Web-based Interactive Graphics

David Banks

Contents

Chapter 1

Introduction

In this report we look at the gridSVG package for R with a focus on how we can use it to solve real-world tasks. R is free statistical software used for statistical computation and graphics [28]. Our aim was to investigate whether we could use gridSVG for a number of tasks, and if so, how difficult it would be.

First we present an introduction to the grid graphics system in R and discuss its main features and capabilities. Then we take a look at what SVG is, how to use it and what we can do with it. Following that we explore the gridSVG package in R, showing how we can use it to export grid graphics into SVG format and to add extra features into exported SVG documents. Then we discuss scripting SVG documents, introducing javascript and showing how it can be used in conjunction with SVG to add interactivity to an SVG document. We then present three applications of the gridSVG package. Each application started with an idea and an aim to get that idea into an interactive web document. We use grid graphics and the gridSVG package to help us with this, and show the steps of how we obtained interactive web documents for each idea. We also discuss the level of success we had and how much work was involved in each application.

Chapter 2

An introduction to SVG

2.1 What is SVG

SVG stands for Scalable Vector Graphics. It is an open standard specification for two dimensional vector graphics [8]. The SVG standard uses an XML file format. XML (Extensible Markup Language) is a specification for encoding documents in a format that is readable by humans, and is widely used in web technologies. SVG is commonly used in web documents to produce dynamic and interactive graphics.

There are a number of important differences between SVG and raster graphics formats such as JPG and PNG. Raster graphics formats store an image as a fixed number of pixels, and so these types of images become pixelated when we zoom in. Pixelation is where the individual pixels become visible, and it makes an image look poor quality. The main difference between SVG and raster graphics is what happens when we zoom in. In SVG, graphical objects are defined by paths and curves, so that when we zoom in these objects are simply rescaled. This rescaling means that there is no pixelation or loss of quality like there is with raster graphics.

SVG documents can be scripted, which allows us to make interactive animated images. Interactive images are more successful in capturing a viewer's attention and interest than plain images (raster graphics). This makes SVG ideal for presenting information graphically on the internet. All major web browsers (Mozilla Firefox, Internet Explorer, Google Chrome, Safari and Opera) now support SVG. Support in Internet Explorer was added in version 9.

2.2 An SVG example

In this section we give a very basic example of an SVG document. Figure 2.1 shows what this document looks like when displayed in a web browser.

```
<?xml version="1.0" encoding="ISO8859-1"?>

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="700px" height="300px">

<circle id="circle1" cx="150" cy="150" r="110" stroke="black"
        stroke-width="2" fill="red" fill-opacity="0.4">
</circle>

<rect id="rect1" x="340" y="70" width="300" height="160"
      stroke="blue" stroke-width="2" fill="green"
      fill-opacity="0.2">
</rect>

</svg>
```

The first line declares that this document is in XML format. The `svg` tag starts the SVG document. The `xlmns` and `xmlns:xlink` attributes point to specifications. The `width` and `height` are called attributes of the svg element. They set the document to have a width of 700 pixels and a height of 300 pixels. These dimensions define a 'canvas' where all of the SVG objects are drawn; any element that extends beyond the edges would be cut off. The `circle` element defines a circle, which is given an `id` of 'circle1' in this example. IDs are unique identifiers for elements and are very useful for accessing SVG elements when scripting (see Section 4.1). The `cx` and `cy` attributes define the x- and y-coordinates of the centre of the circle, specified in pixels from the top-left corner of the SVG document. Note that the coordinate system in SVG defines the (0, 0) point at the top-left corner, with increasing x going from left to right and increasing y going from top to bottom. The `r` attribute defines the radius of the circle (in pixels). The `stroke` and `stroke-width` attributes define the colour and width of the circle border, respectively. The `fill` attribute defines the fill colour and the `fill-opacity` attribute defines the opacity (transparency) of the fill colour; this value ranges from 0 (completely transparent) to 1 (no transparency). The `rect` element defines a rectangle. For rect elements, the `x` and `y` attributes specify the x- and y-coordinates of the top-left corner of the rectangle. Then we close the `<rect>` tag and end the SVG document. In SVG, every element has an opening tag and a closing tag. An opening tag is of the form `<rect>` and a closing tag is of the form `</rect>`. Some elements such as the `<rect>` element are allowed to have self-closing tags that use the `<tag />` syntax. For example, the following two lines define identical rectangle objects and are both written in valid syntax:

```
<rect id=rectangle1" x="100" y="100" width="70" />
<rect id=rectangle1" x="100" y="100" width="70"></rect>
```

Other SVG elements we will commonly use are `<g>` elements, used to group elements together; `<text>` elements, used to display text; `<a>` elements, used to turn elements into hyperlinks; `<polygon>` elements, used to draw arbitrary

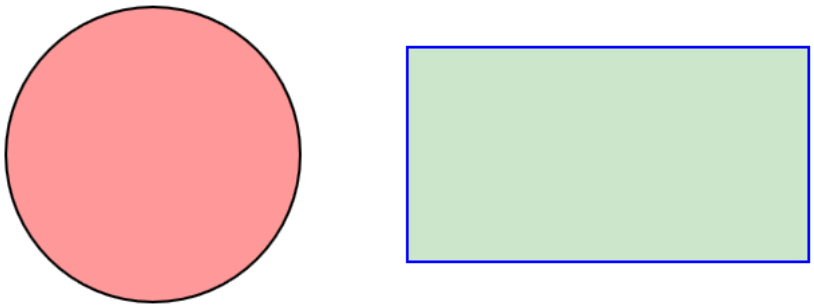

Figure 2.1: The example SVG document viewed in a web browser

shapes; `<path>` elements, used to define a path (which can be curved); and `<polyline>` elements, used to draw lines composed of straight-line segments.

Chapter 3

The gridSVG package for R

3.1 Why use gridSVG?

The gridSVG package is an R package that provides functions for creating SVG images from images built using the grid graphics system within R [24]. Although using gridSVG requires knowledge of how to work with grid graphics, there are a number of advantages in choosing this package rather over other alternatives. The main alternative is the SVGAnnotation package, which may be used with either grid graphics or traditional graphics [27]. We will compare differences between the two packages in the following example. The image we want to export to SVG format is shown in Figure 3.1. We will make export this figure to an SVG document using both SVGAnnotation and gridSVG.

```
> library(SVGAnnotation)
>
> filename = "svgDocs/hello_svgannotation.svg"
> svgPlot({
+     grid.newpage()
+     grid.rect(width = 0.5, height = 0.5, name = "border")
+     grid.text("Hello", name = "helloLabel")
+ }, filename)
```

The contents of the file `hello_svgannotation.svg` are printed below. Long lines have been truncated.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://
  <r:display xmlns:r="http://www.r-project.org" usr="0,1,0,1
grid.newpage()
grid.rect(width = 0.5, height = 0.5)
grid.text("Hello")
}
```

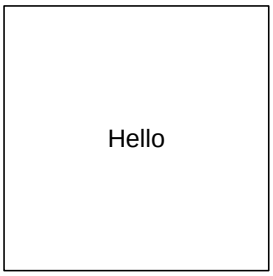

Figure 3.1: The image we want to export to SVG format

```
]]></r:display>
  <defs>
    <g>
      <symbol overflow="visible" id="glyph0-0">
        <path style="stroke:none;" d="M 1 0 L 1 -8.949219 L
      </symbol>
      <symbol overflow="visible" id="glyph0-1">
        <path style="stroke:none;" d="M 5.261719 -2.085938 L
      </symbol>
      <symbol overflow="visible" id="glyph0-2">
        <path style="stroke:none;" d="M 0.800781 0 L 0.80078
      </symbol>
      <symbol overflow="visible" id="glyph0-3">
        <path style="stroke:none;" d="M 0.414063 -3.242188 C
      </symbol>
    </g>
  </defs>
  <g id="surface6">
    <rect x="0" y="0" width="432" height="432" style="fill:
    <path style="fill-rule:nonzero;fill:rgb(100%,100%,100%);
    <g style="fill:rgb(0%,0%,0%);fill-opacity:1;">
      <use xlink:href="#glyph0-0" x="201.757813" y="220.4726
      <use xlink:href="#glyph0-1" x="210.784912" y="220.4726
      <use xlink:href="#glyph0-2" x="217.736816" y="220.4726
      <use xlink:href="#glyph0-2" x="220.513916" y="220.4726
      <use xlink:href="#glyph0-3" x="223.291016" y="220.4726
    </g>
  </g>
</svg>
```

```
> library(gridSVG)
>
> grid.newpage()
> grid.rect(width = 0.5, height = 0.5, name = "border")
> grid.text("Hello", name = "helloLabel")
> gridToSVG("hello_gridsvg.svg")
```

The contents of the file `hello_gridsvg.svg` are shown below (with long lines truncated).

```
<?xml version="1.0" encoding="ISO8859-1"?>
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://
  <g transform="translate(0, 252)  scale(1, -1)">
    <g id="gridSVG" fill="none" stroke="rgb(0,0,0)" stroke-d
      <g id="GRID.rect.78">
        <rect id="GRID.rect.78.1" x="63" y="63" width="126"
      </g>
      <g id="GRID.text.79">
        <g id="GRID.text.79.1" stroke-width=".1" transform="
          <g transform="scale(1, -1)">
            <text x="0" y="0" text-anchor="middle" fill="rgb
              <tspan dy="4.31" x="0">Hello</tspan>
            </text>
          </g>
        </g>
      </g>
    </g>
  </g>
</svg>
```

To anyone unfamiliar with SVG, neither of these documents will look easy to read. But there are some major differences. The main difference in this example is how the text 'Hello' has been coded as SVG. In the document produced by SVGAnnotation, there are four `<symbol>` elements which use `path` elements to define each unique letter in the phrase 'Hello'. Note that these paths are actually very long and are not shown in their entirety in this document. The text string itself is created by defining five `<use>` elements which use the symbols defined previously to form the word 'Hello'. In the document produced by gridSVG, the text string is defined by a `<text>` element with one child `<tspan>` element. The `<tspan>` element defines a single line of text within the `<text>` element. The line corresponding to the start of the `<text>` element has been truncated but is not much longer than what is shown here, and thus it is much shorter than the lines that had been truncated in the previous SVG document. Hence this example demonstrates how gridSVG output is more readable than the SVGAnnotation output. This is important because occasionally we will need to refer to the source code of our SVG documents and we would like it to be as readable as possible.

3.2 The format of gridSVG output

A big advantage of using gridSVG is that it makes scripting (see next chapter) much easier because of the way it creates and formats SVG documents.

Each grob drawn on the current R graphics device has its own `<g>` (group) element wrapped around the corresponding SVG element. This group element has `id` equal to the name of that grob - this is a good reason for us to give suitable names to all of our grobs. Note that `<g>` elements are used to group similar elements together in SVG. Within each `<g>` element is an SVG element that corresponds directly to the grob we have drawn in R, for example a `<text>` element corresponds to a `textGrob`. This SVG element is given `id` equal to the grob name with a `.1` suffix added to it. This can be seen by inspecting the gridSVG output given in section 3.1. The fact that gridSVG gives elements `id` attributes that match the names of grobs in R means that we can write scripts for our SVG document without having to spend great amounts of time inspecting the SVG document source code.

Chapter 4

Scripting with SVG

4.1 The Javascript Language

Javascript is a cross-platform, object-based client-side web language, used to add interactivity to a web document. We can use javascript to access the SVG DOM (Document Object Model). The SVG DOM allows access to all elements of the document and their attributes. We will use javascript extensively to add interactivity to our SVG documents. SVG does its own `<animate>` element which allows an element to be changed in some way over some duration of time, but there are many more very specific interactive features we will want to add to our SVG documents, which makes use of javascript essential.

Objects are the core concept of javascript. An object may have several properties and methods. Properties contain information about an object, and methods are functions that are specific to a certain type of object. We can access an objects properties with the syntax `object.property1`, and we call a method with the syntax `object.method()`.

Javascript statements and functions are defined within `<script>` elements. Assigments that occur outside functions define global variables. Within a function, the `var` keyword can be used to define local variables which are not accessible from outside that function. The `var` keyword can be left out if we desire to define global variables within a function.

Single line comments in javascript are indicated with `//`. Multi-line comments begin with `/*` and end with `*/`. We use comments to help explain what is going on within our javascript code.

4.2 Event Handlers

An event handler describes what should happen when a certain event occurs. We provide an event handler with a javascript statement or function call that is

to be executed every time the event occurs. In SVG, there are are several useful event handlers that we can attach to any element within an SVG document. The most common event handlers include `onclick`, `onmouseover` and `onmouseout`. These are attached to elements as attributes, demonstrated below:

```
<rect id="rectangle1" x="100" y="100" width="70"
      onclick="func1()" onmouseover="func2()" />
```

The code above defines a rectangle element with two event handlers attached to it. When the mouse moves over the rectangle, the `func2()` javascript function is called, and when the rectangle is clicked, the `func1()` function is called. We will see how to attach event handlers to elements using gridSVG in section 4.4.

4.3 Javascript examples

In this section, we present several examples of how we can use javascript within a web document.

In this first example, we use javascript to give a message when we click on a circle. The document called `circleAlert.svg` is shown below. In this example, clicking on the circle with the mouse calls the javascript function `displayInfo()`, as specified by the `onclick` event handler. The `displayInfo()` function simply brings up a dialogue box containing some information about the circle we clicked. The `evt` argument is an object representing the event that called the function. `evt.target` defines the object which triggered the event (an object can contain other objects as properties). In this case it is the circle element that we clicked on. We use the `getAttribute()` method to obtain the values of any attributes that the circle element has. Here we grab the `id` attribute and include it within our message. Note that string concatentation in javascript is done with the `+` operator. The pop-up dialogue box is created via the `alert()` function. Figure 4.1 shows this SVG document in Google Chrome, straight after clicking on the circle.

```
<?xml version="1.0" encoding="ISO8859-1"?>

<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="300px" height="300px">

<circle id="circle1" cx="150" cy="150" r="110" stroke="black"
        stroke-width="2" fill="yellow" fill-opacity="0.5"
        onclick="displayInfo(evt)">
</circle>

<script type="text/javascript">
function displayInfo(evt) {
  alert("I'm a circle. My ID is " + evt.target.getAttribute("id"));
```

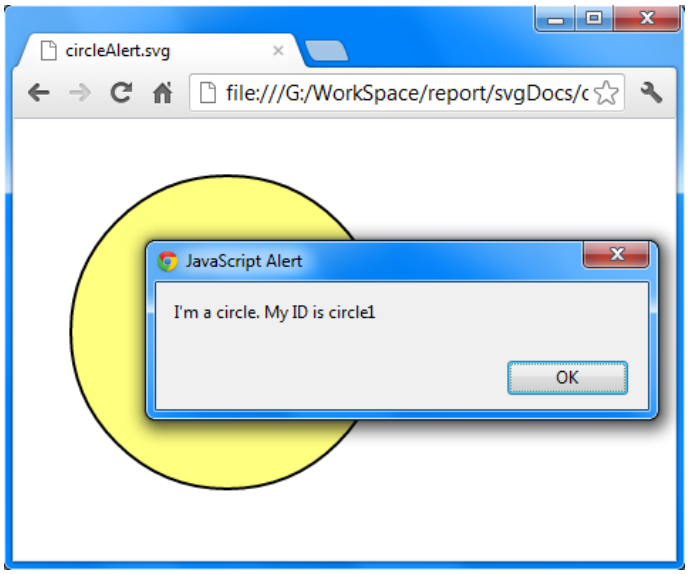

Figure 4.1: The result we get from clicking on the circle

```
}
</script>

</svg>
```

In the next example we demonstrate how to use javascript to clone nodes (SVG elements) within an SVG document. We use node cloning in Chapter 7. The SVG document is shown below, with javascript embedded internally via a `<script>` element. Note the `text/ecmascript` value of the `type` attribute. This and `text/javascript` can be used interchangeably - javascript is actually an implementation of ECMAscript. Just like the previous example, we have a circle element with an `onclick` event handler attached to it. Clicking on the circle calls the javascript function `colourRings()`. This function creates four more circles via clones, and modifies some of the properties of the clones. Node cloning is done with the `cloneNode()` method. Each time we run through the `for` loop, we create a new circle by cloning the previous clone, and set its radius 40 pixels less than the radius of the previous clone. Then using the `setAttribute()` method, we modify the radius and fill colour attributes of the new circle clone. At the end of each run through the loop, we use the `appendChild()` method to add the clone to the document. We add the clone to the parent node of the original circle, which is the `<g id=''circles''>` element. Using the `appendChild()` method on a parent node attaches the new node at the end, after any other child nodes that may exist within that parent node. In SVG, elements defined at the same level as one another follow the rule that later-defined elements will obscure previously-defined elements if there is any overlap. Thus in our example we should expect the result to be a small cyan circle in the middle surrounded by rings of different colours, because each time we created a circle, we decreased the radius and changed its colour

before drawing it on top of previously drawn elements. Figure 4.2 shows this behaviour.

```
<?xml version="1.0" encoding="ISO8859-1"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="440px" height="440px" version="1.0">

<g id="circles">
  <circle cx="220" cy="220" r="200" fill="red"
          onclick="colourRings(evt)" />
</g>

<script type="text/ecmascript">
<![CDATA[
function colourRings(evt) {
  var circle = evt.target;
  var par = circle.parentNode;
  var clone = circle;
  var radius = circle.getAttribute("r");
  var cols = ["orange", "yellow", "lawngreen", "cyan"];

  for (i=0; i<cols.length; i++) {
    radius = radius - 40;
    clone = clone.cloneNode(true);
    clone.setAttribute("fill", cols[i]);
    clone.setAttribute("r", radius);
    par.appendChild(clone);
  }
}
]]>
</script>

</svg>
```

The two examples given demonstrate only a small amount of what we can do with javascript - quite complex and interactive web graphics can be created with SVG-javascript combinations.

4.4 Adding javascript with gridSVG

There are two functions in gridSVG that we can use to add javascript to the SVG document to be created. The first one, `grid.script()`, can be used to create a `<script>` element within the SVG document. The function allows the code to be inserted inline or referenced by a filename.

The second function is the `grid.garnish()` function. We can use this function to add extra information to any grob we have drawn. When we export to SVG,

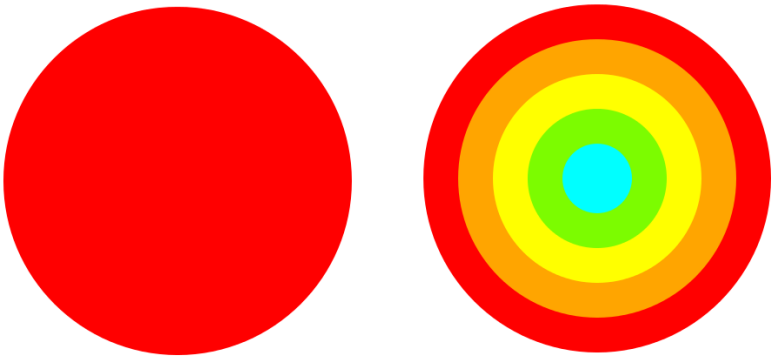

Figure 4.2: The SVG document in Firefox, before and after clicking on the red circle

this extra information is included in attributes of the corresponding elements. As an example, say we have a red-filled circle grob named 'redCircle', whose fill colour will change to blue when hovered over in our SVG document, and change back to red when the mouse leaves the circle element. Then before calling the `gridToSVG()` function, we do the following:

```
> grid.garnish("redCircle",
>              "onmouseover" = "changeColour('blue')",
>              "onmouseout" = "changeColour('red')")
> grid.script(filename = "functions.js", name = "jsFile")
> gridToSVG("circleColour.svg")
```

The functions.js file:

```
function changeColour(col) {
  var theCircle = document.getElementById("redCircle.1");
  theCircle.setAttribute("fill", col);
}
```

The section of SVG output that is relevant in this example:

```
<g id="redCircle" onmouseover="changeColour('blue')"
   onmouseout="changeColour('red')" >
  <circle id="redCircle.1" cx="336" cy="336" r="201.6"
          fill="rgb(255,0,0)" fill-opacity="1" />
</g>

<script type="text/ecmascript" id="jsFile"
        xlink:href="functions.js">
</script>
```

The above example shows how we can use these two functions together to make our SVG documents interactive.

4.5 Adding hyperlinks and animation with gridSVG

SVG is primarily used in web documents, so we want to be able to add hyperlinks to certain SVG elements. In gridSVG, the `grid.hyperlink()` function allows us to do this, as shown below.

```
> grid.hyperlink("textLabel", "http://www.google.com")
```

Running the above R command before calling the `gridToSVG()` function will mean that the corresponding text element in the exported SVG document will have an `<a>` (anchor) element wrapped around it. The anchor element will have a `xlink:href` attribute, with the value `http://www.google.com`. This turns the text element into a hyperlink, so that when it is clicked the user's web browser navigates to the Google home page.

The `grid.animate()` function allows us to animate certain elements by creating `<animate>` elements in the output SVG document. As mentioned earlier in this chapter, the `<animate>` element allows an element to be changed in some way over a period of time. The `grid.animate()` function lets the user specify options such as the duration, start-time and interpolation type of the animation, and whether or not to repeat the animation indefinitely.

Chapter 5

Application 1: Statistics Course Pathways

5.1 Introductory information

The first idea we decided to explore with gridSVG was whether we could create interactive flow diagrams of course pathways for statistics courses. This idea came from a PDF document on the Department of Statistics (University of Auckland) website. This document presents the the required and recommended courses for a variety of career fields in table format. Figure 5.1 shows a small portion of this table, take from the PDF document. Note that this figure only shows 7 columns (career fields) out of 15 columns in the original document due to restricted width here.

Statistics courses and career pathways

Statistical skills support an extraordinarily wide range of careers. Use the following table to help you plan your course programme and achieve your career aspirations.

Stage One							
STATS 10X Introduction to Statistics	Finance	Actuary	Marketing	Market Res	Org Psych	Ecology	Bioinformatics
STATS 125 Probability and its Applications	Finance	Actuary		Market Res		Ecology	Bioinformatics
STATS 150 Lies, Damned Lies, and Statistics			Marketing	Market Res			
Stage Two							
STATS 20X Data Analysis	Finance	Actuary	Marketing	Market Res	Org Psych	Ecology	Bioinformatics
STATS 210* Statistical Theory	Finance	Actuary		Market Res		Ecology	Bioinformatics
STATS 220 Data Technologies	Finance			Market Res		Ecology	Bioinformatics
STATS 255 Intro to Operations Research							

Figure 5.1: A small part of the course pathways table

We want to produce graphs of this data. Here we talk about graphs in the math-

ematical sense. A graph is a set of nodes/vertices and edges, where some pairs of nodes are connected by edges. Edges may be curved; there is no restriction on them being straight lines.

There are some important features we want to be able to include in our graphs, such as distinguishing between courses that a student is required to have completed before taking a certain course (prerequisites) and courses that are recommended but not required (also described as 'Nice to have'). For some courses, there is a choice between prerequisites: only one course is required to have been successfully completed out of a group of courses. In this chapter we use the term 'prerequisite groups'. When we say this we mean a group of one or more prerequisites where only one of them is required. In our graphs, we want to be able to highlight prerequisite groups with more than one member so that is is obvious only one of them is required. It would also make sense to split the nodes of our graphs into three parts, with Stage 1 courses on the top line, Stage 2 courses on the line below and Stage 3 courses below that.

5.2 Getting the data into an R-readable format

All of the data was contained within a PDF file and we could not see any other formats for this data. So we hand-coded small plain-text files for each career field, in such a way that R could accurately read the file and thus interpret the content of each graph. An example of one of these files is shown below. This file corresponds to the bioinformatics career field.

```
10X::
_125::
20X:10X:
210::
220::
302:20X:
330:20X:
_331:20X:
_340:20X:
_380:20X,220:
```

Each line of the file represents a node. The first course code on each line is the name each course that will feature in the graph (the course codes). Courses that are recommended but not required have had an underscore added at the start of the course code. So for bioinformatics, STATS 380 for example is not a required course. Each colon marks the start of a new prerequisite group. The maximum number of prerequisite groups is two, so there are two colons per line. Where only one prequisite group exists, nothing follows the second colon; where there are no prerequisites at all there is nothing following either colon. Members of prerequisite groups are separated by commas. Hence STATS 210 and STATS 220 have no prequisites, STATS 302 and STATS 330 require STATS 20X only, and STATS 380 (not a required course) requires a pass grade in either STATS 20X or STATS 220.

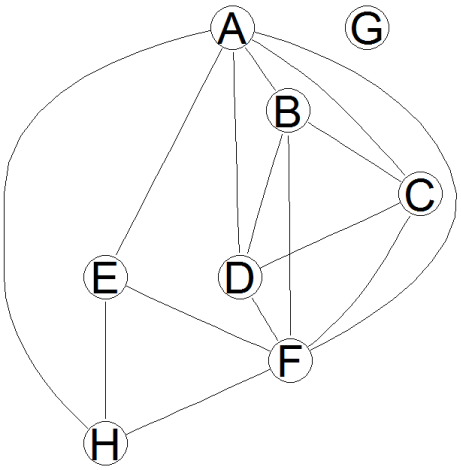

Figure 5.2: An example graph with nodes and edges

Note that prerequisites for each course do not change across the different career fields - only the sets of required and recommended courses change.

5.3 Using R to produce graphs of this data

For each career field we wanted to produce a graph using grid graphics in R. This is made possible using the graph and Rgraphviz packages [14, 15]. We intend to use gridSVG to create SVG documents, so we need two additional packages, gridGraphviz and gridDebug [23, 25], that allow us to plot graphs from the previous two packages using grid graphics. Figure 5.2 shows an example of the sort of graphs we can produce with these packages. This example graph was created using the `randomGraph()` function from the graph package.

We can read the plain-text files we created earlier into R using the `read.table()` function. This reads the data into a data frame object. A data frame is like a set of equal-length variables that shares similar properties to lists and matrices. Using this data frame we define the nodes by extracting information from the first column, and we define a list of edges using information extracted from the second and third columns. We can detect which courses are required and which are not by checking for underscores. In R this can be done with the `grep()` function.

For any given node, it is much easier to determine the incoming edges that it is to determine the outgoing edges. This is because each line in our source file describes a node and the nodes that point to it, not the nodes that it points to. Thus when we form the edge list for the graph, we define it in terms of incoming edges for each node. However the functions that allow us to produce graphs expect the edge list defined in terms of outgoing edges for each node. This means that when we create our graph object, the edges are pointing in the wrong

directions. Fortunately for us, a function called `reverseEdgeDirections()` exists within the graph package. This function takes a graph object, reverses all directed edges in that graph, and returns a modified graph.

Next we want to implement some styles that distinguish between required and recommended courses, and that highlight prerequisite groups which contain more than one member. First we set the fill colour of all nodes to be light yellow and all edges to have a black colour. For recommended courses we set the node fill colour to white. Detection of the course status (whether it is required or recommended) was fairly simple; finding the groups of edges that correspond to each prerequisite group was more complex. Each such group of edges was given a unique colour. The colours were picked carefully so that they were easy to tell apart - this was necessary since some of our graphs contain several prerequisite groups with more than one member. Figure 5.5 shows how colour has been used to highlight such prerequisite groups. All these styles are created using named lists; we call these 'lists of attributes'. Once all the attributes had been specified, we can create a laid-out styled graph using the `agopen()` function from the graph package. We then use the `grid.graph()` function from the gridGraphviz package to draw the laid-out graph using the grid graphics system. Figure 5.3 shows the final graph that we export to SVG (next section).

The graph in Figure 5.3 has not separated out the courses into the three distinct groups of stages. To implement this we can use the `subGraph()` function from the graph package. This function allows us to specify a set of node names that correspond to nodes within the main graph object, and it returns a subgraph which consists of those nodes and any edges that were defined between them in the main graph object. The R code below shows how we go about forming subgraphs and implementing them in our laid-out graph. Figure 5.4 shows the new graph.

```
> ## info is the data frame containing our data
> papers <- info$Course
> papers <- gsub("_", "", papers)
> ## Now papers contains course numbers, e.g. 125, 220
>
> ## Determine the stage of each course (1, 2 or 3)
> papers.stage <- as.numeric(substring(papers, 1, 1))
>
> ## Separate sets of courses for each stage
> level1 <- papers[which(papers.stage == 1)]
> level2 <- papers[which(papers.stage == 2)]
> level3 <- papers[which(papers.stage == 3)]
>
> ## Create three subgraphs for each stage.
> ## gnel is our initial graph object (not laid out yet)
> stage1 <- subGraph(paste("STATS", level1), gnel)
> stage2 <- subGraph(paste("STATS", level2), gnel)
> stage3 <- subGraph(paste("STATS", level3), gnel)
>
> ## Create the laid-out styled graph, containing the subGraphs.
```

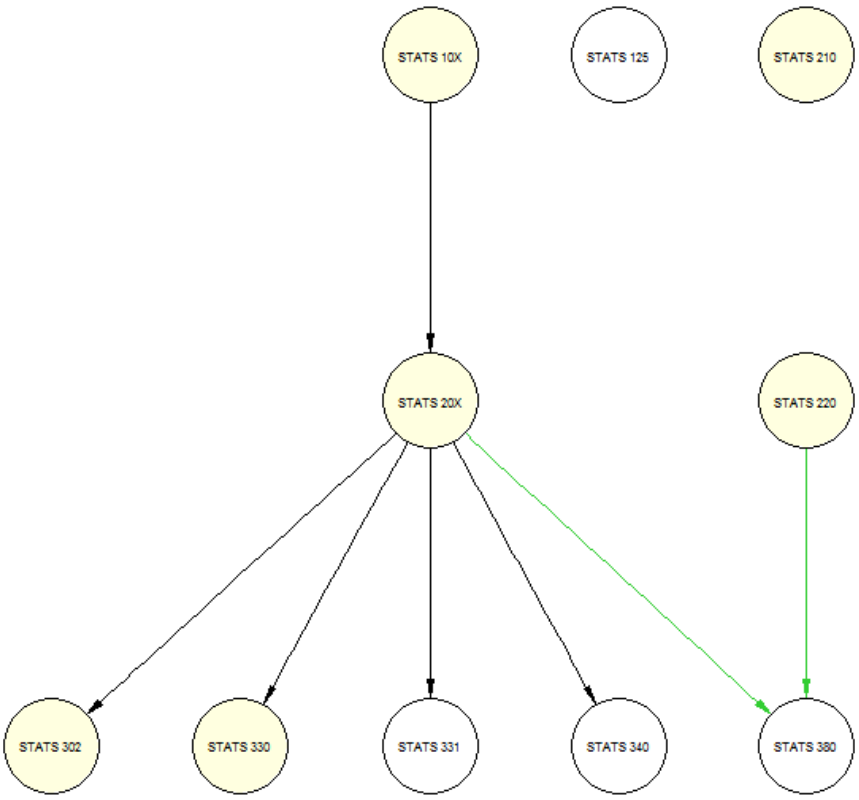

Figure 5.3: The laid-out graph with styles applied

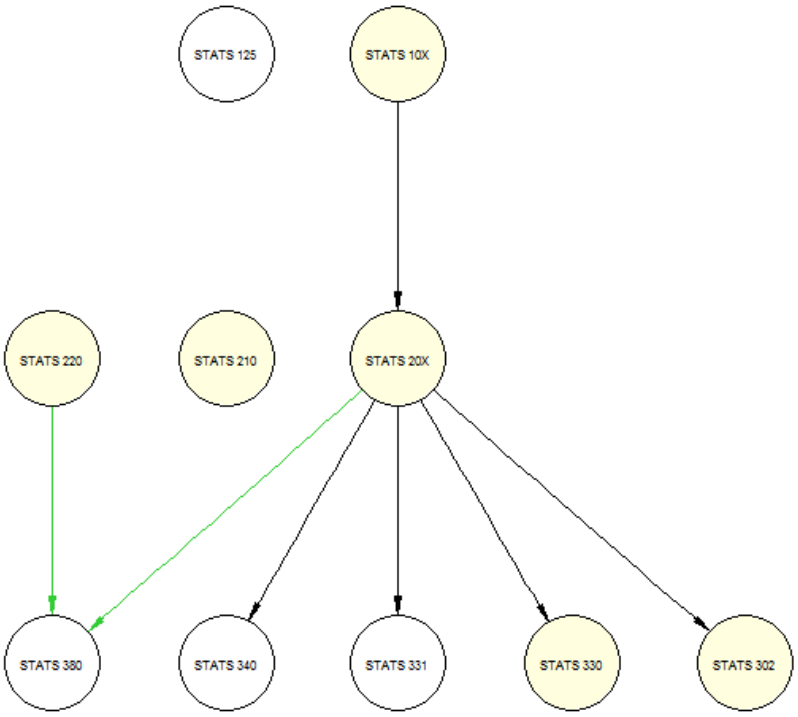

Figure 5.4: The laid-out graph with subgraphs implemented

```
> ## globalA, edgeA and nodeA are lists that we use to apply
> ## styles to certain components of our graph.
> rag <-
>     agopen(gnel, "graph1", attrs = globalA,
>            edgeAttrs = edgeA, nodeAttrs = nodeA,
>            subGList = list(list(graph = stage1),
>                            list(graph = stage2),
>                            list(graph = stage3)))
>
> grid.newpage()
> grid.graph(rag)
```

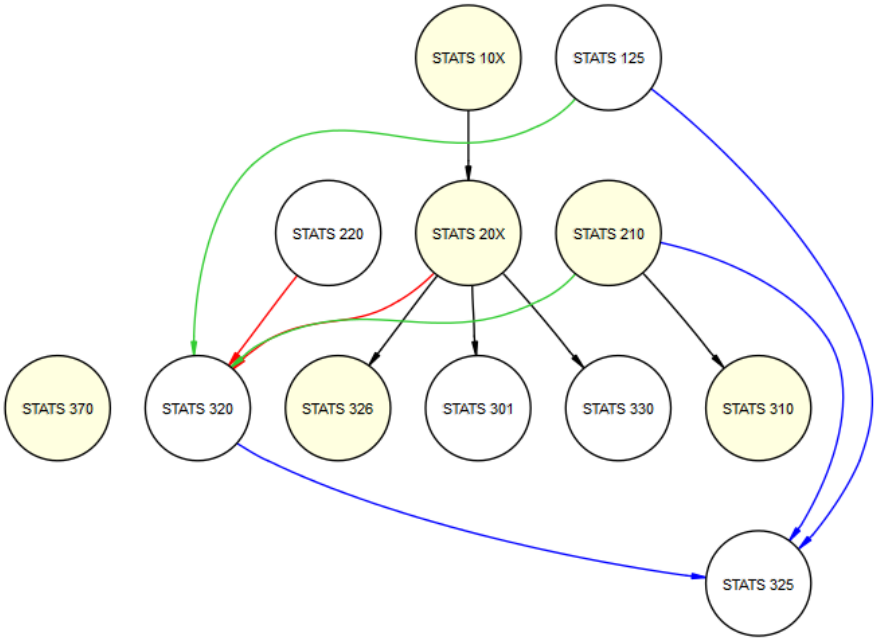

Figure 5.5: Finance course pathways (SVG), showing 3 prerequisite groups

5.4 Exporting to SVG

Before exporting the graph to an SVG document, we add the hyperlinks to each node using the `grid.hyperlink()` function. We want each node to be linked to its corresponding course webpage on the Department of Statistics website. We also garnish the nodes in the graph with a `pointer-events="all"` attribute. When this attribute is given the value 'all' on a certain SVG element, it makes sure that event handlers attached to that element trigger within the interior of the element and not just on the border. When we add event handlers like `onmouseover` and `onmouseout` to an element, we usually want them to apply to the whole interior area spanned by that element. The code below adds the `pointer-events` attributes, adds the hyperlinks and then exports our graph to an SVG document. Note that the syntax `node::box` is used in grid graphics to reference the grob named 'box' within the gTree named 'node'.

```
> ## This statement has been abbreviated here as it is not
> ## necessary to show all the links.
> courseLinks <-
>     c("STATS 10X" = "stage1/STATS_101/G",
>       "STATS 125" = "stage1/STATS_125",
>       "STATS 150" = "stage1/STATS_150",
>       ....
>       "STATS 370" = "stage3/STATS_370",
```

```
>       "STATS 380" = "stage3/STATS_380")
>
> ## The full URLs
> courseLinks <-
>     paste("http://www.stat.auckland.ac.nz/courses/",
>           courseLinks, sep = "")
>
> courseNames <- paste("STATS", papers)
> for (course in courseNames) {
>     grid.garnish(paste(course, "::box", sep = ""),
>                  "pointer-events" = "all")
>     grid.hyperlink(course, courseLinks[pap])
> }
>
> ## Export to SVG
> gridToSVG("bioinformatics.svg")
```

We have shown how we produced the Bioinformatics SVG document. The process is then repeated for the other 14 career fields, making a total of 15 SVG documents.

5.5 Making an interactive index page

With a total of 15 SVG documents, it is a good idea to have an index page linking to all of them. We wanted to have a collection of named bubbles, with each linking to one of the SVG documents produced earlier. It would also be good to have some sort of interactivity, such as changing a bubbles appearance when it is moused over.

To produce the bubbles we used a graph as before. This graph will have 15 nodes and no edges. However, using the same tools as earlier, we cannot get a circular cloud of nodes when they are no edges connecting them. So we connected every node to every other node, giving a total of $14 + 13 + ... + 1 = 105$ edges. To give the impression that no edges are present we will set the edge colour to white, the same colour as the background fill. To get our graph occupying a circular shaped region, we use the *neato* layout algorithm. The creation of the graph and the process of exporting to SVG is shown below. The code is explained with embedded comments.

```
> topics <-
>     c("Finance", "Actuary", "Marketing", "Market Research",
>       "Org Psych", "Ecology", "Bioinformatics", "Medical Stats",
>       "Statistician", "Academic Research", "Education",
>       "Government", "Ops Research", "Engineering", "Energy")
>
> ## Every possible pair of nodes is connected
> ## So for each node, the edge list contains all other nodes
```

```
> edgeList <- vector("list", length(topics))
> for (i in 1:length(topics))
>  edgeList[[i]] <- list(edges = topics[-i])
> names(edgeList) <- topics
>
> ## Create a graph object
> gnel <- new("graphNEL",
>             nodes = topics,
>             edgeL = edgeList,
>             edgemode = "directed")
>
> ## Global node attributes - apply to all nodes
> allNodeA <- list(fillcolor = "#ffedf4",
>                  color = "#ffedf4")
> ## Global edge attributes - apply to all edges
> allEdgeA <- list(color = "white")
> ## List of global attributes
> globalA <-
>     getDefaultAttrs(list(graph = list(),
>                          node = allNodeA,
>                          edge = allEdgeA))
>
> ## Return a laid-out graph, using the neato layout
> rag <- agopen(gnel, "", layoutType = "neato",
>               attrs = globalA)
>
> ## The URLs to the other SVG documents
> links <- paste(gsub(" ", "_", tolower(topics)), ".svg",
>                sep = "")
> names(links) <- topics
>
> ## Draw the graph
> grid.newpage()
> grid.graph(rag)
>
> ## Make some changes, add links, add javascript event handlers
> for (topic in topics) {
>     ## Grab the label and set its font size, change its name
>     grid.edit(paste(topic, "::label", sep = ""),
>               gp = gpar(cex = .9),
>               name =  paste(topic, "label", sep = ""))
>
>     ## Grab the bubble and change its name (explained later)
>     grid.edit(paste(topic, "::box", sep = ""),
>               name =  paste(topic, "box", sep = ""))
>
>     ## Add javascript event handlers to the bubbles
>     grid.garnish(paste(topic, "box", sep = ""),
>                  "pointer-events" = "all",
```

```
>                 "onmouseover" = paste("highlight('", topic,
>                                       "')", sep = ""),
>                 "onmouseout" = "dim()")
>
>     ## Add javascript event handlers to the labels
>     grid.garnish(paste(topic, "label", sep = ""),
>                 "pointer-events" = "all",
>                 "onmouseover" = paste("highlight('", topic,
>                                       "')", sep = ""),
>                 "onmouseout" = "dim()")
>
>     ## Add the hyperlinks to the nodes. By adding these to
>     ## the nodes both bubbles and labels become hyperlinks
>     grid.hyperlink(topic, links[topic])
> }
>
> ## Add external javascript file
> grid.script(filename = "topicsFunctions.js")
>
> ## Export to SVG
> gridToSVG("topics.svg")
```

The reason we change the names of the labels and boxes is related to `id` attributes in SVG. The `grid.graph()` function draws each node of the graph as a gTree (see Section A.6), naming the label 'label' and the circle 'box'. So we have multiple grobs with the same names. In R this does not cause conflict because the gTrees themselves have unique names. However when we export to SVG, elements recieve `id` equal to their corresponding grob name in R, regardless of whether or not they are contained within a gTree. As a result, we get multiple elements with `id` 'label' and multiple elements with `id` 'box'. This would make scripting much more difficult because it violates the unique element identifier requirement of SVG. By changing the grob names before we export to SVG, so that they are unique, we avoid these problems.

The javascript file `topicsFunctions.js` contains the functions `highlight()` and `dim()` functions, which are not shown here. The `highlight()` function, which is called when a bubble is moused over, simply increases bubble size and changes the label style to have bold font and red colour, while `dim()` simply removes these changes when the mouse cursor leaves the bubble. Figure 5.6 shows the interactive SVG document.

This index page can be found online at `http://www.stat.auckland.ac.nz/~paul/Projects/DavidBanks/Courses/topics.svg`.

5.6 Summary

We achieved a successful result in producing interactive web documents for this application. There was a fair amount of initial work required however, such

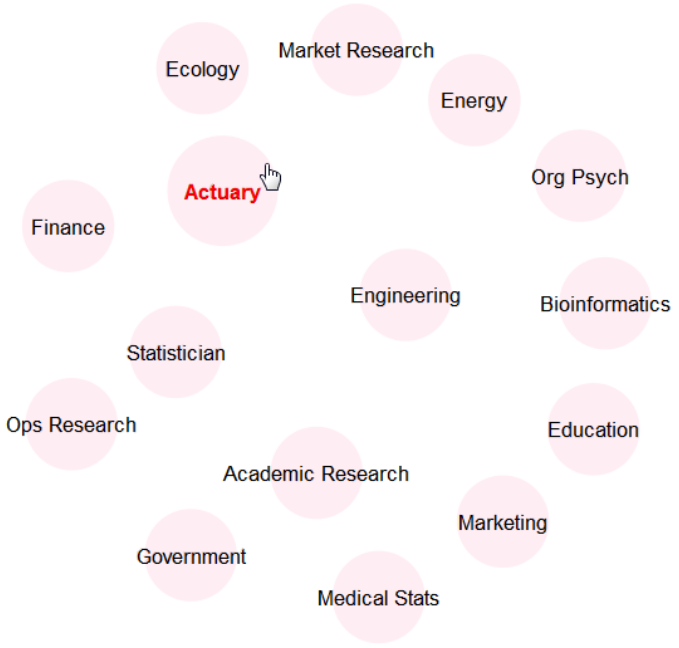

Figure 5.6: Interactive index page for Course Pathways application

as coding 15 plain-text files by hand (with careful inspection of the Course Pathways PDF so we made no mistakes), and then generating R code that would read these text files and interpret exactly what they are describing. This application also required learning how to use extra packages which produce graphs. Some of our graphs appear somewhat chaotic, but that is to be expected when the number of nodes and edges is not very small. Once we had created our graphs it was fairly simple to produce SVG documents and add hyperlinks and basic interactivity using the gridSVG functions.

Chapter 6

Application 2: Health Data for Victoria, Australia

6.1 Background Information

We were contacted by Steve Vander Hoorn, who is a statistical consultant at the Statistical Consulting Centre of the University of Melbourne, Australia. He wanted to use gridSVG to create an interactive map of the state of Victoria that could be used to display health statistics. The map was to be split up into Local Government Areas (LGAs), with the fill colour of each region representing the value of a single health statistic. Steve also suggested that it would be useful to have a scatter plot of two health variables (one of them being the variable represented in the map), and that this scatter plot would be interactively linked to the map.

The two statistics in this application are related to the human heart and were given to us with the names 'SIR' and 'SEIFA'. In R, these were coded as `sir.heart` and `obs.heart`. The map was to display the SIR variable only, and the scatter plot was to display SIR versus SEIFA. Note that the real data was not supplied to us so we worked with random data generated by R. We used uniformly distributed random data for SIR and normally distributed random data for SEIFA, produced with the `runif()` and `rnorm()` R functions.

6.2 Making the map with R

The map coordinates were supplied to us within a shapefile, and the code used to read this data into R and draw a map with them were also supplied to us. A collection of polygon grobs are used to draw the map. This process involved using a number of mapping packages, including maps [13], maptools [19] and rgdal [16]. These contained all the functions we needed to make the grid polygon grobs in R. We used the RColorBrewer package [26] to find a colour palette for

our map, and we used the classInt [12] package to split our health statistic data by quantiles into 8 intervals and to assign colours from the palette to each interval of the health statistic. Splitting by quantiles means the intervals are not necessarily of equal size, it just means we have a similar number of regions that fall within each of the intervals.

After applying the colour scheme to the polygon grobs, we can then create the map using the `spplot()` function from the sp package [29] (which is a dependency of the maptools package). This function also allows us to specify that we want a colour key on the side of our map.

6.3 Creating an image to export

We want more than just a map in our SVG document, so there are several more steps before we are ready to export to SVG. Firstly, we want a scatter plot of sir.heart versus obs.heart underneath the map. To do this we created an object representing this plot using the `xyplot()` function from the lattice package [30]. The lattice package provides functions for drawing standard plots in R using grid graphics instead of base graphics.

To arrange all the components of our image nicely we need to choose a suitable page layout. We did this using the `grid.layout()` function to split the page into two rows. We draw the map in the top row, and then draw the scatterplot in the second row within a viewport that spans 70% of the available width.

We use the `grid.garnish()` and `grid.script()` functions to add event handlers and javascript functions (discussed in next section) before exporting to an SVG document with the `gridToSVG()` function.

6.4 Converting R variables to javascript variables

We want the names of all the LGAs to be available for us to use within our javascript functions, so we need some way of converting from R variables to javascript variables. We did this using the `paste()` function in R, which is used to paste several character strings together into one character string. This was done as follows:

```
> ## Get the region names
> regionNames = as.character(vlga2@data$LGA_name)
> ## Wrap single quotes around the region names
> regionNames = paste("'", regionNames, "'", sep = "")
> ## Paste the names into one string of comma separated names
> regionsArr = paste(regions, collapse = ", ")
```

This code uses `paste()` to wrap single quotes around each region name within `regionNames`, and then uses `paste()` again to join all of the quoted region names

into one long character string, with each name separated by a comma followed by a space (specified by the collapse argument). We can add this javascript variable into our SVG document as follows:

```
> ## Include javascript internally
> grid.script(paste("regions = Array(",
>                   regionsArr, ");", sep = ""))
```

The section of code below shows how the resulting `<script>` element that appears in the SVG document. The code has been reformatted for readability. Most of the region names have been excluded and replaced by `.....`, as it is not necessary for us to list all the elements of the array here.

```
<script type="text/ecmascript" id="GRID.script.grob.19" >
<![CDATA[
regions =
  Array('Alpine (S)', 'Ararat (RC)', 'Ballarat (C)', 'Banyule (C)',
        'Bass Coast (S)', 'Baw Baw (S)', 'Bayside (C)', 'Benalla (RC)',
        'Boroondara (C)', 'Brimbank (C)', 'Buloke (S)', 'Campaspe (S)',

         .....

        'West Wimmera (S)', 'Whitehorse (C)', 'Whittlesea (C)',
        'Wodonga (RC)', 'Wyndham (C)', 'Yarra (C)', 'Yarra Ranges (S)',
        'Yarriambiack (S)'
       );
]]>
</script>
```

We can see this code has created a global javascript variable called `regions`. Making it global means that it will be accessible from within any javascript function contained within the same document. The `<![CDATA[` and `]]>` terms are the opening and closing tags for a `CDATA` section. The `grid.script()` function automatically wraps these around any javascript code that is added internally into the SVG document. Without these, certain characters such as `<` and `&` within javascript can cause parsing errors within our SVG document. In this case we don't have any problematic characters but they are always added just to be safe. Note that each region name in the array has an abbreviation inside brackets. This tells us more about the official name of each LGA. S stands for Shire, C for City, B for Borough and RC for Rural City.

6.5 Making our SVG interactive

We wanted to link the map to the scatterplot, so that when we hover over a region in the map the corresponding point in the scatterplot is highlighted, and vice versa. Recall that we have used the `xyplot()` function from the lattice package to produced our scatterplot. Since we did not name the grobs within

the scatterplot ourselves, we cannot be sure what lattice has named them, and hence we do not know what IDs the corresponding SVG elements have been given. Therefore we need to inspect the source code of the SVG document to find out what the IDs of the elements corresponding to the scatterplot points are - we need to know them for our javascript functions. From the source code we see that individual points from our scatterplot have been coded as SVG circle elements with IDs of the form `plot.xyplot.points.panel.1.1.5` for point 5, `plot.xyplot.points.panel.1.1.24` for point 24 and so on.

The javascript document we used for the SVG document is shown below. The embedded comments help to explain what each section is doing.

```
highlight = function(i, n) {
  // Make n global so we can reuse it in dim().
  // n is the number of polygons that define the region:
  // Some regions are composed of multiple polygon elements
  // (e.g. because of lakes, islands)
  window["n"] = n;

  // Check regionEl is defined, if it is not then define it.
  // RegionEl is the text element at the top.
  if (typeof(regionEl) == "undefined") {
    gEl = document.getElementById("regionLabel.1").childNodes[1];
        regionEl = gEl.childNodes[1];
  }

  // Find all polygons for the region we are highlighting and
  // highlight them. We make them global variables for easier
  // access later on in the dim() function. This gives us global
  // variables called polygon1, polygon2, ...
  for (j=1; j<=n; j++) {
    pgName = "polygon" + j;
    var pgID = "ID-" + (i - 1) + "-" + j + ".1";
    window[pgName] = document.getElementById(pgID);
    window[pgName].oFill = window[pgName].getAttribute("fill");
    window[pgName].setAttribute("fill", "red");
  }

  // get the corresponding point in the scatterplot
  var pointID = "plot.xyplot.points.panel.1.1." + i;
  point = document.getElementById(pointID);

  // Define some properties for the point object. Since point is
  // global, we can access these properties later on in the dim()
  // function.
  point.originalStroke = point.getAttribute("stroke");
  point.originalr = point.getAttribute("r");
  point.originalFill = point.getAttribute("fill");

  // Now change point properties with the setAttribute() method.
```

```
  point.setAttribute("stroke", "red");
  point.setAttribute("r", 4);
  point.setAttribute("fill", "red");
  point.setAttribute("fill-opacity", 1);

  // Display the region name at the top (regionEl text element)
  regionEl.textContent = regions[i-1];
}


dim = function() {
  // Set all the polygons back to their original fill
  for (j=1; j<=n; j++) {
    pgName = "polygon" + j;
    window[pgName].setAttribute("fill", window[pgName].oFill)
  }

  // Set the point back to its normal style
  point.setAttribute("stroke", point.originalStroke);
  point.setAttribute("r", point.originalr);
  point.setAttribute("fill", point.originalFill);
}
```

We then garnish the grobs with SVG attributes using the `grid.garnish()` function. Finally we use the `grid.script()` function to include the javascript code shown above as an external file:

```
> ## Add onmouseover and onmouseout attributes to the points
> grid.garnish("plot.xyplot.points.panel.1.1", group = FALSE,
>              "onmouseover" = paste("highlight(", 1:79, ",",
>                                    numPolygons, ")", sep = ""),
>              "onmouseout" = rep("dim()", 79))
>
> ## If we don't set point-events to all, onmouseover will only
> ## work on the border of the points and not inside
> grid.garnish("plot.xyplot.points.panel.1.1", group = TRUE,
>              "pointer-events" = "all")
>
> ## Add onmousover and onmouseout attributes to the polygons
> for (i in 1:79) {
>     n <- numPolygons[i]
>     for (j in 1:n)
>         grid.garnish(paste("ID", i-1, j, sep = "-"),
>                      "onmouseover" = paste("highlight(", i, ",",
>                                            n, ")", sep = ""),
>                      "onmouseout" = "dim()")
> }
>
> ## Add javascript functions file to the SVG document
```

```
> grid.script(filename = "map.js")
```

An alternative way to get javascript representations of R variables is to use the RJSONIO package. We use this method in the next application (Chapter 7).

6.6 Adding hyperlinks to the regions

We thought it would be useful to add hyperlinks to the regions of the map, since we noticed that Wikipedia had pages on each LGA within Victoria. Conveniently the Wikipedia URLs were consistent, so that it would not take too much work to automatically produce the URLs in R instead of copying and pasting the links manually. There are some variations however - for example the URLs can have either 'City of [LGA name]' or '[LGA name] City'. We use the RCurl package [17] to help us identify which of the alternative URLs is correct. This is done as follows. Embedded comments help explain what each piece of code is doing.

```
> regions = as.character(vlga2@data$"LGA_name")
> numRegions = length(Regions)
> urls = character(numRegions))
> ## Replace any hyphens from the region names with spaces
> regions2 = gsub("-", " ", regions, fixed = TRUE)
> ## All the names follow this pattern
> patt = "([a-zA-Z ]+) \\(([A-Z]*)\\)"
>
> ## Need to change the user agent string that RCurl provides.
> ## This prevents cases where Wikipedia returns a FALSE result
> ## when the URL does actually exist.
> curl <- getCurlHandle()
> curlSetOpt(.opts = list(useragent = "arbitraryString"),
>            curl = curl)
>
> for (i in 1:numRegions) {
>   ## Extracts the LGA name only
>   place = gsub(patt, "\\1", regions2[i])
>   ## Replace any spaces with underscores
>   place = gsub(" ", "_", place, fixed = TRUE)
>   ## Extracts abbreviated LGA type, i.e. B, C, RC, S
>   place.type = gsub(patt, "\\2", regions2[i])
>   ## Full type of LGA
>   region.type = switch(place.type,
>                        S = "Shire",
>                        C = "City",
>                        B = "Borough",
>                        RC = "Rural_City")
>
>   ## One possible URL ending
```

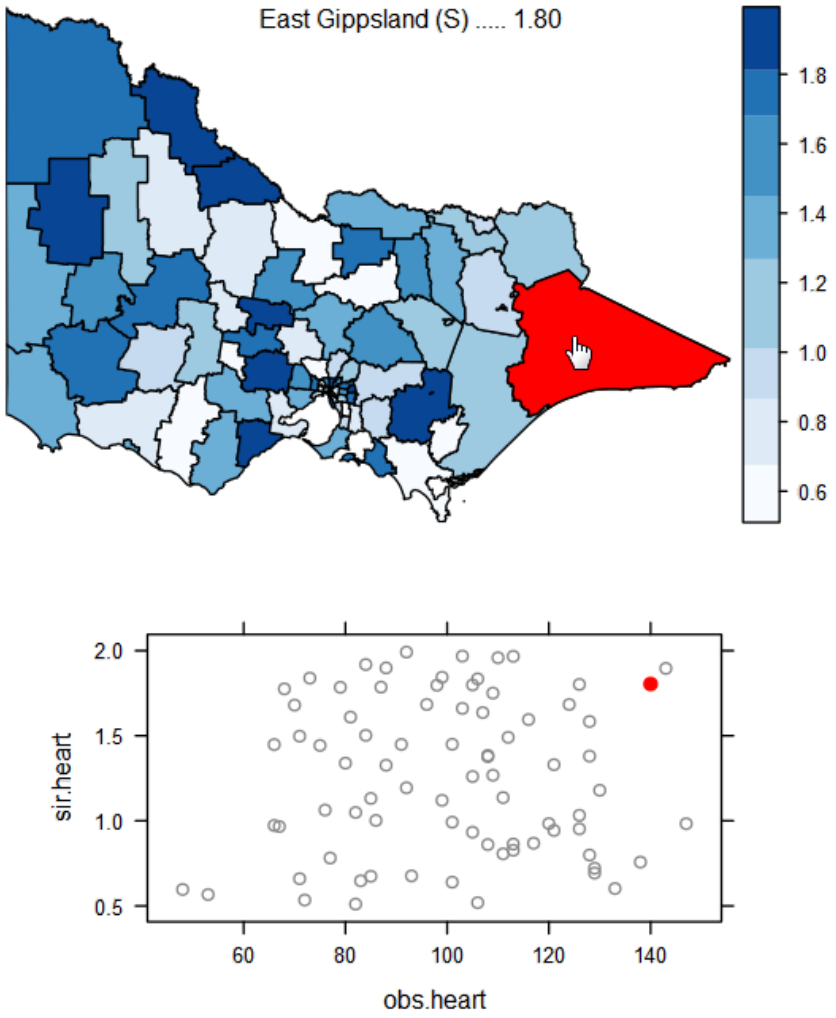

Figure 6.1: Interactive SVG document

```
>   url.id = paste(place, region.type, sep = "_")
>   ## A second possible URL ending
>   url.id2 = paste(region.type, "of", place, sep = "_")
>   ## Full URL
>   url = paste("http://en.wikipedia.org/wiki/", url.id,
>               sep = "")
>
>   ## Run through the alternatives, checking which one is correct
>   if (!url.exists(url))
>     url = paste("http://en.wikipedia.org/wiki/", url.id2,
>                 sep = "")
>   ## Some rural cities codied as cities, so check for this too
>   if (!url.exists(url) & place.type == "RC")
>     url = gsub("Rural_", "", url, fixed = TRUE)
>   urls[i] = url
> }
```

This gives us a vector of 79 URLs, one for each LGA on our map of Victoria. We convert the `urls` variable into a javascript variable and include it internally in our SVG document as demonstrated earlier in Section 6.4.

6.7 Integrating into HTML

HTML (HyperText Markup Language) is the language used to write web pages for the internet. It is an XML based language (like SVG) that is used to define content and structure of web pages. There are several versions of HTML. We used HTML5, which is the most recent version of HTML to be released. Of particular importance for us is that HTML5 allows SVG to be embedded inline within a web page. This means we can literally paste the contents of an SVG file into the HTML document instead of having the SVG content in an external file and using the HTML `<embed>` tag or `<object>` tag to include it at a certain point in the document. One advantage of including SVG inline is that is makes scripting much more simple - interaction between HTML elements and SVG elements becomes easier because they are both part of the same parent document.

We decided to try to integrate this interactive SVG application into an HTML web page with extra features, such as a scrollable/clickable list of region names that interacts with the map and scatterplot. First we needed to get the SVG inline within our HTML document. To do this, we decided to make a template HTML file that contained tags of the form `%tag%`. Then we can use the `readLines()` function to read the template into R, and we use the `gsub()` function to replace the tags with content before finally saving the modified content to a new file using the `writeLines()` function. The template HTML document is shown below.

```
<!DOCTYPE html>
```

```
<html>
<head>
<title>Interactive map</title>
<link rel="stylesheet" href="styles.css" />
<script type="text/javascript">
%js%
</script>
<script type="text/javascript" src="map.js"></script>
</head>

<body>
<div id="h1Wrapper">
<h1>sir.heart for LGAs of Victoria</h1>
</div>

<table id="outerTable">
  <tr>
    <td>
      <table id="columnHeadings">
        <col style="width:200px" />
        <col style="width:70px" />
        <tr>
          <th>Region</th>
          <th>sir.heart</th>
        </tr>
      </table>
      <div id="regionTableWrapper">
        <table id="regionTable">
          <col style="width:200px" />
          <col style="width:70px" />
          %tr%
        </table>
      </div>
    </td>

    <td>
      %svg%
    </td>
  </tr>
</table>

<br />
</body>
</html>
```

There are three tags in this file which mark places where we need to insert content from R. In place of the `%js%` tag we substitute javascript variables that have been created as described in Section 6.4. The `%tr%` tag is where we will insert HTML table rows from within R. We use a `<table>` HTML element to

create our list of region names. Therefore there will be a table row for each region, and we do not want to type all 79 of these out manually. The `%svg%` tag is where we will insert the contents of our SVG document. Substituting content in place of the tags is done as shown below. Just like in the we have done previously with javascript variables, the `tableRows` R variable is created using the `paste()` function.

```
> ## Read in the template
> template = readLines("svgmapTemplate.html")
>
> ## Substitute the javascript and table rows in
> htmlCode = gsub("%js%", javascript, template)
> htmlCode = gsub("%tr%", tableRows, htmlCode)
>
> ## Read the SVG document in
> svg = readLines("map_modified.svg")
>
> ## Remove XML declaration on first line - invalid for inline svg
> svg = svg[-1]
>
> ## Substitute the SVG inline
> htmlCode <- gsub("%svg%", paste(svg, collapse = "\n"), htmlCode)
>
> ## Write the results to a new file
> writeLines(htmlCode, "svgmap.html")
```

For those unfamiliar with HTML, we include a brief summary of how to interpret the above document. The `<head>` section defines the head of the document. This is where we include CSS styles, javascript and other information about the document like the title. The line `<link rel="stylesheet" href="styles.css" />` defines an external CSS stylesheet. CSS stands for Cascading Style Sheets. It is the language used for styling HTML elements. CSS can also be included inline by adding `style` attributes to elements. The `<body>` section contains all of the visible content of the document. A `<div>` element is basically just a container element. They are useful for breaking a page up into sections, positioning content and applying styles to large blocks of content. To create a list of region names we use the `<table>` element. The `<tr>` element defines a table row, the `<td>` element defines a table cell, and the `<th>` element defines a table heading (basically the same as a `<td>` element but with bold text styling by default). We use `<col>` elements to specify the width of each column in the table. The `<br />` element defines a line break.

Using HTML, we have arranged the content of our document as such: A heading at the top, then a table underneath it. This top-level table contains one table row with two cells (one row and two columns). Within the first cell is a table that contains only two headings (`<th>` elements). The second cell contains a second table which has 79 rows and 2 columns containing the region names and the values of sir.heart (the variable which is colour-coded in the map). There is a good reason for splitting the table headings into a separate table. The list of region names is quite long, so we opt for a fixed-height list with a scrollbar. If

we include the table headings inside the scrolling section, they are hidden when we scroll down the list. It is best to have the table headings always visible. To make it look like one table, we used `<col>` elements to make the column widths in each of the tables indentical. The second cell (column) of the top-level table contains the SVG content of this document. We have put our scrollable list of the left hand side of the Victoria map.

We can use CSS styles to improve the appearance of the document, though its use is minimal here. We used CSS styling to specify that the `<div>` element that contains the table of region names and values should not exceed 700 pixels in height, which is how we achieve scroll bars. The title bar at the top also has styling applied. Refer to Figure 6.2 to see what the HTML document looks like in a web browser.

Now we discuss changes we have made to our javascript functions. The highlighting has changed slightly from the standalone SVG we made earlier (Figure 6.1). Filling the regions in with red obscures what category the region belongs to in respect to the colour scale. We changed this so that the border is now highlighted red instead of the fill colour. This change applies to both polygons on the map and the points in the scatterplot. There is some hidden complexity here because polygon borders overlap each other. To successfully highlight the border we need to bring the polygon to the front so that nothing else overlaps its borders. We used the `cloneNode()` javascript method demonstrated in Section 4.3. The clone is made to have no fill colour and a thick red border. Appending the clone as the last child of the polygon-parent `<g>` element (the element that contains all the polygons of the map as child elements) allows us to effectively bring the polygon border to the front of the SVG image, so that none of it is obscured and our highlighting works as desired. Finally a completely new interactive feature has been added. Clicking on a category of the colour scale on the right of the map highlights all regions that fall into that category. Clicking the selected category again de-selects that category and removes the highlighting. Figure 6.3 shows this feature in action.

The HTML document is online at `http://www.stat.auckland.ac.nz/~paul/Projects/DavidBanks/Victoria/svgmap.html`.

6.8 Summary

We produced a fairly interactive HTML document for this application, which we regard as a very successful outcome. For this application, much of the initial work in importing spatial data into R and drawing the map of Victoria was already given to us when we were contacted by Steve Vander Hoorn. This part would have involved learning how to use extra R packages for mapping spatial data. Some additional tasks that were not necessary in the Course Pathways application were the conversion of R variables into javascript variables using the RCurl package to generate valid web links that we could add to our map. The first extra task was fairly simple. We were fortunate with the hyperlinks in that the URLs on the Wikipedia website were generally consistent in the way they were named. The majority of work for us in the Victoria Health Data

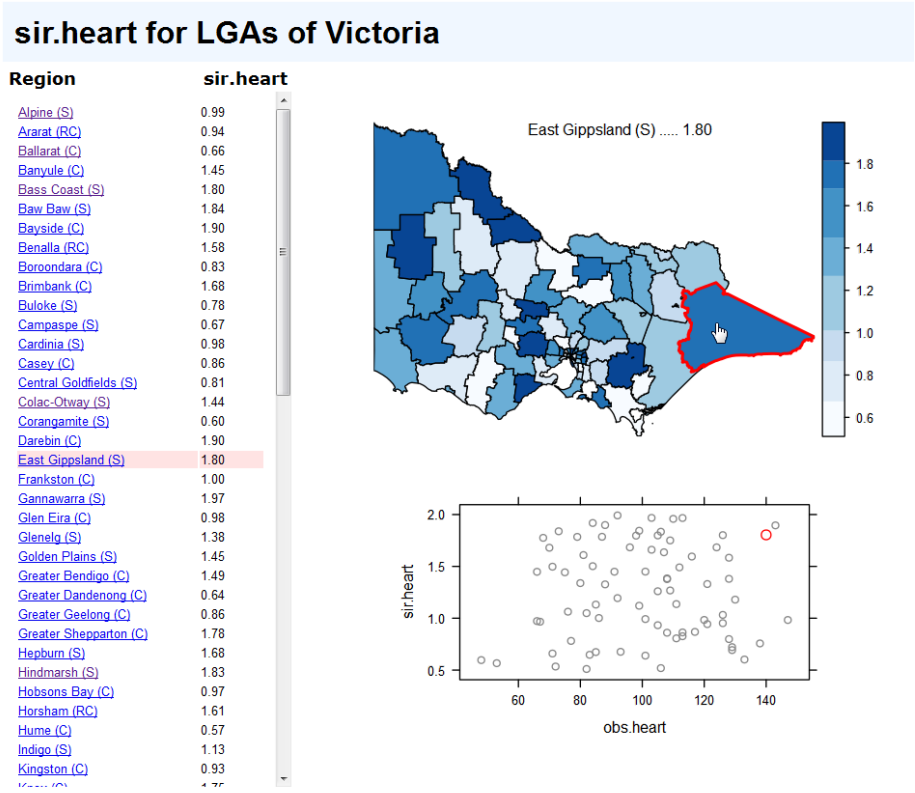

Figure 6.2: HTML version of the Victoria health data application

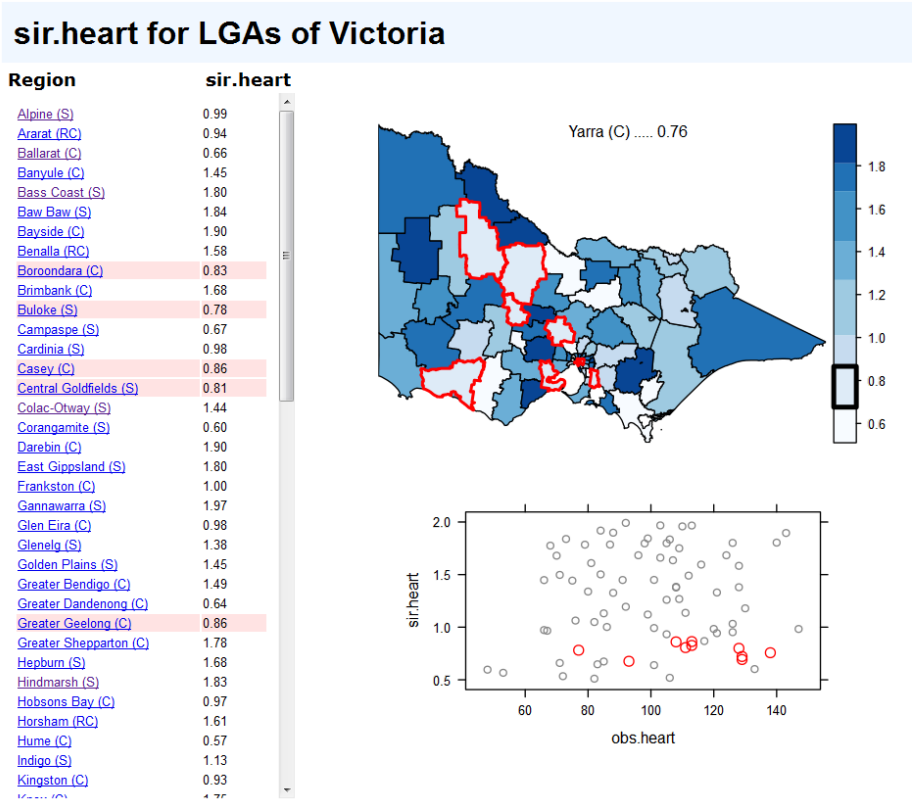

Figure 6.3: Category selection by clicking on the colour scale

application was in writing the javascript functions that provide interactivity, and integration into an HTML web document. Our javascript functions make use of more advanced javascript capabilities such as node cloning. Producing an interactive HTML document allowed us to add even more interactive features, like a scrollable list of regions. This involved a few clever tricks such as inserting information from R into a template HTML document via use of tags, and also including SVG inline. For the web development novice, this task would require learning a fair bit of new material, such as how to write a web document in HTML, and how to successfully blend SVG, HTML, CSS and javascript together to produce a high quality web document.

Chapter 7

Application 3: NZ Price Kaleidoscope

7.1 Origin of the idea

We recieved interest from Statistics New Zealand in having a New Zealand Price Kaleidoscope. The idea came from the Germany Price Kaleidoscope [6], which has been produced by the German Statistical Office. Figure 7.1 is a screenshot of the German Price Kaleidoscope. The German Price Kaleidoscope can be found online at `https://www.destatis.de/Voronoi/PriceKaleidoscope.svg`.

7.2 Using the German Price Kaleidoscope

The circular shape in the middle is split into multiple regions; these regions are subdivided further into sub-regions. Each chunk (sub-region) of the circle represents a category used in the calculation of the CPI (Consumer Price Index). The area of a chunk approximates the weighting that the corresponding category has in the calculation of the CPI, and the fill colour describes the price change between two time periods. From each top-level region, lines run out to the sides of the kaleidoscope. These lines link the top-level regions to labels, which also contain weight and price change information for these categories. When we mouse over the chunks of the kaleidoscope in a web browser, a tooltip appears and follows the direction the mouse cursor moves in. The contents of the tooltip give the category name, the CPI weighting for that category and the price change for that category. The tooltip disappears when the cursor leaves the boundary of the kaleidoscope.

Our aim was to produce something with similar functionality and interactivity, but with New Zealand data instead.

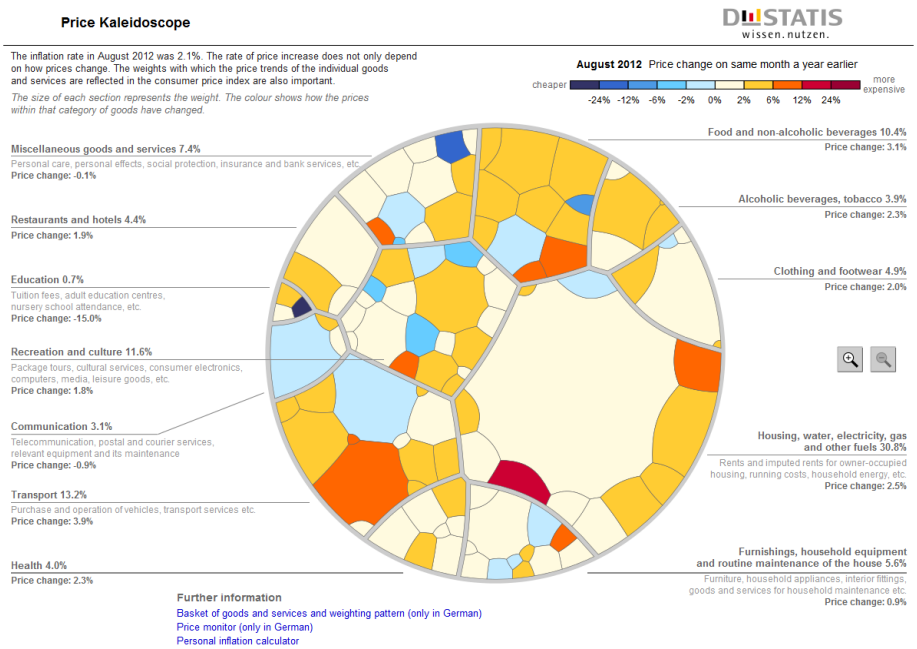

Figure 7.1: The German Price Kaleidoscope

7.3 Acquiring New Zealand CPI data

Before attempting to use gridSVG to see if we could replicate this for New Zealand data, we needed to find CPI weights and price changes for New Zealand. These were found on the Statistics New Zealand website, on the Infoshare feature of their website [3]. The data was composed of groups, which were split up into subgroups. Some of the subgroups were further split up into classes. The data was collected at quarterly intervals, with the most recent set of weights spanning from 2006 to 2012. The file containing the weights had to be manually reformatted to get it into a format that R could read.

7.4 Making our own circle of regions

The next step was to figure out how we could use this information from Statistics New Zealand to produce a segmented circle like in the German Price Kaleidoscope. We achieve this using a Voronoi Treemap diagram. A Treemap is used to present hierarchical data in graphical format, by dividing and subdividing rectangular regions up into smaller rectangular regions. A Voronoi Treemap uses non-rectangular regions, and allows regions of any shape to be subdivided into smaller parts. This is done by an iterative algorithm. The algorithm starts with an initial set of points in two-dimensional space and then tries to define borders around the points which split the total area up into multiple regions. The points and borders are shifted around in an attempt to minimise the difference between the desired areas and the calculated areas. Once the algorithm

has converged, the solution is saved into an RDA file, which is file that stores R objects. The solution is saved as a set of R polygon objects. The objects can be read into R again later using the `load()` function. The objects we saved contain all the information about the polygons that we need to draw the regions. All of the R code for making the Voronoi Treemap can be found in the online article *Voronoi Treemaps in R* [21].

7.5 Creating a suitable page layout

Now we needed to define a suitable page layout for our kaleidoscope. This was done using a viewport tree (see Appendix A). We define a main central viewport to draw all of our polygons, viewports on the left and right sides of this to be used for labelling (like in the German Price Kaleidoscope), and top and bottom viewports for titles and extra information that we may decide to include. The following code defines an appropriate layout:

```
> ## The respect matrix
> respect.mat = matrix(0, nrow = 3, ncol = 3)
> respect.mat[2, 2] = 1
>
> ## The page layout
> layout =
      grid.layout(3, 3,
                  heights = unit(c(1.1, 1, 0.4),
                  c("inches", "null", "inches")),
                  widths = unit(c(0.5, 1, 0.5), "null"),
                  respect = respect.mat)
>
> ## The parent viewport, with layout specified above
> parent.vp = viewport(name = "parent", layout = layout)
>
> main.vp =
      viewport(name = "main",
               xscale = c(-1000, 1000),
               yscale = c(-1000, 1000),
               layout.pos.row = 2,
               layout.pos.col = 2)
> top.vp =
      viewport(name = "top",
               layout.pos.row = 1,
               layout.pos.col = 1:3)
> bottom.vp =
      viewport(name = "bottom",
               layout.pos.row = 3,
               layout.pos.col = 1:3)
> left.vp =
      viewport(name = "left",
               layout.pos.row = 2,
```

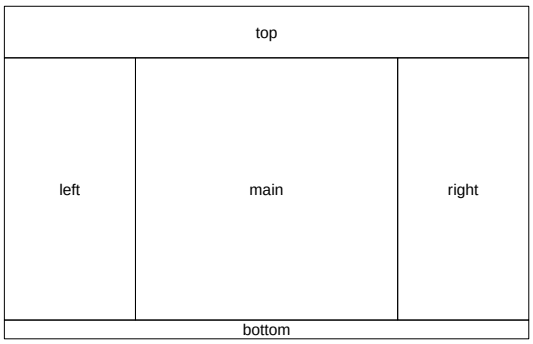

Figure 7.2: How the page has been split into regions

```
                layout.pos.col = 1)
> right.vp =
      viewport(name = "right",
                layout.pos.row = 2,
                layout.pos.col = 3)
>
> ## Put the viewports into a viewport tree
> vp.tree =
      vpTree(parent.vp,
             vpList(main.vp, top.vp, bottom.vp,
                    left.vp, right.vp))
>
> ## Apply the page layout and push all the viewports
> grid.newpage()
> pushViewport(vp.tree)
```

A brief explanation of this code may be useful. The respect matrix is a matrix of zeroes except for the `[2,2]` entry which has value 1. When giving this argument to the `grid.layout()` function, we are forcing the viewport `main` to have equal width and height. We define a parent viewport and apply the layout to it. Then we define the 5 viewports we want in our layout and what parts of the layout they should each occupy. Once these have been defined we put them into a viewport tree. This viewport tree sets the 5 viewports as children of the parent viewport. Then we can call `pushViewport()` once to draw the whole layout and push all the viewports within the viewport tree.

Figure 7.2 shows how our page has been split and where the viewports are located.

7.6 Presenting price changes using colour

We will present price changes as a percentage change between a comparison quarter and a baseline quarter. The baseline quarted is earlier in time than

Figure 7.3: Our colour scale

the comparison quarter. We represent the price changes in each category using a fill colours. We want a scale that assigns cool colours (blue for example) to categories that have a price drop, and assigns hot colours (red for example) to categories that have a price rise. We chose to make the boundaries of categories on the colour scale at the points -20, -10, -5, -2, 0, 2, 5, 10 and 20. This gives us 5 colours either side of zero, which we choose with the help of the `hcl()` function. The following code does this. See Figure 7.3 for the colours we will use. Note that for classes/subgroups with a zero price change, we use a very pale grey fill colour that is not shown as part of the colour scale (because zero is a point value and not an interval).

```
> ## Form the cool colours and warm colours
> numCols = 10
> negCols = hcl(250, 70, seq(25, 95, length = numCols/2))
> posCols = hcl(0, 70, seq(25, 95, length = numCols/2))
```

The colour scale is drawn on the right hand side within the top viewport of our page layout. A title is also drawn in the centre of the same top viewport.

7.7 Positioning the labels

The next task was to figure out how to position the side labels automatically. We want to add side labels for the top level groups only (we have 11 of these). We aim to show labels for the subgroups inside tooltips within our interactive SVG document. To help with this label positioning task we used the `TT.polygon.centroids()` function from the `soiltexture` package [20]. Given a polygon object, this function calculates the centroid of the polygon. Based on the 11 centroids, We use the median x-coordinate to decide which regions will have their labels on the left side and which will regions have their labels on the right side. Using the median rather than the mean ensures we have a similar number of labels on each side (impossible to get an equal number with a total of 11). Our initial y-coordinates for label positions set equal to the y-coordinates of the polygon centroids. Using the centroid coordinates we draw straight lines from the centroids out to the side panels. But then when we add the labels, weights and price changes in the left and right panels we see there is a problem with overlap (Figure 7.4).

To fix this we developed an iterative algorithm with a number of steps that calculates suitable positions for the y-coordinates of the line-endings. The y-coordinates for these depend on the label height, because long labels get bro-

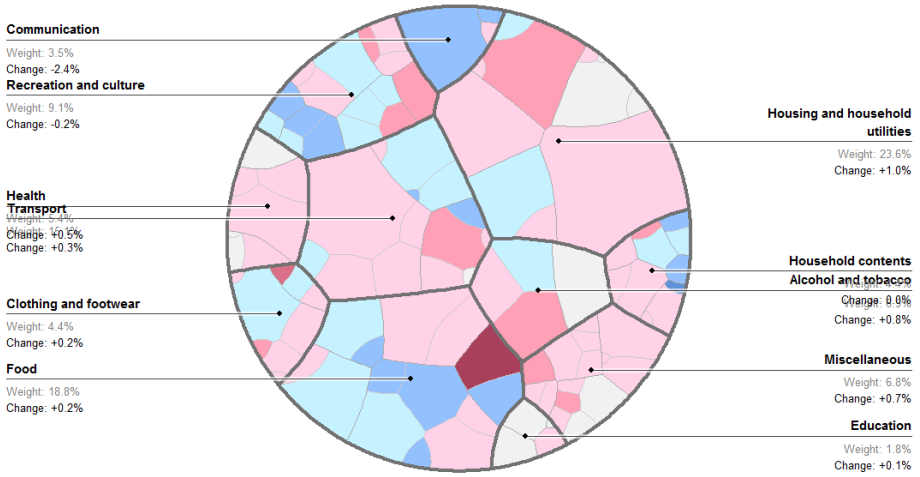

Figure 7.4: The group labels and information overlap

ken onto multiple lines. Our algorithm takes this into account. To calculate label widths and heights we form a set of temporary text grobs with initial y-coordinates equal to the y-coordinates of the polygon centroids. Then we calculate the top and bottom y-coordinates of these grobs based on label height, and use them to determine whether any pair of labels overlap. Labels are shifted up or down if necessary. This process runs a number of times, since shifting a label to fix one overlap may create another. After all overlaps have been remedied, we check if any of the labels have been pushed up or down too far (beyond the boundaries of the page). If they have, we run through another process which reduces unused space between labels. We shift labels vertically up or down to fill in these spaces and hopefully get all the labels back within the boundaries of the page. The final locations of the y-coordinates are shown as black dots in Figure 7.5.

With this final set of coordinates, we can then create a new set of text grobs (with the final coordinates), and we can also create line grobs that join labels to the polygon centroids. Since we have adjusted the y-coordinates on the side panels from their initial values, the y-coordinate for a given label may not match the y-coordinate for the corresponding polygon centroid. As a results some of the lines are draw with two line grobs, so that they appear as lines with a single bend in them. See Figure 7.6 for the complete kaleidoscope with no overlapping labels.

7.8 Exporting to an SVG document

Once the polygons, labels, lines, colour scale and title had been drawn, the next step was to export the image to SVG format. This step is once again made simple with gridSVG: we use the `gridToSVG()` function and provide it the filename that we want our SVG code to be written saved in. The conversion to

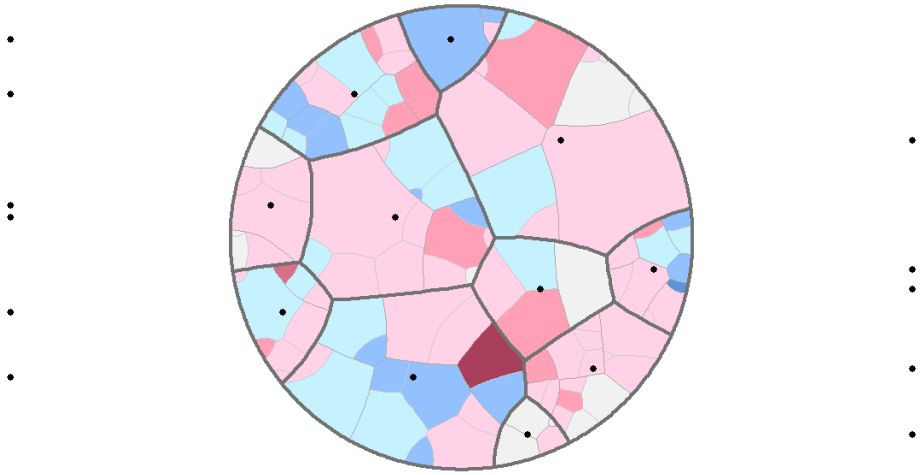

Figure 7.5: Final set of coordinates for the labels and centroids

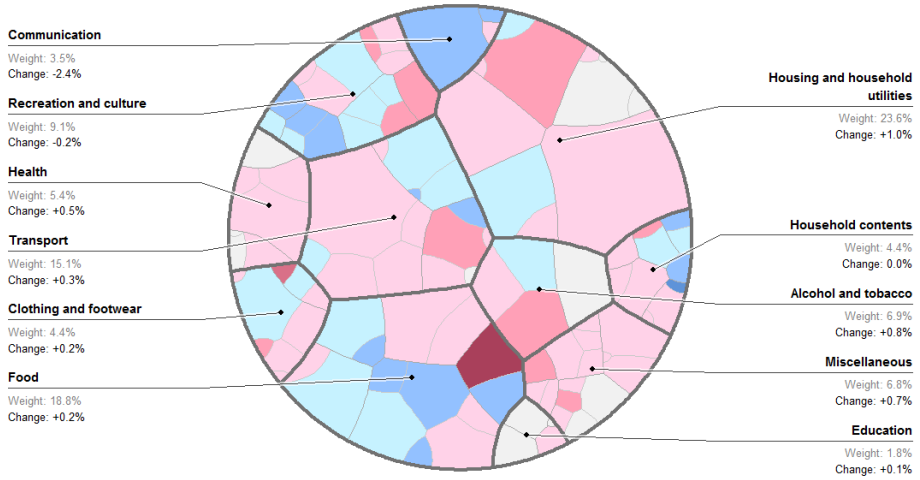

Figure 7.6: The final kaleidoscope)

SVG does not change the appearance of the kaleidoscope at all when compared to the image we drew in R. See Figure 7.6 in the previous section. This figure also shows what the resulting SVG document looks like when viewed in a web browser.

7.9 Converting R variables to javascript variables

Before we begin to write javascript functions which add interactivity to our SVG document, we need certain information to be available to our functions, such as category names, CPI weightings and quarterly prices. All of this data is available to us in R variables - we just need to convert them into javascript variables. In the Victoria Health Application we did this ourselves using the `paste()` function. This time around we used the `toJSON()` function from the RJSONIO package [18]. This function takes an R variable and returns a character string that contains a valid javascript representation of this variable. Two simple examples are given below:

```
> toJSON(1:10)

[1] "[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]"

>
> z = list(item1 = 1:2, item2 = "abc")
> toJSON(z)

[1] "{\n \"item1\": [ 1, 2 ],\n\"item2\": \"abc\" \n}"

> ## \n means newline
> cat(toJSON(z))

{
 "item1": [ 1, 2 ],
"item2": "abc"
}
```

After we have created javascript variables with RJSONIO, we saved them into a file using the `writeLines()` function. Later on we can create a reference to this external file from our SVG document using the `grid.script()` function.

7.10 Adding interactivity

Now that we have the price kaleidoscope in SVG format, we can start to add interactivity with javascript.

The first thing we want to do is to add tooltips to our categories, so that when we hover over a polygon we get a tooltip that displays category name, its CPI weighting and its price change. We did this by creating SVG `<rect>` and `<text>` elements that define the tooltips, and then appended them to the end of the SVG document. Each polygon element in the SVG (each class/subgroup) was given three event handlers: `onmouseover`, `onmouseout` and `onmousemove`. The `onmouseover` event handler calls a function which changes the visibility of the tooltip elements to'visible' and updates the information displayed within the tooltip. The `onmousemove` event handler calls a function that updates the location of the tooltip, so that it follows the mouse cursor as it moves over a region. The `onmouseout` event handler calls a function which sets the visibility of the tooltip elements back to 'hidden'.

To emphasise which region is being moused over, the function called by `onmouseover` also sets the border of the polygon to thick and black. This is not done directly by changing the polygon attributes - it is more complicated than that because polygon borders overlap each other. This is the same problem we ran into in the Victoria Health Data application. To get the desired effect, we cloned the polygon element and then changed the clone so that it has no fill colour and a thick black border. This uses the `cloneNode()` method we used before. By appending this clone onto the end of the document, the cloned polygon has effectively been brought to the front (nothing can overlap it) and it acts like a wire-frame around the original polygon, thus achieving the desired border effect. Figure 7.7 shows the tooltips and border highlighting in action.

A second feature we decided to add was zooming. We wrote our code to produce the kaleidoscope so that it was general enough to be able to produce kaleidoscopes for subsets of the data as well. This enabled us to produce 'sub-kaleidoscopes' for all 11 top-level groups (and thus 11 more SVG document). We used the `grid.hyperlink()` function to add hyperlinks to the side labels of the full kaleidoscope, so that clicking a label takes us to the SVG document containing a sub-kaleidoscope for that group. Mousing over a linked label will show a tooltip that says 'Click to zoom', which lets the user know that this zooming functionality exists. Within each sub-kaleidoscope we added a 'Back' link which takes the user back to the full kaleidoscope. Refer back to Figure **??**. Clicking on the Food side label takes us to the document seen in Figure 7.8.

7.11 Integrating into HTML for further functionality

Finally, we decided to see if we could add further functionality with drop down lists that let the user pick which two quarters to calculate the price changes from. To do this required integrating the SVG document into an HTML web

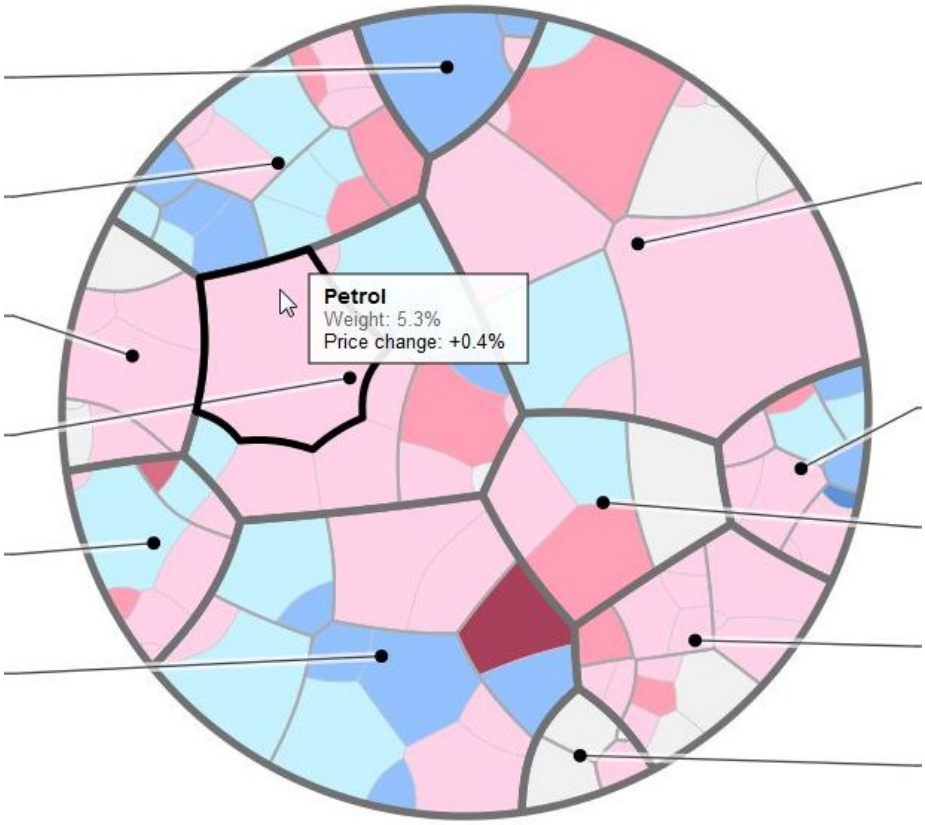

Figure 7.7: Tooltip and border highlighting

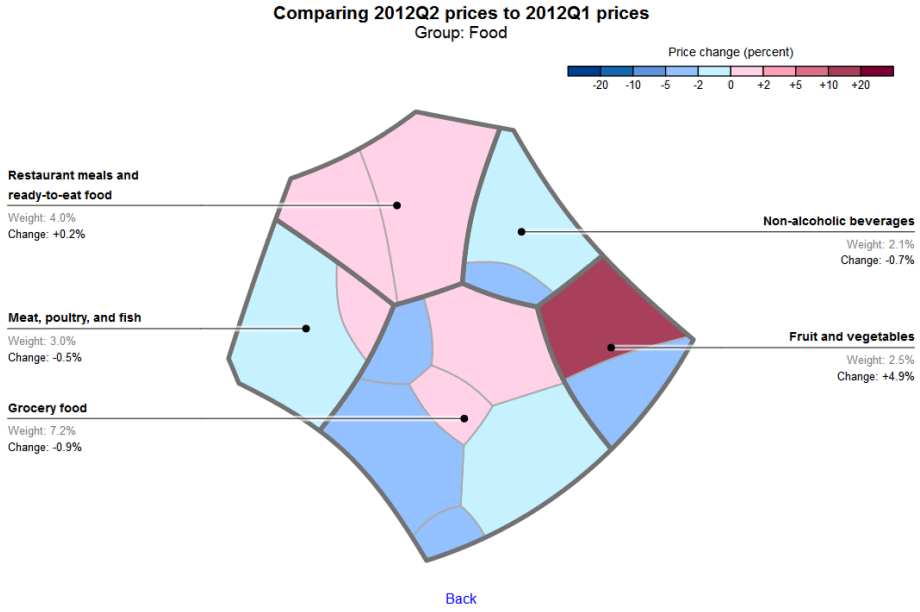

Figure 7.8: Zooming in on the Food group

page like we did previously.

The drop down lists can be created using HTML `<select>` elements within a `<form>` element. In HTML, the `<form>` element contains other elements that allow the user to select and/or enter data, such as text boxes, check boxes and drop down lists and radio buttons. A form can either be submitted and processed by a web server, or we can use javascript functions to complete actions based on user input. We will use an `<input type="button">` element to create a clickable button for our form, which will execute some javascript code. To add two drop down lists and a clickable button, this is what we needed to include in our form (shortened here with `....` indicators):

```
<form>
  <select id="comparison">
    <option value="2012Q2">2012Q2</option>
    <option value="2012Q1">2012Q1</option>
    ....
    <option value="2006Q2">2006Q2</option>
  </select>

  <select id="baseline">
    <option value="2012Q2">2012Q2</option>
    <option value="2012Q1">2012Q1</option>
    ....
    <option value="2006Q2">2006Q2</option>
  </select>
```

```
  <input type="button" onclick="updateValues()" value="Update" />
</form>
```

There are 24 quarters, so to save us typing out 24 `<option>` elements within our HTML document, we used the same method discussed in Section 6.7. A template HTML document was read into R, and tags in it were replaced with HTML content using `gsub()`. The input element in the code above defines a button with the text 'Update' printed on it. When the button is clicked its `onclick` event handler calls the javascript function `updateValues()`. This function looks at what options are selected in the two drop down lists, grabs the price data for the corresponding quarters, calculates the percentage change in price for all groups, subgroups and classes, and then assigns new colours to all the polygons and updates the side labels with the new price change information. It also updates a global javascript variable with the new price changes, so that the tooltips have access to these new values. The javascript functions used in this application are quite long and complicated, so they are not displayed here.

A final addition to our HTML form were 'Back one' and 'Ahead one' hyperlinks that both call javascript functions when clicked. The `backOne()` function changes the selected options in both drop down lists to the previous (time-wise) options if possible, and then calls `updateValues()`. Similarly the `aheadOne()` function selects the next option in both drop lists if possible and then calls `updateValues()`.

The addition of an HTML form to our kaleidoscope meant we had to change how zooming was done. Otherwise, when we click a side label to zoom in, we lose the information that was selected in our form. To get around this, we decided to use AJAX (Asynchronous Javascript And XML) which is a web technology that enables us to exchange information with a web server without reloading the page (or loading a new page). In order to use AJAX we had to upload our work to a web server and continue from there. We set up a simple PHP script on the server that accepts a single parameter, and sends back SVG content and based on its value. PHP (Hypertext Preprocessor) is a server-side language that allows us to run scripts within a web document before sending the results to the client after it has been processed. Now when we zoom in, a javascript function creates an AJAX request which returns the corresponding SVG document. This SVG document replaces the current SVG content that is shown, and then the `updateValues()` function is called to ensure the price changes and polygon fill colours reflect the options currently selected in our HTML form. Due to the size of the main SVG document, the SVG content for the main kaleidoscope is actually saved into a global javascript variable. This means that when we zoom back out after zooming in we don't have to request the document from the server and wait for it to be sent back over the internet every time. The changes we have made by implementing AJAX means that we stay on the same page the whole time, unlike the SVG-only version of the kaleidoscope we produced earlier.

The final HTML interactive kaleidoscope can be viewed at the following URL: `http://www.stat.auckland.ac.nz/~paul/StatsNZ/HTML/htmlkaleid.html`.

7.12 Summary

The NZ Price Kaleidoscope application was the most complicated application we worked on. Initially we had to make manual edits to the Statistics New Zealand data files to get them into a format that could be read into R and to fix up a few mismatching category names. Thankfully, already-existing code allowed us to produce a Voronoi treemap using this data without too much difficulty. Writing our own label positioning algorithm was quite time consuming and proved more difficult than first thought. Our algorithm produced an acceptable solution, which saved us having to manually place labels and lines. This was particularly useful for when we had to change the shapes of the regions part way through. With good label positions calculated we then drew everything and used the standard gridSVG functions to garnish grobs, add scripts and hyperlinks. Conversion to SVG using gridSVG gave us the foundations to start building an interactive web document. Just like in the Victoria Health Data application we had to convert R variables into javascript variables, but this was easy using the `RJSONIO` package. Our javascript functions were much more complicated than what we used previously, utilising advanced concepts including node cloning, SVG element creation, and positioning tooltips so that they always remain within the boundaries of the document and AJAX requests. Integration into HTML was a large task in itself, and requires a certain level of web-development experience, especially with concepts like AJAX and PHP being used. This application was certainly the most difficult and time consuming out of the three applications presented in this report. However with the help of gridSVG we were successful in crafting a professional quality interactive web document.

Chapter 8

Summary and Conclusion

We have presented three applications that demonstrate how gridSVG can be used to produce SVG documents from within R. We have shown how these SVG documents can be modified, with javascript and other web technologies, and how we can develop professional interactive web applications based on gridSVG output. Through gridSVG, we can script SVG images to provide features such as hyperlinks, highlighting and tooltips, to name a few. We can add just about any functionality we like by using gridSVG to embed javascript and event handlers in our SVG documents. In conclusion, the gridSVG package presents R users with an alternate way of presenting statistical data, by allowing images drawn with the grid graphics system to be made interactive and saved in a commonly used web format. These documents can be uploaded to a website so that they can be viewed and explored by anyone with an internet connection. Interactive images are not only able to efficiently present a vast amount of information compared to their static image counterparts, they also hold the viewer's attention and interest more successfully. The gridSVG package should not be overlooked for anyone wanting to use R to present interactive graphics on the web.

Appendix A

Appendix 1: An introduction to grid

A.1 The basics of grid

Grid is an alternative graphics system in R. It has a number of differences from the traditional (base) graphics system. The grid graphics system in R provides only low level graphics functions. For example `grid.rect()` and `grid.text()` draw rectangles and text, respectively. There are no high-level functions that draw whole plots, however there are still slightly higher level functions such as `grid.xaxis()` which draws an x-axis. Complex images and plots are created by making a series of calls to these lower level functions. Grid has a set of graphical parameters used to control settings such as colour, line type and line width. Like the base graphics system, output drawn with grid will cover any previously drawn output that it overlaps with. To start a blank page of grid output we call `grid.newpage()`.

Some basic low-level functions are demonstrated in the following example. The result is shown in Figure A.1.

```
> grid.rect(x = 0.5, y = 0.5)
> grid.circle(x = 0.5, y = 0.5, r = 0.4)
>
> ## A hexagon
> angles = seq(0, 2*pi, length = 7)
> grid.polygon(x = 0.5 + 0.25*cos(angles),
               y = 0.5 + 0.25*sin(angles))
>
> grid.text("The\nmiddle")
```

It is a good idea to name graphical objects that you draw with grid. This makes it easier to identify objects and modify them later on, especially when there are a lot of objects. Naming is done like this:

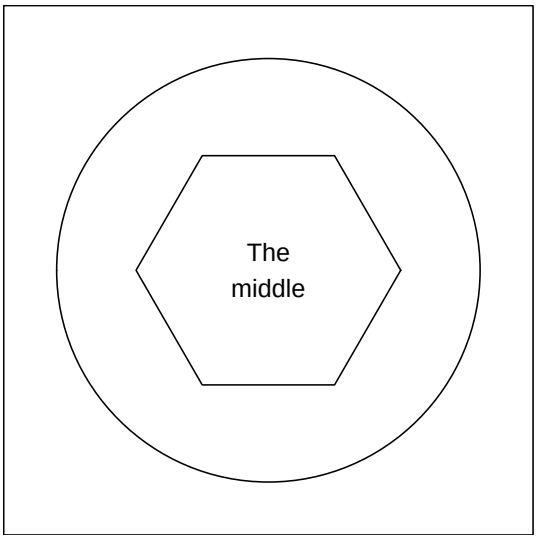

Figure A.1: Some low-level grid graphics functions

```
> grid.text("Some text", name = "myLabel")
```

The names of all the graphical objects that have been drawn on the graphics device can be obtained by calling the `grid.ls()` function.

Because every object in grid is named, we can edit output after it has already been drawn, using the `grid.edit()` function. Suppose we have drawn a circle named circle1 with a red fill colour, but we decide we want to change the fill colour to yellow. Instead of editing our code and then running it again, we can do this:

```
> grid.edit("circle1", gp = gpar(fill = "yellow"))
```

The previous example uses graphical parameters which are discussed in the next section.

A.2 Graphical parameters

We can change the appearance of graphical objects in grid by changing graphical parameters. The following example demonstrates how to do this. The result is a hexagon with each of the six segments filled in a different colour, with some transparency added by setting `alpha` to 0.5.

```
> grid.newpage()
>
> angles = seq(0, 2*pi, length = 7)
```

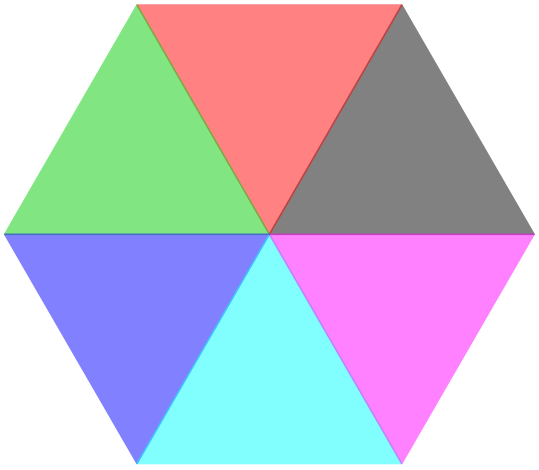

Figure A.2: Using graphical parameters in grid

```
> xc = 0.5*cos(angles)
> yc = 0.5*sin(angles)
> j = c(1:7, 1)
>
> for (i in 1:6) {
+   grid.polygon(x = 0.5 + c(0, xc[j[i + 0:1]]),
+                y = 0.5 + c(0, yc[j[i + 0:1]]),
+                gp = gpar(fill = i, col = i, alpha = 0.5))
+ }
```

A.3 Using units

There are a number of different units we can use in grid to specify locations and sizes of graphical objects. The default units are npc (Normalized Parent Coordinates), which set the bottom-left corner to be (0,0) and the top-right corner to be (1,1). Some of the other units are native units, inches, cm and mm. We use the `unit()` function to specify units. We can also add (or subtract) different units. An example of using units is given below, see Figure A.3 for the result.

```
> grid.newpage()
> vp1 = viewport(height = unit(4, "cm"))
> pushViewport(vp1)
> grid.rect()
>
> ## Add labels on each side
```

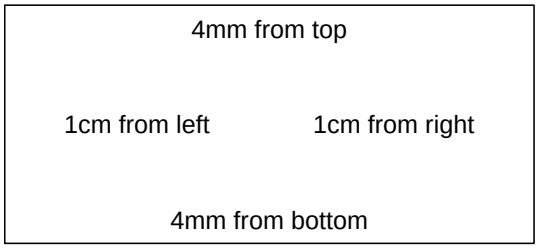

Figure A.3: Using units in grid

```
> grid.text("4mm from top", y = unit(1, "npc") - unit(4, "mm"))
> grid.text("4mm from bottom", y = unit(4, "mm"))
> grid.text("1cm from left", x = unit(1, "cm"), just = "left")
> grid.text("1cm from right", x = unit(1, "npc") - unit(1, "cm"),
            just="right")
```

A.4 Grobs

Grid functions also produce objects that describe output. These are called grobs (shorthand for graphical objects). All grobs in grid are named, which makes them accessible for modification even after they have been drawn. Every drawing function in grid has an accompanying function that returns a grob and does not add anything to the display list, i.e. does not draw anything. (the display list is a list of grobs that have been drawn in the R graphics device).

For example, we can do the following:

```
> circ = circleGrob()
> grid.draw(circ)
```

This is equivalent to the single statement:

```
> grid.circle()
```

By using grob functions, we can define and modify all of the components of a plot without actually drawing them. Any grob can be drawn by calling

`grid.draw(g)`, where `g` is the grob to be drawn on the current graphics device.

It is a good idea to give all grobs suitable names, especially when we are planning to use gridSVG to make an interactive SVG image. The difficulties that arise by not giving grobs meaningful names is demonstrated by the following example. The following code is the same as the code used in section A.3. Refer back to Figure A.3. Suppose we want to change the colour of the label at the top to blue. We need to identify which grob to edit:

```
> grid.newpage()
> vp1 = viewport(height = unit(4, "cm"))
> pushViewport(vp1)
> grid.rect()
>
> ## Add labels on each side
> grid.text("4mm from top", y = unit(1, "npc") - unit(4, "mm"))
> grid.text("4mm from bottom", y = unit(4, "mm"))
> grid.text("1cm from left", x = unit(1, "cm"), just = "left")
> grid.text("1cm from right", x = unit(1, "npc") - unit(1, "cm"),
            just="right")
>
> ## Print the names of the grobs that have been drawn
> grid.ls()


GRID.rect.58
GRID.text.59
GRID.text.60
GRID.text.61
GRID.text.62
```

We can see that it is not clear which names correspond to which text objects. Let us modify our code so that we give all grobs suitable names.

```
> grid.newpage()
> grid.rect(gp = gpar(lty = "dashed"))
> vp1 = viewport(height = unit(4, "cm"))
> pushViewport(vp1)
> grid.rect(name = "myRect")
>
> ## Add labels on each side
> grid.text("4mm from top", y = unit(1, "npc") - unit(4, "mm"),
+           name = "topLabel")
> grid.text("4mm from bottom", y = unit(4, "mm"),
+           name = "bottomLabel")
> grid.text("1cm from left", x = unit(1, "cm"), just = "left",
+           name = "leftLabel")
> grid.text("1cm from right", x = unit(1, "npc") - unit(1, "cm"),
+           just="right", name = "rightLabel")
```

Now it is obvious which grob we need to edit. Then we can change the top label to have a blue colour as follows.

```
> grid.edit("topLabel", gp = gpar(col = "blue"))
```

We can also remove graphical objects from the graphics device, using the `grid.remove()` function.

A.5 Viewports

A strong feature of grid is the concept of viewports. A viewport defines a region where output is drawn. There may be multiple viewports on a graphics device, and they may have different scales and coordinate systems. Viewports can be contained within other viewports, and they may overlap.

A basic example showing how we can use viewports to position elements on the graphics device is shown below. First we define two viewports, `vp1` and `vp2`, both with width and height equal to 30% of the graphics device. In this example we have put `vp1` at the top left and `vp2` at the bottom right. Next we 'push' `vp1` onto the graphics device using the function `pushViewport()`. We can think of this function as setting the viewport to be our current drawing area on the graphics device. We give the viewport a border and draw a text label in it before 'popping' it. Popping a viewport sets the drawing area back to the parent viewport, i.e. the viewport that we pushed from. We repeat the same for the second viewport `vp2`. The result of this code is shown in Figure A.4.

```
> grid.newpage()
> grid.rect()
> vp1 = viewport(x = 0.25, y = 0.75, width = 0.4, height = 0.4)
> vp2 = viewport(x = 0.75, y = 0.25, width = 0.4, height = 0.4)
>
> pushViewport(vp1)
> grid.rect()
> grid.text("vp1")
> popViewport()
>
> pushViewport(vp2)
> grid.rect()
> grid.text("vp2")
> popViewport()
```

We can also nest viewports within each other. In the following example, we define a viewport `vp1` which has width and height equal to half that of the graphics device. Then we push it onto the graphics device draw a blue border around it. Next, we push `vp1` again from within the current viewport, which defines a new viewport half the width and height of the previous one. We give this viewport a red border. Finally we push `vp1` again, creating a viewport with

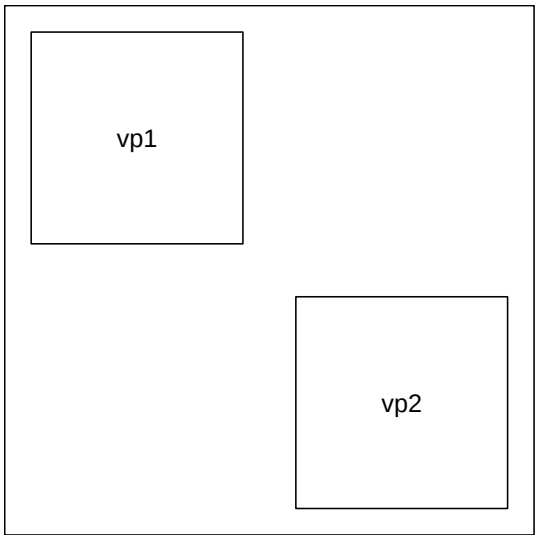

Figure A.4: A basic example of using viewports

one eighth the height and width of the graphics device, and fill it in with yellow. The final command `popViewport(0)` pops all viewports and brings us back to the top-level viewport. The final result is shown in Figure A.5.

```
> grid.newpage()
> vp1 = viewport(x = 0.4, y = 0.4, width = 0.6, height = 0.6)
>
> pushViewport(vp1)
> grid.rect(gp = gpar(col = "blue"))
>
> pushViewport(vp1)
> grid.rect(gp = gpar(col = "red"))
>
> pushViewport(vp1)
> grid.rect(gp = gpar(fill = "yellow"))
>
> popViewport(0)
```

Another feature of viewports that is sometimes useful is clipping. Suppose we enable clipping on viewport A, and then draw a circle inside A whose radius is larger than the width of A. The parts of the circle which overflow the boundaries of A will not be shown - they will be clipped. If we drew this large circle in a viewport without clipping enabled, the circle is allowed to flow outside the viewport boundaries.

Viewports can also be rotated, so that everything drawn inside them is also rotated relative to the parent viewport. This is demonstrated in the following example, the result of which can be seen in Figure A.6.

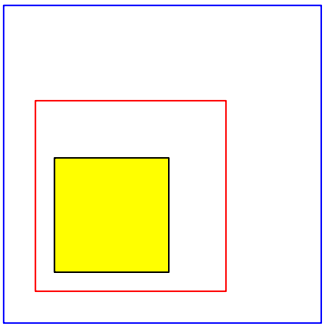

Figure A.5: An example of viewport nesting

```
> grid.newpage()
> grid.rect()
> grid.text("Not rotated", x = unit(2, "mm"),
            y = unit(1, "npc" )- unit(2, "mm"),
            just = c("left", "top"))
>
> pushViewport(viewport(width = 0.7, height = 0.3, angle = 20))
> grid.rect()
> grid.text("Rotated 20 degrees", x = unit(2, "mm"),
            y = unit(1, "npc" )- unit(2, "mm"),
            just = c("left", "top"))
```

A.6 Viewport trees and grob trees

The final features of grid that we present in this introduction to grid are viewport trees and grob trees. A viewport tree, or vpTree, can be thought of as a single parent viewport which has any number of children viewports. When a viewport tree is pushed using the `pushViewport()` function, the parent viewport is pushed first and then the children viewports are pushed in parallel (at the same level as each other). Viewport trees are often useful when using a grid page layout. Using a page layout with the `grid.layout()` function allows us to cut up a drawing area into rows and columns. We use these in conjunction with viewport trees by applying a layout to a parent viewport, and then specifying which cells/rows/columns each of the children viewports will occupy (refer to Section 7.5 to see this idea applied in one of our applications).

A grob tree, or gTree, can be described a grob that contains other grobs. Each

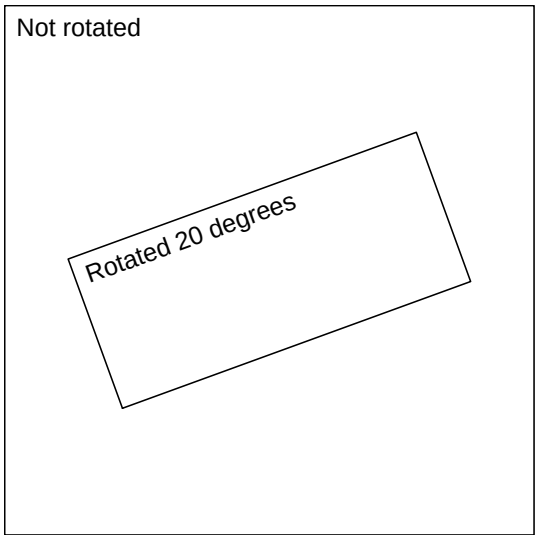

Figure A.6: A rotated viewport

grob in a gTree can contain a viewport path that specifies which viewport that grob should be drawn in. gTrees are useful because we can change properties and graphical parameters of higher level grobs so that they modify their children to match those changes.

Bibliography

[1] Course pathways for undergraduate statistics students at the university of auckland. http://www.stat.auckland.ac.nz/webdav/site/stat/shared/for/future-undergraduates/documents/CoursesncareerpathwaysforUG2012.pdf. Accessed on 17 September 2012.

[2] Health data for victoria - online resource. http://www.stat.auckland.ac.nz/~paul/Projects/DavidBanks/Victoria/svgmap.html.

[3] Infoshare - statistics new zealand. http://www.stats.govt.nz/infoshare/. Accessed on 16 September 2012.

[4] Mozilla developer network: Javascript guide. https://developer.mozilla.org/en-US/docs/JavaScript/Guide. Accessed on 24 September 2012.

[5] Nz price kaleidoscope - online resource. http://www.stat.auckland.ac.nz/~paul/StatsNZ/HTML/htmlkaleid.html.

[6] Price kaleidoscope of the federal statistical office of germany. https://www.destatis.de/Voronoi/PriceKaleidoscope.svg. Accessed on 24 September 2012.

[7] A programming Q and A community. http://stackoverflow.com. Was helpful from time to time in finding solutions to small problems we came across. Accessed on 24 September 2012.

[8] The SVG specification. http://www.w3.org/TR/SVG/. Accessed on 12 September 2012.

[9] W3schools online ajax tutorial. http://www.w3schools.com/ajax/default.asp. Accessed on 4 October 2012.

[10] W3schools online javascript tutorial. http://www.w3schools.com/js/default.asp. Accessed on 16 September 2012.

[11] W3schools online SVG tutorial. http://www.w3schools.com/svg/default.asp. Accessed on 16 September 2012.

[12] Roger Bivand, with contributions by Hisaji Ono, and Richard Dunlap. *classInt: Choose univariate class intervals*, 2012. R package version 0.1-18.

[13] Original S code by Richard A. Becker and Allan R. Wilks. R version by Ray Brownrigg. Enhancements by Thomas P Minka ¡surname@stat.cmu.edu¿. *maps: Draw Geographical Maps*, 2012. R package version 2.2-6.

[14] R. Gentleman, Elizabeth Whalen, W. Huber, and S. Falcon. *graph: graph: A package to handle graph data structures*. R package version 1.34.0.

[15] Jeff Gentry, Li Long, Robert Gentleman, Seth, Florian Hahne, Deepayan Sarkar, and Kasper Hansen. *Rgraphviz: Provides plotting capabilities for R graph objects*. R package version 1.34.2.

[16] Timothy H. Keitt, Roger Bivand, Edzer Pebesma, and Barry Rowlingson. *rgdal: Bindings for the Geospatial Data Abstraction Library*, 2012. R package version 0.7-19.

[17] Duncan Temple Lang. *RCurl: General network (HTTP/FTP/...) client interface for R*, 2012. R package version 1.91-1.1.

[18] Duncan Temple Lang. *RJSONIO: Serialize R objects to JSON, JavaScript Object Notation*, 2012. R package version 1.0-1.

[19] Nicholas J. Lewin-Koh, Roger Bivand, contributions by Edzer J. Pebesma, Eric Archer, Adrian Baddeley, Hans-Jrg Bibiko, Jonathan Callahan, German Carrillo, Stphane Dray, David Forrest, Michael Friendly, Patrick Giraudoux, Duncan Golicher, Virgilio Gmez Rubio, Patrick Hausmann, Karl Ove Hufthammer, Thomas Jagger, Sebastian P. Luque, Don MacQueen, Andrew Niccolai, Tom Short, Greg Snow, Ben Stabler, and Rolf Turner. *maptools: Tools for reading and handling spatial objects*, 2012. R package version 0.8-16.

[20] Julien MOEYS and contributions from Wei Shangguan. *soiltexture: Functions for soil texture plot, classification and transformation*, 2012. R package version 1.2.9.

[21] Paul Murrell. Voronoi treemaps in r. `http://www.stat.auckland.ac.nz/~paul/Reports/VoronoiTreemap/voronoiTreeMap.html`. Accessed on 24 September 2012.

[22] Paul Murrell. *R Graphics*. Chapman & Hall/CRC, 2006. Part of the Computer Science and Data Analysis series.

[23] Paul Murrell. *gridGraphviz: Drawing Graphs with Grid*, 2012. R package version 0.1.

[24] Paul Murrell. *gridSVG: Export grid graphics as SVG*, 2012. R package version 0.9-1.

[25] Paul Murrell and Velvet Ly. *gridDebug: Debugging Grid Graphics*, 2012. R package version 0.4-0.

[26] Erich Neuwirth. *RColorBrewer: ColorBrewer palettes*, 2011. R package version 1.0-5.

[27] Deborah Nolan and Duncan Temple Lang. Interactive and animated scalable vector graphics and R data displays. *Journal of Statistical Software*, 46(1):1–88, 2012.

[28] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[29] Virgilio Gomez-Rubio Roger S. Bivand, Edzer J. Pebesma. *Applied spatial data analysis with R*. Springer, New York, 2008.

[30] Deepayan Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer, New York, 2008. ISBN 978-0-387-75968-5.