "getStyle"
[25] "getSVGNodeTypes"            "getTextPoints"
[27] "getTopContainer"            "getTopG"
[29] "getUSR"                     "getViewBox"
```

# B. Basics of XML

The basic unit in an XML document is an *element*, also known as a node or chunk. An element can contain textual content and/or additional XML elements. By *content*, we mean the simple text, such as the word "Oregon" that appears in the simple SVG document shown in Figure 9. An element begins with a *start-tag*, which has the format `<tagname>`, and the element ends with `</tagname>`. The *end-tag*, where the tag name matches the start tag's name. Those readers who have read or composed HTML (HyperText Markup Language) will recognize this format. For more in-depth information about XML see Harold and Means (2004). SVG, like HTML or WordProcessingML, is an example of a specific grammar of XML.

| Function | Brief Description |
|---|---|
| `addAxesLinks()` | Associates target URLs with axes labels and plot titles so that a mouse click on an axis label or title jumps to the specified URL in the viewer's Web browser. |
| `addToolTips()` | Associates text with elements of the SVG document so that the text can be viewed in a tool tip when the mouse is placed over that object/element in the display. The default action of this function is to add tool tips to points in a scatter plot. |
| `linkPlots()` | Implements a simple linking mechanism of points within an R plot consisting of multiple sub-panels/plots. When the mouse is moved over a point in a sub-plot, the color of all corresponding points in other plots/panels also changes. The sub-plots can be created by, for example, arranging plots via the `mfrow` parameter in `par()`, or via the functions `pairs()` or `splom()` in the **lattice** package. |
| `animate()` | Creates an animated scatter plot where points in the scatter plot move at different points in time, possibly changing size and color as they move. |

Table 3: High-level functions for adding interactivity to SVG plots.

With SVG, the focus is on describing graphical displays. The SVG document rendered in Figure 9 includes element/tag names such as `<circle>` and `<rect>` for drawing circles and rectangles respectively. Notice that some of the elements in the sample SVG document are nested within other start and end tags. This hierarchical structure gives us great flexibility in describing complex, nested data with arbitrary depth.

For more details about the specific tags and how to create SVG graphics, see Section 6.

An XML element can have *attributes* associated with it. These are supplied in the tag itself as name-value pairs as follows: `<tagname attributeName="value">`. For example, the `<text>` tag,

```
<text x = "110" y = "200" fill = "navy" font-size = "15">
       Oregon
</text>
```

has an attribute, `x`, which specifies the horizontal position of the text on the SVG canvas. The value for `x` in this example is "110", indicating position 110 along the x-axis of the canvas (which in our example is 300 by 300 points). The syntax rules for elements and their tags are provided below in Section C.

We should note one important short cut for start and end tags. If an XML element contains no text content or elements within it, i.e., there are no text or sub-elements contained between the start and end tags of the element, then it can be written as `<tagname/>` rather than the longer `<tagname></tagname>`. For example, notice that the `<rect>` element:

```
<rect x = "10" y = "20" width = "50" height = "100" class = "recs"/>
```

is empty. All of the relevant information for drawing the rectangle is contained in the attributes, i.e., its width, height, location on the canvas, and color. Hence the end tag is omitted and the start tag also acts as a closing tag.

| Function | Brief description |
|---|---|
| `addLink()` | Associates target URLs with any SVG element so that a mouse click on an axis label, title, point or some other plot component displays the specified URL in the viewer's Web browser. Set the `addArea` parameter to `TRUE` if you want the link to be associated with a bounding rectangle around the SVG element and not just the pixels along the path of the SVG element. |
| `addECMAScripts()` and `addCSS()` | Add JavaScript/ECMAScript and CSS code to an SVG document. These function either directly insert the content or put a reference (`href` attribute) to the file/URL in the document. |
| `addSlider()` | Add an interactive slider to the SVG document. Uses the Carto.Net GUIlibrary. |
| `radioShowHide()` | Add radio buttons to the SVG document. Uses the Carto.Net GUI library. |
| `getPlotPoints()` | Examines the SVG document and returns the SVG elements/nodes which represent the "points" within the R graphic. These may be in multiple plots or panels, and they may be, e.g., the polygonal regions in a map or hexagons in a hexbin plot. |
| `getAxesLabelNodes()` | Examines the SVG document and returns the SVG elements/nodes which represent the text of the main title and the X and Y axes for each sub-plot within the R graphic. |
| `getPlotRegion()` | Retrieve the SVG elements which correspond to the plotting regions, i.e., the high-level frames or panels, within the display. This returns multiple elements as appropriate for traditional and **lattice** plots. |
| `getPlotRegionNodes()` | Retrieve the SVG elements which house the contents of the data regions of the sub-plots within the display. This works for traditional and **lattice** plots, and also for histograms and bar plots. |
| `getStyle()`, `setStyle()`, `modifyStyle()` | These functions query, set, and reorganize the value of a style attribute of an SVG node, respectively. Use these functions to determine and set the vector of CSS-based style settings. |
| `getLatticeLegendNodes()` | Retrieve the legend in a **lattice** plot. |
| `enlargeSVGViewBox()` | Change the dimensions of the viewing box so that extra elements can be added to the document and displayed within its viewing area, e.g., a slider below a plot or check boxes alongside a plot. |
| `convertCSSStylesToSVG()` | Converts a CSS `style` attribute into presentation attributes. The purpose is to make it easier to change a single attribute in response to a user event. See Section 6 for a description of these types of style specifications. |
| `asTextNode()` | Replaces a `<g>` element (that contains directions for drawing text with `<path>` tags) with a `<text>` node. This makes it easier to modify text in a display. Although it does not have the benefit of scalability and the variety of fonts, it is much simpler for creating interactivity. (See Section 6 for more details on the text created by R's graphics system(s) and libcairo). |

Table 4: Intermediate-level functions for working with SVG elements.

# C. Well-formed XML

For XML to be properly processed it must obey some basic syntax rules, and we say the document is *well-formed* when it satisfies these rules. The following list is a subset of the most important syntax rules. This list covers the vast majority of XML documents. Notice that they are very general and do not pertain to a specific grammar of XML. XML documents that are not well-formed typically produce fatal errors when processed.

- An XML document must have a single *root* element that completely contains all other elements. In our example, `<svg>` is the root element.
- XML tags are case sensitive so start and end tag names must match exactly. For example,

  ```
  <text>
  Oregon
  </Text>
  ```

  is not well-formed because the start-tag begins with a lower-case "t" that does not match the capital "T" in the end-tag.
- All start tags must have a closing tag, unless the tag is empty and so can be contracted to a single tag of the form `<tagname />`.
- Elements must nest properly. That is, when one element contains another element then both the start and end tags of the inner element must be between the start and end tags of the outer element. For example, the following XML content is three nodes deep:

  ```
  <defs>
    <g id="circles">
       <circle id = "pinkcirc" cx= "50" cy="50"  r = "15"
           fill = "pink"/>
   </g>
  </defs>
  ```

  Note the use of indentation is optional, but it makes it easier to see the nesting of elements. (The white space is part of the XML document and is typically preserved during processing.)
- All attribute values must appear in quotes in a `name="value"` format in the start tag. For example the following `<g>` element has a value of main for the identifier attribute `id`: `<g id = "main">` It is common to omit the quotes around attributes within HTML documents, but this is an error for XML.

## C.1. Tree structure

The conceptual model of the XML document as a tree can be very helpful when processing and navigating it. The SVG shown in Section 6 uses indentation to make the nesting of elements clear, and demonstrates that the elements are logically structured into a hierarchical tree. Each element can be thought of as a node in the tree where branches emanate from the node

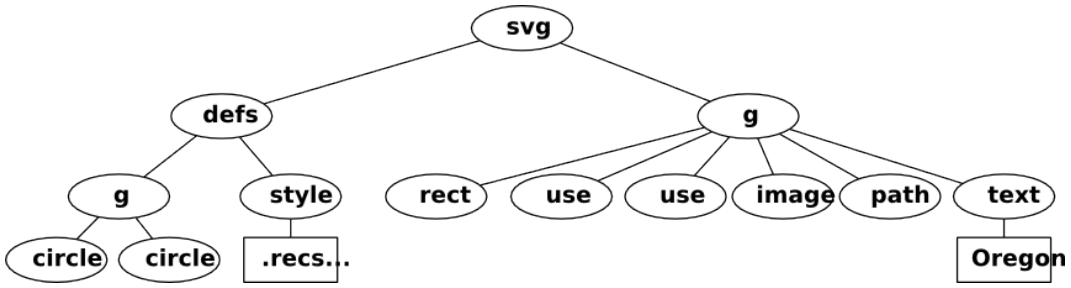


Figure 17: This tree provides a conceptual model for the organization of the SVG document rendered in Figure 9. Each element is represented by a node in the tree and the branches from one node to another show the connections between the nodes that are one layer apart from each other in the nesting of nodes. For example, there are two `<circle>` nodes and both are nested directly as children of (i.e., within) the `<g>` node; the branch between these nodes indicates this connection. That particular `<g>` node is a child of the `<defs>` element. Text content nodes are actually elements but without a tag name and are included in the tree. They are displayed via rectangles rather than ovals to distinguish them from regular tags, e.g., "Oregon" is a text node in this tree.

to those elements that it immediately contains. Figure 17 shows the tree representation of this SVG document.

The *root* of the tree is also referred to as the *document node*, which in this case is the `<svg>` element. As mentioned earlier, there is only one root node per document. Notice that the root node of this tree has two children, a `<defs>` node and a `<g>` node. The `<g>` has six children - a `<rect>` node, two `<use>` nodes, and one each of `<image>`, `<path>` and `<text>`.

The relative position of these nodes in the tree are described using family tree terminology. For example, the `<defs>` element is the *parent* of a `<g>` node, and `<circle>` is the *child* of `<g>`. The `<rect>` element is a *sibling* to `<image>`, and `<defs>` and `<svg>` are both *ancestors* of each of the `<circle>` nodes. Also, note that `<path>` can be referred to as a "following sibling" to `<image>` because it comes after (to the right in the image and below in the actual document) of `<image>`.

The character content of an element is placed in a "text" node. That is, text content is also represented as a node in the tree. In our example, the text Oregon is a child node of `<text>`. The *terminal* nodes in a tree are those that have no children and are known as *leaf nodes*. By design, text content will always be in a leaf node.

## C.2. Additional markup

In addition to elements, XML markup includes the XML declaration, processing instructions, comments, and CDATA section delimiters.

**Declaration** An XML document must start with the XML declaration that identifies it as an XML document and provides the XML version number,

```
<?xml version="1.0" encoding="UTF-8"?>
```

The declaration appears outside of the root element.

**Processing instructions** Similar to the XML declaration, processing instructions must begin with <? and end with ?>. Immediately following the <? is the target of the instruction, i.e., the agent/application for which the instruction is intended. The following processing instruction is for an xml-stylesheet, which is a standard name that means the parameters in this processing instruction are intended for an XML viewer, e.g., a Web browser or XML editor, that can apply a style sheet to the document.

```
<?xml-stylesheet type="text/css"
   href= '~/Rpackages/SVGAnnotation/CSS/RSVGPlot.css'?>
```

Notice that the processing instructions for the xml-stylesheet are provided via name-value pairs in a format that imitates the syntax for attributes, e.g., `type="text/css"`. This format is optional, and other applications could expect the processing instructions to be in some other format. Different applications support different processing instructions. Most applications simply ignore any processing instruction whose target they do not recognize.

(Technically, the XML declaration is not a processing instruction). The XML-Stylesheet processing instructions are always placed in the document prolog between the XML declaration and the root element start tag. Other processing instructions may also be placed in the prolog or at almost any other location in the XML document, i.e., within sub-nodes.

**Comments** A comment must appear between <!-- and -->, and it can contain < and > because all text between these two delimiters is ignored by the XML processor and so is neither rendered nor read. For example,

```
<!-- This is a comment which is so long that
it appears on three lines of
the document before it ends with -- followed by >.
-->
```

Comments can appear anywhere after the XML declaration, e.g., they can appear outside the root element of the document.

**Entities** Because XML uses the < character to start or end an XML element, we need an alternative way to include the literal character < as text within an XML node. We use XML entities for this. An entity is a named symbol within XML that corresponds to some fixed content. We refer to them by prefixing the name of the entity with & and adding the suffix ;. There are numerous built-in entities to which we can refer, e.g., the < character can be written as < and > is expressed as > and & as &.

**CDATA** Entities are convenient for escaping content from XML processing, especially individual characters. There are occasions, however, where we have a significant amount of content that we want to escape from XML processing and to be treated as literal or verbatim content. For example, when we insert the R code used to generate an SVG plot into the SVG document, we want to avoid having to worry about characters an XML processor will interpret. We do this by enclosing the content between between <![CDATA[ and ]]> markers. For example,

```
<![CDATA[
 plot(y ~ x, myData[ (myData$var1 < 10 | myData$var1 > 20) \&
                        myData$var2 %in% c("A", "B", "\&")])
 ]]>
```

This is then treated as a verbatim block of characters and not parsed by the XML processor for XML elements.

## D. Basics of JavaScript

We provide here a brief introduction to JavaScript for programmers who are familiar with the S language. JavaScript (the official name is ECMAScript) is an interpreted scripting language that is widely used within HTML documents and also within SVG displays and Flash applications (using a slight variant named ActionScript). JavaScript can be used within HTML to respond to user actions on buttons, menus, checkboxes, etc. in HTML forms, or to dynamically validate the content of a form before submitting it to a remote server. JavaScript can also be used to dynamically and programmatically construct the content of an HTML document. Within SVG, we use JavaScript to respond to user events on elements of the display (e.g., circles, lines, rectangles, ...) and to provide animation. Similar to HTML, JavaScript can be used to dynamically create SVG elements within the display.

As with R, one does not need to compile JavaScript code since it is interpreted. However, JavaScript is more like C++, Python and Java in its computational model. Much of the use of the language will focus on classes and instances (objects) of these classes. We frequently invoke an object's methods rather than call top-level functions, although, like R, we can have regular functions unattached to objects. JavaScript is not a vectorized language like R, i.e., it does not operate on vectors element-wise unless explicitly programmed to do so using loop constructs. JavaScript requires variables to be declared within the scope in which they are used. Unlike C++ or Java, one does not need to declare the type of a variable and a variable can take on values of different types during its lifetime.

To make use of JavaScript within an HTML or SVG document, we must connect that code with the document. We can do this by either inserting the code as content in the document or alternatively adding a reference to the file containing the JavaScript code. That code file may be located locally or on a remote server, subject to certain security restrictions. Both approaches use the `<script>` node within the HTML or SVG document. We can insert the code content between the start and end tag of the `<script>` node. Alternatively, we can use an attribute in the `<script>` element to refer to the JavaScript file. For SVG, we use an attribute named `href` and for HTML, we use `src`. The value in both cases is a local file name or a URL. The following illustrates the mechanism for SVG:

```
<script xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns="http://www.w3.org/2000/svg"
        type="text/ecmascript"
        xlink:href="SVGAnnotation/tests/multiLegend.js"/>
```

For HTML, we refer to a JavaScript file as

```
<script type="text/javascript" src="alert.js"/>
```

To in-line the JavaScript content, we use

```
<script type="text/ecmascript">

var neighbors = [[0,  1, 31, 20,  3, 29], [2, 8, 9, 24]];
var k = 4;

function showNeighbors(evt, k, neighbors)
{
    var idx = 1 * evt.target.getAttribute('id');
    window.status = "Showing " + idx;
    addLines(evt.target, neighbors[idx], k);
}
</script>
```

Note that we have to specify the `type` attribute but can use either JavaScript or ECMAScript in most HTML browsers and SVG viewers.

Often, JavaScript code that defines functions that will be used in event handler attributes on HTML or SVG elements are placed in a `script` element near the top of the HTML or SVG document. For HTML, these function definitions and global variables often appear in the `<head>` element. JavaScript code can also appear within the `<body>` of an HTML document and is evaluated when it is processed and so can access previously created elements in the document.

We provide here a brief summary of many of the basic syntax features of the language. For more detailed information about JavaScript see Flanagan (2006)

- Executable statements typically end with a semicolon. Although it is not strictly required, it is good practice to use the semicolon.

- Like R, curly braces group statements together into executable blocks and are used for defining the body of a function or a multi-expression if, while or for clause.

- Variables are declared within a specific scope via the `var` prefix, e.g.,

  ```
  var global = 1;
  var debug = false;
  function foo(N) {
   var i;
   for(i = 0; i < N; i++) {
      ...
   }
  }
  ```

  The declaration can assign an initial value to the variable as in the first two expressions immediately above. These are global variables; the variable `i` is local to the function calls for `foo()`. Note that the values for boolean variables are true and false, not `TRUE` and `FALSE` as in R.

- JavaScript supports arrays, including multi-dimensional arrays which can be "ragged", i.e., the sub-arrays can have different length/dimensions. In the JavaScript code above, the variable `neighbors` contains two arrays, one of length 6 and the other of length 4. JavaScript uses 0-based indexing for arrays. For example, in the two-dimensional array `neighbors`, `neighbors[0]` returns the first element of the top-level array and this is itself an array of six integer values. The expression `neighbors[1][0]` returns the first element in the second array, and the value is the scalar 2.

- Multi-line comments are delimited by /* and */ and single line comments begin with //, e.g.,

```
/*  This is a multi-line
    comment.
*/

// This is a comment
var debug = false; // display debug info with alert().
```

- The equal sign (=) is the assignment operator. In addition, `x += y` is equivalent to the assignment `x = x + y`, and `-=`, `*=`, and `/=` are similarly defined. Also, the operator `++` increments the referenced variable in-place by 1 and `--` decrements by 1, i.e., `++x` is equivalent to `x = x + 1`,

- Adding two character variables, pastes the two strings together. If a number and string are "added", the result will be a string, e.g., `"K is " + k` results in the character string "K is 4".

- Comparison operators are the same as in R, e.g.,

```
x == 1
abc < 10
str != "this is a string"
```

  The logical operators are `\&\&`, `||`, and `!`; they correspond to and, or, and not, respectively.

- The control flow syntax is similar to that in R:

```
if (x < 2) {
  ...
} else if( x > 2 && x < 10) {
   ...
} else
   ...
```

  JavaScript provides an if-else construct for conditional evaluation. We cannot assign the value of an if-else statement to a variable as we can in R.

```
 x = if(y > 2) 3 else 10;
```

  is a syntax error and will terminate the processing of the JavaScript code. JavaScript does provide the ternary operator for these simple situations:

```
varname = (condition) ? value1 : value2
```

- JavaScript supports `for` and `do...while` loops. In the code below, the JavaScript within the curly braces will be executed `N + 1` times, as `i` takes on the values 0, 1, ..., `N`.

```
for(var i = 0; i <= N ; i++) {
    var target;
    target = document.getElementById(neighbors[i]);
    ids = ids + " " + neighbors[i];
...
}
```

  Again, note that JavaScript is not vectorized, as R is.

- JavaScript code can be used in-line within `<script>` elements within a document or as the value of attributes of HTML or SVG elements, e.g., event handlers such as `onclick`, `onmouseover`, `onmouseout`. The JavaScript code is one or more JavaScript expressions separated by ';' or new lines.

  Typically these expressions call JavaScript functions. This is especially true of event handler code that invokes a function with specific arguments to respond to the event in a particular way. For example, the SVG attribute `onmouseover = "showNeighbors( evt, k, neighbors)"` results in a call to the JavaScript function `showNeighbors()` when the mouse moves over the corresponding element in the SVG display (see Example 7 for more details). The call passes three arguments: the event object and the global variables `k` and `neighbors`.

  Unlike R, JavaScript functions are not defined and assigned to a variable. Instead, functions are defined using the keyword function as a prefix to the definition as in

```
function functionName(var1, var2, ..., varN)
{
   body-code
}
```

  Like R, the parameters correspond to local variables within each call to the function. Since (non-primitive) variables in JavaScript are passed by reference, changes to these objects are made to the objects in the calling frame and are accessible after the specific function call is completed, i.e., back in the calling frame.

In addition to the syntactic rules for JavaScript above, we spend a great deal of time focused on objects and their methods. JavaScript provides a large library of classes and we either explicitly create instances of these via the `new` operator, e.g., `rx = new Regexp("^[ACGT]+$")` or work with existing objects provided to us. We operate on these objects via their methods. We invoke these methods as if they were functions belonging to the object, e.g., `rx.exec("my string")` or `node.removeChildren()`. Objects also have properties or fields and we can query and set these, e.g., `button.value = "My Label"` and `document.links`.

There are many classes and each has many methods. This is what makes JavaScript useful, but also difficult to learn because you need to find the classes and methods of use for your task. There are a few classes and methods, however, that arise very commonly in JavaScript

code for SVG and HTML documents. One of these is the Document that provides access to the contents of the document being displayed. Another is the general concept of a document element represented by the Node and Element classes and the more specific SVGElement and HTMLElement classes and their sub-classes (e.g., SVGCircleElement, HTMLHeadingElement). These classes allow us to not only query the contents of the document, but also to programmatically modify this content, adding new elements or changing the characteristics of existing elements.

When JavaScript code is evaluated as part of an HTML or SVG document, there is an implicit global variable named `document`. This is the top-level Document object and we can refer to it without any declarations or additional code. Perhaps the most important method this object provides for our use is `getElementById()` which searches the entire SVG/HTML tree and finds the element which has an id attribute with the specified value. Another method, `getElementsByTagName()`, allows us to find SVG/HTML elements based on the name of the element/tag, e.g., "h1" or "path".

Once we have an element in the document, we can use its methods to operate on it. We can retrieve the values of the element's attributes via `getAttribute()`, e.g., `circle.getAttribute("r")`. Similarly, we can set the value of an existing or new attribute using `setAttribute()`. We can query the child nodes of a Node/Element via its `childNodes` field, and the parent node via `parentNode` field. We can add nodes via the `appendChild()` and `insertBefore()` methods, and we can remove nodes via `removeChild()`.

## D.1. Debugging and display messages

As with any programming language, the JavaScript code we write often contains bugs, especially when we are learning the language. Since the code is being run within a Web browser or specialized SVG viewer, it is being run asynchronously (i.e., when an event occurs such as the document is loaded or the viewer clicks on a shape in the plot) rather than explicitly by our direct commands. As a result, debugging JavaScript can seem somewhat difficult. There are several tools to aid us, however. It is essential to look at the error console within the Web browser, e.g., the error cosole is selected in Firefox via the menu Tools -> Error Console, and in Opera under Tools -> Advanced -> Error Console.

While the "print" approach to debugging is not generally a good one, it is a common approach in asynchronous JavaScript programming. The idea is that we display the values of variables or computations as we evaluate the code and display it for the viewer or programmer to see in order to understand how the code is behaving. We can display the message in the viewer's status bar or in a pop-up window. In addition to helping the programmer debug code, this same approach can be used to provide information to the viewer. The `alert()` function is the primary one used to display information in a pop-up window. We construct the text of the message to display and pass this as the sole argument in the call to `alert()`. The viewer must click on the "Okay" button to dismiss the window and continue with the calculations.

In an HTML browser, we can display text in the status bar by merely assigning a string containing the desired text to the variable `status`. For example, we can display the number of links in a document with

```
status = "# of links " + document.links.length
```

A more sophisticated approach to debugging is to use a browser extension such as Firebug (http://getfirebug.com) for Firefox, or Firebug Lite (http://getfirebug.com/lite.html) for various browsers (including Firefox). Firebug is an extension that the programmer has to install. Firebug lite is a JavaScript file that one can include by reference in an HTML document, e.g.,

```
<script type='text/javascript'
        src='http://getfirebug.com/releases/lite/1.2/
                                        firebug-lite-compressed.js'>
</script>
```

These provide a separate panel within the browser's tab and show many aspects of the JavaScript code and HTML document. We can write messages to the console part of this panel via

```
console.log("a string")
```

and use this to track how code is running.

We have seen how we can display information to the viewer via either a pop-up window or the status bar. There are also facilities for getting feedback from the viewer via pop-up windows. The `confirm()` function displays a message and allows the viewer to proceed or cancel an operation. For example,

```
function doConfirm()
{
var con = confirm("Do you really want to do this?");
if (con == true)
  {  ...  }
else
  { ... }
}
```

We can also get information via HTML forms and other active elements of the document rather than via separate (pop-up) windows.

For more information on debugging and programming in JavaScript, see Flanagan (2006)

## Affiliation:

Deborah Nolan
University of California, Berkeley
367 Evans Hall MC:3860
Berkeley CA 94720-3860, United States of America
E-mail: nolan@stat.berkeley.edu

Duncan Temple Lang
University of California, Davis
4210 Mathematical Sciences Building
One Shields Avenue
Davis CA 95316, United States of America
E-mail: duncan@wald.ucdavis.edu

*Journal of Statistical Software* http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 46, Issue 1 *Submitted:* 2010-01-06
January 2012 *Accepted:* 2011-09-19