Web interactive plots in **R**

Abstract

Approaches for creating web-interactive data visualisations have gained momentum in the present day. However many of these existing tools are limited in achieving customised interactions. By using more flexible tools, we aim to potentially create more impactful and fluid interactions for users to help them explore data from their own perspective.

Contents

List of Figures

Chapter 1

Introduction

The purpose of this report is to investigate current solutions for creating web interactive data visualisations in R.

1.1 The need for interactive graphics

TODO: Introduction is still weak. Needs more resources to back up.

Interactive graphics have become popular in helping users explore data freely and explain topics to a wider audience. As Murray (2012) suggests, static visualisations can only 'offer pre-composed 'views' of data', whereas interactive plots can provide us with different perspectives. To be able to interact with a plot allows us to explore data and discover trends and relationships that cannot be seen with a static graph. The power of interactive graphics can aid us during exploratory data analysis, to which we can display data and query it to answer specific questions the user has (Cook and Swayne, 2007).

The term "interactive graphics" can have different meanings. Theus(1996) and Unwin(1999) have suggested that there are 3 broad components: querying, selection and linking, and varying plot characteristics. We can also split it into two broader categories - 'on-plot' and 'off-plot' interactivity. We focus on 'on-plot' interactivity, where a user can interact directly on the plot to query, select and explore the data.

R(Ihaka and Gentleman, 1996) is a powerful open source tool for generating flexible static graphics. However, it is not focused on interactivity. Previously, there have been different programs to help create interactive plots to aid analysis including ggobi(Swayne et al, 2007), cranvas(Xie et al, 2013), iplots(Urbanek, 2007), and Mondrian(Theus, 2013). Despite their capabilities, all these require installation of software which makes it difficult to share results. More recently, new visualisation tools have begun to use the web browser to render plots and drive interactivity.

1.2 The web and its main technologies

The web provides an ideal platform for communicating and exchanging information in the present day. It has become accessible to everyone without the worry of technical issues such as device compatibility and installation. We find web interactive visualisations are commonly used in teaching statistics, education, data journalism and is likely to continue to be demanded for in the future.

The main web technologies are HTML, CSS and JavaScript. Hyper Text Markup Language (known as HTML) is the language used to describe content on a webpage. Cascading style sheets (known as CSS) is the language that controls how elements look and are presented on a web page (such as color, shape, strokes and fills, borders). These can be used to define specific elements on the page. JavaScript is the main programming language for the web, which is used to drive interactions on the web browser(reference!). Whenever we interact with a website that has a button to click on or hover over text, these are driven by JavaScript.

TODO: The Document Object Model...

Many interactive visuals on the web are generally rendered using Scalable Vector Graphics (known as SVG). This XML based format is widely used because it is easy to attach events and interactions to certain elements through the DOM. This cannot be done with a raster image, as a raster image (PNG or JPEG) is treated as an entire element.

1.3 Motivational problem

The main motivation for this project stemmed upon whether there is an easier and intuitive way to generate simple interactive visuals from R without learning many tools. This solution could ideally be used with iNZight, a data visualisation software from the University of Auckland that aims to teach introductory statistics.

The approach is to identify and assess existing tools for creating web interactive visuals in R before building a viable solution that could solve this problem.

Chapter 2

Simple tools for achieving interactive plots in R

Existing tools for creating standard web interactive plots in R can generally be classified as a class of R packages known as **htmlwidgets**. For tools that do not follow that class, the **ggvis** and the **shiny** package are popular alternatives where interactivity is driven in R. We assess these tools and identify their strengths and limitations.

2.1 htmlwidgets

An 'htmlwidget' is an R package that allows users to have access to an existing JavaScript library through bindings between defined R functions and the JavaScript library (RStudio, 2014). These packages can serve different purposes depending on what the original javascript library does. These include popular graphing libraries such as Highcharter(Kunst, 2017) and rbokeh(Hafen, 2014), DataTable(Xie, 2016) that generates interactive tables, and Leaflet(Cheng and Xie, 2017) for interactive maps.

The main package that we have looked at in detail is `plotly` as it has focused on incorporating interactivity on a wide range of plots and is compatible with R packages `shiny` and `crosstalk` (more details are discussed below). The advantages of using an htmlwidget is that it is easily shareable in a standalone sense, where we can save the plot for later use.

2.1.1 plotly

Plotly is a graphing library that uses the Plotly.JS API (Plotly Technologies, 2017) that is built upon D3 (Bostock, 2011). It is powerful in the sense that it can convert plots rendered in ggplot2 into interactive plots. It provides basic interactivity including tooltips, zooming and panning, selection of points, and subsetting groups of data as seen in Figure 1. We can also create and combine plots together using the `subplot()` function, allowing users to create facetted plots manually.

Figure 1: plotly plot of the iris dataset

By considering HTMLwidgets alone, we find that these solutions only provide interactive plots quickly to the user with basic functionalities such as tooltips, zooming and subsetting. We can build upon layers of plot objects but they cannot be pulled apart or modified without replotting. It is hard to customise interactions as the functions that create these plots are well defined unless we know the original JavaScript API well. These plots do not provide more information about the data or can be linked to any other plot, but this can be achieved by combining these widgets with crosstalk.

2.1.1.1 Extending interactivity between HTMLwidgets with crosstalk

Crosstalk (Cheng, 2016) is an add-on package that allows HTMLwidgets to cross-communicate with each other. As Cheng (2016) explains, it is designed to link and co-ordinate different views of the same data. Data is converted into a 'shared' object (via V6), which has a corresponding key for each row observation. When selection occurs, crosstalk communicates what has been selected and the bounded HTMLwidgets will respond accordingly. This is all happens on the browser, where crosstalk acts as a 'messenger' between HTMLwidgets.

Figure 4: Linked brushing between two Plotly plots and a data table

Figure 5: A ggplot2 (rendered as Plotly) scatterplot matrix linked together with crosstalk

Figure 6: Filtering in groups

However, crosstalk has several limitations. As Cheng (2016) points out, the current interactions that it only supports are linked brushing (Figure 4) and filtering (Figure 6) that can only be done on data in a 'row-observation' format. This means that it cannot be used on aggregate data such as linking a histogram to a scatterplot, as illustrated in Figure 5. When we select over points over the scatterplot matrix, the density curves do not change. Furthermore, it only supports a limited number of HTMLwidgets so far - Plotly, dataTable and Leaflet. This is because the implementation of crosstalk is relatively complex. From a developer's point of view, it requires creating bindings between crosstalk and the HTMLwidget itself and customizing interactions accordingly on how it reacts upon selection and filtering. Despite still being under development, it is promising as other HTMLwidget developers have expressed interest in linking their packages with crosstalk to create more informative visualisations.

2.2 ggvis

Another common data visualisation package is ggvis (Wickham, Chang 2014). This package utilises the Vega JavaScript library to render its plots and uses Shiny to drive some of its interactions. These plots follow the "Grammar of Graphics" and aim to be an interactive visualisation tool for exploratory analysis. This package has an advantage over htmlwidgets as it expands upon using statistical functions for plotting, such as layer_model_predictions() for drawing trendlines using statistical modelling (see Figure 3). Furthermore, because some of the interactions are driven by Shiny, we can add 'inputs' that look similar to Shiny such as

sliders and checkboxes to control and filter the plot, but also have the power to add tooltips as seen in Figure 2.

Figure 2: basic ggvis plot with tooltips

Figure 3: Change a trendline with a slider and filters using ggvis alone

However, while we are able to achieve indirect interaction, we are limited to basic interactivity as we are not able to link layers of plot objects together. The user also does not have control over where these inputs such as filters and sliders can be placed on the page. To date, the ggvis package is still under development with more features to come in the near future. With ggvis, we can go further by adding basic user interface options such as filters and sliders to control parts of the plot, but only to a certain extent.

When combining different views of data together, we cannot do this with ggvis and htmlwidgets alone. Fortunately, interactivity can be extended with these packages by coupling it with Shiny.

2.3 Shiny

Shiny (RStudio, 2012) is an R package that builds web applications. It provides a connection of using R as a server and the browser as a client, such that R outputs are rendered on a web page. This allows users to be able to code in R without the need of learning the other main web technologies. A Shiny app can be viewed links between 'inputs' (what is being sent to R whenever the end user interacts with different parts of the page) and 'outputs' (what the end user sees on the page and updates whenever an input changes). There are many different ways to use Shiny to create more interactive data visualisations - we can simply just use Shiny to create interactive plots or extend interactivity in HTMLwidgets and other R packages.

Figure 7: A simplistic Shiny app

The main advantage of using Shiny is that it establishes a connection to R to allow for statistical computing to occur, while leaving the browser to drive on-plot interactions. This allows us to be able to link different views of data easily. Furthermore, RStudio has provided ways to be able to host and share these shiny apps via a Shiny server. However, we are still limited in the sense that for every time we launch a Shiny app, we do not have access to R as it runs that session. Additionally, Shiny has a tendency to rerender entire objects whenever an 'input' changes as seen in Figure 7. This may lead to unnecessary computations and may slow down the experience for the user. Despite this, it remains a popular tool for creating interactive visualisations.

2.3.1 Interactivity with shiny alone

Shiny can provide some interactivity to plots. Figure 8 shows linked brushing between facetted plots and a table. With Shiny, we are able to easily link plots together with other objects. This is done simply by attaching a 'plot_brush' input, and using the brushedPoints() function to return what has been selected to R. As we select on parts of the plot, we see this change

occur as the other plot and the table updates and renders what has been selected. Other basic interactions include the addition of clicks (plot_click) and hovers (plot_hover).

Figure 8: Facetted ggplot with linked brushing and hovers

However, these basic interactive tools only work on base R plots or plots rendered with ggplot2 and work best on scatter plots. This is because the pixel co-ordinates of the plot are correctly mapped to the data. When we try this on a lattice plot as seen below in Figure 9, this mapping condition fails as the co-ordinates system differs between the data and the plot itself. It is possible to create your own mappings to a plot or image, however it may be complex to develop.

Figure 9: A lattice plot that fails to produce correct mapping

With Shiny alone, we can achieve some basic interactivity along with user interface options that are outside of the plot. Despite being limited to plot interactions (clicks, brushes and hovers), we can link these plot interactions to other parts that may give us more information about the data. However, these methods only work for plots that are rendered in base R graphics and ggplot2, and cannot be extended onto other R plots. Because the plots displayed are in an image format, we can only view these plots as a single object and cannot pull apart elements on the plot. We are unable to further extend and add onto a plot, such as add a trendline when brushing or change colors of points when clicked on.

2.3.2 Linking plotly or ggvis with Shiny

Although Shiny is great at facilitating interactions from outside of a plot, it is limited in facilitating interactions within a plot. It does not have all the capabilities that plotly provides. When we combine the two together, more interaction can be achieved with less effort.

Figure 10: a Shiny app with a Plotly plot with linked brushing

Most HTMLwidgets and ggvis have their own way of incorporating plots into a Shiny application. In Figure 10, we can easily embed plots into Shiny using the `plotlyOutput()` function. The plotly package also has its own way of co-ordinating linked brushing and in-plot interactions to other Shiny outputs under a function called `event_data()`. By combining it with Shiny, we are able to link different plots together and to the data itself that is displayed as a table below.These in-plot interactions are very similar to what Shiny provides for base plots and ggplot2. They work well on scatter plots, but not on other kinds of plots that plotly can provide. These can help generate or change different outputs on the page, but not within themselves. By combining the two together, we get 'on-plot' functionalities from the HTMLwidget, with 'off-plot' driven interactions from Shiny. Similarly, when we can combine ggvis and Shiny together we get similar results as seen in Figure 11. ggvis has its own functions (`ggvisOutput()` and `linked_brush()`) that allow for similar interactions to be achieved.

Figure 11: an example of linked brushing between ggvis plots

2.3.3 A comparison to non-web interactive tools

The interactions that Shiny achieves are not interactions on the plot itself, but rather an interaction driven outside of the plot that causes it to rerender. With HTMLwidgets and ggvis, we are unable to easily customize our own interactions into the plot. Furthermore, there are complications on trying to link different plots together. Many of the tools discussed above only provide co-ordinated links to scatter plots. This is a major disadvantage in comparison to older non-web interactive tools such as iplots and Mondrian.

iPlots is an R package designed to produce interactive graphs for exploratory data analysis using Java. The main features that are included are querying, highlighting and color brushing. It supports a wide range of different graphs including mosaic plots, parallel plots, box plots and histograms. It is easy to modify plots generated as we can easily track what plot objects are added or removed through an object list.

Figure 12: Several windows of iPlots that all link together

In Figure 12, we can generate several windows of plots that are seamlessly linked together. When we select on a scatterplot, the selection will appear on any of the other plots including box plots, histograms and bar plots. Furthermore, we can query back the selection to find out the data. The performance and efficiency of these plots is uncompromised when dealing with large datasets.

However, both tools require installation of software. Both iPlots and Mondrian require Java in order to run its interface and produce its plots. Despite being able to handle large queries and seamlessly link many plots together, it does not look visually appealling which may deter users to use something with a more modernistic feel like Shiny. Furthermore, the results generated cannot be easily to shared with others. With these tools, we find that there is a gap between existing tools on the web.

After assessing these tools, it is difficult to expand on-plot interactions as it requires an in-depth knowledge of the tool itself and how these interactions are defined. Interactivity and querying on large datasets is also another issue in which many developers have suggested using webGL and canvas to render elements rather than SVG for efficiency and performance. We can easily use these tools for ease of visualisation and for achieving standard interactive plots, but if the user wishes to customise interactivity or extend it further, it presents a 'dead end'. Next, we look at how we could achieve specific on-plot interactions by combining JavaScript with lower levels tools including gridSVG, shiny and DOM.

Chapter 3

Stretching limitations with 'lower level' tools

Interactions discussed above can be achieved by R users without the knowledge of HTML, CSS and JavaScript. However, we find that more can be achieved if we have some knowledge of these main web technologies. In this section, we discuss more flexible tools that may help us combat these limitations.

3.1 gridSVG

gridSVG (Murrell and Potter, 2017) is an R package that allows for the conversion of grid graphics in R into SVG (Scalable Vector Graphics). This is powerful because in most cases (including plotly and ggvis), plots are rendered in SVG which makes it easy to attach interactions to specific elements on the page. The advantage of using gridSVG over others is that there is a clear mapping structure between data and svg elements generated. This is not present in htmlwidgets and their JavaScript libraries, which makes it hard to identify or trace data back to the elements and vice versa. This also explains why it may be complex to customise interactions on the plot. With gridSVG, we can easily add simple javascript to grid elements in R using `grid.script()` and `grid.garnish()`, or attach an external javascript file to it after the SVG plot has been generated as we can easily identify which elements to attach interactions to.

Figure: 3.1 simple conversion of a grid plot to an SVG using gridSVG

3.1.1 Customising simple plot interactions

A clear limitation that is present in the existing tools discussed previously allowing the user to customise and make their own interactions on the plot.

Figure 3.2: An example of a customised box plot interaction on an iNZight plot using gridSVG, JavaScript

One such example is highlighting part of a box plot to show certain values between the median and the lower quartile (Figure 13). While this can be easily achieved with more flexible and lower level tools such as gridSVG and custom JavaScript, it is not with the existing tools such as plotly or ggvis. Despite plotly and ggvis rendering graphs in SVG, we are unable to easily identify which elements to target and add interactions to.

The limitations of this package are clear by its name: only plots that are defined by the `grid` graphics system can be converted into SVG. This means that plots defined in base R cannot be directly converted. A solution to this is the `gridGraphics` package which acts as a translator by converting base R graphics into `grid` graphics. Another point to note is that the process of converting elements to SVG becomes slow if there are many elements to render, which suggests it is not suitable for rendering large data sets.

3.1.2 Preventing redraws in Shiny using JavaScript messages and gridSVG

When we interact with a web page, most of these interactions are driven by JavaScript. Shiny provides a way of sending messages from the browser back to R through two different functions: `shiny.onInputChange()`, `Shiny.addCustomMessageHandler`.

TODO: explain how shiny allows sending between browser + R

This gives us a way of targeting certain elements through Shiny when it is not necessary to redraw the entire plot, especially when it comes to plots with many data points. An example of this is the alteration of a trendline using the slider.

Figure 14: A replica of Figure 7, but only the trendline changes

When we compare Figure 13 back with Figure 7. In Figure 7, the plots are rendered using PNG format and as a single image to which whenever an input is changed (such as whenever the slider moves), the plot is recomputed back in R before being sent back as an image to the browser. In Figure 14, because the plot is rendered as an SVG, we can target only the element that renders the trendline and recompute its coordinates whenever the slider changes.

This solves the problem of being only limited to base plots and ggplot2, as now we can render grid graphics and achieve the same effect, but also customise our own interactions while maintaining a connection between R and the browser using Shiny. However, the main limitation that we face when we render in SVG is that the DOM cannot handle too many SVG elements at once. This compromises the performance and efficiency, thus we are limited to smaller datasets. This is a general problem that occurs across all existing tools. Furthermore, it requires the knowledge of JavaScript and the limitations of how much information can be sent through this 'channel' are unknown as it is uncommonly used.

Figure 15: Plot a smoother and return selected values

To stretch this example further, we added in a feature where the user can highlight over a set of points. When highlighted, we return the information about these points in order to further compute a smoother over these points. To achieve this in shiny, we can add in an external javascript file which can be read and added to the web page.

3.2 An alternative to shiny - DOM package

The DOM package (Murrell, 2016) is an R package that allows for DOM requests to be sent from R to a browser. It aims to provide a basis for using the web browser as an 'interactive output device'. It is similar to Shiny as it establishes a connection between R and the browser, however it requires the user to know about the Document Object Model (DOM) and the core web technologies involved. To compare it to Shiny, we have replicated Figure 7 using DOM.

Figure 16: DOM example of Figure 14 - changing a trendline

The advantages of using DOM over Shiny as seen in this example is that we have control over the entire page. From a developer's perspective, we can continue to modify elements on the page with ease. This also allows users to still have access to R while the page is running. We can also run a number of interactive web pages in a single R session. In Shiny we are unable to use R in a single session or be able to change it without stopping the application. This creates a way of being able to receive information of what has been interacted with back to R.

However, there are many limitations with this package. As this package is still developmental, only part of the DOM API has been expanded, and the connection between R and the browser requires extra attention. Murrell (2016) states that it can only be run locally and is aimed at a single user rather than multiple users.

In both cases, these interactions cannot be achieved without being able to write JavaScript. gridSVG, DOM and Shiny provide ways in which we can bind custom JavaScript to elements, but requires the user to be able to define what kind of interactions they wish to achieve.

From investigating existing tools, there is a tradeoff between existing tools. It is possible to customise interactions on existing plots, but this requires a knowledge of JavaScript in order to do so. Comparatively, tools that provide standard interactive web plots are easier to use but are complex to modify and extend further.

Chapter 4

Designing a more flexible way of producing simple interactions

By using gridSVG, DOM and JavaScript, we can customise interactions on plots. However, these are too specific and assumes a lot of knowledge from the user. We need a way to provide 'generic' interactions that can be easily customised and defined by the user without a steep learning curve.

4.1 The main idea

In each of the previous examples defined, they all have a certain pattern. In order to define a single interaction, it requires the need to know which SVG element to target, what type of interaction or event is to be attached, and how to define what happens when an interaction occurs. This idea can be broken down into 5 simple steps: - Draw the plot in R - Identify elements to interact with - Define interactions and process, which elements to target - Attach interactions and define events - Send defined interactions and plot to the browser

[TODO: include main idea diagram]

4.2 Introducing the interactr package

To demonstrate this idea, we have created the `interactr` package which builds upon three main packages: DOM, grid and gridSVG. It is designed to allow users to define their own interactions to plots in R, without a full understanding of the web and these lower level tools.

TODO: to explain more thoroughly? Or not?

4.2.1 Relationship between grid and gridSVG objects

4.2.2 Using DOM to handle events and send to the browser

To put the idea into practice, the following examples have been replicated using the `interactr` package.

4.3 Examples

4.3.1 Linking box plots

The goal in this example is to link the interquartile range of the box plot to a scatter plot, followed by a density plot. When the user clicks on the box plot, it highlights the range of the box plot on the other respective plots.

Our first step is to draw the box plot in R.

```
library(lattice)
bw <- bwplot(iris$Sepal.Length, main = "boxplot")
```

Here, we have stored the boxplot object into a variable called 'bw'. In order to attach interactions, we need to identify what elements have been drawn. We can do that easily by listing the elements.

```
listElements(bw)
```

This will print and return a list of all the elements that make up the box plot in R. Here, the user can identify which element to target and refer to in order to attach interactions.

Next, we can define a simple interaction. We want to achieve an interaction where when the user hovers over the box, it will turn red. In the case of a 'hover', we have defined it as a type of interaction to which we can specify the 'attributes' and styles of the box (here, we have made it so that it turns red when hovered).

```
box <- "plot_01.bwplot.box.polygon.panel.1.1"
interactions <- list(hover = styleHover(attrs = list(fill = "red",
                                                     fill.opacity = "1")))
```

Note that the interaction has only been defined, but not attached yet. Finally, in order to view our plot in the web browser, we send the plot and our interactions.

```
draw(bw, box, interactions, new.page = TRUE)
```

In this step, we have sent the plot we drew in R to a new web page. We can also attach an interaction to the `box` element we identified earlier to attach the hover interaction to. On the page, we see that when the user hovers over the box, the box turns red.

INSERT FIGURE!

Before we move on to drawing the scatter plot, we need to make sure we identify the interquartile range of the box plot and extract any other information we may require from the plot before moving onto the next.

This is one of the disadvantages of this process - if there is anything

Here, we can return the range of the box plot and store it in a variable called `range`.

```
range <- returnRange(box)
```

We proceed to add a scatter plot by drawing the scatter plot, listing the elements and identifying the 'points', before sending it to the same web page.

```
sp <- xyplot(Sepal.Width ~ Sepal.Length,
             data = iris,
             main = "scatterplot")
listElements(sp)
points <- "plot_01.xyplot.points.panel.1.1"
draw(sp) #by default, new.page = FALSE
```

Here, we see that the boxplot and the scatterplot we drew in R are now on the same web page.

[FIGURE]

To highlight the points in the scatterplot that lie in the range of the box, the user can define the function as follows. We can determine the index of the points that lie within the range of the box, and then pass that index through a function called `setPoints` to highlight these in red and group them together in a class called `selected`.

```
highlightPoints <- function(ptr) {
  #identify index
  index <- which(min(range) <= iris$Sepal.Length
                 & iris$Sepal.Length <= max(range))
  #identify points and set them to red
  setPoints(points,
            type = "index",
            value = index,
            attrs = list(fill = "red",
                         fill.opacity = "1",
                         class = "selected"))
}
```

This function can be easily modified by the user and requires them to make the connection between the data they are dealing with (in this case, the iris data). As we have defined this interaction, we need to define the event to call this interaction to before we can send it to the browser.

```
boxClick <- list(onclick = 'highlightPoints')
addInteractions(box, boxClick)
```

Here, we insert the function name to run when a 'click' is done. Next, we append this defined

interaction to the box element, so that when we click on the box, the points in the scatterplot that lie within that range should light up in red.

insert Figure here

This example can be further extended by linking the box plot to both a scatter plot and density plot. Here, we have taken some census data and wish to find out the density of girls who have the heights that lie within that interquartile range of the boys heights.

We begin by drawing the box plot of boys heights.

```
bw <- bwplot(boys$height, main = "Boxplot of boys' heights")
bw.elements <- listElements(bw, "boys_height")
box <- "boys_height.bwplot.box.polygon.panel.1.1"
interactions <- list(hover = styleHover(attrs = list(fill = "red",
                                                     fill.opacity = "0.5",
                                                     pointer.events = "all")))
draw(bw, box, interactions, new.page = TRUE)
range <- returnRange(box)
```

Next, we add the scatterplot to the page.

```
sp <- xyplot(boys$armspan ~ boys$height,
             main = "Height vs armspan (boys)",
             xlab = "Height(cm)",
             ylab = "Armspan")
sp.elements <- listElements(sp, "sp_bheight")
points <- "sp_bheight.xyplot.points.panel.1.1"
draw(sp)
```

Then, we add the density plot of girls heights.

```
dplot <- densityplot(~girls$height,
                     main="Density plot of girl's heights",
                     xlab="Height(cm)")
d.elements <- listElements(dplot, "girls_height")
dlist <- list(points = "girls_height.density.points.panel.1.1",
              lines = "girls_height.density.lines.panel.1.1")
draw(dplot)
```

[FIGURE]

In order to highlight a certain region of the density plot, we need to add a new element to the page. This can be done using the `addPolygon` function. Ideally, it should be added to the same group as where the density lines are located. We can use the `findPanel` function to identify the correct viewport to attach to.

```
# add invisible polygon to the page:
panel <- findPanel(dlist$lines)
addPolygon("highlightRegion", panel, class = "highlight",
```

```
            attrs = list(fill = "red",
                         stroke = "red",
                         stroke.opacity = "1",
                         fill.opacity= "0.5"))
```

Note that this will be invisible to the page as we have not defined the coordinates of the region. We only want this to appear when the user has clicked on the box plot.

Here we write a function that defines what happens after the box plot is clicked. We identify which the coordinates of the density line lie within the range of the box plot. This can be used to define the points of the region that we wish to highlight. We can also highlight the points in the scatter plot in the same way as we have done in the previous example.

```
highlightRange <- function(ptr) {

  coords <- returnRange(dlist$lines)
  index <- which(min(range) <= coords$x & coords$x <= max(range))
  xval <- coords$x[index]
  yval <- coords$y[index]

  # add start and end:
  xval <- c(xval[1], xval, xval[length(xval)])
  yval <- c(-1, yval, -1)

  pt <- convertXY(xval, yval, panel)

  #set points on added polygon
  setPoints("highlightRegion", type = "coords", value = pt)

  # highlight points in scatter plot:
  index <- which(min(range) <= boys$height
                 & boys$height <= max(range)
                 & !is.na(boys$armspan))
  # note that if armspans are missing,
  # then it will return 'element is undefined',
  # requires !is.na(boys$armspan) to remove missing values
  setPoints(points,
            type = "index",
            value = index,
            attrs = list(fill = "red",
                         fill.opacity = "0.5",
                         class = "selected"))

}
```

Finally, we simply just define and attach our interactions to the page.

```
boxClick <- list(onclick = "highlightRange")
addInteractions(box, boxClick)
```

When the user now clicks on the box plot, it lights up the points and the density that lie within that range.

FIGURE

4.3.2 Changing a trendline

Another example that can be done is driving an off-plot interaction with a slider. The slider controls the smoothing of the trendline.

Here, it's a bit more complex:

NEED TO WRITE THIS...

4.4 Compatibility with other graphics systems:

A possibility of extending this idea further is the compability with other R graphing systems including **lattice**, **graphics**, and **ggplot2**. As we have seen above, many of these examples are done with **lattice** plots. However, it is possible to achieve the same with different graphing systems. To demonstrate, we have taken the box plot example in Figure 4.2 and replicated it in graphics and ggplot2.

4.4.1 Using graphics plots

As briefly mentioned in Chapter 3, in order to convert SVG objects using gridSVG the objects must be grid objects. In the case of graphics plots, we cannot directly call gridSVG to convert it into an SVG. A simple solution to this is to use the **gridGraphics** package, which acts as a translator by converting graphics plots into grid plots with a consistent naming scheme. We used `grid.echo()` to achieve this. Another problem that arises is that when we plot or try save it into a variable, it does not plot to the graphics device. To solve this, we can use `recordPlot` to record the plot that has been drawn to further process it.

Simplistically, the only change that we need to do is run an extra `recordPlot` command before we `listElements`.

Once again, we begin by drawing the plot. We then record the plot before listing its elements. This will automatically convert the plot into a grid plot using `grid.echo()` underneath.

```
boxplot(iris$Sepal.Length, horizontal = TRUE)
pl <- recordPlot()
listElements(pl)
```

Next, the same process occurs. We see that the code is very similar to what was done previously with `lattice`.

```
box = "graphics-plot-1-polygon-1"
interactions <- list(hover = styleHover(attrs = list(fill = "red",
                                                     fill.opacity = "1")))
draw(pl, box, interactions, new.page = TRUE)
range <- returnRange(box)

# plot a graphics scatter plot
plot(iris$Sepal.Length, iris$Sepal.Width)
sp <- recordPlot()
listElements(sp)
draw(sp)

#add interactions
points <- 'graphics-plot-1-points-1'
highlightPoints <- function(ptr) {
  #find the index of points that lie within the range of the box
  index <- which(min(range) <= iris$Sepal.Length
                 & iris$Sepal.Length <= max(range))
  setPoints(points,
            type = "index",
            value = index,
            attrs = list(fill = "red",
                         fill.opacity = "1",
                         class = "selected"))
}

boxClick <- list(onclick = "highlightPoints")
addInteractions(box, boxClick)
```

Figure

This shows that there is potential for customising interactions onto graphics plots. Simplistically, the process is the same, except due to the nature of the graphics plot not being a 'grid' plot.

4.4.2 Using ggplot2

ggplot2 (Wickham et al, 2017) is a popular plotting system in R based upon the "Grammar of Graphics". It is built upon the `grid` graphics system, which makes it compatible with gridSVG.

```
library(ggplot2)
p <- ggplot(data = iris, aes(x = "", y = Sepal.Length)) + geom_boxplot()
p.elements <- listElements(p)
box <- findElement("geom_polygon.polygon")
```

```
interactions <- list(hover = styleHover(attrs = list(fill = "red",
                                                     fill.opacity = "1",
                                                     pointer.events = "all")))
draw(p, box, interactions, new.page = TRUE)
```

However, because it works on a completely different co-ordinates system, we cannot simply use the `returnRange` function to define the range of the box.

The native coordinates given by `grid` do not return that data coordinates of the ggplot. A simple solution to this is that the information about the plot can be extracted from `ggplot_build`. Below, we have identified the range of the box manually.

```
#find the range of the box:
boxData <- ggplot_build(p)$data[[1]]
#for a box plot - IQR: lower, upper
range <- c(boxData$lower, boxData$upper)
```

Next, we add the scatterplot to the page, define our interactions before sending it to the browser.

```
sp <- ggplot(data = iris, aes(x = Sepal.Width, y =Sepal.Length)) + geom_point()
sp.elements <- listElements(sp)
draw(sp)
points <- findElement("geom_point.point")

highlightPoints <- function(ptr) {
  #find the index of points that lie within the range of the box
  index <- which(min(range) <= iris$Sepal.Length
                 & iris$Sepal.Length <= max(range))
  setPoints(points,
            type = "index",
            value = index,
            attrs = list(fill = "red",
                         fill.opacity = "1",
                         class = "selected"))
}

boxClick <- list(onclick = "highlightPoints")
addInteractions(box, boxClick)
```

insert figure

Note that the only difference are the tags for how each element is defined. Because ggplot2 does not have a consistent naming scheme like **lattice**, we can use the `findElement` function to allow the user to easily find which element they want to refer to.

This demonstrates that there is a possible way of achieving interactions with ggplot2 and there is potential for the `interactr` to plug into different R plotting systems. Consequently it also means that when we assess the compatibility of different plotting systems, we need to take

into account of possible detours that could occur. Because this is a simplistic example, it may become more complex when we try to achieve more sophisticated interactions.

The `interactr` package acts as a proof-of-concept that aims to be a general solution for adding simple interactions to plots generated in R. There is potential in plugging into different plotting systems, however it depends on how compatible these systems are with the underlying tools of the `interactr` package. It is based upon a very simple process of defining what you want to draw in R, identifying elements drawn, and defining specific interactions to attach to certain elements drawn that can be viewed in a web browser.

Chapter 5

Discussion

The `interactr` package provides a way of generating simple interactive R plots that can be viewed in a web browser. It is advantageous in the sense that we can easily define and customise interactions with flexibility. It can also achieve unidirectional linking between a variety of plots.

TODO: incorporate these - WARNING from Paul about DOM: > "DOM does nothing to help with synchronising cascades of updates (OR infinite loops of updates)"

- requires that units that are converted to 'native' via grid should represent the data. (for ggplot2, this doesn't hold and requires a different conversion scale. In cases like this, there should be an alternative based upon where it gets data from: use `ggplot_build()`)
- assumes no missing values and that plots generated via gridSVG should be in the order of the data frame. (ie point order should match with indices of the df.) In cases where data taken in is rearranged and sorted (like iNZightPlots), this causes the 'indexes' of the points to differ to the original dataframe.
- assumes that most grid objects represent a single object in SVG (which sometimes is not the case - see iNZightPlot boxplot version)

5.1 Limitations

Firstly, there are limitations that arise from using the gridSVG and DOM packages. With gridSVG, one of the major limitations is that only grid objects can be converted into SVG. This limits us to plots that must be drawn in R before being sent to the browser. A further limitation is that gridSVG is relatively slow when we plot many points or objects at once. This can compromise the efficiency when we need to send the plot to the browser. Hence, we are limited to small data sets.

As `interactr` is mainly built upon the `DOM` package, many of its limitations highlighted in Chapter 3 are carried forward. This includes the shareability and accessibility of these plots, where they are generated for a single user in a single session only. Furthermore, because the `DOM` package is still under development, it cannot be used for production purposes yet. Just as

how shiny applications would require a shiny server to be hosted on the web, DOM would require something similar to allow for shareability. This has not been done before, but it could be experimented with in the future. Furthermore, because the underlying system consists of requests being sent between R and the web browser, this can be relatively slower when compared to plots that are driven fully by JavaScript.

There are further limitations within the package. The user must call `listElements` before sending the plot to the browser as it prints the plot and a correct list of elements that make up the plot. This is crucial for plotting systems that do not have a consistent naming scheme. If we reprint the plot, the tags will constantly change which may cause a mismatch between element matching between the plot on the web and the plot in R. A further limitation is that the user will need to deduce which element is on the plot as the naming for these objects in `grid` may not be clear. If it is a plot that is made directly from grid where the user has named everything clearly, then this is not a problem. However, if it is based upon a defined plotting system like ggplot2, then there is a need for deducing which elements correspond to which part of the plot.

Another limitation is the number of interactions that can be attached. So far, the examples expressed in Chapter 4.3 require a single element to be controlled. We can attach many interactions and events to a single element at a time, but not many elements to many different interactions at once. There is a need for a more flexible system when dealing with multiple interactions for achieving more complex interactions. Furthermore, only one kind of interaction can be expressed for a single event. This means that the function created by the user must be fully defined in a single function rather than multiple functions. For example, if a hover requires both adding a tooltip and to turn the element red, then this would need to be written as a single function as we can only attach one to each event. Code must also be written in a certain order to work. Plots in R must be drawn to a graphics device before being sent to the browser, while a new web page must be set up before we can start adding elements and interactions to the page. In cases of dealing with multiple plots, one of the disadvantages is that we lose information about the previous plot in R. This means that the user is required to identify what kind of information they need to extract before they move onto the next plot. This is demonstrated in the example of linking box plot to other plots together - before the user can move onto the scatter plot, the range of the box and viewports were stored in order to be used in the defined function. This means that we cannot jump back and forth between plots. A possible solution for this is to be able to store the information about each plot that is sent to the web browser that can be retrievable by the user if needed.

5.2 Comparison to existing tools

In most of the existing tools discussed in Chapters 2, they all rely on a JavaScript library to render their plots. These are not R plots. `interactr`'s main selling point is the ability to replicate plots or objects drawn in R and to easily achieve on-plot and off-plot interactivity. `shiny` can do this but you cannot easily attach specific interactions as the plot is rendered as a png. Furthermore, many of these existing tools rely on the `shiny` framework. One of the major disadvantages that `shiny` possesses is a tendency to recompute and redraw

entire plots whenever an input changes. In `interactr`, only part of the plot that the user specifically targets is modified, and more on-plot interactions can be achieved. Furthermore, the `interactr` package provides a possible way of linking different types of plots together, whereas existing tools, specifically crosstalk, have focused on linked brushing between scatter plots. To put in perspective, the simple example of linking box plots to other kinds of plots together (Figure 4.2) is an interaction that is difficult to achieve without expert knowledge of their respective APIs.

TODO: consider these - Would be difficult to refute ggvis - since it's got a more simpler API, and it's one of the few that does not do redrawing on plots + can do the trendline/slider challenge in more simpler steps, but cannot do the box plot example - could mention that in Shiny, whenver they redraw, axes will automatically change. In `interactr` axes don't change, so it's a 'fair' comparison (however, if it requires changing, then we've got a limitation)

Because many of these existing tools are still being developed, it is likely that they may resolve these limitations in the future. However, some tools still require the user to be very familiar with its API. An example of this is the `plotly` package that has expanded further into achieving linking between other types of plots without the use of crosstalk or shiny and the ability to prevent redrawing when used with `shiny`.

5.3 Future directions and further problems

The problem of handling large data sets is still present simply because the browser cannot handle too many SVG elements at once. A solution is to render using webGL and HTML canvas environments which allow for many elements to be rendered without compromising speed. However, the problem with this is that it is not as straightforward to attach events to thses elements as they are simply treated as a single element.

There is potential in developing `interactr` further to try achieve more sophisticated interactions that are more useful in exploratory data analysis. Currently only a small number of examples have been successful. There is also a need for making it more simpler and versatile for users, but it could compromise the flexibility in which the user can define interactions. Possible ideas would include integrating plots with **D3** and other **htmlwidgets** to achieve special effects such as zooming and panning of a plot and to achieve multi-directional linking.

Chapter 6

Conclusion

There is more that can be achieved in expanding interactive graphics to create better data visualisations for users. Despite having many tools available such as HTMLwidgets and Shiny, we find that they all achieve similar results without allowing the user to further extend or customize unless they have knowledge of the main web techologies involved. Options are even more limited when it comes to dealing with large datasets, querying or linking plots which are handled effortlessly with non-web solutions. There is potential in creating more meaningful web visualisations for the future.

Note that this conclusion was written for the first half.

- Many existing tools are relatively new (still developing) and produce 'standard' interactive plots outside of R.
  - There is a possible solution to allow control and creation of simple interactive R plots that can be viewed in a web browser.
  - Simple on-plot interactivity can be achieved + a non-RStudio way of driving interactions without the need for learning 'lower level tools' + web tech
  - Utilises R's power of statistical computing to aid changes in graphical plots
  - More assessment required on building more sophisticated interactive visualisations + looking beyond "SVG" (. . . maybe it might be better to stick to JavaScript for the web?)

6.1 Additional resources

The `interactr` package is currently hosted on Github here. For more details about this project, visit this repository which contains codes, findings, other information. To view the interactive version of this report to see all the examples live, click here.

Need to add links!

Missing a bibliography!