Web interactive plots in **R**

Abstract

TODO: WRITE ABSTRACT once everything is finalized

Executive Summary

Interactive statistical graphics have been successful through desktop applications since the nineties, 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 do not possess the all the capabilities that these desktop applications have.

The purpose of this research is to try 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** and **shiny**. 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 for users.

A different approach was taken by investigating lower level tools, specifically **gridSVG** and **DOM** to try solve these limitations. 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 are difficult to extend further.

To solve this, a new approach was developed by combining lower level tools (grid, gridSVG and DOM) to create **interactr**, an R package 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**. This stands out as it brings interactivity to plots that were originally generated in R. However, it only serves as a proof-of-concept. There are several limitations including that only objects originally drawn in R can be used and that only a few interactions have been achieved via targeting a single element. 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. The interactr package may become a solution for allowing users to create and control interactions more easily on plots generated in R and thus for iNZight, but requires more attention and development for creating more sophisticated and stable visuals.

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) have suggested that there are 3 broad components: querying, selection and linking, and varying plot characteristics. We can also split it into two broader categories - ‘on-plot’ and ‘off-plot’ interactivity. We focus on ‘on-plot’ interactivity, where a user can interact directly on the plot to query, select and explore the data.

R (Ihaka and Gentleman (1996)) is a powerful open source tool for generating flexible static graphics. However, it is not focused on interactivity. Previously, there have been different programs to help create interactive plots to aid analysis including ggobi(Swayne et al, 2007), cranvas(Xie et al, 2013), iplots(Urbanek, 2007), and Mondrian(Theus (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 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. Cascading style sheets (known as CSS) is the language that controls how elements look and are presented on a web page (such as color, shape, strokes and fills, borders). These can be used to define how specific types of elements are rendered on the page. JavaScript is the main programming language for the web, 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' (MDN, 2017). 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 routines, protocols, and tools for building software applications' (MISSING REFERENCE). 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 through the DOM. This cannot be done with a raster image, as a raster image (PNG or JPEG) is treated as an entire element.

1.3 Motivational problem

The main motivation for this project stemmed upon whether there is a more intuitive way to generate simple interactive visuals from R without learning many tools. The solution could ideally be used to advance features in iNZight, 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 (Chapters 2 and 3) before building a viable solution (Chapter 4) that could potentially solve this 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 for generating interactive tables, Leaflet (Cheng and Xie, 2017) for rendering interactive maps and many popular graphing libraries including Highcharter(Kunst, 2017), rbokeh(Hafen, 2014), googleVis and the rCharts package. Other tools use R rather than JavaScript to drive interactivity, including ggvis and shiny. The few that are discussed in this section in detail are the plotly, ggvis, shiny, and animint packages.

briefly define what an htmlwidget DOES rather than what it is

2.1 plotly

plotly is a JavaScript graphing library built upon D3(Bostock, 2011). The plotly package in R calls upon this library to render web interactive plots. One of the reasons the plotly R package is useful is that it can automatically convert plots rendered in the very popular ggplot2 package into interactive plots. 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.

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")
```

The purpose of plotly is to provide a convenient way of visualising data. With its simple API in R, we can easily generate a standard plot that can be shared and saved as an HTML web page. However, like many other htmlwidgets, we find that these solutions only provide interactive plots quickly to the user with basic functionalities such as tooltips, zooming and subsetting. We can build upon layers of plot objects but they cannot be pulled apart or modified without 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.

TODO: plotly has been updated... - include an example where customisation is difficult (reference Sievert's book + example)

2.1.0.1 Extending interactivity between htmlwidgets with crosstalk

Crosstalk (Cheng, 2016) is an add-on package that allows htmlwidgets to cross-communicate with each other. As Cheng (2016) explains, it is designed to link and co-ordinate different views of the same data. Data is converted into a R6 'shared' object, which has a corresponding key for each row observation. When selection occurs, crosstalk communicates which keys have been selected and the bounded htmlwidgets 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

```r
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 only supports are linked brushing (Figure 2.2) and filtering (Figure 2.3) that can only be done on data in a 'row-observation' format. This means that it cannot be used on aggregate data such as linking a histogram to a scatterplot, as illustrated in Figure 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.

Figure 2.4: Scatterplot matrix

Furthermore, crosstalk only supports a limited number of htmlwidgets so far - plotly, DT and Leaflet. This is because the implementation of crosstalk is relatively complex. From a developer's point of view, it requires creating bindings between crosstalk and the htmlwidget itself and customizing interactions accordingly on how it reacts upon selection and filtering. Despite being under development, it is promising as other htmlwidget developers (notably, Kunst with highcharter and Hafen 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 (Wickham, Chang 2014). This package utilises the Vega JavaScript library to render its plots and uses shiny to drive some of its interactions. These plots follow the "Grammar of Graphics" and aim to be an interactive visualisation tool for exploratory analysis. This package has an advantage over htmlwidgets as it expands upon using statistical functions for plotting, such as `layer_model_predictions()` for drawing trend lines using statistical modelling (see Figure 2.6). Furthermore, because some of the interactions are driven by Shiny, we can add 'inputs' that look similar to Shiny such as sliders and checkboxes to control and filter the plot, but also have the power to add tooltips as seen in Figure 2.5.

Figure 2.5: basic ggvis plot with tooltips

```r
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 control over where these inputs such as filters and sliders can be placed on the page. We also cannot save these interactions to a web page like htmlwidgets as they 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.

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

2.3 shiny

shiny (RStudio, 2012) is an R package that builds web applications. It provides a connection of using R as a server and the browser as a client, such that R outputs are rendered on a web page. This allows users to be able to code in R without the need of learning the other main web technologies. A Shiny application 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).

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.

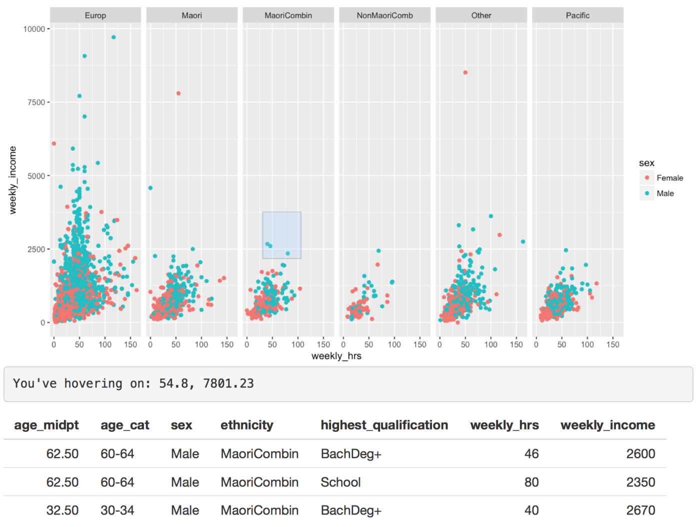
Figure 2.8: A simplistic shiny application that has a slider to control the smoothness of the trend line

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. This allows us to be able to link different views of data easily. Furthermore, RStudio has provided ways to be able to host and share these shiny apps via a Shiny server. However, we are still limited in the sense that for every time we launch a Shiny app, we do not have access to R as it runs that session. Additionally, shiny has a tendency to redraw entire objects whenever an 'input' changes as seen in Figure 2.7. This may lead to unnecessary computations and may slow down the experience for the user. Despite this, it remains a popular tool for creating interactive visualisations.

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 extend interactivity in htmlwidgets and other R packages.

2.3.1 Interactivity with shiny alone

Shiny can provide some interactivity to plots. 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`).

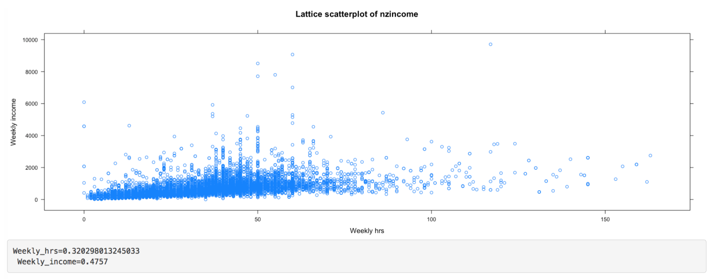


```
You've hovering on: 54.8, 7801.23
```

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. 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 may be complex to develop.



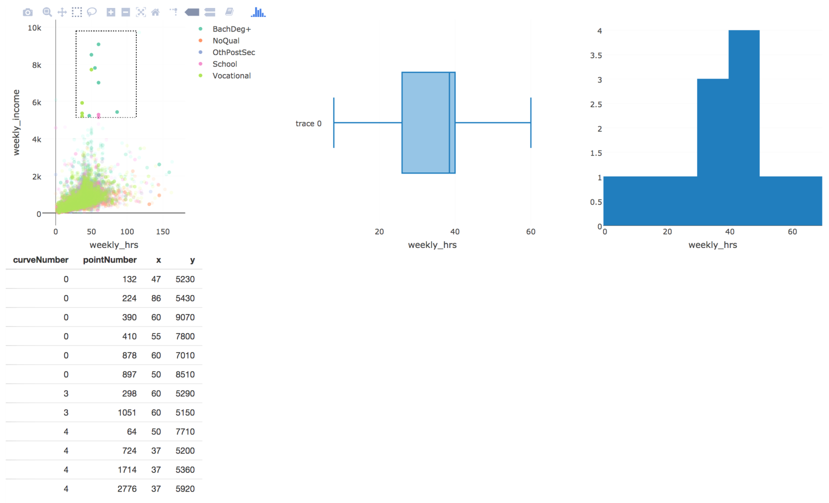
```
Weekly_hrs=0.320298013245033
 Weekly_income=0.4757
```

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

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

2.3.2 Linking plotly or ggvis with Shiny

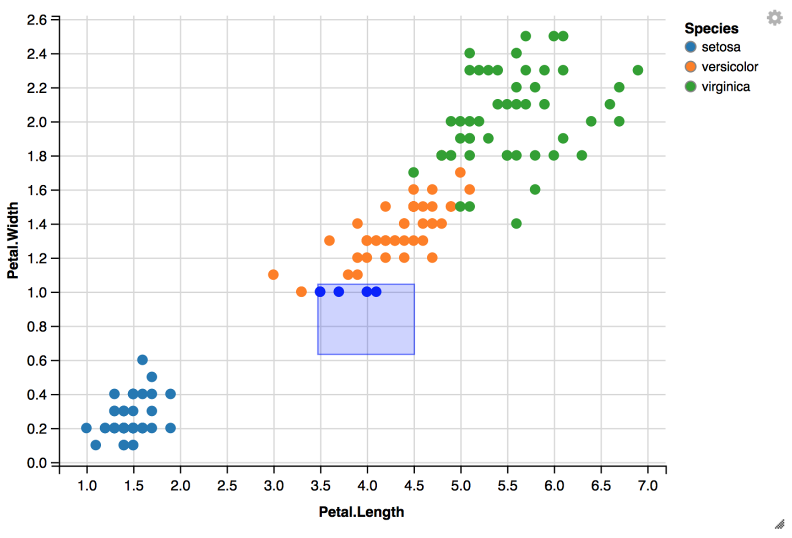
Although Shiny is great at facilitating interactions from outside of a plot, it is limited in facilitating interactions within a plot. It does not have all the capabilities that plotly provides. When we combine the two together, more interaction can be achieved with less effort.

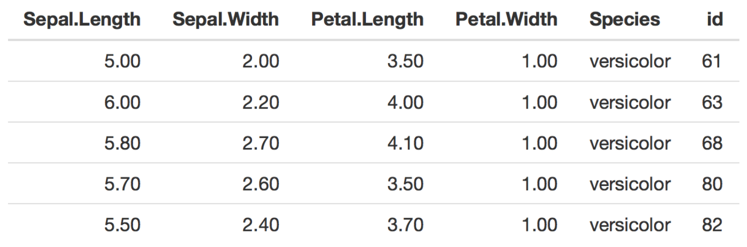


curveNumber	pointNumber	x	y
0	132	47	5230
0	224	86	5430
0	390	60	9070
0	410	55	7800
0	878	60	7010
0	897	50	8510
3	298	60	5290
3	1051	60	5150
4	64	50	7710
4	724	37	5200
4	1714	37	5360
4	2776	37	5920

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

Most 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()`. By combining it with Shiny, we are able to link different plots together and to the data itself that is displayed as a table below.These in-plot interactions are very similar to what Shiny provides for base plots and ggplot2. They work well on scatter plots, but not on other kinds of plots that plotly can provide. These can help generate or change different outputs on the page, but not within themselves. By combining the two together, we get 'on-plot' functionalities from the HTMLwidget, with 'off-plot' driven interactions from Shiny. Similarly, when we can combine ggvis and Shiny together we get similar results as seen in Figure 2.4. ggvis has its own functions (`ggvisOutput()` and `linked_brush()`) that allow for similar interactions to be achieved.





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

2.4 animint

The animint package... **TODO!**

From assessing these tools, it is difficult to expand on-plot interactions as it requires an in-depth knowledge of the tool itself and how these interactions are defined. 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. Next, we look at how we could achieve specific on-plot interactions by combining JavaScript with lower levels tools.

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 involved generally do not preserve plots that are drawn in R and continually redraw each time the user interacts with it. This section discusses how we can use two lower level packages, gridSVG and DOM, along with a knowledge of web technologies to incorporate interactions into R plots and prevent redrawing.

One approach to avoid redrawing in plots 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 be related back to data. In R, we can use the gridSVG package.

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 data 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, 2011).

```
library(grid)
library(gridSVG)
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")
```

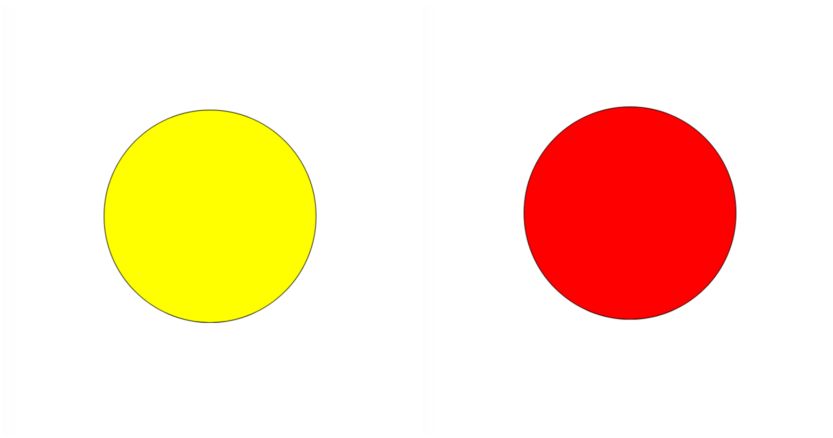
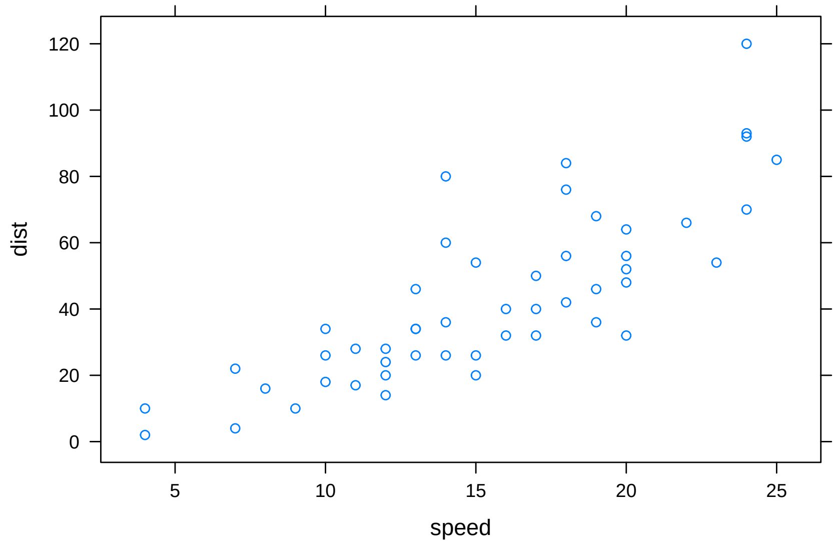


Figure 3.1: An interactive circle made using gridSVG - when the user hovers over the circle, it will turn red (shown on the right)

In Figure 3.1, the circle has been originally drawn in R 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 that gridSVG has is the ability to translate between data and SVG coordinates. 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 in the plot (Murrell and Potter, 2012). We can use this information to convert data to SVG coordinates.

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



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

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

gridSVG::gridSVGCoords(coords)
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
```

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

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

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

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

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

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

In the example above, we have drawn a plot using the cars dataset and exported its coordinates system with its corresponding SVG. Suppose we have a new point at (4, 5). This can be easily translated into SVG co-ordinates and back using the functions `viewportConvertX` and `viewportConvertY` along with the correct viewport identified as `panel` where the points lie on the plot. Here, the native co-ordinates (4, 5) have been translated as (77.9213158, 70.74) in the SVG co-ordinate system. This can be further added to the plot as we have the co-ordinates in the SVG space using JavaScript.

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 `graphics` cannot be directly converted. Another point to note is that the process of converting elements to SVG becomes slow when there are many elements to render.

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 the plot.

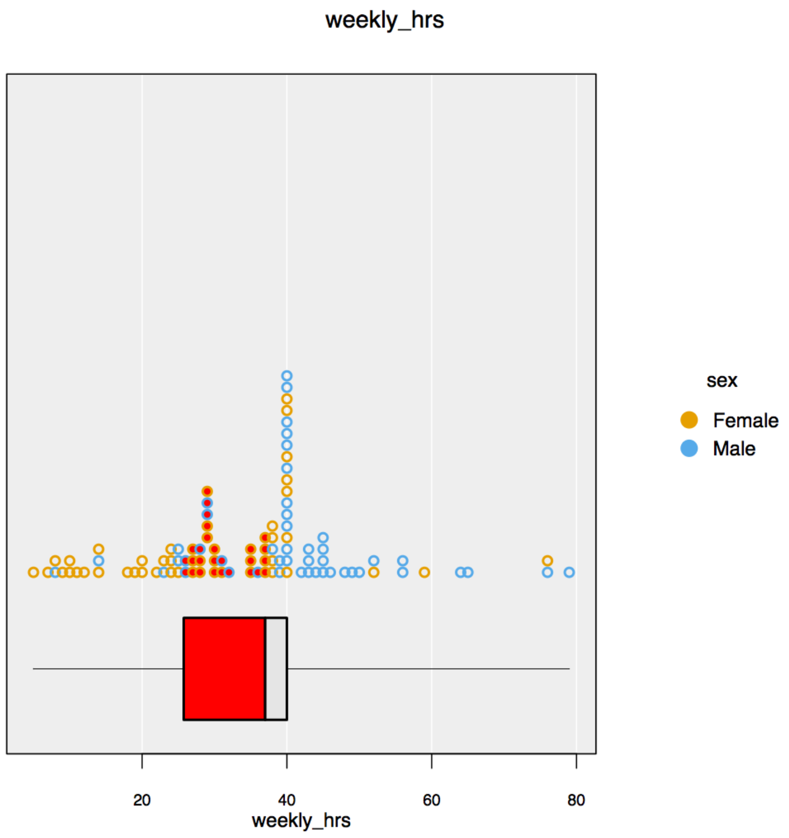


Figure 3.2: An example of a customised box plot interaction on an iNZight plot using gridSVG, JavaScript

One such example is highlighting part of a box plot to show certain values between the median and the lower quartile (Figure 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 rendering graphs in SVG, we are unable to identify which elements to target and add interactions to.

3.1.2 Preventing redraws in Shiny using JavaScript messages and gridSVG

As mentioned in Chapter 2, one of the downsides for using shiny (with or without htmlwidgets) is its nature to redraw plots every time an input changes. In the case of R plots that are rendered using `renderPlot` function, redrawing is required because the plot is viewed as a raster image. This means that we cannot specifically target elements on the page as the image is viewed as a single object. A way to get around this is to render the plot in SVG using gridSVG.

A complication to this is that we can no longer use the usual input and output functions that link everything on the page. To control specific elements on the page, we need a different way of passing data between the browser and back to R using JavaScript. shiny provides a way of sending messages through this channel using two JavaScript functions: `shiny.onInputChange()` and `shiny.addCustomMessageHandler()`. 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 that 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 the trend line 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.

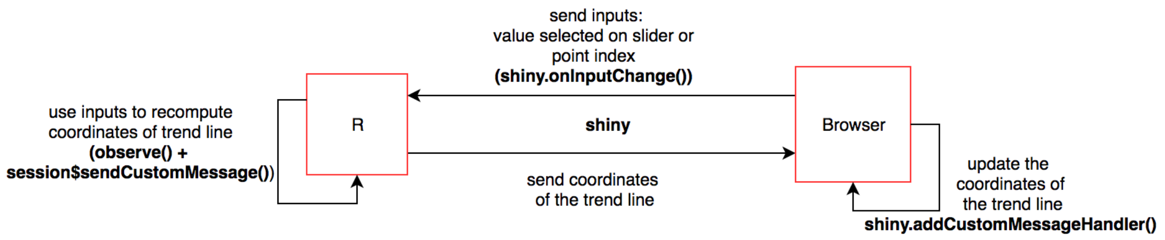
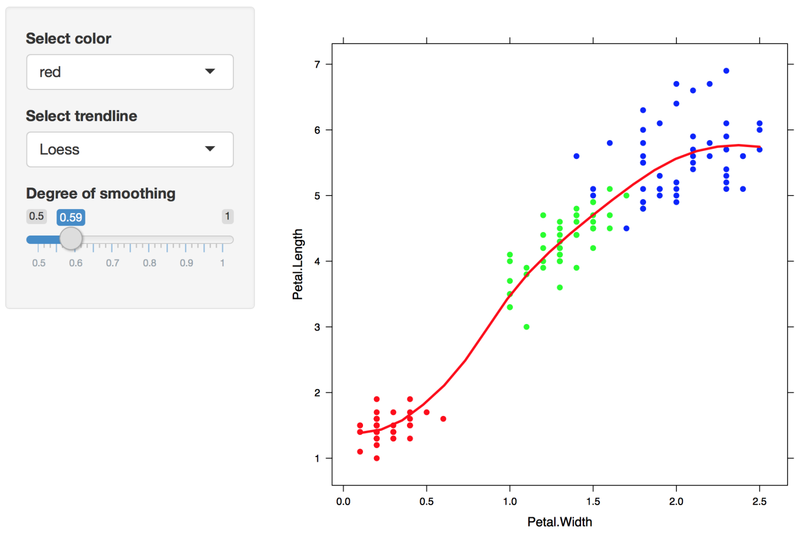


Figure 3.3: Diagram of how things work using shiny's JavaScript functions in Figure 3.4

In Figure 3.3, we can pass the value of the slider back to R (either through inputs or `shiny.onInputChange`) which is then used to recalculate the x and y co-ordinates of the smoother. 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.


Select color
red

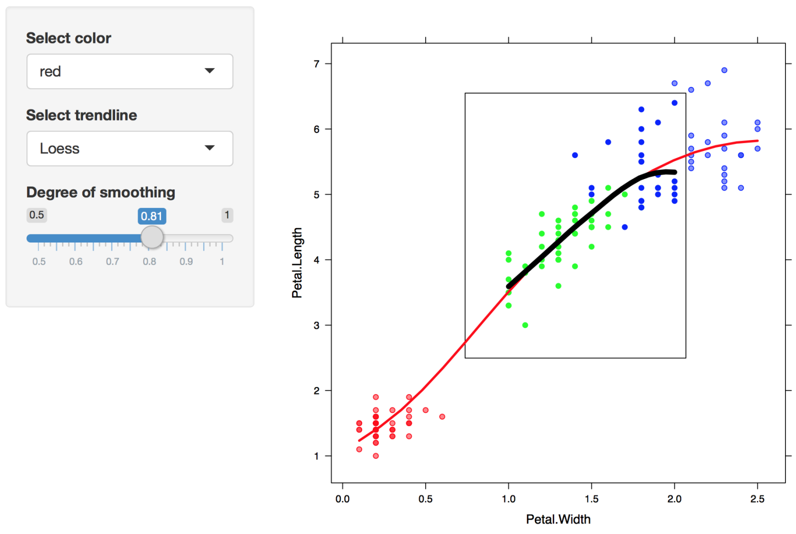
Select trendline
Loess

Degree of smoothing
0.59

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. Rather than redrawing the entire plot, we have only changed the trend line. However, this method requires the knowledge of JavaScript and the limitations of how much information can be sent through are unknown as it is uncommonly used.

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


Select color

red

Select trendline

Loess

Degree of smoothing

0.5 0.81 1

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

3.2 DOM package

The DOM package (Murrell, 2016) is an R package that allows for DOM (Document Object Model) requests to be sent from R to a browser. It aims to provide a basis for using the web browser as an ‘interactive output device’ (Murrell, 2016a).

Using DOM 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
#draw circle using grid and generate SVG:
grid.circle(x = 0.5, y = 0.5, r = 0.25,
            name = "circle.A", gp = gpar(fill = "yellow"))
svg <- grid.export(NULL)$svg

library(DOM)
#setting up and adding the circle to the page
page <- htmlPage()
appendChild(page,
            child = svgNode(XML::saveXML(svg)),
            ns = TRUE,
            response = svgNode())

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

```r
setAttribute(page, circle, "fill", "red")
```

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

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

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.

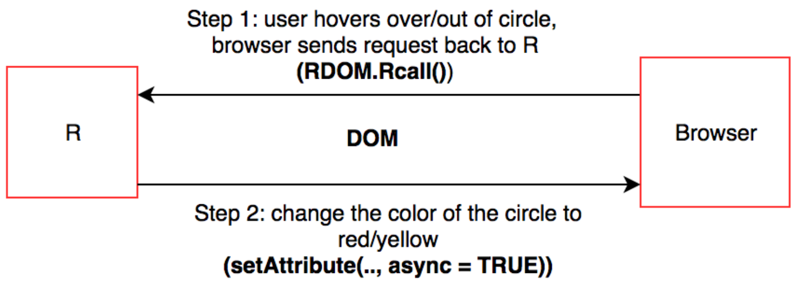

Step 1: user hovers over/out of circle, browser sends request back to R (RDOM.Rcall())
R
DOM
Browser
Step 2: change the color of the circle to red/yellow (setAttribute(.., async = TRUE))

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

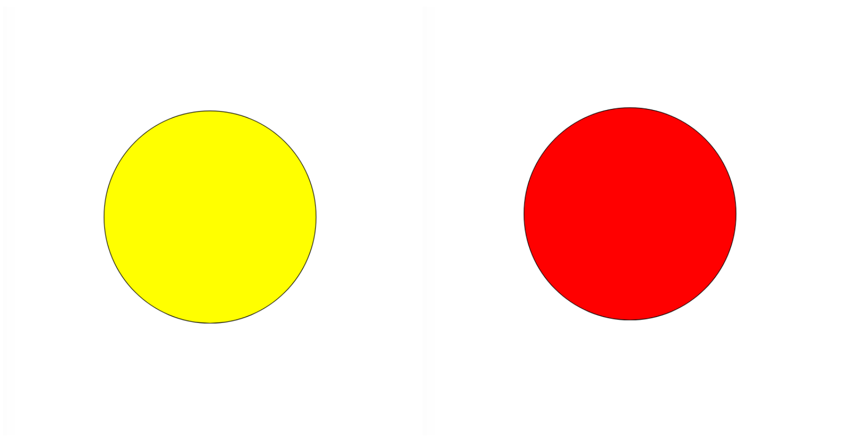


Figure 3.7: DOM example of Figure 3.1 - when hovered, the circle turns red (right)

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

```r
#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)")
```

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.

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.

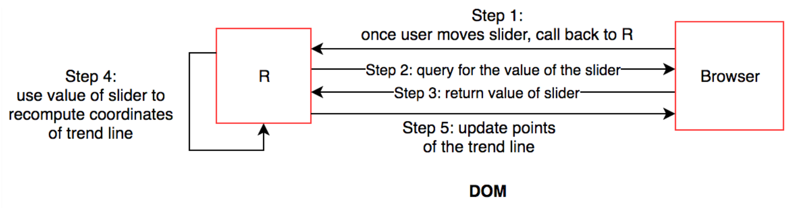


Figure 3.8: Steps on how a trend line can be altered using the DOM package

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 are 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.

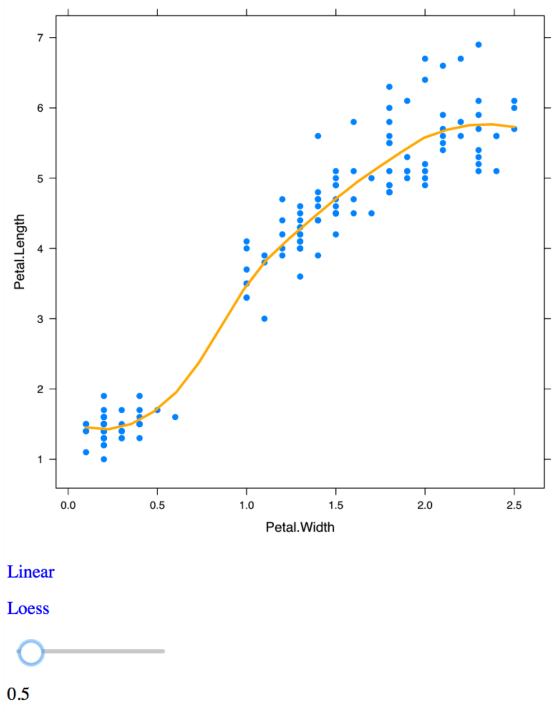


Figure 3.9: DOM example of Fig 3.4 for changing a trendline using a slider

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. However, the DOM example 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. 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.

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

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 interactive web plots are easier to use but are complex to modify and extend further.

Chapter 4

Designing a more flexible way of producing simple interactions

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

4.1 The main idea

In each of the previous examples defined, they all have a certain pattern. In order to define a single interaction, it requires the need to know which SVG element to target, what type of interaction or event is to be attached, and how to define what happens when an interaction occurs. This idea can be broken down into 5 simple steps:

- Draw the plot in R
- Identify elements to interact with
- Define interactions and process, which elements to target
- Attach interactions and define events
- Send defined interactions and plot to the browser

Figure 4.1: Diagram illustrating the basic process of creating interactions

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

TODO: some key points to note, but should these be mentioned in chapter 3?

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

4.2 Examples

4.2.1 Linking box plots

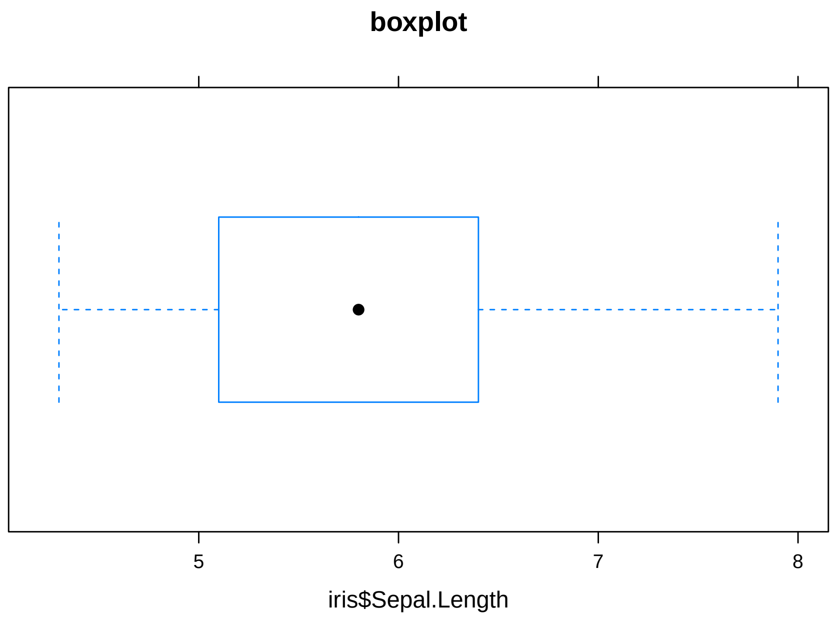
The goal in this example is to link the interquartile range of the box plot to a scatter plot, followed by a density plot. When the user clicks on the box plot, it highlights the range of the box plot on the other respective plots.

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

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

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

```
listElements(bw)
```



```
## 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
```

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

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

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

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

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

In this step, we have sent the plot we drew in R to a new web page. We can also attach an interaction to the `box` element we identified earlier to attach the hover interaction to. On the page (Figure 4.1), we see that when the user hovers over the box, the box turns red.

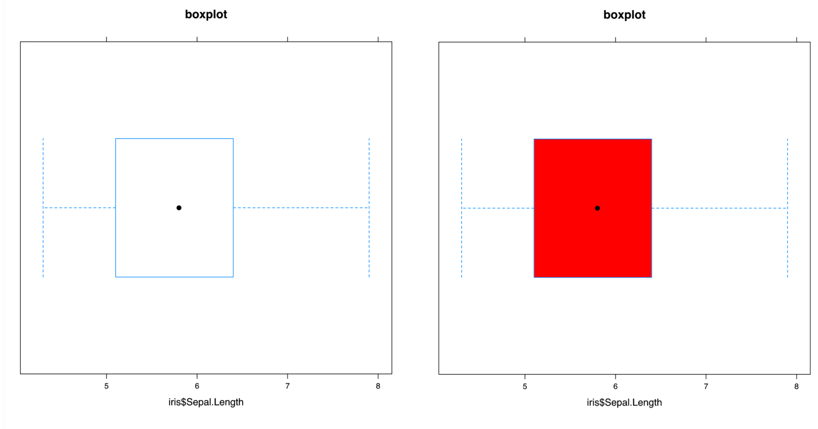


Figure 4.1: Boxplot with hover interaction

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 this process - if there is anything

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

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

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

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

Here, we see that the box plot and the scatter plot we drew in R are now on the same web page (Figure 4.2).

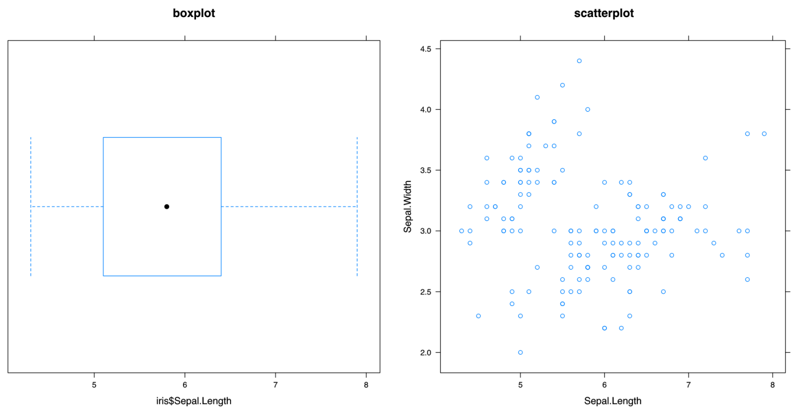


Figure 4.2: Boxplot and scatter plot on the same web page

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

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

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

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

Here, we insert the function name to run when a ‘click’ is done. Next, we append this defined interaction to the box element, so that when we click on the box, the points in the scatterplot that lie within that range should light up in red. This is shown in Figure 4.3.

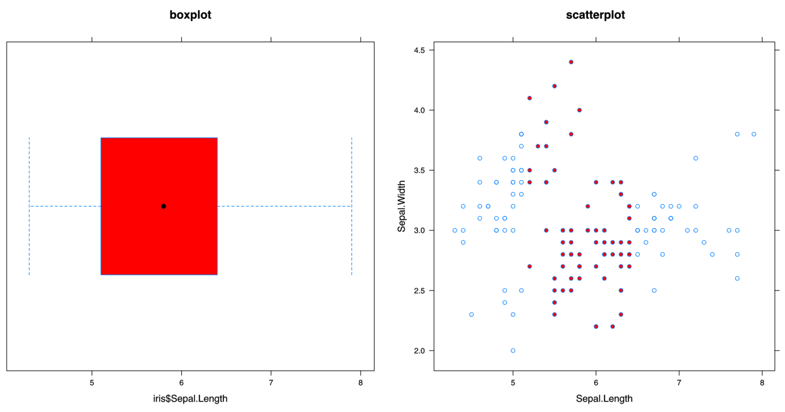


Figure 4.3: Click on box plot to light up points on scatter plot

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

We begin by drawing the box plot of boys heights.

```r
#get dataset off web:
census <- read.csv("http://new.censusatschool.org.nz/wp-content/uploads/2016/08/CaS2009_subs
         header = TRUE)

#subset the first 500 values
census <- census[1:500, ]

# separate girls and boys:
boys <- census[census$gender == "male", ]
girls <- census[census$gender == "female", ]

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

Next, we add the scatterplot to the page.

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

Then, we add the density plot of girls heights.

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

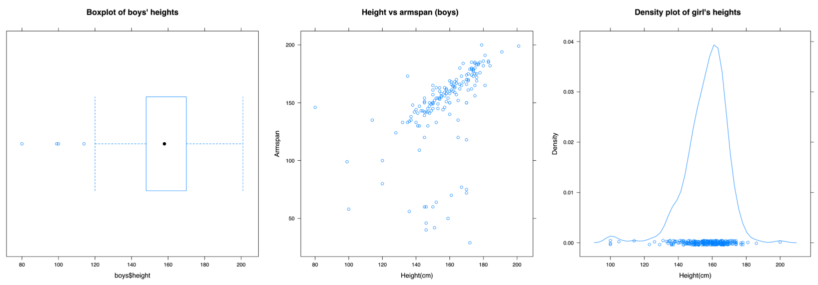


Figure 4.4: All three plots on the same web page

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

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

```
highlightRange <- function(ptr) {

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

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

  pt <- convertXY(xval, yval, panel)

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

  # highlight points in scatter plot:
  index <- which(min(range) <= boys$height
                 & boys$height <= max(range)
                 & !is.na(boys$armspan))
  # armspans may be missing
  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 4.5.

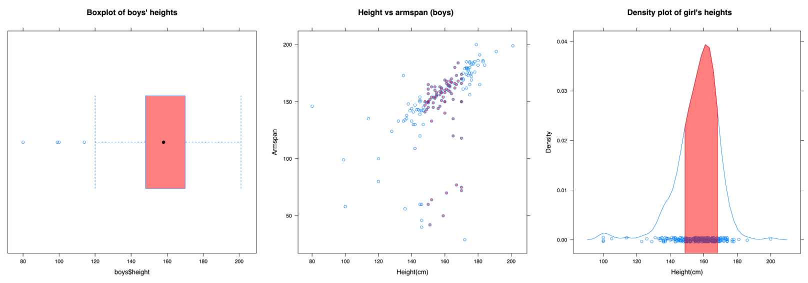


Figure 4.5: A single click on the box plot links the density and scatterplot together

4.2.2 Changing a trendline

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

Here, it becomes more complex as it requires information to be sent and queried in both ways: from R to the browser and from the browser back to R.

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

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

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

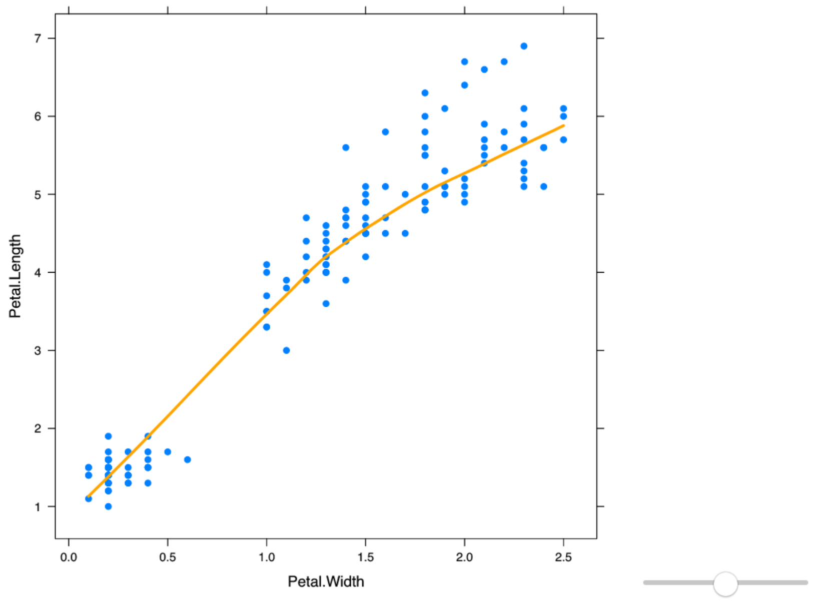


Figure 4.6: Plot with slider

To define what happens when the slider moves, the user can write a function with the argument `value`. This passes the value of the slider (as a character) from the web page back to R. Here, we can use the value to control the span of the trend line. To translate the new x and y values of the trend line, we need to convert them into the right scale 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. Here, we have a special function called `sliderCallback` for this. This redefines and creates the entire function that is now called `sliderValue`.

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

Finally, we can link this certain interaction 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 4.7.

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

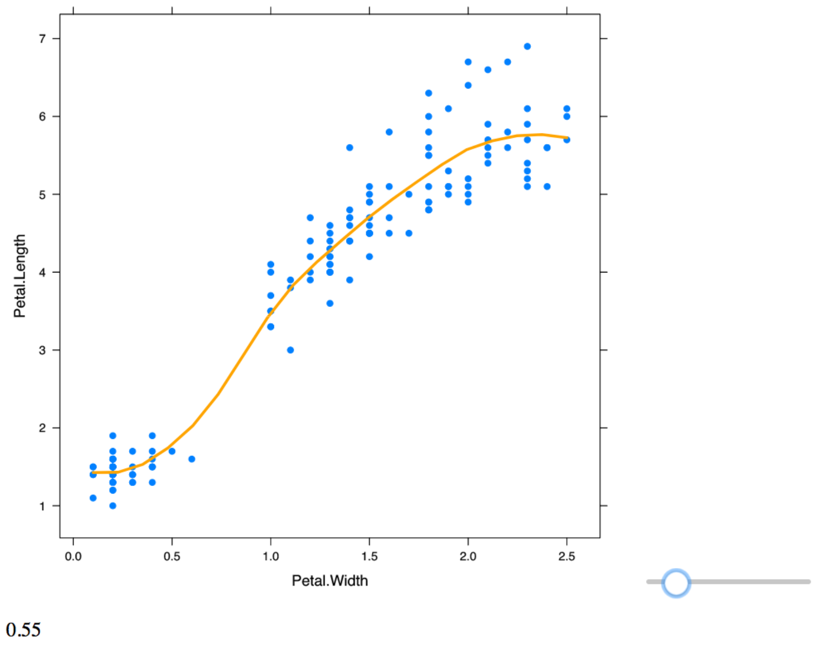


Figure 4.7: Plot with slider that controls the smoothness of the trend line

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.

```
panel <- findPanel("plot_01.xyplot.points.panel.1.1")
addLine("newSmooth", panel, 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, the function needs to be defined to be able to compute this new smoother.

```r
#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, panel)
  } 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 kind of interaction, we can pass our defined function through to the `addSelectionBox` function which adds on the selection box and links the defined interactions together to compute the new smoother.

```r
#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")
```

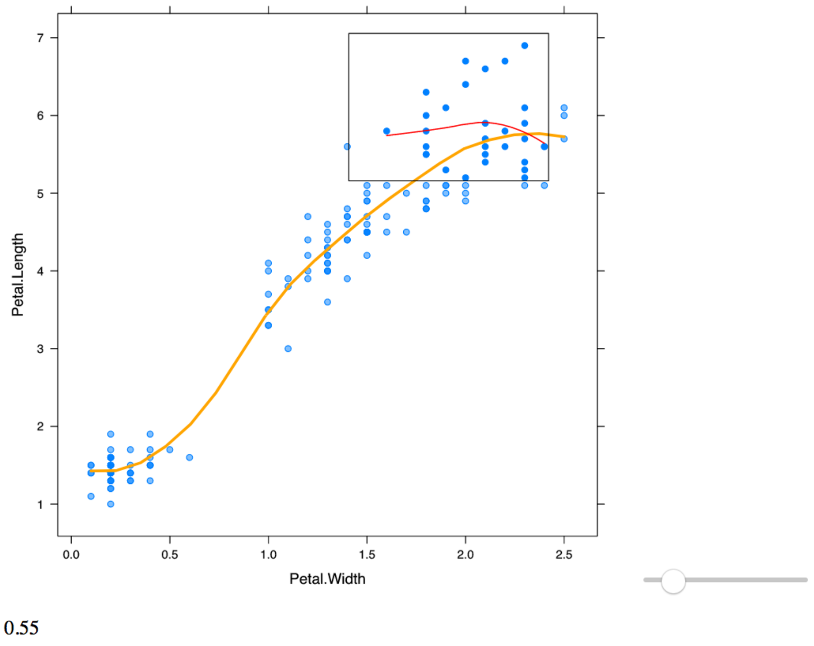


Figure 4.8: Plot that has a selection box feature that draws a separate smoother

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

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

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

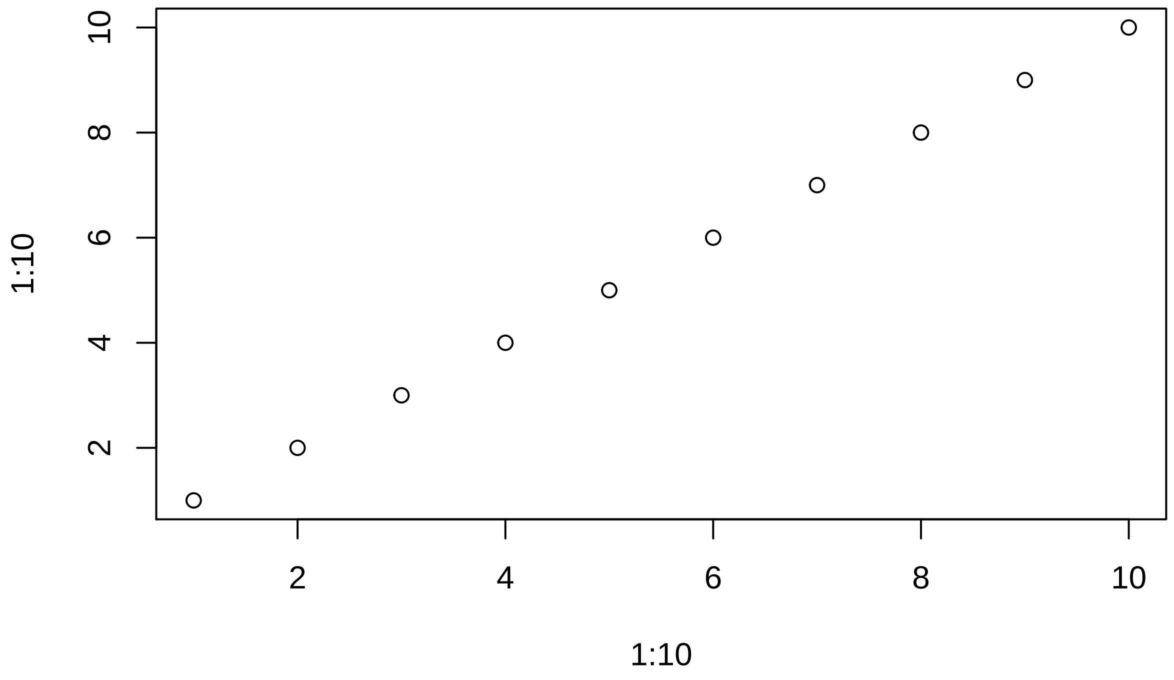


Figure 4.9: An eample of a graphics plot that is converted into a grid plot using gridGraphics

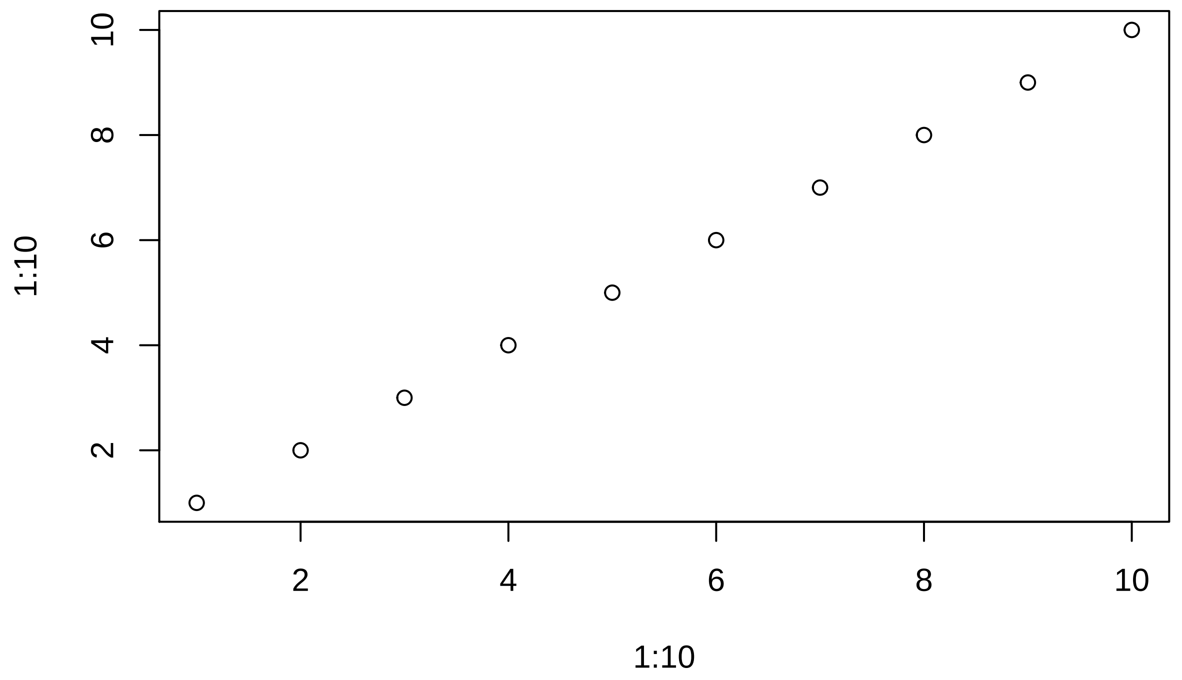


Figure 4.10: An eample of a graphics plot that is converted into a grid plot using gridGraphics

```
## 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 4.9). 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 that arises is that when we plot or try save it into a variable, it does not plot to the graphics device. To solve this, we can use `recordPlot` to record the plot that has been drawn to further process it.

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`.

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

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

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

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

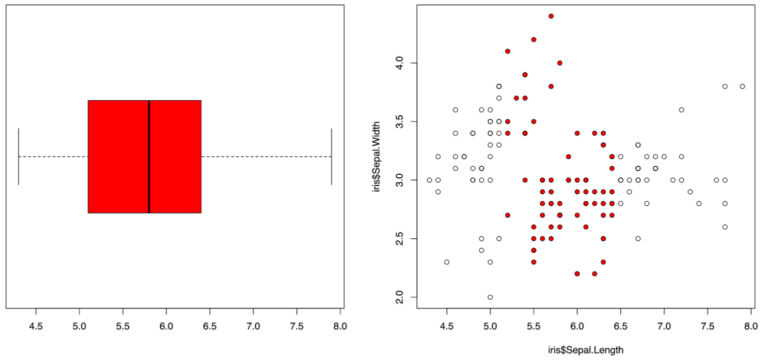


Figure 4.11: Box plot example replicated using a graphics plot

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

4.3.2 ggplot2

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

```
library(ggplot2)
p <- ggplot(data = iris, aes(x = "", y = Sepal.Length)) + geom_boxplot()
p.elements <- listElements(p)
box <- findElement("geom_polygon.polygon")
interactions <- list(hover = styleHover(attrs = list(fill = "red",
                                                      fill.opacity = "1",
                                                      pointer.events = "all")))
draw(p, box, interactions, new.page = TRUE)
```

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

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

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

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

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

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

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

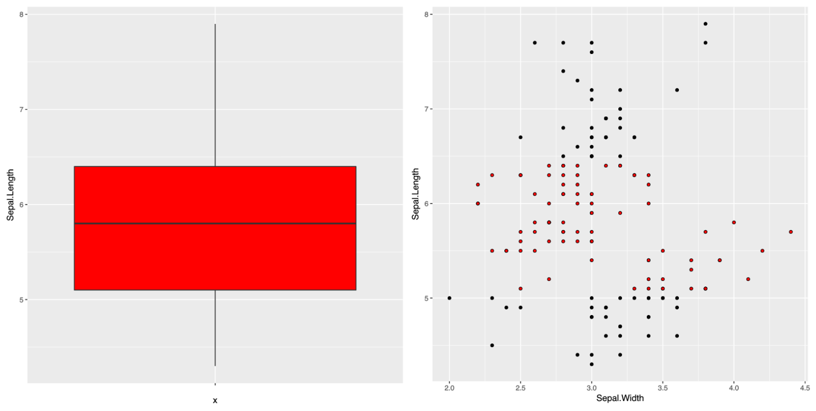


Figure 4.12: Box plot example replicated using a plot rendered with ggplot2

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

This demonstrates that there is a possible way of achieving interactions with ggplot2 and there is potential for the **interactr** to plug into different R plotting systems. Consequently it also means that when we assess the compatibility of different plotting systems, we need to take into account of possible detours that could occur. Because this is a simplistic example, it may become more complex when we try to achieve more sophisticated interactions.

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

Chapter 5

Discussion

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

5.1 Limitations

Firstly, there are limitations that arise from using the gridSVG and DOM packages. 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 before being 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. To note, work is proceeding to make gridSVG faster.

As `interactr` is mainly built upon the `DOM` package, many of its limitations highlighted in Chapter 3 are carried forward. This includes the shareability and accessibility of these plots, where they are generated for a single user in a single session only. Furthermore, because the `DOM` package is still under development, it cannot be used for production purposes yet. Just as how shiny applications would require a shiny server to be hosted on the web, DOM would require something similar to allow for shareability and accessibility. This has not been done before, but may be experimented with in the future. Furthermore, because the underlying system consists of requests being sent between R and the web browser, this can be relatively slower when compared to plots that are driven fully by JavaScript or without a client-server framework.

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

a simple example of naming grid objects that can be easily recovered

in comparison a ggplot that doesn't have a clear naming scheme

However, if it is based upon a defined plotting system without a consistent naming scheme like ggplot2, then there is a need for deducing which elements correspond to which part of the plot.

Another limitation is the number of interactions that can be attached. So far, the examples expressed in Chapter 4.3 require a single element to be controlled. We can attach many interactions and events to a single element at a time, but not many elements to many different interactions at once. There is a need for a more flexible system when dealing with multiple interactions for achieving more complex interactions. Furthermore, only one kind of interaction can be expressed for a single event. This means that the function created by the user must be fully defined in a single function rather than multiple functions. For example, if a hover requires both adding a tooltip and to turn the element red, then this would need to be written as a single function as we can only attach one to each event.

Code must also be written in a certain order to work. Plots in R must be drawn to a graphics device before being sent to the browser, while a new web page must be set up before we can start adding elements and interactions to the page. In cases of dealing with multiple plots, one of the disadvantages is that we lose information about the previous plot in R. This means that the user is required to identify what kind of information they need to extract before they move onto the next plot. This is demonstrated in the example of linking 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 that can be retrievable by the user if needed.

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

- requires that units that are converted to 'native' via grid should represent the data. (for ggplot2, this doesn't hold and requires a different conversion scale. In cases like this, there should be an alternative based upon where it gets data from: use `ggplot_build()`)
- assumes no missing values and that plots generated via gridSVG should be in the order of the data frame. (ie point order should match with indices of the df.) In cases where data taken in is rearranged and sorted (like iNZightPlots), this causes the 'indexes' of the points to differ to the original dataframe.
- assumes that most grid objects represent a single object in SVG (which sometimes is not the case - see iNZightPlot boxplot version)
- DOM and shiny have similar frameworks that establish a connection between R and the browser - more complex and requires a setup cost (either remotely or locally)
- This setup cost also comes with a speed cost as well - as every request made must be communicated back and forth between R and the browser, thus slower standalone

solutions that operate on JavaScript

- On the other hand, the upside to using these frameworks is that you keep data on the server side (when things get huge, it can become slow) - crosstalk + plotly is really slow? (no idea why)
- NAMING IS AN ISSUE IN GRID -> GRIDSVG.

5.2 Comparison to existing tools

In most of the existing tools discussed in Chapter 2, they all rely on a JavaScript library to render their plots. These are not R plots. interactr's main selling point is the ability to replicate plots or objects drawn in R and to easily achieve on-plot and off-plot interactivity. shiny can do this but you cannot easily attach specific interactions as the plot is rendered as a png. Furthermore, many of these existing tools rely on the shiny framework. One of the major disadvantages that shiny possesses is a tendency to recompute and redraw entire plots whenever an input changes. In `interactr`, only part of the plot that the user specifically targets is modified, and more on-plot interactions can be achieved. It provides a possible way of linking different types of plots together, whereas existing tools, specifically crosstalk, have focused on linked brushing between 'row-as-observations' data. To put in perspective, the simple example of linking box plots to other kinds of plots together (Figure 4.2) is an interaction that is difficult to achieve without expert knowledge of their respective APIs.

In comparison to ggvis, the interactr package may not be able to win against its simpler API. ggvis is one of the few packages that uses the shiny infrastructure but does not redraw when ever an input changes.

control density bandwidth using ggvis

control density bandwidth using interactr

control density bandwidth using shiny

- could mention that in shiny, whenever they redraw, axes will automatically change. In interactr axes don't change, so it's a 'fair' comparison (however, if it requires changing, then we've got a limitation)

Because many of these existing tools are still being developed, it is likely that they may resolve these limitations in the future. However, these tools still require the user to be very familiar with its API. An example of this is the plotly package that has expanded further into achieving linking between other types of plots without the use of crosstalk or shiny and the ability to prevent redrawing when used with shiny. It requires the user to know both the plotly and shiny API, along with some D3 knowledge.

5.3 Future directions and further problems

The problem of handling large data sets is still present because 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.

There is potential in developing interactr further to try achieve complex interactions that are more useful in exploratory data analysis. Currently, only a small number of examples have been successful. There is also a need for making it more simpler and versatile for users without compromising the flexibility in which the user can define interactions. Possible ideas would include integrating plots with **D3** and other **htmlwidgets** to achieve special effects such as zooming and panning of a plot and to achieve multi-directional linking.

Chapter 6

Conclusion

There is a need in expanding web interactive graphics to create better data visualisations for users. Despite having many tools available including plotly, ggvis and shiny, 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. 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 be used in the future.

6.1 Additional resources

The **interactr** package is currently hosted on Github here.

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

```
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.

References

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.

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

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

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