Web interactive plots in R

Yu Han Soh

October 16, 2017

Bachelor of Science (Honours)
Department of Statistics
The University of Auckland
New Zealand

Abstract

Web interactive graphics have become popular for sharing exploratory data analysis. There are many approaches for creating web interactive graphics, however, they are limited and do not allow users to customise certain interactions. This report gives an overview of existing tools for creating web interactive plots before developing and discussing a more flexible solution called interactr, a prototype package that allows users to customise interactions on plots produced in R and aims to remove the need for understanding how web technologies work.

Executive Summary

Interactive statistical graphics have been achieved through desktop applications since the 90's, however they are generally inaccessible to users and require special software to be installed. Results are hard to reproduce and share. Recently, new tools have focused on using the web as a platform to solve this but they do not possess the capabilities that these desktop applications have.

The purpose of this research is to make progress towards designing and prototyping a more extensible infrastructure for creating web interactive graphics in R. The motivation behind this research comes from the idea of creating interactive plots with iNZight, a data visualisation software from the University of Auckland.

An overview of modern web tools were investigated including plotly, ggvis, shiny and animint. It is easy to achieve certain interactions, but hard to extend beyond their capabilities without a deeper understanding of these packages and lower level coding. This makes it inaccessible to the majority of users. Furthermore, many online systems have a tendency to redraw everything every time any graphical element is changed. This leads to unnecessary computations and a slow experience to users.

A different approach was taken by investigating lower level tools, specifically gridSVG and DOM. These tools are extensible, however, to use them effectively requires a knowledge about how the grid system works with gridSVG and web technologies including the Document Object Model. This presents a steeper learning curve than using plotly and ggvis, and consequently a trade off - to achieve custom interactions, a user would be required to know how to link all these tools together, where as other tools are easier to use but cannot be extended further.

To solve this, we have developed a new approach by combining lower level tools (grid, gridSVG and DOM) to create the interactr package. This is designed to create simple interactive plots in R without a steep learning curve. It is based upon a simple idea of knowing what object to target, what kind of interaction to attach to which objects and defining what happens after an interaction is initiated. To test this idea, we implemented and recreated simple examples that were compatible with other plotting systems including those made with graphics, lattice, and ggplot2.

The interactr package stands out as it brings interactivity to plots that were originally generated in R. However, it only serves as a proof-of-concept. It presents several limitations including that only objects originally drawn in R can be used and that only a few interactions have been achieved. It is currently not shareable in a multi-user environment nor ready for production purposes.

The future of web interactive statistical graphics remains dynamic as many of these tools are developing over time. It is possible that the interactr package may become a solution for allowing users to control interactions more easily on plots in R and thus for iNZight, but requires more attention and development for creating more sophisticated and stable visuals.

Acknowledgements

Thank you to Dr. Paul Murrell for his technical expertise and witty analogies that helped me understand the concepts and challenges of research and software development. Thank you for being an inspirational role model for becoming a better programmer.

Thank you to Prof. Chris Wild for introducing me to the idea of interactive graphics, for inspiring me to investigate further into this topic for iNZight and for my own interests. Thank you for sharing ideas and challenging me further.

This project would not have been possible without their dedicated supervision and support. I hope that it has helped in their own research and inspires others to have fun making their own interactive visuals in R.

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 before designing a more flexible approach for customising interactions onto plots.

1.1 The need for interactive graphics

Interactive graphics have become popular in helping users explore data freely and explain topics to a wider audience. As Murray (2013) suggests, static visualisations can only ‘offer pre-composed views of data’, where as interactive plots can provide us with different perspectives. To be able to interact with a plot allows us to explore data, 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 and query data to answer specific questions the user has (Cook and Swayne 2007).

The term “interactive graphics” can have different meanings. Theus(1996) and Unwin(1999) (as cited in Unwin, Theus, and Hofmann (2006)) have suggested that there are 3 broad components: querying, selection and linking, and varying plot characteristics. Querying involves finding out more about features that the user may be interested in, selection and linking involves subsetting a certain group and linking to different displays of the same data set, while varying plot characteristics involve changing parts of the plot to get more information which could include “rescaling, zooming, reordering and reshading” (Unwin, Theus, and Hofmann 2006). We can also split it into two broader categories: on-plot and off-plot interactivity. On-plot interactivity refers to when a user can interact directly on the plot to query, select and explore the data. These include clicking and creating drag-boxes to select components of a plot. Off-plot interactivity refers to interactions that are driven outside of the plot, such as a slider to control certain plot characteristics and using dropdown menus to filter and select groups.

Together, these encapsulate the concept of interactive graphics with a certain goal of informing the user of certain patterns and relationships that they may be interested in.

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(Cook and Swayne 2007), iplots(Urbanek and Wichtrey 2013), and Mondrian(Theus 2002). Despite their capabilities, all these require installation of software which makes it difficult to share and reproduce 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 is an ideal platform for communicating and exchanging information in the present day. It has become accessible to everyone without the worries of device compatibility and installation. Web interactive visualisations are becoming more commonly used in areas including data journalism, informative dashboards for business analytics and decision making, and education. These will be continually 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 and cascading style sheets (known as CSS) is the language that controls how elements look and are presented on a web page such as colour, shape, strokes and fills, borders (W3C 2016). These can be used to define how specific types of elements are rendered on the page. JavaScript is the main programming language for the web (Crockford 2008), which is used to add interactivity to web pages. Whenever we interact with a website that has a button to click on or hover over text, these are driven by JavaScript.

The Document Object Model (known as the DOM) is the 'programming interface for HTML and XML documents' (W3C 2009). A single web page can be considered as a document made up of nodes and objects with a certain structure. We can use the DOM to refer to specific elements, attributes and nodes on the page that we wish to modify and get information about using different programming languages, specifically JavaScript. This allows us to create and change a dynamic web page.

Application programming interfaces (APIs) are defined as a set of tools that help developers connect and build applications (Jacobson, Brail, and Woods 2011). For example, when we see a map from Google Maps embedded in a web page, that web page is calling the GoogleMaps API to provide the map. Through the context of this report, APIs generally refer to JavaScript libraries that are called upon and used to render plots.

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 and sub-components 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 arises from whether there is a more intuitive way to generate simple interactive visuals from R without the user having to learn many tools. The solution could ideally be used to advance features in iNZight (Elliott and Kuper (2017)), a data visualisation software from the University of Auckland.

The approach is to identify and assess existing tools for creating web interactive visuals in R (discussed in Chapters 2 and 3). The key limitations that were found were (1) a tendency to reproduce entire plots, (2) the inability to customise interactions and add certain layers to a plot and (3) a need for learning web technologies and respective APIs. This led us to a design and prototype a viable solution (discussed in Chapters 4 and 5) that could potentially solve these limitations and the overall problem.

Chapter 2

An overview of tools for achieving web interactive plots in R

There are many R packages that create different interactive data visualisations. Many of these connect R to specific JavaScript libraries. These include **DT**(Xie 2016) for generating interactive tables, Leaflet (Cheng, Karambelkar, and Xie 2017) for rendering interactive maps and many popular graphing libraries including highcharter(Kunst 2017), rbokeh(Hafen and Continuum Analytics 2016), **googleVis**(Gesmann and Castillo 2011) and the **rCharts**(Vaidyanathan 2013) package. These generate interactive plots or widgets known as htmlwidgets that can be viewed on a web page. Other tools use R rather than JavaScript to drive interactivity, including ggvis(Chang and Wickham 2016) and shiny(Chang et al. 2017). The few that are discussed in this section in detail are the plotly(Sievert et al., n.d.), ggvis, shiny, and **animint**(Hocking et al. 2017) packages.

2.1 plotly

`plotly.js` is a JavaScript graphing library built upon D3 (Bostock, Ogievetsky, and Heer 2011). The plotly package in R calls upon this library to render web interactive plots. The purpose of plotly in R is to provide a convenient way of creating interactive data visualisations (Sievert 2017b). With its API, we can generate a standard plot that can be shared and saved as an interactive HTML web page. One of the reasons the plotly R package is useful is that it can automatically convert plots rendered in the very popular `ggplot2` (Wickham 2016) package into interactive plots by simply applying the `ggplotly()` function to the plot drawn. It provides basic interactivity including tooltips, zooming and panning, selection of points, and subsetting groups of data as seen in Figure 2.1. We can also create and combine plots together using the `subplot()` function, allowing users to create facetted plots manually or combine different sets of types of plots together.

Figure 2.1: plotly plot of the iris dataset

```r
plotly::plot_ly(data = iris, x = ~Sepal.Width,
                y = ~Sepal.Length, color = ~Species,
                type = "scatter", mode = "markers")
```

Like many other htmlwidgets, plotly can provide interactive plots quickly to the user with basic functionalities such as tooltips, zooming and subsetting. In plotly, there are a lot of features for building different plots. However, while We can build upon layers of plot objects, they cannot be pulled apart or modified without re-plotting. These plots natively do not provide more information about the data or be linked to any other plot, but this can be achieved by combining these widgets with crosstalk (Section 2.1.1) or shiny (Section 2.3).

It is difficult to customise interactions without a knowledge of the D3, JavaScript and the use of the `onRender` function from the htmlwidgets package. The other difficulty for the majority of users is knowing which elements to target and how it has been defined on the page. Sievert (2017b) has shown an example of how a set of scatter points drawn with plotly can be selected via clicking which are linked to a google search page.

plotly is constantly being developed. As of writing, it has begun to expand on different methods of linking different views of plots and is able to create animated plots(Sievert 2017b).

2.1.1 Extending interactivity with crosstalk

crosstalk (Cheng 2016) is an add-on package that allows htmlwidgets to 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 R6 'shared' object, which has a corresponding key for each row observation. When selection occurs, crosstalk communicates which keys have been selected and these widgets will respond accordingly. This is all happens on the browser, where crosstalk acts as a 'messenger' between these widgets.

Figure 2.2: Linked brushing between two plotly plots and a data table

```r
#transform our data into a shared object
shared_iris <- SharedData$new(iris)
#generate plots
p1 <- plot_ly(shared_iris, x = ~Petal.Length,
              y = ~Petal.Width, color = ~Species, type = "scatter")
p2 <-  plot_ly(shared_iris, x = ~Sepal.Length,
               y = ~Sepal.Width, color = ~Species, type="scatter")
#layout the plots on the page, along with the data table
p <- subplot(p1, p2)
bscols(
  widths = 12, #need to scale accordingly
  p,
  datatable(shared_iris)
  )
```

In Figure 2.2, we have linked two plots generated by plotly with a table generated by the DT package. When we select over a set of points in one of the plots, the table will respond by filtering all the points that have been selected, while this selection is also highlighted on the other plot. Similarly, if we highlight on the other plot, that selection should change and be updated. This creates a form of multi-directional linking between different views of the iris dataset.

Figure 2.3: Additional filtering and selection tabs using crosstalk

```
shared_income <- SharedData$new(income)
bscols(
  widths = 6,
  list(filter_checkbox("sex", "Gender", shared_income, ~sex, inline  = TRUE),
       filter_slider("weekly_hrs", "Weekly Hours", shared_income, ~weekly_hrs),
       filter_select("ethnicity", "Ethnicity", shared_income, ~ethnicity)),
  plot_ly(shared_income, x = ~weekly_hrs, y = ~weekly_income, color = ~sex, type = "scatter"
)
```

In figure 2.3, crosstalk can also be used for filtering. We can add specific inputs for filtering parts of our data set using sliders, checkboxes, and dropdown menus to allow more control over how we can subset and query our data.

However, crosstalk has several limitations. As Cheng (2016) points out, the current interactions that it supports are only linked brushing (Figure 2.2) and filtering (Figure 2.3) that can only be done on a single data set in a 'row-observation' format. This means that it cannot be used on aggregate data such as linking a density plot to a scatterplot, as illustrated in Figure 2.4 below. When we select over points over the scatterplot matrix, the density curves do not change as it cannot convert the selection into aggregated data.

```
mtcars$cyl <- as.factor(mtcars$cyl)
shared_cars <- SharedData$new(mtcars[1:5])
pl <- GGally::ggpairs(shared_cars, aes(color = cyl))
ggplotly(pl)
```

Figure 2.4: A scatterplot matrix based upon the first five variables in the mtcars dataset

Sievert (2017b) explains that the densities do not update because there are no tools available in the `plotly.js` or in the browser to recompute these densities. However, it may be possible in a client-server framework such as shiny (discussed later in Section 2.3), where we can call upon R to do the calculation. Because the `plotly.js` library recently has support for certain statistical functions that can aggregate data, plotly has expanded beyond being limited to linking between row-observation data. As of writing, these are still being continually developed. One of the main limitations of using crosstalk together with plotly is speed - there is a certain time lag before a user completes their query via selection or clicks (Sievert 2017b).

Crosstalk only supports a limited number of htmlwidgets so far - plotly, DT and Leaflet (Cheng 2016). 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 being under development, it is recognised as being very promising as other htmlwidget developers (notably, Kunst with highcharter(2017) and Hafen(2016) with rbokeh) have expressed interest in linking their packages with crosstalk to create more informative visualisations.

2.2 ggvis

Another common data visualisation package is ggvis (Chang and Wickham 2016). This package utilises the Vega JavaScript library (Trifacta 2014) to render its plots and uses the shiny framework to drive its interactions from R. It aims to be an interactive visualisation tool for exploratory analysis while following the "Grammar of Graphics" from Wilkinson and Leland (Wilkinson 2005), similar to ggplot2 for static plots. It has an advantage over htmlwidgets as it expands upon using statistical functions for plotting, such as `layer_model_predictions()` for drawing trend lines using statistical modelling (see Figure 2.6). Furthermore, because some of the interactions are driven by shiny (Wickham and Chang 2016), 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 2.5.

Figure 2.5: basic ggvis plot with tooltips

```
ggvis(iris, ~Sepal.Width, ~Sepal.Length, fill = ~Species) %>%
    layer_points() %>%
    add_tooltip(function(iris) paste("Sepal Width: ", iris$Sepal.Width, "\n",
                                     "Sepal Length: ", iris$Sepal.Length))
```

Figure 2.6: Change a trendline with a slider and filters using ggvis alone

```
ggvis(cocaine, ~weight, ~price, fill = ~state) %>%
  layer_points() %>%
  layer_smooths(stroke:= "red", span = input_slider(0.5, 1, value = 1, label = "Span of loes
  layer_model_predictions(stroke:="blue",
                          model = input_select(c("Loess" = "loess", "Linear Model" = "lm", "
```

However, while we are able to achieve indirect interactions, we are limited to basic interactivity as we are not able to link layers of plot objects together. The user also does not have finer control over where these inputs such as filters and sliders can be placed on the page. We also cannot save these interactions to a standalone web page as ggvis plots are driven by the shiny framework which requires R. There is an option of saving the plot as a static plot, either as an SVG or PNG format. 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.

We cannot do combine different views of data using ggvis, plotly and other htmlwidgets alone. Interactivity can be extended with these packages by coupling it with shiny.

2.3 shiny

shiny (Chang et al. 2017) 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 application (RStudio 2017c) can be viewed as 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).

To show how this works, we have created a simple shiny application that has a slider that controls the smoothness of the trend line. Whenever the user moves the slider, the plot will be redrawn and updated with a new smoother.

Figure 2.7: A diagram showing how inputs and outputs work

To show how this works, we have created a simple shiny application that has a slider that controls the smoothness of the trend line. Whenever the user moves the slider, the plot will be redrawn with a new smoother.

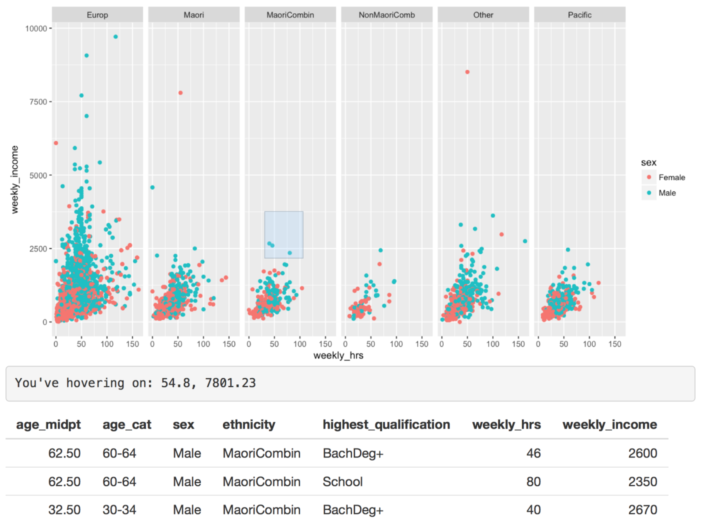
Figure 2.8: A simplistic shiny application that has a slider to control the smoothness of the trend line

These applications can become more complex when more inputs and outputs are added. 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 and off-plot interactions (briefly defined in Section 1.1). This allows us to be able to link different views of data easily. Furthermore, RStudio (2017d) 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 redraw entire objects whenever an 'input' changes as seen in Figure 2.7. This can lead to unnecessary computations and traffic between R and the webpage slows down the experience for the user. Despite this, it remains a popular tool for creating interactive visualisations.

There are many different ways to use shiny to create more interactive data visualisations - we can simply just use it to create interactive plots or we can go further and use it to extend the interactivity in plotly, ggvis and other R packages.

2.3.1 Interactivity with shiny

shiny alone can provide some interactivity to plots (RStudio 2017a). Figure 2.2 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`).

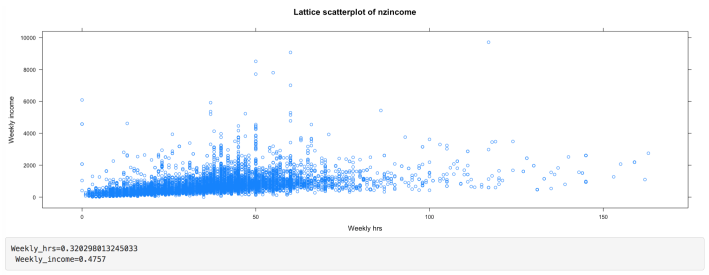


```
You've hovering on: 54.8, 7801.23
```

age_midpt	age_cat	sex	ethnicity	highest_qualification	weekly_hrs	weekly_income
62.50	60-64	Male	MaoriCombin	BachDeg+	46	2600
62.50	60-64	Male	MaoriCombin	School	80	2350
32.50	30-34	Male	MaoriCombin	BachDeg+	40	2670

Figure 2.1: Facetted ggplot with linked brushing and hovers

However, these basic interactive tools only work on base R plots or plots rendered with **ggplot2** and best with scatter plots. It is possible to extend this to bar plots, but it requires more thought. This is because the pixel co-ordinates of the plot are correctly mapped to the data (RStudio 2017b). When we try this on a lattice plot as seen below in Figure 2.2, 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 is complex to develop.



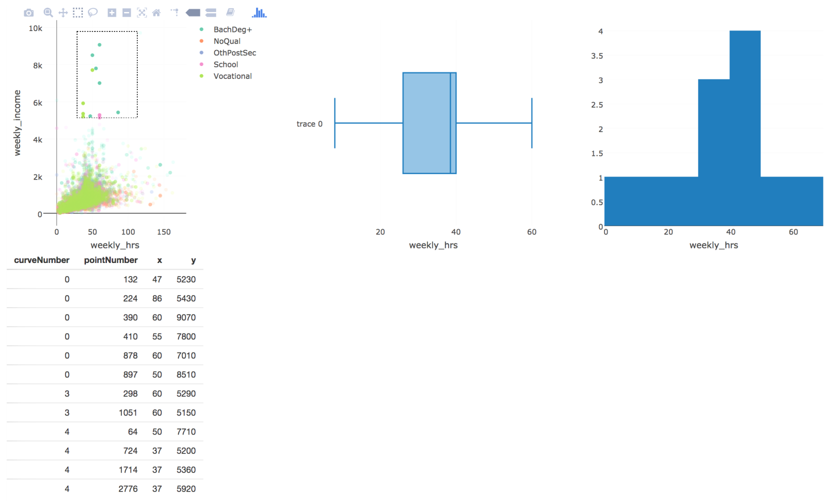
```
Weekly_hrs=0.320298013245033
 Weekly_income=0.4757
```

Figure 2.2: A lattice plot that fails to produce correct mapping

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 trend line when brushing or change colours of points when clicked on. Despite being limited to plot interactions (clicks, brushes and hovers), we can use shiny to link multiple views of the data set.

2.3.2 Linking plotly or ggvis with shiny

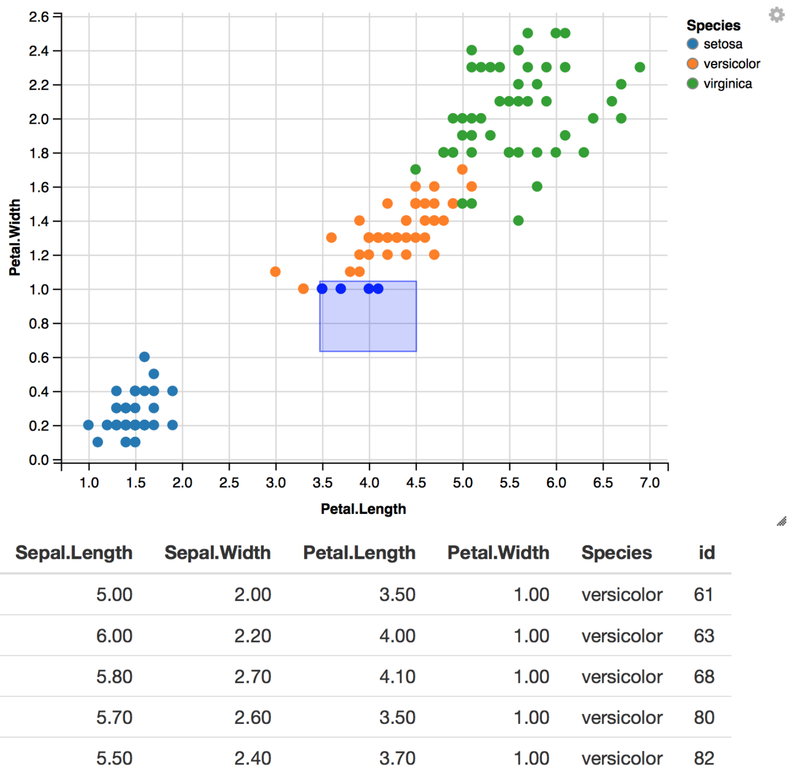
Although shiny is great at facilitating interactions from outside of a plot, it is limited in facilitating interactions within a plot. It does not have all the capabilities that plotly provides. When we combine the two together, more interaction can be achieved with less effort.



curveNumber	pointNumber	x	y
0	132	47	5230
0	224	86	5430
0	390	60	9070
0	410	55	7800
0	878	60	7010
0	897	50	8510
3	298	60	5290
3	1051	60	5150
4	64	50	7710
4	724	37	5200
4	1714	37	5360
4	2776	37	5920

Figure 2.3: a shiny app with a plotly plot with linked brushing

plotly (along with many other R packages that generate htmlwidgets) and ggvis have their own way of incorporating plots into a shiny application. In Figure 2.3, 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()` (Sievert 2017b). 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 **graphics** 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 2.4. ggvis has its own functions (`ggvisOutput()` and `linked_brush()`) that allow for similar interactions to be achieved (Wickham and Chang 2016).



Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	id
5.00	2.00	3.50	1.00	versicolor	61
6.00	2.20	4.00	1.00	versicolor	63
5.80	2.70	4.10	1.00	versicolor	68
5.70	2.60	3.50	1.00	versicolor	80
5.50	2.40	3.70	1.00	versicolor	82

Figure 2.4: an example of linked brushing between ggvis plots

However, we are still left with a general problem of shiny (with the exception of ggvis) recomputing and redrawing a plot or widget every time an input changes. As of writing, work has been developed to prevent plots generated by plotly to only change certain parts of a plot whenever a plot is implemented with shiny. As of writing, this was not possible. However, in a recent version of plotly(4.7.1), Sievert(2017a) has shown that this is possible with a new feature called `plotlyProxy()`, but requires knowledge of the `plotly.js` library and how these proxy objects work.

2.4 animint

animint (Hocking et al. 2017) is an R package designed to allow users to create interactive and animative visuals using **ggplot2**. It uses the concept of direct manipulation defined in Scheiderman (1982). It focuses on adding two main aesthetics to ggplot2 - `clickSelects` to allow the user to click on a selection, and `showSelects` that shows the current selection. The user is able to directly click on the plot, which can be used to link multiple views of data on the same page. It uses D3 to generate the interactive plot on the page, and stores all the data in multiple TSV files that can be viewed locally.

To illustrate, we create a simple example linking between the groups of ethnicity to different views of the nzincome data set.

```r
library(animint)
plot1 <- ggplot(income) + aes(x = sex, clickSelects = sex) + geom_bar()
plot2 <- ggplot(income) + aes(x = weekly_hrs, y = weekly_income, showSelected = sex,
                              color = highest_qualification) + geom_point()

plotAll <- (list(p1 = plot1, p2 = plot2))
structure(plotAll, class = "animint")
```

Figure 2.12: animint example

If we click on any of the bars on the bar plot (left), the scatter plot (right) shows the selected points that correspond that that group.

Hocking's(2017) example of the displaying different views of the World Bank dataset shows how complex these interactive and animative plots can be achieved with less than 100 lines of code. It is simple and straightforward, and is not restricted to linking scatter plots as discussed with crosstalk and plotly. Because the plots are rendered entirely in JavaScript using D3, the plots are relatively more responsive and faster than compared to using a web-server framework like shiny (which has an overhead cost from communicating between a remote server with R and the browser).

The key strength of **animint** is also its weakness as the only type of interactivity that can be achieved is clicking and showing what has been selected (Hocking et al. 2017). Currently, it cannot achieve brushing or zooming. It is only compatible with **ggplot2**. For more developed users of **ggplot2**, not all `geoms` are supported, and may remain static when rendered with **animint**. Furthermore, because everything is computed and rendered beforehand, this means that if a selection requires a recomputation in R before it can be displayed, this is not possible. Hocking (2017) suggests that a solution to this is to use **animint** with shiny, but this means that a new **animint** plot is rendered every time the user interacts with it. The unfortunate situation with creating stand-alone interactive plots this way is that the amount of data that needs to be generated to power the plots increases as we increase the number of subsets. If a data set has multiple subsets that need to be rendered, **animint** will need to make all the different combinations for each subset to link every plot together. The more number of subsets and the larger the dataset, the number of files that need to be generated to drive the interactive or animated plot increases. In this case, using a client-server framework like shiny would be more suitable.

The **animint** package proves itself as a success in implementing a complex system that achieves simple interactive and animative plots that can easily linked and implemented by users using clicks and selection.

2.5 Summary

From assessing all these tools, we can summarise the features and drawbacks for each tool in the table below.

Tool	Type of plot	Compatible with shiny	Types of interactions	Redraws entire plot	Framework type
plotly	plotly(plotly.js), ggplot2	Yes	Clicks, brushing, subsetting, filters	Yes * (unless proxy)	standalone HTML
ggvis	ggvis(Vega)	Yes	off-plot interactions, hovers, brushing, filters	No	client-server
shiny	R plots, anything compatible with it	-	clicks, brushing, filters, subsetting, hovers	Yes	client-server
animint	ggplot2 (D3)	Yes	clicks + selects	No (unless used with shiny)	standalone HTML

Table 1: A summary table of all the tools available and their main capabilities

Note: anything that is compatible with shiny will end up adopting its client-server framework.

Most of these tools can be extended using shiny. However the general problem is that when these systems are implemented with shiny (with the exception of ggvis), every time a user interacts with an input, the plot or corresponding widgets will be recomputed and redrawn. Furthermore, many of these do now allow us to customise our own interactions into the plot. We can use these tools for easily visualise our data with standard interactive plots, but if the user wishes to customise interactivity or extend it further, it presents a dead end or a need for learning its respective API. The other significant factor is that most these tools use a JavaScript library to render their plots. While graphics plots generated in R are supported by shiny and ggplot2 across plotly and **animint**, there is no support for graphics generated with other plotting systems in R. Next, we will look at how we can achieve specific on-plot interactions on static R plots by combining JavaScript with lower levels tools and avoid reproducing entire plots whenever the user interacts with it.

Chapter 3

Interactive R plots using lower level tools

Web interactive graphics can be achieved by R users without the knowledge of HTML, CSS and JavaScript. However, many of these tools use an external JavaScript library to render their plots. This section discusses how we can use two lower level packages, gridSVG(Murrell and Potter 2017) and DOM(Murrell 2016b) to incorporate interactions into R plots and prevent redrawing entire plots. One approach to avoid this is to target parts of the plot that need to be updated. We need a system that renders SVG elements but has a mapping structure that allows elements to be related back to data. In R, we can use the gridSVG package. By combining gridSVG, shiny and JavaScript, we are able to update specific parts of the plot when the user interacts with an input by passing JavaScript messages between R and the browser. Because interactions are achieved by manipulating web content using the DOM, we can alternatively use the DOM package that directly allows us to drive web content from R without the need for writing JavaScript. We will discuss how these different approaches work.

3.1 gridSVG

gridSVG (Murrell and Potter 2017) is an R package that allows for the conversion of grid graphics in R into SVG. This is powerful because it is 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 elements in the data set and SVG elements generated. This is not clear in plotly or ggvis and their JavaScript libraries, which makes it hard to identify or trace data back to the elements on the page. This also explains why it may be difficult to customise interactions on the plot. With gridSVG, we can add JavaScript to grid elements in R using `grid.script()` and `grid.garnish()` (Murrell and Potter 2014).

```r
grid.circle(x = 0.5, y = 0.5, r = 0.25, name = "circle.A",
            gp = gpar(fill = "yellow"))

grid.garnish('circle.A', onmouseover = "allred()",
             onmouseout = "allyellow()", "pointer-events" = "all")

grid.script("allred = function() {
  var circle = document.getElementById('circle.A.1.1');
  circle.setAttribute('fill', 'red');
  }")

grid.script("allyellow = function() {
  var circle = document.getElementById('circle.A.1.1');
  circle.setAttribute('fill', 'yellow');
  }")

grid.export("circle.svg")
```

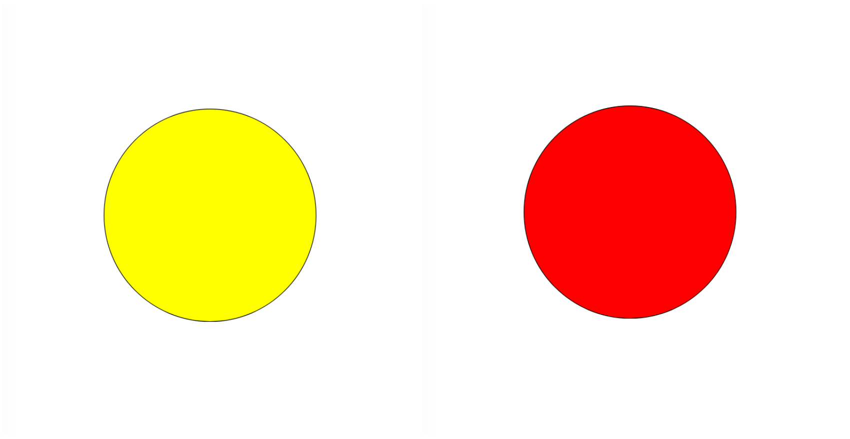


Figure 3.1: An interactive circle made using gridSVG - when the user hovers over the circle, it will turn red (shown on the right)

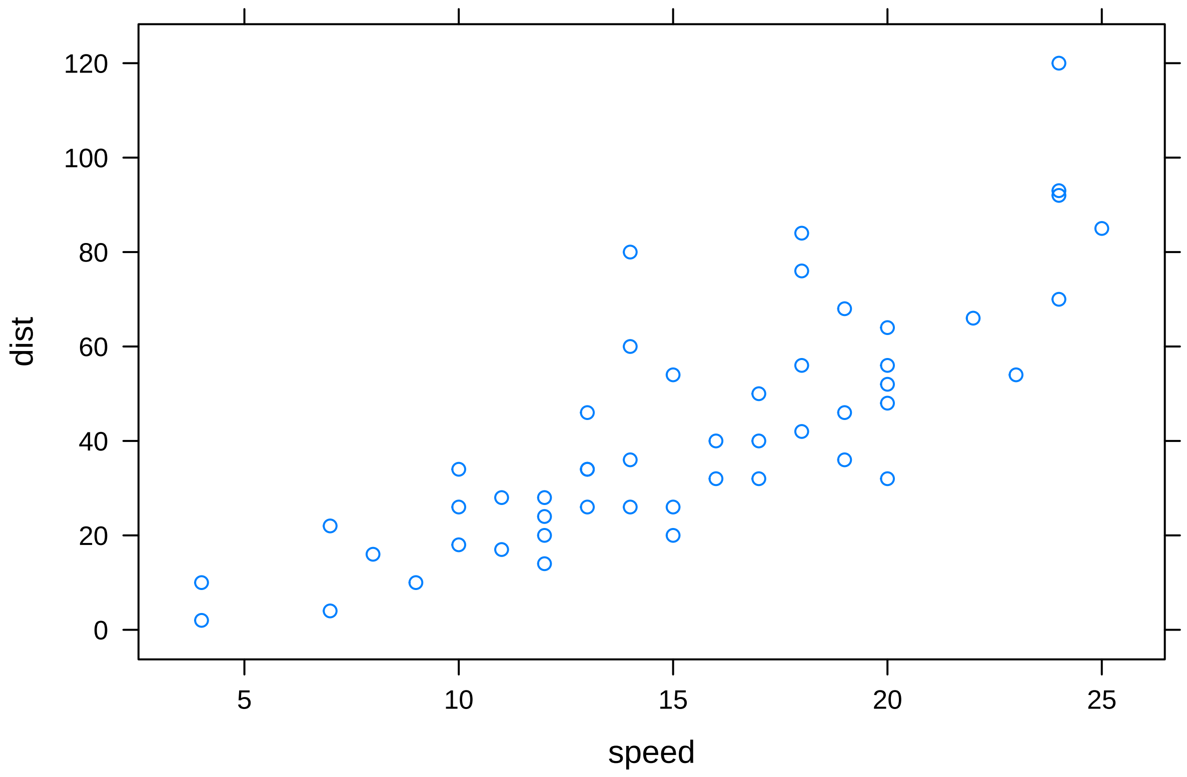
In Figure 3.1, the circle has been drawn in R, named and have interactive elements added before being exported out as an SVG. A simple interaction has been attached to the circle where if the user hovers over the circle, it will turn red.

This shows that there is a relationship between grid objects and SVG objects that are generated. In grid, we have named the circle as `circle.A`. gridSVG maintains this as an grouped SVG element with an id attribute of `circle.A.1`, where inside lies a single SVG circle element called `circle.A.1.1`. In R, we can refer back to these grid objects to attach interactions to their SVG counterparts.

Another important feature gridSVG has is the ability to translate between data and SVG coordinates(Murrell and Potter 2012). Suppose that a plot has been generated. The `exportCoords` argument in `grid.export` is able to generate data that retains the locations of viewports and scales from the original plot in R (Murrell and Potter, 2012). We can use this information to convert data to SVG coordinates and vice versa.

To demonstrate, we have drawn a plot using the `cars` data set and exported its coordinate system with its corresponding SVG. We have separated the svg and the coordinates.

```
xyplot(dist ~ speed, data = cars)
```



```
svgdoc <- grid.export(NULL, exportCoords = "inline")

# separate the svg and coordinates
svg <- svgdoc$svg
coords <- svgdoc$coords
```

To be able to use the coordinate system in R to convert between data and SVG coordinate systems, we need to load it in by calling `gridSVGCoords`.

```
gridSVGCoords(coords)
```

Suppose we have a new point at (4, 5). To be able to convert this in the correct coordinate space, we need to find the correct viewport (identified as `panel`) it lies in. This can then be easily translated into SVG co-ordinates and back using the functions `viewportConvertX` and `viewportConvertY` with `panel`.

```r
# to identify the correct panel:
panel <- "plot_01.toplevel.vp::plot_01.panel.1.1.vp.2"

# if there's a new point we want to find the SVG coordinates of:
(x <- viewportConvertX(panel, 4, "native"))
```

```
## [1] 77.92132
```

```r
(y <- viewportConvertY(panel, 5, "native"))
```

```
## [1] 70.74
```

The native co-ordinates (4, 5) have been translated as (77.92, 70.74) in the SVG co-ordinate system. This can be further added on to the web page without redrawing the rest of the plot as we have the co-ordinates in the SVG space using JavaScript. We can also translated the coordinates back into data to return (4, 5).

```r
# to translate back to data (ie native):
viewportConvertX(panel, x, "svg", "native")
```

```
## [1] 4
```

```r
viewportConvertY(panel, y, "svg", "native")
```

```
## [1] 5
```

The main 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 using base R cannot be directly converted (Murrell and Potter 2014). There is a solution to this using the gridGraphics (Murrell 2015) package that can converts base R graphics into grid graphics (further demonstrated in Section 4.2.1). Another point to note is that the process of converting elements to SVG becomes slow when there are many elements to render although work is under way on this to speed up conversion.

3.1.1 Customising simple plot interactions

A clear limitation that is present in the existing tools discussed previously is letting the user add their own interactions on an existing plot.

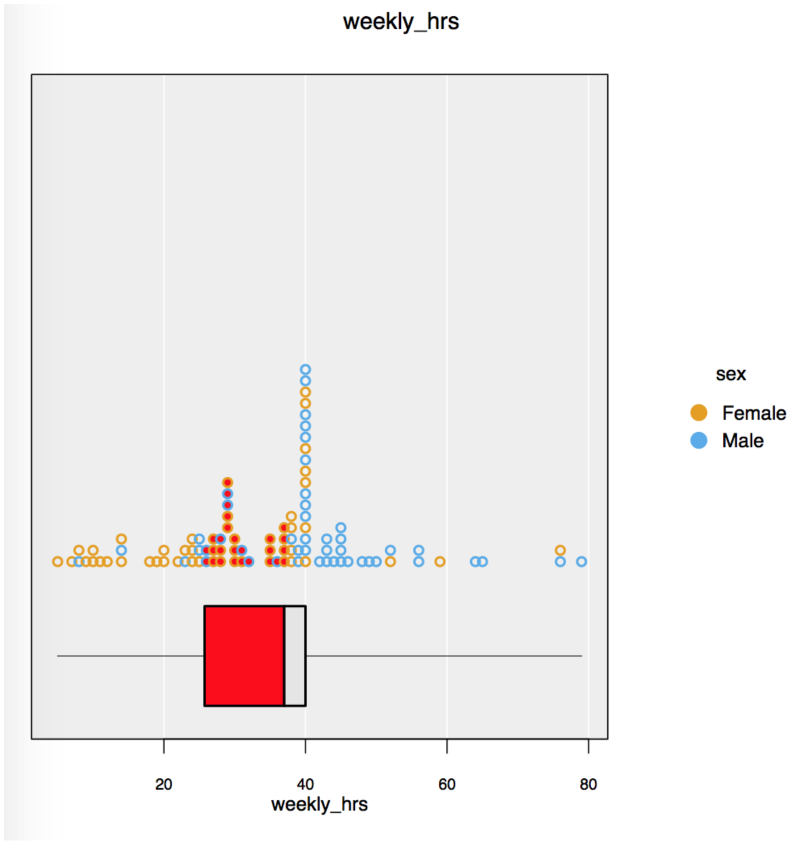


Figure 3.2: An example of a customised box plot interaction on an iNZight plot using gridSVG, JavaScript

One such example is highlighting part of a box plot to show certain values between the median and the lower quartile (Figure 3.2). When the user clicks on this box, it will highlight the points that lie within this range. While this can be achieved with gridSVG and custom JavaScript, it is not as straightforward with plotly or ggvis. Despite plotly and ggvis also rendering graphs in SVG, it is more difficult to identify which elements to target and add interactions to with these systems.

3.1.2 Preventing redraws in shiny using JavaScript messages and gridSVG

As mentioned at the end of Chapter 2, one of the downsides of using shiny along with plotly or other htmlwidgets is its nature to redraw plots every time an input changes. With R plots that are rendered using the `renderPlot` function, redrawing is required because the plot is viewed as a raster image. In other cases, shiny simply re-runs code when a user interacts with an input, which causes the plot to be redrawn. This means that we cannot specifically target elements on the page as the plot is viewed as a single object.

A new approach is to render the plot in SVG and target certain elements that need to be redrawn while using shiny to communicate back to R. If we use SVG, we can separate out which components to target and add interactions without changing the rest of the plot. A complication to this is that we can no longer use the usual shiny input and output functions that link everything on the page. shiny also does not have specific functions to control SVG content. A different way to do this is to pass data between the browser and back to R using JavaScript to change certain elements on the web page. shiny provides a set of functions that allow for messages to be sent through this channel using two JavaScript functions: `shiny.onInputChange()` and `shiny.addCustomMessageHandler()` (Heckmann 2013). To send data from the browser back to R, we use `shiny.onInputChange()`. This allows JavaScript objects to be sent back to the shiny server in a way which can be recognised in R. To send data from R back to the browser, we use `shiny.addCustomMessageHandler()`.

To demonstrate how this is useful in updating certain parts of a plot, we provide an example by altering a smoothing curve using gridSVG and these JavaScript functions. First, we use gridSVG to generate our plot and identify the element corresponding to the trend line. We also need to export the coordinates in order to be able to transform data into the correct SVG coordinates when we update the co-ordinates of trend line.

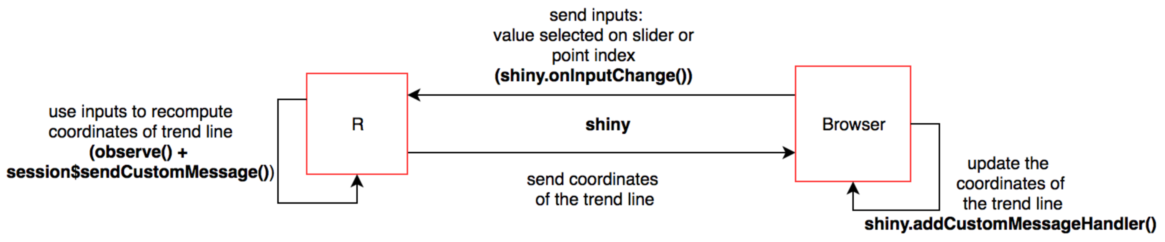


Figure 3.3: Diagram of how things work using shiny's JavaScript functions in Figure 3.4

In Figure 3.3, we can pass the degree of smoothing value from the slider back to R. R then recalculates the x and y co-ordinates of the new smooth. Once these co-ordinates are calculated, they are sent back to the browser using `session$sendCustomMessage`. These coordinates are passed to `shiny.addCustomMessageHandler()` to run a JavaScript function that will update the points of the line. This process is used in Figure 3.4 with a lattice plot of the iris data set.

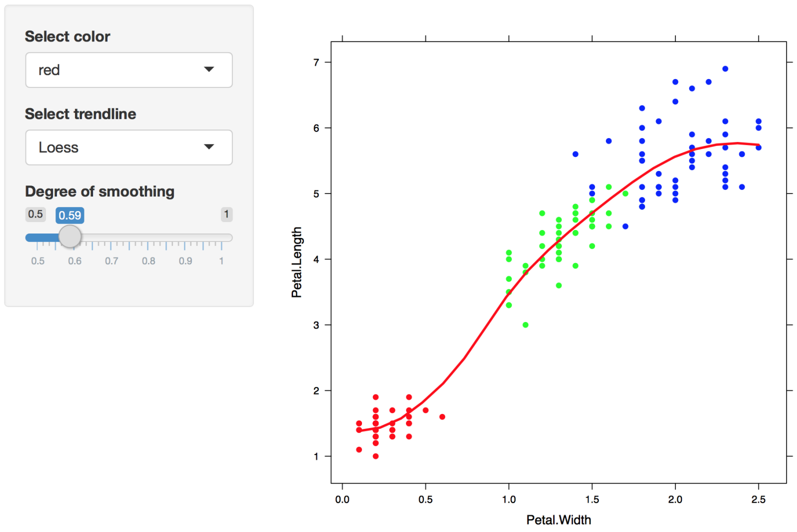


Figure 3.4: A replica of Figure 2.7, but only the trendline changes

This example (Figure 3.4) is extensible as we can render grid graphics (such as lattice) and customise interactions while maintaining a connection between R and the browser using shiny. By doing this rather than redrawing the entire plot, we have only changed the trend line. This method does, however, require the knowledge of JavaScript and the limitations of how much information can be sent through are unknown as it is not commonly used.

To stretch this example further, we added in a feature where the user can highlight over a set of points by dragging the mouse (as seen in Figure 3.5). We return the information about these highlighted points in order to further compute a smoother for just these points. To achieve this in shiny, we have written some JavaScript that returns the indices of these selected points back to R with `shiny.onInputChange()` to compute a suitable smoother which is then displayed.

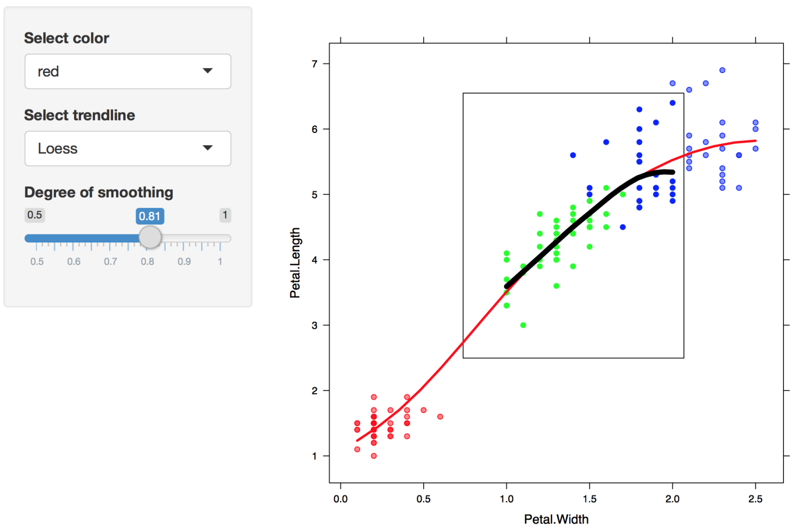

Select color
red
Select trendline
Loess
Degree of smoothing
0.81

Figure 3.5: Select over a set a points to show a smoother

3.2 DOM package

As highlighted in section 1, many interactions driven on the web are done by DOM (Document Object Model) manipulation. In brief, the Document Object Model is a programming interface that allows developers to manipulate content on a web page (W3C 2009). We can use it to navigate and pinpoint specific elements on the page to modify and add interactions to. Generally, this is accessed through by writing JavaScript functions.

Because most interactions are driven by JavaScript and involve modifying content on the page, the DOM package (Murrell 2016b) allows for us to directly do this from R. We can send requests back and forth between R and the browser. This provides a basis for using the web browser as an 'interactive output device' (Murrell 2016a).

Using the DOM package allows us to write certain commands that are analogous to what is written in JavaScript. This removes the burden of traversing between the two programming languages. Rather than writing JavaScript, we can write DOM commands in R that produce similar results. Going back to our circle example in Figure 3.1, we can change the colour of the circle by directly sending this request to the web page.

```r
#changing the circle to red:
circle <- getElementById(page, "circle.A.1.1", response = nodePtr())
setAttribute(page, circle, "fill", "red")
```

In contrast, the JavaScript code for changing this circle from yellow to red:

```
var circle = document.getElementById('circle.A.1.1');
circle.setAttribute(fill, "red");
```

The DOM package is also special in which we are able to do asynchronous programming (`async = TRUE`) (Murrell 2016a). Asynchronous programming is a concept where we are able to start an initial task and run different tasks at the same time. Here, the DOM package is able to run a task from R to the browser but also be able to run commands in R while the call to the browser is still running. This is important as it makes our web applications more efficient than trying to run each command or task one at a time. When we need to call back to R from the browser, these are all asynchronous events that can easily react to user interactions, making it more responsive and creates a smoother experience for the user.

DOM allows R to be called from the browser and for requests from R to be sent to the browser. To demonstrate this, we will replicate the hover effects on the circle as shown in Figure 3.1. Figure 3.6 shows how this can be set up using DOM. We can use `setAttribute` to set the colour of the circle, and use the `RDOM.Rcall` function to send requests from the browser back to R. When the user hovers over the circle, the browser will send a request back to R to run the `turnRed` function, which in turn sends a request back to the browser to change the colour of the circle to red. Once the user hovers out, the browser will send a request back to R to turn it back to yellow. Our result is shown in Figure 3.7.

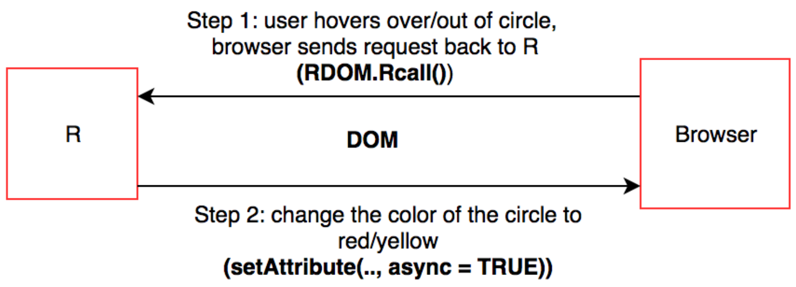

Step 1: user hovers over/out of circle, browser sends request back to R
(RDOM.Rcall())
R
DOM
Browser
Step 2: change the color of the circle to red/yellow
(setAttribute(.., async = TRUE))

Figure 3.6: Simple diagram showing how DOM works with from replicating Figure 3.1

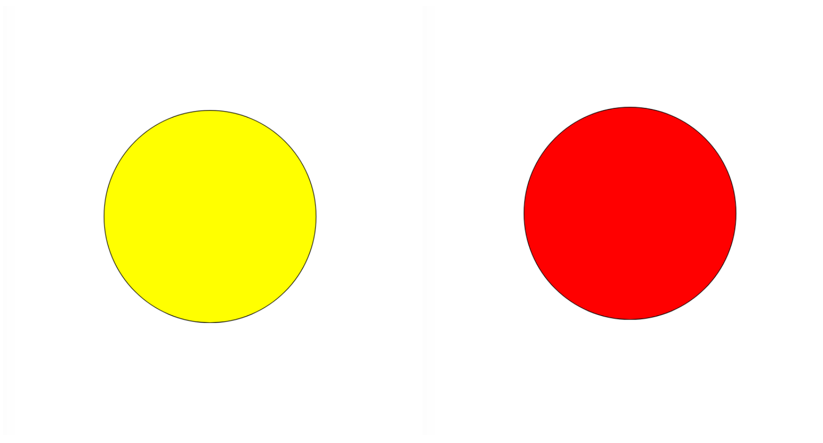


Figure 3.7: DOM example of Figure 3.1 - when hovered, the circle turns red (right)

```r
#draw circle in grid
grid::grid.circle(x = 0.5, y = 0.5, r = 0.25,
                  name = "circle.A", gp = gpar(fill = "yellow"))
#export SVG
svg <- gridSVG::grid.export(NULL)$svg
dev.off()

#set up new page and add circle:
library(DOM)
page <- htmlPage()
appendChild(page,
            child = svgNode(XML::saveXML(svg)),
            ns = TRUE,
            response = svgNode())

circle <- getElementById(page, "circle.A.1.1", response = nodePtr())
# hover effects:
turnRed <- function(ptr) {
  setAttribute(page,
              circle,
              "fill",
              "red",
              async = TRUE)
}

turnYellow <- function(ptr) {
  setAttribute(page,
              circle,
              "fill",
              "yellow",
              async = TRUE)
}

setAttribute(page,
             circle,
             "onmouseover",
             "RDOM.Rcall('turnRed', this, ['ptr'], null)")

setAttribute(page,
             circle,
             "onmouseout",
             "RDOM.Rcall('turnYellow', this, ['ptr'], null)")
```

The example (Figure 3.7) takes approximately 40 lines of code for a hover effect. It is much more 'lower level' and requires the user to know how the Document Object Model and main web technologies work together. A different approach would be to write some JavaScript and send it across to the browser from R. Since we are only just changing the colour of the circle, it is more efficient to write JavaScript and send it to the browser instead rather than telling the browser to call back to R. It is better to call back to R when a computation is necessary (such as recomputing a trend line's co-ordinates in the Section 3.2.1).

3.2.1 Comparing DOM to shiny

DOM is similar to shiny as it establishes a connection between R and the browser. To compare, we have replicated Figure 3.4 using DOM.

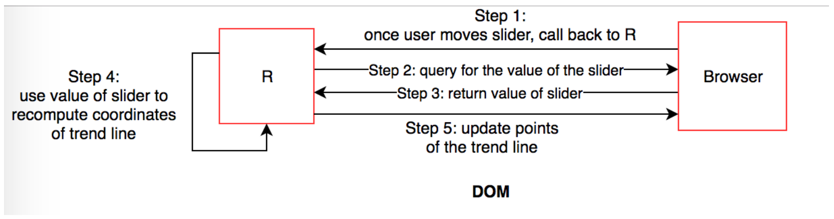

Step 1:
once user moves slider, call back to R
Step 4:
use value of slider to recompute coordinates of trend line
R
Step 2: query for the value of the slider
Step 3: return value of slider
Browser
Step 5: update points of the trend line
DOM

Figure 3.8: Steps on how a trend line can be altered using the DOM package

The process of creating this example is similar to what was done with shiny. However, it is more difficult to set up as it requires the user to manually link all the components on the page. First, we draw the plot and save it as an SVG in memory. Next, we can add the SVG plot and a slider to the page. We identify which element corresponds to the trend line, and define what happens when the slider moves or when text is clicked. This requires an additional query to the browser to return the value of the slider before it can be returned back to R, as shown in Figure 3.8. These are co-ordinated using asynchronous callbacks, where once a response is returned, we can schedule another task behind it. These can be viewed as a series of steps that are linked together. Once the value of the slider is returned, we can use it to recalculate the coordinates of the trend line before updating it on the page. Our final result is put together in Figure 3.9.

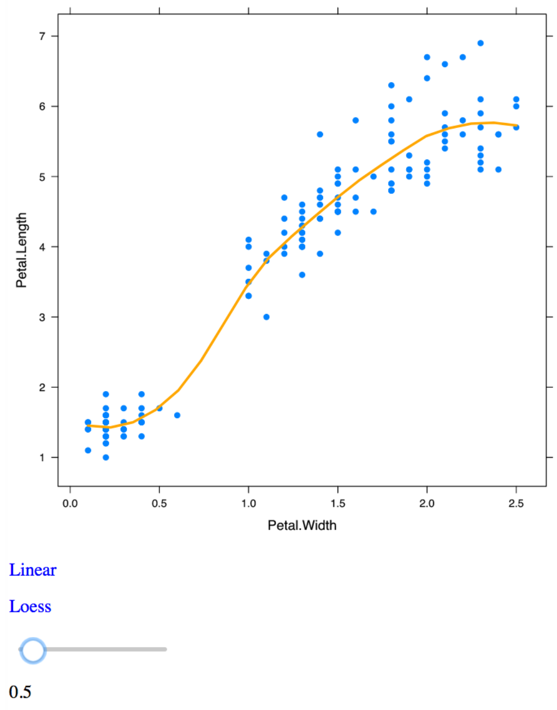


Figure 3.9: DOM example of Fig 3.4 for changing a trend line using a slider

DOM allows for more flexibility as we have control over the entire page. From a developer's perspective, we can continue to modify elements on the page. Users have access to R while the the connection to the web 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. Furthermore, a shiny application can only do one task at a time, and cannot run tasks asynchronously. However, this may be resolved by the promises package (Cheng 2017b) in the near future, which allows for asynchronous programming within R and thus, more responsive shiny applications (Cheng 2017a). A caveat of using the DOM package is that requires a lot more code to link everything together. In shiny, these links between inputs and outputs are much easier to co-ordinate.

Internally, there are many limitations with this package. As this package is still developmental, only part of the DOM (Document Object Model) API has been expanded, and the connection between R and the browser requires extra attention (Murrell 2016a). In some cases, it is still not possible to achieve certain interactions without JavaScript, such as capturing where the mouse's position is on screen. Murrell (2016a) states that it can only be run locally and is currently aimed at a single user rather than multiple users.

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.

There is a clear trade off 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 web interactive plots are easier to use but are complex to modify and extend further. In the next section, we discuss how we can simplify the implementation of certain interactions on plots originally rendered in R and build a solution using these tools.

Chapter 4

Designing a more flexible way of producing simple interactions

By using gridSVG, DOM and JavaScript, we can customise interactions onto plots. However, these are too specific and assume a lot of knowledge from the user. We need a way to provide interactions that can be easily customised and defined by the user with a much less steeper learning curve. This section discusses a potential solution using grid, gridSVG and DOM to drive web interactive graphics.

4.1 The main idea using grid, gridSVG and DOM

In each of the previous examples, they is 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 to that element, and how to define what happens when an interaction occurs. This idea can be broken down into 5 simple steps:

- Draw the plot or elements in R
- Identify elements to interact with
- Determine what kind of interaction is to be achieved
- Attach and link interactions and events to targeted elements
- Send interaction instructions and plot to the browser

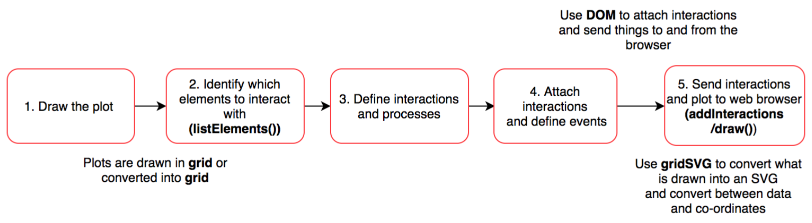

Use **DOM** to attach interactions and send things to and from the browser
1. Draw the plot
2. Identify which elements to interact with **(listElements())**
3. Define interactions and processes
4. Attach interactions and define events
5. Send interactions and plot to web browser **(addInteractions /draw())**
Plots are drawn in **grid** or converted into **grid**
Use **gridSVG** to convert what is drawn into an SVG and convert between data and co-ordinates

Figure 4.1: grid, gridSVG and DOM in the process

The process above (Figure 4.1) can be implemented using grid, gridSVG and DOM. We can use the relationship between grid and gridSVG elements explained in Section 3.1 to allow users to define and identify which elements to target. We can use DOM to attach interactions to certain elements and send these across to a web page. The reason for using DOM rather than shiny is that there are more complexities that work under the shiny framework including reactive programming, which is particularly difficult to grasp in detail. Because it is so low level, we can use DOM to create different types of interactions, but it is unreasonable for the majority of users as it requires some understanding of web technologies and takes too much effort. As seen in Figure 3.7, it takes roughly 40 lines of code to achieve a simple hover effect (with or without writing JavaScript), and about 200 lines of code to link a slider to a smoother (Figure 3.9).

We need a system that is more convenient for users, not too strenuous to code up, does not require too many pre-requisites, but is flexible enough to achieve different interactions. We have created the interactr package that attempts to prototype this idea. It acts as a convenience wrapper for defining interactions with DOM, gridSVG and grid. It aims to allow users to define their own interactions to plots in R without the need for a full understanding of the web technologies involved.

We have recreated some examples using interactr to demonstrate this idea. Many of the examples discussed below use functions that are found in this package.

4.2 Examples

4.2.1 Linking box plots

The goal for 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.

We note that everything that will be done is coded in R so it requires no knowledge JavaScript or other web technologies from the user.

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

```r
library(interactr)
library(lattice)
bw <- bwplot(iris$Sepal.Length, main = "Sepal length", xlab = "Sepal length")
```

Here, we have stored the box plot into a variable called `bw`. To attach interactions, we need to identify what elements have been drawn. We can do that by listing the elements.

```r
listElements(bw)
```

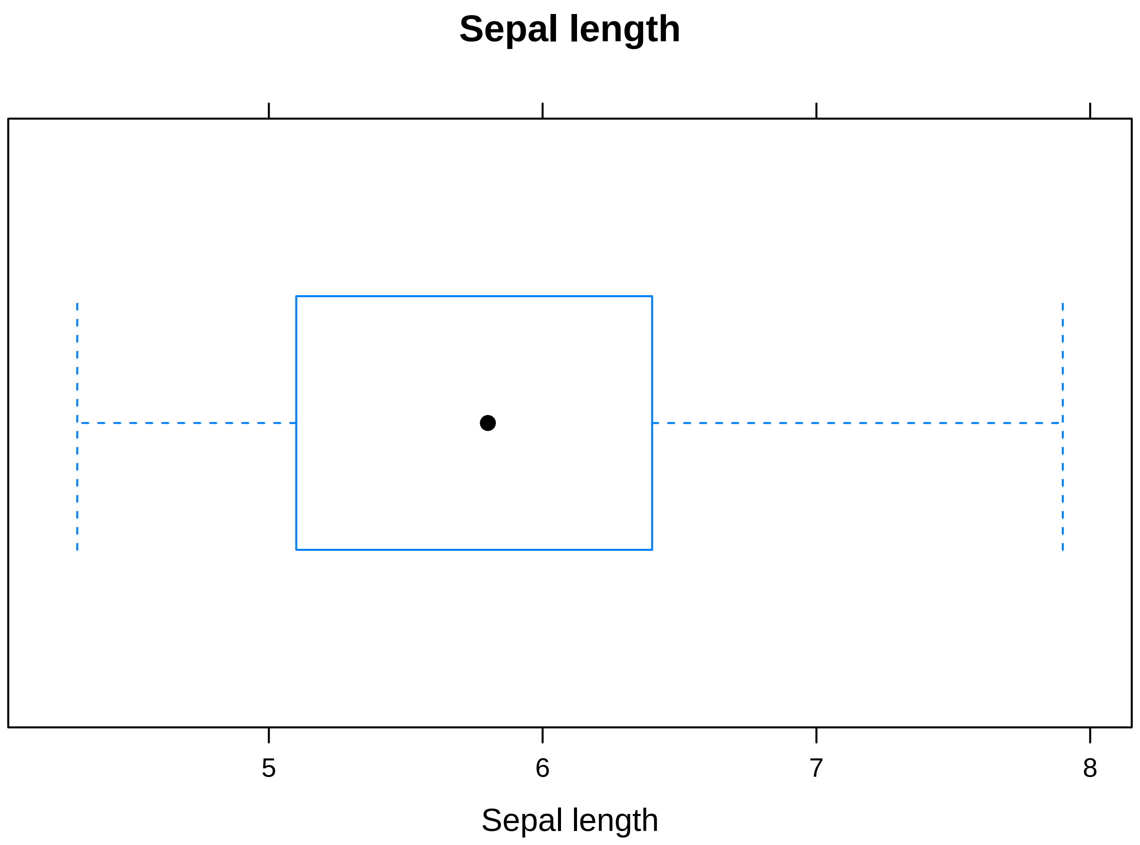


Figure 4.2: Producing a box plot in R

```
## plot_01.background
## plot_01.main
## plot_01.xlab
## plot_01.ticks.top.panel.1.1
## plot_01.ticklabels.left.panel.1.1
## plot_01.ticks.bottom.panel.1.1
## plot_01.ticklabels.bottom.panel.1.1
## plot_01.bwplot.box.polygon.panel.1.1
## plot_01.bwplot.whisker.segments.panel.1.1
## plot_01.bwplot.cap.segments.panel.1.1
## plot_01.bwplot.dot.points.panel.1.1
## plot_01.border.panel.1.1
```

```r
box <- "plot_01.bwplot.box.polygon.panel.1.1"
```

This will print the plot (Figure 4.2) and return a list of all the elements that make up the box plot in R. The user can identify which element to target to attach interactions. This is one of the disadvantages (further discussed in Section 5.2) of this process - the user must deduce which element to target through the names listed. In some cases, this is straightforward like in the example above, we suggest that the 'box' should refer to the box plot. We have identified the box that marks between the lower quartile and upper quartile.

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.

```r
interactions <- list(hover = styleHover(attrs = list(fill = "red",
                                                     fill.opacity = "1")))
```

Note that the interaction has only been defined but not linked to the targeted element (which is the box) yet.

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

This line of code (the `draw` function) both links the interaction we defined before to the box element and sends the plot across to a new web page. We see that when the user hovers over the box, the box turns red as seen in Figure 4.3.

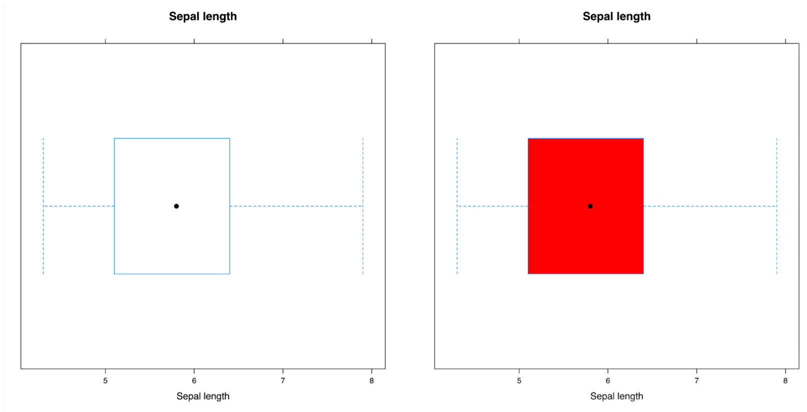


Figure 4.3: Box plot with hover interaction

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 using this package which is further discussed in Section 5.2 when plots are separately drawn to the graphics device each time.

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

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

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

We see that the box plot and the scatter plot we drew in R are now on the same web page (Figure 4.4).

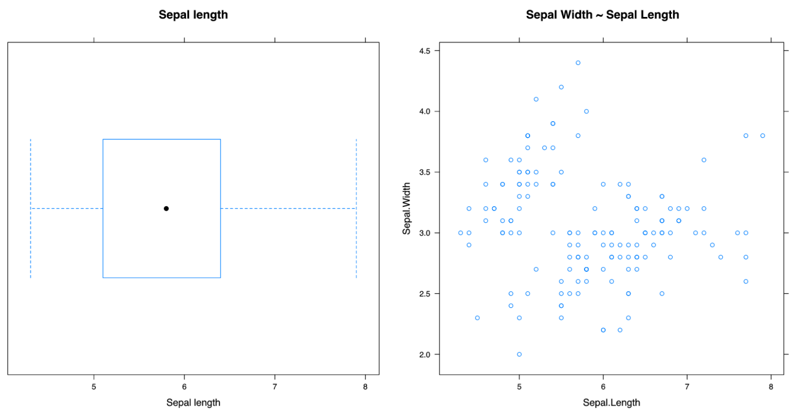


Figure 4.4: Boxplot and scatter plot on the same web page

To highlight the points in the scatterplot that lie in the range of the box, the user can define the function as follows. We will determine the indices 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 indices of selected points
  index <- which(min(range) <= iris$Sepal.Length
                 & iris$Sepal.Length <= max(range))
  # set identified points 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 now need to define the event that will invoke this interaction before we can send it to the browser.

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

We have inserted the function name to run when a ‘click’ is performed (line 1). Next in the second line of code, we have appended this 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. This is shown in Figure 4.5.

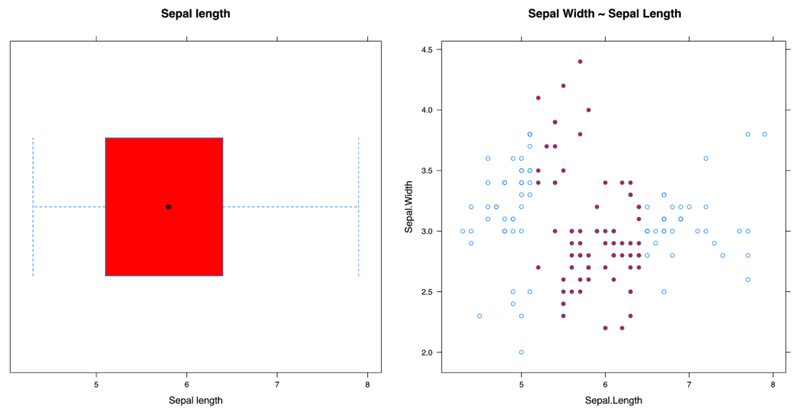


Figure 4.5: Click on box plot to light up points on scatter plot

This example can be further extended by linking the box plot to both a scatter plot and density plot. Here, we have taken the first 500 observations from a survey conducted with school children in 2009 (CensusAtSchool 2009) 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.

First, we draw a box plot of boys heights and attach a hover effect to the box, similar to what was done previously. The range of the box is identified for further use.

```
bw <- bwplot(boys$height, main = "Boxplot of boys' heights",
             xlab = "Boys' heights (cm)")
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 scatter plot between boys' heights and armspan 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)
```

The next line of code adds the density plot of girls heights. Note that no interactions have been defined yet.

```
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)
```

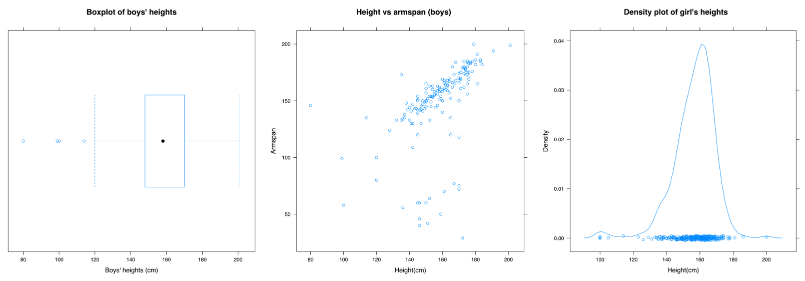


Figure 4.6: All three plots on the same web page

Figure 4.6 shows all three plots on the same web page.

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"))
```

This polygon will remain 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.

Next, 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 points for drawing the region to be highlighted
  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 on scatter plot, remove missing values
  index <- which(min(range) <= boys$height
              & boys$height <= max(range)
              & !is.na(boys$armspan))

  # set points that will highlight according to index
  setPoints(points,
            type = "index",
            value = index,
            attrs = list(fill = "red",
                         fill.opacity = "0.5",
                         class = "selected"))

}
```

Finally, we 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 as seen in Figure 4.7.

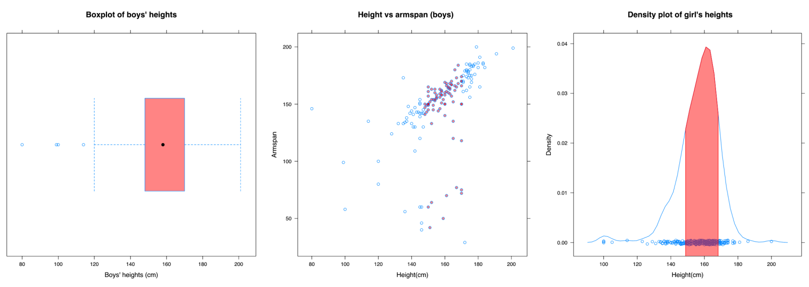


Figure 4.7: A single click on the box plot links the density and scatterplot together

4.2.2 Changing the degree of smoothing of a trend line

Another example that can be done with interactr is driving an interaction using a slider. The slider controls the smoothing of the trend curve. We do not want to redraw the scatter plot when the smoothing settings change.

Here, it becomes more complex as it requires information to be sent and queried back and forth between R and the browser.

Once again, we begin by drawing a plot.

```
iris.plot <- xyplot(Petal.Length ~ Petal.Width,
                    data = iris,
                    pch = 19,
                    type = c("p", "smooth"),
                    col.line = "orange", lwd = 3)
#list elements and print plot
listElements(iris.plot)
#send plot to browser
```

```r
draw(iris.plot, new.page = TRUE)
```

Next, we add a slider to the page. This has not been linked up to any elements yet (Figure 4.8).

```r
#add slider to page:
addSlider("slider", min = 0.5, max = 1, step = 0.05)
```

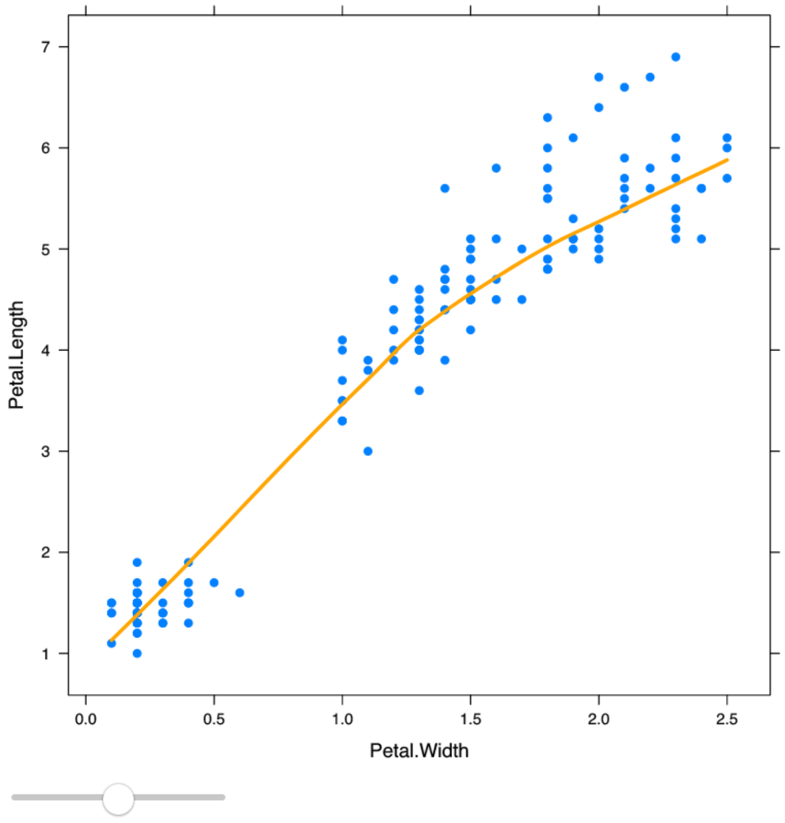


Figure 4.8: Plot with slider

The user can write a function with the argument `value` to define what happens when the slider moves. This passes the value of the slider from the web page back to R. Here, we want to use the value to control the span of the trend line. To translate the new x and y values of the points that define the drawn trend line, we need to convert them into SVG co-ordinates (as mentioned before in Chapter 3.1) before updating these points.

```r
controlTrendline <- function(value) {
  showValue(value) # to show value of the slider
  value <- as.numeric(value)
  #user defines what to do next (here, recalculates x and y)
  x <- seq(min(iris$Petal.Width), max(iris$Petal.Width), length = 20)
  lo <- loess(Petal.Length~Petal.Width, data = iris, span = value)
  y <- predict(lo, x)

  #convert coordinates and set points
  panel <- findPanel('plot_01.xyplot.points.panel.1.1')
  pt <- convertXY(x, y, panel)
  setPoints("plot_01.loess.lines.panel.1.1", type = "coords", value = pt)
}
```

Once this is done, we need to pass this function to retrieve the value of the slider as it moves. To do this, have a special function called `sliderCallback`. This redefines and creates the entire function that is now called `sliderValue`.

```r
# pass defined function through sliderCallback to pass slider value correctly
sliderValue <- sliderCallback(controlTrendline)
```

Finally, we can link the `sliderValue` function back to the slider such that when the slider moves, the trend line will be updated based upon the value of the slider as seen in Figure 4.9.

```r
int <- list(oninput = "sliderValue")
addInteractions("slider", int)
```

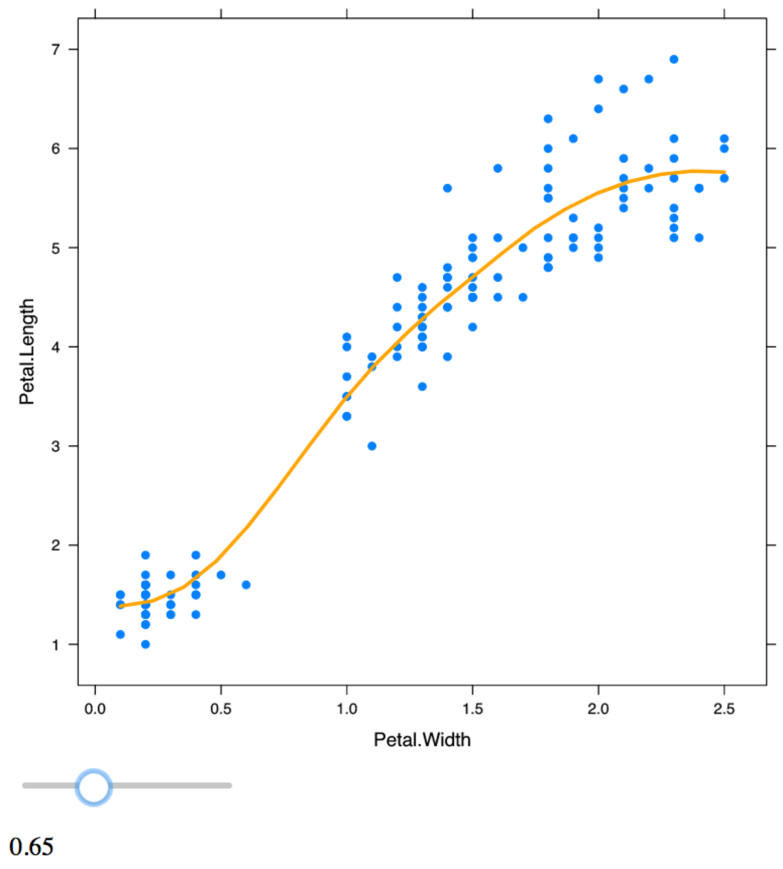


Figure 4.9: Plot with slider that controls the smoothness of the trend line

Another feature that the user may want to achieve is to be able to select a set of points and compute a trend line using those specific points. To be able to do this, we need to add a new element to the page to represent this special trend line. This can be done using the `addLine` function. Here, we have added it to the same group where these points are.

```
pointsPanel <- findPanel("plot_01.xyplot.points.panel.1.1")
addLine("newSmooth", pointsPanel, class = "hello", list(stroke = "red",
                                                        stroke.width = "1",
                                                        fill = "none"))
```

Note that this appears to be hidden on the page, as the points of this line have not been defined yet. Next, a new function needs to be defined to be able to compute this new smoother.

```
#create new smoother:
createSmooth  = function(index) {
  #this returns the indices of the points selected
  index <- as.numeric(unlist(strsplit(index, ",")))
  #filter selected points:
  if (length(index) > 20) {
    selected <- iris[index, ]
    x <- seq(min(selected$Petal.Width), max(selected$Petal.Width), length = 20)
    lo <<- loess(Petal.Length ~Petal.Width, data = selected, span = 1)
    y <- predict(lo, x)
    #convert co-ordinates:
    pt <- convertXY(x, y, pointsPanel)
  } else {
    pt <- ""
  }
  setPoints("newSmooth", type = "coords", value = pt)
}
```

Because the index of the points need to be returned from the browser back to R, we use `boxCallback` to help us link these functions together. As linking a selection box is a special type of interaction, we can pass our defined function through to the `addSelectionBox` function which adds on the selection box and links it together to compute the new smoother.

```
#link callback functions together to pass index values to function
boxIndex = boxCallback(createSmooth)
addSelectionBox(plotNum = 1, el = "plot_01.xyplot.points.panel.1.1", f = "boxIndex")
```

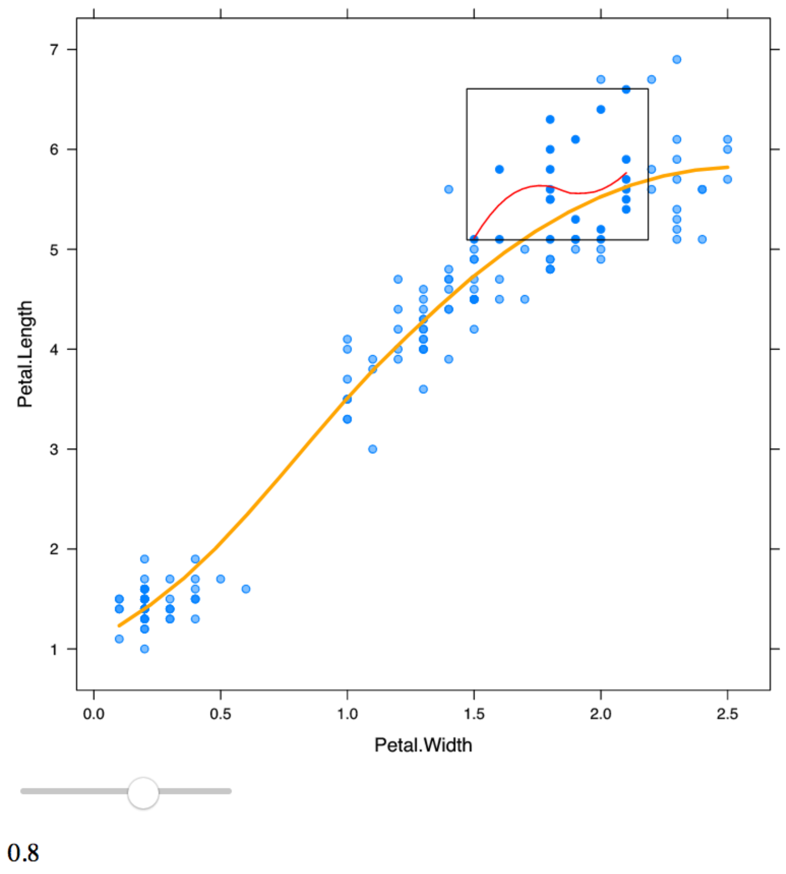


Figure 4.10: Plot that has a selection box feature that draws a separate smoother

The user now can draw a selection box over a set of points, and a new smoother should render on the page based upon these points (shown in Figure 4.10).

The notion of having special functions is required when there is a need for querying the browser for more information (such as the value of the slider, or the points selected on a page). In the box plot example in Section 4.2.1, the information was stored back in R which did not require a special callback function. These types of interactions are more complex to handle.

4.3 Compatibility with other graphics systems

A useful feature of interactr is its compatibility with other R graphing systems including lattice, graphics (also known as base R plots), and ggplot2.

Before we begin, we will briefly mention the two different graphics systems in R. One is known as the graphics, the other known as grid . A major difference between the two systems is that grid is not made to draw complete plots by a single function (Murrell 2011). Rather, it is seen as a lower level graphics tool that has been used to build successful higher level plotting packages, including lattice and ggplot2. Packages that are built on top of the grid system can be accessible to the other grid tools available, including gridSVG and grImport (Murrell 2011). Likewise, there are other tools that are only compatible with the graphics system. There are many other packages that are built on top of these two major systems that make up the graphics that we can produce in R.

The examples discussed so far with interactr have been done with lattice plots. However, it is possible to achieve the same with other plotting systems. To demonstrate, we have taken the box plot example in Figure 4.5 and replicated it using graphics and ggplot2.

4.3.1 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 (Murrell 2015), which acts as a translator by converting graphics plots into grid plots with a consistent naming scheme. The `grid.echo()` function achieves this.

```r
library(grid)
plot(1:10, 1:10)
grid.ls()
gridGraphics::grid.echo()
grid.ls()
```

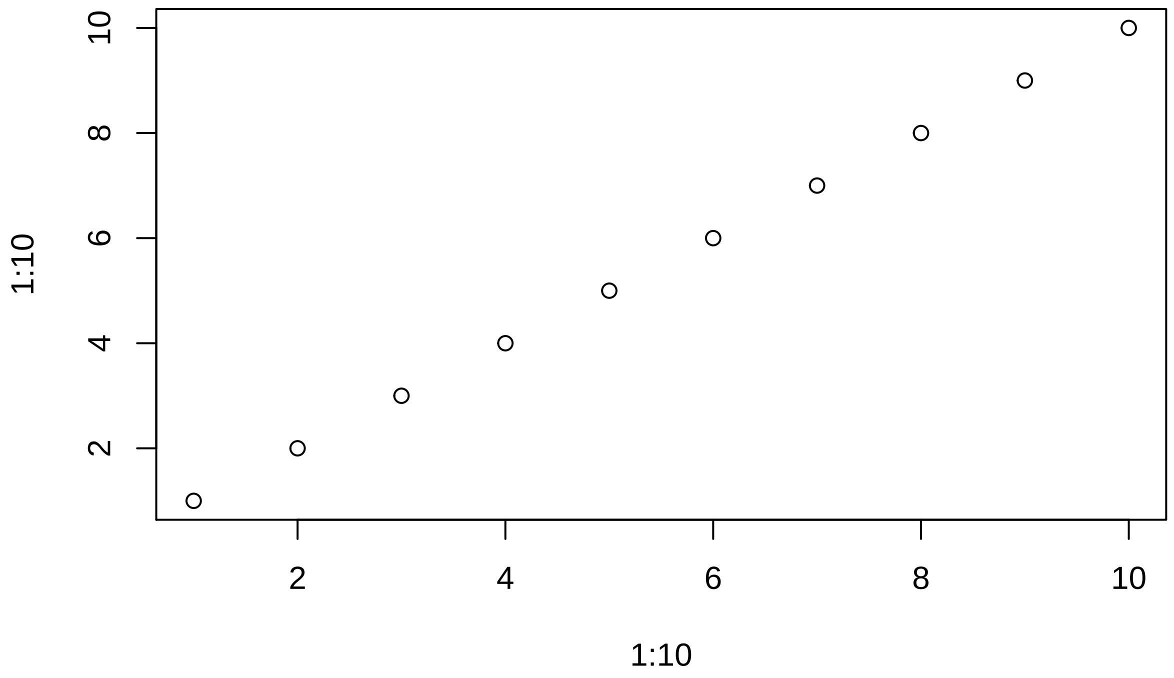


Figure 4.11: An example of a graphics plot that is converted into a grid plot using gridGraphics

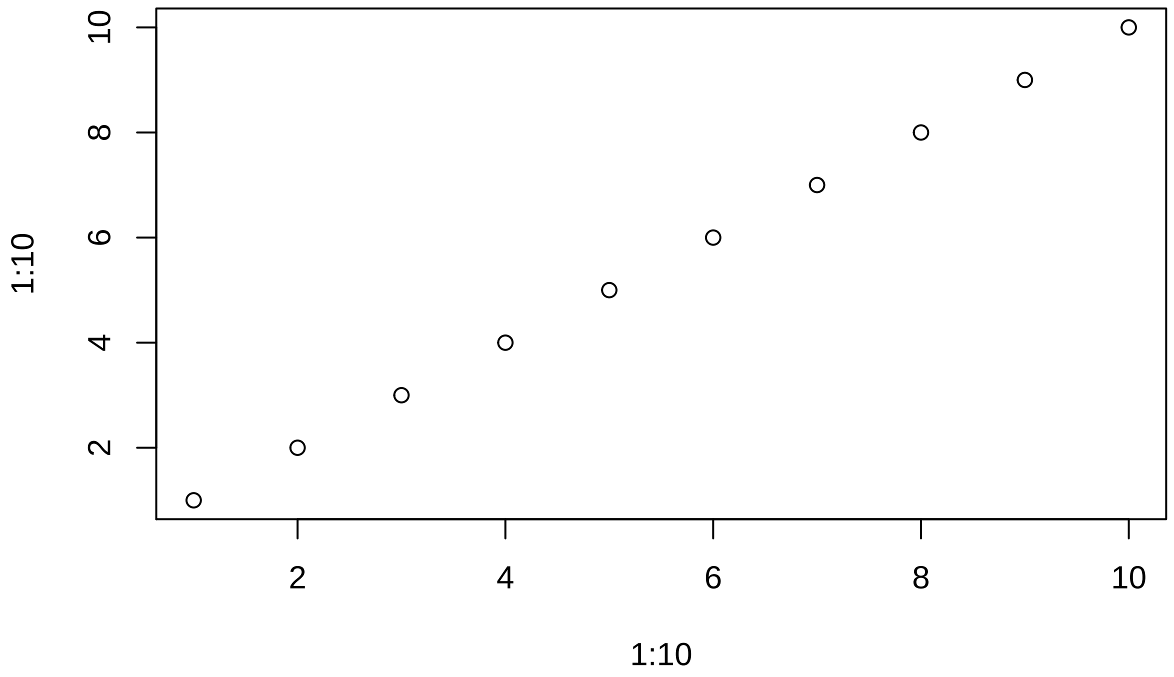


Figure 4.12: An example of a graphics plot that is converted into a grid plot using gridGraphics

```
## graphics-plot-1-points-1
## graphics-plot-1-bottom-axis-line-1
## graphics-plot-1-bottom-axis-ticks-1
## graphics-plot-1-bottom-axis-labels-1
## graphics-plot-1-left-axis-line-1
## graphics-plot-1-left-axis-ticks-1
## graphics-plot-1-left-axis-labels-1
## graphics-plot-1-box-1
## graphics-plot-1-xlab-1
## graphics-plot-1-ylab-1
```

The code above produces a graphics plot that has been converted to a grid plot (as seen in Figure 4.12). To check this, we have called `grid.ls()` to check whether is a grid object. In the first call, it returns nothing because it is not a grid object. Once we call `grid.echo()`, `grid.ls()` returns a list of elements that make up the plot.

Another problem 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 (Paul Murrell and Allaire 2015).

The only change that we need to do is run an extra `recordPlot` command before we call `listElements`.

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

```r
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`. We can use the same `highlightPoints` function defined back in the `lattice` example.

```r
# identify box in box plot and send the plot to the browser
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
```

```r
points <- 'graphics-plot-1-points-1'
boxClick <- list(onclick = "highlightPoints")
addInteractions(box, boxClick)
```

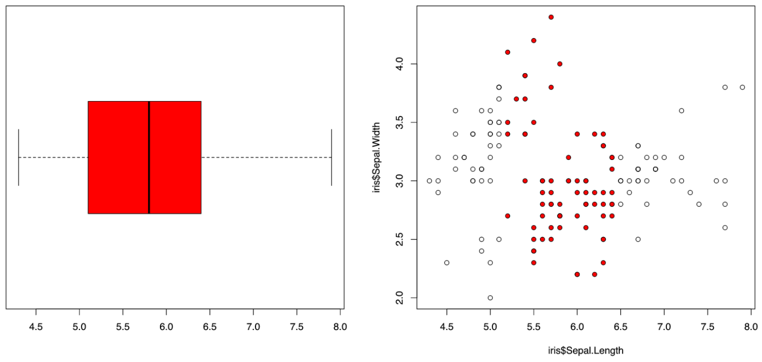


Figure 4.13: Box plot example replicated using a graphics plot

The same interaction has been achieved in **??**. This shows that there is potential for customising interactions onto graphics plots. The process is the same, except for an additional step to convert a graphics plot into a grid type plot.

4.3.2 ggplot2

ggplot2 (Wickham 2009) 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.

```r
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 manually identified the range of the box.

```
# 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.

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

The difference is the naming of these grid elements do not have a clear structure in ggplot2. To locate the points on the plot, we can use `findElement` to return the element corresponding to these points.

```
points <- findElement("geom_point.point")
```

Next, we can use the same function `highlightPoints` defined before sending these interactions to the browser (Figure 4.14).

```
#using highlightPoints defined previously in 3.1
boxClick <- list(onclick = "highlightPoints")
addInteractions(box, boxClick)
```

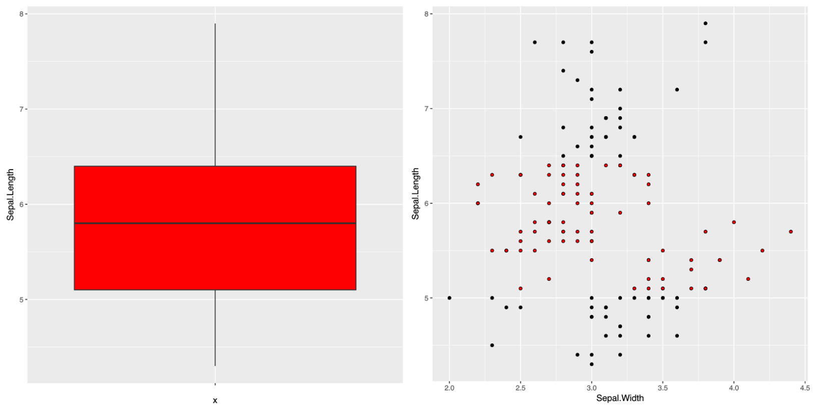


Figure 4.14: Box plot example replicated using a plot rendered with ggplot2

This demonstrates that there is a possible way of achieving interactions with ggplot2 and that there is potential for interactr to plug into different R plotting systems. Consequently, 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 and a starting point for what 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. But the most commonly used systems are covered by our examples. The process is based upon 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. Next, we discuss the limitations and future directions of using this process as a way of creating web interactive graphics.

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 define and customise interactions with flexibility. It can also achieve unidirectional linking between plots. However, there are many limitations present with its current implementation and is not yet recommended for general use. But it does have potential to be further developed in the future. In this section we will discuss the advantages (Section 5.1) and limitations of this method (Section 5.2), further compare it to existing tools (Section 5.3), and briefly comment on future directions for developing interactr.

5.1 Advantages

The idea of interactr is inspired from finding a more easier way to customise certain interactions onto plots drawn in R without the need to learn JavaScript. It takes advantage of R's flexible graphics system and caters for both graphics and grid plots.

Using the DOM package allows us to use R to recompute and do calculations that originally cannot be done in JavaScript, such as recomputing densities on a plot based upon a selection. Below is an example of changing density plots based upon what the user has selected (Figure 5.1).

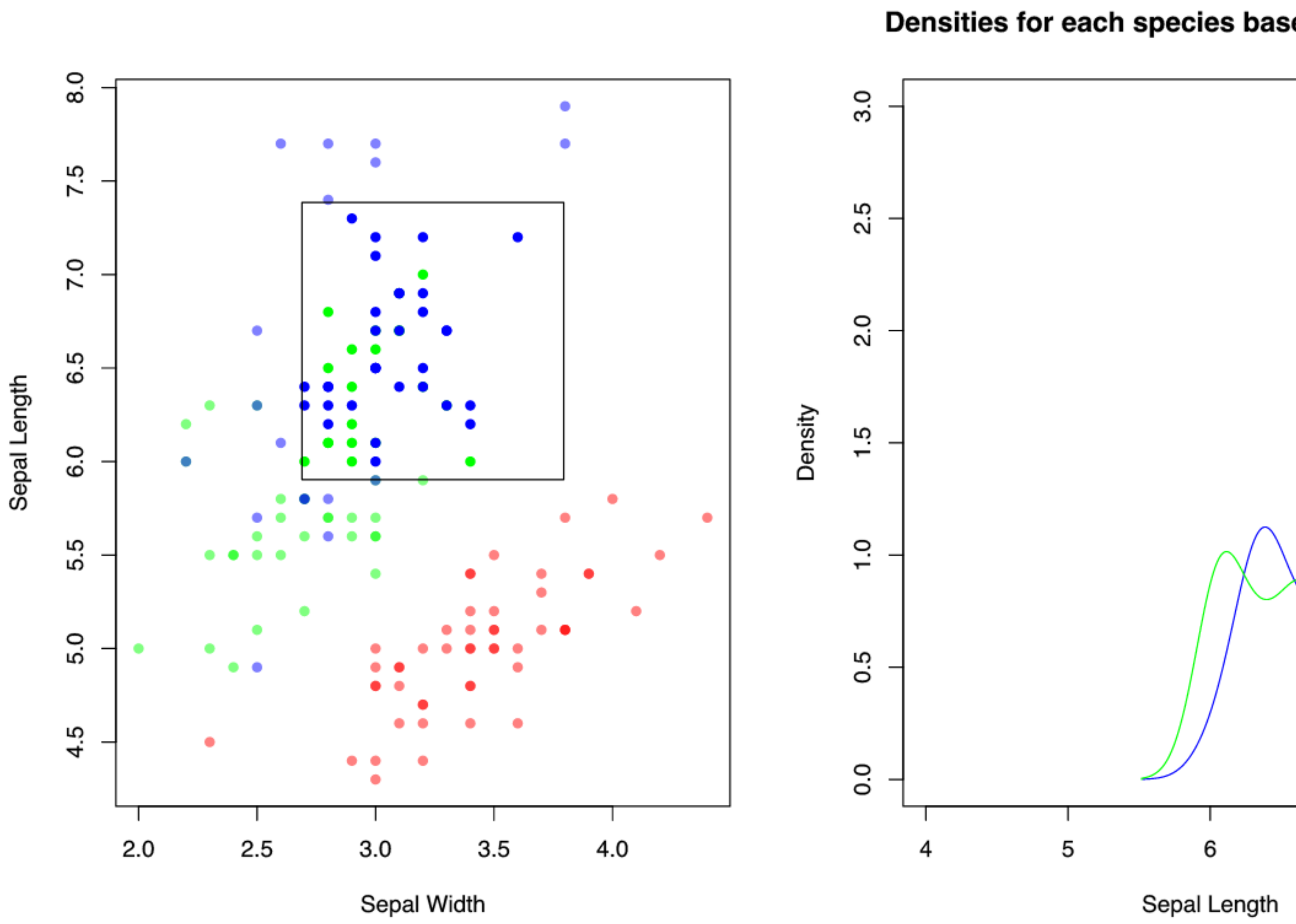


Figure 5.1: Different selections (left) recompute different densities (right) with interactr

This cannot be done with any of the existing tools unless paired with shiny. However, shiny simply reproduces the plot each time. Using an asynchronous system like DOM also creates a more responsive application.

The main advantage for identifying graphical elements to data is that we have more flexibility in targeting sub components of a plot. A prime example of an interaction that is difficult to implement across other tools is linking a part of the box plot to other plots (seen in Section 4.2.1).

Another advantage is that we are able to add layers on top of plots and draw shapes that can help provide more information. This cannot be easily done by the existing tools as it requires information on how these plots are layered and rendered. With gridSVG underneath to provide a clear mapping structure of these layers, we can add on elements to existing plots on the web page to show more about a user's interaction such as a selection or a click. They can also be used to highlight regions and draw new elements. The is exemplified in the trend line example in Section 4.2.2, where an additional smoother can be added to the plot.

interactr can be used to create links between plots via clicks and selection boxes, where a single plot can control the rest of the plots. It may be possible to create multi-directional links, but becomes more complex to co-ordinate. It is a success in its own for providing basic interactions to any type of plot drawn in R. However, there are still many limitations that are present with its current implementation.

5.2 Limitations

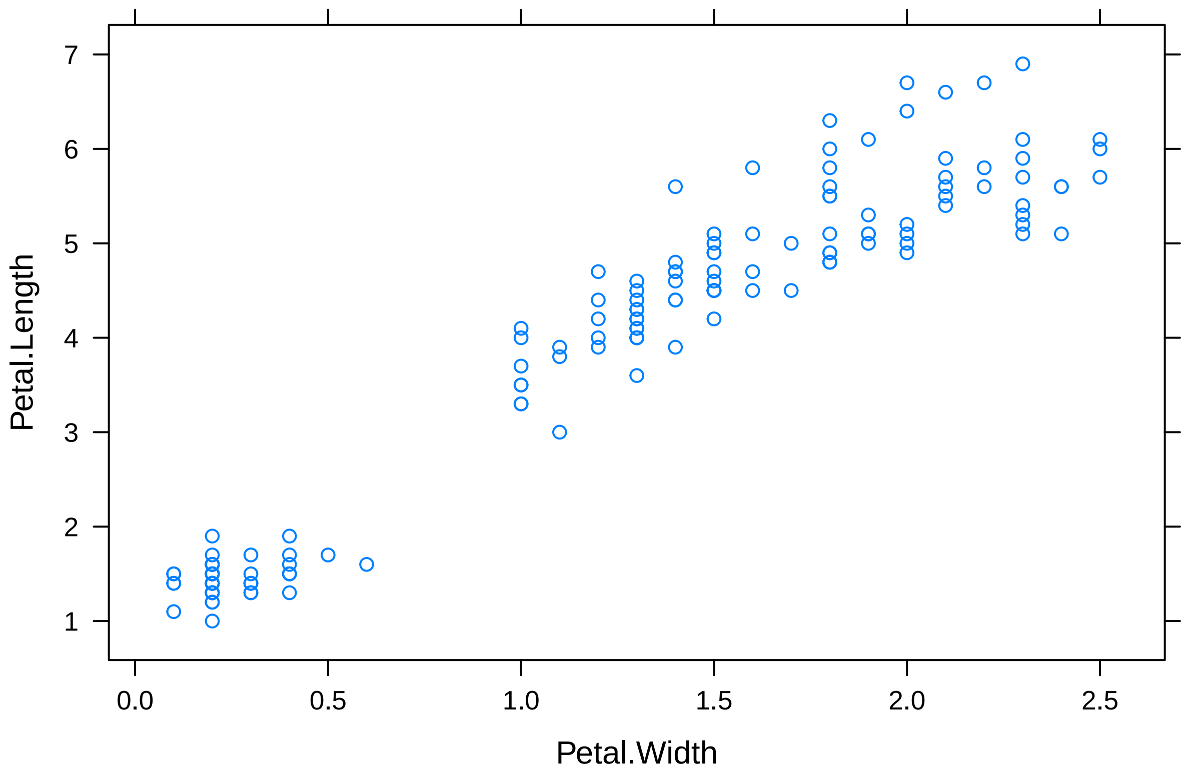
With gridSVG, one major limitation is that only grid objects can be converted into SVG. This limits us to plots that must be drawn in R to a graphics device before it can be sent to the browser. A further limitation is that gridSVG is relatively slow when we try to convert a plot made up of many elements. Currently, work is proceeding to make gridSVG faster.

Because interactr is mainly built upon the DOM package, many of the limitations of DOM highlighted in Section 3.2 are carried over. Applications made with DOM 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 shiny applications require a shiny server for them to be hosted on the web, DOM would require something similar to allow for applications to be shared and accessed. A few trials using a shiny server have been successful, but these only act as a provisional solution. Furthermore, because the underlying system involving requests being sent between R and the web browser, this can be slower than plots that are driven fully by JavaScript.

There are further limitations with its current implementation. The approach is based upon the graphical elements produced. This requires the user to be able to link the particular elements of the data which is more tedious than the existing tools discussed in Section 2 that link data to graphical elements on the page. The user must call `listElements` before sending the plot to the browser as it prints the plot to a current device and returns a 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. This occurs with ggplot2, where if we re-plot with the exact same command, the names of these elements change every time. Another problem with using `listElements` is that the user will need to deduce which element corresponds to what is seen on the plot as the naming for these objects in by these plotting systems 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. The code below shows the difference between the naming scheme in lattice and ggplot2.

Listing the elements from a plot produced with lattice:

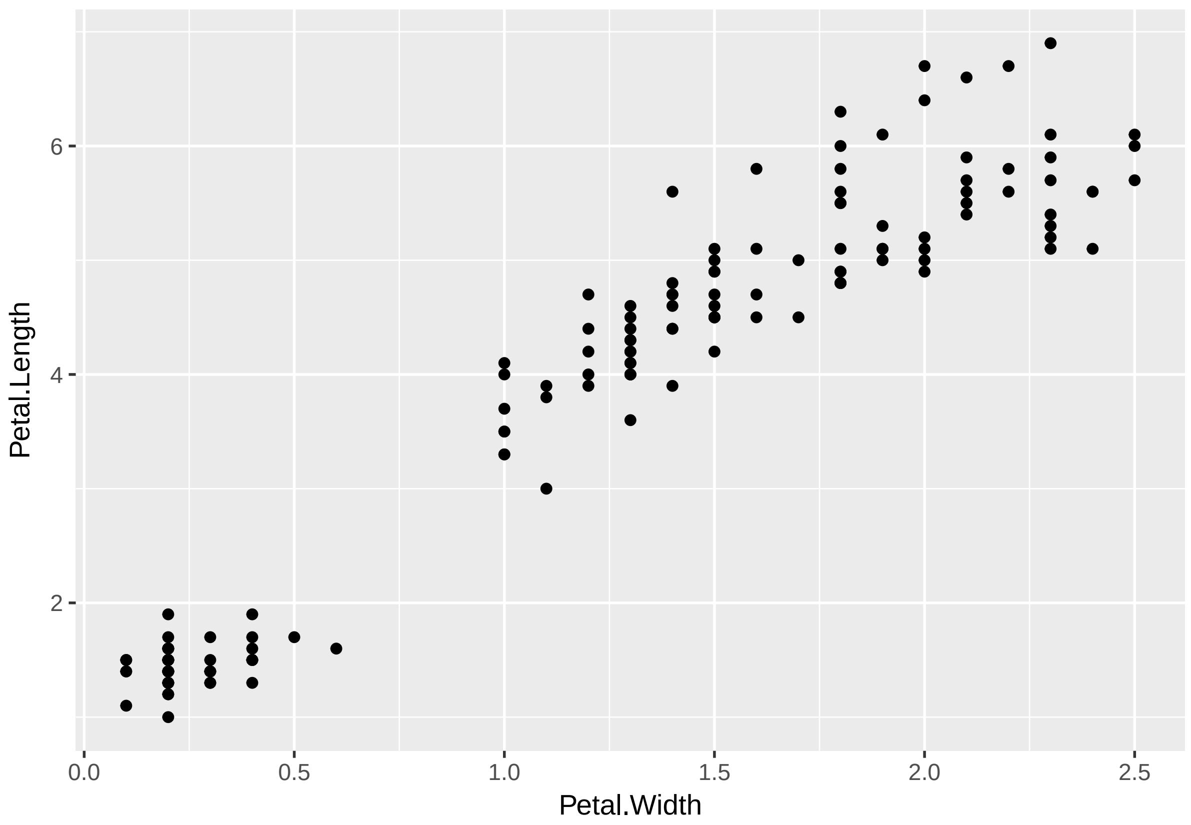
```r
sp <- xyplot(Petal.Length ~ Petal.Width, data = iris)
listElements(sp)
```



```
## plot_01.background
## plot_01.xlab
## plot_01.ylab
## plot_01.ticks.top.panel.1.1
## plot_01.ticks.left.panel.1.1
## plot_01.ticklabels.left.panel.1.1
## plot_01.ticks.bottom.panel.1.1
## plot_01.ticklabels.bottom.panel.1.1
## plot_01.ticks.right.panel.1.1
## plot_01.xyplot.points.panel.1.1
## plot_01.border.panel.1.1
```

Listing the elements from a plot produced with ggplot2:

```r
p <- ggplot(iris) + aes(x = Petal.Width, y = Petal.Length) + geom_point()
listElements(p)
```



```
## layout
##   background.1-7-10-1
##   panel.6-4-6-4
##     grill.gTree.829
##       panel.background..rect.820
##       panel.grid.minor.y..polyline.822
##       panel.grid.minor.x..polyline.824
##       panel.grid.major.y..polyline.826
##       panel.grid.major.x..polyline.828
##     NULL
##     geom_point.points.816
##     NULL
##     panel.border..zeroGrob.817
##   spacer.7-5-7-5
##   spacer.7-3-7-3
##   spacer.5-5-5-5
##   spacer.5-3-5-3
##   axis-t.5-4-5-4
##   axis-l.6-3-6-3
##     axis.line.y..zeroGrob.848
##     axis
##        axis.1-1-1-1
##          GRID.text.845
##        axis.1-2-1-2
##    axis-r.6-5-6-5
##    axis-b.7-4-7-4
##      axis.line.x..zeroGrob.841
##      axis
##        axis.1-1-1-1
##        axis.2-1-2-1
##          GRID.text.838
##    xlab-t.4-4-4-4
##    xlab-b.8-4-8-4
##      GRID.text.832
##    ylab-l.6-2-6-2
##      GRID.text.835
##    ylab-r.6-6-6-6
##    subtitle.3-4-3-4
##    title.2-4-2-4
##    caption.9-4-9-4
```

Another limitation is the number of interactions that can be attached. So far, the examples expressed in Section 4.3 require a single element to be controlled and assumes that the each grid object listed corresponds to a single SVG element. 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. These devices must still be open in order to communicate and retrieve existing information about the plot. 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 in section 4.2.1 of linking a 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 to this is to store the information about each plot that is sent to the web browser so that it can be retrieved by the user if needed in R. Another approach would be to plot all necessary plots in a single window which would eliminate the need for this.

A further assumption that the interactr package currently has is that the `native` units in textsf{grid} represent the data values that are plotted. As discussed in Section 4.3.2, ggplot2 uses a different co-ordinate system and this assumption does not hold. Instead, we need to take a detour and get the data values that are stored in `ggplot_build()`.

5.3 Comparison to existing tools

interactr's main point of difference is the ability to replicate plots or objects drawn in R (in both graphics systems) and to achieve on-plot and off-plot interactivity. shiny can do this but you cannot easily attach specific interactions as the whole plot is rendered as a single raster image file (such as a png). Furthermore, many of these existing tools rely on the shiny framework. As highlighted in Section 2.3, one of the major disadvantages that shiny possesses is a tendency to recompute and redraw entire plots whenever an input changes. In interactr, only the part of the plot that the user specifically targets is modified and customised interactions can be achieved. It provides a possible way of linking different types of plots together, whereas existing tools, specifically crosstalk, have focused on linked brushing between 'row-observation' data. To put this in perspective, the simple example of linking box plots to other types of plots in Figure 4.2 is an interaction that is difficult to achieve without expert knowledge of their respective APIs.

In comparison to the existing tools discussed in Section 2, interactr has a more complex API for users. The main reason for this is to increase flexibility across creating different types of interactions. But this is entirely developmental. In comparison to using DOM and gridSVG directly, it provides convenience for certain processes, such as drawing an SVG plot and adding elements to the web page and styling hovers. Currently, only certain interactions highlighted in Section 4.2 can be achieved.

There is potential for designing a more simpler and structured API that is more intuitive for developers and users. An example to highlight this is changing the bandwidth of a density plot (shown below with interactr in Figure 5.2). This example has been replicated with ggvis(Figure 5.3), shiny(Figure 5.4), and with plotly+shiny (Figure 5.5). The animint package will not be able to do this because it is restricted to clicks and selection.

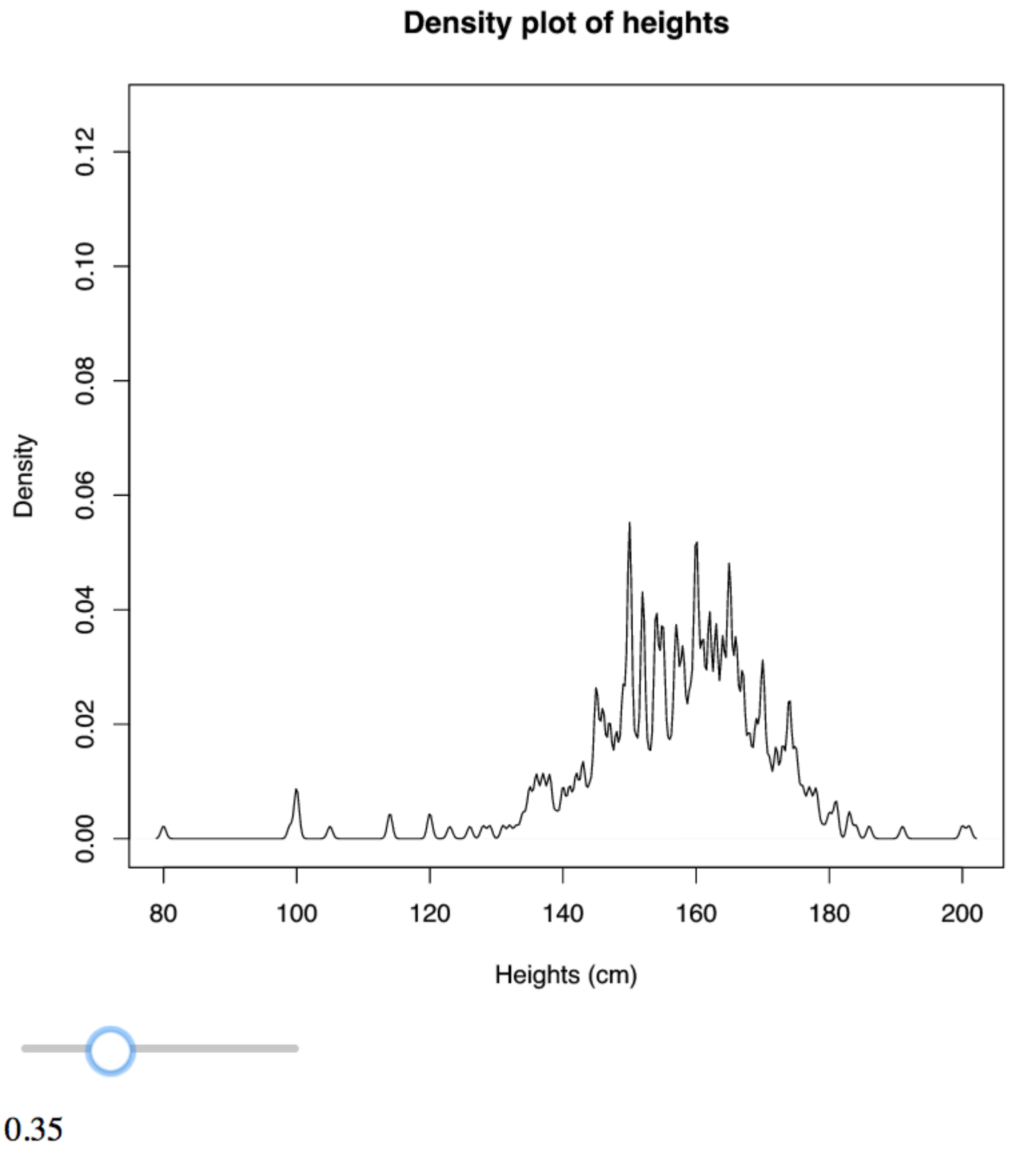


Figure 5.2: control density bandwidth with interactr

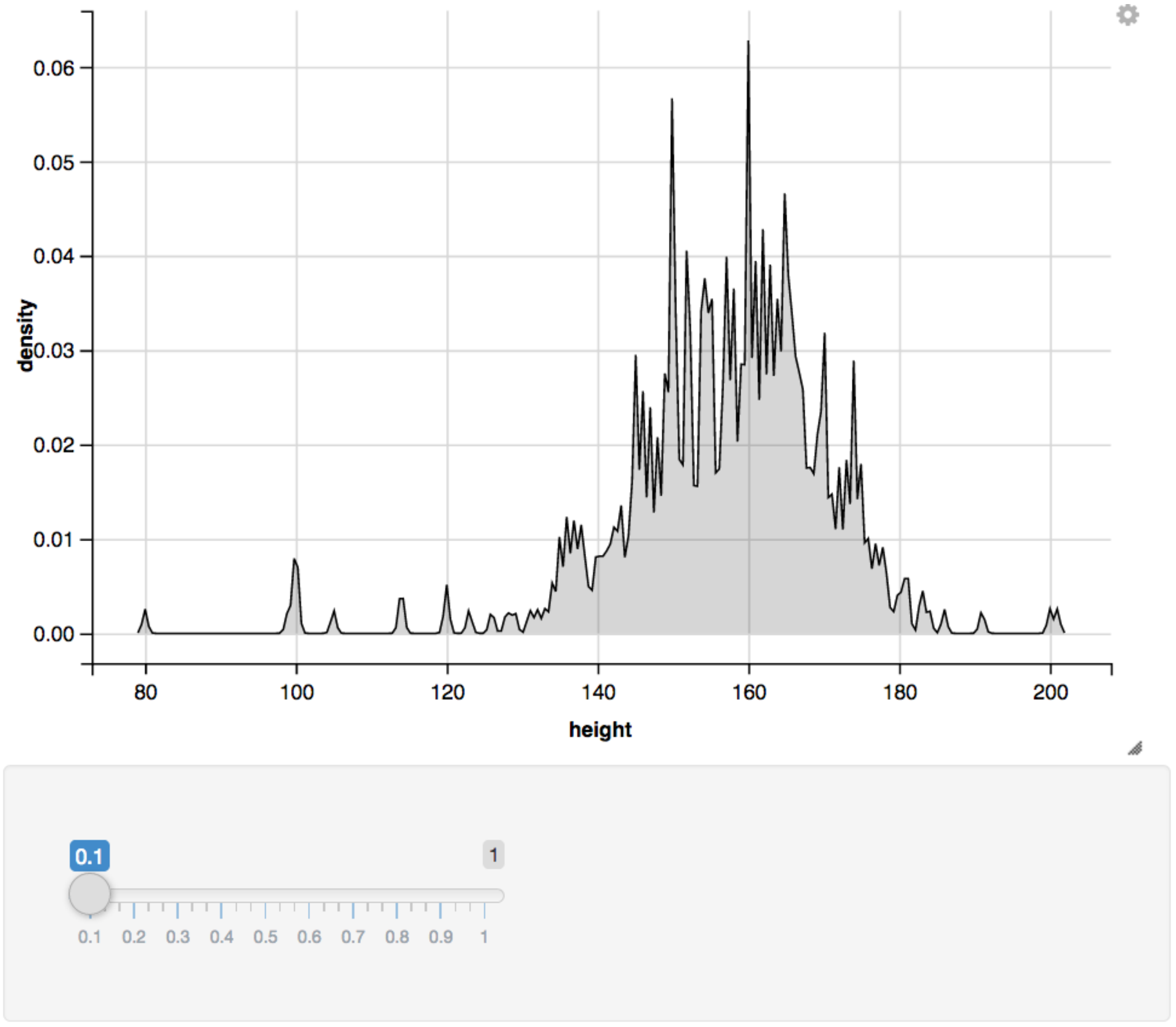


Figure 5.3: control density bandwidth with ggvis

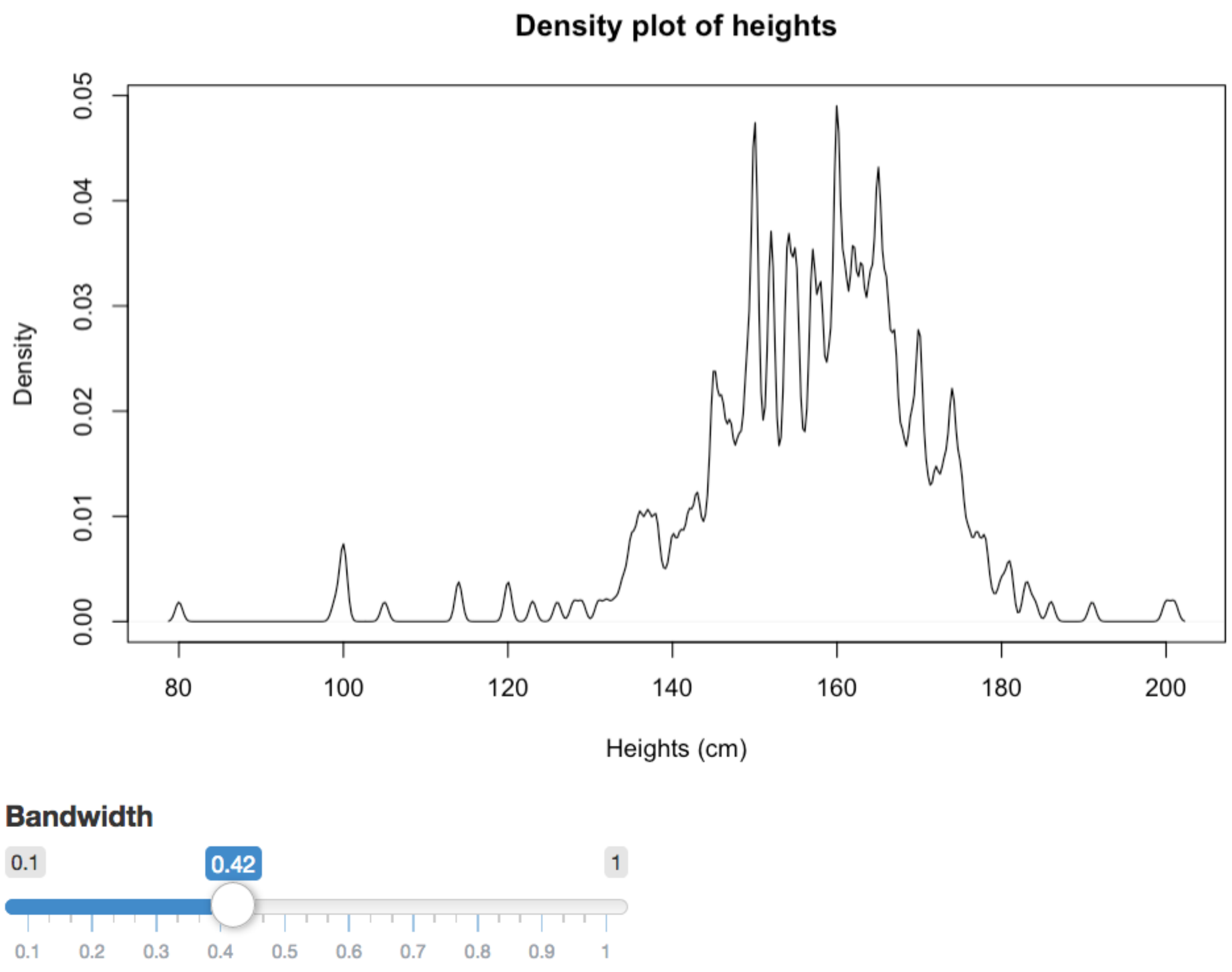


Figure 5.4: control density bandwidth with shiny

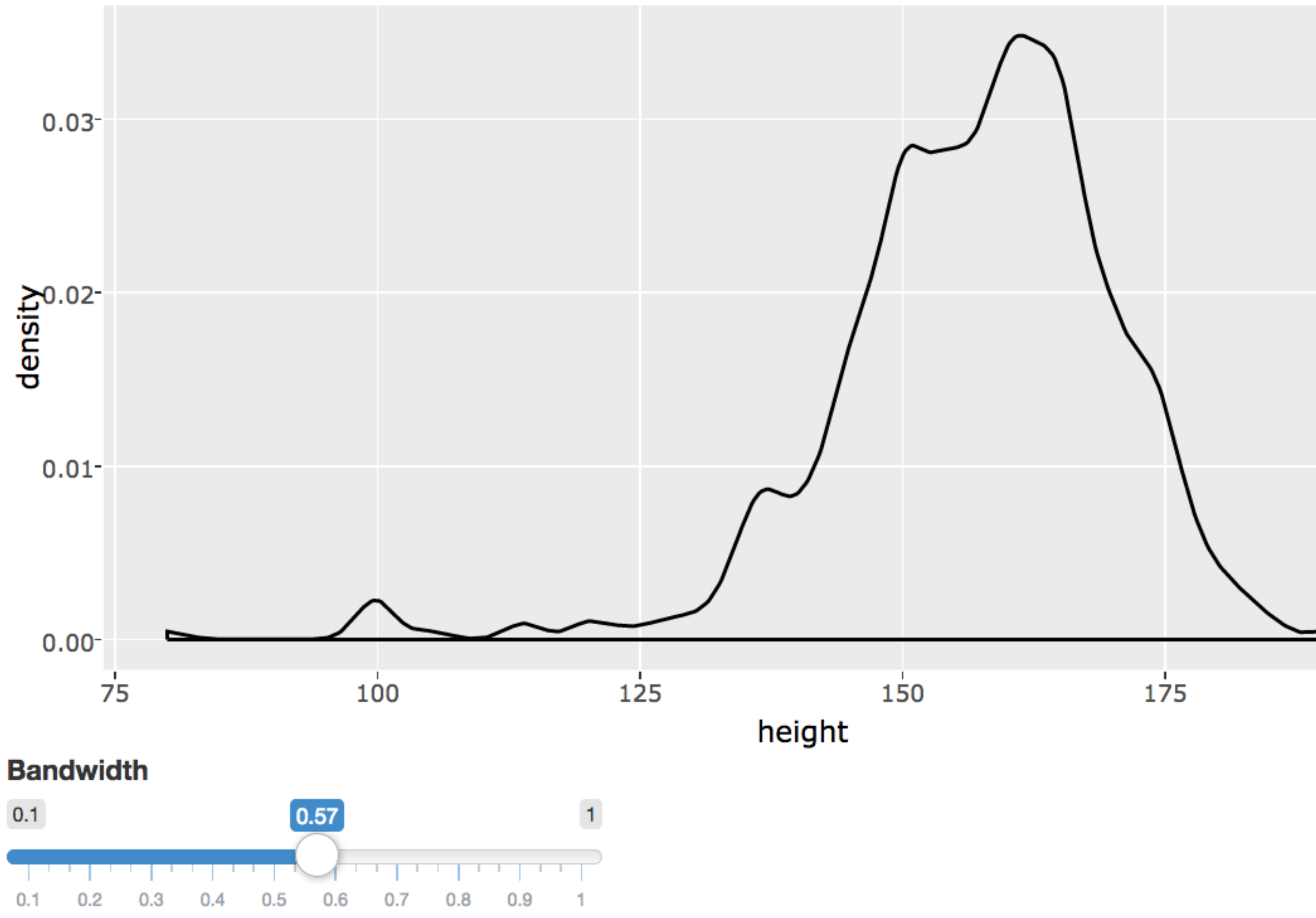


Figure 5.5: control density bandwidth using plotly (rendered with ggplot2) and shiny

Tool	Approximate number of lines of code	Redraws/reproduces entire plot	Scale of axes change	Plot type rendered
interactr (DOM+gridSVG+grid)	18	No	No	base R
ggvis	3	No	Yes	Vega
plotly+shiny	10	Yes	Yes	ggplot2 (via ggplotly)
shiny	10	Yes	Yes	base R

Table 5.1: A comparison table between existing tools and interactr on changing the bandwidth

of a density plot

From replicating each example, we find that the interactr package will not change the scale of its axes when the bandwidth is changed. This is because we are only updating the density line relative to the coordinates of the axes. In comparison to the rest of the tools, more lines of code are required to generate the same effect, and those involving shiny reproduce the entire plot every time.

Because many of these existing tools are still being developed, it is likely that they will resolve some of the limitations discussed in Section 2 in the future. However, they require the user to be very familiar with their APIs. An example of this is the plotly package that has expanded further into achieving linking between other types of plots and the ability to prevent redrawing when used with shiny. It requires the user to know both the plotly API, shiny, and the `plotlyProxy()` functions (Sievert 2017a) as briefly mentioned in Section 2.3.2. The same applies for interactr. There is still a long way to go before we are confident enough to claim that users would not need to know DOM, grid, and gridSVG.

5.4 Further problems and future directions

The problem of handling large number of individual objects in a plot remains unresolved as the browser cannot handle too many SVG elements at once. This is a general problem that occurs across all existing tools. 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 these elements as they are generally treated as a single object thus making it difficult to address sub-components.

There is potential in developing interactr further to try achieve complex interactions that are more useful in exploratory data analysis. Currently, it is only a proof-of-concept prototype and still undeveloped in many areas. Only a very small number of examples have been successful and a limited number of interactions have been implemented. There is still a need for a simpler and versatile system for users without compromising the flexibility in which the user can define interactions and interactr is a step along this path. Other possible ideas may that may be incorporated 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. It may become compatible with iNZight which also uses the grid system to produce its plots. However, this requires assessment on how the underlying grid objects are named and drawn before being implemented.

5.5 Conclusion

There is a need in expanding web interactive graphics to create better data visualisations for users. Despite having many tools available including plotly, ggvis, shiny and animint and many other packages that produce htmlwidgets, these generally produce standard interactive plots outside of R that are hard to customise. interactr provides a way of driving interactions without the need of learning web technologies while utilising R's power of statistical computing to aid changes in plots originally drawn in R. However, more assessment and development is required on building more informative interactive visualisations before it can catch up to the capabilities of older desktop applications and existing tools to be used in the future.

5.6 Additional resources

The interactr package is currently hosted on Github here.

To install interactr, you need to install DOM v0.4:

```r
install.packages("https://github.com/pmur002/DOM/archive/v0.4.tar.gz",
                 repos = NULL, type = "source")
devtools::install_github('ysoh286/interactr')
```

For more details about this project, visit this repository which contains code and additional notes.

Bibliography

Bostock, Michael, Vadim Ogievetsky, and Jeffrey Heer. 2011. "D3 Data-Driven Documents." *IEEE Transactions on Visualization and Computer Graphics* 17 (12). Piscataway, NJ, USA: IEEE Educational Activities Department: 2301–9. doi:10.1109/TVCG.2011.185.

CensusAtSchool. 2009. *CensusAtSchool 2009 Data Subset.* http://new.censusatschool.org.nz/resource/2009-censusatschool-data-subset/.

Chang, Winston, and Hadley Wickham. 2016. *Ggvis: Interactive Grammar of Graphics.* https://CRAN.R-project.org/package=ggvis.

Chang, Winston, Joe Cheng, JJ Allaire, Yihui Xie, and Jonathan McPherson. 2017. *Shiny: Web Application Framework for R.* https://CRAN.R-project.org/package=shiny.

Cheng, Joe. 2016. *Crosstalk: Inter-Widget Interactivity for Html Widgets.* https://CRAN.R-project.org/package=crosstalk, https://rstudio.github.io/crosstalk/.

———. 2017a. *Async Programming in R and Shiny.* https://medium.com/@joe.cheng/async-programming-in-r-and-shiny-ebe8c5010790.

———. 2017b. *Promises: What the Package Does (Title Case).*

Cheng, Joe, Bhaskar Karambelkar, and Yihui Xie. 2017. *Leaflet: Create Interactive Web Maps with the Javascript 'Leaflet' Library.* http://rstudio.github.io/leaflet/.

Cook, Dianne, and Deborah F. Swayne. 2007. *Interactive and Dynamic Graphics for Data Analysis - with R and Ggobi.* Use R. Springer. doi:10.1007/978-0-387-71762-3.

Crockford, Douglas. 2008. *JavaScript: The Good Parts.* O'Reilly Media, Inc.

Elliott, Tom, and Marco Kuper. 2017. *INZight: INZight Gui for Data Exploration and Visualisation.*

Gesmann, Markus, and Diego de Castillo. 2011. "GoogleVis: Interface Between R and the

Google Visualisation Api." *The R Journal* 3 (2): 40–44. https://journal.r-project.org/archive/2011-2/RJournal_2011-2_Gesmann+de~Castillo.pdf.

Hafen, Ryan. 2016. *Rbokeh Version 0.5.0 Released.* http://ryanhafen.com/blog/rbokeh-0-5-0.

Hafen, Ryan, and Inc. Continuum Analytics. 2016. *Rbokeh: R Interface for Bokeh.* https://CRAN.R-project.org/package=rbokeh.

Heckmann, Mark. 2013. *Sending Data from Client to Server and Back Using Shiny.* https://ryouready.wordpress.com/2013/11/20/sending-data-from-client-to-server-and-back-using-shiny/.

Hocking, Toby Dylan, Susan VanderPlas, Carson Sievert, Kevin Ferris, Tony Tsai, and Faizan Khan. 2017. *Animint: Interactive Animations.* https://github.com/tdhock/animint.

Ihaka, Ross, and Robert Gentleman. 1996. "R: A Language for Data Analysis and Graphics." *Journal of Computational and Graphical Statistics* 5 (3): 299–314.

Jacobson, Daniel, Greg Brail, and Dan Woods. 2011. *APIs: A Strategy Guide.* O'Reilly Media, Inc.

Kunst, Joshua. 2017. *Highcharter: A Wrapper for the 'Highcharts' Library.* https://CRAN.R-project.org/package=highcharter, https://github.com/jbkunst/highcharter.

Murray, Scott. 2013. *Interactive Data Visualization for the Web.* O'Reilly Media, Inc.

Murrell, Paul. 2011. *R Graphics.* 2nd ed. Boca Raton, FL, USA: CRC Press, Inc.

———. 2015. *GridGraphics: Redraw Base Graphics Using 'Grid' Graphics.* https://CRAN.R-project.org/package=gridGraphics.

———. 2016a. *An Introduction to the 'Dom' Package.* https://www.stat.auckland.ac.nz/~paul/Reports/DOM/Intro/DOM-Intro.html.

———. 2016b. *DOM: Interact with Web Browser Dom.*

Murrell, Paul, and Simon Potter. 2012. *Working with the gridSVG Coordinate System.* https://www.stat.auckland.ac.nz/~paul/Reports/gridSVGcoords/coordinates.html.

———. 2014. "The gridSVG Package." *R Journal* 6 (1): 133–43. https://journal.r-project.org/archive/2014/RJ-2014-013/RJ-2014-013.pdf.

———. 2017. *GridSVG: Export 'Grid' Graphics as Svg.* https://CRAN.R-project.org/package=gridSVG.

Paul Murrell, Jeroen Ooms, and JJ Allaire. 2015. *Recording and Replaying the Graphics Engine*

Display List. https://www.stat.auckland.ac.nz/~paul/Reports/DisplayList/dl-record.html.

RStudio. 2017a. *Interactive Plots*. https://shiny.rstudio.com/articles/plot-interaction.html, https://shiny.rstudio.com/articles/plot-interaction-advanced.html.

———. 2017b. *Selecting Rows of Data*. https://shiny.rstudio.com/articles/selecting-rows-of-data.html.

———. 2017c. *Shiny*. https://shiny.rstudio.com/.

———. 2017d. *Shiny Server*. https://www.rstudio.com/products/shiny/shiny-server/.

Sievert, Carson. 2017a. *Plotly 4.7.1 Now on Cran*. http://moderndata.plot.ly/plotly-4-7-1-now-on-cran/.

———. 2017b. *Plotly for R*. https://plotly-book.cpsievert.me/.

Sievert, Carson, Chris Parmer, Toby Hocking, Scott Chamberlain, Karthik Ram, Marianne Co rvellec, and Pedro Despouy. n.d. *Plotly: Create Interactive Web Graphics via 'Plotly.js'*. https://plot.ly/r, https://cpsievert.github.io/plotly_book/, https://github.com/ropensci/plotly.

Theus, Martin. 2002. "Interactive Data Visualization Using Mondrian." *Journal of Statistical Software, Articles* 7 (11): 1–9. doi:10.18637/jss.v007.i11.

Trifacta. 2014. *Vega*. https://vega.github.io/vega/.

Unwin, Antony, Martin Theus, and Heike Hofmann. 2006. *Graphics of Large Datasets: Visualizing a Million (Statistics and Computing)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Urbanek, Simon, and Tobias Wichtrey. 2013. *Iplots: IPlots - Interactive Graphics for R*. https://CRAN.R-project.org/package=iplots.

Vaidyanathan, Ramnath. 2013. *RCharts: Interactive Charts Using Javascript Visualization Libraries*.

W3C. 2009. *Document Object Model (Dom)*. https://www.w3.org/DOM/.

———. 2016. *HTML & Css*. https://www.w3.org/standards/webdesign/htmlcss.

Wickham, Hadley. 2009. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. http://ggplot2.org.

———. 2016. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

http://ggplot2.org.

Wickham, Hadley, and Winston Chang. 2016. *Interactivity.* http://ggvis.rstudio.com/interactivity.html.

Wilkinson, Leland. 2005. *The Grammar of Graphics (Statistics and Computing).* Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Xie, Yihui. 2016. *DT: A Wrapper of the Javascript Library 'Datatables'.* http://rstudio.github.io/DT.