Interactive and Static Statistical Graphics: Bridge to Integration

by

Zehao Xu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Statistics

Waterloo, Ontario, Canada, 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Paul Ross Murrell Associate Professor, Dept. of Statistics, University of Auckland
Supervisor:	Richmond Wayne Oldford Professor, Dept. of Statistics and Actuarial Science, University of Waterloo
Internal Member:	Greg Bennett Professor Emeritus, Dept. of Statistics and Actuarial Science, University of Waterloo Ryan Browne Assistant Professor, Dept. of Statistics and Actuarial Science, University of Waterloo Marius Hofert Associate Professor, Dept. of Statistics and Actuarial Science, University of Waterloo
Internal-External Member:	Jian Zhao Assistant Professor, Dept. of Cheriton School of Computer Science, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

The research in this thesis is based upon the design and implementation of several R software packages novel to this thesis. The base package is `loon` (Waddell and Oldford, 2020) based on the thesis by Waddell (2016) supervised by Wayne Oldford.

I am the principal author of all new packages (viz., `loon.ggplot`, `ggmulti`, `loon.shiny`, and `loon.tourr`) and the research in all chapters. Software design and implementation benefited from review by Wayne Oldford, as did the writing of all chapters.

Exceptions to sole authorship of material are as follows:

Research presented in Chapter 1:

The software bridge abstraction is based on an idea of Wayne Oldford and developed jointly in this chapter.

Research presented in Chapter 2:

The `l_compound` data structure already existed in `loon` but was extended in the current work.

Research presented in Chapter 3:

I was the main author of the research in Chapter 3 under the supervision of Wayne Oldford.

Research presented in Chapter 4:

The `loonGrob` software bridge had already been initiated in `loon`; in the thesis, I re-designed and fully developed this bridge under the supervision of Wayne Oldford.

Abstract

There are plenty of graphical packages in R which play an important role in building graphics for data analysis, either static graphics (e.g., graphics, grid, ggplot2) or interactive graphics (e.g., loon, shiny). Each of them has certain strengths and weaknesses. Typically, analysts only use one graphical system at a time during data analysis. However, it may not be sufficient in some circumstances. To better achieve goals, analysts sometimes need more than one graphical packages. For example, an analyst aims to use interactive plots to uncover patterns of interest in data exploration, in which case, a web-based app or an animation could better deliver the analysis dynamically in the presentation. Unfortunately, due to the dissimilarity of the design, data analysis using multiple graphical systems could be too complicated to accomplish. To simplify the process, the idea of "bridge" is introduced. A bridge is a peer to peer transformation and works as a connection to map elements (i.e., visual display or visual structure) from one graphical system to another. Usually, the difficulty level of building a bridge mainly depends on how well the abstraction level can be matched.

In this thesis, we mainly focus on four packages. The graphical system loon provides interactive visualization toolkit for data exploration. The package ggplot2 offers tools to extend the flexibility of drawing static plots in data analysis based upon a grammar of graphics. The package grid is a core graphical system in R, providing low-level, general purpose graphics functions. The package shiny provides interactive web applications in R.

To integrate the strengths of each, three bridges are introduced: bridge loon.ggplot is to transform a loon widget to a ggplot object, or backwards; bridge loonGrob is to turn a loon widget to a static grid graphic; bridge loon.shiny is to render a loon widget into a shiny web app. In addition, a new package loon.tourr is also discussed. Even though it is not a bridge, it could be useful to help find interesting lower projections from a high dimensional subspace in an interactive way.

Acknowledgements

The very first I want to thank is my parents, Xiaodong Xu and Hongbo Jiang. Words simply cannot tell how much I love them. Each time when I was upset, they are the first to give me hugs; when I had any accomplishments, they could be happier than me. They give me the most generous supports so that I can pursue my dreams to a large extent. One of the toughest time I had experienced was the College Entrance Examination. Thanks for their understanding and encouragements so that I could have a chance to receive a good education, from Southwestern University of Finance and Economics to University of Waterloo. I feel so lucky to be their kid. Also, I would like to thank my grandparents, Li Xu and Cuilan Sheng. They give me a wonderful childhood. I hope I make them proud.

I want to give my sincere thanks to my supervisor Wayne Oldford. The first course I took in University of Waterloo was STAT431 (Generalized Linear Regression) in 2015 fall and Wayne was the lecturer. It was an amazing experience. His knowledge, humor and attitude on academic left me with a deep impression. I first felt that statistics could be so interesting. In 2016 winter, I took his another course, STAT444 (Statistical Learning - Function Estimation). In this course, I first "met" the interactive graphical system `loon` (which I would dedicate with my whole PhD career) and immediately was obsessed by its powerful functionalities. More than interests, I found the direction of my career. In 2016 fall, I became his master student. The master career was a hard time for me. Wayne had very high expectations on me but at the beginning, I was not so capable to accomplish his requirements. Nevertheless, as time went by, I realized I was better and better in research. Wayne pushed me hard and gave me a chance to see "a better me"! After the master, I decided to pursue a PhD degree with him. During my PhD career, the weekly meeting was becoming a joyful moment. Our talk could always innovate many good ideas. All of a sudden, it is my time to say goodbye. I am so grateful for his time, patience and guidance among these six years. It is the greatest treasure in my life. Also, I want to express my thanks to Dr.Bennett who has spent so many hours to help me polish my dissertation. Many thanks!

Next I want to thank my friends, Tianbo Wang, Xinyu Zhang and Siyu Hao. Tianbo, Xinyu and I have known each other since year 2015. They gave me many helps. Tianbo is a selfless guy who is always willing to help people without considering gain or loss. Xinyu is an optimist, always confident about the future. It is a ten years' friendship between Siyu Hao and I. We first met in my fourth year high school. He is definitely one of the smartest and most knowledgeable person I have ever known. Besides, I would like to express my thanks to Dr.Yijun Xie, "Wayne Gang bad boys" (Pavel Shuldiner and Chris Salahub), Jiayue Zhang, Chenghao Liao, Hao Luo. It is great to have you to make my Canada life colourful. Special thanks to Mary Lou Dufton (Administrative Coordinator Graduate Studies of our department) who helped me with all my administrative work in the past 4 years.

In the end, I would like to thank my girl friend, Yuanyue Yang who helped me a lot in writing. It is so great to have her company. Also, I would like to thank two little cute dogs, mochi and teemo (two pomeranians) who bring me so many happy hours.

Dedication

This is dedicated to my parents, Xiaodong Xu and Hongbo Jiang.

Table of Contents

List of Figures

Chapter 1

Introduction

> "It has been widely noted that between 1940 and 1960 there had been a great decline in the attention paid by academic statisticians to graphical representation of data. Academic statisticians found the new analytical and conceptual aspects of their field more exciting. Lately, however, there has begun a substantial change in the attention paid to graphics as an intellectual discipline with important uses."
> Chernoff, Herman (1978)

Statistical graphics have often been distinguished as whether they are static, interactive, dynamic, presentation, or exploratory graphics. This chapter begins with a discussion of these distinctions, focusing mainly on static versus interactive and presentation versus exploratory graphics. A detailed example of interactive graphics is then given to show the particular value of interactive graphics in exploratory data analysis.

As all graphics in this thesis are embedded in the R environment, an overview of the base pre-installed R graphical packages (viz., `grid`, `graphics`, and `tcltk`) is given followed by a summary of many of the R graphical packages built on the base packages (e.g., `lattice`, `ggplot2`, and `loon`). Various connections between these packages are shown and the idea of a software "bridge" from one graphical package to another introduced.

Each graphical package has its own strengths and weaknesses. The idea of a bridge is to combine the strength of different graphical systems by allowing the user to transform plots from one graphical system to another. A bridge would allow the user to select whichever graphical system is best suited to a given purpose and, as the purpose changes, to transform the graphic to a graphical system better suited to the new purpose. Several examples are given to illustrate the definition of a bridge followed by a more formal description of the "bridge" as an abstract mapping of elements from one graphical system to elements of another.

The chapter closes with a brief outline of the remaining chapters of the thesis.

1.1 Interactive Graphics and Static Graphics

In terms of interactivity, a distinction can be made between static graphics and interactive graphics. Both static graphics and interactive graphics are good for exploratory data analysis (EDA) where we hope to uncover patterns, both anticipated and unanticipated, in data, but their aims are different.

> "The aim of interactive graphics is not to improve and polish a particular display till it conveys its message in an effective manner, but to use sets of displays to explore data sets and discover the information in them."
> Unwin (1999)

Static graphics can be polished so as to present the intended information to the user as clearly as possible. This is not the strength nor purpose of interactive graphics. Rather their strengths are that they are the more efficient means for the analyst to discover the patterns in data in the first place.

Static graphics are typically not changeable after rendering (e.g., after appearing in a print publication). In a statistical analysis system, functions might be provided which allow adding to the static graphic after it has been rendered. For example, in the base `R` `graphics` (R Core Team, 2013) package, the function `lines()` will add lines to the current plot already rendered. In principle, the whole plot could be constructed via a sequence of commands that add to the existing display (e.g., `axis()`, `mtext()`, etc.).

In contrast, after interactive graphics have been rendered, users are still able to take an action on the plot resulting virtually instantaneous change of elements (e.g., Becker, Cleveland, and Wilks, 1988). For example, a simple interaction in the `R` `graphics` package can be realized by the `identify(x, y)` command. Once `identify` is executed, the position of the graphics pointer can be read when the mouse button is pressed over the current plot. Hitting the escape key causes the end of the execution and the indices to be displayed on the plot. Other statistical graphical systems allow much more interactivity and more immediate feedback.

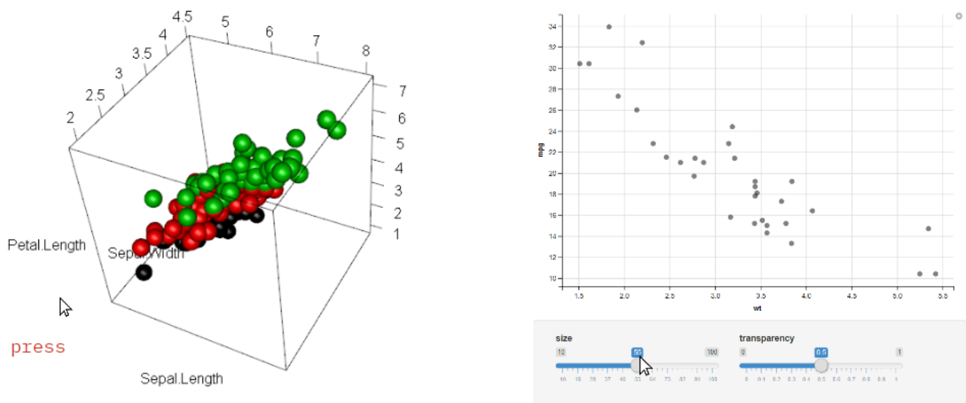
Since the earliest systems, data analysts have explored a variety of ways to directly interact with a statistical graphic. For example, in PROMENADE (Ball and Hall, 1967) (Ball and Hall, 1970), they considered using a light pen to interact with the graphic and finally chose to use a "mouse" (described in details in the paper) to select data points or move items on the screen. In PRIM-9 (Friedman, Fisherkeller, and Tukey, 1974), button switches were used to interact with the graphic. For example, the coordinate displayed was changed in the vertical (or horizontal) direction by depressing one of button switches. In ORION I (McDonald, 1982), a trackerball was used to paint on two linked scatterplots. The cursor was positioned in the active plot by moving the trackerball − points near the cursor were colored red, those at an intermediate distance were painted purple, and points far away from the cursor were blue. The same points on the other plot were given the same color. Brushing was used by Becker and Cleveland (1987), where the analyst used a mouse to construct a rectangular region and points within this region would be highlighted. In DINDE (Oldford and Peters, 1988), contextual menus popped up associated with points or with whole displays depending on where the user was focusing. Any number of these methods would be used to interact with a statistical graphic thereafter. For a more complete history, see, for example, Cook and Swayne (2007) or Wilhelm (2005).

The immediacy of the direct manipulation is associated with the physical proximity of the means to effect the change to the graphic element being changed. For example, a "tool tip" might be accessed to provide information about a point by hovering the mouse over the point being queried. This close proximity between object displayed (the point), the query or interaction (the hovering mouse), and the rendered result (the displayed tool tip), provides an immediacy which allows the user to efficiently and effortlessly change the display without changing their visual focus. In contrast, changing a plot by choosing items from a distant menu-bar or control panel is an example of a less immediate interaction. Least immediate might be a command executed in a console to effect the same change.

In this thesis, we distinguish direct manipulation from command-line manipulation (programmatic interaction) as having interactions be some combination of mouse gestures (possibly modified by a few keyboard events) conducted near the graphic to be affected. Ideally, both direct manipulation and command-line manipulation will be available to the user, for example, see Oldford (1999). The nearer the user's visual focus remains to the graphic being changed the more immediate is the direct manipulation.

It is sometimes useful to distinguish those direct manipulations which are immediate (and proximate to the visual focus) from those which are "mediated" by the user accessing a control panel, or slider, or any other intermediary display, to effect the changes (and requiring user focus to move from the graphic element)[1].

An example of an R package that provides immediate manipulation would be the package rgl (Adler and Murdoch, 2020), as shown in Figure 1.1 (a). Points and cube are rotated in the direction of the mouse movement. The whole time, the focus of the viewer remains on the plot itself.



(a) (b)

Figure 1.1: Figure (a) is created by rgl which provides immediate manipulation. Figure (b) is created by ggvis which provides mediated manipulation via the slider bars. The size and transparency of the points are changed as we drag the bar.

An example of an R package that provides "mediated" manipulation would be the package ggvis (Chang and Wickham, 2018), shown as Figure 1.1 (b). Two slider bars control the size and transparency of points respectively. To modify these two plotting states, the focus of the viewer is slightly moved away from the plot itself to the values on the sliders.

The phrase "dynamic graphics" (Cleveland and McGill, 1988) has sometimes been used to cover both statistical graphics which change dynamically in real time, whether under user control or not, and what we have called interactive graphics. A better distinction would be to use "motion graphics" or "kinematic graphics" (or even "animated graphics")

[1]"Immediate manipulation" and "mediated manipulation" are sometimes distinguished as direct and indirect manipulation, such as Swayne and Klinke (1999) and Sievert et al. (2019).

to describe graphics changing in real time as if in a motion picture. For example, the plots in Figure 1.2 can be thought of as five frames of a movie. When played in sequence, the illusion is created of points moving. Commonplace examples would include animated GIFs (e.g., Pedersen and Robinson, 2019). The phrase "dynamic graphics" will not be used in any technical sense in this thesis.

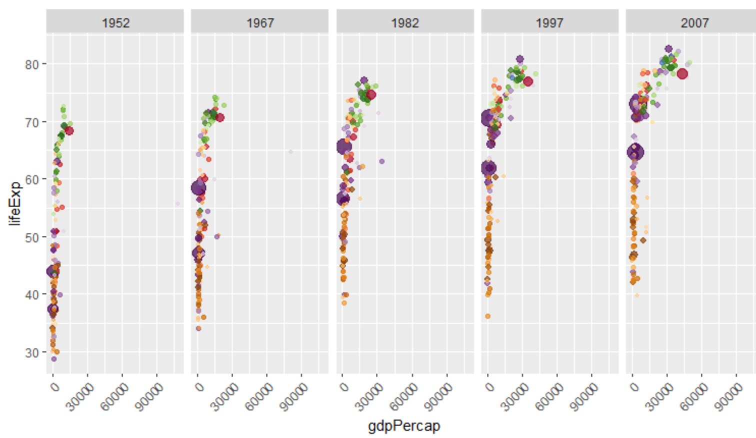


Figure 1.2: It illustrates the life expectancy versus GDP per capita from 1952 to 2007 (the data is contributed by Bryan, 2017). The point colors represent continents (yellow is Africa; red is American; purple is Asia; green is Europe and blue is Oceania) and the point sizes represent population. As they five are shown in sequence, the viewer may have an illusion that the graphic is "dynamic" and points are moving.

1.2 Exploratory Graphics and Presentation Graphics

Graphics can also be distinguished between exploratory graphics and presentation graphics by audience (e.g., see Theus and Urbanek, 2008).

Exploratory graphics are graphics created in data exploration. Data analysts can create whatever they want to best discover the particular patterns of interest. They are used necessarily before presentation graphics.

Presentation graphics are graphics used to present. These plots usually focus on one or more data sets particular to the analysis to present the result. No changes, or very limited changes, can be made on these plots.

In Figure 1.3 'A', data analysts explore a data set and choose some graphical tools to convey the statistical summaries. This process is open and graphics created in this process are called the exploratory graphics. After that, data analysts may share those graphical summaries to a small group (Figure 1.3 'B'); give an on-site talk (Figure 1.3 'C'); or hold an online presentation (Figure 1.3 'D'). The graphics we mentioned in 'B', 'C' and 'D' are curated (pre-designed) and called presentation graphics.

Analysts often alternate between using graphics for exploration and graphics for presentation. Once an analyst gets feedback or is inspired by a small group or audiences, presentation graphics can also be turned back to the exploratory graphics to discover more interesting patterns. Presentation graphics are not necessarily static. Figure 1.3 'E', for example, is an interactive `shiny` (Chang et al., 2019) app (Salahub and Oldford, 2018). The graphics provide some interactions in this presentation. However, the interactions are limited.

1.3 An Example of Interactive Graphics in EDA

In an exploratory data analysis, we often begin without a complete description of how the analysis will unfold. Instead, we view different graphics and, depending on what we see, we may proceed in one direction or another. It is an iterative process where each step depends upon what was learned before. Since 1960s, Tukey pioneered and promoted graphical tools as a central tool for exploratory data analysis (e.g., Tukey and Wilk, 1966; Friedman, Fisherkeller, and Tukey, 1974 and Tukey, 1977). Compared with static graphics, interactive graphs allow us to make these steps very efficiently through direct manipulation.

When there are multiple logical conditions to be enforced on the display, static graphics are much slower to reflect results and require relatively more complex means to effect the changes across multiple displays; than do interactive graphics. Additionally, static graphics are often displayed on a very limited space (e.g., RStudio), making it hard to compare many plots simultaneously. In contrast, with interactive graphics, changes are typically reflected across multiple plots immediately, making it relatively easy to compare multiple plots at the same time.

This efficiency suggests interactive graphics are well suited to data exploration. To illustrate, we will investigate how the ratio of coronary heart disease (the number of people with heart disease to those without heart disease) changes with family history of such disease and as systolic blood pressure change, using the interactive graphical package `loon` (Waddell and Oldford, 2020). The data is from the package `loon.data` (Oldford and Waddell, 2020).

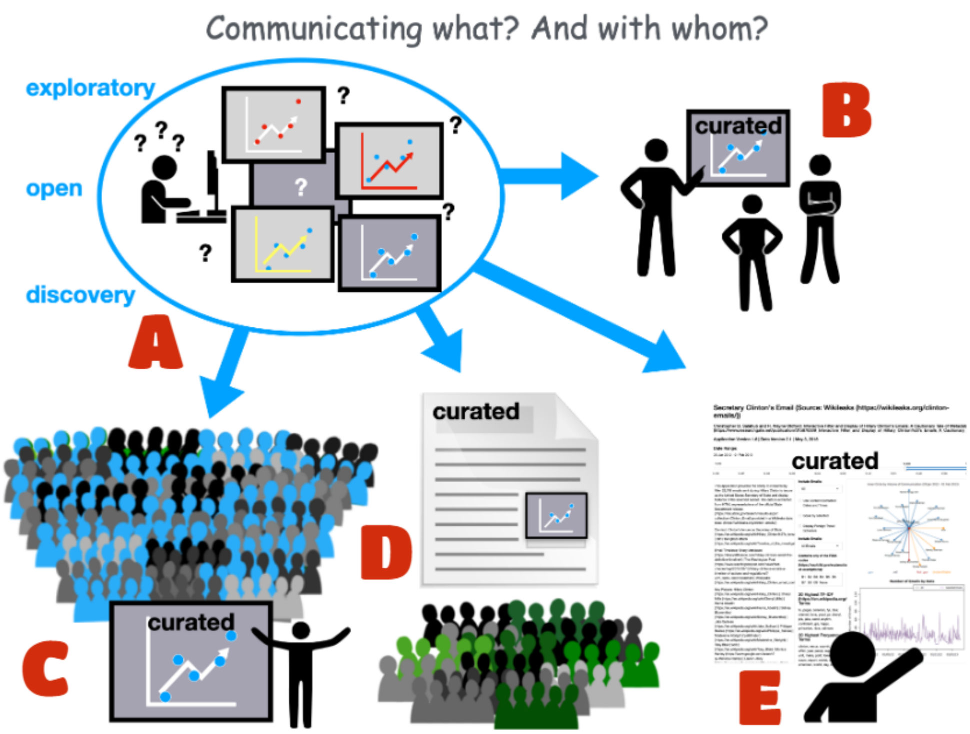


Figure 1.3: Graphics created in ‘A’ are exploratory graphics. The graphics presented to a small group (‘B’), a talk (‘C’) or an online session (‘D’) are presentation graphics. Presentation graphics can be interactive as well, but they are curated and the modifications are very limited (e.g., plot ‘E’). See Oldford (2019).

"Hastie and Tibshirani (1987) selected a subset of 465 subjects from the 3,357 white males (in these communities, male mortality rates were about two and a half times that of the females; see Rossouw et al., 1983). The 465 subjects consisted of all 162 cases having had coronary heart disease as well as 303 controls sampled from the remaining set of survey subjects.

The same (or similar) data seems to be used again for illustration in Hastie, Tibshirani, and Friedman (2009) and it is that which is now ported here from the book's accompanying website. Curiously, this data set (viz., that recorded here) contains values on only 462 subjects, of which now only 160 are cases and 302 are controls."
Oldford (2020b)

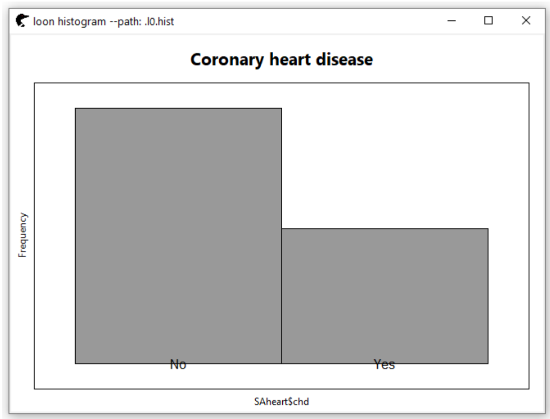
Consider the following three variables as described by Oldford (2020b):

- "sbp: Systolic blood pressure in millimetres of mercury (mm Hg)."
- "famhist: Factor indicating presence or absence of a family history of ischaemic heart disease."
- "chd: The response, a factor identifying whether the subject had been diagnosed as having coronary heart disease or not."

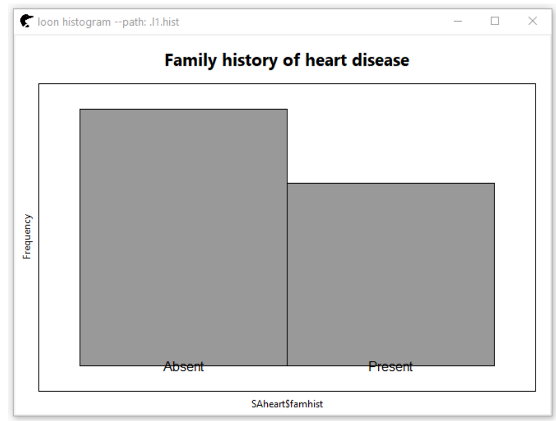
From bar plot 1.4 (a), we observe that the ratio of the coronary heart disease is approximately 2:1; from Figure 1.4 (b), we observe that the proportion of people without a family history of heart disease is higher than those with.

We now interactively explore how the ratio changes depending upon whether people have a family of heart disease or not. Shown as Figure 1.5 (b), when users click on the bar corresponding to people with family history on plot 1.4 (b), the bar is highlighted (the color turns to magenta). Since the two barplots are linked, changes on one plot will produce changes on the other, shown as Figure 1.5 (a). The ratio of coronary heart disease, shown by the ratio of the highlighted bars in Figure 1.5 (a), is approximately to 1:1. Alternatively, these changes can be effected programmatically, though this requires typing texts in the command-line and so not be as immediate.

High blood pressure may be another risk factor for heart disease. Suppose we are interested in the ratio for those people who either have high blood pressure ("sbp" $>$ 175 mmHg) or have a family history of heart disease. To see this ratio, we first need a histogram of the "sbp" (as in Figure 1.6 c) that is linked to these two bar plots.



(a)



(b)

Figure 1.4: They are two linked bar plots. Figure (a) is the bar plot representing heart disease ('No' and 'Yes'). Figure (b) is the bar plot representing a family history ('Absent' and 'Present') of heart disease.

To effect this query, we first click on those who have a family history of heart disease as in Figure 1.5 (b). Then, we need to select the high blood pressure from the "sbp" histogram as in Figure 1.6 (c). But now with the <`shift`> key pressed at the same time, so as to preserve the highlighted bins. Notice this has picked up some people who have no history of heart disease as seen in Figure 1.6 (b). All three plots reflect these two selections. Now the ratio can be determined by the highlighted bins in Figure 1.6 (a). While these heights have been changed (c.f., Figure 1.5 a), the ratio remains the same as 1:1. Many such logical queries can be made interactively (e.g., see Oldford, 2020a).

All these graphics are interactive exploratory graphics. They are open and flexible. After exploration, some of these graphics are typically turned into presentation graphics. The quality of a screenshot of these graphics is not particularly good. For example, the quality of the display can be low when the screenshot is simply saved as an image. So, a question may be raised, "how can high quality static graphics for storage or publishing be produced from an interactive plot?" We will answer this question in succeeding chapters.

(a) (b)

Figure 1.5: As these two plots are linked, highlight the barplot (b) will cause changes in the barplot (a). When a family history of heart disease presents, the ratio of people gets heart disease is approximately 1:1.

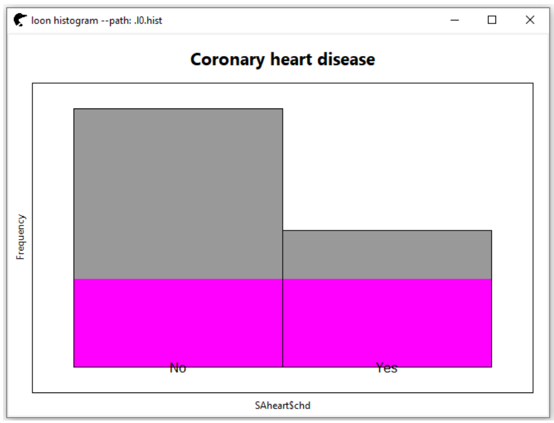
1.4 Graphical Packages in R

R (Ihaka and Gentleman, 1996) (R Core Team, 2013) is one of the most widely used software in statistical analysis. It provides a free software environment, a large number of statistical and computing methods, and several elegant graphical packages. Researchers work both independently and collaboratively to contribute to the R project. As of May 2021, over 17,000 packages have been created on CRAN which cover almost all aspects of statistics.

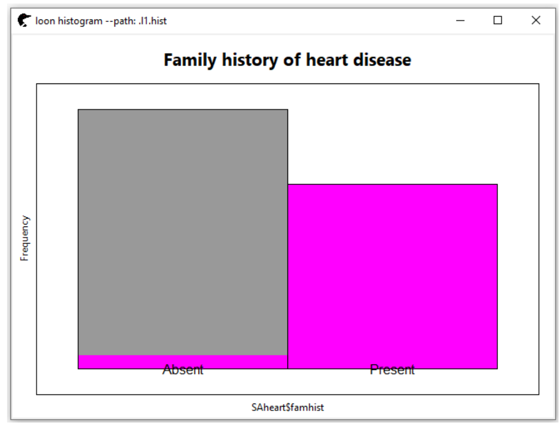
In this thesis, all graphics discussed belong to the R system. We begin with an overview of visualization systems available in R.

1.4.1 Base Graphical Packages

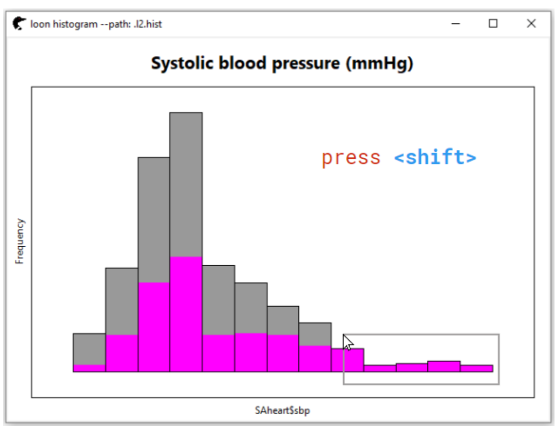
There are over fifty graphical packages in R, among which are the base packages `graphics`, `grid` (Murrell, 2002), and `tcltk`. They are installed automatically. Nearly all graphical packages are based on one or more of these three packages. For example,



(a)



(b)



(c)

Figure 1.6: Conditional on a history of family heart disease (the right bin in Figure b) or high ($>$ 175mg) systolic blood pressure (the rightmost bins in Figure c), the ratio of people getting heart disease is approximately 1:1 (i.e., the ratio of the highlighted heights in Figure a).

> "The `graphics` package, which will be referred to as the traditional graphics system, provides a complete set of functions for creating a wide variety of plots plus functions for customizing those plots in very fine detail.
>
> The `grid` package provides a separate set of basic graphics tools. It does not provide functions for drawing complete plots, so it is not often used directly to produce statistical plots. It is more common to use functions from one of the graphics packages that are built on top of `grid`, especially either the `lattice` (Sarkar, 2008) package or the `ggplot2` (Wickham, 2016) package."
> Murrell (2018)

The `tcltk` package provides access to the platform-independent "Tool Command Language", `Tcl`, and its toolkit, `Tk`, which were developed by John Ousterhout in the late 1980s. `Tcl` is a scripting language and `Tk` provides a number of widgets commonly used to develop interactive applications. The `tcltk` package in R combines both and has everything needed to provide interactive graphics for data exploration. Some comments and strengths of these three core packages are shown in Table 1.1.

1.4.2 Extensions of the Base Graphical Packages

Most graphical packages rely on one or more of the three packages, as shown in Figure 1.7. There, packages with dark gray backgrounds are pre-installed graphical systems and most are maintained by the R core team. Packages in light gray are extensions built upon one or more of the three packages discussed above. Package dependency is represented by the direction of the arrow. For example, the package `ggplot2` is built on top of, and so depends on, the package `grid`.

Some graphical packages that provide direct manipulation are built upon `tcltk`. For example, the package `tkrplot` (Tierney, 2021) provides a simple mechanism to place R graphics in a `Tk` widget. In contrast, the package `loon` is a large and extendable toolkit for data exploration.

The packages `diagram` (Soetaert, 2020), `maps` (Becker et al., 2018), `tourr` (Wickham et al., 2011), `pixmap` (Bivand et al., 2021) and `gplots` (Warnes et al., 2020) are built based on the base graphical package `graphics`. The package `diagram` plots small networks, flow charts, and webs; `maps` draws geographical maps; `tourr` provides a variety of projection methods for kinematically exploring high dimensional data (e.g., random tour, guided tour, etc.); `pixmap` provides functions for import, export, plotting and other manipulations of

Table 1.1: Base Graphical Packages in R

packages	comments	strengths
`graphics`	automatically loaded in a standard installation of R; traditional base graphics	high rendering speed; simple commands; good for prototyping new graphics; easily customized
`grid`	automatically loaded in a standard installation of R; it only provides low-level, general purpose graphics functions	classic computer graphical abstractions (viewports, coordinate systems, clipping, etc.), flexible and open-ended, excellent for prototyping (especially complex designs), arbitrary layout
`tcltk`	automatically loaded in a standard installation of R; interface to third party `tcltk` library	GUI (Graphical User Interface) toolkits providing powerful mouse gesture abilities are conveniently available in R. Graphics can be kinematic and interactive.

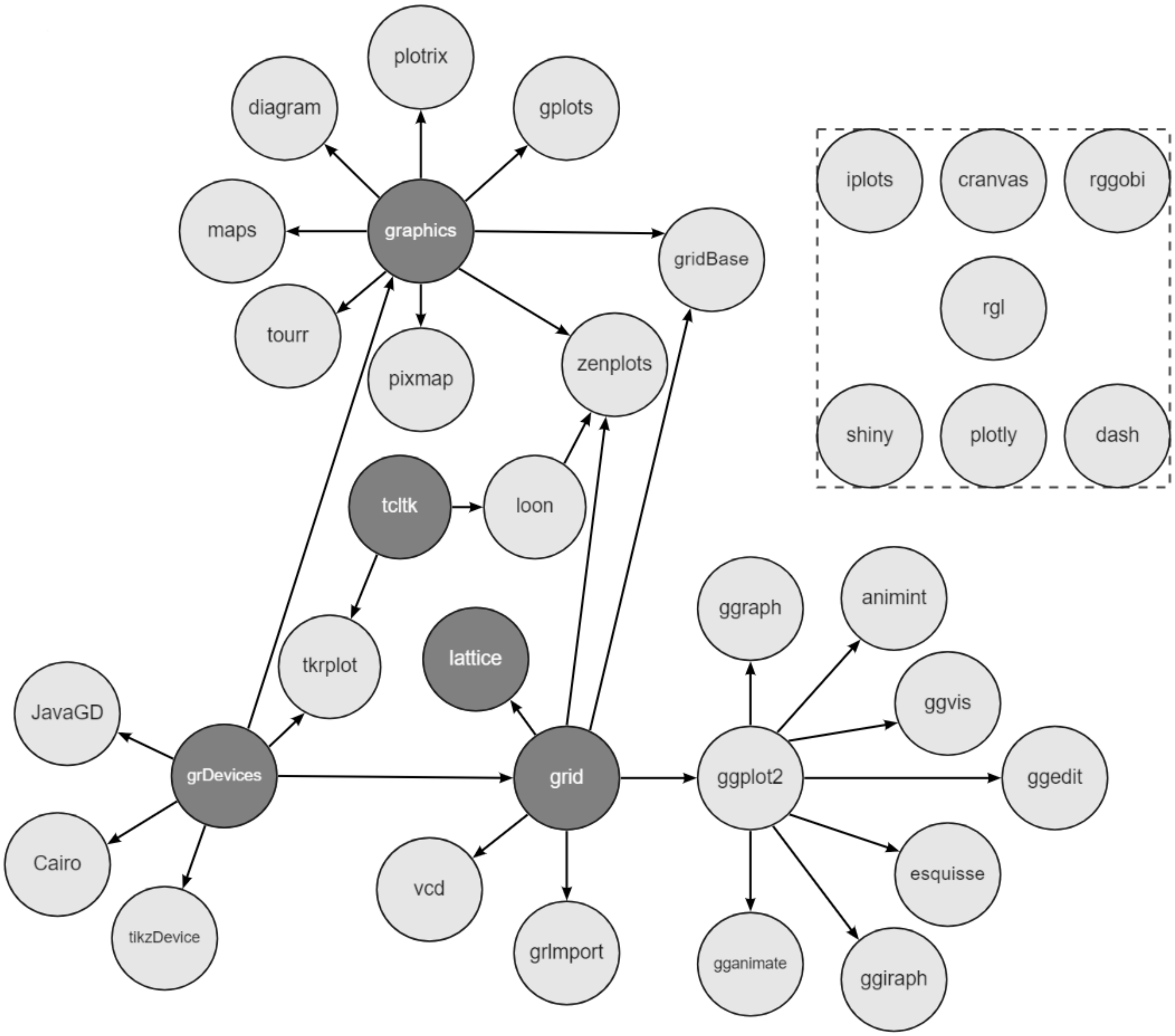


Figure 1.7: It only provides a small collection of R graphical packages and is extended from the figure in (Murrell, 2018, p. 19) (as shown in Figure A.1, appendix)

images; gplots provides a variety of enhanced statistical plots (e.g., balloonplot, barplot2 and hist2d).

Other graphical packages are based on grid graphics. For example, the package vcd (Meyer et al., 2020) provides tools, such as mosaic plots and sieve displays (Meyer et al., 2006) to visualize categorical data; grImport enables users to convert, import, and draw postscript pictures in R plots; lattice (Sarkar, 2008), a pre-installed R graphical system, implements the trellis graphics for multivariate data (Becker et al., 1996).

The package ggplot2, built on top of grid, implements a grammar of graphics (Wilkinson, 2005), providing tools to extend the flexibility of drawing static plots in data analysis. Over 100 extensions of ggplot2 have been contributed by many researchers. For example, the package ggraph (Pedersen, 2021) extends ggplot2 to draw graph and networks; gganimate (Pedersen and Robinson, 2019) extends ggplot2 to provide kinematic graphics. The packages animint2 (Hocking et al., 2020), ggvis (Chang and Wickham, 2018), ggedit (Sidi, 2020), esquisse (Meyer and Perrier, 2020) and ggiraph (Gohel and Skintzos, 2019), each provide a browser-based interactive graphics from ggplot2. More details of these interactive graphical systems will be given in Section 3.3.

Some packages depend on more than one of these three graphical systems. For example, gridBase (Murrell, 2014) is an integration of base graphics and grid graphics. The package zenplots (Hofert and Oldford, 2019), visualizing high dimensional data via spatial layout in a zig-zag pattern, implements its plot in the user's choice of grid, graphics and loon.

Some graphical systems provide additional graphics devices for R, such as JavaGD (Urbanek, 2020) which routes all R graphics commands to a Java program, tikzDevice (Sharpsteen and Bracken, 2020) which records plots in LaTeX, and Cairo (Urbanek and Horner, 2020) which uses Cairo graphics library for creating high-quality vector (PDF, PostScript, SVG) or bitmap output (PNG, JPEG, TIFF) as well as high-quality rendering in screen displays (X11, Win32).

There are other graphical packages entirely separated from the pre-installed ones as shown within the rectangular region of Figure 1.7. These provide interfaces between R and third-party graphics. For example, iplots (Urbanek and Theus, 2003), (Urbanek and Wichtrey, 2018), providing fluid and responsible interactive graphics, even with a million points, is based on Java; cranvas (Xie et al., 2013), constituted of graphics layers, is based on Qt; rggobi (Wickham et al., 2006), R implementation of GGobi (Cook and Swayne, 2007), is based on Gtk; rgl (Adler and Murdoch, 2020), providing interactive sophisticated 3D images, is based on OpenGL; the packages shiny (Chang et al., 2019), plotly (Sievert, 2018), and dash (Parmer et al., 2020), each create interactive web graphics, by wrapping

Javascript, HTML, and CSS in R functions.

1.4.3 Transformations between Different Graphical Packages

From exploration to presentation, users typically use a single graphical system. There are many available graphical systems in R and each has its own strengths and weaknesses. To best achieve a user's goals, more than one graphical systems may be needed. For example, one may use interactive plots (e.g., loon) in data analysis to explore the patterns of interest. After exploration, a web-based app (e.g., shiny or plotly) can be used to present the final analysis dynamically. However, due to the different design of each graphical system, it is often difficult for a user to switch from one to another.

Figure 1.8 adds dotted lines which can be imagined as a transformation from one graphical package to another. For example, the dotted arrow now between grid and ggplot2 is there, because ggplot2 has its own graphics objects which are displayed by transforming them into grid objects. The same is true for the dotted arrow between lattice and grid. Note that there is no dotted arrow between vcd and grid, because vcd does not have graphical data structures that are transformed to grid objects. One can imagine a transformation between a lattice object and a ggplot object; then users can explore and visualize in both graphical systems at the same time. Such a transformation will be from one static graphical package to another.

Another interesting possible connection could be between loon and iplots. Both are interactive graphics. The strength of loon is to provide richer direct manipulation tools, such as a floating palette interface − loon inspector which controls the plot views and modifications of graphical elements. The strength of iplots is the ability to interactively visualize a very large number of observations. If the two graphical systems were connected, loon users who wanted to visualize a million points would turn their loon plots into iplots plots; iplots users who favoured rich interactive toolkit like loon could do the reverse. This transformation would be from one interactive graphical package to another.

The package plotly already provides a link which can transform a ggplot object to a plotly object, which is a Javascript based graphical system. Data structures in ggplot2 are simply transformed to corresponding data structures in plotly. This transformation is from one static graphical package to an interactive graphical package.

This thesis will focus on transformations between interactive loon plots and plots from other packages. For example, loon.ggplot (Xu and Oldford, 2019a) transforms an interactive loon plot to a ggplot object, back and forth; loonGrob turns an interactive loon plot into a static grid graphic object; loon.shiny (Xu and Oldford, 2019b) renders a loon plot in a shiny app.

We will call each dotted line connection, shown in Figure 1.8, a "bridge" from one graphical system to another.

1.5 Bridges

The majority of statistical analysis tools only provide single graphical system for visualization. Developers who do not want to constrain users to one specific graphical system might provide a variety of choices in visualization. For example, zenplots (Hofert and Oldford, 2020) visualizes high dimensional data by laying out alternating one and two dimensional plots, as shown in Figure 1.9, and allows a user to draw a plot in their choice of graphical system graphics, grid or loon. The following code shows the construction of a zenplot on the iris data set (Anderson, 1935) (Fisher, 1936), alternating 2D scatterplots and 1D histograms, rendered interactive by the package loon.

```
> library(zenplots)
> # The R package used for plotting, can be one of
> # "grid", "graphics" and "loon"
> p <- zenplot(iris[, -5],
+               plot1d  = "hist",
+               plot2d  = "points",
+               pkg = "loon")
```

To accommodate the different graphical systems, the zenplot developers created functions with prefix hist_1d_* and points_2d_* and suffix * given by the value of the pkg argument (e.g., pkg = ''loon'' in this case).

The package zenplots provides eleven 1D and seven 2D geometry objects. In other words, Hofert and Oldford (2020) had to create $(11 + 7) \times 3 = 54$ functions to realize this accommodation! Due to the dissimilar software models of packages graphics, grid and loon, it requires a lot of work.

To combine the strengths of different graphical systems, the idea of a "bridge" is introduced. Given a display in the graphical system $\mathcal{G}$, the "bridge" would return the same (or as similar as possible) display realized in a different graphical system $\mathcal{K}$. With "bridges", users who favour multiple graphical systems could transform plots designed for exploration into plots designed for presentation; developers who want to provide a flexible environment

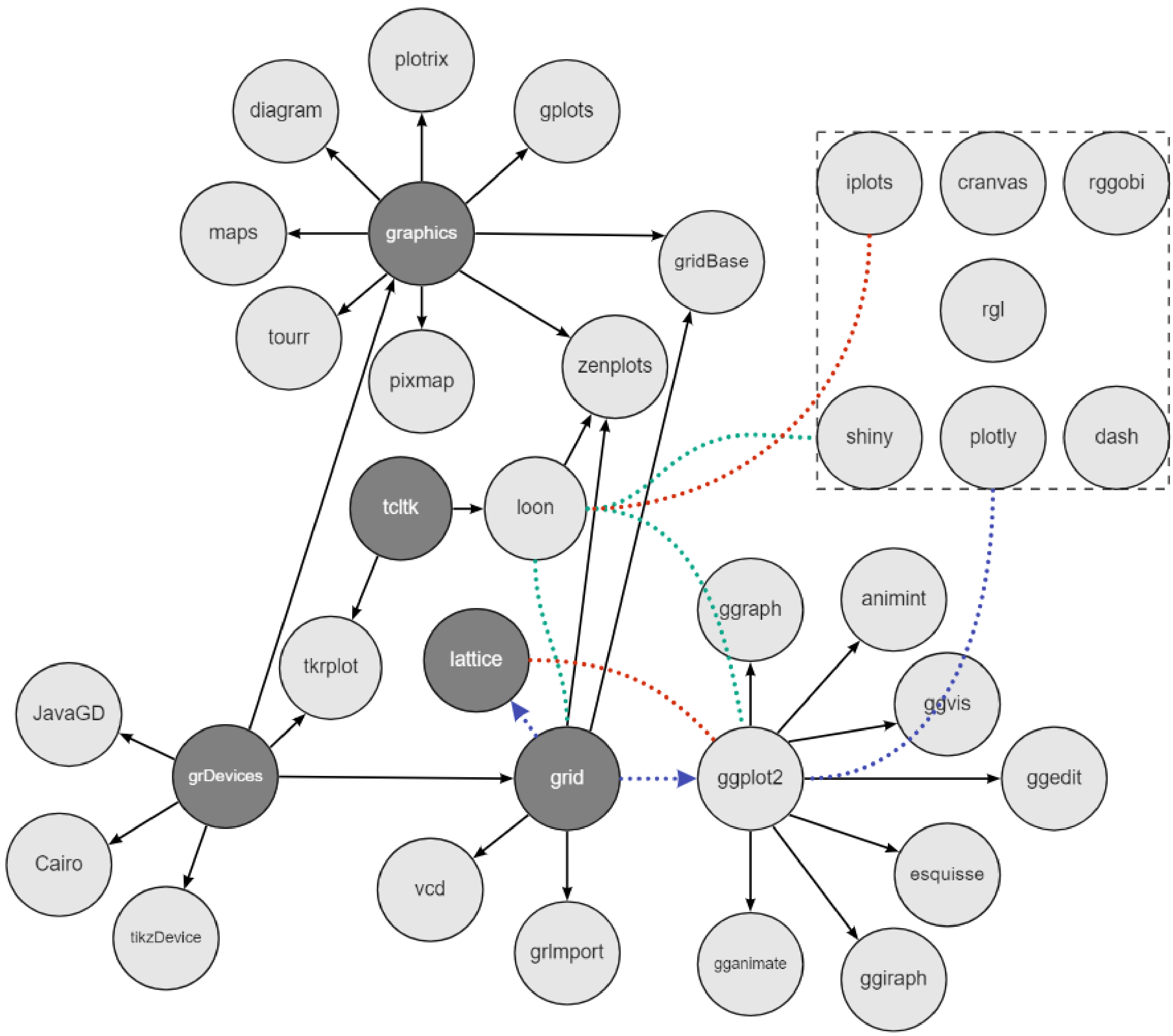


Figure 1.8: The dotted lines can be imagined as transformations. Lines are painted in three different colors: transformations that are not present are colored red (viz., `loon` and `iplots`, `ggplot2` and `lattice`); purple means that such transformations have already existed before this thesis; and the green ones represent the transformations to be introduced in this thesis. A dotted arrow means that `ggplot2` (or `lattice`) depends on `grid` but also renders by transforming the `ggplot` (or `lattice`) object into a `grid` object.

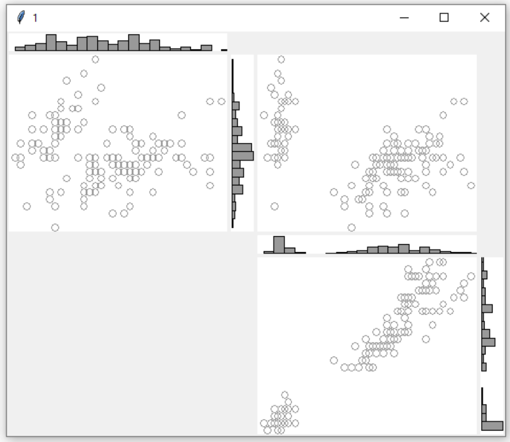


Figure 1.9: A `zenplot` of the data set `iris`.

can focus on their design on one graphical system and use the "bridge" to transform the plots to other graphical systems. For example, imagine `zenplot` only provided `loon` graphics. Rather than re-programming the same functionality in multiple graphical systems, a "bridge" would transform a `loon` plot produced to another graphical systems such as `grid` or `ggplot2`.

1.5.1 Bridge Abstraction

In this thesis, the word "bridge" is used to describe a peer to peer transformation from one graphical system to another (and possibly back). "Transformation" means that objects recognized in one side are mapped to objects recognized in the other side.

More abstractly, think of this transformation as a mapping f from a graphical system $\mathcal{G}$ with elements $g_1, \ldots, g_m$ to another graphical system $\mathcal{K}$ with elements $k_1, \ldots, k_n$. We can imagine a function $f : \mathcal{G} \rightarrow \mathcal{K}$ that transforms elements in $\mathcal{G}$ to elements in $\mathcal{K}$ and possibly another function $h : \mathcal{K} \rightarrow \mathcal{G}$ that does the reverse. The bridge is defined by the sets $\mathcal{G}$ and $\mathcal{K}$ (and their elements) together with the function(s) f and/or h. If only one of f and h exist, we say the bridge is "one-way", and say it is "two-way" if both f and h exist. Ideally $h = f^{-1}$, but this need not be the case for a bridge to exist.

To better clarify what is a bridge and what is not, some examples are now given. Note that as a bridge is more general idea and not necessarily restricted to any particular systems.

An example of something that is not a bridge is the implementation of a bi-directional link between the geographic information system `ArcView` (Breslin, 1999) and the interactive graphical system `XGobi` (Swayne et al., 1998), as presented by Symanzik et al. (2000). The two graphical systems were linked by remote procedure call; as each point/location was brushed in `XGobi`, corresponding point/location would be highlighted in `ArcView`. This linking is not a bridge because no transformations are implemented in this process.

`Quail` (Oldford, 1998) is an interactive, display-oriented programming environment for data exploration and visualization. It has a hierarchically organized object-oriented graphical system called `Views` (Hurley and Oldford, 1988) that renders plots either interactively on screen (e.g., `Mac`, `PC` or `X11`) for direct manipulation or statically in `postscript` for publication graphics, as with `zenplots`, the users chooses which. As no transformations are from elements in one graphical system to another, there is no bridge here.

Although, a `zenplot` can be rendered into a `loon` plot (or a `grid` object), but this is not a bridge. There are no transformations between objects in `zenplots` package and `loon` package (or `grid` package).

An interesting example of a bridge is to render a `ggplot` object into a `grid` object. The reason is that a `ggplot` object has its own graphical data structure even without being drawn. To display a `ggplot` object (viz., via `print()` or `plot()`, see Chapter 3); the data structures are mapped to corresponding `grid` data structures. Imagine the plotting engine is not `grid` but some other graphical systems such as `tcltk`. When displayed, elements in `ggplot2` are transformed to `tk` widgets.

In contrast, a `loon` plot is always rendered as a `tk` widget as soon as it is created; a `loon` plot must be plotted. Therefore, it is not a bridge between `loon` and `tcltk`.

A challenge of the bridge of abstraction is to determine what we mean by a graphical system $\mathcal{G}$ and its elements $g_1, \ldots, g_m$ and by a graphical system $\mathcal{K}$ and its elements $k_1, \ldots, k_n$. There are at least two choices for what might be elements in each graphical system – it might be enough to imagine each element as simply its visual display, that is simply how it appears when rendered; or, the element might be a visual structure, that is the data structure(s) which define the visualization.

For visual displays, a bridge would be successful in one direction, from $\mathcal{G}$ to $\mathcal{K}$, if $k_j = f(g_i)$ for some $k_j \in \mathcal{K}$ such that g_i and k_j appear to be (nearly) visually indistinguishable to the viewer. Alternatively, for visual structures g_i in $\mathcal{G}$ to be successful, $k_j = f(g_i)$ would have to contain all the data information of g_i as a properly constructed visual structure of k_j in $\mathcal{K}$. Of course, if the visual structure of g_i is mapped to the visual structure of k_j, then the visual display must be also mapped. However, the opposite is not always true.

An example would be to map a histogram. Imagine a g_i is a histogram visual structure, containing all essential features, such as data, bin-width and bin-origin. Suppose in $\mathcal{K}$, k_j represents a histogram visual structure, but k_l represents a visual structure simply drawing rectangles (e.g., storing `xmin`, `xmax`, `ymin`, and `ymax` for each rectangle and no other information that it is a histogram). If g_i is mapped to k_j, then both the visual display and the visual structure are mapped; if g_i is mapped to k_l, then only the visual display is mapped.

Graphical systems can be roughly characterized according to the level of abstraction at which they operate. Some, such as `grid`, have functionality primarily at a relatively low level of graphical abstraction, such as drawing (e.g., lines, circles, polygons, etc.) and layout (e.g., graphical parameters, viewports, etc.) functionality common to nearly all graphics. Slightly higher are functions more tailored to statistical graphics, such as data determined coordinate systems or even a points function (e.g., to lay out a scatterplot). Slightly higher again, levels of abstraction are enabled by functions, like `gTree`, that allow composition of the elements of a graphic. Other statistical graphical systems, such as `loon` and `ggplot2`, offer primarily functions at a high level abstraction to users, such as histogram, pie chart, or scatterplot functions, to create complete plots.

When peers $\mathcal{G}$ and $\mathcal{K}$ have matching abstraction levels, then a bridge is relatively easy to build, high-level elements to high-level elements and low-level elements to low-level elements. For example, in the `loon.ggplot2` bridge, high-level elements (e.g., histogram, scatterplot), are matched and low-level abstraction elements (e.g., lines, polygons) are matched. When peers $\mathcal{G}$ and $\mathcal{K}$ do not match abstraction levels, typically, two solutions are considered: 1. extend non-matched elements in $\mathcal{K}$; 2. break high-level elements down to several low-level elements (e.g., histogram to rectangles).

In all cases, we would like the visual displays to be as similar as possible, if not mapping exactly. Ideally, the two would also match on level of abstraction.

1.6 Thesis Overview

Bridges between four key graphical packages will be the main focus of this thesis. The package `loon` provides both direct and command-line manipulation allowing users to change elements of a plot by a mouse and programmatically. `Ggplot2`, implementing a grammar of graphics in R, has many handy functions to produce high quality graphics for data analysis. `Grid` is a core, pre-installed graphical package, mainly providing functionality at a relatively low level abstraction. `Shiny`, providing web applications, is often used for interactive presentation graphics.

Bridges would allow users to transform from one graphical system to another. For example, using interactive graphics loon for data exploration and transforming a subset of these to an interactive shiny app or static ggplot2 or grid graphics for presentation, as shown in Figure 1.8. Furthermore, a bridge from loon to ggplot2 graphics could then be followed, by the bridge ggplotly (contributed by plotly), to transform a loon display to a plotly display in two steps.

When building a bridge from one graphical system $\mathcal{G}$ to another graphical system $\mathcal{K}$, it is important to first consider the level of abstraction of each graphical system. When the levels of abstraction match (either at a high-level or a low-level), then, it should be straightforward to match visual structures. Matching the visual displays should follow from matching the visual structures.

It may still be the case that some element g_i of $\mathcal{G}$ has no counterpart in $\mathcal{K}$. To complete the bridge from $\mathcal{G}$ to $\mathcal{K}$, $\mathcal{K}$ would have to be extended to include an element k_i to match g_i. Conversely, an element k_j in $\mathcal{K}$ might have no counterpart in $\mathcal{G}$ so that not all elements in $\mathcal{K}$ are reachable by a bridge from $\mathcal{G}$ to $\mathcal{K}$ without first extending $\mathcal{G}$. For example, in Chapter 2, the graphical system loon ($\mathcal{G}$) is extended by adding facet plots as in the graphical system ggplot2 ($\mathcal{K}$), and in Chapter 3, the graphical system ggplot2 ($\mathcal{K}$) is extended by the package ggmulti to match some of the high dimensional graphics, available in loon ($\mathcal{G}$).

When the levels of abstraction do not match, for instance, the graphical system $\mathcal{G}$ mainly provides high-level elements and the graphical system $\mathcal{K}$ mainly provides low-level elements, then most high-level elements $g_i, \ldots, g_n$ in $\mathcal{G}$ will not have exact counterparts in $\mathcal{K}$. Visual structures are not easily mapped without creating high level visual structures in $\mathcal{K}$, effectively changing $\mathcal{K}$ from its low level design. Alternatively, we could treat elements $g_1, \ldots, g_n$ in $\mathcal{G}$ as visual displays and reproduce them in $\mathcal{K}$ using only the low level abstractions of $\mathcal{K}$. For example, in Chapter 4, a histogram (g_i) in loon ($\mathcal{G}$) is expressed in grid ($\mathcal{K}$) as a number of stacked rectangles (k_i) using the low level grid abstraction rectGrob because no histogram abstraction exists in grid.

In this thesis, three bridges are introduced and discussed, loon.ggplot (a two-way bridge), loonGrob (a one-way bridge) and loon.shiny (a one-way bridge). Each of these bridges encounters one or more of above issues in their construction. The issues and how they were resolved, as well as shortcomings of the solutions, will be discussed.

The package loon is reviewed in Chapter 2 and new functionality, such as l_facet, is introduced in Section 2.4 to extend loon. This extension is needed to accommodate a bridge between loon and ggplot2. Similarly, the graphical system ggplot2 is extended in Chapter 3 by functionality in a new auxiliary package ggmulti (Xu and Oldford, 2020).

This extension adds more elements (e.g., non-primitive glyphs and serialaxes plots) for `ggplot2`.

Chapter 3 describes in detail a two way bridge between `loon` and `ggplot2`. As `loon` and `ggplot2` generally match levels of abstraction, the bridge maps visual structures of one graphical system to those of the other. For example, `geom_point` structures in `ggplot2` are mapped to interactive scatterplots (or point layers) in `loon` and `geom_histogram` structures to interactive histograms (or rectangular layers). In this way, for most cases, the visual structures are mapped from one system to the other.

However, for some elements, there are no counterparts in `loon` to match the elements in `ggplot2` (e.g., a `ggplot` pie chart cannot be mapped to an interactive `loon` pie chart). In this case, either the visual display (e.g., see Figure 3.10) is mapped, or no mapping at all is provided (e.g., see Figure 3.11). As not all visual structures are perfectly mapped from `ggplot2` to `loon`, the bridge in this direction could be improved only by a substantial extension to `loon` that is not considered in this thesis (see Section 7.4).

In Chapter 4 and 5, two one-way bridges are introduced, `loonGrob` and `loon.shiny`. The `loonGrob` bridge maps a `loon` visual display to a `grid` visual display. All high-level elements in `loon`, such as a histogram, can be broken down into low-level ones (e.g., rectangles) and mapped to their counterparts in `grid`. In the `loon.shiny` bridge, a `loon` visual display is mapped first to a `grid` visual display via the `loonGrob` bridge, then control features (e.g., buttons, slider bars, etc.) are added in `shiny` to render the `grid` visual display interactive in `shiny`.

In Chapter 6, a new package `loon.tourr` is introduced. We use only the tour engine from `tourr` and replace the non-interactive kinematic `tourr` graphics by interactive `loon` graphics. This is not a bridge (because no transformations are from graphical elements in `tourr` to graphical elements in `loon`); it is an extension of `loon` that provides interactive tours on high dimensional data.

In the last chapter, we review our work, discuss benefits and limitations of bridges, and suggest some future work.

Chapter 2

Loon

The package `loon` (see https://great-northern-diver.github.io/loon/) (Waddell and Oldford, 2020) is an interactive toolkit designed for open-ended, creative and unscripted data exploration, allowing both immediate and mediated manipulation when interacting with a plot. Immediate manipulation is often realized by a mouse or keyboard gestures. Mediated manipulation is supported by a floating palette interface − `loon` inspector (a GUI that can be used to modify plotting states).

`Loon` is the backbone of this thesis where all the bridges are established as connections with it. This chapter begins with introducing the `loon` data structure (details see Waddell, 2016). Typically, `loon` has three different structures, model layer, dependent layer and compound object. Model layer is the main graphics model (e.g., scatterplot, histogram, etc.), controlling the interactivity of a `loon` plot. Dependent layer controls the primitive visuals (e.g., `l_layer_lines`, `l_layer_polygons`, etc.) or adds additional functionality (e.g., `l_navigator`) to a model layer. An overview of five model layers (viz., `l_plot`, `l_plot3D`, `l_hist`, `l_graph` and `l_serialaxes`) and five dependent layers (viz., `l_layer`, `l_graphswitch`, `l_navigator`, `l_context` and `l_glyph`) is given followed by a summary of states controlling their appearance (e.g., plot region, data, aesthetic attributes, linking, selection and etc.).

Compound object, `l_compound`, is a new data structure. It is a compound of `loon` widgets working together to explore data. We focus mainly on two `l_compound` objects, an `l_pairs` plot and an `l_facet` plot. In an `l_pairs` plot (in Subsection 2.4.2), more than a scatterplot matrix, histograms and a high dimensional visualization plot (parallel or radial coordinate plot) can be added. In an `l_facet` plot (in Subsection 2.4.3), a `loon` plot is partitioned into multiple plots where each represents a subset of data.

Before we start, here's a quick clarification on **widget** and **object**. Typically, a **widget** means a `tcltk` data structure, such as `l_plot` widget and `l_layer` widget which are mostly written in `tcltk`. An **object** often refers to an R data structure, such as `l_compound` object. Essentially, an `l_compound` object is a list and each element in this list is a `loon` **widget**.

2.1 Loon Data Structure

A `loon` plot has its own data structure with certain functionality, as shown in Figure 2.1. The dashed line represents *is*. For example, an `l_plot3D` widget *is* an `l_plot` widget, which *is* a `loon` widget; an `l_facet` object *is* an `l_compound` object, which *is* a `loon` object.

The solid line represents *contain*. For example, an `l_ts` object *contains* `l_plot` widgets and `l_layer` widgets.

The curly brace represents *potentially contain*. For example, an `l_facet` object *potentially contains* `l_plot` widgets, `l_hist` widgets, or `l_serialaxes` widgets. An `l_pairs` object *contains* `l_plot` widgets, but it also *potentially contains*`l_hist` widgets or an `l_serialaxes` widget.

Other than naming with the word "loon", three colors, dark gray, light gray and black, are also used to represent three different `loon` models. Dark gray represents a model layer which is the only interactive one and allows contents to be accessible via direct manipulation. Light gray represents a dependent layer which must be attached onto a model layer and cannot be created independently. Black represents a compound object that consists of several model layers.

All `loon` widgets have plotting states which can be queried or changed. For example, in following code, an `l_plot` widget is assigned to a plot handle `p`. The states can be queried by the function `l_cget()` and modified by the function `l_configure()` through this handle `p`.

```
> p <- l_plot(x = 1, y = 1,
+             color = "black")
> # query the color of p
> l_cget(p, "color")
"#000000000000" # black 12 digits hex code
> # modify the color of `p` to red
> l_configure(p, color = "red")
```

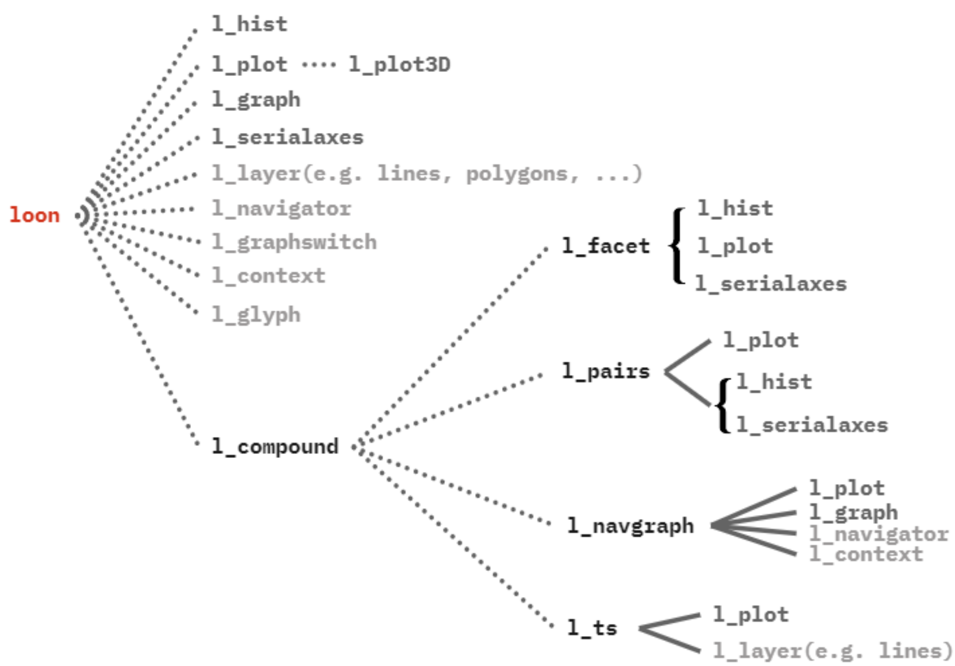


Figure 2.1: The `loon` data structure

```
> l_cget(p, "color")
"#FFFF00000000" # red 12 digits hex code.
```

2.2 Model Layer

Currently, `loon` has five model layers, which are `l_plot`, `l_plot3D`, `l_hist`, `l_graph` and `l_serialaxes`.

2.2.1 Plot Region

Widgets `l_plot`, `l_plot3D`, `l_hist` and `l_graph` display data in the Cartesian coordinate system so that they share similar states which are displayed in Table 2.1. An `l_serialaxes` widget is embedded in the parallel or radial coordinate system which does not share the same graphical states.

Table 2.1: Plot Region

l_hist l_plot l_plot3D l_graph l_serialaxes	Dim	Type	Default
panX, panY	1	double	
zoomX, zoomY	1	pos.double	
deltaX, deltaY	1	pos.double	5/6
minimumMargins	4	nneg.int	(20,20,20,20)
labelMargins	4	nneg.int	(30,30,60,0)
scalesMargins	4	nneg.int	(30,80,0,0)
swapAxes	1	boolean	FALSE

The states `panX`, `panY`, `zoomX`, `zoomY`, `deltaX` and `deltaY` define the data coordinate. The left bottom corner of the data coordinate is [panX, panY] and the top right corner is $[\text{panX} + \frac{\text{deltaX}}{\text{zoomX}}, \text{panY} + \frac{\text{deltaY}}{\text{zoomY}}]$. To map the data coordinate to the plot region is to map [panX, panY] and $[\text{panX} + \frac{\text{deltaX}}{\text{zoomX}}, \text{panY} + \frac{\text{deltaY}}{\text{zoomY}}]$ to [0, 0] and [1, 1] respectively.

A main graphics model splits the display area into four regions, illustrated in Figure 2.2. The three states `minimumMargins`, `scalesMargins` and `labelMargins` control the size of the margin region, the scales region, and the labels region respectively. Each state is a 4 dimensional vector representing the bottom, left, top and right margins (in pixel).

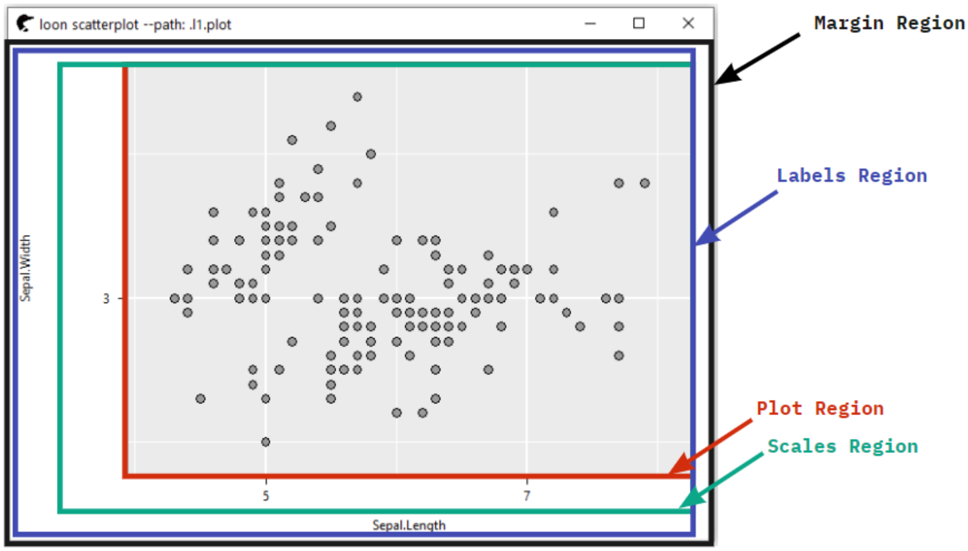


Figure 2.2: The margins of a main graphics model

2.2.2 Data

Table 2.2 shows the input data of each model layer. An `l_hist` widget is to display a one dimensional `loon` histogram. The data in an n-dimensional state `x` is binned before it is displayed. Any changes of attributes (see Subsection 2.2.3) may cause a re-binning of `x` data so that the display is changed accordingly.

An `l_plot` widget is an interactive scatterplot. The states `x` and `y` are used to locate the points. Meanwhile, it also provides states `xTemp` and `yTemp`. The dimensions of these two are either 0 or n. If they are not length 0, the scatterplot will display those temporary coordinates rather than the real coordinates in `x` and `y`. The `l_plot3D` widget is a 3D interactive scatterplot, controlled by states `x`, `y`, and `z`.

A graph display is closely related to a scatterplot display; however, the n dimensional state is now nodes. Edges connect nodes and are specified by the attributes `from` and `to`. The boolean state `isDirected` specifies whether edges have directions.

An `l_serialaxes` widget provides tools to visualize high dimensional data interactively. Data is rendered into a parallel coordinate or a radial coordinate. Each column of the data frame is placed on each axis and each row of the data frame is displayed as an individual line.

Table 2.2: Data

<table>
<tr><th>l_hist</th><th>l_plot</th><th>l_plot3D</th><th>l_graph</th><th>l_serialaxes</th><th>Dim</th><th>Type</th><th>Default</th></tr>
<tr><td>x</td><td colspan="3">x, y</td><td></td><td>n</td><td>double</td><td></td></tr>
<tr><td></td><td></td><td>z</td><td></td><td></td><td>n</td><td>double</td><td></td></tr>
<tr><td></td><td colspan="3">xTemp, yTemp</td><td></td><td>0||n</td><td>double</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>data</td><td>n</td><td>data.frame</td><td></td></tr>
<tr><td></td><td></td><td></td><td>nodes</td><td></td><td>n</td><td>string</td><td></td></tr>
<tr><td></td><td></td><td></td><td>from, to</td><td></td><td>p</td><td>string</td><td></td></tr>
<tr><td></td><td></td><td></td><td>isDirected</td><td></td><td>1</td><td>boolean</td><td></td></tr>
</table>

2.2.3 Attributes

Table 2.3 shows the attributes of each model. The `active` state is a length n boolean vector to control the visibility of elements (e.g., points, bins, lines) for all models. Once the ith one is set as `FALSE`, then the ith element is invisible.

Colors in `loon` use the `Tk` color specifications which are a 12 digit hexadecimal color representations. No transparency is allowed.

Querying is one of the most intuitive interactions in interactive graphics. It is a natural way to display detailed information associated with an element (e.g., a point in a scatterplot or a line in a serialaxes plot). In `loon`, one can customize the information displayed in a toolbox by modifying the `itemLabel` state. For example, in Figure 2.3, the toolbox shows the detailed aspects of the automobile design (e.g., model, year, drv). Once, the mouse hovers over the top-most point, the motor vehicle's information (e.g., "year", "drive way" and "fuel type") is displayed.

The `tag` state is often used in the item bindings which is described in details in Waddell (2016) (Chapter 5).

The `by` state is to split the data into subsets and display each subset of interest in different panels. More details will be described in Subsection 2.4.3.

A binning origin of an `l_hist` widget is controlled by the state `origin` which determines where to start counting the bins. The default value is the minimum value of `x`. A `binwidth` can affect the shape of a histogram and different bin sizes could reveal different features of data. In general, bins need not be of equal width. However, so far, `l_hist` only accepts an equal bin width and the default is computed by Scott (2015) rule. In `l_hist`, the `x` data

Table 2.3: Attributes

<table>
<tr><th>l_hist</th><th>l_plot</th><th>l_plot3D</th><th>l_graph</th><th>l_serialaxes</th><th>Dim</th><th>Type</th><th>Default</th></tr>
<tr><td>—</td><td>—</td><td>active</td><td>—</td><td>—</td><td>n</td><td>boolean</td><td>TRUE</td></tr>
<tr><td>—</td><td>—</td><td>color</td><td>—</td><td>—</td><td>n</td><td>string</td><td>gray60</td></tr>
<tr><td></td><td>—</td><td>size</td><td>—</td><td></td><td>n</td><td>double</td><td>4</td></tr>
<tr><td></td><td>—</td><td>glyph</td><td>—</td><td></td><td>n</td><td>string</td><td>ccircle</td></tr>
<tr><td></td><td>—</td><td>itemLabel, tag</td><td>—</td><td>—</td><td>n</td><td>string</td><td>item0,...,n-1</td></tr>
<tr><td>—</td><td>by</td><td>—</td><td></td><td>—</td><td>0||n</td><td>string</td><td>NULL</td></tr>
<tr><td>origin</td><td></td><td></td><td></td><td></td><td>1</td><td>double</td><td>min(x)</td></tr>
<tr><td>binwidth</td><td></td><td></td><td></td><td></td><td>1</td><td>double</td><td>Scott's Rule</td></tr>
<tr><td>yshows</td><td></td><td></td><td></td><td></td><td>1</td><td>string</td><td>frequency|density</td></tr>
<tr><td>colorStackingOrder</td><td></td><td></td><td></td><td></td><td>g</td><td>string</td><td>selected</td></tr>
<tr><td></td><td></td><td>rotate3DX</td><td></td><td></td><td>1</td><td>double</td><td>0</td></tr>
<tr><td></td><td></td><td>rotate3DY</td><td></td><td></td><td>1</td><td>double</td><td>0</td></tr>
<tr><td></td><td></td><td>axesCoords</td><td></td><td></td><td>3</td><td>list</td><td>(1,0,0)(0,1,0)(0,0,1)</td></tr>
<tr><td></td><td></td><td></td><td>activeNavigator</td><td></td><td>0||1</td><td>string</td><td>char(0L)</td></tr>
<tr><td></td><td></td><td></td><td>activeEdge</td><td></td><td>p</td><td>boolean</td><td>TRUE</td></tr>
<tr><td></td><td></td><td></td><td>orbitDistance</td><td></td><td>1</td><td>double</td><td>15</td></tr>
<tr><td></td><td></td><td></td><td>orbitAngle</td><td></td><td>n</td><td>double</td><td>evenly $0 \sim 2\pi$</td></tr>
<tr><td></td><td></td><td></td><td>colorEdge</td><td></td><td>p</td><td>string</td><td>black</td></tr>
<tr><td></td><td></td><td></td><td></td><td>linewidth</td><td>n</td><td>pos.double</td><td>1</td></tr>
<tr><td></td><td></td><td></td><td></td><td>scaling</td><td>1</td><td>string</td><td>variable|data|
|observation|none</td></tr>
<tr><td></td><td></td><td></td><td></td><td>axesLayout</td><td>1</td><td>string</td><td>radial|parallel</td></tr>
<tr><td></td><td></td><td></td><td></td><td>andrews</td><td>1</td><td>boolean</td><td>FALSE</td></tr>
<tr><td></td><td></td><td></td><td></td><td>sequence</td><td>p</td><td>string</td><td>column.names</td></tr>
</table>

'g': the number of groups; bold string are the default one of the multiple choices

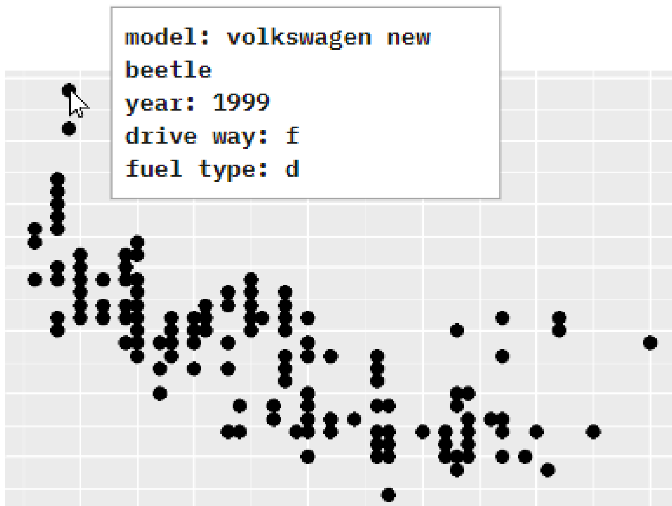


Figure 2.3: The data we used is `mtcars` which was extracted from the 1974 Motor Trend US magazine (Henderson and Velleman, 1981). It comprises the fuel cost and eleven aspects of automobile design (e.g., the number of gears, horsepower, etc.) for 32 automobiles. Here is a scatterplot of the vehicle's horsepower versus miles per gallon.

is partitioned into a `selected` group and separate groups (each group is in one color). A histogram for each group is stacked in order in the display − the selected bins are placed at the bottom and the remaining bins are stacked. One can modify the stacking order via the `colorStackingOrder` state.

In a `loon 3D` plot (an `l_plot3D` widget), the rotation mode can be turned on by typing the `<R>` key on the keyboard. One can either left click the mouse or press the arrow keys to rotate the plot. The states `rotate3DX` and `rotate3DY` represent the rotation angle in the horizontal and vertical direction respectively. The `axesCoords` state is a matrix of projection vectors to draw the axis visual. In the end, typing the `<R>` key again will turn off the rotation mode. Then, one can select or brush as any `l_plot` widget.

Figure 2.4 shows a `loon` graph. `orbits` are labels of `nodes`. Once a `node` is deactivated, the `node`, its `orbit` and all `edges` connected with the `node` will be invisible immediately. The large orange `node` on the right is called `navigator`. Usually, more than one navigators can be added onto a `loon` graph. Details will be introduced in Subsection 2.3.2.

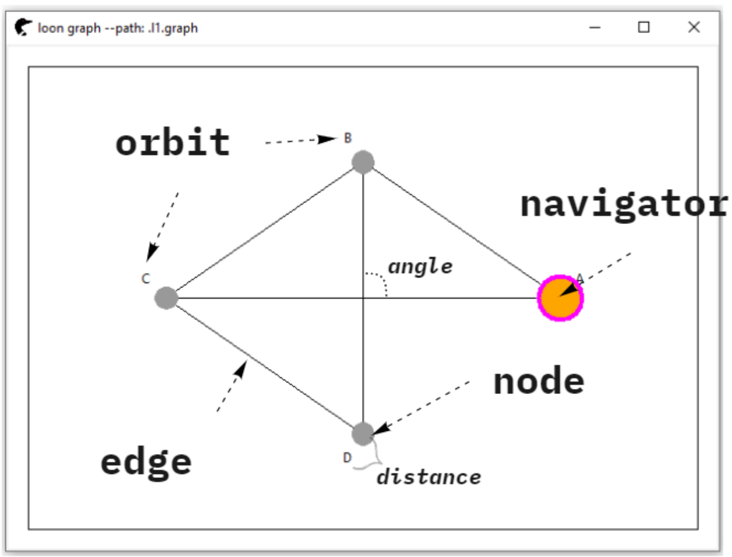


Figure 2.4: A loon nav-graph

In a `loon` serialaxes plot, the `scaling` state determines the way to map data to a plot region. The mathematical expressions are shown in Subsection 4.3.2. The `axesLayout` state is to specify the coordinate system, either `radial` or `parallel`.

If the boolean state `andrews` is set to be `TRUE`, then an Andrews curve (Andrews, 1972) is created (shown in Figure 2.5), as in

```
> s <- l_serialaxes(iris[, -5], andrews = TRUE,
+                   color = iris$Species,
+                   title = "Andrews Curve",
+                   showGuides = FALSE,
+                   axesLayout = "parallel")
```

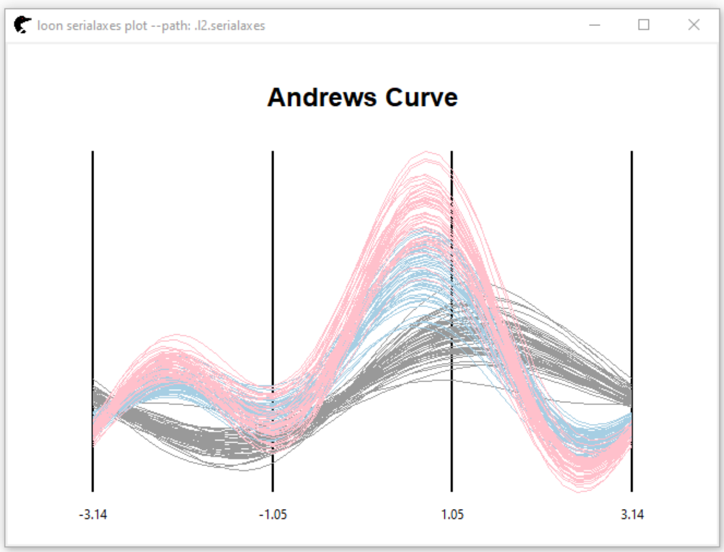


Figure 2.5: Andrews curve of iris data set

The `sequence` state defines the axes sequence of the variables and the default is all columns of the input data.

2.2.4 Linking and Selection

Table 2.4 illustrates the linking and selection states of each `loon` model. `Loon`'s standard linking model is based on three states, `linkingGroup`, `linkingKey` and `linkedStates`. The default `linkingGroup` is `none` which leaves a display un-linked.

Observations in a `loon` plot are uniquely identified (for the purpose of linking) by their `linkingKey`. Within the same `linkingGroup` (not `none`), the same element (determined

Table 2.4: Linking and Selection

<table>
<tr><th>l_hist l_plot l_plot3D l_graph l_serialaxes</th><th>Dim</th><th>Type</th><th>Default</th></tr>
<tr><td>———————— linkingGroup ————————</td><td>1</td><td>string</td><td>none</td></tr>
<tr><td>———————— linkingKey ————————</td><td>n</td><td>string</td><td>(0,...,<n-1>)</td></tr>
<tr><td>———————— linkedStates ———————-</td><td></td><td>string</td><td></td></tr>
<tr><td>———————— selected ————————</td><td>n</td><td>boolean</td><td>FALSE</td></tr>
<tr><td>—————- selectBy —————</td><td>1</td><td>factor</td><td>sweeping|brushing</td></tr>
<tr><td>————- selectionLogic ————-</td><td>1</td><td>factor</td><td>select|deselect|invert</td></tr>
</table>

by `linkingKey`) in different displays will share the same visual features (`linkedStates`). The available `linkedStates` are the n-dimensional attributes listed in Table 2.5 (the bold ones are the default linked states).

Selection is one of the most fundamental tools in interactive graphics. A subset of visual objects of interest can be highlighted. In a scatterplot, the visual objects are point glyphs; in a histogram, they are bins; in a graph, they are nodes and in a serialaxes plot, they are lines. Note that only the `active` (visible) elements can be selected.

There are three `selectionLogics` in `loon`: `select`, `deselect` and `invert`. The first highlights observations as selected; the second downlights them; and the third inverts them (downlighting highlighted observations and highlighting downlighted ones).

In `loon`, selection comes with two modes. Default is `sweeping` where a rectangular region is reshaped or "swept" out to select observations; alternately `brushing` will indicate that a fixed rectangular region is moved about the display to select observations.

2.2.5 Non-data Element States

All non-data element states are listed in Table 2.6. When the `showItemLabels` is set to be `TRUE`, hover the mouse over a point (in scatterplot) or a line (in serialaxes plot), a toolbox will be displayed (as Figure 2.3). When the boolean state `showLabels` or `showScales` is `FALSE`, the labels or scales (axes) will be invisible (meanwhile, the labels margin or the scales margin, as shown in Figure 2.2, will be set as $[0, 0, 0, 0]$). When the state `showGuides` is `TRUE`, the guidelines will show up to help determine point locations.

Table 2.5: Linkable States

Model Layer	linked States
l_hist	**selected, active, color**
l_plot	**selected, active, color, size**, glyph, itemLabel, tag
l_plot3D	**selected, active, color, size**, glyph, itemLabel, tag
l_graph	**selected, active, color, size**, glyph, itemLabel, tag
l_serialaxes	**selected, active, color**, linewidth, itemLabel, tag

The `showBinHandle` state turns the graphical element ⊏⟼ on and off for an `l_hist` widget. If `TRUE`, with the binwidth-handle, one can adjust the bin width or the bin origin directly on the graphic.

The boolean state `showStackedColors` controls whether bins could be partitioned into several groups; if `FALSE`, all non-selected bins are partitioned into one single group and colored thistle (default).

In a serialaxes plot, axes, labels, axesLabels are shown in Figure 2.6. They can be turned on and off by setting states `showAxes`, `showLabels`, and `showAxesLabels`. The default geometric elements are lines (i.e., `showArea = FALSE`). When the state `showArea` is set as `TRUE`, the geometric elements are polygons.

2.3 Dependent Layer

A dependent layer cannot be created alone and must be attached onto a model layer. It does not support linking or selection. Table 2.7 shows the model layer of each dependent layer.

The geometric layer visuals, `l_layer` widgets can only be embedded on the main graphics model (e.g., scatterplot, histogram and graph); an `l_graphswitch` widget provides a graphical user interface element for switching graphs interactively; an `l_navigator` widget turns a graph into a navgraph and the meaning of a navigator's position on the graph can be defined by an `l_context` widget; in a scatterplot, the `l_glyph` widget assigns each point an additional visual representation.

Table 2.6: Non-data Element States

State	l_hist	l_plot	l_plot3D	l_graph	l_serialaxes	Dim	Type	Default
background	—	—	—	—		1	color	white
foreground	—	—	—	—		1	color	black
guidesBackground	—	—	—	—		1	color	gray92
guidelines	—	—	—	—		1	color	white
title	—	—	—	—	—	1	string	
xlabel	—	—	—	—		1	string	x.name
ylabel	—	—	—	—		1	string	y.name
showScales	—	—	—	—		1	boolean	FALSE
showItemLabels		—	—	—	—	1	boolean	FALSE
showLabels	—	—	—	—	—	1	boolean	TRUE
showGuides	—	—	—	—	—	1	boolean	TRUE
showBinHandle	—					1	boolean	FALSE
showOutlines	—					1	boolean	TRUE
showStackedColors	—					1	boolean	TRUE
colorFill	—					1	color	thistle
colorOutline	—					1	color	black
showArea					—	1	boolean	FALSE
showAxesLabels					—	1	boolean	TRUE
showAxes					—	1	boolean	TRUE
axesLabels					—	p	string	col.names

Table 2.7: Dependent layer

Dependent layer	Model layer
l_layer	l_hist, l_plot, l_plot3D, l_graph
l_graphswitch	l_graph
l_navigator	l_graph
l_context	l_navigator
l_glyph	l_plot, l_plot3D, l_graph

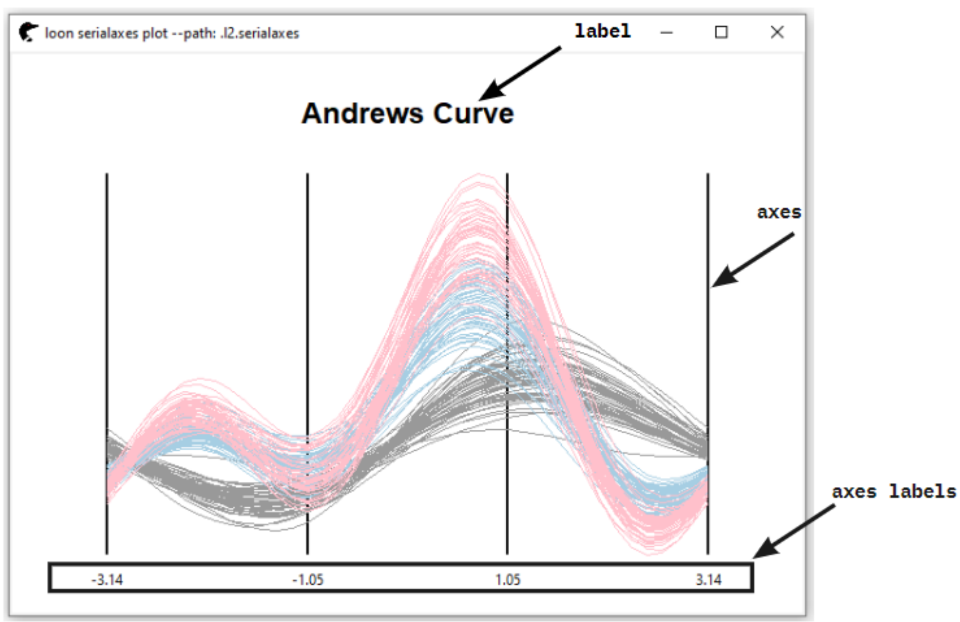


Figure 2.6: Non-data element states in a serialaxes plot

Table 2.8: one-dimensional `l_layer` object

<table>
<tr><th>l_layer_</th><th>text line polygon rectangle oval</th><th>Dim</th><th>Type</th></tr>
<tr><td>Data</td><td>——————- x ——————-
——————- y ——————-</td><td>0||1
0||1</td><td>double
double</td></tr>
<tr><td>Attributes</td><td>—————— color ——————
—————— tag ——————
—————— itemLabel ——————
——————- linewidth ——————
———- linecolor ———-
text
angle
anchor
justify
dash</td><td>0||1

0||1
0||1
1
1
1
1
0||1</td><td>color
string
string
color
color
string
double
string
string
pos.int</td></tr>
</table>

2.3.1 `l_layer`

The one-dimensional geometric layering visuals supported by `loon` are illustrated in Table 2.8. One-dimension means that **x** and **y** are $1 \times k$ numerical vectors where k is determined by the geometric object. For a text visual, k is 1; for a rectangle or oval visual, k is 2; for a line or polygon visual, k can be any number greater than or equal to 2.

For closed geometric objects (i.e., polygon, rectangle and oval), the `color` state represents the filled color. The states `anchor` and `justify` control the direction of a text to be displayed in the widget. The state `anchor` must be one of the values "n", "ne", "e", "se", "s", "sw", "w", "nw", or "center", as shown in Figure 2.7 (a). The state `justify` is to align different text lines and must be one of the "left", "center" and "right", as shown in Figure 2.7 (b).

Table 2.9 shows the p-dimensional geometric layering visuals. The **x** and **y** are p dimensional states. For texts or points, the dimension is $p \times 1$; for rectangles, the dimension is $p \times 2$; for lines or polygons, the dimension of each element (i.e., a line or a polygon) is $1 \times k_j$, where $k_j \geq 2$ and $j \subseteq [1, ..., p]$. The p dimensional ones have a state `active`, controlling the visibility of each element. It is a length p boolean vector.

(a) (b)

Figure 2.7: Figure (a) and (b) show the text anchor and justify. In (a), the pink dot is the reference of the position; in (b), from top to bottom, the justify is “right”, “center” and “left”.

Table 2.9: p-dimensional `l_layer` object

l_layer_	texts	points	lines	polygons	rectangles	Dim	Type
Data	———	———	x	———	———	p	double|list
	———	———	y	———	———	p	double|list
Attributes	———	———	active	———	———	p	color
	———	———	color	———	———	p	color
	———	———	tag	———	———		string
	———	———	itemLabel	———	———		string
		———	linewidth	———	———	p	color
		———	linecolor	———	———	p	color
	text					p	string
	angle					p	double
	anchor					p	string
	justify					p	string
		size				p	string
			dash			p	pos.int

Table 2.10: Attributes of `l_navigator`, `l_graphswitch`

l_navigator	l_graphswitch	Dim	Type
	activewidget	1	string
tag			string
color		1	color
animationPause		1	double
animationProportion-Increment		1	double
scrollProportionIncrement		1	double
from			string
to			string
proportion		1	double
label		0||1	string

2.3.2 `l_navigator` and `l_graphswitch`

Navigators (the large orange node shown in Figure 2.8) could turn a graph into a navgraph (Hurley and Oldford, 2011) and can only be dragged along the edge (path). A `graphswitch` widget provides tools to switch the graphs interactively. Table 2.10 shows the states of these two widgets.

In Figure 2.8, for example, the interface surrounded by a red region is a `graphswitch` widget. The current `activewidget` is a 3D transition. It can be switched to a 4D transition or a 3D and 4D transition directly.

The three states `animationProportionIncrement`, `scrollProportionIncrement` and `animationPause` control the moving speed of a navigator. The first controls the proportion increment in animation; the second controls the delay in ms before moving the navigator to the next position; and the third controls the proportion increment when moving the navigator with the mouse scroll wheel.

The `from`, `to` and `proportion` states record the trace of a navigator (the trace is highlighted in orange). For example, in Figure 2.8, a thick orange line is a `from` path, while a thin orange line is a `to` path. The navigator moves from A:B to A:D, then to C:D and pauses in the half way of the A:D and C:D. Thus, the `from` state is [‘A:B’, ‘A:D’], the `to` state is [‘C:D’] and the `proportion` state is 0.5.

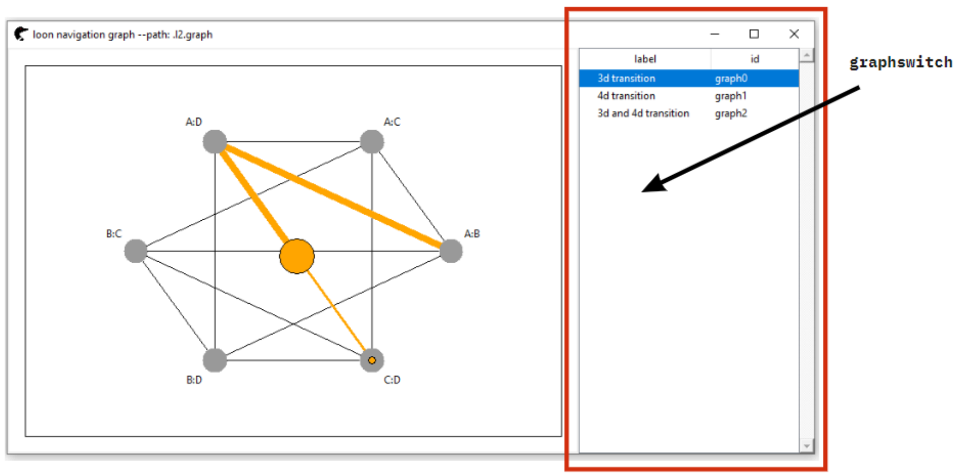


Figure 2.8: The graph switch and navigator

2.3.3 `l_context`

Contexts are implemented in a navgraph, providing standard graph semantics. Every move of a navigator will execute the callback function defined by the `command` state. Table 2.11 shows the states of an `l_context` widget.

Table 2.11: `l_context`

l_context_	context2d geodesic2d slicing2d	Dim	Type
Attributes	————————- command ————————	1	string
	————————- separator ————————	1	string
	———————— interchange4d ————————	1	blooean
	———————— data ————————	p	string
	scaling	1	string
	proportion	1	double
	conditioning4d	1	string

Three contexts `context2d`, `geodesic2d` and `slicing2d` specify three different aspects of a navgraph. The `context2d` maps every location on a graph to a list of x and a list of

y; the geodesic2d maps every location on a graph as an orthogonal projection of the data onto a two-dimensional subspace; the slicing2d context implements slicing by navigation graphs and a scatterplot to condition on one or two variables.

The state separator is a symbol to separate variable names in the nodes of a 2D graph (e.g., ":" in "A:B"). In a 4D transition, for example, in a R^4 space, "A:B" to "C:D", if the state interchange4d is set as TRUE, then the column space A is projected onto the space D and B is projected onto the space C; else the column space A is projected onto C and B is onto D.

The state data contains the data used for projections. Note that the variable names of the data state need to match the node names of a graph. The state scaling determines how the data is scaled to the projection. It could be one of the variable, observation and none (see Subsection 4.3.2 for mathematical expressions).

In the slicing2D context, the state proportion presents the navigator location along the edge to the total length of the edge. The conditioning4d specifies the conditioning method with a 4D edge transition and has to be one of intersection (default), union and sequential (see Waddell, 2016, Chapter 4 for more details).

2.3.4 l_glyph

Glyphs are typographical symbols that are used to introduce items in a plot. Typically, the primitive glyphs are also known as "bullets" in various shapes (e.g., circle, triangle, rectangle and etc.). For each shape, it could be either empty ○, solid ● or filled ◉.

Except these primitive glyphs, loon provides non-primitive glyphs that conveying more information for each point. This information could range from providing a more evocative picture for each point (e.g., a flag for countries' data) to incorporating quantitative information. Table 2.12 shows the states of loon non-primitive glyphs and Figure 2.9 illustrates each of them.

Some aesthetic attributes of an l_glyph widget, such as color, size and coordinates, are determined by its model layer – a scatterplot display. The states shown in Table 2.12 are used for specific glyphs.

A pointrange glyph represents a vertical interval defined by ymin and ymax. When showArea is TRUE, the shapes of points are empty; else, they are solid. In a polygon glyph, states x and y contain the bounding coordinates of the vertices of the polygon. When showArea = TRUE, the polygon will be filled. A serialaxes glyph shows either a radial

Table 2.12: `l_glyph`

l_glyph_	text	pointrange	polygon	serialaxes	image	Dim	Type
data		ymin, ymax				n	double
			x, y			n	double
				data		n	data.frame
Attributes	text					n	string
		——	linewidth	——		n	double
				sequence		p	string
				scaling		1	string
				axesLayout		1	string
				andrews		1	boolean
					images	n	string
Non-data Element States		——	showArea	——		1	boolean
				showAxes		1	boolean
				showEnclosing		1	boolean
				axesColor		1	color
				bboxColor		1	color

or a parallel coordinate glyph by the optional `data`. In an `image` glyph, a picture is used to represent an individual point.

2.4 Compound Object

An `l_compound` object is a list with named elements, each represents a separate interactive `loon` widget and typically, all plots are linked. Currently, `loon` provides five compound objects, `l_navgraph`, `l_ng_plots` (details of these two are shown in Waddell, 2016, Chapter 7), `l_ts`, `l_pairs` and `l_facet`.

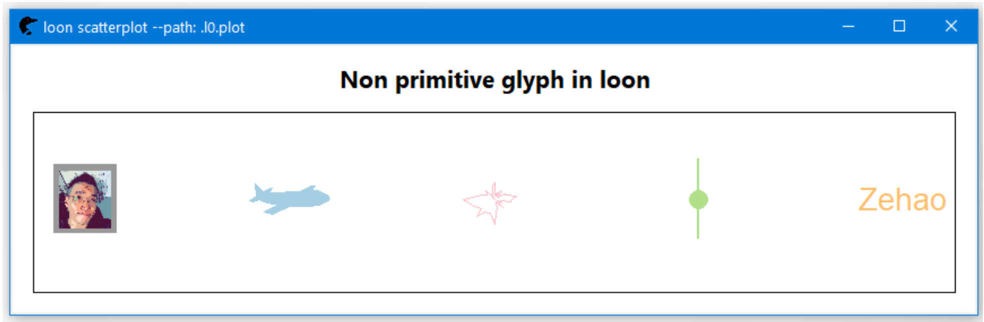


Figure 2.9: Loon non-primitive glyphs. From left to right, the glyph is image, polygon, radial axes, point range and text

2.4.1 `l_ts`

An `l_ts` object is created from a `decomposed.ts` (by the function `decompose()`) object or an `stl` object (by the function `stl()`) (R Core Team, 2013). It has four interactive scatterplots, from top to bottom, drawing the original data, the seasonal component, the trend component and the residuals component (remainders). All four plots are linked so that changes on the plotting states (e.g., `color`, `selected`, `active` and etc.) are synchronized.

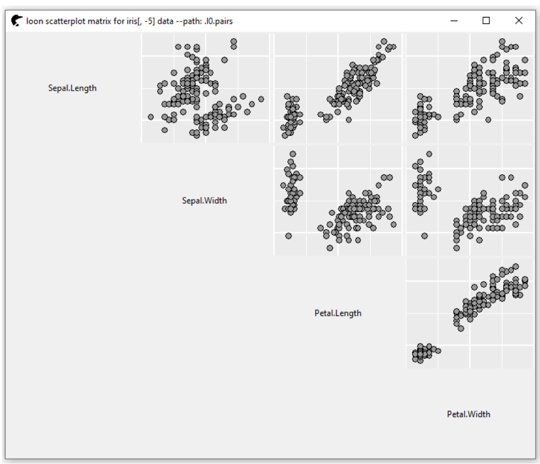
2.4.2 `l_pairs`

An `l_pairs` object of the initial version is a basic scatterplot matrix, as shown in Figure 2.10 (a), containing a collection of scatterplots organized into a grid. All plots are linked. Each shows the relationship between a pair of variables. Scales are aligned either vertically or horizontally; therefore, the patterns can be diagnosed between two plots sharing the common axis.

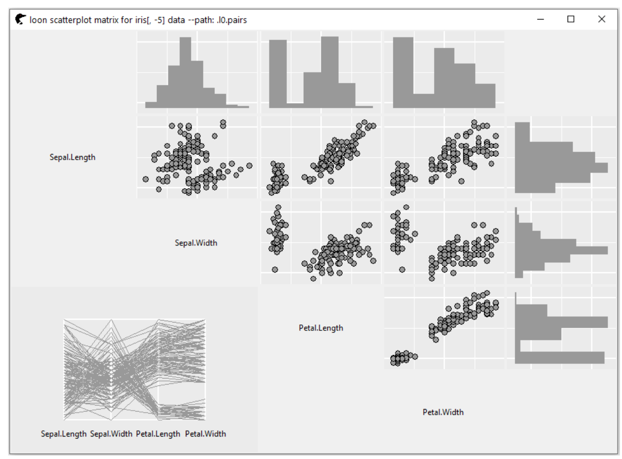
In version 1.2.2, we re-implement it and add new features, such as adding histograms and a serialaxes plot to a pairs plot, as in

```
> pairs <- l_pairs(iris[, -5],
+                  showHistograms = TRUE,
+                  histLocation = "edge",
+                  showSerialAxes = TRUE)
```

In the plot shown below, the histograms are displayed on the edge (the other option is to lay diagonally) and a parallel coordinate plot is displayed in the lower triangle (bottom



(a)



(b)

Figure 2.10: A `loon` pairs plot. The (a) is a traditional scatterplot matrix. In (b), six histograms and a parallel coordinate plot are packed. All these plots (including scatterplots) are linked and some states (e.g., `selected`, `color`) are sharing.

left corner), as shown in Figure 2.10 (b). Each has a set of linked states associated with others so that all plots will change simultaneously whenever linked states get changes in any plot.

In a `loon` pairs plot, the vertical scaling is synchronized in each row and the horizontal scaling is synchronized in each column. The states `zoomY`, `deltaY` and `panY` of plots in the same row are identical, so do the states `zoomX`, `deltaX` and `panX` of plots in the same column. For example, in Figure 2.11, the scales of scatterplot "Petal.Width" versus "Sepal.Width" (the center one in this grid) are panned towards the top right. The direction and the moving distance forms a vector that is able to be decomposed into two orthogonal vectors. All plots in the same column move along horizontally and all plots in the same row move along vertically.

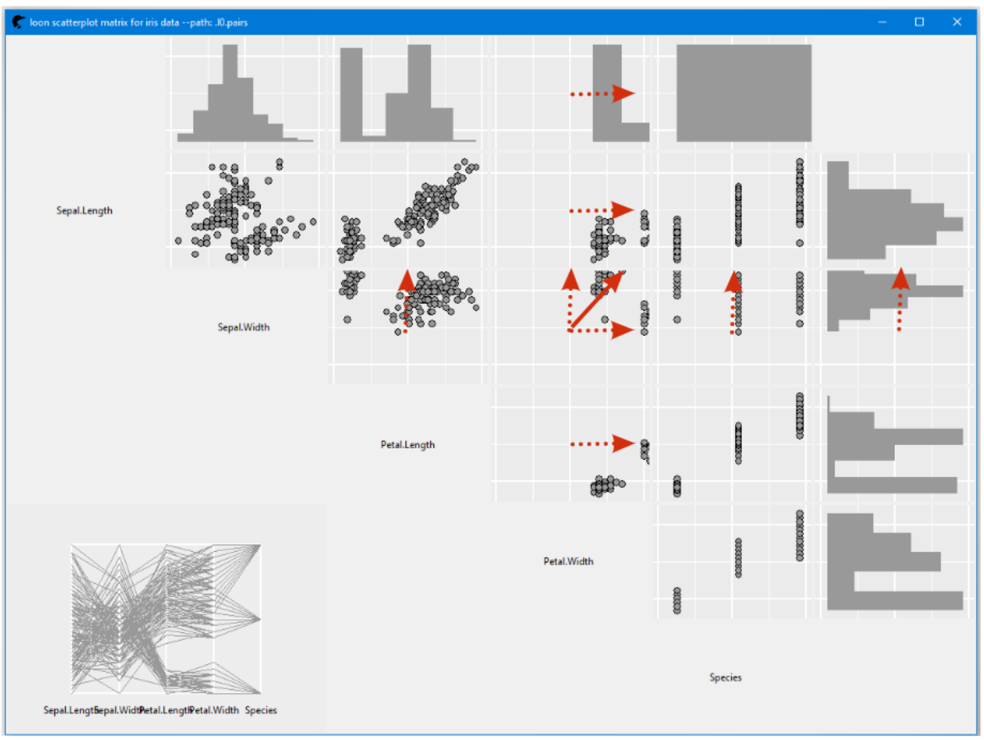


Figure 2.11: Scaling synchronization in a `loon` pairs plot. The center scatterplot is moving towards the north east. Then, all plots sharing the same vertical scaling would move towards the north and all plots share the same horizontal scaling would move towards the east.

The names of each plot are the layout positions. For example, the name of the scatterplot, “Sepal.Width” versus “Sepal.Length”, is “x2y2” (the order is from left to right, from top to bottom). No names are given to a serialaxes plot because at most one serialaxes plot can be displayed in an `l_pairs` object and the layout position must be the lower triangle.

2.4.3 `l_facet`

With facets, the original plot can be partitioned into multiple panels and each panel illustrates one subset of data. The motivation of creating the `l_facet` object comes from transforming a `ggplot2` object to a `loon` widget (discussed in Chapter 3). The package `ggplot2` provides `Facet` components to assign data to different panels, which is not available in `loon` before version 1.3.0.

Table 2.13 shows an arbitrary data set. With the data, the following code creates a `loon` scatterplot, as shown in Figure 2.12,

Table 2.13: Arbitrary Data

Data	Coords	Factor 1	Factor 2	color
observation 1	(1,1)	A	C	red
observation 2	(2,2)	A	D	red
observation 3	(3,3)	A	C	blue
observation 4	(4,4)	B	D	blue
observation 5	(5,5)	B	C	blue
observation 6	(6,6)	B	D	blue

```
> color <- c(rep("blue", 2), rep("red", 4))
> fp <- l_plot(x = 1:6, y = 1:6, size = 50,
+              color = color)
> g <- l_glyph_add_text(fp, text = 1:6)
> fp['glyph'] <- g
```

In `loon`, facets are created in two ways: from an existing `loon` plot (by the function `l_facet()`) or ab initio at the time that the `loon` plot is created (by setting argument `by`).

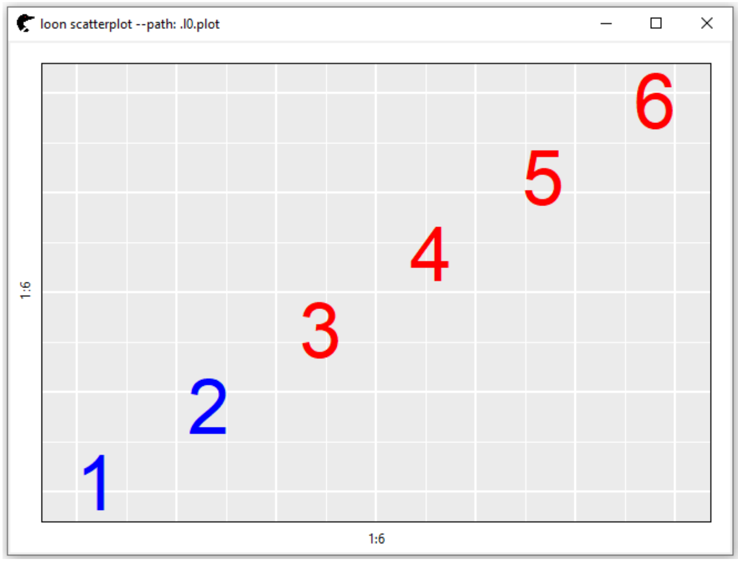


Figure 2.12: Plot for the arbitrary data set

Argument `by`

The widget `fp` can be split by:

- **the** n**-dim states**: `color`, `size`, `glyph` (see Table 2.3), as in

  ```
  > f1 <- l_facet(fp, by = "color")
  > # which is equivalent to
  > # f1 <- l_plot(x = 1:6, y = 1:6, size = 50,
  > #              color = color, by = "color")
  ```

  `f1` is an `l_facet` object composed of two `loon` scatterplots: one has two blue points with label 1, 2 and the other has 4 red ones with label 3, 4, 5 and 6, as shown in Figure 2.13 (a).

- **an** n**-dim data frame or list**, as in

  ```
  > multiFac <- data.frame(Factor1 = c(rep("A", 3),
  +                                    rep("B", 3)),
  +                        Factor2 = rep(c("C", "D"), 3))
  > f2 <- l_facet(fp, by = multiFac)
  > # which is equivalent to
  ```

```
> # f2 <- l_plot(x = 1:6, y = 1:6, size = 50,
> #              color = color, by = multiFac)
```

In Figure 2.13 (b), the original data (with 6 observations) is split into four groups by this arbitrary data frame “multiFac” where the observation 1 and 3 are in the group “A:C”; the 2 is in group “A:D”; the 4 and 6 are in group “B:D” and the 5 is in group “B:C”.

- **a formula**, as in

  ```
  > f3 <- l_facet(fp, by = Factor2 ~ Factor1, on = multiFac)
  > # which is equivalent to
  > # f3 <- l_plot(x = 1:6, y = 1:6, size = 50,
  > #              color = color, by = Factor2 ~ Factor1,
  > #              on = multiFac)
  ```

  `f3` is identical to `f2`, as shown in Figure 2.13 (b). An optional data frame `on` contains the variables in this formula. When the variables are not found in the data frame `on`, they are taken from the environment, typically the environment from which the function is called. Note that, the formula also accommodates n dimensional states. For example, setting `by = ∼ color` will return a identical graph as Figure 2.13 (a).

Argument `layout`

The function `l_facets()` provides three layouts, “grid”, “wrap” and “separate”.

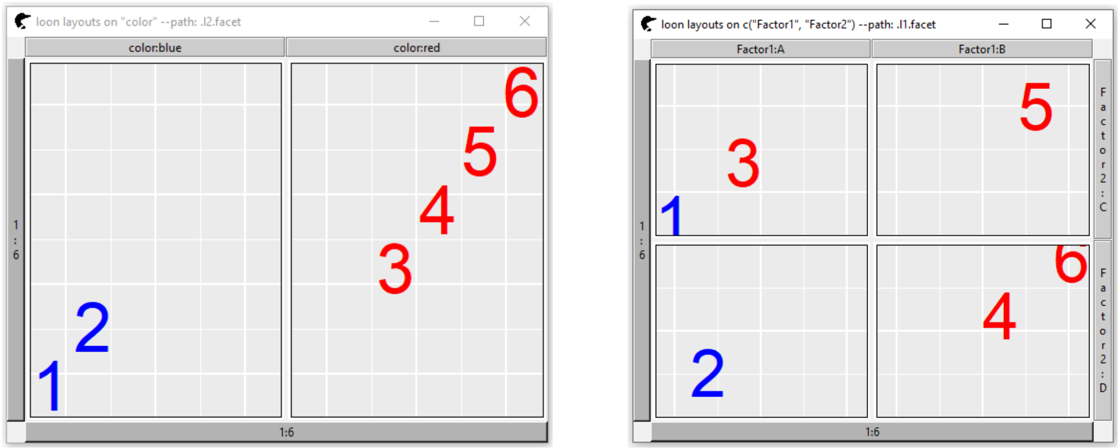
- `layout = ‘‘grid’’`: by default; the panels are packed in two dimensions, row and column, as in

  ```
  > # facet grid (# same as ‘f2’)
  > f4 <- l_facet(fp, by = multiFac, layout = "grid")
  ```

  Figure 2.13 shows the “grid” layout. The values of the “Factor1” is spread down the rows and “Factor2” is spread across the columns. If the `by` is a formula, the variables before the tilde will be taken as the row names (vertically) and the ones after the tilde will be considered as the column names (horizontally).

- `layout = ‘‘wrap’’`: makes a long ribbon of panels and wraps it into 2D, as in

  ```
  > # facet wrap
  > f5 <- l_facet(fp, by = data, layout = "wrap")
  ```



(a) (b)

Figure 2.13: The logic is set by argument `by` which accommodates three types: an n dimensional state, a data frame and a formula.

In Figure 2.14 (a), a long ribbon of four panels are wrapped in a 2×2 table. The arguments `nrow` and `ncol` control the number of rows and columns. If not set, the function `n2mfrow()` is applied to find an appropriate number of rows and columns.

- `layout = "separate"`: the panels are unpacked, as in

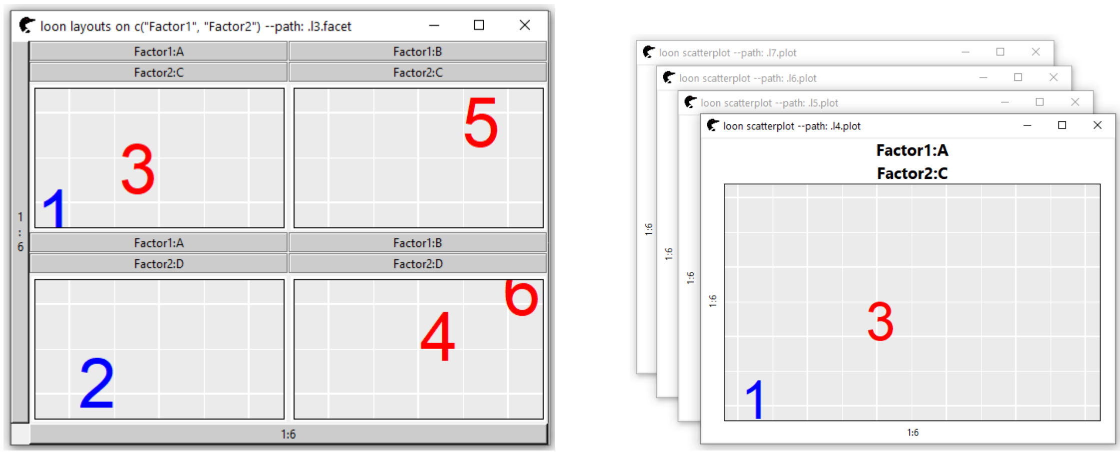
```
> # facet separate
> f6 <- l_facet(fp, by = data, layout = "separate")
```

In Figure 2.14 (b), four isolated windows are displayed.

Scaling Synchronization

As an `l_pairs` object, the scales of each plot are synchronized by `connectedScales` that determines how the scales of the facets are connected. For each `layout` type,

- `layout = "grid"`: when `connectedScales` is
  - `cross`: only the scales in the same row and the same column are connected;



(a) (b)

Figure 2.14: The display is set by argument `layout` which accommodates three types: "grid" (default, shown as Figure 2.13), "wrap" (a) and "separate"(b)

  - `row`: both `x` and `y` scales of facets in the same row are connected;
  - `column`: both `x` and `y` scales of facets in the same column are connected;
  - `x`: only the `x` scales are connected;
  - `y`: only the `y` scales are connected;
  - `both`: both `x` and `y` scales are connected;
  - `none`: neither `x` nor `y` scales are connected.

- `layout = ‘‘wrap’’`: for all plots, when `connectedScales` is
  - `x`: only the `x` scales are connected;
  - `y`: only the `y` scales are connected;
  - `both`: both `x` and `y` scales are connected;
  - `none`: neither `x` nor `y` scales are connected.
- `layout = ‘‘separate’’`: the `connectedScales` will be set as `none` in mandatory that neither scales are connected.

2.5 Summary

This chapter reviewed the loon visual structures. We mainly discussed five model layers (e.g., l_plot, l_hist) and five dependent layers (e.g., l_layer, l_glyph), as well as their plotting states. This gave an overview of the abstraction levels in loon plots which need to be matched, where possible, in any target graphical system $\mathcal{K}$ that is to be bridged to the graphical system loon ($\mathcal{G}$). The chapter also introduced new visual structures via l_compound and l_facet, extending loon in preparation for bridges between loon and other packages. These bridges will be the focus of following chapters.

The new structure, l_compound, was introduced to extend loon to organize multiple plots in a single display. This is very much like the arrangeGrob functionality from the package gridExtra (Auguie, 2017) which extends grid to multiple displays. Extending loon by l_compound allows multiple displays to be matched in a bridge between loon and grid. This will be used in Chapter 4. Similarly, l_compound matches the patchwork structure of the package patchwork (Pedersen, 2020a) which extends the package ggplot2 to accommodate multiple plots in a single display. The bridge between loon and ggplot2 is developed in Chapter 3.

The new structure l_facet is an l_compound object specially designed to extend the graphical system loon to better match the visual structure − a Facet object in ggplot2. Again, this allows a more complete, and two-way, bridge to be constructed between loon and ggplot2 in Chapter 3.

These new visual structures both extend loon by adding elements g_i in $\mathcal{G}$ to match k_i in different graphical packages $\mathcal{K}$ (i.e., grid or ggplot2) and do so by matching levels of abstraction. Following chapters develop bridges between the now extended loon and other graphical systems in R.

Chapter 3

Loon.ggplot

The ggplot2 package uses the base grid package to produce publication quality graphics. Based on a grammar of graphics, ggplot2 also provides a lot of functionality that can be extremely useful in data analysis. The loon package provides interactive graphics which are especially valuable in exploratory data analysis.

The package loon.ggplot (see https://great-northern-diver.github.io/loon.ggplot) (Xu and Oldford, 2019a) is a two-way bridge, bringing both packages ggplot2 and loon together. Data analysts who value the ease with which ggplot2 can create meaningful graphics can now turn these ggplots into interactive loon plots for more direct interaction with their data. Conversely, data analysts who explore data interactively can turn a snapshot of their interactive loon plots into ggplots at any time.

This chapter begins with an overview of the package ggplot2. Both loon and ggplot2 packages mainly provide functionalities at a relatively high level of graphical abstraction to create complete plots, but some of the high-level elements may not be able to get mapped from one package to the other. In order to have more high-level ones mapped, an extension of the package ggplot2, ggmulti is introduced which enables more high dimensional visualization functionality.

Furthermore, details of how the bridge (i.e., loon.ggplot) is constructed are discussed, such as the transformations of aesthetic attributes, how to map elements from one to the other and et cetera. Additionally, with the bridge, we are able to extend a grammar of graphics to a grammar of interactive graphics.

This chapter closes with a summary of this bridge, including lessons learned and loon.ggplot's limitations.

3.1 Introduction of `ggplot2`

3.1.1 A Grammar of Graphics

A grammar of graphics is a tool to describe deep features of statistical graphics.

> "Many have used some type of data flow to illustrate how visualization systems work. Few have identified the necessary sub-sequences these systems must follow."
> (Wilkinson, 2005)

The grammar tells people how to map from a data set to aesthetic attributes (e.g., color) of geometric objects (e.g., bars). "Such a grammar allows us to move beyond named graphics (e.g., the scatterplot) and gain insight into the deep structure that underlies statistical graphics" (Wickham, 2010).

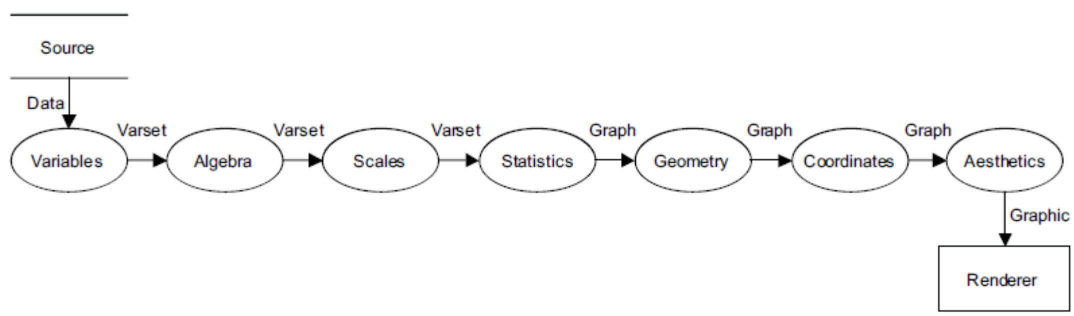


Figure 3.1: From data to graphic (Wilkinson, 2005).

Once the raw data is given, variables of interest are extracted (sometimes, some "algebra" is applied on the variables). Then, "scales" are to determine how to perceive aesthetic attributes (e.g., color, size). "Statistics", "geometry" and "coordinates" are operated one after the other, mainly to: alter the position of graphics; determine the geometric objects; and locate the points in space, respectively. We have created a graph so far, but the graph is a mathematical abstraction. To sense mathematical abstractions, perceivable forms should be given to the abstraction. "Aesthetic" functions are to translate a graph into a graphic. After rendering, the graphic is displayed.

The ordering of stages in the pipeline are not changeable. Obviously, we cannot apply aesthetics before we determine geometric objects. For a more detailed description, see Wilkinson (2005).

The package `ggplot2` (Wickham, 2016) is a statistical graphical system with an underlying grammar called layered grammar of graphics (Wickham, 2010) which is based on the grammar of graphics and embedded in the R environment. "Layered" means that each layer can have its own geometric object, statistical transformation, position adjustment, data set and mapping system.

3.1.2 Components

Each `ggplot` object is composed of six components.

1. "Data": what users want to visualize. Mapping aesthetics are required from users to locate the variables mapped onto the axes.

2. "Layers": control geometric objects (e.g., points, lines, polygons) and statistical transformations (used to summarize the data of interest).

3. "Scales": help map values from the data space to the aesthetic space. For example, use color, size or shape to represent variables.

4. "Coordinate System": combines the two position aesthetics (x and y) to produce a 2D position on the plot. The most commonly used coordinate system is the Cartesian coordinate system (by default). `Ggplot2` provides various systems, such as polar coordinate system to produce pie charts and map coordinate system to project a spherical plane onto a flat 2D plane.

5. "Facets": divide data into subsets and display them on multiple panels. They are extremely useful when comparing the pattern of interest across different subsets. The default is no facet (i.e., `facet_null()`).

6. "Theme": a central control of non-data elements display, such as title, fonts and legends position.

3.1.3 Programming

The package ggplot2 is programmed based on prototype programming (a style of object-oriented programming) where a generalized object can be cloned and extended. For example, a geom_**() function (e.g., geom_path()) returns a layer, responsible for rendering the data in a plot. Essentially, the output is a Geom** object (e.g., GeomPath) which is a prototype object ggproto.

The top-level ggproto (viz., Geom, Coord, Stat, Facet, Position and Scale) declares general elements, such as the element required_aes in Geom. For a particular object descending from the top-level object, any non specified elements (e.g., required_aes) are inherited from the settings of the default top-level object and any specified elements will overwrite the default settings.

Another useful design in ggplot2 is that each ggplot object is only executed at the printing time. For example,

```
> p <- ggplot(data = mtcars,
+             mapping = aes(x = mpg, y = hp)) +
+      geom_point()
```

p is a ggplot object with a point layer visual. The graphic will not be displayed until we print() or plot() it in the console

```
> # print(p)
> p
```

Note that, the function print() is called automatically in R. As we type p in the console, print(p) is executed to return its argument invisibly.

The print() function is a generic function (an extended function object, containing information used in dispatching methods for this function). In our case, since p is a ggplot object, a method will be dispatched to function print.ggplot() to render the p as a graphic. More of the function print() will be discussed in Subsection 3.3.3.

The prototype programming design can make the extension of the package ggplot2 easily: the package ggmulti is created to extend ggplot2 in high dimensional data visualization (see Section 3.2); a grammar of graphics is extended to a grammar of interactive graphics (see Subsection 3.3.3).

3.2 ggmulti: an Extension of ggplot2

The package ggmulti (see https://great-northern-diver.github.io/ggmulti/) (Xu and Oldford, 2020) extends the ggplot2 package to add high dimensional visualization functionality such as serialaxes coordinates (e.g., parallel and radial) and multivariate scatterplot glyphs (e.g., encoding many variables in a radial axes or star glyph).

3.2.1 Serialaxes in ggplot2

Serialaxes coordinate is a methodology for visualizing a high dimensional data (typically, $p > 2$). The axes can be a finite p space or an infinite space (e.g., Fourier transformation).

In a finite p space, all axes can be displayed in parallel (the parallel coordinate plot) or under a polar coordinate (the radial coordinate plot). In an infinite space, a mathematical transformation needs applying.

A point in Euclidean p-space, R^p, is represented as a line in a serialaxes coordinate and a point $\leftrightarrow$ line duality is induced in the Euclidean plane R^2 (Inselberg and Dimsdale, 1990).

In the ggplot2 syntax, one sets coordinate systems by adding a coord_** component. To be consistent, the serialaxes coordinate is realized by adding coord_serialaxes(). In a ggplot object, states x and y are often provided in the mapping aesthetics (aes()) to locate the variables on axes. However, x and y are not necessarily required in the serialaxes coordinate system. Any variable could be used to declare a single axis. For example, the following code shows how to create a serialaxes object in ggplot2.

```
> pm <- ggplot(iris,
+         mapping = aes(
+           x1 = Sepal.Length,
+           x2 = Sepal.Width,
+           x3 = Petal.Length,
+           x4 = Petal.Width
+         )) +
+   geom_path(alpha = 0.3) +
+   coord_serialaxes(scaling = "variable")
> pm
```

In Figure 3.2, the axes, x1, x2, x3 and x4 are defined in function aes() by variables sepal length, sepal width, petal length and petal width. The layer geom_path() is used to draw lines and each line represents an observation. In this case, the data has been scaled by `variable` so that values for each axis is scaled to [0, 1]. To modify scaling methods, one can set:

```
> geom_path(stat = "serialaxes", scaling = **)
```

where ** is one of the `variable`, `observation`, `data` and `none` (mathematical expressions are shown in Subsection 4.3.2).

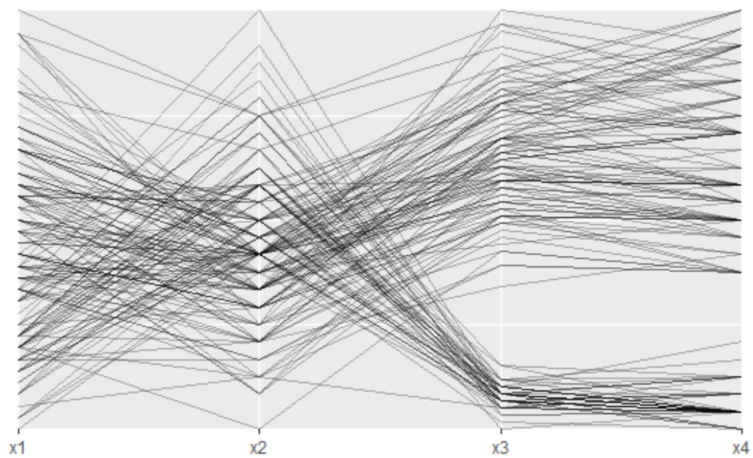


Figure 3.2: Serialaxes in `ggplot`

Other than lines, on each axis, 1D layers (e.g., `geom_histogram`) or the quantile layer (`geom_quantiles`) can be added to reveal the pattern of interest as well. For example,

```
> pm <- pm +
+   geom_histogram(alpha = 0.5) +
+   geom_quantiles(color = c("firebrick", "steelblue", "khaki"),
+                  quantiles = c(0.25, 0.5, 0.75),
+                  size = 2)
> pm
```

In Figure 3.3, the distributions of axes `x1` and `x2` are nearly symmetric. The distances between q_3 (upper quantile) and q_2 (median), and q_2 and q_1 (lower quantile) are almost identical. However, the distributions of axes `x3` and `x4` are highly right-skewed. The distances between q_3 and q_2 are significantly smaller than the distances between q_2 and q_1.

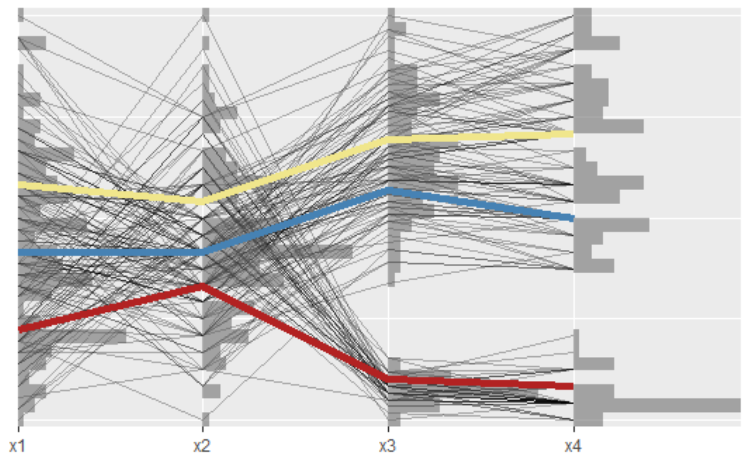


Figure 3.3: Serialaxes in `ggplot` with histogram layer and quantiles layer

Andrews (1972) plot is a way to project multi-response observations into a function $f(t)$, by defining $f(t)$ as an inner product of the observed values of responses and orthonormal functions in t

$$f_{\mathbf{y}_i}(t) =< s(\mathbf{y}_i), \mathbf{a}_t >$$

where $\mathbf{y}_i$ is the ith response; function $s()$ is a scaling method (see 4.3.2); $\mathbf{a}_t$ is the orthonormal functions within a certain interval. Andrew suggests to use the Fourier transformation

$$\mathbf{a}_t = [\frac{1}{\sqrt{2}}, \sin(t), \cos(t), \sin(2t), \cos(2t), ...]^{\mathsf{T}}$$

which are orthonormal in the interval $(-\pi,\pi)$. In this way, a p dimensional space is projected onto an infinite space. To implement the dot product statistical transformation, one can easily set `stat` in `geom_path()`. For example,

```
> ggplot(iris,
+        mapping = aes(Sepal.Length = Sepal.Length,
+                      Sepal.Width = Sepal.Width,
+                      Petal.Length = Petal.Length,
+                      Petal.Width = Petal.Width,
+                      color = Species)) +
+   geom_path(alpha = 0.2,
+             stat = "dotProduct",
```

```
+                scaling = "none") +
+   coord_serialaxes()
```

Figure 3.4 shows these curves.

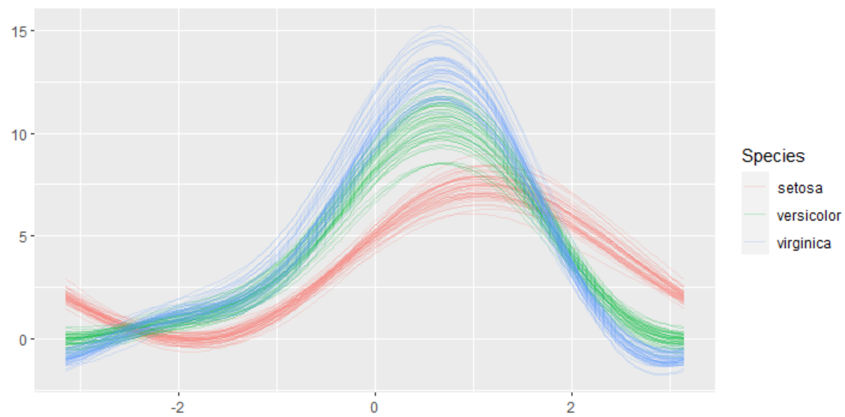


Figure 3.4: Andrews curve for `iris` data

In addition, one can customize projection vectors. In order to cover more space of the sphere, Tukey proposed the idea of $\mathbf{a}_t$ as

$$\mathbf{a}_t = [\cos(t), \cos(\sqrt{2}t), \cos(\sqrt{3}t), \cos(\sqrt{5}t), ...]^\mathsf{T}$$

where $t \in [0, k\pi]$ (Gnanadesikan, 1977). The following code shows how to implement Tukey's curve in `ggmulti`, as shown in Figure 3.5.

```
> tukey <- function(p = 4, k = 50 * (p - 1), ...) {
+   t <- seq(0, p* base::pi, length.out = k)
+   seq_k <- seq(p)
+   values <- sapply(seq_k,
+                    function(i) {
+                      if(i == 1) return(cos(t))
+                      if(i == 2) return(cos(sqrt(2) * t))
+                      Fibonacci <- seq_k[i - 1] + seq_k[i - 2]
+                      cos(sqrt(Fibonacci) * t)
+                    })
+   list(
+     vector = t,
```

```
+     matrix = matrix(values, nrow = p, byrow = TRUE)
+   )
+ }
+ ggplot(iris,
+        mapping = aes(Sepal.Length = Sepal.Length,
+                      Sepal.Width = Sepal.Width,
+                      Petal.Length = Petal.Length,
+                      Petal.Width = Petal.Width,
+                      color = Species)) +
+   geom_path(alpha = 0.2,
+             stat = "dotProduct",
+             transform = tukey
+             scaling = "none") +
+   coord_serialaxes()
```

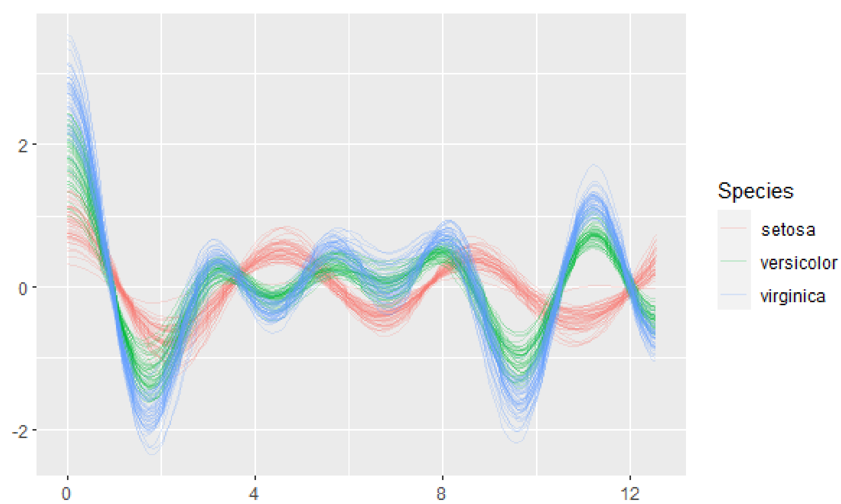


Figure 3.5: Tukey's curve

3.2.2 Non-primitive Glyphs in `ggplot2`

Glyphs can be used as point symbols in a scatterplot to convey more information on each point. This information could range from providing a more evocative picture for each point (e.g., an airplane for flight data or a team's logo for sports data) to incorporating quantitative information (e.g., the values of other variables in a serialaxes or star glyph or as a Chernoff face, Chernoff, 1973). The following code shows how to construct three points in `ggplot2`, as shown in Figure 2.12, from left to right, a serialaxes glyph, a maple (polygon) glyph and an image glyph.

```
> ggplot(mapping = aes(x, y)) +
+   geom_serialaxes_glyph(
+     data = data.frame(x = 1, y = 1),
+     serialaxes.data = iris[1L, ],
+     # parallel or radial axes
+     axes.layout = "parallel",
+     # scaling method
+     scaling = "variable",
+     # sequence of serialaxes
+     axes.sequence = sample(colnames(iris), 10, replace = TRUE)
+   ) +
+   geom_polygon_glyph(
+     data = data.frame(x = 2, y = 1),
+     polygon_x = ggmulti::x_maple,
+     polygon_y = ggmulti::y_maple,
+     fill = "red"
+   ) +
+   geom_image_glyph(
+     data = data.frame(x = 3, y = 1),
+     images = png::readPNG("me.png"),
+     imagewidth = 1,
+     imageheight = 1
+   ) +
+   coord_cartesian(xlim = extendrange(c(1,3)),
+                   ylim = extendrange(c(1,2)))
```

3.3 ggplot2 to loon

3.3.1 Making ggplot2 Interactive

The grammar of graphics does not include interactivity; therefore the `ggplot2` package only creates static plots. How does a `ggplot` object become interactive?

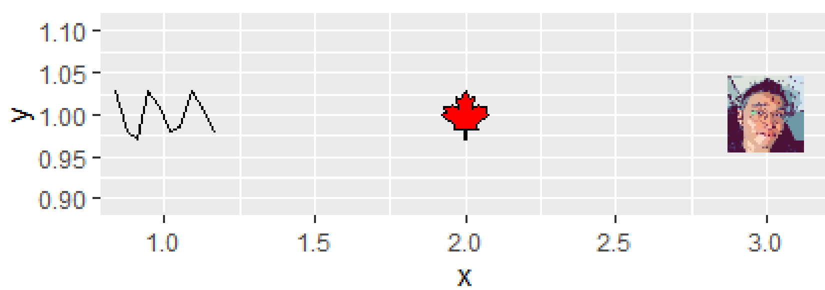


Figure 3.6: Non primitive glyph in `ggplot` object

Functions `identify()` and `locator()` are provided in the base R standard graphics `grDevices` to mark the mouse click. They read the position of the graphics pointer when the (first) mouse button is pressed, which helps realize simple point selection. The functions can even be used to build a simple GUI such as one for the game tic tac toe (e.g., see Lawrence and Verzani, 2018). However, besides not supporting common graphical tools such as buttons, checkboxes, and sliders, the base R functions run without interruption, forcing users to halt the interactive session whenever they want to interact programmatically with the selection information.

Several packages have been developed over the last few years which add some interactivity to `ggplot`. The most interactive of these are browser based. Of these, the most downloaded include `ggvis`, `ggiraph`, and `animint2`. Others such as `gganimate` provide tools to render plots kinematically.

The `ggvis` (Chang and Wickham, 2018) package is based on `shiny`'s reactive programming model. The graphics are rendered in a web browser using `Vega`, providing a rich set of GUI tools. For example, users can brush, link, and even adjust a histogram's bin width with a slider bar. Moreover, a `ggvis` widget can be embedded to a `shiny` app. Unfortunately, every interactive `ggvis` plot must be connected to a running R session which means once any `ggvis` widget is rendered, its components are determined at compile time and fixed at run time. No modification from the R console is possible until the running session is stopped.

The `ggiraph` (Gohel and Skintzos, 2019) package allows users to add tooltips, animations and `Javascript` actions to `ggplot` graphics to achieve interactivity. The `animint2` (Sievert et al., 2019) package, an extension of `ggplot2`'s implementation of a grammar of graphics, allows one to write `ggplot2` code and produce a standalone web page with multiple linked views (Hocking et al., 2020). However, they both suffer a very similar problem with ggvis. Once the plot is rendered, graphical components specified at compile time become difficult to manipulate outside the browser interface. Moreover, interactive graphics built on top of browser languages such as HTML, Javascript will result in all associated widgets being stuck in one large scrollable window (e.g., a single browser). Browser windows are independent, making it difficult, if not impossible, for graphics to be shared between different browsers. Browser design makes it hard to interact with many plots at once without effectively having the browser implement an entire desktop interface within it.

The gganimate (Pedersen and Robinson, 2019) package includes the description of animation and returns a gif_image object. Consequently, gganimate is more a kinematic graphics package than it is an interactive graphics one.

Other possible ways to make ggplot interactive depend on third party GUI available platforms. These include, for example, the rJava package based on Java GUI system, the RGtk2 package which binds R and Gtk+, the qtbase package based on the Qt framework, and the tcltk package based on Tk GUI components of Tcl. All these systems provide rich toolkits to realize interactivity.

Here, we choose tcltk (loon) as our interactive system. The main reasons are: first, Tk is one of the most widely used for GUIs; second, many languages can bind Tk including R, Python, Ruby and Perl; third, to use rJava, qtbase or RGtk2, R users have to install platform, Java, Qt or Gtk+ first. In contrast, no such installation issues arise for R users of the tcltk. Some OSes, like the MS windows system and McIntosh's OS X ship with tcltk already installed (Lawrence and Verzani, 2018); so does R itself. It is a base package in R maintained by R Core Team (2013).

The package loon.ggplot provides the function ggplot2loon() that could transform a ggplot object to an interactive loon widget.

3.3.2 Transformations

Attributes

The graphical system ggplot2 uses the engine of grid to draw graphics. It inherits almost all aesthetic attribute settings in grid (see Section 4.2).

The package ggplot2 provides the most commonly used 25 primitive point symbols, while loon only provides 12. If a ggplot object uses point symbols defined in Table 4.1 (right), a corresponding glyph from the left would be mapped in loon. However, if a ggplot object uses point symbols not defined in Table 4.1, the default `loon` glyph (`ccircle`: circle with boundary) would be mapped.

The package `ggplot2` uses 8 (6 + 2 transparency) digit hexadecimal color code. Regardless of the last 2 digit (transparency), the color is converted to a 12 digit hexadecimal color in `loon`. However, if a point is drawn with a very low alpha value (almost transparent), completely ignoring the transparency would not be a good mapping. Consequently, we set: if the alpha value is less than 0.5 (alpha level in [0,1]), a point symbol with a filled or bounded shape will be mapped to a point with an empty shape (e.g., ● or ◉ to ○).

The package `ggplot2` multiplies size by two constants `.pt` and `.stroke` in order to convert the unit `lwd` and `fontsize` to the unit `mm`. To transform a `ggplot` point size to a `loon` size, first convert the unit `mm` to `px` (pixel), then map the size based on the transformations shown in Figure 4.2.

Layer

To build a `ggplot` object, layer `ggplot()` initializes the whole object such as to declare input data frame and to specify a set of aesthetic attributes. The data and aesthetic attributes are intended to be common throughout all subsequent layers.

Note that layer `ggplot()` does not specify a particular geometric output (which are specified by adding `Geom`/`Stats` layers). In contrast, a `loon` model always begins with a model layer that specifies the geometric visual. For example, model layer `l_plot` creates an interactive scatterplot; `l_hist` creates an interactive histogram.

During transformation, a `ggplot2` histogram layer is mapped to `l_hist` and a points layer is mapped to `l_plot` (see Table C.1, C.2, C.3, C.4, from right to left[1]). Suppose a `ggplot` object has two layers, a histogram layer and a points layer, in the mapping, this object should be mapped to a `loon` widget composed of `l_hist` and `l_plot`. However, it is not available in `loon` as `loon` only supports one model layer. So, how to choose the interactive term?

To provide the option to the analyst, an argument `activeGeomLayers` is introduced to identify which layer to be interactive. A `ggplot` object is shown as follows

```
> php <- ggplot(data = data.frame(x = rnorm(100), y = rep(0, 100)),
+               mapping = aes(x = x)) +
```

[1]The tables show the matched visual structure between `loon` and `ggplot2`. However, in the current version 1.3.0, some layers are not yet perfectly matched (e.g., layer `l_layer_smooth` is mapped to a `geom_path` layer for the fitted line and a `geom_polygon` layer for the confidence interval).

```
+           geom_histogram(fill = "pink") +
+           geom_point(mapping = aes(y = y), alpha = 0.5,
+                      size = 4, color = "skyblue")
```

The data is generated from a standard normal distribution. Here `php` is a `ggplot` object with two geometric layers: a histogram layer and a point layer with the points along the x axis. To transform this `ggplot` object to a `loon` widget, an analyst needs to choose which layer should be the model layer.

The `geom` position is used to set the `activeGeomLayers` argument. For example, Figure 3.7 shows three possible transformations from `php` to `loon`. To have neither layers

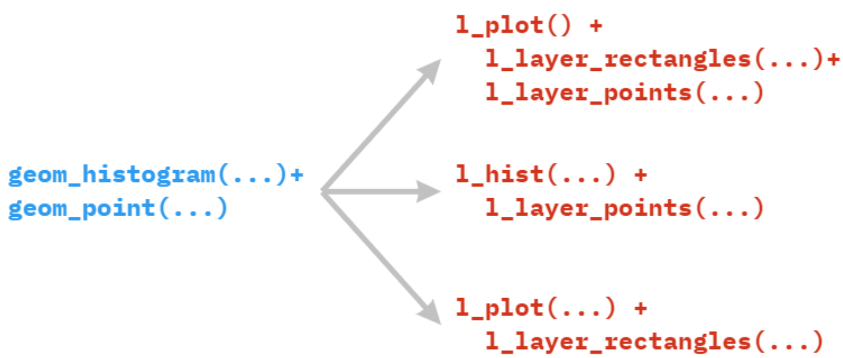


Figure 3.7: Which layer should be interactive, neither, bins or points?

interactive, `activeGeomLayers` is set 0, as in

```
> ggplot2loon(php, activeGeomLayers = 0)
```

Then an empty `l_plot` is returned with two static layers, one for the static points and one for the rectangles of the histogram (the data structure is shown in Figure 3.7 topmost and the graphic is shown in the leftmost plot of Figure 3.8).

When `activeGeomLayers = 1`, as in:

```
> ggplot2loon(php, activeGeomLayers = 1)
```

an `l_hist` widget is created with interactive bins and a static points layer is added (the data structure is shown in Figure 3.7 middle and the graphic is shown in the middle chart of Figure 3.8).

Alternatively, when `activeGeomLayers = 2`, as in

```
> ggplot2loon(php, activeGeomLayers = 2)
```

then an `l_plot` widget is returned with interactive points and bins layered as static rectangles (the data structure is shown in Figure 3.7 bottom and the graphic is shown as the rightmost chart in Figure 3.8).

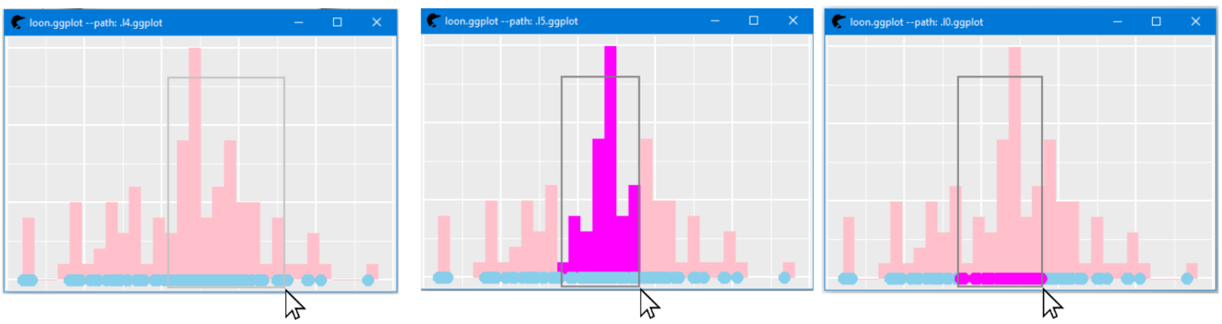


Figure 3.8: The same static graphics but different interactive motions

If `loon` could accommodate multiple interactive layers, it would be natural to specify them by setting `activeGeomLayers` to a vector of geom positions (e.g., `activeGeomLayers = c(1, 2)` would have bins and points become interactive simultaneously, or, perhaps, switchable). Unfortunately, `loon` does not support multiple interactive layers for now; therefore, vector valued `activeGeomLayers` is not yet allowed in the transformation.

Imagine the transformation to be a mapping, setting argument `activeGeomLayers` as 0 is equivalent to only having the visual display mapped. In order to map the visual structure instead, one has to set the argument `activeGeomLayers` to the `geom` position (as it appeared in the order it was added to the `ggplot`) of either a `geom_point` or `geom_histogram` layer. Suppose the `geom` layer does not have a counterpart in `loon`, for example a boxplot as in

```
> ggplot(iris, aes(Species, Sepal.Length)) + geom_boxplot()
```

then, regardless the values of argument `activeGeomLayers`, only the display is mapped as if `activeGeomLayers = 0`.

The package `ggmulti` extends the package `ggplot2` in non-primitive point glyphs. When a `ggplot` object has a non-primitive point glyph layer (e.g., `geom_image_glyph`, `geom_serialaxes_glyph`, `geom_polygon_glyph`), to transform it to a `loon` widget, one can choose either to map the display only (each point is static) or to map the visual structure (each point is interactive). For example,

```
> ps <- ggplot(data = iris,
+              mapping = aes(Sepal.Length, Sepal.Width,
+                            color = Species)) +
+   geom_serialaxes_glyph(serialaxes.data = iris[, -5],
+                         axes.layout = "radial")
```

If `activeGeomLayers = 1` (default), as in

```
> ggplot2loon(ps, activeGeomLayers = 1) # default setting
```

then an `l_plot` widget is returned with interactive non-primitive glyph points (as in the left plot of Figure 3.9). If `activeGeomLayers = 0`, as in

```
> ggplot2loon(ps, activeGeomLayers = 0L)
```

then an empty `l_plot` is returned with one static layer (as in the right plot of Figure 3.9).

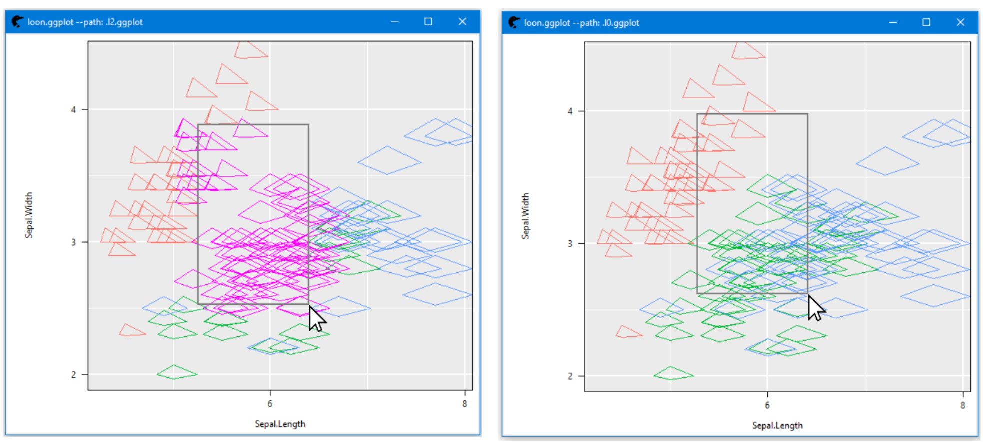


Figure 3.9: Interactive non-primitive glyphs and static non-primitive glyphs.

Coordinate Systems

A coordinate system combines two positions to produce a 2D projection on a plot. In general, there are two types of coordinate systems, linear coordinate systems (e.g., Cartesian coordinate system) and non-linear coordinate systems (e.g., polar coordinate system, serialaxes coordinate system).

A linear coordinate system is also known as the Cartesian coordinate system. Table 3.1 shows the linear transformations for both graphical systems. Suppose one modifies the `ggplot` Cartesian system, such as setting limits, changing ratios or flipping axes, in the transformation, the corresponding states upon these changes will be set in `loon`.

Table 3.1: Linear Coordinate Systems

	ggplot	loon
Zooming	`coord_cartesian()`	Direct Manipulation: scroll a mouse; Command-line: set states `zoomX` and `zoomY`
Display Ratio	`coord_fixed()`	Direct Manipulation: scroll a mouse with holding <`shift`> (vertically) or <`Alt`>/<`cmd`> (horizontally); Command-line: set states `deltaX` and `deltaY`
Swap Axes	`coord_flip()`	Direct Manipulation: click `swap` on `loon` inspector; Command-line: set state `swapAxes` as `TRUE`

Unlike the linear coordinate system, a non-linear coordinate system (as shown in Table 3.2) does not preserve the shape of geometric objects. The `ggplot2` package focuses on the transformations from the Cartesian coordinate system to other coordinate systems (e.g., in the polar coordinate system, a bar chart can be turned into a pie chart; in the map coordinate system, a portion of the earth, approximately spherical, is mapped onto a flat 2D plane). Unfortunately, these coordinate systems are not supported in the package `loon`. In other words, suppose a `ggplot` object is embedded in one of *polar*, *map* or *trans* coordinate systems, the visual structure cannot be mapped to a `loon` plot. Sometimes, even the visual display cannot be mapped.

Figure 3.10 (a) shows a `loon` bar plot transformed from a `ggplot` bar chart. The bins can be highlighted as we click or brush. Figure 3.10 (b) shows a `loon` pie chart

Table 3.2: Non-linear Coordinate Systems

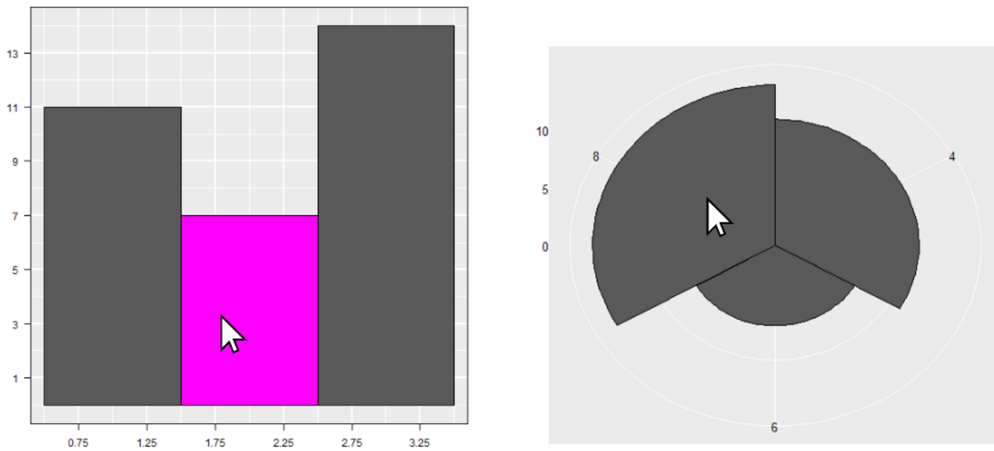
loon	ggplot
`l_serialaxes()`	`ggmulti::coord_serialaxes()`
	`coord_polar()`
	`coord_map()`
	`coord_trans()`

transformed from a `ggplot` pie chart. Although, it has the same appearance, it is static. In this example, only the visual display is mapped.

Figure 3.11 (a) is a `ggplot` object with orthographic projection (looking down at North pole). When it is transformed to a `loon` widget, the orthographic projection will be omitted, as shown in Figure 3.11 (b). In this example, neither the visual display nor the visual structure gets mapped.

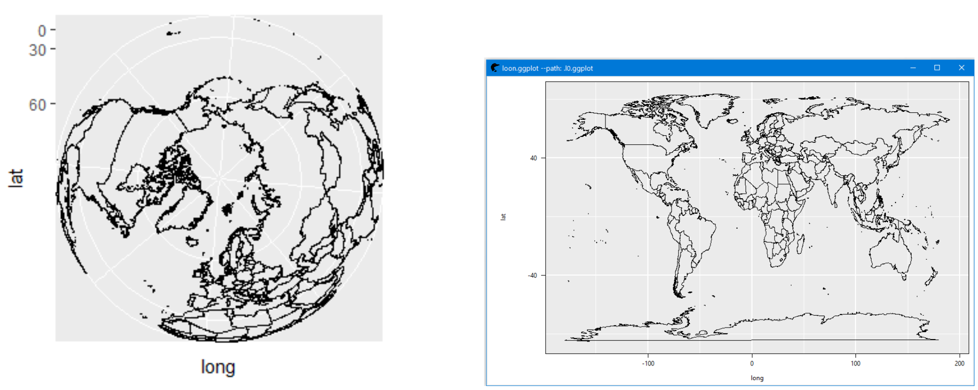
When a `ggplot` object, in the serialaxes coordinate system (e.g., see Figure 3.2; by the package `ggmulti`), is transformed to a `loon` plot and the object has geometric layers (e.g., histograms, density), as shown in Figure 3.3, two scenarios may happen:

- to map the visual structure – an `l_serialaxes` widget will be returned but no geometric layers will be displayed in the `loon` plot (as shown in Figure 3.12 left), as the primitive layer visuals (e.g., polygons) are not yet available in an `l_serialaxes` widget;
- to map the visual display – an `l_plot` will be returned with stacked rectangles and lines (as shown in Figure 3.12 right).



(a) (b)

Figure 3.10: Map the visual display only.



(a) (b)

Figure 3.11: The mapping is incomplete.

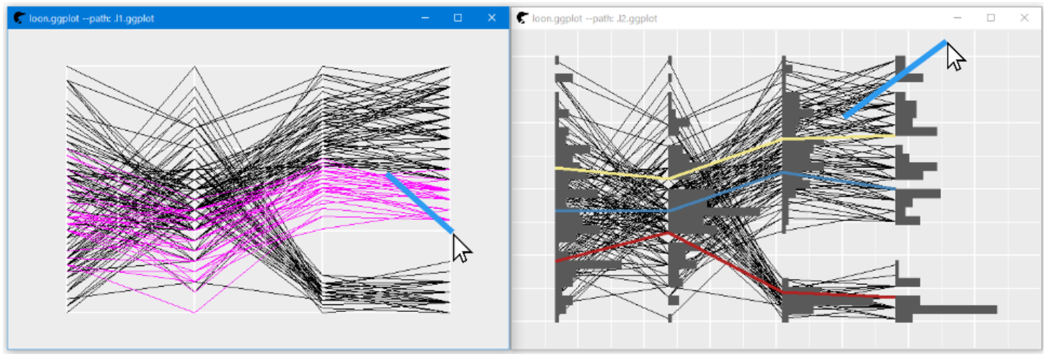


Figure 3.12: To transform a `ggplot` object (with statistical layers) in the serialaxes coordinate system to a `loon` widget: if it is transformed to an interactive `l_serialaxes` widget, the layers are missing; if it is transformed to an `l_plot` widget, all layers are preserved but static.

Facets

The package `ggplot2` provides two types of faceting, `facet_grid()` and `facet_wrap()`. The function `facet_grid()` (a 2D facet) forms a matrix of panels defined by row and column faceting variables, while the function `facet_wrap()` wraps a 1D sequence of panels into 2D.

As we transform a `ggplot` object with multiple facets to a `loon` widget, Figure 3.13 shows two possible designs: 1. the package `ggplot2` provides the function `ggBuild()` to split the original data set into multiple subsets. In the transformation, we pass these subsets to `loon` and then each `loon` widget is created by an individual subset. In the end, all widgets are packed as an `l_facet` object; 2. we pass the original data set into `loon`. Then, construct an `l_facet` object in `loon` via setting the states `by` and `layout` to split the data and draw facets.

So far, the first design has been used. An explicit benefit is that the function `ggBuild()` is maintained by the package `ggplot2`. If the logic or application programming interface is updated for functions `facet_grid()` and `facet_wrap()`, `ggBuild()` could still work. However, if the 2nd design is used, `loon.ggplot` developers should always track the updated news of `ggplot2` which may require much more efforts for the purpose of maintenance.

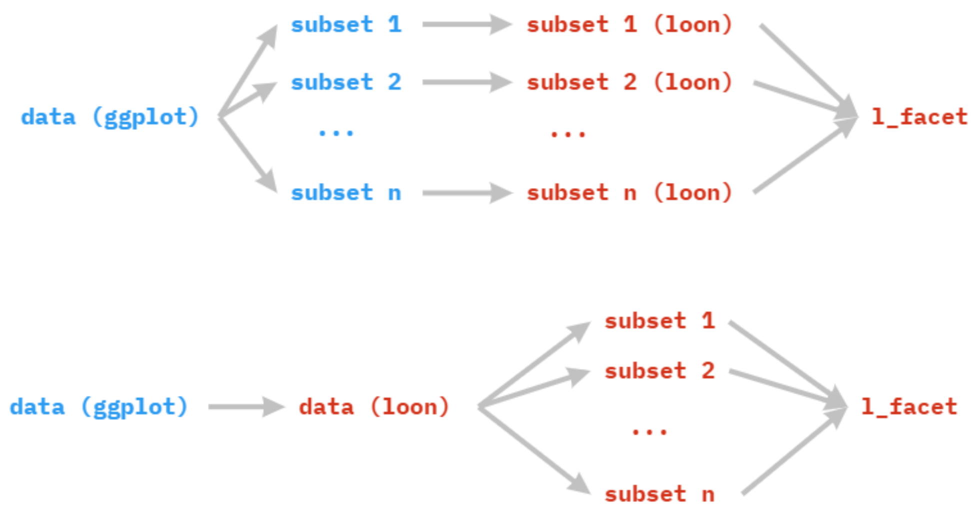


Figure 3.13: Two possible designs to turn a ggplot object with multiple facets to a loon object. The sky blue means the data is still maintained by ggplot object and the firebrick red represents the data has been passed into a loon widget

3.3.3 A Grammar of Interactive Graphics

Function `l_ggplot()`

Typically, a loon widget is constructed by calling functions with prefix `l_**` (e.g., `l_plot`, `l_histogram`) and adding layers by calling functions `l_layer_**(widget, ...)`. In the package loon.ggplot, the function `l_ggplot()` is built, providing the ggplot syntax to create a loon plot. Users who prefer the design of constructing a ggplot object but also want to keep dynamic interactions with their data can now replace `ggplot()` with `l_ggplot()`, as in

```
> lp <- l_ggplot(data = mtcars,
+                mapping = aes(x = mpg, y = hp)) +
+         geom_point() +
+         geom_smooth()
```

`lp` is an `l_ggplot` object with duality. When users type `print(lp)` (or only `lp`) in the console, a loon widget is then created with interactive points and a static smooth line. When users type `plot(lp)`, a ggplot object will be created (imagine `print()` and `plot()`

are two bridges: in the `print()` function, a `ggplot` object is transformed to a `loon` widget; while in the `plot()` function, a `ggplot` object is transformed to a `grid` object).

`lp` is a variable. All `ggplot` components can be added to `lp`. The function `facet_wrap()`, for example, can be added to `lp` so that the data can be split into multiple panels and each panel represents a subset of transmission (0 = automatic, 1 = manual), as in (shown in Figure 3.14)

```
> lp1 <- lp + facet_wrap(~am)
> lp1
```

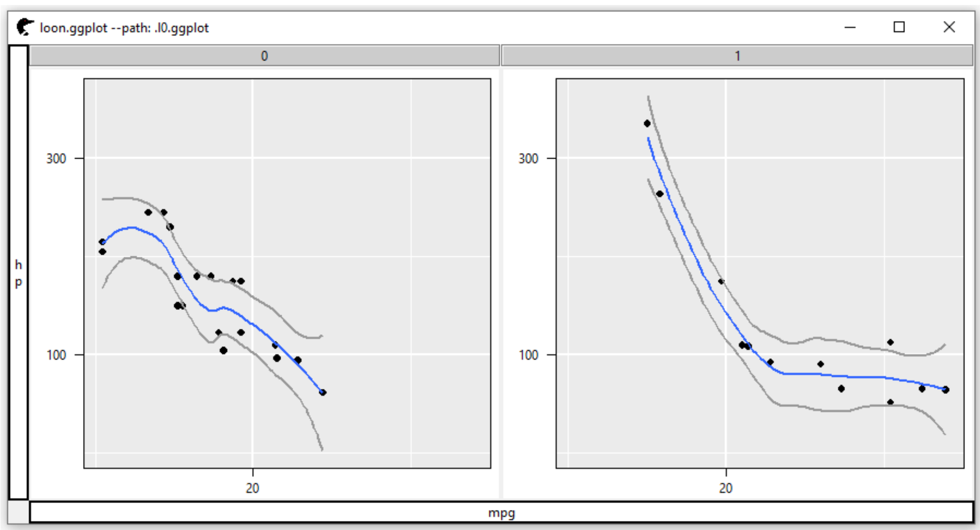


Figure 3.14: A `loon` widget created by `ggplot2` syntax

Interactive Components

Five interactive components (functions) are available to control linking, selecting, activation, querying and scaling (or zooming) respectively, as shown in Table 3.3. Most states of these components (e.g., `linkingGroup`, `selectBy`) have been discussed in Chapter 2, except the state `layerId`, `activeGeomLayers` (see Subsection 3.3.2) and `scaleToFun`.

The `zoom` component is adopted to configure a plot region. Unlike `coord_cartesian()` whose main purpose is to set the limit of the view, it uses `layerId` to scale the plot region to an individual geometric layer.

The `scaleToFun` state is a function to determine how to scale the region. Available ones are `l_scaleto_plot()` (scale to a model layer), `l_scaleto_active()` (scale to all active elements), `l_scaleto_selected()` (scale to selected elements) and `l_scaleto_layer()` (scale to any dependent layers). The following code shows how to zoom in to focus on economical cars whose fuel consumption is greater than 20 mile per gallon, as in (shown in Figure 3.15)

```
> lp_higlighted <- lp +
+       selection(selected = ~mpg > 20) +
+       linking(linkingGroup = "mtcars") +
+       zoom(layerId = 1,
+            scaleToFun = loon::l_scaleto_selected)
> lp_higlighted
```

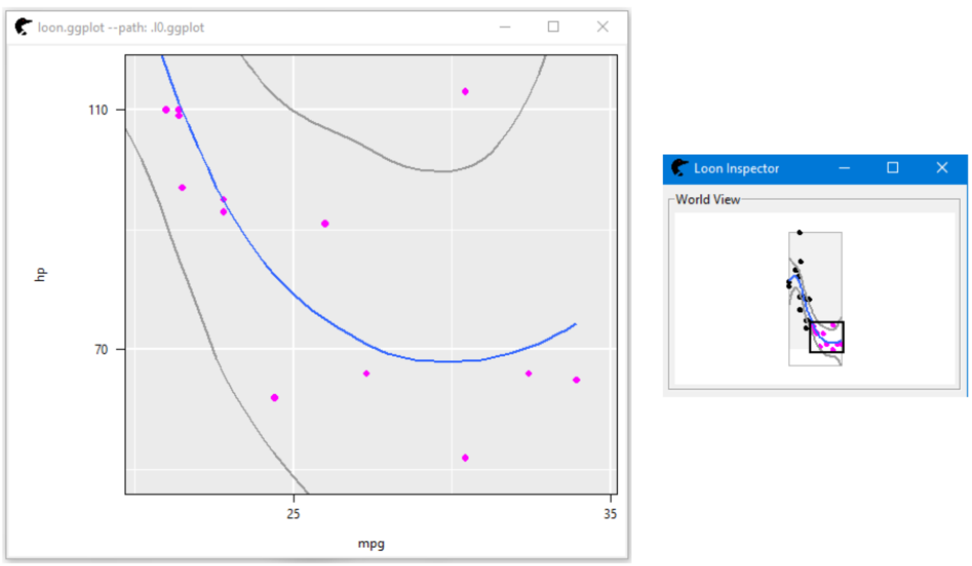


Figure 3.15: The points with high mpg are highlighted. The region is scaled to the highlighted points.

The static `ggplot` can be turned into an interactive `loon` plot (even without calling the function `l_ggplot()`) by adding any of these interactive components. For example,

```
> gp <- ggplot(data = mtcars,
+              mapping = aes(x = mpg, y = hp)) +
+        geom_point() +
```

Table 3.3: Interactivity Components

Interactivity	Description	Subfunction	States
Zoom	Region Modification	`zoom()`	`layerId`, `scaleToFun`
Linking	Linking several plots to discover the pattern of interest	`linking()`	`linkingGroup`, `linkingKey`, `linkedStates`, `sync`
Select	Highlight the subset of interest	`selection()`	`selected`, `selectBy`, `selectionLogic`
Active	Determine which points appear or which layer is the interactive layer	`active()`	`active` `activeGeomLayers`
Query	Query in interactive graphics	`hover()`	`itemLabel`, `showItemLabels`

```
+          geom_smooth()
```

`gp` is a `ggplot` object. One can add any interactive components (e.g., `selection()`) to turn the static plot into an interactive `loon` plot, such as

```
> gp + selection()
```

With these interactive components, a grammar of interactive graphics is implemented.

Logic of Implementation

The logic of implementation is that a `ggplot` object is a variable at construction time and is rendered as a graphic at printing time. In the function `l_ggplot()`, a new class called `l_ggplot` gets assigned to the `ggplot` object, then, a new object − `l_ggplot` object is created with the class attribute `[l_ggplot, gg, ggplot]`. It has an identical data structure with a `ggplot` object, expect the class attribute (the class attribute of a `ggplot` object is `[gg, ggplot]`)

In R, when a generic function `fun()` (an S3 method) is applied to an object with class `[first, second]`, a function called `fun.first()` will be looked for first. If it is found, then this function will be applied to the object. If not, then a function called `fun.second()` will be tried. The `print()` is a generic function. Therefore, at printing time, `print.l_ggplot()` will be executed (rather `print.ggplot()`). Within `print.l_ggplot()`, the `ggplot` object is transformed into a `loon` widget, as in

```
> print.l_ggplot <- function(x, ...) {
+   p <- ggplot2loon(x, ...)
+   invisible(p)
+ }
```

where `x` is an `l_ggplot` object.

The `plot()` function is a generic function as well. Since there is no function called `plot.l_ggplot()`, when we execute `plot(x)`, the function `plot.ggplot()` will be executed and a static `ggplot` (based on `grid` graphics) will be displayed.

Any interactive components (see Table 3.3) is an `Interactivity` object (a prototype object, `ggproto`). Once it is added to a `ggplot` object or an `l_ggplot` object, the function `ggplot_add()` will be activated. As it is generic, `ggplot_add.Interactivity()` is created to ensure the objects in the interactive components are dispatched to the correct place. In this function, if the input object is a `ggplot` object, the class `l_ggplot` will be added to force it to be an `l_ggplot` object, as in

```
> ggplot_add.Interactivity <- function(object, plot, object_name) {
+   if(!is.l_ggplot(plot)) {
+     class(plot) <- c("l_ggplot", class(plot))
+  }
+  ...
+ }
```

Then an l_ggplot object is returned.

3.4 loon to ggplot2

The package loon.ggplot is a two-way bridge, in that a loon plot can be transformed to a ggplot object as well. To set up a bridge from loon to ggplot2, high-level elements will be mapped to high-level elements and low-level elements to low-level elements (see Table C.1, C.2, C.3, C.4, from left to right).

Unfortunately, not all high-level elements in loon can be mapped in ggplot2 (e.g., serialaxes coordinates, non-primitive glyphs) without first extending the graphical system ggplot2 to include these elements. The package ggmulti is created to provide this extension to ggplot2. Table 3.4 shows the mappings of high dimensional graphics from loon to ggplot2 via ggmulti.

The ggplot2 package provides high-level functionality such as whether to treat data as variables or as constants, whether or not to display a more intuitive legend and et cetera. These require users to make choices to better tailor for their own specific problems. In the following subsection, these choices will be discussed, as well as corresponding arguments to make these decisions.

Note that the conversions of aesthetic attributes will not be discussed in this section. As ggplot2 is built on top of grid, they have similar settings of aesthetics and in Chapter 4, transformations of aesthetics, from loon to grid will be discussed, in details (see Section 4.2).

3.4.1 Arguments

Argument asAes

There are two ways to construct geometric visuals in ggplot2: mapping aesthetic attributes to variables (set in function aes()) or setting them as constants. For example,

Table 3.4: Extensions by ggmulti

loon	ggplot
`l_serialaxes()`	`ggmulti::coord_serialaxes()`
`l_glyph_add_serialaxes()`	`ggmulti::geom_serialaxes_glyph()`
`l_glyph_add_image()`	`ggmulti::geom_image_glyph()`
`l_glyph_add_polygon()`	`ggmulti::geom_polygon_glyph()`

```
> data <- data.frame(x = seq(4), y = seq(4),
+                    color = c(rep("red", 2),
+                              rep("green", 2)))
> p1 <- ggplot(data = data,
+              mapping = aes(x = x, y = y)) +
+        geom_point(color = data$color)
> p1 # as constants
> p2 <- ggplot(data = data,
+          mapping = aes(x = x, y = y,
+                        color = color)) +
+        geom_point() +
+        scale_color_manual(
+          values = c("green", "red")
+        )
> p2 # as variables
```

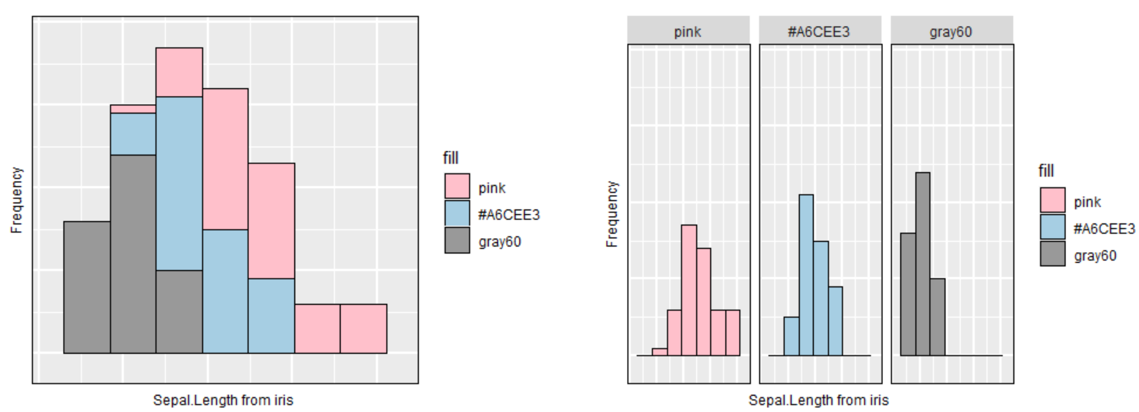
Both `p1` and `p2` show two red points and two green points; nevertheless, the construction is very different. For `p1`, color is treated as a general attribute of points, like `loonGrob`. For `p2`, it maps the color aesthetic attribute to a variable named "color" which has two categories "red" and "green" (they are just two specified factors, not colors!). If `scale_color_manual()` is not applied, the color of each category will use the default ggplot2 color scales where category "red" will be colored #00BFC4 (turquoise) and category "green" will be colored #F8766D (selmon).

In order to accommodate both ways to construct geometric visuals for the transformation, an argument asAes is introduced in the function loon2ggplot(). When it is set as TRUE, aesthetic attributes color, fill, and size, will be treated as variables; otherwise constants. It is beneficial to take an aesthetic attribute as a variable, especially for the further analysis, for example,

```
> h <- l_hist(iris, color = iris$Species)
> gh <- loon2ggplot(h, asAes = TRUE)
> gh
```

Here, a ggplot histogram (geom_histogram()) is mapped, as shown in Figure 3.16 (a). Other components for statistical analysis, such as facet_wrap(), could be added to gh, as shown in Figure 3.16 (b).

```
> gh + facet_wrap(~fill)
```



(a) (b)

Figure 3.16: To transform a loon l_hist widget to a ggplot histogram object. If the asAes is set as TRUE, the ggplot histogram is constructed by geom_histogram; else by geom_rect.

However, setting asAes = TRUE in a scatterplot may cause some unexpected problems. Currently, ggplot2 legend has a bug that filled glyphs are always black if the shape is set as a vector, as in (see Figure 3.17)

```
> d <- data.frame(x = c(1,2),
+                  y = c(1,2),
+                  color = c("red", "blue"),
+                  fill=c("black", "gray"))
> ggplot(d, aes(x = x, y = y)) +
+     geom_point(aes(fill=fill, color = color),
+                size=4,
+                shape = c(19, 21)) +
+     scale_fill_manual(values = c("black", "gray")) +
+     scale_color_manual(values = c("red" = "red",
+                                   "blue" = "blue"))
```

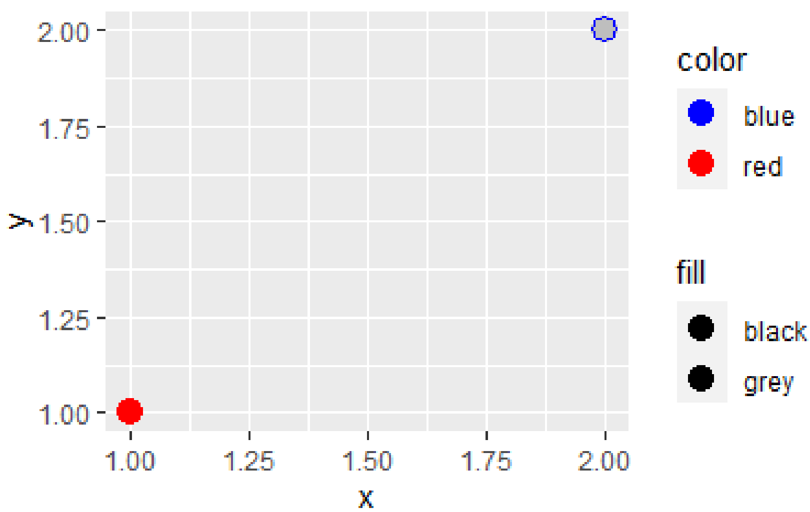


Figure 3.17: A bug in the legend for filled shapes. In the fill legend, the colors should be black and gray.

If all points are with borders (e.g., all pchs are in [21, 22, ..., 25]) or without borders (e.g., all pchs are in [0, 1, ..., 20]), setting `asAes = TRUE` (treating `color` or `fill` as a variable) is fine; else, points with and without borders are mixed so that the legend will not be displayed as expected (e.g., see Figure 3.17).

If the `asAes` is set as `FALSE`, an aesthetic attribute will be treated as a constant and only the display is mapped. The `ggplot2` graphics will be constructed by the primitive `geom` layers (e.g., `geom_rectangle`) and no statistical transformations are required. Thus, the plot rendering speed is faster. Continue with the previous histogram example, Table 3.5 shows the evaluation time via micro benchmark.

Table 3.5: Time Consumption

expression \ summary (ms)	min	lq	mean	median	up	max	eval
`loon.ggplot(h, asAes = TRUE)`	195	213	230	222	238	543	100
`loon.ggplot(h, asAes = FALSE)`	83	87	95	90	98	303	100

Obviously, if the `asAes` is set `FALSE`, the construction speed is twice faster than that as `TRUE`. Note that, there are only 150 elements in data set `iris`. If the number increases (e.g., over 2,000), the difference would be non-negligible.

Argument `selectedOnTop`

In `loon`, the highlighted points are always displayed at the front. In the transformation, when the argument `selectedOnTop` is set as `TRUE` (default), the highlighted points are drawn at the front of the non-highlighted points. In this way, the order of the data is altered (highlighted points are drawn last). When the argument `selectedOnTop` is set as `FALSE`, the order of the data remains unchanged. However, in this way, when points are overlapped, highlighted points may be drawn at the back.

For example, in Figure 3.18 left, the order of the data is changed where the highlighted point is drawn last (at the front). While, in the right figure, all points are displayed in order and the highlighted point are displayed at the back.

Argument `showNearestColor`

When we transform a `loon` widget to a `ggplot` object, a 12-digit hexadecimal color is turned into a 6-digit hexadecimal color. Suppose `color` and `fill` are set in `aes()` (as variables), a legend will be displayed whose labels are color names or hexadecimal codes.

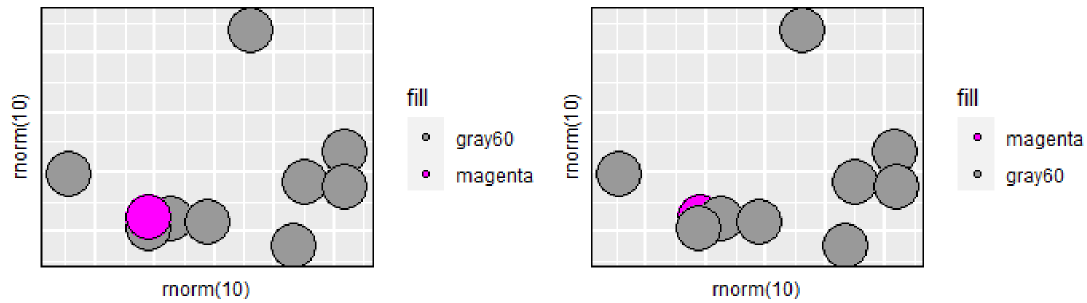


Figure 3.18: Whether to force the highlighted points to be displayed at the front.

For a majority of users, a color name (e.g., gray) is more intuitive than a hex code (e.g., #808080). To convert a hex code to a real color name, we first convert all built-in colors (by the function `colors()`; approximate 657 in total) and the hex code (to be transformed) to RGB (red/green/blue) values (e.g., gray → [50, 50, 50]). Then, the closest (Euclidean distance) built-in color can be determined using the RGB vector value. If the `showNearestColor` is set as `FALSE` (default), whenever the minimum distance is zero, the hex code will be replaced by the color name; otherwise (set as `TRUE`), the hex code will be replaced by the nearest R color (it is "approximate").

For example, in Figure 3.16, the hex code #A6CEE3 does not have an exact color name (minimum Euclidean distance between its RGB value and built-in colors' RGB vector values is 12.2474, not 0). If `showNearestColor` is set as `FALSE`, the color name is still #A6CEE3; else (`TRUE`), the closest color name of #A6CEE3 is lightblue2. Consequently, in the legend, #A6CEE3 would be replaced by lightblue2. In contrast, the hex code #FFC0CB has an exact color name (minimum Euclidean distance is 0). Whatever the `showNearestColor` is set, it is converted to the color name pink.

It is very helpful to set it as `TRUE` when analysts are satisfied with the "approximation" and need a neat color legend. However, we should be careful when colors are too close to be distinguished. For example, 20 colors are created via the following code with the same hue and chroma, but slightly different value.

```
> redGradient <- matrix(hcl(0, 80, seq(49, 68, 1)),
+                       nrow=4, ncol=5, byrow = TRUE)
> p <- l_plot(x = rep(1:5, each = 4), y = rep(1:4, 5),
+             color = redGradient,
+             glyph = "square", size = 100)
> loon2ggplot(p, showNearestColor = FALSE) +
```

```
+   # legend is displayed in two columns
+   guides(color = guide_legend(ncol=2))
> loon2ggplot(p, showNearestColor = TRUE)
```

The argument `showNearestColor` is set as `FALSE` for the left Figure 3.19, and `TRUE` for the right. When it is `TRUE`, the "approximate" colors are applied so that 20 different colors are shrunk to 4.

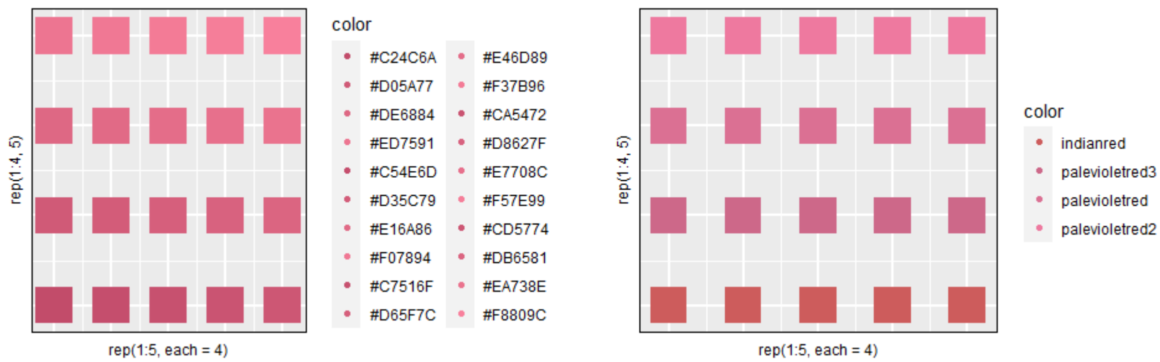


Figure 3.19: When colors are similar (similar hue, chroma or luminance), the "approximate" colors (setting `showNearestColor` as `TRUE`) may shrink the number of unique colors.

3.4.2 Compound Plot

To transform a `loon` `l_compound` object, the package `patchwork` is applied, providing a `patchwork` data structure that can arrange multiple `ggplot` objects in a single panel. The main reason to use the `patchwork` rather than the `gridExtra` (Auguie, 2017) (see Subsection 4.3.3) is it extends the `ggplot2` data structure to multiple displays.

The symbol "+" or "/" could be used to connect several `ggplot`s. For example, suppose `p1` and `p2` are two `ggplot` objects, `p1` plus `p2`, as in,

```
> library(patchwork)
> p1 + p2
```

two plots would be placed side by side; `p1` divided by `p2`, as in,

```
> p1 / p2
```

two plots would be stacked (`p1` is on top of `p2`). See Pedersen, 2020b for more details.

3.5 Summary

The `loon.ggplot` package provides a two way bridge, through which interactive `loon` plots can be transformed to static `ggplot`s, and vice versa. Both `ggplot2` and `loon` graphical systems primarily provide graphics at a high level of graphical abstraction. Consequently, in the `loon.ggplot` bridge, the priority is to map visual structures, for example, a `geom_histogram` data structure is mapped to an `l_hist` data structure.

Not each visual structure can be mapped from one to the other. For example, layer `geom_histogram` embedded in a polar coordinate system (i.e., a pie chart) cannot be mapped to an interactive `loon` pie plot; a `loon` scatterplot with polygon glyphs cannot be mapped to a corresponding structure in `ggplot2`. In order to establish successful mappings, we extend `ggplot2` using the package `ggmulti`. A non-primitive glyph in `loon` can then be mapped to a `geom_**_glyph` structure in `ggplot2`. Nevertheless, in cases when extensions are not created, visual displays are used for mapping purpose. For instance, to transform a `geom_histogram` object embedded in a polar coordinate system from `ggplot2` to `loon`, we break a `ggplot2` pie chart down into polygon visuals and use the primitive graphical data structure `l_layer_polygon` to build a static plot in `loon`.

Furthermore, `loon.ggplot` helps extend a grammar of graphics to a grammar of interactive graphics. Interactive components (e.g., `selection()`) can be added onto a `ggplot` object; therefore, the `ggplot` object is transformed to an interactive `loon` widget.

3.5.1 Lessons

An important lesson from building this bridge is: even though matching the level of abstraction is the most natural transformation method, it is possible that not every high-level element in $\mathcal{G}$ has a mapping in $\mathcal{K}$ (visual structure cannot be mapped), in which case, extending $\mathcal{K}$ should be considered first. If the extension is hard to build, we break this high-level element down to low-level ones and then map visual displays.

3.5.2 Limitations

Mappings are useful not only in designing a bridge, but also in reviewing and assessing a bridge. When assessing the bridge `loon.ggplot`, limitations in design are detected and addressed.

First of all, when the visual display is mapped but the visual structure is not, it may become difficult to follow from this bridge to another (e.g., `loon` with facets → `ggplot` → `plotly`). For instance, when a `loon` widget with multiple facets is transformed to a `ggplot` object, the package `patchwork` was applied to return a `patchwork` object. A better approach would be to use the function `facet_wrap()`, or `facet_grid()`, in `ggplot2` so that the visual structure also gets mapped.

Secondly, rendering a `ggplot` object created by `ggplot2` is faster than rendering a `ggplot` object created via the bridge (i.e., after already mapping the `loon` plot to a `ggplot` via `loon.ggplot`). Tables 3.6 and 3.7 show the rendering time benchmark (Mersmann, 2019) of a scatterplot with 1,000, 10,000 and 100,000 points first rendered simply by `ggplot2` and then by `loon.ggplot`. The rendering times are close when the number of points is small (e.g., less than 10,000), but, as the number increases to 100,000, the speed of drawing a `ggplot` object directly is much faster than drawing a `ggplot` object from the bridge. In addition, much less memory is consumed by a standard `ggplot` object

Table 3.6: `ggplot2` Rendering Time

summary (sec) / number of points	min	lq	mean	median	up	max	eval
`1,000`	0.136	0.192	0.210	0.215	0.234	0.247	100
`10,000`	0.572	0.651	0.664	0.675	0.689	0.706	100
`100,000`	5.046	5.324	5.418	5.412	5.566	5.623	100

than that of one obtained from `loon.ggplot`, as shown in Table 3.8. The principal reason seems to be that `loon` provides a vector of data values (e.g., color, size, etc.), even when all values are identical; `ggplot2` does not.

Thirdly, when transforming a `loon` scatterplot with a combination of primitive glyphs and non-primitive glyphs, or multiple non-primitive glyphs, a single layer in `loon` will be mapped to more than one layer in `ggplot2`, which destroys the layer data structure. For example, transforming a `loon` plot with two points, one being filled-circle and the other a

Table 3.7: `ggplot2` Rendering Time via `loon.ggplot`

number of points \ summary (sec)	min	lq	mean	median	up	max	eval
`1,000`	0.138	0.168	0.184	0.187	0.204	0.213	100
`10,000`	0.772	0.842	0.844	0.848	0.863	0.868	100
`100,000`	7.057	7.320	7.364	7.347	7.430	7.543	100

Table 3.8: Memory Consumption (MB)

Number of points	ggplot2 (by bridge)	ggplot2
1,000	0.261	0.022
10,000	2.389	0.159
100,000	24.362	1.532

text glyph, the returned ggplot object will have two layers, one geom_point layer for the filled-circle and one geom_text layer for the text glyph. With many more points, there will be a different ggplot2 layer for each type of glyph.

Furthermore, in ggplot2, only a point layer and a histogram layer can be made interactive in the current version of loon (i.e., map visual structure). Other geometric objects, such as rectangles, polygons and lines, are currently only static (i.e., map visual display). Without extending loon, the loon.ggplot bridge must specify which geoms in the ggplot are to be made interactive in the loon plot.

The animint2 package suggests how interactivity might be specified in any layer. For example, the code below shows how to make a static segment layer selectable.

```
> geom_segment(
+   ...
+   showSelected = ***,
+   clickSelects = ***)
```

The tcltk design of loon would have to be refactored to accommodate similar specifications for different layers. This would be a major extension of loon (see Section 7.4).

3.5.3 Further Work

In the future, we would like to: firstly, redesign the transformation from an l_facet object in loon to a ggplot object by using the function facet_wrap() or facet_grid(), in order to map the visual structure; secondly, optimize the returned ggplot data structure to have a faster rendering speed and a less memory consumption.

Chapter 4

LoonGrob

The loon package is mainly designed for interactive data exploration. Upon exploring the events of interest, we need a tool to turn interactive plots into static for the purpose of storage (or publication). Taking screenshot is an option, however, the quality of a screenshot of these graphics is not particularly good. In addition, the details of graphical elements are not programmatically editable. One solution is to transform the loon plot to a static graphical system plot, not only to preserve the high quality of the loon plot, but also to allow further modifications in command-line.

The package loon.ggplot can turn a loon widget into a static ggplot object. However, ggplot2 is not a pre-installed package. Additionally, it has many other dependencies (not maintained by R core team). Depending upon ggplot2 to return a static loon plot is not as stable as depending upon the package grid (plotting engine of ggplot2) (Murrell, 2018), which is pre-installed in R, and has been maintained by R core team now.

This chapter begins with an overview of the grid graphics package. Then, we discuss about the transformation of plotting states, such as color, point size and point shape from loon to grid. As the abstraction levels of loon and grid do not perfectly match, how to map the graphical elements between these two packages is discussed. This chapter closes with a summary of the bridge loonGrob. Lessons learned from building this bridge, its limitations and further work required are shared in details.

4.1 Introduction of `grid`

The graphical system `grid`, grew out of Murrell (1998)'s PhD thesis, provides functionality at a relatively low level of graphical abstraction. Rather than creating plots for data analysis directly, it is more commonly used by other graphical packages (e.g., `ggplot2`, `lattice`) that are built on top of `grid` (Murrell, 2018).

In `grid`, any geometric layers (e.g., lines, points) have two functions: one is `**Grob()` which produces a **gr**aphical **ob**ject, named `grob`; the other is `grid.**()` which draws the actual graphical output. For example, the output of `lineGrob(x,y)` is an object storing all coordinates (x and y) and other essential plotting states (e.g., line color, line dash, the name, etc.). It does not actually draw a line. If one wants to display it, call `grid.line(x,y)` (alternatively, `grid.draw(lineGrob(x,y))`).

A `grob` is composed of four major components: features, graphical parameters, a viewport and a name.

1. features: named slots describing important features of a `grob` (e.g., x, y coordinates).

2. `gpar` (**g**raphical **par**ameters): a list of graphical parameter settings used to control the output appearance (e.g., color).

3. `vp` (viewports): describe rectangular regions on a graphics device and define a number of coordinate systems within those regions.

4. `name`: a character identifier for a `grob`.

Besides, `grid` provides a data structure `gTree` that can have other `grob`s or even nested `gTree`s as children. When a `gTree` is drawn (by `grid.draw()`), all of its children are displayed.

4.2 Conversion of the Aesthetic Attributes

To transform a `loon` widget into a `grid` object, all aesthetic attributes (e.g., color, point size, point shape) should be mapped accurately (or as accurately as possible).

4.2.1 Color

Color specifications are normalized to a 12 digit hexadecimal color representation in `tcltk` while R uses 6 (or 8) hexadecimal colors. During transformation, a 12 digit hexadecimal color is converted to a 6 digit hexadecimal color, for example,

```
> "#FFFF00000000" is a 12 digit hex code of red
> as_hex6color("#FFFF00000000")
[1] "#FF0000" # is a 6 digit hex code of red
```

4.2.2 Shape

The graphical system `loon` provides four different primitive point shapes: circle, triangle, rectangle and diamond. Each could be either empty ○, solid ● or filled ◉. The `grid` graphical system provides richer point symbols (at least twenty five). Table 4.1 shows the conversion of the point glyph from `loon` to `grid`.

The package `loon` also supports non-primitive glyphs such as an image glyph, a polygon glyph, a point-range glyph, a text glyph and a serialaxes glyph. As `grid` does not offer such glyphs, they are mapped to the primitive geometrical visuals (e.g., `polygonGrob`, `polylineGrob`, etc.), as shown in Figure 4.1. The arrow represents a mapping. For example, a point glyph and a text glyph are mapped to a `pointsGrob` and a `textGrob` accordingly. The curly brace means options. For example, a polygon glyph can be mapped to a `polylineGrob` data structure or a `polygonGrob` data structure, depending upon it is filled or not. The solid line means *contain* that an image glyph is mapped to a `gTree` data structure containing a `rectGrob` as the background (for selection) and a `rasterGrob` for image drawing.

Table 4.1: Mapping Glyph

loon ("glyph")	grid ("pch")
circle	19
ocircle (empty circle)	1
ccircle (circle with boundary)	21
square	15
osquare (empty square)	0
csquare (square with boundary)	22
triangle	17
otriangle (empty triangle)	2
ctriangle (triangle with boundary)	24
diamond	18
odiamond (empty diamond)	5
cdiamond (diamond with boundary)	23

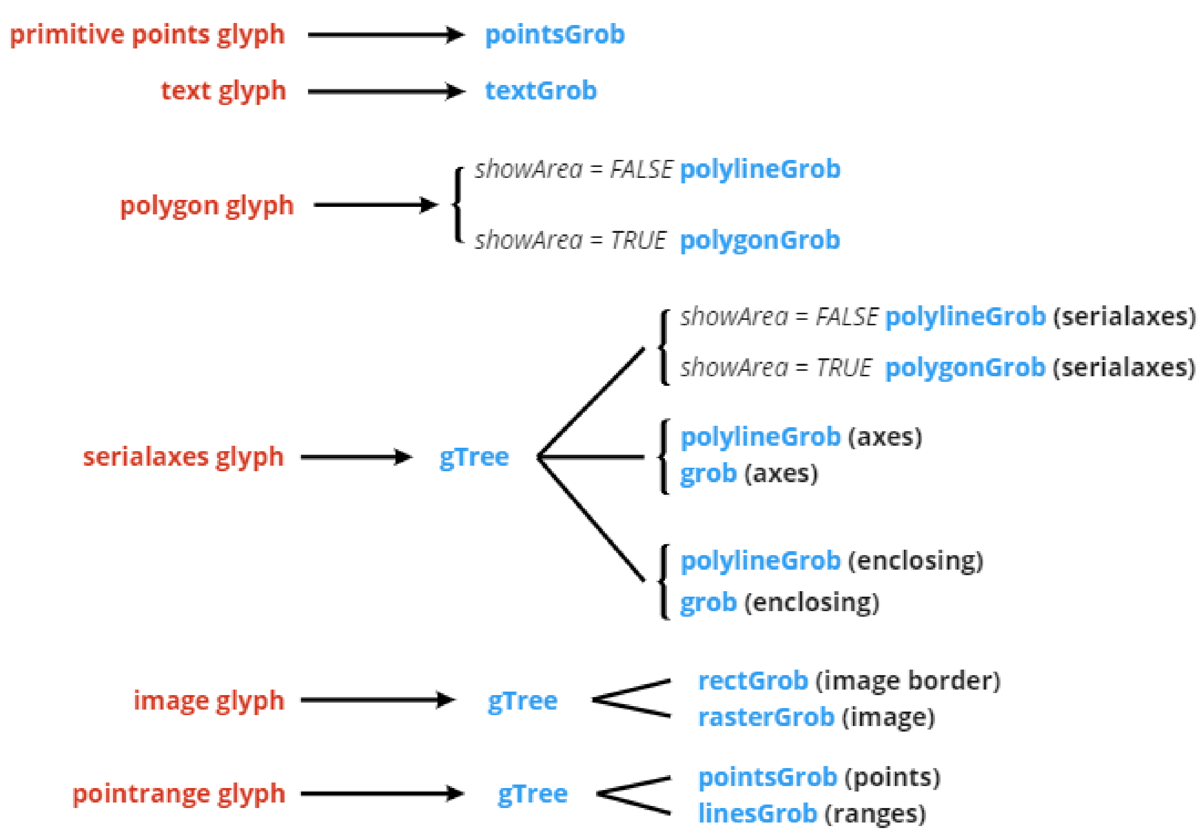


Figure 4.1: Map a `loon` point glyph to a `grid` data structure

4.2.3 Size

The package `loon` uses pixel to define the size of points while `grid` uses `pt` (point size, determined by `fontsize` in `gpar`).

Table 4.2 shows the mapping from size to area (except for polygon and text glyphs) for a glyph (in pixel^2) in `loon`.

For the area based glyphs, we transform the input size to area, then compute the diameter (circle glyph), side (square, triangle, diamond glyph) or length/width (image, star and parallel glyph). In the end, we convert the unit from pixel to `pt`.

The conversion between pixel and `pt` is affected by the resolution of the screen. For example, a pixel (px) at 96`DPI` (dots per inch) is equal to 0.75 point size. Accordingly, an argument `adjust` is applied to linearly twist the conversion, as in

```
> px2pt <- function(adjust = 1) 0.75 * adjust
```

The default `adjust` is 1, meaning no adjustment is applied. However, the sizes of polygon glyph and image glyph are not mapped very well. After several tries, an `adjust` value 0.6 seems satisfying.

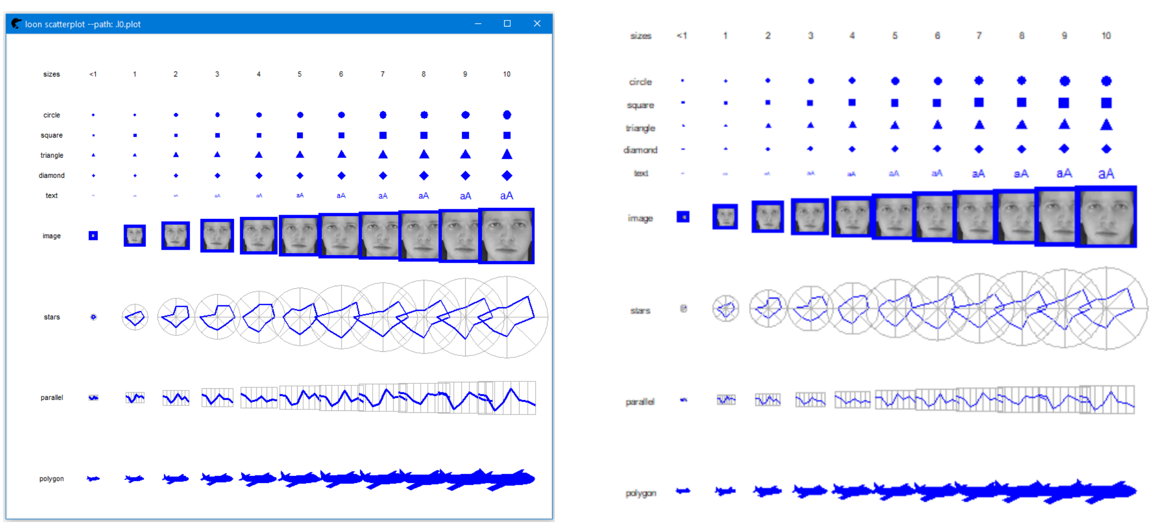


Figure 4.2: Point glyph size mapping from `loon` (left) to `grid` (right).

Figure 4.2 shows the mapping from all available glyphs in `loon`, each with sizes from 1 to 10, to `grid` graphics. In the two graphics above, apparently, element sizes are fairly

Table 4.2: The size mappings in `loon` (Waddell, 2016)

Glyph Type	Area in $\texttt{pixel}^2$
Circle Square Triangle Diamond	$\begin{cases} \texttt{size} < 1: & 8 \\ \texttt{size} \geq 1: & 12 \cdot \texttt{size} \end{cases}$
Text (font size)	$\begin{cases} \texttt{size} < 1: & 2 \\ \texttt{size} \geq 1: & 2 + \texttt{size} \end{cases}$
Images	$\begin{cases} \texttt{size} < 1: & 20 \\ \texttt{size} \geq 1: & 600 \cdot \texttt{size} \end{cases}$
Star Glyphs (for Enclosing)	$\begin{cases} \texttt{size} < 1: & 25 \\ \texttt{size} \geq 1: & 400 \cdot \texttt{size} \end{cases}$
Parallel Coordinate Glyphs (p is number of axes)	$\begin{cases} \texttt{size} < 1: & 9 \cdot (p-1) \\ \texttt{size} \geq 1: & 64 \cdot (p-1) \cdot \texttt{size} \end{cases}$
Polygon Glyphs	`size` does not map to glyph area directly but multiplies the polygon coordinates by $\begin{cases} \texttt{size} < 1: & 4 \\ \texttt{size} \geq 1: & 6 \cdot \sqrt{\texttt{size}} \end{cases}$

close but not identical. Ideally, a new function is required so that we can automatically query the DPI of the machine and precisely convert the unit from pixel to pt.

4.3 loonGrob Data Structure

The package loon mainly provides functionality at a relatively high level graphical abstraction, such as histogram and scatterplot. It also provides some low-level elements, such as the primitive geometrical visuals, as shown in the left column in Table 4.3.

The graphical system grid provides functionality at a relatively low level abstraction. To map low-level elements, the loon primitive layered data structures on the left, in Table 4.3, are mapped to the corresponding grid data structures on the right. However, to map high-level elements, one has to extract the statistical summaries of a complete plot in loon, and then, use primitive graphical visuals to reproduce this plot in grid. In this mapping, from loon to grid, visual display is mapped.

4.3.1 Main Graphics Model

The main graphics model is embedded in the Cartesian coordinate system. To better explain its loonGrob data structure, an example is given as follows, showing a loon scatterplot with a single point and a red line,

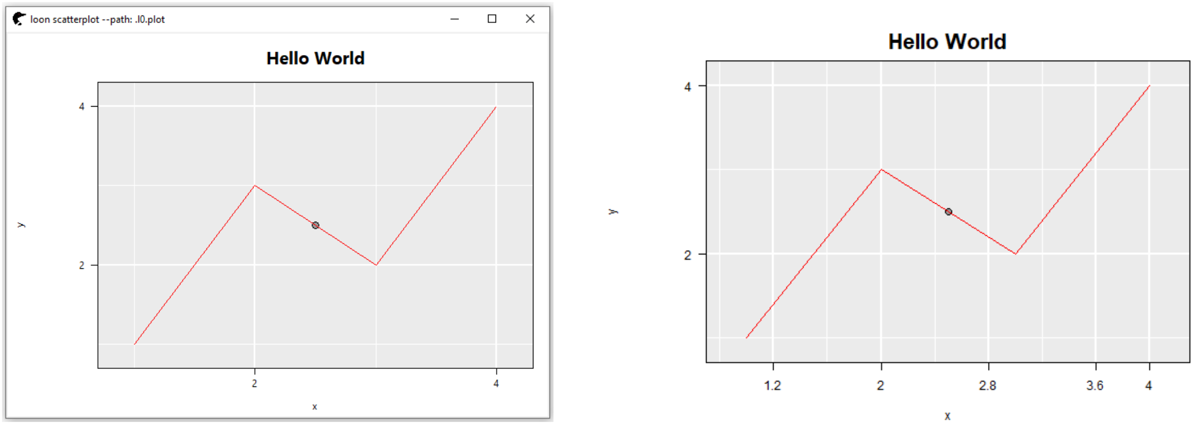
```
> library(loon)
> p0 <- l_plot(x = 2.5, y = 2.5,
+              color = "gray60", size = 4,
+              showScales = TRUE, showLabels = TRUE,
+              xlabel = "x", ylabel = "y",
+              title = "Hello World")
> l0 <- l_layer_line(p0, x=c(1,2,3,4), y=c(1,3,2,4),
+                    color="red", linewidth = 1)
> l_scaleto_world(p0)
```

We can then get the loonGrob of p0 by calling the function loonGrob(), as shown in Figure 4.3 (b).

```
> g0 <- loonGrob(p0)
> grid.draw(g0)
> # or ‘grid.loon(p0)’
```

Table 4.3: Mapping Low-Level Elements

loon	grid
`l_layer_points()`	`pointsGrob()`
`l_layer_group()`	`gTree()`, `gList()`
`l_layer_line()`, `l_layer_lines()`	`linesGrob()`, `polylineGrob()`, `segementGrob()`, `xsplineGrob()`, `curveGrob()`, `pathGrob()`
`l_layer_rectangle()`, `l_layer_rectangles()`	`rectGrob()`, `roundrectGrob()`
`l_layer_polygon()`, `l_layer_polygons()`	`polygonGrob()`
`l_layer_text()`, `l_layer_texts()`	`textGrob()`
`l_layer_oval()`	`circleGrob()`



(a) (b)

Figure 4.3: Figure (a) is a screenshot of the `loon` plot and Figure (b) is a `grid` graphic.

Non-data Element `loonGrob` Data Structure

Figure 4.4 shows the hierarchical `loonGrob` data structure of `g0`. It is a tree-based model. The solid line means *beneath*. For instance, the `bounding box gTree` and `loon plot gTree` are beneath the `l_plot gTree`.

The `bounding box gTree` controls the visuals (e.g., background color) of the margin region and the `loon plot` controls the visuals of the plot region. Beneath the `loon plot gTree`,

- the `guides gTree` contains the vertical and horizontal guide `lineGrob`s;
- the `label gTree` has three `textGrob`s representing the x label, y label and title;
- the `axes gTree` stores the `xaxisGrob` and `yaxisGrob`;
- the `clipping region grob` sets the clipping region within the current viewport;
- the `boundary rectangle grob` draws the plot region boundary lines.

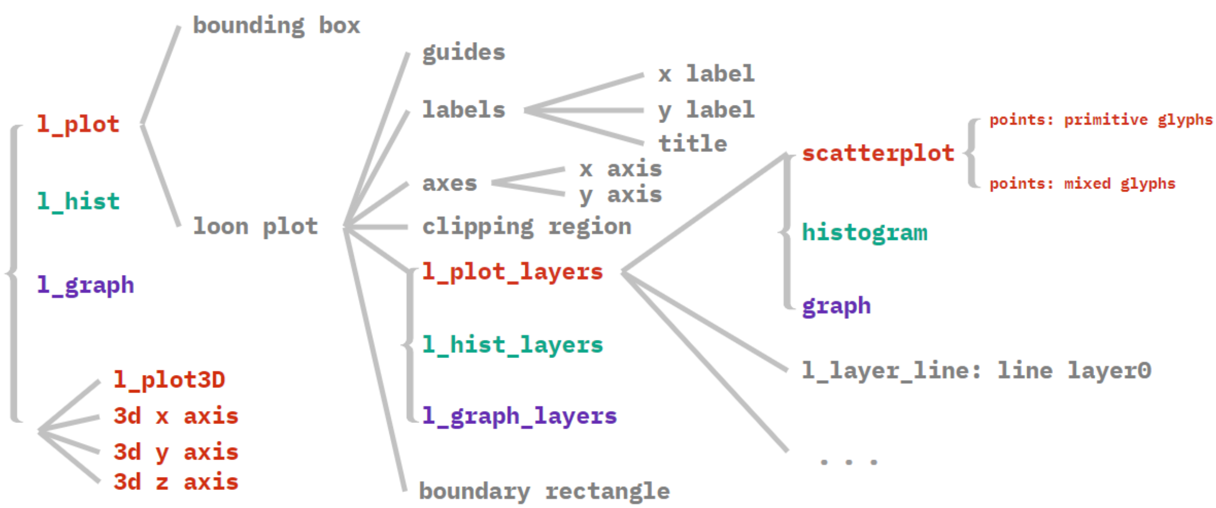


Figure 4.4: Each label represents a `gTree` or a `grob`. Gray ones are applicable to all models, while the colored ones can only be applied to the corresponding colored models. For example, when the main graphic model is histogram, the label names at the corresponding levels would be `l_hist`, `l_hist_layers` and `histogram`.

loonGrob **Data Structure of** l_plot

The `l_plot_layers` `gTree` contains all the geometric graphical visuals and plays a key role in the display. All the points' features are stored in the layer `scatterplot` whose children is either `points: primitive glyphs` or `points: mixed glyphs`.

The `points: primitive glyphs` is a `pointGrob`; therefore, all points have primitive shapes (e.g., pch is from 0 to 25).

The `points: mixed glyphs` is a `gTree` structure and each child represents an individual point. For example, suppose some non-primitive glyphs are displayed in `loon`, as Figure 2.9, after transformation, the child of the `scatterplot` `gTree` would be `points: mixed glyphs` and its data structure is shown as 4.5. The glyph of point 1 is an image. After

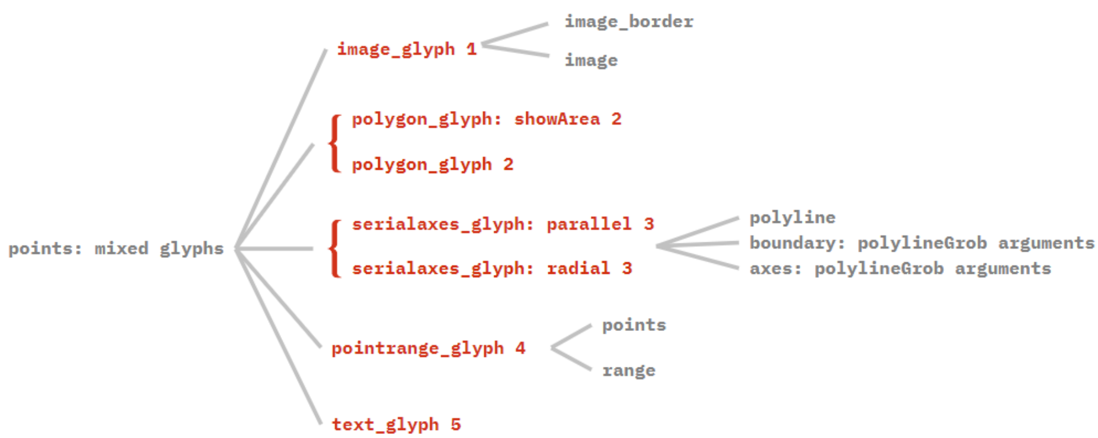
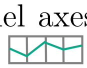


Figure 4.5: The data structure of a `loonGrob` non-primitive glyph

transformation, it is mapped to a `gTree` structure containing a `rectGrob` for the image border and a `rasterGrob` for drawing the image. The point 2 has a polygon glyph. When `showArea = TRUE`, it is mapped to a `polygonGrob`; else, a `polylineGrob`. The point 3 has a serialaxes glyph embedded in the parallel coordinate. After transformation, it is mapped to a `gTree` structure containing three children: serialaxes lines, boundaries and axes. When a parallel axes glyph point is displayed with boundaries and axes, it would be, for example, like . The "unusual" names, `boundary: polylineGrob arguments` and `axes: polylineGrob arguments` will be discussed later. The glyph of point 4 is a point range. After transformation, it is mapped to a `gTree` containing a `pointsGrob` and a `lineGrob`. The point 5 has a text glyph and is mapped to a `textGrob`.

`loonGrob` Data Structure of `l_plot3D`

The `loonGrob` data structure of an `l_plot3D` widget is almost the same as that of an `l_plot` widget, but contains additional three axes objects (viz., `3d x axis`, `3d y axis` and `3d z axis`) measuring the rotation angle.

`loonGrob` Data Structure of `l_hist`

In the `loonGrob` data structure of an `l_hist` widget, the bin visuals are stored beneath the `histogram` `gTree`, constructed by `rectGrob`s.

`loonGrob` Data Structure of `l_graph`

In the `l_graph` `loonGrob` data structure, the children of the `graph` `gTree` are `graph edges`, `graph nodes` and `graph labels`. If the `l_graph` widget is a navgraph, children `navpath navigator*` and `navpoints navigator*` are created to record the location of the navigator(s) and their paths (`*` represents a navigator's index, starting from 0).

`loonGrob` Data Structure of Dependent Layers

The name of the dependent layer `grob` is composed of the layer name, layer type and layer id. For `g0`, the name of the line is `l_layer_line: line layer0`.

Query and Modify a `loonGrob`

One can query the graphical objects via the function `getGrob()` using the object's name. For example, the name of the `grob` displaying the title (call `titleGrob`) is "title", one can get the `grob` by its name "title" as follows;

```
> library(grid)
> # extract the title
> titleGrob <- getGrob(g0, gPath = "title")
> titleGrob
[1] text[title]
```

Then, one can access the title (i.e., "Hello World") of this `loonGrob`.

```
> titleGrob$label # title
[1] "Hello World"
```

In addition, one can modify the graphical objects of `g0` by using `editGrob()` or `grid.edit()`. For example, the attributes of the line can be altered using the following code, as shown in Figure 4.6.

```
> # modify the line color and line type
> grid.edit("l_layer_line: line layer0",
+           gp = gpar(col = "blue", lwd = 3, lty = 2))
```

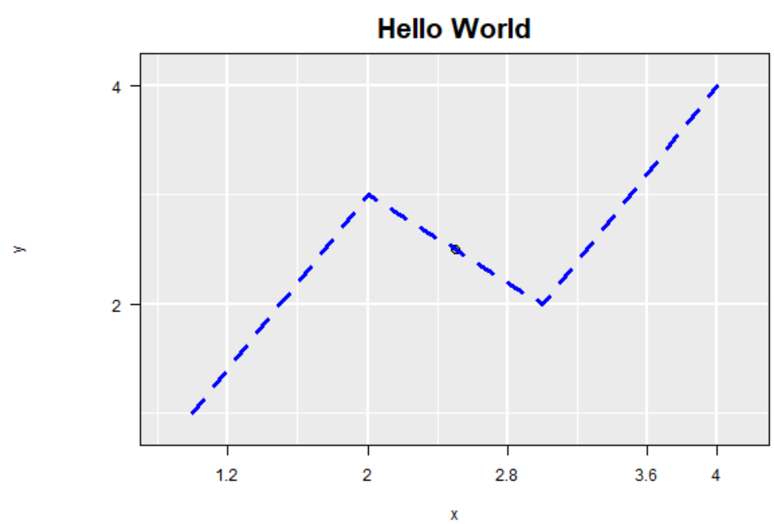


Figure 4.6: This is a modified version of Figure 4.3 (b). The red line is changed to a blue thick dashed line.

Completeness of a `loonGrob`

All of the elements of a `loon` plot appear in the `grob`, either explicitly or, if they were not drawn in the `loon` plot, as an empty `grob` (by the function `grob()`) containing the relevant arguments to drawing them.

For example, if the axes/labels were not displayed (set `showAxes = FALSE` or set `showLabels = FALSE`) in `loon`; or the points were deactivated (invisible) in `loon`, after transformation, they could still be found in `grid`, but their names in the new `loonGrob` were changed.

Continuing with the `loon` plot `p0`, we first turn off the labels,

```
> p0['showLabels'] <- FALSE
```

and then get a new `loonGrob` `g1`, as follows,

```
> g1 <- loonGrob(p0)
```

The name of the `titleGrob` is not "title" but "title: textGrob arguments". It is composed of the name of a graphical element (e.g., "title"), a geometric `grob` name (e.g., "textGrob") and a string "arguments".

```
> titleGrob <- getGrob(g1, gPath = "title")
NULL
> titleGrob <- getGrob(g1, gPath = "title: textGrob arguments")
> titleGrob
grob[title: textGrob arguments]
> titleGrob$label
[1] "Hello World"
```

This `titleGrob` is constructed by the function `grob()` which stores all essential features but does not produce any geometric visuals.

4.3.2 Serialaxes Model

A serialaxes plot, embedded in either the parallel coordinate system or the radial coordinate system, is used to visualize high dimensional data. To map the visual display of a serialaxes model from `loon` to `grid` graphics, we break down the high-level elements (serialaxes plot) to low-level geometric visuals such as lines or polygons, then map them to `lineGrob`s or `polygonGrob`s.

A `loon` parallel plot is created below and the data structure is shown in Figure 4.7.

```
> s <- l_serialaxes(iris[, -5],
+                   color = iris$Species,
+                   showGuides = FALSE,
+                   axesLayout = "parallel")
```

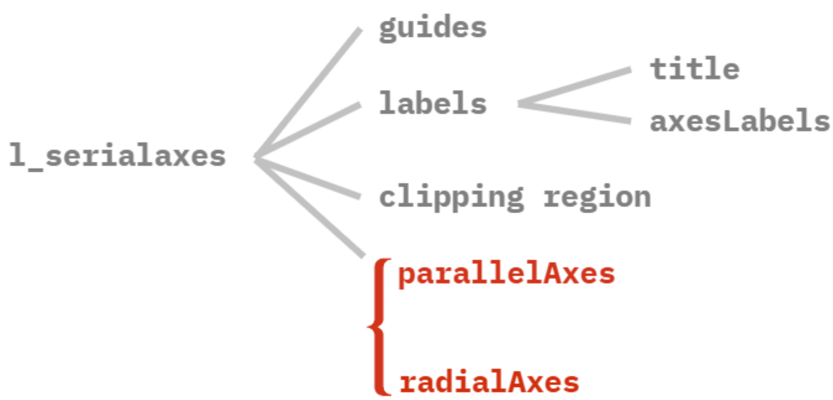


Figure 4.7: The `l_serialaxes loonGrob` data structure. The order (from top to bottom) is the default stacked order. If the graph was embedded in the parallel coordinate, the children was `parallelAxes`, else it was `radialAxes`

The children of the `parallelAxes` (or `radialAxes`) are `lineGrob`s that each represents an observation. The details of how to construct a serialaxes plot in `grid` are described as follows:

Suppose the data $\boldsymbol{X} = [x_{ij}]$ has n observations and k variables, where $i \in [1, ..., n]$ and $j \in [1, ..., k]$. The data $\boldsymbol{X}$ will be scaled to $\boldsymbol{X}^*$ by one of the four scaling methods `variable`, `observation`, `data` or `none`.

- `variable`: $\mathbf{x}_j^* = \frac{1}{a_j - b_j}(\mathbf{x}_j - b_j\mathbf{1})$ where $\boldsymbol{X} = [\mathbf{x}_1, ..., \mathbf{x}_p]$, $\mathbf{x}_j$ is a $n \times 1$ vector, $a_j = \max(\mathbf{x}_j)$ and $b_j = \min(\mathbf{x}_j)$;
- `observation`: $\mathbf{x}_i^* = \frac{1}{c_i - d_i}(\mathbf{x}_i - d_i\mathbf{1})$, $\boldsymbol{X} = [{\mathbf{x}_1}^\mathsf{T}, ..., {\mathbf{x}_n}^\mathsf{T}]^\mathsf{T}$, $\mathbf{x}_i$ is a $p \times 1$ vector, $c_i = \max(\mathbf{x}_i)$ and $d_i = \min(\mathbf{x}_i)$;
- `data`: $x_{ij}^* = \frac{1}{e-f}(x_{ij} - f)$, where $\boldsymbol{X} = [x_{ij}]$, $e = \max(x_{ij})$, $f = \min(x_{ij})$;
- `none`, $x_{ij}^* = x_{ij}$.

In a parallel coordinate system, upon transformation, each data point x_{ij}^* is restricted within a $[0, 1] \times [0, 1]$ canvas. For observation i, the horizontal coordinate of the line is $\boldsymbol{l}_i = [\frac{0}{k-1}, ..., \frac{j}{k-1}, ..., 1]_{1\times k}^\mathsf{T}$ and the vertical coordinate is $\mathbf{x}_i^*$. For a point parallel axes glyph, located at x_p and y_p, with size s_p, then $\boldsymbol{l}_i$ and $\mathbf{x}_i^*$ will be mapped to $\theta \times \boldsymbol{l}_i + x_p \times \mathbf{1}$ and $\theta \times \mathbf{x}_i^* + y_p \times \mathbf{1}$ ($\theta \propto s_p$).

In a radial coordinate, the angle 2π are equally split into k angles where the $\boldsymbol{a} = [a_j]_{k\times 1}$ ($a_j = 2\pi\frac{j-1}{k}$). To display the ith observation $\mathbf{x}_i^*$ of the radial axes plot, we first construct a diagonal matrix $\boldsymbol{D}_i$, where $\text{diag}(\boldsymbol{D}_i) = \mathbf{x}_i^*$; then the coordinates, $\mathbf{x}_i = \boldsymbol{D}_i\boldsymbol{a}_x + \boldsymbol{c}_x$ and $\boldsymbol{y}_i = \boldsymbol{D}_i\boldsymbol{a}_y + \boldsymbol{c}_y$, where $\boldsymbol{a}_x = [r\cos(\boldsymbol{a})]_{k\times 1}$, $\boldsymbol{a}_y = [r\sin(\boldsymbol{a})]_{k\times 1}$, $\boldsymbol{c}_x$ and $\boldsymbol{c}_y$ are the center of the `x` and `y`. With the default setting, $r = 0.5$, $\boldsymbol{c}_x = \boldsymbol{c}_y = [0.5]_{k\times 1}$ (all radial axes are centered at $[0.5, 0.5]$ with a maximum radius 0.5). For a point radial axes glyph, located at x_p and y_p, with size s_p, then, $c_x = x_p$, $c_y = y_p$ and radius $r = \theta s_p$ (θ is some scaling scalar).

4.3.3 Compound Plot

The `loon` package provides compound objects (e.g., `l_pairs`, `l_facet` and `l_ts`) that several `loon` graphics are drawn simultaneously to uncover the patterns of interest of data. For example, Figure 2.10 is a `loon` compound pairs plot. To transform an `l_compound` object to a `grid` object, we first transform each widget in this compound object to an individual `loonGrob`; then, query the names (the name of each plot in an `l_compound` object is the layout position) and construct a layout matrix. Finally, we arrange multiple `loonGrob`s in a single panel using packages `gridExtra` and `gtable` (Wickham and Pedersen, 2019).

4.4 Summary

The package `loonGrob` is a one way bridge to turn interactive `loon` plots into static `grid` graphics by mapping visual displays from one to the other. The main reason why we have to use visual displays in this mapping is visual structure mapping requires the same level of graphical abstraction (e.g., high to high or low to low). However, the package `grid` provides a general-purpose graphical system which is only equipped with structures at a relatively low level of abstraction. In contrast, many `loon` visual structures, such as scatterplot with non-primitive glyphs, histogram and serialaxes plot, are at a high level of abstraction and cannot be mapped in `grid`.

4.4.1 Lessons

An important lesson we have learned in building loonGrob is the ease of constructing a bridge significantly depends on how well the abstraction levels match.

When the abstraction levels are completely matched, it is fairly easy to create a bridge such as mapping the primitive layer visual (e.g., l_layer_line to lineGrob). When the abstraction levels are not perfectly matched, it is relatively hard to build a bridge. As there is no high-level structure in grid, some actions are required to break high-level elements down to low-level ones before the mapping (e.g., a loon histogram to rectangle structures, rectGrob in grid).

4.4.2 Limitations

The hierarchical data structure of a loonGrob is extremely useful to retrieve, edit and replace an object. It preserves the whole loon data structure. However, this preservation comes at a cost of rendering speed and memory consumption.

Table 4.4 shows the benchmark of a loonGrob scatterplot with 1,000, 10,000 and 100,000 points. Compared to the same for ggplot scatterplots (Table 3.6), the rendering time of a loonGrob is shorter when the number of points is small (e.g., 1,000); as the number increases to 10,000, the speeds of two objects are almost the same; if the number is large (e.g., reach to one million or above), the rendering of a ggplot object is approximately 1 second faster on average.

Table 4.5 shows the memory consumption of a loonGrob and a ggplot object. When the number of points is small, a ggplot object only requires one-fifth of memory of a loonGrob being used. As the number increases, the difference of the memory consumption is getting smaller, but a ggplot object is more economic. That is because when the data size is small, the design of loonGrob data structure which reproduces all loon data structure, dominates the memory consumption; in contrast, when the data size is large, the data dominates.

4.4.3 Further Work

The major obstacle we encountered is the size conversion. In loon, the size is defined by pixel which is affected by screen resolution, while in grid, there is no pixel unit. To map the size in perfect, we have to find the DPI of the screen and precisely convert the pixel to pointsize. So far, it is not available yet.

Table 4.4: `loonGrob` Rendering Time

summary (sec) / number of points	min	lq	mean	median	up	max	eval
`1,000`	0.067	0.069	0.071	0.070	0.073	0.077	100
`10,000`	0.643	0.658	0.666	0.662	0.676	0.700	100
`100,000`	5.908	6.517	6.593	6.541	6.597	7.825	100

Table 4.5: Memory Consumption (MB)

Number of points	loonGrob	ggplot2
1,000	0.109	0.022
10,000	0.250	0.159
100,000	1.623	1.532

Chapter 5

Loon.shiny

5.1 Introduction

The package `shiny` (Chang et al., 2019), wrapping `Javascript`, `CSS` and `HTML` in R functions, allows users with little experience in those areas to build nice web applications. Packages `loon` and `shiny` both provide direct manipulation; nevertheless, there are still some reasons to motivate us to render a `loon` widget into a `shiny` app.

A `shiny` app is composed of two components, a `ui` (user interface) object and a `server` function. The `ui` object is responsible for creating the layout of an app, with which users can insert inputs (e.g., `selectInput`, `textInput`) to control the specification. The `server` is an inner function responsible for generating the logic of an app. When users manipulate on the page, the graph of dependencies specified inside the `server` function allows it to arrange changes immediately to reflect the interactivity. This `ui`/`server` pair is passed as arguments to the function `shinyApp()` to create an interactive `shiny` app (Wickham, 2021).

The package `shiny` provides powerful presentation graphics. However, most web-based graphics are not satisfying in data exploration, as they usually suffer from one issue: once the plot is rendered, it is difficult (or impossible) to manipulate graphical components (specified in compilation) outside the browser interface. Thus, users have to render the session to check the output and stop the session to modify the layout or the logic. Additionally, although `shiny` has already simplified the procedure of creating a web app, users still need extra work to come up with a powerful data analysis toolkit, in order to accommodate questions like how to set logic to best achieve the interactivity.

`Loon` provides powerful tools for data exploration, however, after exploring data interactively, how to efficiently present the analysis to the audience. A short video or GIF could be an option, however, they are kinematic graphics and do not offer any direct manipulation.

The package `loon.shiny` (Xu and Oldford, 2019b) (see https://github.com/great-northern-diver/loon.shiny) wraps the `ui` design and `server` function to create a `loon.shiny` app whose layout and logic specification largely restore `loon`'s design. For example, users can sweep the mouse to create a region and points falling into this region can be highlighted; plotting states of highlighted points can be easily modified by the inspector and et cetera. Analysts who explore data in `loon` now can present their interactive graphics easily in a `shiny` web app.

This chapter begins with an introduction of the `ui` design of a `loon.shiny` app. After specifying the layout, the `server` specification is discussed illustrating how direct manipulation of this app is realized. Sometimes, direct manipulation may cause a change of the user interface. Therefore, the details of the dynamic `ui` are discussed. This chapter closes with limitations and a summary of `loon.shiny`.

5.2 User Interface

A user interface can be considered as a guideline telling users what objects can be manipulated in an application.

The `ui` in a `loon.shiny` web app restores the original `loon` appearance to a great extend. It is composed of two views, a fixed display (output graphics) and a fluid inspector (input toolkits). The inspector consists of a *World View* window (the graph under the navigation bar menu), a *Plot* panel, a *Linking* panel, a *Select* panel, a *Modify* panel, a *Layer* panel (not for the serialaxes plot) and a *Glyph* panel (only for the scatterplot).

5.2.1 Singleton Design

A `loon` widget comes with two windows, a graphic window and a "singleton" inspector window. "Singleton" means that there is only one instance of it. Each graphic model (scatterplot, graph, histogram, serialaxes plot) has its own specified inspector. When more than one `loon` displays are presented, the shown one depends upon which display receives the last mouse gesture input or the window focus event.

In a `loon.shiny` web app, since only one window (containing graphics and inspectors) is displayed, in order to realize the singleton design, a navigation bar menu is required.

The shown inspector (only one instance) can be switched by toggling the tabpanel or by the last mouse gesture input (`<double-click>`) on the graphics.

For example, the following code shows a `loon.shiny` app (as shown in Figure 5.1). There are three linked `loon` widgets: the top left is a histogram with the variable "Sepal length"; the bottom right is a swapped histogram with the variable "Sepal width" and the bottom left is a scatterplot with `x` representing the "Sepal length" and `y` representing the "Sepal width", as in

```
> library(loon.shiny)
+ p1 <- l_plot(iris,
+              linkingGroup = "iris",
+              showLabels = FALSE)
+ p2 <- l_hist(iris$Sepal.Length,
+              linkingGroup = "iris",
+              showLabels = FALSE,
+              showStackedColors = TRUE)
+ p3 <- l_hist(iris$Sepal.Width, linkingGroup = "iris",
+              color = iris$Species, sync = "push",
+              showLabels = FALSE, swapAxes = TRUE,
+              showStackedColors = TRUE)
+ loon.shiny(list(p1, p2, p3),
+            layoutMatrix = matrix(c(2,NA,1,3),
+            nrow = 2, byrow = TRUE))
```

In this case, the window focus is on the scatterplot. To switch the window focus to the top histogram, one can double click any area on the top histogram or toggle the navigation bar to "Histogram2".

5.2.2 *World View* Window

Figure 5.2 shows the *World View* window, one for a `loon` widget (left) and one for a `loon.shiny` app (right). The *World View* only displays active elements (e.g., points, bins). The black thick outline shows the display view port.

In `loon`, the *World View* allows users to interact with the `loon` plot by scrolling the mouse wheel (zooming) or dragging the view area (panning).

In contrast, the *World View* is static in a `loon.shiny` app because in the current `shiny` version, one plot cannot be used to interact with another plot(s). To zoom or pan, one has to drag slider bars in the *Plot* panel (see next subsection).

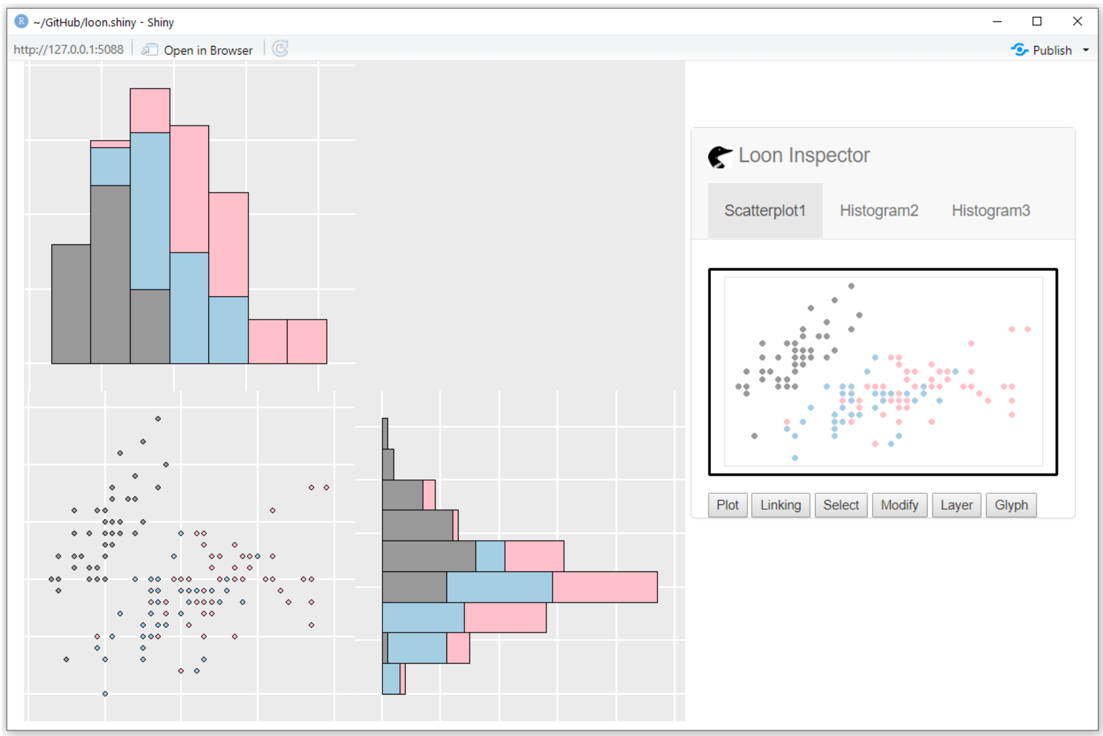


Figure 5.1: This `loon.shiny` app is composed of three linked `loon` plots, a scatterplot and two histograms.

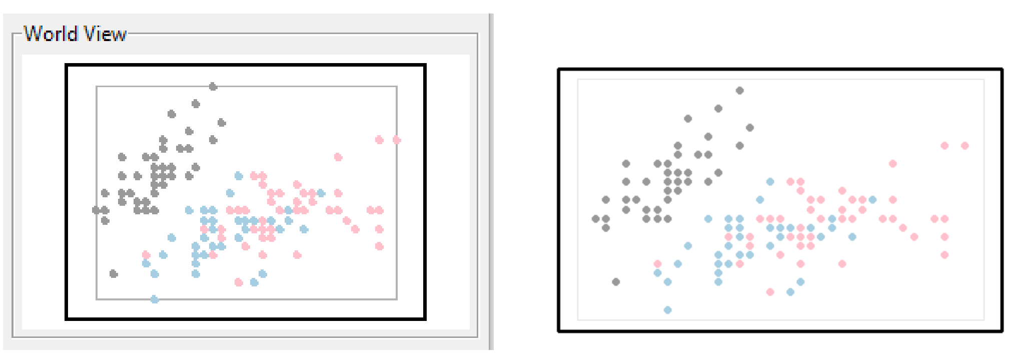


Figure 5.2: `loon` (left) and `loon.shiny` (right) *World View* windows

5.2.3 *Plot* Panel

The *Plot* panel controls the display over all non-data elements such as swapping axes, showing or hiding labels/scales/guides.

Model `l_plot`

Figure 5.3 shows the `loon` (left) and `loon.shiny` (right) `l_plot` *Plot* panels. Two slider

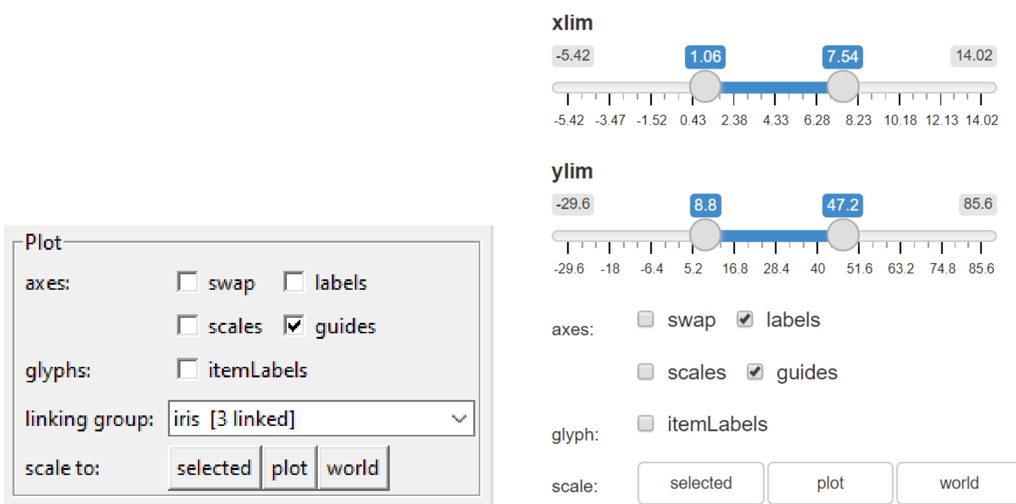


Figure 5.3: `l_plot` `loon` (left) and `loon.shiny` (right) *Plot* panels

bars are provided to control the `x` limit and `y` limit. Reasons and limitations are demonstrated in Subsection 5.5.2

In the "axes" channel, labels/scales/guides can be turned on/off and axes can be swapped.

In `loon` and `loon.shiny`, querying is triggered by hovering the mouse over a point (with `itemLabels` is on). Then a tooltip pops up with detailed information of this element.

When points are overlapped, in a `loon` widget, the toolbox only shows queries of the top-most point, as shown in Figure 2.3. In contrast, in a `loon.shiny` app, the toolbox shows all points' queries, as shown in Figure 5.4.

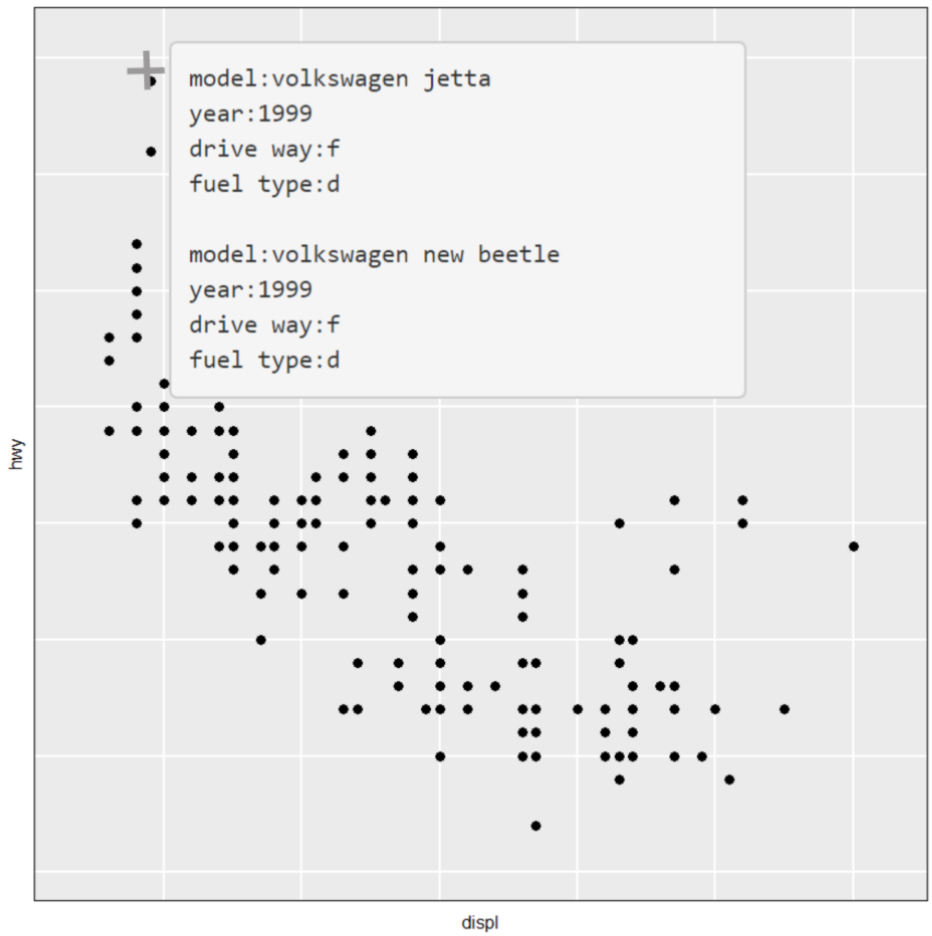


Figure 5.4: When querying, a `loon.shiny` app shows the detailed information of all overlapped points, in this case, the shown automobiles are "Jetta" and "New Bettle".

In the "scale to" channel, the plot interior can be adjusted to: the scales of selected points (the "selected" button); the scales of all points in the plot layer (the "plot" button); the scales of all plot objects in all layers (the "world" button).

Model `l_hist`

Figure 5.5 shows the `loon` (left) and `loon.shiny` (right) `l_hist` *Plot* panels.

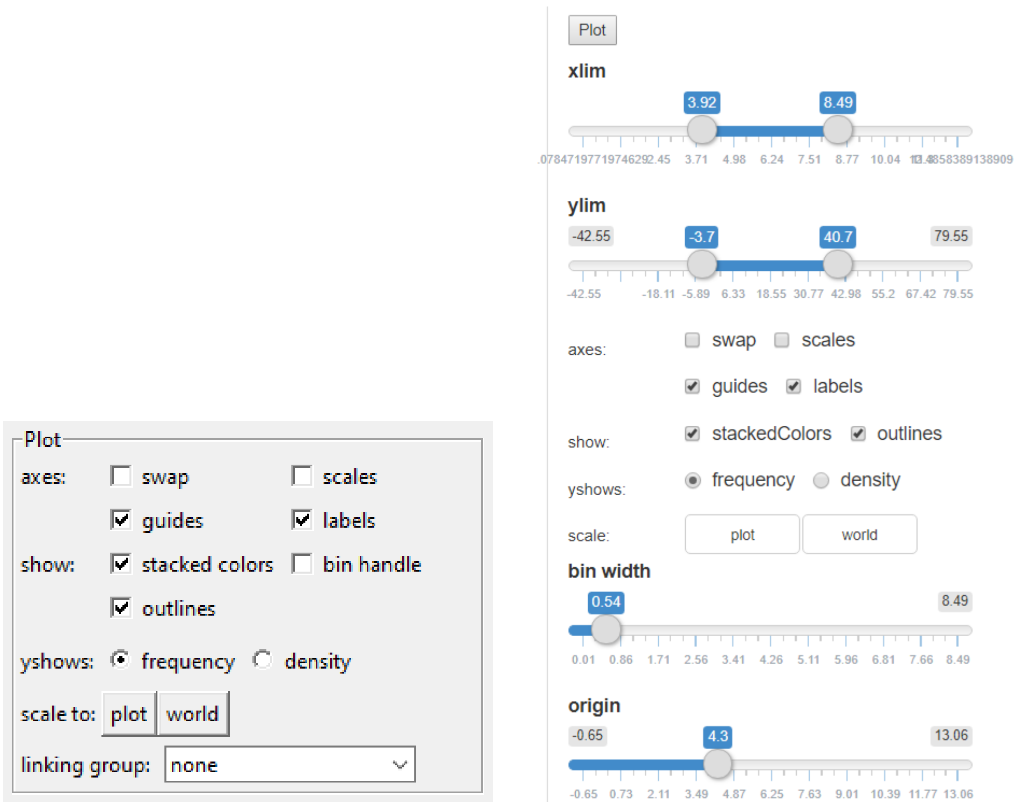


Figure 5.5: `l_hist` `loon` (left) and `loon.shiny` (right) *Plot* panels

When `stackedColors` is toggled off, the color of all bins will be set as thistle; otherwise, the elements are split into several groups and each group is represented by an individual color. When `outlines` is toggled off, no bin boundaries are displayed. The `yshows` controls the representation (e.g., frequency and density) of the y axis.

In `loon`, bins can be modified via the graphical element ▯↦ that is inside a histogram graphic. Since `Shiny` does not support inserting a graphical element into a plot yet, the element ▯↦ is replaced by two slider bars, controlling the bin-width and bin-origin accordingly.

Model `l_serialaxes`

Figure 5.6 shows the `loon` (left) and `loon.shiny` (right) `l_serialaxes` *Plot* panels. Two

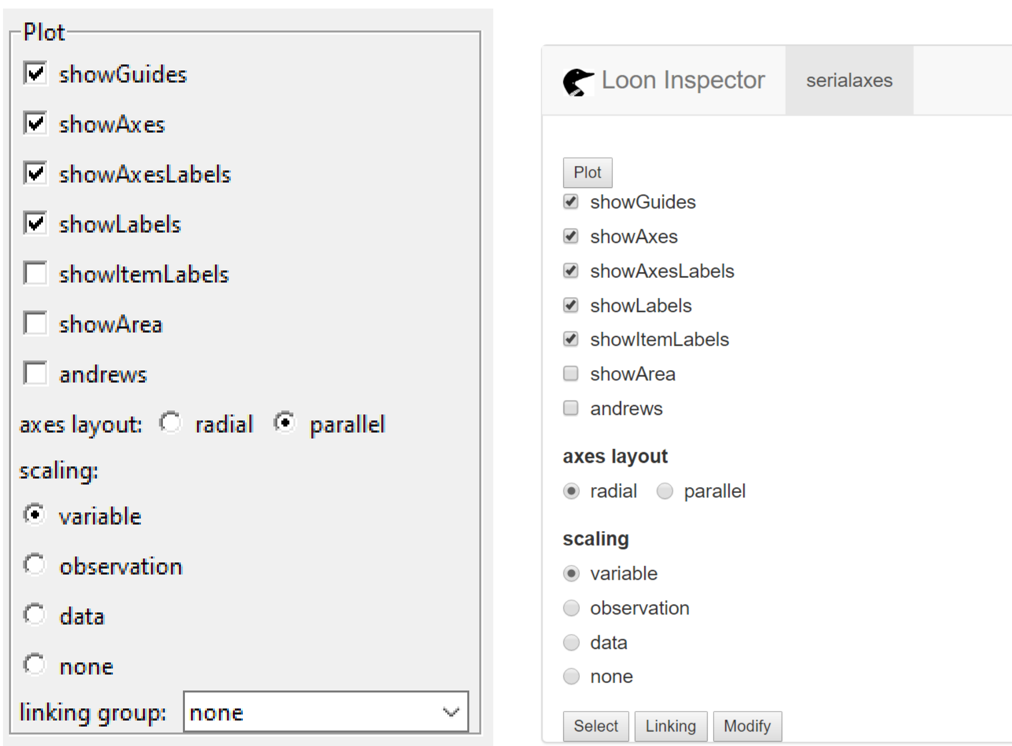


Figure 5.6: `l_serialaxes` `loon` (left) and `loon.shiny` (right) *Plot* panels

designs are almost the same.

5.2.4 *Select* Panel

Figure 5.7 shows the `loon` (left) and `loon.shiny` (right) *Select* panels.

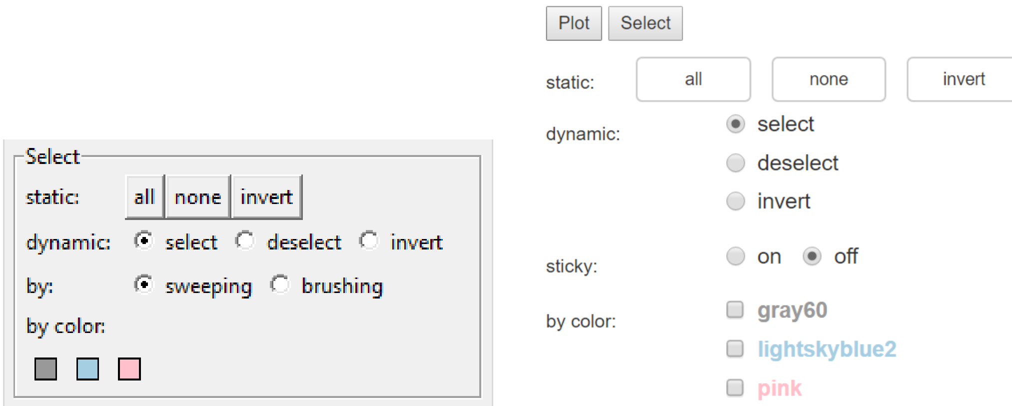


Figure 5.7: `loon` (left) and `loon.shiny` (right) *Select* panels

The `loon` inspector provides a `by` channel that one can select points either **by** `brushing` or `sweeping`. However, in a `loon.shiny` app, `by` channel is not available and `brushing` as well as `sweeping` has been pre-defined. Once the app is rendered, the way of selection is no longer changeable.

In `loon`, multiple-steps selection can be realized by holding the `<shift>` key. However, tracking a key input in `shiny` is not easy. In contrast, multiple-steps selection is realized by a "sticky" radio box in a `loon.shiny` app.

In addition, channel `by color` in a `loon` inspector is replaced by a checkbox-group input in a `loon.shiny` app. More details will be introduced in Subsection 5.4.2.

5.2.5 *Linking* Panel

In `loon`, suppose one wants to push (or pull) the linked states of one plot to (or from) other plots or to reset the linked states, command-line is often adopted (see Subsection 2.2.4 for the details of linking in `loon`).

However, in `loon.shiny`, once the app is rendered, settings are not allowed to be changed programmatically. Therefore, we display all linking associated states within the panel *Linking* (for all five models), as shown in Figure 5.8.

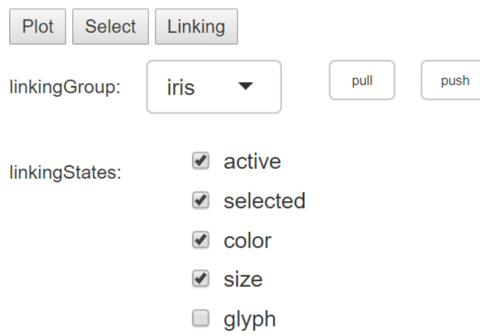


Figure 5.8: The *Linking* panel

5.2.6 *Modify* Panel

Figure 5.9 shows the `loon` (left) and `loon.shiny` (right) *Modify* panels of an `l_plot` widget (the designs are similar for rest models).

In `loon`, new color buttons can be added to the panel by clicking the "+" or "+5" button. However, in `shiny`, once an app is rendered, no new buttons can be created. In contrast, in `loon.shiny`, the color picker widget (Attali, 2020) is used.

Additionally, the transparency settings are also allowed in `loon.shiny` (e.g., `l_plot`, `l_graph` and `l_serialaxes`) by modifying an "alpha" slider bar and an "apply" button.

5.2.7 *Layer* Panel

The *Layer* panel (for `l_plot`, `l_hist` and `l_graph`) is to modify layers, as shown in Figure 5.10. In `loon` (left), a listbox widget is displayed showing the layer name, layer type and layer id. As `shiny` does not yet provide a listbox widget, in `loon.shiny` (right), the design is simplified by a select box.

The buttons beneath the "layer" channel are used to move this layer up or down a level; make this layer visible or invisible; add a new layer group (not implemented yet) or delete this layer; scale the view port to the region of this layer. The last command "label name:" is to customize the label of the focused layer. Note that buttons that help move the focused layer inside or outside a group layer in `loon` are not implemented in a `loon.shiny` app yet.

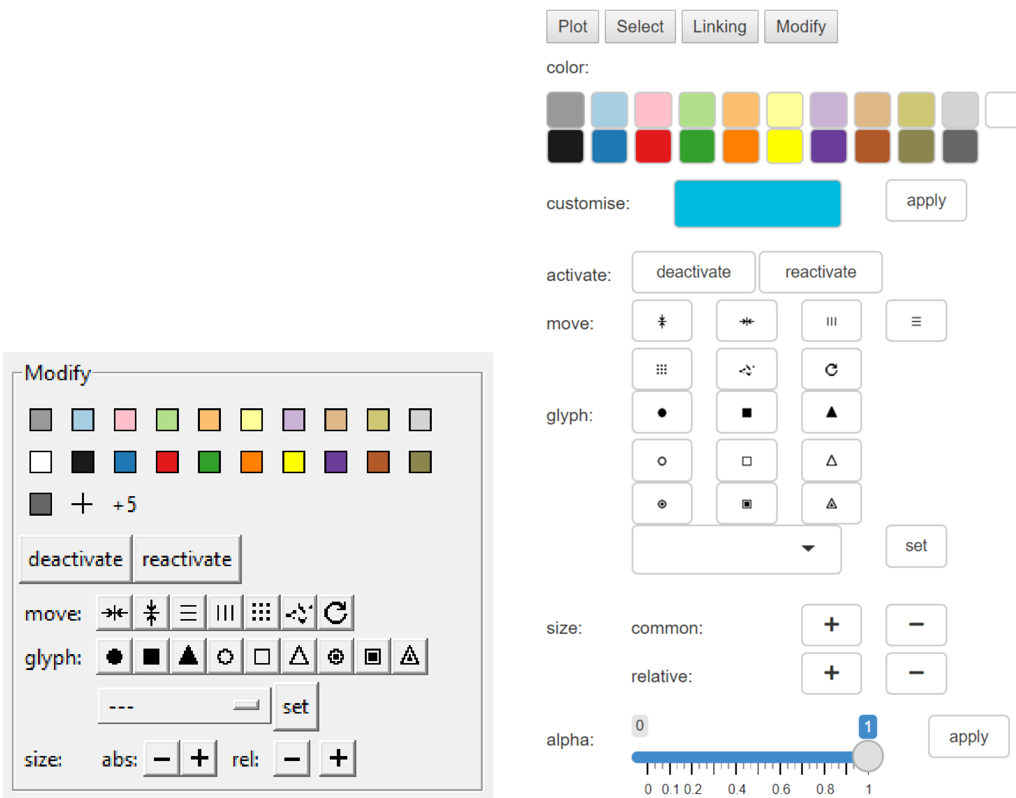


Figure 5.9: `loon` (left) and `loon.shiny` (right) `l_plot` *Modify* panels

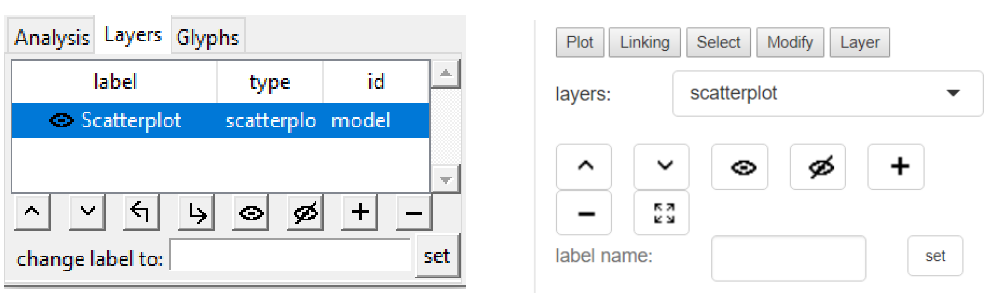


Figure 5.10: `loon` (left) and `loon.shiny` (right) *Layer* panels

5.2.8 *Glyph* Panel

The *Glyph* panel (for `l_plot` only) is to modify the appearance of non-primitive glyphs. Different non-primitive glyphs have different designs,

- Serialaxes glyph: three checkbox toolkits are to control, whether to show enclosing boxes; whether to display axes; whether to fill the glyph regions, respectively;
- Polygon glyph: a checkbox toolkit is to control whether to fill the area;
- Point-range glyph: a checkbox toolkit is to control the shape of the point, solid or empty.

5.3 Interactivity

A `loon.shiny` app is based on the `loonGrob` bridge (see Section 5.6). As a `loon` widget is passed into the function `loon.shiny()`, it will be transformed to a `grid` object first using the function `loonGrob()`. Changes on the app can cause a re-evaluation of the `server`, and inside the `server`, the data structure of the `loonGrob` would be modified, then an updated `grid` graphic would be displayed.

An `observer` object (by the function `observe()`) is used to construcut a `server` function for updating the graphics and dynamic `ui` (introduced in Section 5.4). It monitors changes in all reactive values such as pressing a button (inputs) or selecting points in a graphic (outputs) in its environment. Meanwhile, it uses eager evaluation which can automatically re-execute the app whenever any changes are detected. The following code shows the design of the `loon.shiny` `server` function,

```
>  server <- function(input, output, ...) {
+    ...
+    shiny::observe({
+      # to update the user interface
+      ...
+      # to update graphics
+      ...
+    })
+  }
```

The `input` is a list-like object, named according to the input ID, that contains all the input data sent from the app; the `output` object is very similar to `input`, also a list-like object named according to the output ID, but used to send output (i.e., graphics).

5.3.1 Plot Region

One can hover over the `xlim` (or `ylim`) slider bar (e.g., see Figure 5.3) till it shows ⟷. Then dragging the double-headed arrow will pan the view port horizontally (or vertically). In order to zoom in or out the view port, one can drag the round circle button on the `xlim` (or `ylim`) slider bar.

The logic is that, if any of the `xlim` or `ylim` was modified, then, in the `observer`, the new xlim and ylim would be assigned to handles `newxlim` and `newylim`, as in

```
> newxlim <- input$xlim
> newylim <- input$ylim
```

The new viewport of the `grid` graphics will be edited as in

```
> grid::setGrob(
+     gTree = lg,
+     gPath = "loon plot",
+     newGrob = grid::editGrob(
+       grob = grid::getGrob(lg, "loon plot"),
+       vp = dataViewport(xscale = newxlim,
+                         yscale = newylim,
+                         name = "dataViewport")
+     )
+   )
```

where `lg` is a `loonGrob` transformed from a `loon` widget.

Buttons "plot", "world", "selected" are used to re-scale the window. When any of them are pressed, in the `observer`, `newxlim` and `newylim` are assigned to corresponding scales to modify the data view port.

5.3.2 Non-data Element States

When the axes are swapped, in the `observer`, the coordinates, labels and scales are flipped.

Labels, axes and guides can be turned on or off. When any of them is turned off, for example, axes, its `grob` will be set as a `nullGrob` – a `NULL` graphical object that generates nothing, as in

```
> grid::setGrob(
+   gTree = lg,
+   gPath = "axes",
+   newGrob = grid::nullGrob(name = "axes")
+ )
```

When any of them is turned on, the `nullGrob` is switched back to the `grob` which draws corresponding graphical elements (e.g., `nullGrob()` → `xaxisGrob()`).

Note that, once the labels or scales are turned on (or off), the margins (label margins and scales margins) of the plot view port will be adjusted simultaneously.

5.3.3 Selection

To select, we need to query the elements' indices falling inside the brushing region.

When a brushing area is constructed, the plot will send coordinates of the region to the `observer`. A named list with `xmin`, `xmax`, `ymin`, and `ymax` is returned to locate the brushing area (scaled to a [0, 1] plate). To match the scales of the brushing region, we need to transform the coordinates of the current data viewport to [0, 1]. As different models have different graphical elements, the selection strategies differ from one to another.

Model `l_plot` and `l_graph`

As the brushing region and the data viewport have the same scales, in a `loon` scatterplot, we can easily identify points within the region.

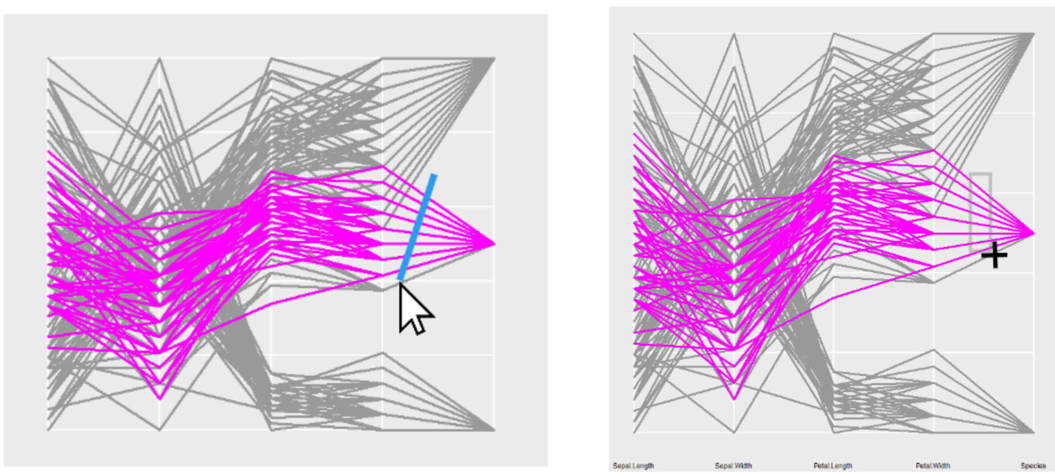
Model `l_hist`

The elements of an `l_hist` widget are partitioned into a separate group for each color. To determine which elements are selected, the following process is applied: go through all active bins. For each bin, determine whether any corner of the brushing area is inside this bin area or any corner of this bin is inside the brushing area. If any of these happens, this bin will be highlighted and all elements in this bin will be marked as "selected".

Model `l_serialaxes`

In `loon`, an `l_serialaxes` widget can be highlighted by a one dimensional selection tool "crosser". One can draw a single segment and all the elements (i.e., lines) intersecting with this segment are highlighted, as shown in Figure 5.11 (a).

However, this one dimensional selection tool is not available in `shiny` yet. In contrast, a two dimensional rectangular region is used, as shown in Figure 5.11 (b).



(a) (b)

Figure 5.11: Selection in `loon` (left) and `loon.shiny` (right) of an `l_serialaxes` widget

When determining whether one segment intersects with another segment, typically, one of the three scenarios could happen:

- the segments do not intersect;
- there is a unique intersection point;
- the intersection is another segment.

Each element is composed of k end-to-end segments. To identify if an element is selected, we need to check if any segments (of the k end-to-end) intersect with any side of the brushing region. Therefore, it is relatively hard to find selected elements directly.

To simplify the process, we turn each element into a sequence of points. If any point falls into the region, this element will be marked as "selected".

5.3.4 Linking

In a `loon.shiny` app, the `linkedStates` such as color and size, are set via the checkbox (see Figure 5.7). Suppose plot 'A' belongs to the `linkingGroup` 'groupA' and the state 'S' (e.g., color) is a linked state, any changes to 'S' in plot 'A' will lead to an update on a list called `linkingInfo` (inside the `observer`). Then, the rest of the plots in 'groupA' will be checked and their state 'S' will be changed accordingly.

Once the `linkingGroup` is set as `none` for plot 'A', this plot will disconnect from 'groupA'. Any modifications applied to this plot will not affect the `linkingInfo`.

Suppose this plot now joins back to 'groupA', nothing happens – neither this plot's states or other plots' states are modified. That is because, this `linkingGroup` select-box `input` is wrapped inside the function `isolate()` so that the modification of this widget will not trigger an evaluation of the `observer`. This select-box input is waiting for a command, whether to `push` its states to other linked plots or `pull` states from other linked plots.

If `pull`, this plot's linked states will be changed by the `linkingInfo`; else, the linked states of this plot remain, but the linked states of the `linkingInfo` will be updated based upon this plot's states. Then every member in 'groupA' will be modified by the new `linkingInfo`.

5.3.5 States Modification

Model `l_plot` and Model `l_serialaxes`

In the model `l_plot` and model `l_serialaxes`, the elements (points and lines) are independent of each other, so that the state modification of an individual element would not affect other elements. Note that for each evaluation of the `observer`, only the states of those selected points/segments will be modified.

To modify colors, other than clicking the 21 listed color buttons, one can also use the color picker widget to retrieve richer colors, as shown in Figure 5.9. Note that, the color picker widget is wrapped inside the function `isolate()` as well.

The "deactivate" button and "reactivative" button are used to switch element state between invisible and visible mode. When the "deactivate" button is clicked, the graphical function used to draw the visuals (e.g., `pointsGrob()`) is replaced by the function `grob()`, for example,

```
> for (i in id) {
+   newGrob$children[[i]] <-
+     do.call(grid::grob,
+             getGrobArgs(newGrob$children[[i]]))
+ }
```

where `id` is the indices of the deactivated elements and `getGrobArgs()` is used to record all the arguments of this `grob`. Neither `grob()` nor `nullGrob()` produces any geometric displays, however, the `grob()` can accommodate all arguments as a basic creator. For example:

```
> grid::grob(arrow = grid::arrow(type="open"))
grob[GRID.grob.6]
> grid::nullGrob(arrow = grid::arrow(type="open"))
Error in grid::nullGrob(arrow = grid::arrow(type = "open")) :
  unused argument (arrow = grid::arrow(type = "open"))
```

Storing all attributes in `grob()` is helpful for reactivation, through which all elements will be visible again. Once reactivated, all elements will be gone through and the function `grob()` will be replaced by the graphical function which draws the original visuals (e.g., `grob()` → `pointsGrob()`).

Model `l_hist`

Unlike a scatterplot or a serialaxes plot, any states modification of attributes may cause a re-binning of `x` data so that the display of a histogram is changed accordingly. Consequently, in each evaluation of an `l_hist` `loon.shiny` app, the coordinates of all bins are recalculated.

5.4 Dynamic `ui`

`Ui` is to control the layout specification of an app. Sometimes, a change to a `loon.shiny` app may not only update the output graphics, but also update its input interface, in which case, the messages of the new changes will be collected and sent to `update**()` (e.g., updateSelectInput()) functions.

5.4.1 Update Slider Bars

Two slider bars are used to control the `x` limit and `y` limit to realize panning or zooming. In a `loon.shiny` app, the slider bars (values or labels) can be affected by different scenarios: whether the coordinates are swapped or whether the graphics are scaled to the "plot" region, the "world" region or the "selected" region.

The "swap" checkbox is to flip the Cartesian coordinates. Once the swap checkbox is toggled on, the function `shiny::updateSliderInput()` gets activated. Within this function, we swap the labels of these two bars, "xlim" to "ylim" and "ylim" to "xlim".

Suppose any of the display region is scaled to "plot", "world" or "selected", the values of the slider bars will be updated immediately; then, the `server` function is re-evaluated to change the display view port.

5.4.2 Update "by color"

In a `loon` inspector, the channel "by color" shows a list of buttons (e.g., see Figure 5.7 left), each represents a unique color in the plot. By selecting a certain color, all elements in that color will be highlighted. When the element colors are changed in the graphics, the buttons will be updated accordingly, either in the color of each button widget or in the number of buttons.

In `shiny`, the dynamic `ui` system only supports the update of the existing button widgets (e.g., change the label or icon of an existing action button on the client), instead of creating new button widgets. In contrast, in a `loon.shiny` app, all color buttons are replaced by a checkbox-group `input` as the checkbox can be updated in time via the function `updateCheckboxGroupInput()`. Besides, here are some other benefits of this design:

- color names are displayed beside each check box (e.g., see Figure 5.7 right);
- to select multiple groups (each group is in one color), one can simply toggle multiple choices through the checkbox-group input without holding the `<shift>` key (`loon` inspector).

5.5 Limitations

5.5.1 Computing Speed

Most code in `loon.shiny` runs in R but R is not a fast language (Wickham, 2014). Based on our test on a machine with i7-6700HQ CPU, GeForce GTX 970 Desktop Graphics Cards, as the number of observations reaches 2000, the interactivity is not satisfying.

Right now, in the `server` function, only one `observer` is used so that in each execution, all computations in this object will be evaluated. Even though many logical blocks are built to avoid unnecessary runs, it is still not perfectly efficient. The only reason we still stay with this design is that: most of the time, many interactions are highly related. Putting everything in one can make the logic easier to be followed.

One possible way to improve the performance of a `loon.shiny` app is to break the single `observer` design down into several pieces and each piece is in charge of one or several functionalities, for example,

```
>  server <- function(input, output, session) {
+    ...
+    shiny::observe({
+      # non-data element states modification
+      ...
+    })
+    shiny::observe({
+      # selection modification
+      ...
+    })
+    ...
+  }
```

5.5.2 Scales Control

In `loon`, panning and zooming are realized by immediate manipulation with a mouse and the modifier key (`<shift>` or `<ctrl>`). To zoom in and out a plot, one can scroll the mouse wheel. To pan a plot, select the plot interior with the right (or secondary) mouse button and move the mouse (with the button still down). The direction of panning can be constrained by holding down the named modifier keys (`<shift>` or `<ctrl>`) while panning.

In shiny, the function plotOutput() (used to display the graphic) cannot trace scrolling yet. Consequently, two sliders bars are provided in a loon.shiny app. The problem is that slider bars have limits but scrolling is unlimited (a loon widget provides an infinite space).

5.5.3 Design of Plot Window and Inspector

In a loon plot, both the display window and the inspector can be dragged, resized or closed. However, in a loon.shiny app, they are designed differently.

- Dragging: in a loon.shiny web app, the user interface can be dragged but the main plot cannot. That is because panels in a shiny app are dragged by holding the left mouse button. However, if we set the main plot drag-able, there was no way for users to sweep (or brush). So far, we cannot find a way to realize both dragging and sweeping (or brushing) simultaneously on the main plot.
- Resizing: a loon window is a fluid panel. However, in a loon.shiny app, using a fluid panel may make the display look bad. Thus, a fixed panel is used.
- Closing: in loon, closing a loon widget results a true termination; closing a loon inspector results in creating a new loon inspector as soon as a display reporting to the loon inspector receives a mouse gesture input or window focus event (Waddell, 2016). In a loon.shiny app, the interface or plot output cannot be closed until we end the running session.

5.5.4 Mouse Gestures

In loon, a <single click> can deselect elements or activate the window focus. However, in loon.shiny, if a <single click> mouse gesture was implemented, suppose one wanted to use sweeping (press down the left button and sweep out an area) to select points, then, the server function would be executed twice – one for the click (at the moment a user presses down the left mouse button) and one for the sweeping.

To reduce the duplicated evaluation, in loon.shiny, a <single click> mouse gesture is replaced by a <double click> mouse gesture; therefore, to deselect highlighted points or switch the window focus, one has to double click on the window. Though, compared with the <single click>, <double click> is less natural, the efficiency of the app improves.

5.5.5 Event Bindings

The reactive logic of a `loon.shiny` app is curated and designed based on `loon`'s default logic specifications. `Loon` also provides another framework called event bindings which offers the functionality of binding code to specific event types. The following code shows a state event bindings example,

```
> p <- l_plot(iris)
> l_bind_state(target = p,
+              event = "selected",
+              callback = function() {
+                sel <- p['selected']
+                p['size'][sel] <- 12
+                p['color'][sel] <- "firebrick"
+              })
```

Here, when one selects a point, more than highlighted, the size of it gets bigger. Once the selected point is downlighted, the color of this point is changed to firebrick. Currently, the binding functionality is not available in a `loon.shiny` app.

5.6 Summary

The `shiny` package is a graphical system whose principal structures are a user interface, `ui`, and a `server` function. Any R graphic created with either the base graphics (by the `graphics` package) or the `grid`-based graphics can be displayed in a web browser using `shiny`. Since `ggplot2` is built on `grid`, `ggplot`s could also be used in `shiny`. A bridge between the graphical system `loon` and the graphical system `shiny` can therefore rely on a bridge between `loon` any of the base graphical systems in R, including `ggplot2`.

In this chapter, the `loon.shiny` bridge was built via the `loonGrob` bridge to produce the plots in `shiny`. The package `loon.shiny` then extends `shiny` by adding the functionality of the `loon` inspector and other interactive features of `loon` (e.g., panning, zooming, brushing, linked plots).

To transform `loon` plots into interactive plots in a `shiny` web app, the visual display of each `loon` plot is first mapped to the visual display of a `grid` graphic (using `loonGrob`), then, the control features (i.e., input) such as slider-bars, buttons and checkbox, provided by `shiny`, are added to the app to reproduce the floating palette interface – `loon` inspector. To allow the interface to interact with the `grid` graphic, the logic specifications in the `server` function were edited so that changes on the `ui` can update the output plot, just as (as similar as possible) what `loon` does.

Again, in place of the `loonGrob` bridge, a `loon.shiny` bridge could have been constructed using the `loon.ggplot` bridge of Chapter 3. The `ui` design would not change but the `server` would have to be changed to effect changes in a `ggplot2` visual structure instead of a `grid` structure (i.e., visual display). A future `loon.shiny` bridge might accommodate both `grid` and `ggplot2` packages by redesigning the `server` function to switch between the `loonGrob` and `loon.ggplot` bridges by user choice.

Chapter 6

Loon.tourr

6.1 Introduction

A tour is a motion graphic designed to study the joint distribution of multivariate data (Asimov, 1985)(Buja and Asimov, 1986). A sequence of low-dimensional projections is created by a high dimensional data set and tours are thus used to find interesting projections. In mathematics, $\boldsymbol{X}_{n \times p}$ represents the original data set; $\boldsymbol{P}_{p \times d}$ is the matrix of projection vectors and $d < p$.

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{P}$$

where $\boldsymbol{Y}$ is the lower dimensional sub-space.

The tour was first implemented in the software `Dataviewer` (Buja et al., 1986)(Hurley, 1987)(Buja et al., 1987) in Symbolics Lisp machine. A smoothly moving scatterplot could be created to visualize the tour paths. Swayne et al. (1998) implemented the software `XGobi` in the X Window System, providing portability across a wide variety of workstations (i.e., X terminals, personal computers, even across a network). The software `GGobi` (Swayne et al., 2001)(Cook and Swayne, 2007) redesigned and extended its ancestor `XGobi`, can be embedded in other software, like R. The package `rggobi` (Wickham et al., 2006) is an R interface of `GGobi`, however, it has been removed from the CRAN (can be accessed in archive).

The package `tourr` (Wickham et al., 2011), inherited most functionality from `rggobi`, implements geodesic interpolation and provides various tour generation functions (e.g., grand tour, guided tour, etc.) in R. Unlike earlier tour implementations (`rggobi`), no interactive manipulation of the plot elements in `tourr` is allowed.

The `loon` package is a toolkit that enables highly interactive data visualization. The package `loon.tourr` (see https://great-northern-diver.github.io/loon.tourr/) (Xu and Oldford, 2021) adds full functionality of `loon`'s interactive graphics to `tourr`. For example, in `loon.tourr`, interactive selection, coloring, and deactivating of points in a tour display and also linking that display to any other `loon` plots are allowed. Interesting projections discovered during the tour can be accessed at any point in the tour. In addition, random tours displaying more than 2 dimensions are also accommodated in the parallel or radial coordinate system.

This chapter begins with an introduction of the data structure of the `l_tour` object (not a widget) and `l_tour_compound` object. Then, we discuss the specifications of a `loon` tour object, such as setting different tour techniques (e.g., guided tour), projecting the original data onto 1D, 2D, or higher than 2D subspace and adding interactive visual layers. This chapter closes with a summary of the package `loon.tourr`.

6.2 Tour Object

An `l_tour` interface is composed of a `loon` widget and a GUI system (i.e., a slider bar to control the tour, a refresh button and scaling radio buttons). Figure 6.1 shows a basic 2D random tour in `loon`.

```
> library(loon.tourr)
> ltp <- l_tour(iris[, -5],
+               color = iris$Species)
```

These plots are interactive for users to pan, zoom or select. By default, there are 30 random projections and 40 steps between each two. Thus, the number of matrices of projection vectors becomes $30 \times 40 + 1$ (start position) in total. To navigate the tour, scroll the rightmost sliderbar to transform the projection from one to the other (e.g., in Figure 6.1, from left to right). Unfortunately, if none of the projections is interesting, press the "refresh" button at the left-bottom corner to generate new random tours. At the bottom of the plot, a menu of scaling methods is available for selection, `data`, `variable`, `observation` and `sphere`. The projected space can be rewritten based on the scaling as

$$\boldsymbol{Y} = s(\boldsymbol{X})\boldsymbol{P}$$

The first three scaling methods have been introduced in Subsection 4.3.2. The last one `sphere` is defined as

$$s(\boldsymbol{X}) = \boldsymbol{X}^{\star}\boldsymbol{V}$$

(a) (b)

Figure 6.1: Basic `loon` tour for data `iris`.

where $\boldsymbol{X}^{\star} = (\boldsymbol{I} - \frac{1}{n}\boldsymbol{1}\boldsymbol{1}^{\mathsf{T}})\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}$. $\boldsymbol{U}$ is an $n \times p$ semi-unitary matrix and $\boldsymbol{V}$ is an $p \times p$ semi-unitary matrix, such that $\boldsymbol{U}^{\mathsf{T}}\boldsymbol{U} = \boldsymbol{V}^{\mathsf{T}}\boldsymbol{V} = \boldsymbol{I}$ and $\boldsymbol{D}$ is a diagonal matrix.

With pre-set tour paths, the interactivity of the GUI system is realized by customizing the `command` option in `tk`. To modify the interface, a callback function (defined in the `command`) is executed. For example, suppose one drags the bar, the callback function will be evaluated and the corresponding projection matrix will be located ($\boldsymbol{P}_{\text{new}}$). Based on this projection, the coordinates ($\boldsymbol{Y}_{\text{old}} = \boldsymbol{X}\boldsymbol{P}_{\text{old}}$) will then be transformed to the new one ($\boldsymbol{Y}_{\text{new}} = \boldsymbol{X}\boldsymbol{P}_{\text{new}}$).

An `l_tour` object is not a `loon` widget,

```
> loon::l_isLoonWidget(ltp)
[1] FALSE
```

but it inherits some functionality of `loon`. For example, one can query the aesthetic attributes of an `l_tour` object by ‘‘`[`’’ or `l_cget()` and modify its plotting states by ‘‘`[<-`’’ or `l_configure()`. However, as not all `loon` functionality is inherited by an `l_tour` object, one cannot simply add a geometric layer, as in

```
> l_layer_line(l_tour, ...)
```

or set non-primitive glyphs, as in

```
> l_add_glyph_polygon(l_tour, ...)
```

To get a `loon` widget, the function `l_getPlots()` can be used, as in

```
> p <- l_getPlots(ltp)
```

The returned object is a true `loon` widget,

```
> loon::l_isLoonWidget(p)
[1] TRUE
```

Then, one can add geometric layers or set non-primitive glyphs via the returned `loon` widget (i.e., `p`; see Section 6.4).

An `l_tour` object stores a matrix of projection vectors reflecting the current projections. For example, the matrix of projection vectors in Figure 6.1 (b) can be accessed as in

```
> ltp['projection']
           [,1]        [,2]
[1,]  0.2101051 -0.54962269
[2,] -0.6903045  0.41549149
[3,]  0.4055277 -0.01074593
[4,]  0.5611442  0.72468355
```

Note that the state `projection` cannot be configured. For example, `ltp['projection'] <- **` is illegal.

In `loon`, the `l_getFromPath()` function is often used to create a `loon` widget handle from the path name (e.g., `.l0.plot`). However, simply calling `l_getFromPath()` only produces a `loon` widget instead of an `l_tour` object (expected result). To convert the `loon` widget to an `l_tour` object, one has to fire the callback function. Once the callback function gets activated, all elements in the environment in which the function was called will be scanned. If the unique path name was detected, the projection matrix would be assigned to that `loon` widget which would then become an `l_tour` object. The simplest way to fire the callback function is to manipulate the tour GUI (e.g., scroll the bar, refresh the sequences, apply a different scaling method, etc.).

In an `l_tour_compound` object, all plots have the same matrix of projection vectors. Currently, `loon.tourr` provides two `l_tour_compound` objects, a facet tour object (by setting the argument `by` in the function `l_tour()`) and a pairs tour object (by the function `l_tour_pairs()`), as shown in Figure 6.2.

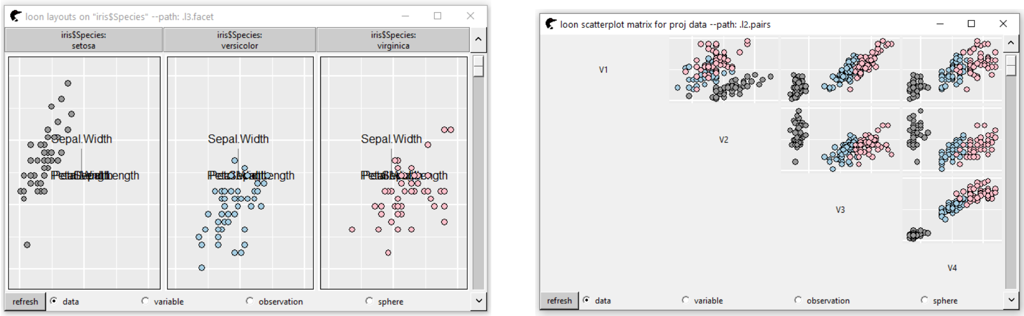


Figure 6.2: The left figure is a facet tour that each panel displays a species of iris. The right one is a pairs tour. Each scatterplot in the matrix visualizes the relationship between a pair of variables in $\boldsymbol{Y}$

6.3 Tour Specifications

6.3.1 Tour Techniques

In `tourr`, several tour techniques are introduced to better explore a low-dimensional space:

- Grand Tour: the next target basis is selected randomly. The default tour mechanism in `loon.tourr` is `grand_tour(d = 2)`, as in

  ```
  > l_tour(data, tour_path = grand_tour(d = 2))
  ```

  where d represents the lower sub-space dimension and will be discussed more in Subsection 6.3.2.

- Guided Tour: the next target basis is selected by a criterion function $g(\boldsymbol{Y})$ that specifies some features of interest.

  $$\arg\max g(\boldsymbol{Y}), \forall \boldsymbol{P}$$

  where $\boldsymbol{Y} = [\mathbf{y}_1^\mathsf{T}, ..., \mathbf{y}_n^\mathsf{T}]^\mathsf{T} = \boldsymbol{X}\boldsymbol{P}$, $\mathbf{y}_j$ is a $p \times 1$ vector. The package `tourr` introduces several projection pursuit indexes such as "holes" and "central mass" (Cook et al., 1993) which are defined as follows:

– Holes:

$$g(\boldsymbol{Y}) = \frac{1 - \frac{1}{n}\sum_{i=1}^{n} \exp(-\frac{1}{2}\mathbf{y}_i^\intercal \mathbf{y}_i)}{1 - \exp\left(-\frac{p}{2}\right)}$$

In `loon.tourr`, it is implemented as in

```
> l_tour(data, tour_path = guided_tour(holes(), d = 2L))
```

– Central Mass:

$$g(\boldsymbol{Y}) = \frac{\frac{1}{n}\sum_{i=1}^{n} \exp(-\frac{1}{2}\mathbf{y}_i^\intercal \mathbf{y}_i) - \exp\left(-\frac{p}{2}\right)}{1 - \exp\left(-\frac{p}{2}\right)}$$

It is implemented as in

```
> l_tour(data, tour_path = guided_tour(cmass(), d = 2L))
```

The "holes" and "central mass" indexes are inspired from the normal density function. See Cook and Swayne (2007) for more details.

Additionally, the package `tourr` also provides other tour paths. For example, in a frozen tour (`frozen_tour()`), one variable is designated as the manipulation variable so that the projection coefficient is fixed; a local tour (`local_tour()`) alternates between the starting position and a nearby random projection.

Besides, a slicing tour (or a section tour) is introduced in `tourr`. In this tour, only the projected points whose orthogonal distances are smaller than a cutoff value *st* (slice thickness) are highlighted (Laa et al., 2020). In `loon.tourr`, its application is slightly different. In interactive `loon` plots, several linked plots may share the same `selected` state. In a slicing tour, rather than altering selection, points whose orthogonal distances within the range will be visible and others become invisible. A slicing tour can be approached by,

```
> l_tour(data, slicing = TRUE, slicingDistance = st)
```

6.3.2 Lower Sub-space Dimensions

Other than a defaulted 2D scatterplot, other dimensional sub-spaces are also implemented by modifying the parameter `d` in functions `**_tour()` (e.g., `grand_tour()`). The 1D sub-space is embedded in a histogram and the higher dimensional sub-space is embedded in a serialaxes plot (i.e., parallel and radial). For example, Figure 6.3 gives examples of 1D and 4D tour plots.

```
> # 1D tour
> lth <- l_tour(iris[, -5],
+               color = iris$Species,
+               tour_path = grand_tour(1))
> # 4D tour
> lts <- l_tour(iris[, -5],
+               color = iris$Species,
+               axesLayout = "parallel",
+               tour_path = grand_tour(4))
```

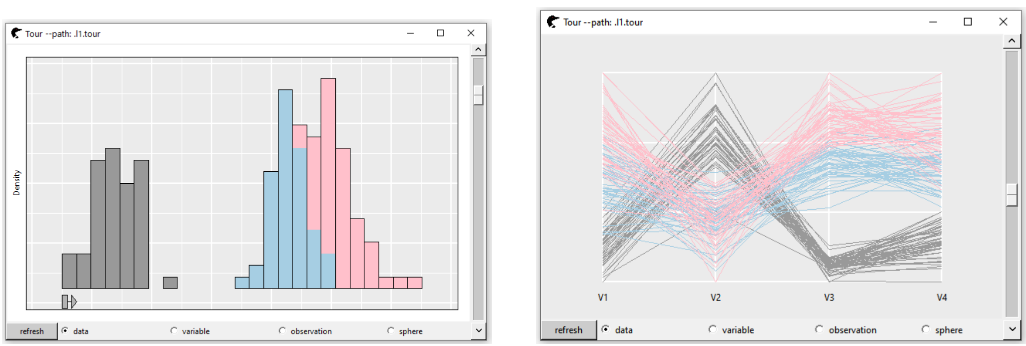


Figure 6.3: One dimensional tour and four dimensional tour.

An Andrews (1972) tour plot can be created by setting the state `andrews = TRUE`, as in (shown in Figure 6.4),

```
> lts['andrews'] <- TRUE
```

6.4 Layers in Tour

Sometimes, layer visuals could provide additional information in the search for interesting patterns.

In geometry, the convex hull of a planar set is the minimum-area convex polygon containing the planar set. Figure 6.5 shows a convex hull layer (by the algorithm `CONVEX` Eddy, 1977) on the tour plot `ltp`.

```
> l_layer_hull(ltp, group = iris$Species)
```

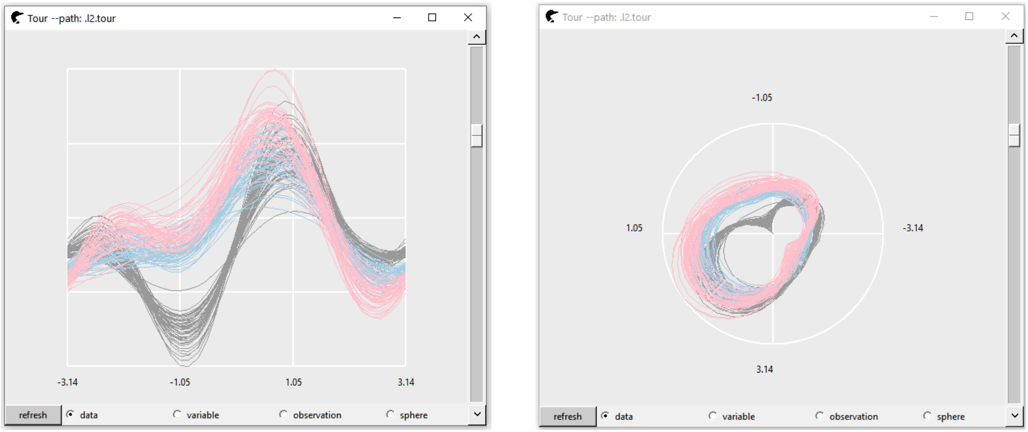


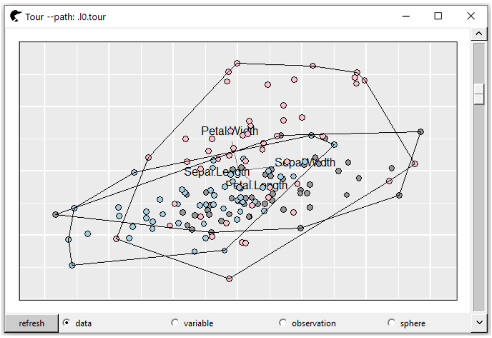
Figure 6.4: Grand tour with Fourier transformation. The left one is embedded in a parallel coordinate system and the right one is embedded in a radial coordinate system.

Each species is an individual set and each hull is constructed by the vertices of each species. By navigating the tour (e.g., drag the bar, modify the scaling methods, etc.), the hull is recalculated immediately based on the new projection. The hull layer is extremely useful in determining clusters. When the hulls barely overlap with each other, the projection could be an interesting one. For example, the Figure 6.5 (a) is not an interesting projection because all three clusters are completely overlapped – none of the groups could be easily distinguished from others. In contrast, Figure 6.5 (b) could be considered an interesting projection. Species "virginica" is entirely isolated. Species "setosa" and "versicolor" are well-isolated as well.

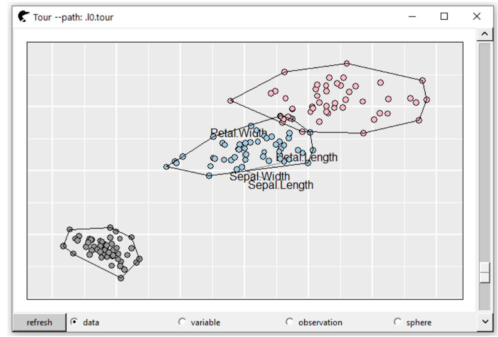
Besides, inspired by `animate_density2D()` and `animate_trails()` in `tourr`, a density 2D layer and a points trail layer are provided in `loon.tourr`. A density 2D layer deals with the overplotting issues, as shown in Figure 6.6 (a). In a points trail layer, a trail appears behind every single point where the angle measures the direction and the length measures the size of the step, as shown in Figure 6.6 (b).

The function `l_layer_callback()`, controlling the interactivity of tour layers, is the backbone of the hull layer, the density 2D layer and the points trail layer. It is a generic method (see Subsection 3.3.3) so that one can customize this function to realize the interactivity of any `loon` layers.

For example, one can draw a 1D density layer visual on top of the histogram `lth`, as shown in Figure 6.7 (a).

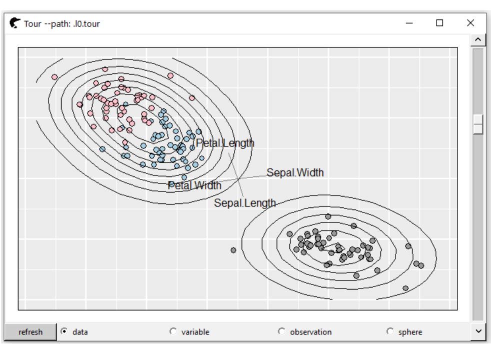


(a)

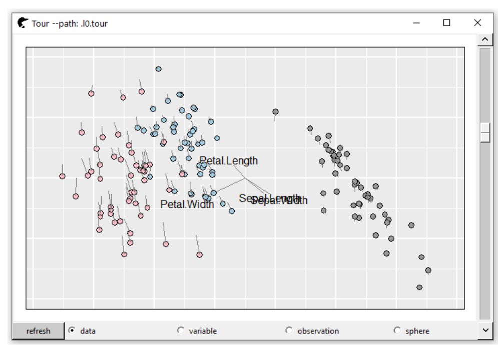


(b)

Figure 6.5: The convex hull layer for data `iris`. With the layer hull, each cluster is easier to be distinguished. All three species are clearly separated in (b).



(a)

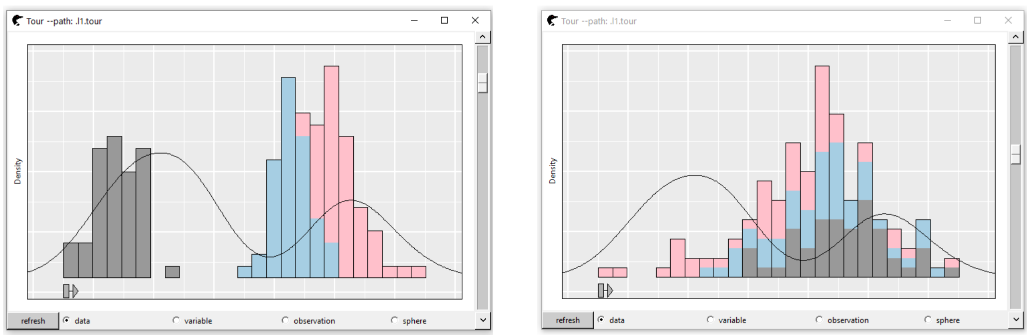


(b)

Figure 6.6: A density 2D layer and a trail layer

```
> # Note that 'lth' is not a loon widget
> # one can query the loon widget by calling 'l_getPlots()'
> l <- l_layer(l_getPlots(lth),
+              stats::density(lth['x']),
+              label = "density1D")
```

However, this layer will not reflect the change along with the tour. As the tour is being navigated, the density curve is not updated, as shown in Figure 6.7 (b).



(a) (b)

Figure 6.7: One can add any layers to an `l_tour` object. Nevertheless, if the function `l_layer_callback()` was not set, the layer would not be updated along with the tour, as shown in (b)

The reason is that the callback function of this density 1D layer does not exist. To update the 1D layer instantaneously, one should create an layer 1D callback function, such as,

```
> l_layer_callback.density1D <- function(target, layer, ...) {
+
+     layer <- loon::l_create_handle(c(l_getPlots(target), layer))
+     den <- stats::density(target['x'])
+
+     loon::l_configure(layer,
+                       x = den$x,
+                       y = den$y)
+ }
```

Within this function, the density is computed through the new basis so that the density layer can be configured simultaneously along with the changes of the tour, as shown in Figure 6.8.

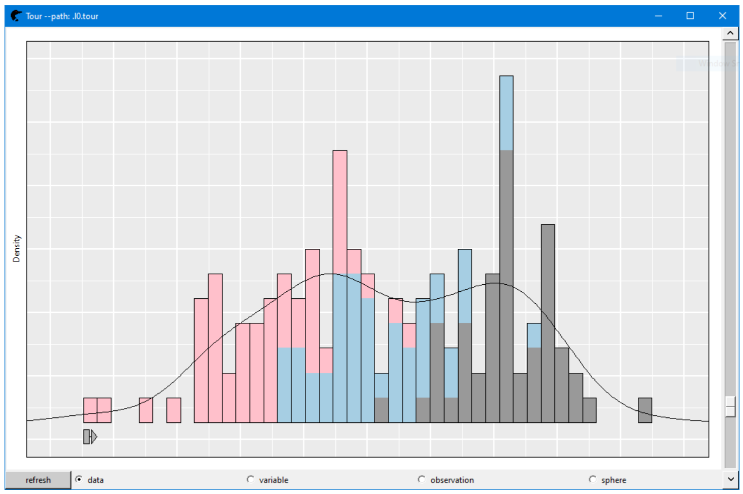


Figure 6.8: After executing the function `l_layer_callback.density1D()`, density 1D layer is updated as the tour is being navigated.

The function `l_layer_callback()` is a generic function. In the callback procedure, to make sure the method dispatching to the function `l_layer_callback.density1D()`, one has to set the label of the layer as *density1D*.

6.5 Summary

The package `tourr` extends the base graphics system in R to provide kinematic graphics following any computed tour. The package `loon.tourr` extends `loon` to also take advantage of tours computed by `tourr`. However, the kinematic tours in `loon.tourr` are now interactive and integrated with any other plots in `loon`. This makes them more powerful: 1. analysts can directly manipulate on the plot in a tour process, such as selecting, linking, and querying; 2. any interesting matrix of projection vectors can be queried at any moment; 3. layers can be added and automated with the tour to provide richer information.

The visual displays of `tourr` *are* mapped to visual displays in `loon` via `loon.tourr`. So, in this sense, `loon.tourr` *is* a bridge from `tourr` to `loon`, in that it maps visual displays.

However, by relying only on the projection information, no *graphical element* in `tourr` is *actually* transformed to a *graphical element* in `loon`. So, as formally defined in Chapter 1, `loon.tourr` is not *formally* a bridge.

Chapter 7

Discussion and Further Work

This thesis introduced the idea of a bridge between graphical systems in R. Three specific bridges and their design were discussed in detail in previous chapters (i.e., `loonGrob`, `loon.ggplot`, `loon.shiny`). In this chapter, we focus on the benefits and some limitations of a bridge in general. Some further work is also proposed.

7.1 Bridge

A bridge was introduced in Chapter 1, as a mapping from one graphical system, $\mathcal{G}$, to another, $\mathcal{K}$; elements $g_1, ..., g_n \in \mathcal{G}$ are mapped to elements $k_1, ..., k_m \in \mathcal{K}$. The elements g_i and k_j are imagined to be either a visual display or a visual structure.

7.1.1 On Elements

When the elements g_i and k_j are thought of as visual displays, success is measured by how closely the display k_j visually resembles g_i. When the elements g_i and k_j are thought of as visual structures, success is measured by how well the full set of information of g_i is incorporated in k_j, including matching levels of abstraction.

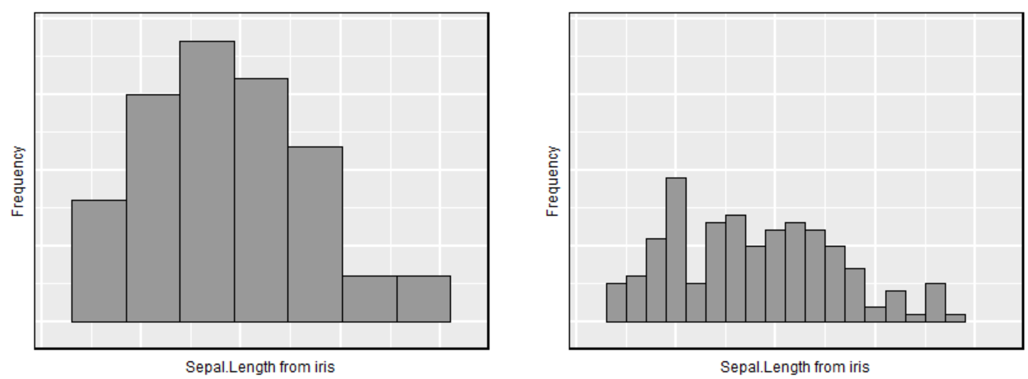
When visual structures are being mapped, analysts should ideally be able to continue exploring the data in the graphical system $\mathcal{K}$. This aligns with our initial intention of the bridge − allowing analysts to use more than one graphical systems in different aspects of data analysis.

For example, the following code shows how to map a visual structure from a loon histogram to a ggplot object.

```
> library(loon.ggplot)
> h <- l_hist(iris)
> hg1 <- loon2ggplot(h)
> hg1
```

The output of hg1 is shown in Figure 7.1 (a). The hg1 is constructed by the function geom_histogram() in ggplot2. In geom_histogram(), the data structure (e.g., bin width and bin color) is inherited from the loon l_hist widget. The hg1 object is editable due to the nature of the visual structure mapping, as in the following code (shown in Figure 7.1 b):

```
> hg1$layers[[1]]$stat_params$binwidth <- 0.2
> hg1
```



(a) (b)

Figure 7.1: Transform a histogram from loon to ggplot2

In contrast, when mapping only visual displays, the changes of k_j are often very limited (e.g., aesthetic attributes could be changed but statistical modifications not allowed). For example, the code below maps a visual display from a loon histogram widget to that of a ggplot object (the display is identical to Figure 7.1 a).

```
> hg2 <- loon2ggplot(h, asAes = FALSE)
> hg2
```

In order to build hg2, we treat the input histogram h as stacked rectangles; therefore, the function geom_rect() is adopted for the bridge loon.ggplot. The locations of the four corners of each bin (i.e., xmax, xmin, ymax, ymin) are extracted and set in geom_rect(). Even though hg1 and hg2 have exactly the same output, the fact is that they are very different objects. No histogram parameters (e.g., bin width and bin origin) can be set in hg2 at all.

7.1.2 On the Level of Abstraction

Usually, mapping a visual structure provides more functionality for the elements than mapping a visual display, and so is generally preferred. However, mapping visual structures can be difficult, if not impossible, when the levels of abstraction do not match between systems.

If the abstraction level matches (i.e., high-level to high-level, low-level to low-level), the bridge is easy to build (e.g., a loon l_hist widget and a ggplot2 geom_histogram object). If the abstraction level does not match (e.g., $f(g_i) \notin \mathcal{K}$), the bridge is relatively hard to create (e.g., loon to grid).

Usually when elements at the level of abstraction do not match, we have two solutions: 1., extend $\mathcal{K}$ to create corresponding high-level element k_j and then match g_i (e.g., ggmulti); 2., break g_i down to several low-level elements g_l (e.g., loonGrob). The former one is still a visual structure mapping, but the later one is a visual display mapping.

7.1.3 Zenplots Revisited

In Section 1.5, we noted that some developers solve the problem of user preference for different graphical systems in R by implementing their visualization using more than one graphical systems. The package zenplots (Hofert and Oldford, 2019) accommodates three graphical systems at the same time, graphics, grid and loon. With the proper bridge, however, this is no longer necessary.

Suppose users want to visualize high dimensional data using the zigzag layout of zenplots but prefer a graphical system ggplot2. No need to implement a ggplot2 set of functions for zenplots when the loon.ggplot is available. Instead, with a bridge, users can create a zenplot using pkg = "loon", then use loon.ggplot to return a patchwork object (extensions of ggplot2).

For example, the following code shows the construction of an interactive `zenplot` with 2D scatterplots and 1D histograms for the `iris` data (see earlier as a screenshot in Figure 1.9).

```
> library(zenplots)
> zp <- zenplot(iris[, -5],
+               plot1d = "hist",
+               plot2d = "points",
+               pkg = "loon")
```

Now, the bridge function `loon.ggplot()` is called on `zp` to produce a `ggplot` (i.e., a `patchwork`) object, as shown in Figure 7.2.

```
> loon.ggplot(zp)
```

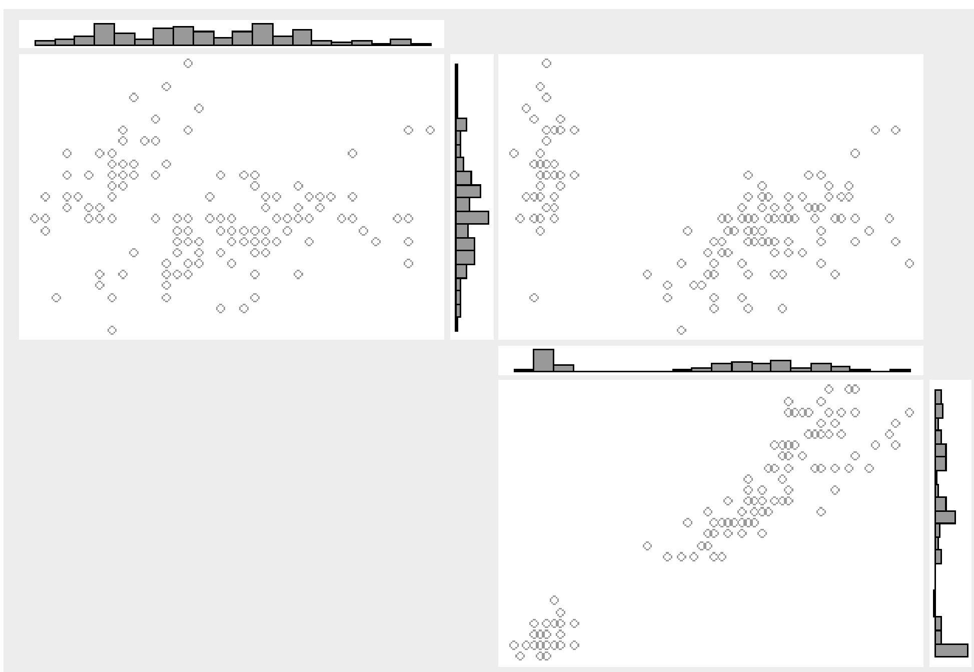


Figure 7.2: A `ggplot` version of Figure 1.9 via the bridge `loon.ggplot`.

This `patchwork` object could also be transformed back to an interactive compound `loon` plot via the `loon.ggplot` bridge (after version 1.3.0). Note that an `l_compound` object would be returned, not a `zenplot` object.

7.2 Extension and Suite Connection

A two-way bridge is defined by graphical systems $\mathcal{G}$ and $\mathcal{K}$ with the functions f and h, where f is a function to transform elements in $\mathcal{G}$ to elements in $\mathcal{K}$ and h does the reverse. By both f and h, the functionalities of $\mathcal{G}$ and $\mathcal{K}$ might be shared with each other. Furthermore, the suite of $\mathcal{G}$ (related packages of $\mathcal{G}$) and the suite of $\mathcal{K}$ can be connected by this two-way bridge.

For example, `loon.ggplot` extends the functionalities of both `loon` and `ggplot2`. Additionally, it connects the suite of `loon` (e.g., `loon.shiny`) and the suite of `ggplot2` (e.g., `gganimate`).

7.2.1 Extension

The package `ggplot2` extends the graphical API (Application Programming Interface) of the package `loon` (which uses a traditional graphical API). Similar to the base `graphics` package, a collection of commonly used graphical functions are provided, such as the function `l_plot()` for drawing a scatterplot, as shown in following code,

```
> # scatterplot
> l0 <- l_plot(x = iris$Sepal.Length, y = iris$Sepal.Width)
```

It is fairly easy for new users to get started with this `API` as it is very intuitive and only requires minimum typing – only one function `l_plot()` and two coordinate arguments `x` and `y`. However, this design requires a lot of manual work to add complexity (e.g., plot points in a radial axes).

Alternatively, based on a grammar of graphics, the package `ggplot2` provides a new grammar-based API. With the bridge `loon.ggplot`, users are able to create the same `loon` scatterplot by this new design, as in

```
> # grammar based
> l1 <- l_ggplot(data = iris,
+            mapping = aes(x = Sepal.Length,
+                          y = Sepal.Width)) +
+   geom_point()
> l1
```

The benefit of this design is to add complexity (and equally easy to take the complexity away) on an existing plot easily. For example, for a scatterplot that has been rendered into the Cartesian coordinate system, having it rendered again into the radial coordinate system can be simply achieved by adding the function `coord_polar()` to the existing `l1` (i.e., `l1 + coord_polar()`). However, compared with the traditional API, it needs more typing. Arguments "data" and "mapping" in the main function `l_ggplot()` have to be defined. In addition, the layer `geom_point()` is also required for the purpose of drawing points.

On the other hand, the package `loon` extends the implementation of a grammar of graphics to a grammar of interactive graphics (allowing interactivity in `ggplot2`). With the bridge `loon.ggplot`, an interactive plot can be built by adding an `interactivity` component, as in

```
> ggplot(data = iris,
+        mapping = aes(x = Sepal.Length, y = Sepal.Width)) +
+   geom_point() +
+   selection(selectBy = "sweeping")
```

In the expression above, the `ggplot` object "understands" that the analyst wants it to be "selectable" and that's how an interactive `loon` plot gets returned.

7.2.2 Suite Connection

With the bridge `loon.ggplot`, the suite of `loon` and the suite of `ggplot2` are connected. Figure 7.3 shows the connections between these two suites.

`Loon` to the Suite of `ggplot2`

For `loon` users, after exploration, the plots can be turned into either static graphics through the bridge `loonGrob` or `loon.ggplot`, or an interactive web app via `loon.shiny`. Sometimes, neither a static nor an interactive plot is the best choice, rather than an animation (dynamic graphics) may convey the story to a large extent. To transform a `loon` plot to a dynamic graphic, a video editor (e.g., "photoshop") is an option. While, it suffers a very similar problem with taking a screenshot. The quality might be low. In addition, a lot of manual work might be required (e.g., one has to turn interactive `loon` plots static first. Then, save all plots for edition).

Fortunately, in the `ggplot2` suite, the package `gganimate` can turn a `ggplot` object to a kinematic plot. As the package `loon.ggplot` connects `loon` and `ggplot2`; therefore, a

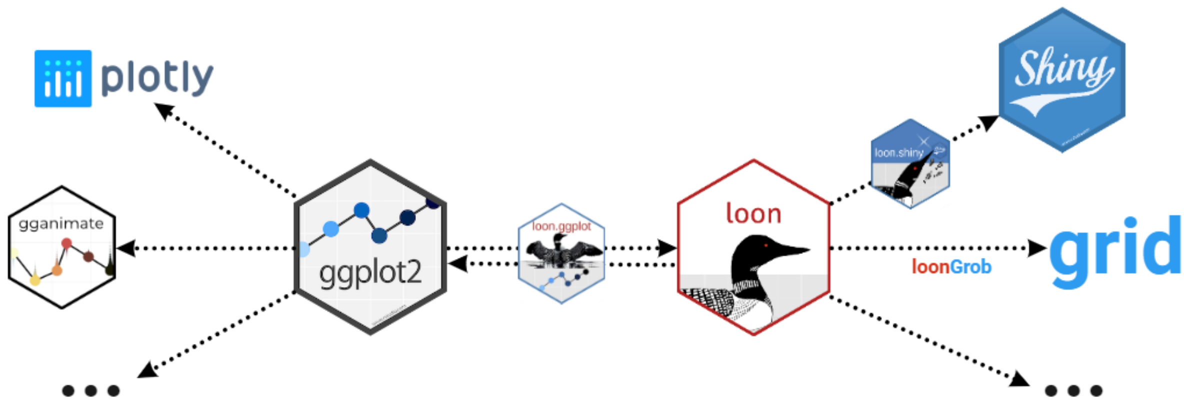


Figure 7.3: With `loon.ggplot`, the features in one suite can be brought into the other.

bridge from `loon` to `ggplot2` graphics could then be followed, then to transform a `loon` display to a `gganimate` kinematic graphic in two steps.

For example, suppose an analyst is interested in the relationship between GDP per capita and life expectancy across different countries for the past 70 years. Specially, he is more interested in the changes of those countries with large populations. The following code shows the `loon` plot (`x` represents the GDP per capita; `y` represents the life expectancy; point size represents the population and point color represents the continent), as in

```
> library(loon)
> library(gapminder)
> p <- with(gapminder,
+           l_plot(gdpPercap, lifeExp,
+                  # scale the size into certain amounts
+                  size = scales::rescale(pop, to = c(4, 50)),
+                  color = continent))
```

Then, query top 10 most populous countries (measured in year 2007), as in

```
> library(dplyr)
> top10in2007 <- gapminder %>%
+   filter(year == 2007) %>%
+   top_n(10, pop)
```

Highlight these countries and split the plot by "continent" and "year", as

```
> p['selected'][gapminder$country %in% top10in2007$country] <- T
> fp <- l_facet(p, by = continent ~ year,
+                   on = gapminder)
```

Figure 7.4 shows the facets (`fp`). As years go by, the GDP per capita and the life expectancy increase simultaneously. For the top 10 most populous countries (highlighted points), most of them are in Asia and Americas; one is in Africa and none of them are in Europe or Oceania. This figure conveys the relationship between GDP per capita and

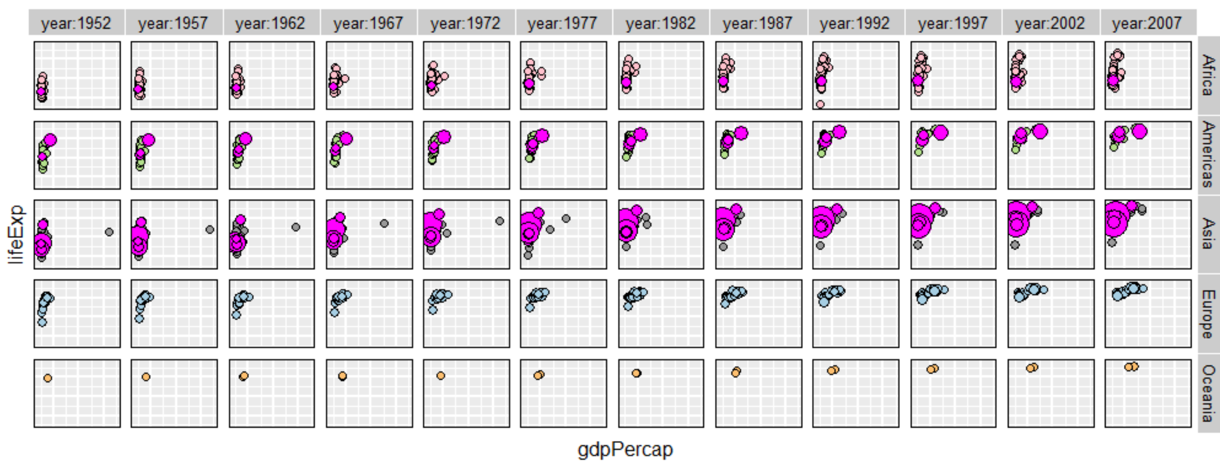


Figure 7.4: The figure shows the life expectancy versus GDP per capita, faceted by year and continent by `loon`.

life expectancy across different countries since 1952. However, displaying too many plots all at once may distract the audience. A better graphic to present this study is to return an animation, in which, the plot can be split by five panels and each panel represents a continent. As time goes by, points in each panel move and the movement of the points represents how the relationship of GDP per capita and life expectancy changes across years.

Benefited from the bridge `loon.ggplot`, creating an animation as well as highlighting the points of interest could be relatively easy, as in (a screenshot of the animation is shown in Figure 7.5),

```
> library(gganimate)
> library(loon.ggplot)
> loon.ggplot(p, selectedOnTop = FALSE) +
```

```
+   facet_wrap(gapminder$continent) +
+   theme(legend.position = "none") +
+   labs(title = 'Year: {frame_time}',
+        x = 'GDP per capita',
+        y = 'life expectancy') +
+   transition_time(gapminder$year) +
+   ease_aes('linear')
```

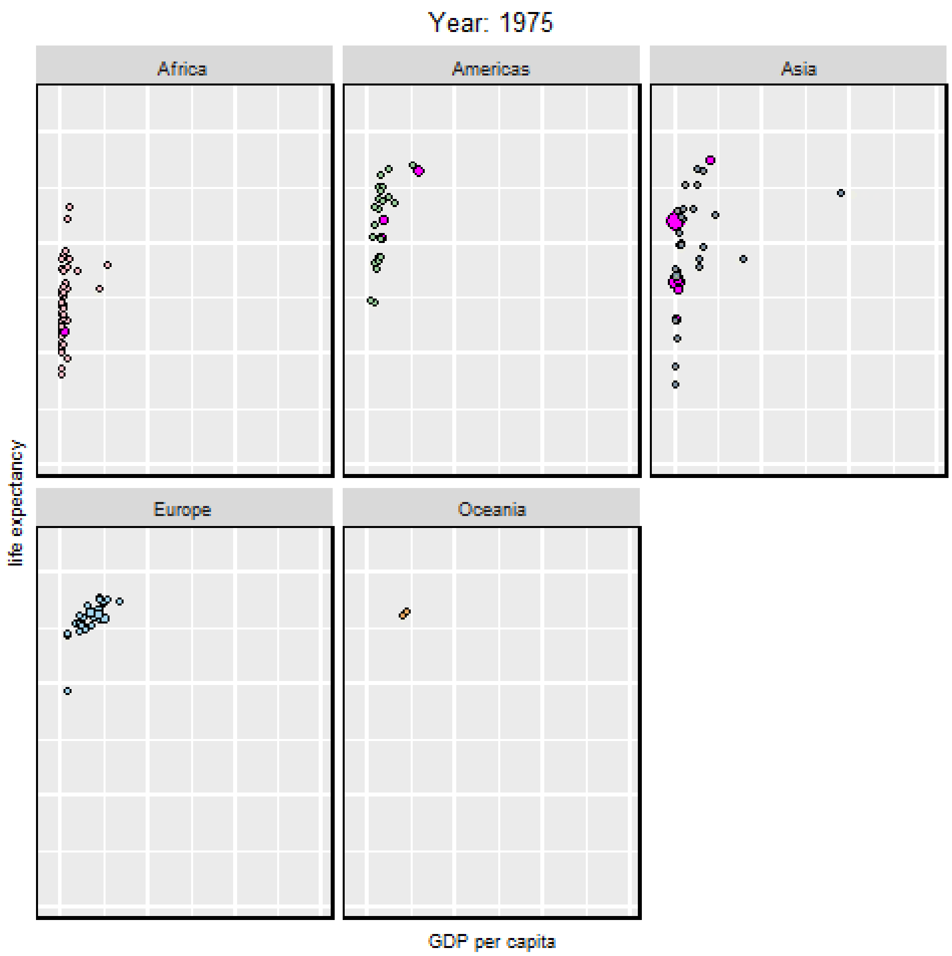


Figure 7.5: The screenshot of the animation

Ggplot2 to the Suite of Loon

In contrast, for ggplot2 users, graphics with some interactions could make the story more vivid in presentation (e.g., publish the findings online). Suppose an analyst starts the analysis in static graphical system ggplot2 and is only interested in the relationship of GDP per capita and life expectancy in year 2007, as in

```
> gp <- gapminder %>%
+   filter(year == 2007,
+          continent != "Oceania") %>%
+   ggplot(mapping = aes(gdpPercap, lifeExp,
+                        color = continent)) +
+   geom_point(mapping = aes(size = pop)) +
+   geom_smooth(mapping = aes(weight = pop),
+               method = "lm",
+               se = FALSE)
> gp
```

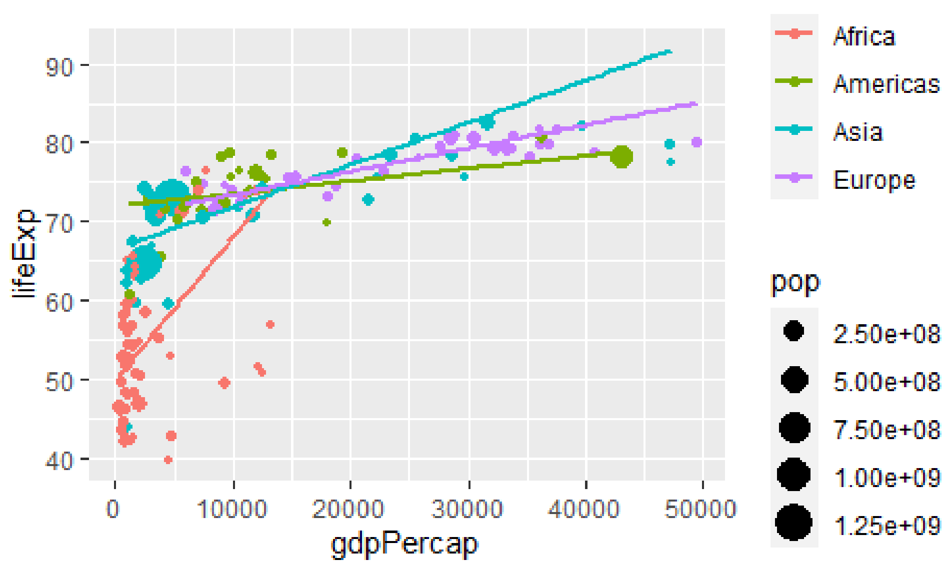


Figure 7.6: The figure shows the life expectancy versus GDP per capita in year 2007 using ggplot2. Each line represents a weighted regression fit.

In Figure 7.6, the line represents the weighted regression which plots the GDP per capita against the life expectancy and sets population as weights. Apparently, the line of Africa has the largest slope. We may conclude that, as the GDP per capita increases by one, people living in Africa may have a higher life expectancy than people from other continents.

Although Figure 7.6 is capable to convey the phenomenon, including some interactivity could be more helpful in presentation. The package **shiny** provides some interactive functionality however one has to build a `ui` and a `server` function on his own. Many logic specifications in the `server` need to be written to make sure that any manipulations on the plot can give the correct response.

With bridges `loon.ggplot` and `loon.shiny`, a handy solution would be that one transforms this `ggplot` object to a `loon` plot then to a `shiny` application, as in

```
> library(loon.shiny)
> library(loon.ggplot)
> gp %>%
+   loon.ggplot() %>%
+   loon.shiny()
```

A shiny web app is created and a screenshot of the app is shown in Figure 7.7. In this shiny

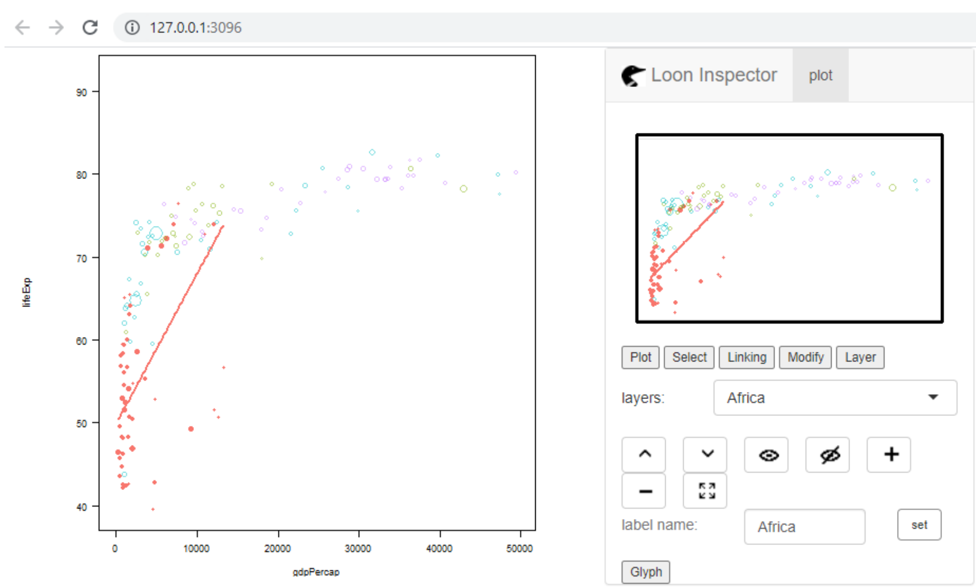


Figure 7.7: A shiny app, based on a `ggplot` object

app, the guide lines are turned off. Except the points representing Africa, the shape of all other points is modified as open circle and the transparency is turned as 0.5. Meanwhile, we only make the fitted line of Africa visible. In this way, without dropping any points, we can better focus on the relationship of the life expectancy and GDP per capita in Africa countries.

7.3 Limitations

The design of a bridge depends upon both $\mathcal{G}$ and $\mathcal{K}$, thus the update of a bridge is always one-step behind the updates of $\mathcal{G}$ or $\mathcal{K}$. If new changes in $\mathcal{G}$ or $\mathcal{K}$ are tiny, without updating the bridge, the effect on the transformation may not be significant. However, if changes applied are dramatic, during the gap period (when updating the bridge), the transformation may be completely broken.

The convenience of a bridge comes with a cost. There are now at least three packages involved, the two graphical systems, $\mathcal{G}$ and $\mathcal{K}$ and the bridge between them. The quality of the bridge will always depend upon the quality of $\mathcal{G}$ and of $\mathcal{K}$. Moreover, the less stable either of these are, the less the stable will be the bridge.

7.4 Further Work

Waddell (2016) listed several further possibilities in his dissertation about `loon`. It is necessary to discuss what has been done yet, what has not been achieved but important in building a bridge and what is not mentioned but valuable.

Possible future directions discussed by Waddell (2016) are:

- "Tk desired improvements"
- "Making every layer type interactive"
- "Linkable layers and linkable arbitrary dimensional states"
- "More sophisticated event patterns for state change bindings"
- "Dealing with missing values"
- "Embedding `loon` in Python"

- "Context specific menus"
- "Annotating Tab for Inspectors"

"Dealing with missing values" and "Embedding `loon` in Python" have been accomplished so far. In the current version of `loon` (1.3.7), the missing values have been accommodated. We choose not to draw them and instead, leave warnings to users.

Some work has been done on embedding `loon` in the environment Python (see the link https://github.com/great-northern-diver/loon/tree/master/Python). Based on the most recent approach, some basic features can be realized. Further work is to implement more new features such as `l_facet` in Python.

Among all these future directions, "making every layer type interactive" would be one of the most interesting things to do. Waddell (2016) suggests a `loon` widget could get an `activelayer` state to determine which layer can receive the mouse and keyboard gestures.

So far, it is not always achievable to map a `ggplot` visual structure to a `loon` visual structure (e.g., expect points and histogram, all other layers are static, etc.). Imagine "making every layer type interactive" is realized, the visual structure mapping could be completed in `loon.ggplot`. For example, to build an interactive pie chart, rather than mapping the visual display or creating a new interactive widget (e.g., `l_pie`), one can simply set the layer which is usually used to draw the polar coordinate (a polygon) as the interactive layer.

Besides, based on further study and research, we discover some other interesting future directions.

- **Embedding layers in an `l_serialaxes` widget**: in the current version, only the main graphics model (i.e., histogram, scatterplot and graph displays) supports adding primitive layer visuals. We find that layering can also be useful in a serialaxes plot, as shown in Figure 3.3. One dimensional statistical graphics can be displayed on each axis.

- **Implementing `loon.tourr` in `shiny`**: when we render a `loon.tourr` object into a `shiny` web app, an `l_tour` object is taken as an ordinary `loon` widget; the features of the interface, such as a dragging bar, a refresh button and scaling radio buttons, to specify `tour` techniques, are missing. In the future, all those specified features are expected to be implemented into a `loon.shiny` app.

- **Visualizing large quantities of data with `loon`**: Unwin et al. (2006) stated that "the number 'a million' is a useful symbolic target" in interactive data visualization. However, Waddell (2016) pointed out "in that respect, `loon` cannot visualize large quantities of data. Although it is possible to create a scatterplot in `loon` with one million points, the interaction speed is likely not satisfactory." Indeed, the performance of visualizing a million data in `iPlots` is outstanding. We try to create two linked plots, a scatterplot and a histogram, with a million observations each. As we brush, the reaction is immediate.

  An interesting project for `loon` suite developers in the future is to build a bridge to transform a `loon` widget to an `iplots` object. Therefore, if the size of data is large and the interaction speed of `loon` is not satisfying. Users can immediately transform the `loon` widget to an `iplot` object for more fluent direct manipulation.

Bibliography

Daniel Adler and Duncan Murdoch. *rgl: 3D Visualization Using OpenGL*, 2020. URL https://CRAN.R-project.org/package=rgl. R package version 0.100.50.

Edgar Anderson. The irises of the Gaspe Peninsula. *Bull. Am. Iris Soc.*, 59:2–5, 1935.

David F. Andrews. Plots of high-dimensional data. *Biometrics*, pages 125–136, 1972.

Daniel Asimov. The grand tour: a tool for viewing multidimensional data. *SIAM journal on scientific and statistical computing*, 6(1):128–143, 1985.

Dean Attali. *colourpicker: A Colour Picker Tool for Shiny and for Selecting Colour in Plots*, 2020. URL https://CRAN.R-project.org/package=colourpicker. R package version 1.1.0.

Baptiste Auguie. *gridExtra: Miscellaneous Functions for "Grid" Graphics*, 2017. URL https://CRAN.R-project.org/package=gridExtra. R package version 2.3.

Geoffrey H. Ball and David J. Hall. PROMENADE-AN ON-LINE PATTERN RECOGNITION SYSTEM. Technical report, STANFORD RESEARCH INST MENLO PARK CA, 1967.

Geoffrey H Ball and David J Hall. Some implications of interactive graphic computer systems for data analysis and statistics. *Technometrics*, 12(1):17–31, 1970.

Richard Becker, William Cleveland, and Allan Wilks. Dynamic Displays of Data Analysis. *Dynamic Graphics for Statistics, Cleveland WS and McGill ME (eds.) Wadsworth Inc., Belmont, California*, pages 1–72, 1988.

Richard A. Becker and William S. Cleveland. Brushing scatterplots. *Technometrics*, 29 (2):127–142, 1987.

Richard A. Becker, William S. Cleveland, and Ming-Jen Shyu. The visual design and control of trellis display. *Journal of Computational and Graphical Statistics*, 5(2):123–155, 1996.

Richard A. Becker, Allan R. Wilks., Ray Brownrigg., Thomas P. Minka, and Alex Deckmyn. *maps: Draw Geographical Maps*, 2018. URL https://CRAN.R-project.org/package=maps. R package version 3.3.0.

Roger Bivand, Friedrich Leisch, and Martin Maechler. *pixmap: Bitmap Images / Pixel Maps*, 2021. URL https://CRAN.R-project.org/package=pixmap. R package version 0.4-12.

Pat Breslin. *Getting to know ArcView GIS: the geographic information system (GIS) for everyone*. ESRI, Inc., 1999.

Jennifer Bryan. *gapminder: Data from Gapminder*, 2017. URL https://CRAN.R-project.org/package=gapminder. R package version 0.3.0.

A. Buja, D. Asimov, and C. Hurley. *Elements of a viewing pipeline for data analysis*. Bell Communications Research. Morris Research and Engineering Center ..., 1986.

Andreas Buja and Daniel Asimov. Grand tour methods: an outline. *Computing Science and Statistics*, 17:63–67, 1986.

Andreas Buja, Catherine Hurley, and Johnalan McDonald. A data viewer for multivariate data. In *Colorado State Univ, Computer Science and Statistics. Proceedings of the 18 th Symposium on the Interface p 171-174(SEE N 89-13901 05-60)*, 1987.

Winston Chang and Hadley Wickham. *ggvis: Interactive Grammar of Graphics*, 2018. URL https://CRAN.R-project.org/package=ggvis. R package version 0.4.4.

Winston Chang, Joe Cheng, J.J. Allaire, Yihui Xie, and Jonathan McPherson. *shiny: Web Application Framework for R*, 2019. URL https://CRAN.R-project.org/package=shiny. R package version 1.3.2.

H. Chernoff. The use of faces to represent statistical association. *JASA*, 68:361–368, 1973.

Chernoff, Herman. Graphical Representations as a Discipline. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE DEPT OF MATHEMATICS, 1978.

William C. Cleveland and Marylyn E. McGill. *Dynamic graphics for statistics*. CRC Press, Inc., 1988.

Dianne Cook and Deborah F. Swayne. *Interactive and Dynamic Graphics for Data Analysis With R and GGobi*. Springer Publishing Company, Incorporated, 2007. ISBN 0387717617, 9780387717616.

Dianne Cook, Andreas Buja, and Javier Cabrera. Projection pursuit indexes based on orthonormal function expansions. *Journal of Computational and Graphical Statistics*, 2 (3):225–250, 1993.

William F. Eddy. A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software (TOMS)*, 3(4):398–403, 1977.

Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

J.H. Friedman, M.A. Fisherkeller, and J.W. Tukey. PRIM-9: An interactive multidimensional data display and analysis system. In *Proc. Fourth International Congress for Stereology*, 1974.

Ram Gnanadesikan. *Methods for statistical data analysis of multivariate observations*. Wiley New York, 1977. ISBN 0471308455.

David Gohel and Panagiotis Skintzos. *ggiraph: Make 'ggplot2' Graphics Interactive*, 2019. URL https://CRAN.R-project.org/package=ggiraph. R package version 0.7.0.

Trevor Hastie and Robert Tibshirani. Non-parametric logistic and proportional odds regression. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 36(3): 260–276, 1987.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

Harold V. Henderson and Paul F. Velleman. Building multiple regression models interactively. *Biometrics*, pages 391–411, 1981.

Toby Hocking, Hadley Wickham, Winston Chang, Nicholas Lewin-Koh, Martin Maechler, Randall Prium, Susan VanderPlas, Carson Sievert, Kevin Ferris, Jun Cai, Faizan Khan, Vivek Kumar, and Himanshu Singh. *animint2: Animated Interactive Grammar of Graphics*, 2020. URL https://CRAN.R-project.org/package=animint2. R package version 2020.9.18.

Marius Hofert and Wayne Oldford. *zenplots: Zigzag Expanded Navigation Plots*, 2019. URL https://CRAN.R-project.org/package=zenplots. R package version 1.0.0.

Marius Hofert and Wayne Oldford. Zigzag Expanded Navigation Plots in R: The R Package zenplots. *Journal of Statistical Software*, 95(1):1–44, 2020.

C. Hurley and R.W. Oldford. Higher hierarchical views of statistical objects. *Available from the video library of the ASA sections on Statistical Graphics: http://stat-graphics.org/movies*, 1988.

Catherine Hurley. *The Data Viewer: A Program For Graphical Data Analysis*. PhD thesis, University of Washington, 1987.

Catherine B. Hurley and R.W. Oldford. Graphs as navigational infrastructure for high dimensional data spaces. *Computational Statistics*, 26(4):585–612, 2011.

Ross Ihaka and Robert Gentleman. R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

Alfred Inselberg and Bernard Dimsdale. Parallel coordinates: a tool for visualizing multi-dimensional geometry. In *Proceedings of the First IEEE Conference on Visualization: Visualization90*, pages 361–378. IEEE, 1990.

Ursula Laa, Dianne Cook, and German Valencia. A slice tour for finding hollowness in high-dimensional data. *Journal of Computational and Graphical Statistics*, 29(3):681–687, 2020.

Michael Lawrence and John Verzani. *Programming graphical user interfaces in R*. Chapman and Hall/CRC, 2018.

John Alan McDonald. *Interactive Graphics for Data Analysis*. PhD thesis, Stanford University, 1982.

Olaf Mersmann. *microbenchmark: Accurate Timing Functions*, 2019. URL `https://CRAN.R-project.org/package=microbenchmark`. R package version 1.4-7.

David Meyer, Achim Zeileis, and Kurt Hornik. The Strucplot Framework: Visualizing Multi-Way Contingency Tables with vcd. *Journal of Statistical Software*, 17(3):1–48, 2006. URL `https://www.jstatsoft.org/v17/i03/`.

David Meyer, Achim Zeileis, and Kurt Hornik. *vcd: Visualizing Categorical Data*, 2020. R package version 1.4-8.

Fanny Meyer and Victor Perrier. *esquisse: Explore and Visualize Your Data Interactively*, 2020. URL https://CRAN.R-project.org/package=esquisse. R package version 0.2.3.

Paul Murrell. The grid graphics package. *R News*, 2(2):14–19, 2002.

Paul Murrell. *gridBase: Integration of base and grid graphics*, 2014. URL https://CRAN.R-project.org/package=gridBase. R package version 0.4-7.

Paul Murrell. *R Graphics*. Chapman and Hall/CRC, 2018.

Paul R. Murrell. *Investigations in Graphical Statistics*. PhD thesis, ResearchSpace@ Auckland, 1998.

R. Wayne Oldford and Stephen C. Peters. DINDE: Towards More Sophisticated Software Environments For Statistics. *SIAM Journal on Scientific and Statistical Computing*, 9 (1):191–211, 1988.

R. Wayne Oldford and Adrian Waddell. *loon.data: Data Used to Illustrate 'Loon' Functionality*, 2020. URL https://CRAN.R-project.org/package=loon.data. R package version 0.0.8.

R.W. Oldford. The Quail Project: Overview and Current Directions. *COMPUTING SCIENCE AND STATISTICS*, pages 397–402, 1998.

R.W. Oldford. Mental Models and Interactive Statistics: Design Principles. In *Computing Science and Statistics*, volume 31, pages 254–262. Interface Foundation of North America, 1999.

R.W. Oldford. Interactive Visualization For Exploratory Data Visualization. Northern Arizona University, Flagstaff AZ, 2019. URL https://www.math.uwaterloo.ca/~rwoldfor/talks/Arizona2019/assets/player/KeynoteDHTMLPlayer.html.

R.W. Oldford. *Logical queries*, 2020a. URL https://great-northern-diver.github.io/loon/articles/logicalQueries.html. Loon Articles.

R.W. Oldford. South African Heart Disease Data. https://great-northern-diver.github.io/loon.data/, 2020b.

Chris Parmer, Ryan Patrick Kyle, Carson Sievert, and Hammad Khan. *dash: An Interface to the 'dash' Ecosystem for Authoring Reactive Web Applications*, 2020. URL https://CRAN.R-project.org/package=dash. R package version 0.5.0.

Thomas Lin Pedersen. *patchwork: The Composer of Plots*, 2020a. URL https://CRAN.R-project.org/package=patchwork. R package version 1.1.1.

Thomas Lin Pedersen. *patchwork*, 2020b. URL https://patchwork.data-imaginist.com/.

Thomas Lin Pedersen. *ggraph: An Implementation of Grammar of Graphics for Graphs and Networks*, 2021. URL https://CRAN.R-project.org/package=ggraph. R package version 2.0.5.

Thomas Lin Pedersen and David Robinson. *gganimate: A Grammar of Animated Graphics*, 2019. URL https://CRAN.R-project.org/package=gganimate. R package version 1.0.3.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. URL http://www.R-project.org/.

J.E. Rossouw, J.P. Plessis Du, A.J. Benadé, P.C. Jordaan, J.P. Kotze, P.L. Jooste, and J.J. Ferreira. Coronary risk factor screening in three rural communities. The CORIS baseline study. *South African medical journal= Suid-Afrikaanse tydskrif vir geneeskunde*, 64(12): 430–436, 1983.

Christopher D. Salahub and R. Wayne Oldford. About "her emails". *Significance*, 15(3): 34–37, 2018.

Deepayan Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer, New York, 2008. URL http://lmdvr.r-forge.r-project.org. ISBN 978-0-387-75968-5.

David W. Scott. *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 2015.

Charlie Sharpsteen and Cameron Bracken. *tikzDevice: R Graphics Output in LaTeX Format*, 2020. URL https://CRAN.R-project.org/package=tikzDevice. R package version 0.12.3.1.

Jonathan Sidi. *ggedit: Interactive 'ggplot2' Layer and Theme Aesthetic Editor*, 2020. URL https://CRAN.R-project.org/package=ggedit. R package version 0.3.1.

Carson Sievert. *plotly for R*, 2018. URL https://plotly-r.com.

Carson Sievert, Susan VanderPlas, Jun Cai, Kevin Ferris, Faizan Uddin Fahad Khan, and Toby Dylan Hocking. Extending ggplot2 for linked and animated web graphics. *Journal of Computational and Graphical Statistics*, 28(2):299–308, 2019.

Karline Soetaert. *diagram: Functions for Visualising Simple Graphs (Networks), Plotting Flow Diagrams*, 2020. URL https://CRAN.R-project.org/package=diagram. R package version 1.6.5.

D. Swayne, Duncan Temple Lang, Andreas Buja, and Dianne Cook. GGobi: XGobi Redesigned and Extended. In *Proceedings of the 33th Symposium on the Interface: Computing Science and Statistics*, 2001.

Deborah F. Swayne and Sigbert Klinke. Introduction to the special issue on interactive graphical data analysis: What is interaction? *Computational Statistics*, 14(1):1–6, 1999.

Deborah F. Swayne, Dianne Cook, and Andreas Buja. XGobi: Interactive dynamic data visualization in the X Window System. *Journal of computational and Graphical Statistics*, 7(1):113–130, 1998.

Jürgen Symanzik, Dianne Cook, Nicholas Lewin-Koh, James J. Majure, and Inna Megretskaia. Linking ArcView™ and XGobi: Insight behind the Front End. *Journal of Computational and Graphical Statistics*, 9(3):470–490, 2000.

Martin Theus and Simon Urbanek. *Interactive Graphics for Data Analysis*. Chapman & Hall/CRC, 2008. ISBN 1584885947, 9781584885948.

Luke Tierney. *tkrplot: TK Rplot*, 2021. URL https://CRAN.R-project.org/package=tkrplot. R package version 0.0-26.

John W. Tukey. *Exploratory Data Analysis*. Mass: Addison-Wesley, 1977.

John W. Tukey and Martin B. Wilk. Data analysis and statistics: an expository overview. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 695–709, 1966.

Antony Unwin. Requirements for interactive graphics software for exploratory data analysis. *Computational Statistics*, 14(1):7–22, 1999.

Antony Unwin, Martin Theus, and Heike Hofmann. *Graphics of Large Datasets: Visualizing a Million*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387329064.

Simon Urbanek. *JavaGD: Java Graphics Device*, 2020. URL https://CRAN.R-project.org/package=JavaGD. R package version 0.6-4.

Simon Urbanek and Jeffrey Horner. *Cairo: R Graphics Device using Cairo Graphics Library for Creating High-Quality Bitmap (PNG, JPEG, TIFF), Vector (PDF, SVG, PostScript) and Display (X11 and Win32) Output*, 2020. URL https://CRAN.R-project.org/package=Cairo. R package version 1.5-12.2.

Simon Urbanek and Martin Theus. iPlots: high interaction graphics for R. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*, 2003.

Simon Urbanek and Tobias Wichtrey. *iplots: iPlots - interactive graphics for R*, 2018. URL https://CRAN.R-project.org/package=iplots. R package version 1.1-7.1.

Adrian Waddell. *Interactive Visualization and Exploration of High-Dimensional Data*. PhD thesis, University of Waterloo, 2016.

Adrian Waddell and R. Wayne Oldford. *loon: Interactive Statistical Data Visualization*, 2020. URL http://great-northern-diver.github.io/loon/. R package version 1.3.0.

Gregory R. Warnes, Ben Bolker, Lodewijk Bonebakker, Robert Gentleman, Wolfgang Huber, Andy Liaw, Thomas Lumley, Martin Maechler, Arni Magnusson, Steffen Moeller, Marc Schwartz, and Bill Venables. *gplots: Various R Programming Tools for Plotting Data*, 2020. URL https://CRAN.R-project.org/package=gplots. R package version 3.1.1.

H. Wickham. *Mastering Shiny: Build Interactive Apps, Reports, and Dashboards Powered by R*. O'Reilly Media, Incorporated, 2021. ISBN 9781492047384. URL https://books.google.ca/books?id=nrvAzQEACAAJ.

Hadley Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010.

Hadley Wickham. *Advanced R*. Chapman and Hall/CRC, 2014.

Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer, 2016.

Hadley Wickham and Thomas Lin Pedersen. *gtable: Arrange 'Grobs' in Tables*, 2019. URL https://CRAN.R-project.org/package=gtable. R package version 0.3.0.

Hadley Wickham, Micheal Lawrence, Duncan Temple Lang, and Deborah F. Swayne. *An introduction to rggobi*, 2006.

Hadley Wickham, Dianne Cook, Heike Hofmann, and Andreas Buja. tourr: An R Package for Exploring Multivariate Data with Projections. *Journal of Statistical Software*, 40(2): 1–18, 2011. URL http://www.jstatsoft.org/v40/i02/.

Adalbert Wilhelm. Interactive statistical graphics: the paradigm of linked views. *Handbook of statistics*, 24:437–537, 2005.

Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg, 2005. ISBN 0387245448.

Yihui Xie, H. Hofmann, Di Cook, X. Cheng, B. Schloerke, M. Vendettuoli, T. Yin, H. Wickham, and M. Lawrence. cranvas: Interactive statistical graphics based on Qt. *R package version 0.8*, 3:555, 2013.

Zehao Xu and R. Wayne Oldford. *loon.ggplot: Making 'ggplot2' Plots Interactive with 'loon' and Vice Versa*, 2019a. R package version 1.2.1.

Zehao Xu and R. Wayne Oldford. *loon.shiny: Automatically Create a 'Shiny' App Based on Interactive 'Loon' Widgets*, 2019b. R package version 1.0.0.

Zehao Xu and R. Wayne Oldford. *ggmulti: High Dimensional Data Visualization*, 2020. R package version 0.1.0.

Zehao Xu and R. Wayne Oldford. *loon.tourr: Tour in 'Loon'*, 2021. R package version 0.1.1.

Appendices

All sources of this thesis can be found in the following links:

- `loon` and `loonGrob`: https://great-northern-diver.github.io/loon/
- `loon.ggplot`: https://great-northern-diver.github.io/loon.ggplot/
- `ggmulti`: https://great-northern-diver.github.io/ggmulti/
- `loon.shiny`: https://great-northern-diver.github.io/loon.shiny/
- `loon.tourr`: https://great-northern-diver.github.io/loon.tourr/

Appendix A

Introduction

Figure 1.1

```
> library(rgl)
> # rgl 3D
> with(iris, plot3d(Sepal.Length, Sepal.Width, Petal.Length,
+                   type="s", col=as.numeric(Species)))
+ mtcars %>%
> library(ggvis)
> # ggvis
> ggvis(~wt, ~mpg,
+       size := input_slider(10, 100, label = "size"),
+       opacity := input_slider(0, 1, label = "transparency")
+ ) %>%
+ layer_points()
```

Figure 1.2

```
> library(gganimate)
> library(gapminder)
> ###################### gganimate #################
> ggplot(gapminder, aes(gdpPercap, lifeExp, size = pop, color = country)) +
+   geom_point(alpha = 0.7, show.legend = FALSE) +
+   scale_color_manual(values = country_colors) +
+   scale_size(range = c(2, 12)) +
+   scale_x_log10() +
+   facet_wrap(~continent) +
+   # Here comes the gganimate specific bits
```

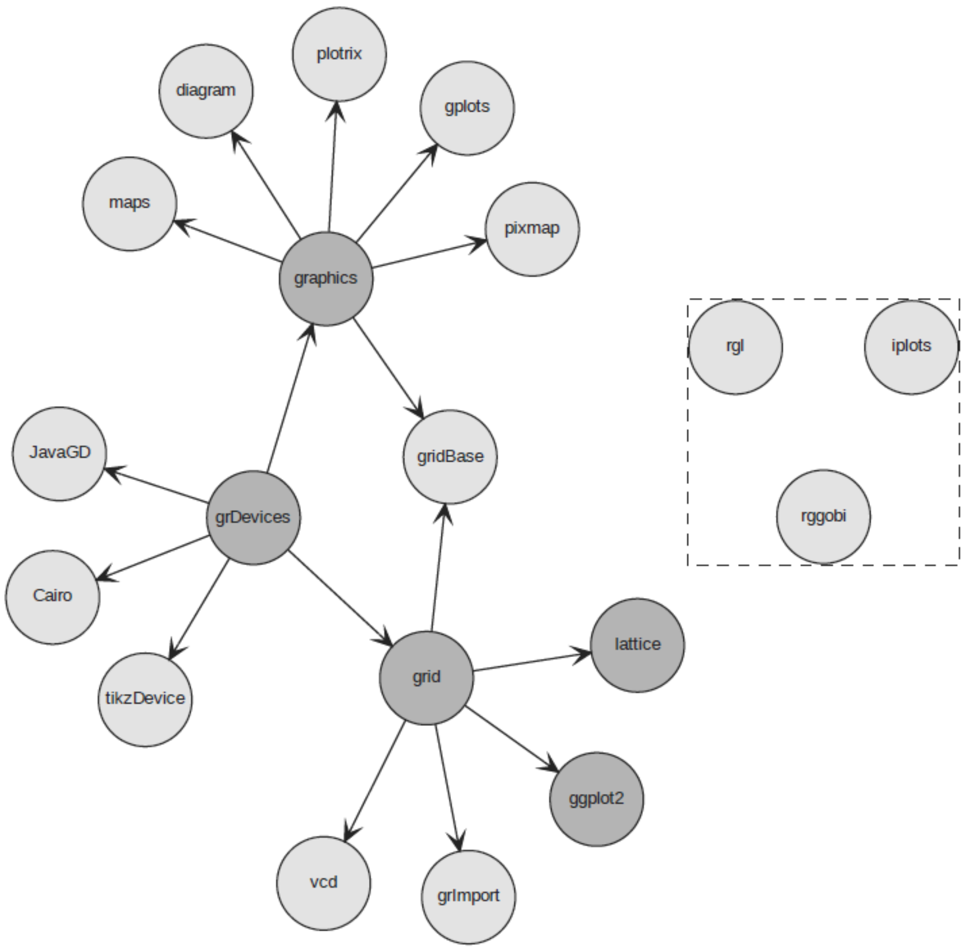


Figure A.1: The structure of the R graphics system (Murrell, 2018)

```
+   labs(title = 'Year: {frame_time}',
+         x = 'GDP per capita',
+         y = 'life expectancy') +
+   transition_time(year) +
+   ease_aes('linear')
```

Figure 1.4

```
> library(loon)
> library(loon.data)
> data("SAheart")
> h1 <- l_hist(SAheart$chd,
+                linkingGroup = "chd",
+                title = "Coronary heart disease"))
> h2 <- l_hist(SAheart$famhist,
+                linkingGroup = "chd",
+                title = "Family history of heart disease"))
```

Highlight bins via programming

```
> withFamilyHistory <- SAheart$famhist == "Present"
> h2['selected'] <- withFamilyHistory
> h3 <- l_hist(SAheart$sbp, linkingGroup = "chd",
+                title = "Systolic blood pressure (mmHg)"))
```

Figure 1.6 (a) can also be created by base `graphics`.

```
> with(SAheart,
+      barplot(table(chd),
+      col = 'thistle',
+      main = "Coronary heart disease"))
> with(SAheart,
+       barplot(table(chd[highBloodPressure|withFamilyHistory]),
+     col = 'magenta',
+     add = TRUE)
+ )
```

Appendix B

loon

Figure 2.2

```
> lp <- with(mpg, l_plot(displ, hwy,
+                        showItemLabels = TRUE,
+                        color = "black"))
```

Figure 2.3

```
> lp['itemLabel'] <- with(mpg,
+                         paste0("model:", manufacturer, " ",
+                                model, "\n",
+                                "year:", year, "\n",
+                                "drive way:", drv, "\n",
+                                "fuel type:", fl)
+ )
```

Figure 2.4

```
> g <- loongraph(
+    nodes = c("A", "B", "C", "D"),
+    from = c("A", "A", "B", "B", "C"),
+    to   = c("B", "C", "C", "D", "D")
+ )
> ## Not run:
> # create a loon graph plot
> p <- l_graph(g)
> l_navigator_add(p)
```

Figure 2.7 (a)

```
> p <- l_plot(x = c(rep(1, 3),
+                   rep(2, 3),
+                   rep(3, 3)),
+             y = rep(1:3, 3),
+             color = "pink",
+             glyph = "circle",
+             xlabel = "",
+             ylabel = "")
> dat <- data.frame(
+   x = c(rep(1, 3),
+         rep(2, 3),
+         rep(3, 3)),
+   y = rep(1:3, 3),
+   anchor = c("sw","w", "nw",
+              "s", "center", "n",
+             "se", "e", "ne")
+ )
> for(i in 1:9) {
+   d <- dat[i, ]
+   l_layer_text(p,
+                x = d$x,
+                y = d$y,
+                size = 20,
+                text = d$anchor,
+               anchor = d$anchor)
+ }
```

Figure 2.7 (b)

```
> p <- l_plot()
> dat <- data.frame(
+   x = c(rep(1, 3)),
+   y = 3:1,
+   text = c('This is right \n Right',
+            'This is center \n Center',
+            'This is left \n Left'),
+   justify = c("right", "center", "left")
+ )
> for(i in 1:3) {
+   d <- dat[i, ]
```

```
+   l_layer_text(p,
+                x = d$x,
+                y = d$y,
+                size = 20,
+                text = d$text,
+                justify = d$justify)
+ }
> l_scaleto_world(p)
```

Appendix C

loon.ggplot

Figure 3.12 (a)

```
> ggplot2loon(pm)
```

Figure 3.12 (b)

```
> ggplot2loon(pm, activeGeomLayers = 0L) %>%
+   l_scaleto_world()
```

Table C.1: Primitive Layers

loon	ggplot
l_layer()	layer()
l_layer_group()	
	geom_blank()
l_layer_line(), l_layer_lines()	geom_segment(), geom_path(), geom_line(), geom_step()
l_layer_rectangle(), l_layer_rectangles()	geom_rect(), geom_tile()
l_layer_polygon(), l_layer_polygons()	geom_polyon()
l_layer_texts(), l_layer_text()	geom_label(), geom_text()
l_layer_oval()	
l_layer_points()	geom_point(), geom_jitter()

Table C.2: One Dimension Layers

loon	ggplot
	`geom_dotplot()`
`l_layer.density()`	`geom_density()`, `geom_freqpoly()`
`l_hist()`	`geom_bar()`, `geom_col()`, `geom_histogram()`
	`geom_area()`, `geom_ribbon()`

Table C.3: Two Dimension Layers

loon	ggplot
	geom_boxplot(), geom_violin()
	geom_quantile()
l_glyph_add_pointrange()	geom_pointrange()
l_glyph_add_text()	geom_text()
	geom_crossbar(), geom_errorbar(), geom_linerange()
l_plot(), l_graph()*	geom_point()
l_layer.map()	geom_map()
	geom_count()
l_layer_smooth()	geom_smooth()
	geom_bin2d(), geom_hex()
	geom_rug()

Table C.4: Three Dimension Layers

loon	ggplot
`l_plot3D()`	`geom_point(aes(x, y, z))`
`l_layer_contourLines()`	`geom_density2d()`, `geom_contour()`
`l_layer_heatImage()`, `l_layer_rasterImage()`	`geom_raster()`