

# **Interactive Visualization and Exploration of High-Dimensional Data**

by

**Adrian Waddell**

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, 2016

© Adrian Waddell 2016

### **Author's Declaration**

I hereby declare that I am the sole author of this 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.

## Abstract

Visualizing data is an essential part of good statistical practice. Plots are useful for revealing structure in the data, checking model assumptions, detecting outliers and finding unanticipated patterns. Post-analysis visualization is commonly used to communicate the results of statistical analyses. The availability of good statistical visualization software is key in effectively performing data analysis and in exploring and developing new methods for data visualization. Compared to static visualization, interactive visualization adds natural and powerful ways to explore the data. With interactive visualization an analyst can dive into the data and quickly react to visual clues by, for example, re-focusing and creating interactive queries of the data. Further, linking visual attributes of the data points such as color and size allows the analyst to compare different visual representations of the data such as histograms and scatterplots.

In this thesis, we explore and develop new interactive data visualization and exploration tools for high-dimensional data. The original focus of our research was a software implementation of navigation graphs. Navigation graphs are navigational infrastructures for controlled exploration of high-dimensional data. As part of this thesis, we developed the first interactive implementation of these navigation graphs called RnavGraph. With RnavGraph we explored various features for enhancing the usability of navigation graphs. We concluded that a powerful interactive scatterplot display and methods to deal with large graphs were two areas that would add great value to the navigation graph framework.

RnavGraph's scatterplot display proved to be particularly useful for data analysis and we continued our research with the design and implementation of a general-purpose interactive visualization toolkit called loon. The core contributions of loon are as follows. loon implements a general design for interactive statistical graphic displays that supports layering of visual information such as point objects, lines and polygons. These displays further support zooming, panning and selection, and modification and deactivation of plot elements and layers. Interactions with plots are provided with mouse and keyboard gestures as well as via command line control and with inspectors. These inspectors provide graphical user interfaces for modifying and overseeing the plots. loon also im-

plements a novel dynamic linking mechanism that can be used to assign the plots that are to be linked and the linking rules at run time. Additionally, loon's design provides several different types of event bindings to add and customize functionality of loon's displays. In this thesis, we discuss loon's design and framework by giving concrete examples that show how these design choices can be used to efficiently explore and visualize data interactively. These examples revolve around loon's statistical interactive displays such as histograms, scatterplots and graph displays. We also illustrate how loon's design can be used to layer on plots relevant statistical information and model fits such as density estimates, contours, regression lines and geographical maps for spatial data analysis.

loon is implemented in Tcl and Tk and we explore the integration of loon's framework into a complete statistical computing environment such as R. We show examples of statistical analyses performed in R that are enhanced with interactivity using loon.

loon also implements a number of new tools for high-dimensional data exploration. The serialaxes display represents the data using parallel or radial coordinates. The scatterplot display supports high-dimensional point glyphs such as serialaxes glyphs, polygons and images. loon's navigation graphs allow for multiple navigators and for direct manipulation of a graph which includes deactivating nodes and their adjoining edges.

To deal with large graphs, we propose and implement environments for creating navigation graphs interactively by filtering the nodes with respect to some node-associated relevant measures. Such measures include the correlation of variable pairs and the graph-based scagnostics measures. We use sliders, histograms and scatterplot matrices to interactively filter the nodes based on the value of their associated measure. Measures are kept generic and can be recalculated for the subset of selected data points. As another tool for exploring high-dimensional data, we introduce a setup that allows the analyst to select points and have their k-nearest neighboring points highlighted automatically. The space to calculate the inter-point distances that determine the k-nearest neighbors can be chosen dynamically. Finally, we propose a new high-dimensional point glyph called the spiro glyph.

While some of loon's interaction features have appeared in part or in whole in statistical systems in the past 40 years (e.g. brushing, panning, zooming, linking plots, etc.),



no other equally rich system has provided (or continues to provide) an interactive data visualization system integrated with a widely available and stable statistical system like R. Both Tcl and R are well suited for rapid prototyping of software and statistical methods; with soon rapid prototyping of interactive data visualization tools and methods become possible as well.

## **Acknowledgements**

I would like to thank my advisor Wayne Oldford for his guidance, support and generosity. His deep insight and knowledge have been most influential on my research work and academic development.

I am also grateful to those who contributed to making R and Tcl the excellent technologies that they are today.

Above all, I wish to thank my parents for their constant encouragement and support. Also, many thanks to my siblings and friends. I am particularly thankful to my friends Max and Tobias for their company and for all the ideas we exchanged over the years. Finally, my very special thanks go to Oana for being a wonderful and supportive partner.

# Table of Contents

<b>Author's Declaration</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>vi</b>
<b>List of Figures</b>	<b>xii</b>
<b>List of Tables</b>	<b>xvii</b>
<b>1 Background</b>	<b>1</b>
1.1 On High-Dimensional Data . . . . .	1
1.2 On Low-Dimensional Views . . . . .	2
1.3 Navigation Graphs . . . . .	5
1.3.1 The Canonical Graph Semantic . . . . .	6
1.3.2 Automatic Graph Construction and Exploration . . . . .	8
1.3.2.1 Saturated 3d and 4d transition graph . . . . .	8
1.3.2.2 Graph Products . . . . .	11
1.3.2.3 Automatic Graph Traversal . . . . .	12
1.4 On the Problem of Large Graphs . . . . .	13
1.4.1 Finding an Interesting Subgraph . . . . .	14
1.4.2 Dimensionality Reduction/Constructing Dimensions . . . . .	16
1.5 Other Graphs of Possible Interest . . . . .	21
1.6 Interactive Data Visualization Software . . . . .	22

<b>2</b>	<b>RnavGraph</b>	<b>26</b>
2.1	A Default RnavGraph Session . . . . .	27
2.2	The Navigation Graph Display . . . . .	27
2.3	The tk2d Scatterplot Display . . . . .	32
2.4	Software Architecture . . . . .	35
2.4.1	navGraph . . . . .	36
2.4.2	scagGraph . . . . .	40
2.4.3	Extending RnavGraph . . . . .	41
2.5	Lessons Learned . . . . .	43
<b>3</b>	<b>Loon By Example</b>	<b>46</b>
3.1	An Exploratory Data Analysis . . . . .	47
3.2	Performing the Exploratory Analysis with the loon R package . . . . .	54
3.2.1	Plot States . . . . .	54
3.2.2	Graphical User Interface . . . . .	55
3.2.3	Linking . . . . .	57
3.2.4	Layers . . . . .	57
3.2.5	Star Glyphs . . . . .	58
3.3	Conclusions . . . . .	59
<b>4</b>	<b>Loon Framework</b>	<b>60</b>
4.1	Introduction to the Displays . . . . .	62
4.1.1	Scatterplot . . . . .	62
4.1.2	Histogram . . . . .	62
4.1.3	Serialaxes Display . . . . .	64
4.1.4	Graph Display . . . . .	64
4.1.5	Inspectors . . . . .	67
4.2	Main Graphics Model . . . . .	69
4.2.1	Plot Layout . . . . .	69
4.2.2	Mapping Data Onto the Plot Region . . . . .	71
4.3	Plot States . . . . .	73
4.3.1	Abstract Dimensions . . . . .	76

4.3.2	Configuration Pipeline . . . . .	78
4.3.3	State Normalization . . . . .	80
4.4	Graphical User Interface . . . . .	81
4.4.1	Zoom & Pan . . . . .	81
4.4.2	Visual Query . . . . .	82
4.4.2.1	Item Labels . . . . .	82
4.4.2.2	Interactive Selection . . . . .	84
4.4.3	Temporarily Relocating Points . . . . .	86
4.4.4	Inspectors . . . . .	88
4.4.4.1	loon Inspector . . . . .	89
4.4.4.2	Worldview . . . . .	89
4.4.4.3	Analysis Inspectors . . . . .	90
4.4.4.4	Layers Inspector . . . . .	90
4.5	Standard Linking Model . . . . .	91
4.6	Layers . . . . .	95
4.6.1	Functions and Methods for Layering Data in R . . . . .	98
4.7	Display Design Decisions . . . . .	103
4.7.1	Histogram . . . . .	103
4.7.2	Point and Node Glyphs . . . . .	103
4.7.3	Serialaxes Display and Serialaxes Glyphs . . . . .	107
4.7.4	Graph Display . . . . .	109
4.7.4.1	Graphswitch . . . . .	109
4.7.4.2	Navigators . . . . .	110
4.7.4.3	Navigator Contexts . . . . .	113
<b>5</b>	<b>Advanced Loon Framework</b> . . . . .	<b>121</b>
5.1	Implementation . . . . .	121
5.2	Event Bindings . . . . .	124
5.2.1	R function callbacks . . . . .	125
5.2.2	State Change Bindings . . . . .	126
5.2.3	Item Bindings . . . . .	128
5.2.4	Canvas Bindings . . . . .	131

5.2.5	Content Bindings . . . . .	132
5.3	Custom Linking . . . . .	134
5.3.1	One Directional And One-To-Many Linking . . . . .	134
5.3.2	Linking States with Different Names . . . . .	136
5.3.3	Linking Items Within a Plot . . . . .	137
5.3.4	Avoiding Circularity . . . . .	138
5.3.5	Linking Model with Non-Model Layers . . . . .	139
5.4	Geometry Management . . . . .	140
5.5	Writing an Inspector . . . . .	144
5.6	Other Topics . . . . .	145
5.6.1	Export as an Image . . . . .	145
5.6.2	Animations . . . . .	146
5.6.3	Color Mapping . . . . .	146
<b>6</b>	<b>General Statistical Interaction Examples</b>	<b>150</b>
6.1	Power Transformations . . . . .	151
6.2	Interactively Adding Regression Lines . . . . .	153
6.3	Sensitivity Analysis of a Simple Linear Regression . . . . .	155
6.4	Interactive K Nearest Neighbor highlighting . . . . .	158
6.4.1	A Quick Solution . . . . .	158
6.4.2	A Solution With Control Panel . . . . .	160
<b>7</b>	<b>Exploring High-Dimensional Data</b>	<b>163</b>
7.1	Navigation Graphs . . . . .	164
7.1.1	Canonical Navigation Graph Setup . . . . .	164
7.1.2	Dynamic Navigation Graphs Based on Measure Ranges . . . . .	168
7.1.3	Dynamic Navigation Graph based on Plots . . . . .	172
7.1.4	Closures of Measures . . . . .	173
7.1.5	Exploring New Graph Semantics . . . . .	180
7.2	Spiro Glyphs . . . . .	184
<b>8</b>	<b>Conclusions and Future Work</b>	<b>190</b>

8.1	Conclusions for loon . . . . .	190
8.2	Future Work . . . . .	194
8.2.1	loon in General . . . . .	194
8.2.2	Current Displays . . . . .	197
8.2.3	New Displays . . . . .	198
8.2.4	Navigation Graphs . . . . .	199
	<b>References</b>	<b>201</b>
	<b>APPENDICES</b>	<b>208</b>
<b>A</b>	<b>R Code for Chapter 6 Examples</b>	<b>209</b>
A.1	Power Transformations . . . . .	209
A.2	Interactively Adding Regression Lines . . . . .	210
A.3	Sensitivity Analysis Simple Linear Regression . . . . .	211
A.4	Interactive K nearest neighbour highlighting . . . . .	216

# List of Figures

1.1	The Italian olive oil data – recorded fatty acids and sampled areas. . . . .	7
1.2	Saturated 3d and 4d transition graphs for the olive data. . . . .	8
1.3	3d rigid rotation and linked scatterplot display. . . . .	9
1.4	4d transition along a geodesic. . . . .	9
1.5	Complete variable graph $G$ for a 4 dimensional data set with variates labelled $A, B, C,$ and $D$ . Line graph $L(G)$ . Complement graph $\overline{L(G)}$ . . . . .	10
1.6	Automatic saturated 3d and 4d transition graph construction with a non-complete variable graph. . . . .	10
1.7	Two variable graphs $G$ and $H$ capturing a certain structure. . . . .	12
1.8	Three graph products of $G$ and $H$ from Figure 1.7 . . . . .	12
1.9	Scatterplot matrix example from Hurley and Oldford [44] Section 4.1. . . . .	13
1.10	a) Scatterplot of the scagnostic measures – convex vs. monotonic. b) Particular scatterplots with high monotonic and convex measure, and high convex and low monotonic measure. . . . .	16
1.11	Scatterplot of the scagnostic measures – convex vs. monotonic – overlaid on the scagnostic measures along two 4d transitions from the olive data. The blue path shows the scagnostic measures when linolenic transitions into oleic and arachidic transitions into palmitoleic. The green path shows the scagnostic measures when linolenic transitions into palmitoleic and arachidic transitions into oleic. . . . .	17
1.12	Comparing seven dimensional reduction methods using navigation graphs with the canonical semantic. . . . .	25



2.1	Default RnavGraph session with the default group colors. . . . .	28
2.2	Path tool . . . . .	32
2.3	Different plot types supported by the tk2d scatterplot display. . . . .	34
2.4	Rectangular brush tool. . . . .	35
2.5	Saturated 3d transition graph for olive data. . . . .	38
2.6	RnavGraph session. . . . .	39
2.7	scagNav session . . . . .	41
2.8	User-defined plot for an RnavGraph session. . . . .	44
3.1	Visible minority population versus total population for each of the 33 Canadian metropolitan census areas. Here $t$ = total population count, $m$ = total visible minority count. . . . .	48
3.2	Visible minority population versus total population for each of the 33 Canadian metropolitan census areas – range from 0.0015 for Trois-Rivières to 0.18 for Vancouver. Here $t$ = total population count, $m$ = total visible minority count and $c$ = total Chinese minority count. . . . .	49
3.3	Radial axis plots for the two largest Canadian census areas. . . . .	50
3.4	Radial axis plots, or star glyphs, for all 33 metropolitan census areas. Colors are assigned based on a red-yellow-green gradient from west to east. The radial axis order is shown at the right. . . . .	51
3.5	Zooming in on a region. (a) A worldview plot. (b) The area of focus – Southwestern Canada and the Prairies. The region shown in (b) is highlighted with a white rectangle in (a). . . . .	52
3.6	The default loon inspector is context specific for the active loon plot. For a scatterplot display it shows a worldview, an analysis, layers and glyphs inspector. . . . .	53
4.1	loon’s scatterplot display. . . . .	63
4.2	loon’s histogram display. . . . .	65
4.3	loon’s serialaxes display plots the data either as a stacked star glyphs plot (a) or as a parallel coordinates plot (b). . . . .	66
4.4	loon’s graph display. . . . .	67
4.5	loon’s inspectors. . . . .	68

4.6	Main graphics model. . . . .	69
4.7	Plot layout . . . . .	71
4.8	The configuration pipeline for state modifications. . . . .	78
4.9	Zoom and pan gestures for the histogram, scatterplot and graph display. Zooming requires a mouse scroll gesture. Panning requires a right mouse button drag. Two superimposed mice with an arrow indicate a drag gesture. . . . .	82
4.10	loon’s item labels that are displayed with a “tool-tip” pop-up. . . . .	83
4.11	Interactive mouse/keyboard selection techniques. . . . .	84
4.12	Selection gestures for the histogram, scatterplot and graph display. Two superimposed mice with an arrow indicate a drag gesture. . . . .	85
4.13	Temporary relocating points on a scatterplot. . . . .	87
4.14	Worldview inspector and its composition in perspective. . . . .	90
4.15	Layers inspector. . . . .	91
4.16	Scatterplot with a layered regression line, a 95% confidence interval and a 95% prediction interval of a simple linear fit. . . . .	96
4.17	Natureearth data displayed with loon’s scatterplot display. . . . .	99
4.18	Two layers: a heat image and contour lines of a 2d density estimation. . . . .	102
4.19	Point glyph examples. . . . .	105
4.20	Point glyph size mapping. . . . .	106
4.21	Glyphs inspector. . . . .	108
4.22	Graphswitch and graph display. . . . .	110
4.23	Navigator example. . . . .	112
4.24	Context2d mapping scheme to xvars and yvars. . . . .	114
4.25	Context2d example session. . . . .	115
4.26	loon’s slicing2d context setting for 3d transitions. . . . .	119
4.27	loon’s slicing2d context setting for 4d transitions. . . . .	120
5.1	Life expectancy (in years) vs. fertility (number of children per women) for different countries in 2002. The data is from the Gapminder data project [35]. The choropleth plot (right panel) encodes life expectancy as color. . . . .	141
5.2	Geometry management. . . . .	142
5.3	Scatterplot matrix using grid geometry manager. . . . .	143

5.4	Custom inspector for aspect ratio. . . . .	145
5.5	Hcl colors: luminance is 70, chroma on circle is 66. The numbers indicate loon's default color mapping order. . . . .	148
5.6	Examples of mapping data values to colors. . . . .	149
6.1	Power transformation example. . . . .	152
6.2	Interactively adding regressions lines. . . . .	154
6.3	Influential points in regression analysis. . . . .	156
6.4	Influential points in regression: recolor points to remove outliers. . . . .	156
6.5	Influential points in regression: sensitivity analysis. . . . .	157
6.6	3 nearest neighbors highlighted. . . . .	159
6.7	$K$ nearest neighbors highlighting for subspaces. . . . .	162
7.1	<code>l_navgraph</code> setup. . . . .	166
7.2	Star glyphs. . . . .	167
7.3	<code>l_ng_ranges</code> setup with 2d measures. . . . .	169
7.4	<code>l_ng_ranges</code> setup with 1d measures. . . . .	171
7.5	<code>l_ng_ranges</code> using scganostics measures and the olive data. . . . .	172
7.6	<code>l_ng_plots</code> setup. . . . .	176
7.7	Frey faces image glyphs. . . . .	177
7.8	Scatterplot of “clupmy” olive data. . . . .	178
7.9	<code>l_ng_ranges</code> setup with <code>measures1d</code> . . . . .	178
7.10	<code>l_ng_plots</code> for <code>scagnostics2d</code> and high sparse and low outlying points selected. . . . .	179
7.11	Implementing a custom graph semantic. . . . .	183
7.12	Spiro glyphs from 3d transitions arranged on a grid. . . . .	186
7.13	Spiro glyphs from 4d transitions arranged on a grid. . . . .	187
7.14	Spiro glyphs from 4d transitions on a scatterplot. . . . .	188
7.15	Zoomed in on spiro glyphs from Figure 7.14. . . . .	188
7.16	Temporarily arranged spiro glyphs in Figure 7.15 on a grid with loon. . . . .	189

8.1 Example of a color cross table of the plot states group vs. color. Note that the first row represents the selected state and not the color state. The data used here is the olive data and its Area variable is assigned to the group state. Also, the color state has a different color for each area in Area. Here, the color of a cell  $(i, j)$  is according to the fraction of points that have the color from row  $i$  out of the points that are from the group in column  $j$ . For the first row the color of a cell represents the fraction of points that are selected for a given group. . . . 199

# List of Tables

1.1	Elementary codes for quantitative information, ranked by the accuracy with which people can extract them from graphs based on experiments from Cleveland and McGill [18]. The slope judgement spans over a range of ranks as the accuracy of judging two slopes relative to each other largely depends on the magnitude of those slopes. Also, the ranking for density, saturation, and hue are not based on experiments, but are conjectural. . . . .	2
2.1	Possible arguments for the user-specified function used in combination with <code>ng_2d_myplot</code> . . . . .	43
4.1	Important Display States. Dominant States do not have a default value. For states of type factor the default value column shows all possible factor levels and highlights the default factor level in bold. . . . .	74
4.2	Dominant states for each Display. . . . .	77
4.3	Functions for temporarily moving points on scatterplot. . . . .	87
4.4	loon’s inspectors . . . . .	88
4.5	loon’s default “used linkable” states. . . . .	92
4.6	linked states for example . . . . .	93
4.7	loon’s layer types . . . . .	95
4.8	Functions that work on layers . . . . .	100
4.9	loon provides <code>l_layer</code> methods for the geospatial data classes in this table. . .	101
4.10	Primitive point/node glyphs. . . . .	104
4.11	Non-primitive point/node glyphs and their creator function. . . . .	104

4.12	Functions for working with glyphs . . . . .	105
4.13	The mapping of size to point glyph area. . . . .	107
4.14	loon’s graphswitch widget. . . . .	109
4.15	Working with navigators. . . . .	111
4.16	Navigator context-related functions. . . . .	113
4.17	Scaling methods for context2d. . . . .	117
5.1	State change binding substitutions. . . . .	127
5.2	Item binding substitutions. . . . .	129
5.3	Item tags for visuals for plots based on the main graphics model. . . . .	130
5.4	Canvas binding substitutions. . . . .	131
5.5	Content binding substitutions. . . . .	132
5.6	Overview binding events. . . . .	133

# Chapter 1

## Background

### 1.1 On High-Dimensional Data

Understanding high-dimensional spaces is intrinsically difficult for humans. Our visual system is limited to perceiving up to three dimensions; therefore, 4+ dimensional spaces are abstract to us. There are many examples that show that high-dimensional spaces can have counter-intuitive properties (Lee and Verleysen [48] and Friedman and Stuetzle [31]). For example, imagine a  $p$  dimensional sphere with a thin shell whose thickness is much smaller than the radius of the sphere. The ratio of the volume of the thin shell to the volume of the interior of the sphere is very small when  $p = 3$ . However, as  $p$  grows the ratio approaches 1. Hence, in high-dimensional spaces, most of the hypersphere volume is located close to its edge. This and many other examples related to “unusual” properties of high-dimensional spaces build upon the fact that, as the dimensionality increases, the space volume grows exponentially. One consequence of the rapid space growth is that the available data can become sparse which, in turn, could limit statistical model fitting (Friedman and Stuetzle [31]). Oftentimes, such problems are collectively referred to as the “curse of dimensionality”, a term coined by Bellman [9].

## 1.2 On Low-Dimensional Views

High-dimensional data are omnipresent and data visualization is widely considered part of good statistical practice for any data analysis. However, finding revealing visualizations for high-dimensional data is oftentimes not a straightforward task. To create a good visualization that helps answer a research question two main decisions have to be made: what is the information to be visualized and how should this information be visually encoded onto a graphic. In other words, what is relevant and how can we see it?

The latter decision can be guided by considering how the information will be decoded by the viewer. Cleveland and McGill [18, 19, 20, 21] rank ten “fundamental geometric, colour, and textural aspects that encode quantitative information” on a graphic based on the accuracy with which people can extract this information. Their ranking, shown in Table 1.1, is based on a number of experiments, but also on some theoretical considerations.

Rank	Code
1	Positions along a common scale
2	Positions along identical, nonaligned scales
3	Lengths
4	Angles
4-10	Slopes
6	Areas
7	Volumes
8	Densities
9	Colour saturation
10	Colour hues

Table 1.1: Elementary codes for quantitative information, ranked by the accuracy with which people can extract them from graphs based on experiments from Cleveland and McGill [18]. The slope judgement spans over a range of ranks as the accuracy of judging two slopes relative to each other largely depends on the magnitude of those slopes. Also, the ranking for density, saturation, and hue are not based on experiments, but are conjectural.



For continuous data, a two-dimensional scatterplot performs well in the ranking in Table 1.1. That is, the values of each variate are encoded as positions along a common scale. Also, the pattern of the point glyphs (i.e. the geometrical representation of a data point on the plot) can expose a possible relationship between the two variates. Additional information could be encoded onto a scatterplot by modifying the point glyph style which is usually a filled circle by using color, size or special glyphs such as star glyphs. However, according to Table 1.1, such additional information will be poorly encoded compared to information that is decoded as the scatterplot coordinates.

For three-dimensional data, a three-dimensional scatterplot or point cloud would perform well according to Table 1.1. To visualize a 3d scatterplot one either needs a three-dimensional display or has to project the point cloud onto a 2d plane. However, a perspective projection will encode depth with shape or size of the glyphs, whereas with an orthogonal projection the depth information gets lost.

In conclusion, it is desirable to visualize some low-dimensional representation of the data that captures relevant information.

Many visualization techniques take the approach of laying out multiple low-dimensional plots either spatially or temporally so that the data analyst can visually link them. Scatterplot matrices are the best known of these multivariate visualization techniques. A scatterplot matrix for  $p$ -variate data lays out the  $p(p - 1)$  scatterplots of every ordered variable pair onto a  $p \times p$  grid such that the scatterplots within a column or row share their  $x$ -axis or  $y$ -axis, respectively.

Elmqvist et al. [25] propose the use of scatterplot matrices as navigational infrastructures to guide a 3d rigid rotation. The resulting projections of the 3d space onto a 2d subspace will then be shown on a 2d scatterplot display and appears as a smooth movie, i.e. the viewer can track the points. This way, the visual linkage between plots is done spatially and temporally.

The idea of looking at linear projections rather than just plotting the original variates has been used for a long time. For example, Friedman [30, p. 249] writes that “Any structure seen in a projection is a shadow of an actual (usually sharper) structure in the full dimensionality”.

Friedman and Tukey [32] introduced the term *projection pursuit* as an algorithm that seeks highly revealing low-dimensional linear projections of multivariate data. This algorithm requires the definition of a projection index which is a measure corresponding to some feature of interest in the subspace defined by the projection (e.g. spread of points). Projections that optimize this index are then visually inspected by the analyst. Local optima for a projection index are also of interest as they may reveal relevant information that cannot be seen in the globally optimal projection.

Another approach to finding interesting projections involves looking at all possible projections or at least at a dense subset of them. If the projections from the dense set are ordered such that the changes in projections are small then the resulting visualization will be a smooth movie. Asimov [5] proposed this approach and called the sequence of such projections a “grand tour”. However, even for a small number of dimensions, such a grand tour can take a very long time to watch. Huber [38, p. 438] argues that if some “interesting” features can only be seen within a “squint angle” of about  $10^\circ$  then a grand tour in four dimensions would take about 3 hours.

With projection pursuit one can hope to find an interesting projection in a more timely fashion. But the value of watching a smooth movie of moving points in a scatterplot is compelling. Buja and Asimov [12] argue that

the speed vectors of data points in a grand tour provide two additional dimensions of information in addition to the two dimensions of location, thus letting us perceive a full 4-dimensional space at any given point in time.

Cook et al. [23] discuss some of the shortcomings of projection pursuit and grand tours

Unfortunately static plots suffer from a lack of context because they have been removed from their neighborhood in the projection space, and although a grand tour provides the neighborhood context it has a tendency to spend too much time away from, or indeed never visit, the interesting projections.

As an alternative, Cook et al. [23] propose projection pursuit guided tours in an effort to combine the benefits of projection pursuit and grand tours into a dynamic graphical

tool. Projection pursuit guided tours display a grand tour movie until the data analyst decides to use a particular projection as the starting point of a projection pursuit. The movie continues for the projection index optimization, showing the projection for each optimization iteration.

In summary, high-dimensional data can be difficult to comprehend, but good visual exploration can help overcome this issue by revealing relevant information in the data. One approach for visualizing high-dimensional data is to lay out different low-dimensional views either spatially or temporally so that the analyst can visually link them. Well known examples of this approach include scatterplot matrices, projection pursuit and grand tours.

### 1.3 Navigation Graphs

Hurley and Oldford [44] propose using graphs that are sets of vertices (nodes) and edges to navigate a view of the data. The nodes of such navigation graphs represent low-dimensional spaces whereas the edges represent some transition between those spaces. That is, every location on the graph defines a view of the data. In a graphical user interface, these graphs become navigable when a “bullet” is placed onto the graph. The location of the bullet can be then linked to some display showing a particular view of the data. In this thesis, we call the relationship between a location on the navigation graph and the corresponding view the “graph semantic”.

For example, in Hurley and Oldford [44] the graph semantic of their main example associates every location on the graph with an orthogonal projection of the data onto a two-dimensional subspace. The nodes then represent the subspace spanned by a pair of variates and the edges either a 3d- or 4d-transition of one scatterplot into another, depending on how many variates the two adjoining nodes share. Hereafter, we refer to this particular graph semantic as the *canonical graph semantic*.

Hurley and Oldford [44] use the analogy between a navigation graph and a city map. One can explore a new city by either randomly driving around (grand tour), by always choosing the most interesting looking path at each intersection (projection pursuit) or by using a city map with marked routes that show regions of interest (navigation graphs).

In the light of this analogy, when exploring a new high-dimensional data set, we first need to create an accurate map that highlights the most interesting regions of the data. For the canonical graph semantic applied to the original data variates, the most exhaustive navigation graph that can be created is a complete graph with  $\binom{p}{2}$  nodes representing the unordered variate pairs of a  $p$ -dimensional data set. Using an exhaustive graph does not guarantee showing an interesting view of the data as interesting 2d projections could be lying on arbitrary 2d subspaces. However, we believe that exploring the original variates before transforming the data is important. In general, a model that explains a phenomenon based on few original (and easily interpretable) variates is preferred over a model that includes arbitrary linear combinations of the variates.

Using navigation graphs encourages the analyst to spend time on visually exploring the data similarly to playing a captivating game. Dragging a bullet along a navigation graph is an easy task and the immediate result is a smooth movie that can provide valuable insight into the data.

In the remainder of this chapter, we give a detailed overview of the theory and application of navigation graphs as proposed by Hurley and Oldford [44]. Later on we review some tools that can be used to find small and relevant navigation graphs.

### 1.3.1 The Canonical Graph Semantic

The navigation graph framework was introduced by Hurley and Oldford [44] as a general concept; nodes represent views and edges represent a smooth morphing from one view into another. However, most of the theory and examples in [44] were built around the canonical graph semantic linking every location on a navigation graph to a 2d subspace. Graphs whose nodes represent 2d subspaces are called *2d space graphs*. A 2d space graph that has only edges connecting two nodes whose union span a 3d space is called a 3d transition graph. Similarly, if the graph edges only connect two nodes whose union span a 4d space then the graph is called a 4d transition graph. In addition, when an edge exists for any possible 3d or 4d transition we call such a graph a *saturated* 3d or 4d transition graph, respectively.

For illustrative purposes, we use the Italian olive oil data set from Forina et al. [29]. These data record the percentage composition of 8 fatty acids found in the lipid fraction of 572 Italian olive oils sampled from 9 different areas (see Figure 1.1).

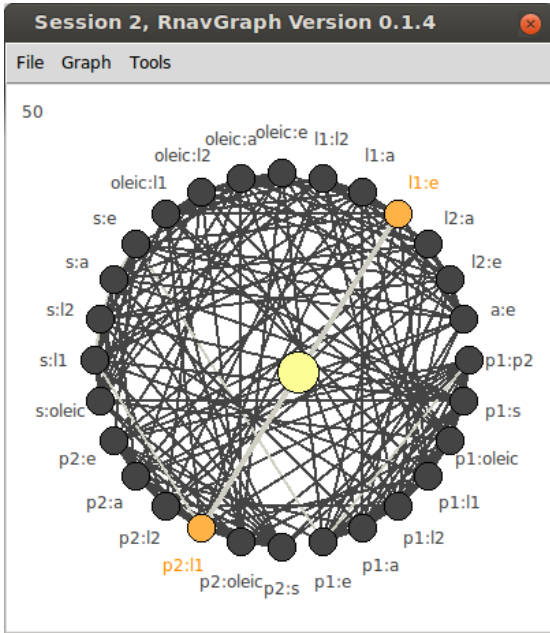


Figure 1.1: The Italian olive oil data – recorded fatty acids and sampled areas.

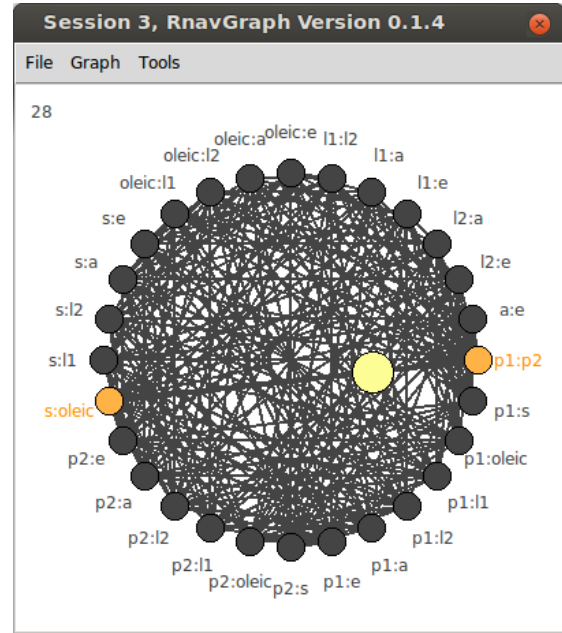
For the olive data, there are  $\binom{8}{2} = 28$  unique unordered variate pairs and hence 28 possible nodes in a navigation graph with the canonical semantic. Figure 1.2 shows the saturated 3d and 4d transition graphs. We call the large yellow circle in each navigation graph a “bullet”. The bullet location defines the projection to be visualized in a linked scatterplot display and can be interactively dragged along the graph.

Regarding the particular implementation of a 3d and 4d transition, Hurley and Oldford [44] propose to rotate the starting plane along a geodesic path into the target plane, as described by Hurley and Buja [41].

For a 3d transition, this rotation results in a 3d rigid rotation. For example, the location of the bullet in Figure 1.2(a) is 30% between the two 2d spaces (stearic, eicosenoic) and (archidic, eicosenoic). Hence, the linked scatterplot display shows the orthogonal projection of the 3d point cloud onto a plane through the eicosenoic axis that is 30 degrees rotated from the stearic axis towards the archidic axis, as shown in Figure 1.3(a) and 1.3(b). Moving the bullet from the (stearic, eicosenoic) node towards the (archidic, eicosenoic) node is equivalent to rotating the  $x - y$  plane from the (stearic, eicosenoic) axis into the (archidic, eicosenoic) axis around the eicosenoic axis.



(a) saturated 3d transition graph



(b) saturated 4d transition graph

Figure 1.2: Saturated 3d and 4d transition graphs for the olive data.

For a 4d transition, Hurley and Oldford [44] interpolate the plane along the geodesic path which in this setting is equivalent to orthogonally rotating one variate into another, for both  $x$  and  $y$  axes. This is illustrated in Figure 1.4 and in practice will result in a smooth movie that is less intuitive than the one from a 3d rigid rotation.

## 1.3.2 Automatic Graph Construction and Exploration

### 1.3.2.1 Saturated 3d and 4d transition graph

Saturated 3d and 4d transition graphs, as shown in Figure 1.2 for  $p = 8$ , can be constructed automatically for any  $p$ -dimensional data set by using graph theoretic algorithms.

A *variable graph* associates one variate to each of its nodes. An edge of such a variable graph represents a pairing of the two variates; for example, an edge could represent the correlation between the two variates. For any variable graph,  $G$ , the *line graph* of  $G$ ,

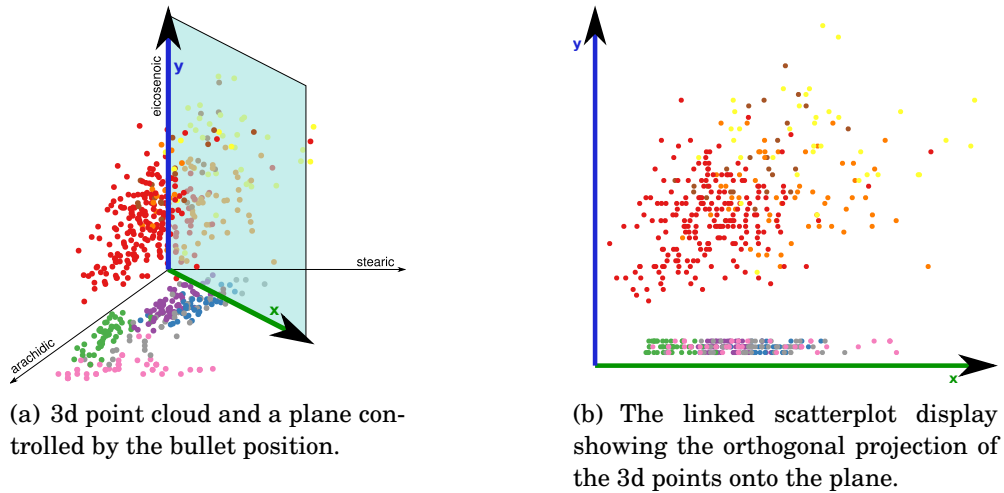


Figure 1.3: 3d rigid rotation and linked scatterplot display.

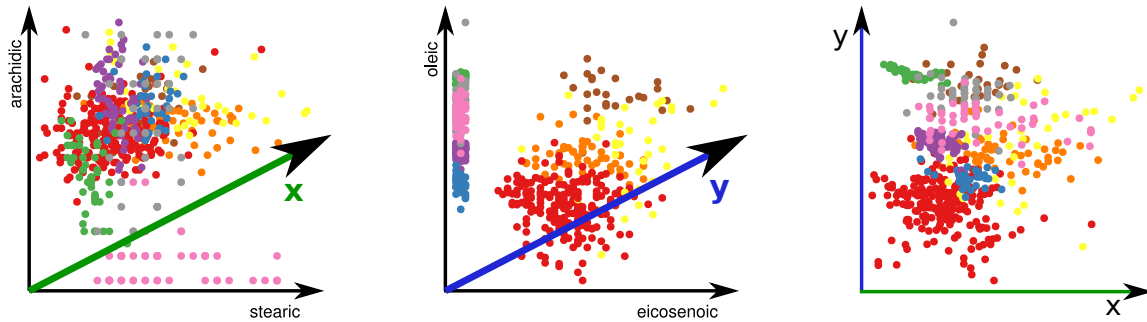


Figure 1.4: 4d transition along a geodesic.

$L(G)$ , constructs a saturated 3d transition graph.  $L(G)$  turns every edge of  $G$  into a node of  $L(G)$ , and nodes in  $L(G)$  are adjacent if the corresponding edges in  $G$  share a node. Hence,  $L(G)$  will show every variable pairing of interest (i.e. an edge in  $G$ ) as a node and every possible 3d transition for these nodes. Further, the complement of  $L(G)$ ,  $\overline{L(G)}$ , constructs a saturated 4d transition graph.

Figure 1.5 shows a complete variable graph  $G$ , its line graph  $L(G)$  and the complement of the line graph  $\overline{L(G)}$  for four variates labelled  $A$ ,  $B$ ,  $C$  and  $D$ . In this example, the

variable graph  $G$  is saturated and hence the relationships between all paired variates are of interest.

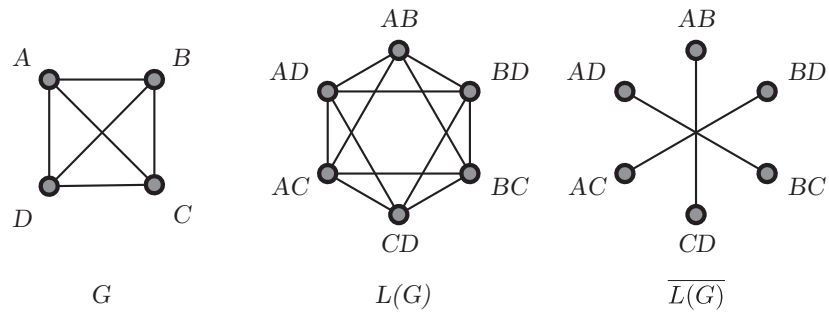


Figure 1.5: Complete variable graph  $G$  for a 4 dimensional data set with variates labelled  $A, B, C,$  and  $D$ . Line graph  $L(G)$ . Complement graph  $\overline{L(G)}$ .

If the variable graph is not complete such as in cases where certain relations are not of interest then the corresponding line graph and its complement are still saturated 3d and 4d transition graphs, but with fewer nodes. For example, the variable graph  $G$  in Figure 1.6 does not have the edges  $(A,C)$  and  $(A,D)$  which indicates that these variable pairings are not of interest here.

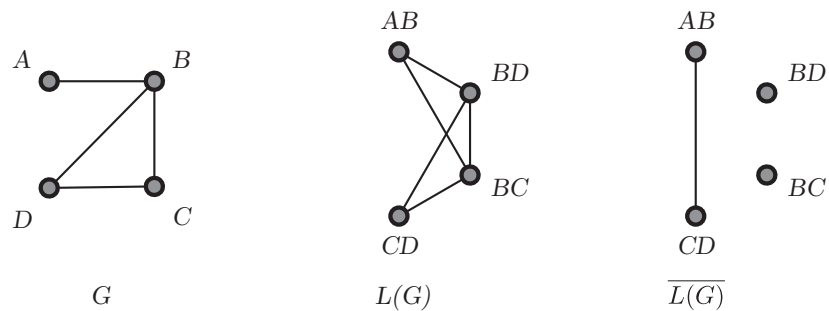


Figure 1.6: Automatic saturated 3d and 4d transition graph construction with a non-complete variable graph.



### 1.3.2.2 Graph Products

Graph products of two graphs  $G$  and  $H$  are useful for constructing transition graphs that preserve certain structures of the initial graphs  $G$  and  $H$ . Of particular interest here are statistically meaningful examples where

- 1) the variates separate into two distinct sets and
- 2) there may be additional structure between the variates within each set that can be captured by a graph.

Hurley and Oldford [44] give several such examples for

- 1) variates separating into two distinct sets
  - response vs. explanatory
  - endogenous vs. exogenous
  - design variates vs. covariates
  - original vs. derived
  - causal vs. associated
- 2) structure between variates can be captured by a graph
  - time ordering
  - regression structure
  - Markov property
  - conditional independence
  - path diagram

We now briefly review two concrete examples from Hurley and Oldford [44] that illustrate the use of graph products.

Figure 1.7 shows two variable graphs  $G$  and  $H$  that both capture some ordering of the variates within the graph. All three graph products in Figure 1.8 preserve some of the structure in  $G$  and  $H$ ; in particular, these graph products have only edges that are permitted by  $G$  and  $H$ . In addition, the Cartesian product  $G \square H$  has only edges representing 3d transition, the Tensor product  $G \times H$  has only edges representing 4d transitions, and the strong product  $(G \boxtimes H)$  equals  $(G \square H) + (G \times H)$ , as the notation suggests. Hence,  $G \square H$

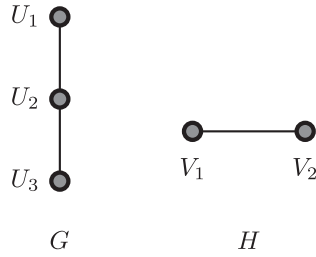


Figure 1.7: Two variable graphs  $G$  and  $H$  capturing a certain structure.

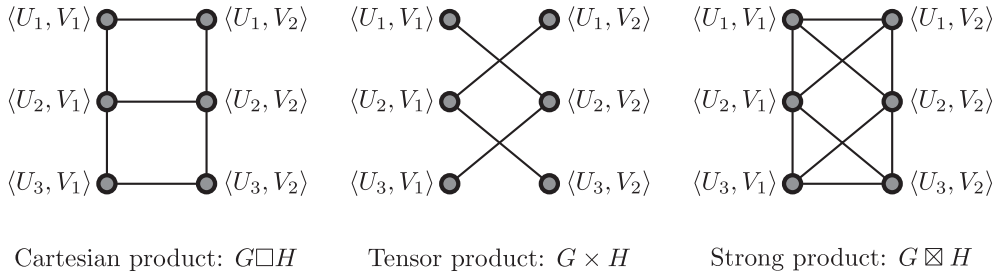


Figure 1.8: Three graph products of  $G$  and  $H$  from Figure 1.7

is a restricted 3d transition graph,  $G \times H$  a restricted 4d transition graph, and  $G \square H$  and  $G \times H$  are complements in  $G \boxtimes H$ .

Another example of using graph products given in Hurley and Oldford [44] is shown in Figure 1.9, where the restricted transition graphs  $G \square H$ ,  $G \times H$  and  $G \boxtimes H$  correspond to moving within a rectangle in a scatterplot matrix. Note that the rectangular shape in Figure 1.9 is due to  $G$  and  $H$  being complete graphs.

### 1.3.2.3 Automatic Graph Traversal

In addition to automatic graph construction, Hurley and Oldford [44] explore algorithms that construct meaningful paths and cycles on a graph to automatically traverse the graph. For example, a Hamiltonian path visits each node exactly once. A Hamiltonian cycle is a Hamiltonian path concatenated with the start node of that path. An Eulerian path visits each edge exactly once. When they exist, cycle decompositions, Hamiltonian decom-

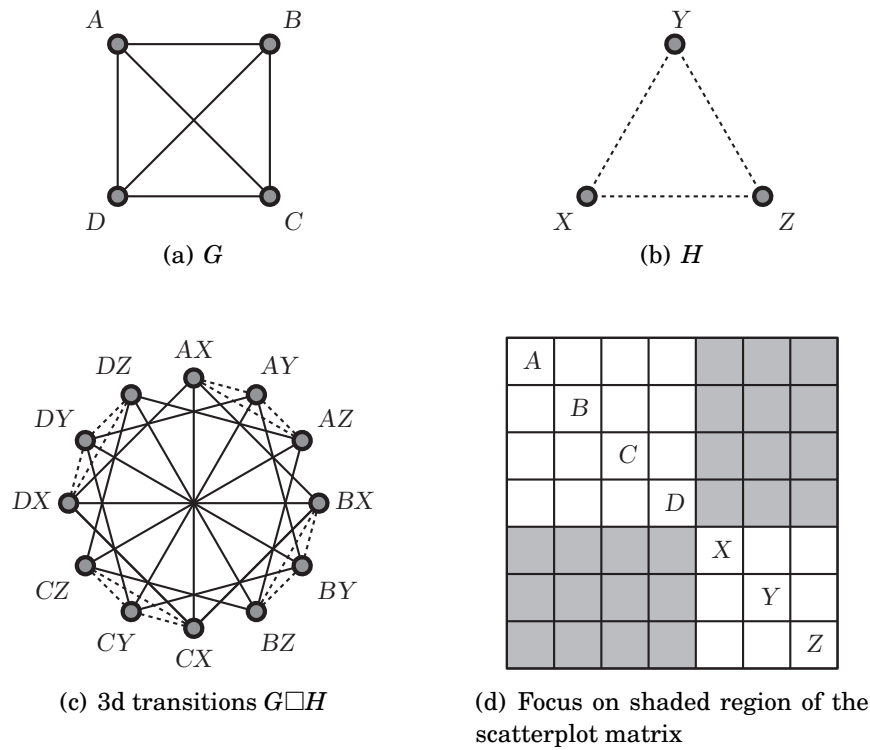


Figure 1.9: Scatterplot matrix example from Hurley and Oldford [44] Section 4.1.

positions, 2-factorizations and Eulerian tours might also be of interest (see Hurley and Oldford [44]). Greedy algorithms such as a greedy Eulerian might be of additional value for traversing weighted transition graphs (e.g. visiting the “most interesting” edges first, where what is interesting is expressed with the weights). We discuss possible weights in more detail in the next section.

## 1.4 On the Problem of Large Graphs

The saturated 3d and 4d transition graphs for the 8-dimensional olive data, as shown in Figure 1.2, have 28 nodes and 168 and 210 edges, respectively. Analyzing these  $168+210 = 378$  transitions would require a long concentration span from an analyst. The problem gets even harder with increased dimensionality; for example, for 15 dimensions the total

number of edges of a saturated 3d and 4d transition graph is  $\binom{15}{2} = 5460$  which for a fast edge transition of 1 second would take over 1.5 hours to explore.

As mentioned in Section 1.3, navigation graphs should optimally be “road maps” with only interesting routes regardless the dimensionality of the data. In addition, even for data with thousands of dimensions, it would be great to be able to construct some small navigation graphs that capture the essential patterns of the data.

Next, we discuss two ways to deal with large navigation graphs. First, we look at ways to find interesting subgraphs that could reveal relevant information. Second, we discuss methods for creating a reduced new set of variates that capture interesting patterns in the data; these methods are commonly called dimensionality reduction algorithms.

### 1.4.1 Finding an Interesting Subgraph

An explorative data analysis using a complete 2d space graph and the canonical graph semantic is equivalent to using an interactive scatterplot matrix as proposed by Elmqvist et al. [25]. However, navigation graphs are a more powerful navigational infrastructure; that is, aside from the fact that navigation graphs allow for a general  $k$ d space navigational framework using any imaginable graph semantic, they also allow for the reduction of the complete 2d space graph to any arbitrary subgraph.

Finding an interesting subgraph of a saturated 3d or 4d transition graph is preferable to dimensionality reduction if the original variates are important to interpret and communicate the results. Although finding an interesting subgraph can be done manually, we will focus here on automated methods based on  $k$ d space measures.

Any measure that can be associated with either the nodes or the edges of some 2d space graph could be used to determine a subgraph. For example, regression coefficients, projection pursuit indexes and *scagnostic measures* (Wilkinson et al. [83] and Wilkinson and Wills [85]) are all useful 2d space measures that also have a statistical meaning. One could use these 2d measures to determine edge weights on a complete variable graph and, consecutively, construct a saturated 3d or 4d transition graph from a variable graph whose edge weights are greater than some threshold  $w_0$ , i.e.  $L(G(w > w_0))$  and  $\overline{L(G(w > w_0))}$ .

Scagnostic measures are particularly interesting for the canonical navigation graph semantic as they are specifically designed to assess some characterizations of 2d scatterplots. The concept of scagnostic measures is similar to the one of projection pursuit indices; John W. Tukey was involved in defining both concepts. However, scagnostic measures and projection indices differ in their motivation; that is, finding interesting scatterplots vs. interesting projections. John and Paul Tukey discussed the concept of scagnostics in [77] but have never published details about their particular scagnostic indices. About 20 years later Wilkinson et al. published nine graph-theoretic scagnostics measures [83, 85].

These scagnostic measures are named according to the properties of the scatterplots patterns that they are measuring: outlying, skewed, clumpy, convex, skinny, striated, stringy, straight and monotonic. For example, the monotonic measure is the squared Spearman correlation coefficient and it is the only scagnostic measure that is not based on geometric graphs. All the other measures are derived from the following 2d Euclidean geometric graphs where nodes lie on a 2d space: the convex hull, alpha hull and the minimum spanning tree. For example, the stringy measure is the ratio of the diameter to the length of a minimal spanning tree. Prior to constructing these geometric graphs, identified outliers (see Wilkinson et al. [83]) are deleted in order to keep the scagnostic measures robust and the scatterplot points are binned using adaptive hexagonal binning (Carr et al. [14]) in order to improve computational performance. All scagnostic measures are designed to lie within the closed unit interval.

Figure 1.10(a) shows the scatterplot of the convex vs. the monotonic scagnostic measure for all 28 scatterplots of the original olive variates. That is, each point represents one of the  $\binom{8}{2}$  scatterplots of the olive data. Figure 1.10(b) shows the two scatterplots corresponding to the two coloured dots in Figure 1.10(a).

A computationally more involved way of finding an interesting subgraph based on 2d measures is to calculate those measures for a dense set of 2d projections along the 3d or 4d transition. For example, Figure 1.11 shows how the convex and monotone scagnostic measures change during the two 4d transitions from (palmitoleic, oleic) into (linolenic, arachidic) and from (palmitoleic, oleic) into (arachidic, linolenic). Given such a transition trajectory of the scagnostic measures, one could then make up rules on whether to include or not an edge as part of a smaller (sub)graph. Also, Fu and Oldford [33] propose

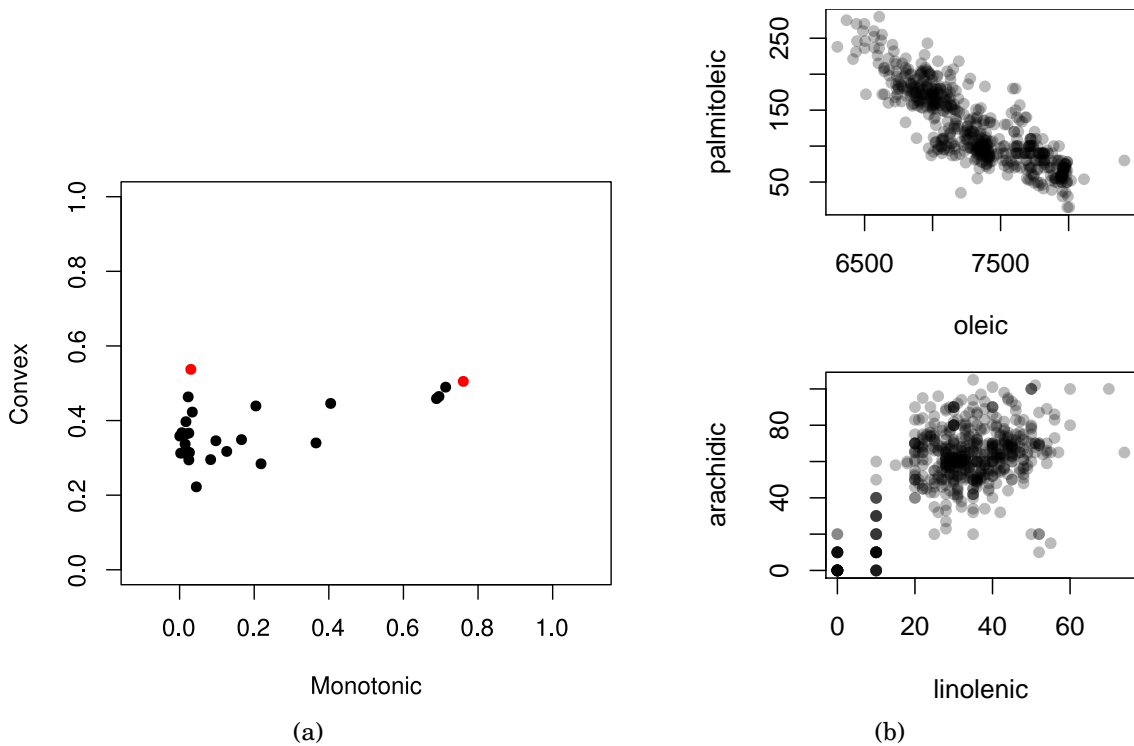


Figure 1.10: a) Scatterplot of the scagnostic measures – convex vs. monotonic. b) Particular scatterplots with high monotonic and convex measure, and high convex and low monotonic measure.

3d scagnostic measures that could be used as edge weights of 3d transition graphs.

## 1.4.2 Dimensionality Reduction/Constructing Dimensions

Finding subgraphs for very high-dimensional data such as image data (i.e.  $n$  images) might not be a computationally feasible option. Even a very small  $32 \times 32$  pixel greyscale image such as this one 🐱 has 1024 dimensions with  $\binom{1024}{2} = 523,776$  possible scatterplots. An alternative to finding interesting subgraphs is to construct fewer new dimensions using a dimensionality reduction technique.

Dimensionality reduction is a large research field and many techniques have been proposed especially in the last two decades. One major motivation for dimensionality reduction has always been to visualize high-dimensional data. Hence, many such methods arose from geometric intuition and the results are usually interesting to look at.

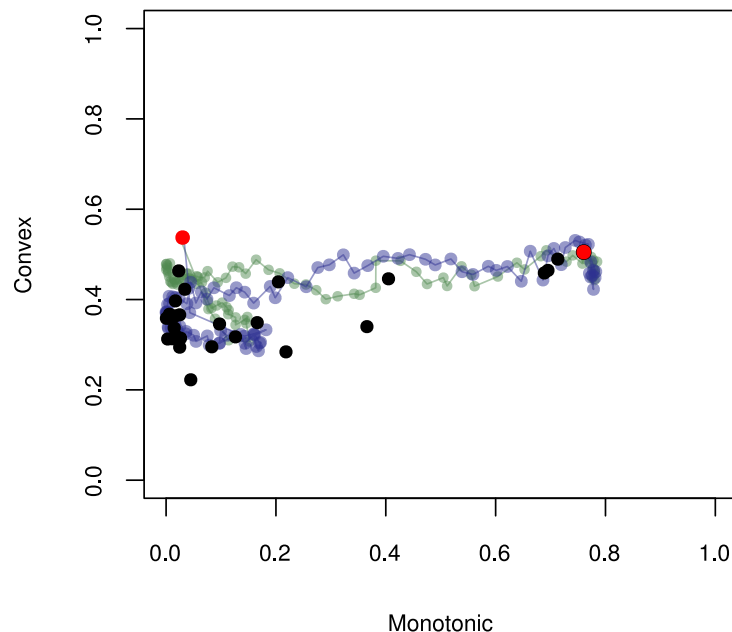


Figure 1.11: Scatterplot of the scagnostic measures – convex vs. monotonic – overlaid on the scagnostic measures along two 4d transitions from the olive data. The blue path shows the scagnostic measures when linolenic transitions into oleic and arachidic transitions into palmitoleic. The green path shows the scagnostic measures when linolenic transitions into palmitoleic and arachidic transitions into oleic.

We now review some popular dimensionality reduction techniques that originate from interesting geometric motivations. That is, we provide the geometric intuition for principal component analysis, Fisher discriminant analysis, multidimensional scaling, kernel methods and two manifold learning methods, isomap and locally linear embedding.

Principal component analysis (PCA) goes back to Pearson in 1901. Today, PCA is probably the best known dimensionality reduction method. There are two ways of looking at PCA that lead to the same algorithm. A geometric motivation defines the principal directions as a basis of a hyperplane for which the average squared distance between the points and their projection onto this hyperplane is minimized. Alternatively, principal directions are the orthogonal directions with maximum variance of the projected data points. Hence, the orthogonal projections of the data onto the first principal direction has maximum vari-

ance among all possible orthogonal projections. Further, the original axes can be projected onto the principal directions and visualized with the projected points. Plots that include observations and variates in the context of PCA are called biplots and were proposed by Gabriel [34].

For labelled data, such as the olive data with its area label, one can also use a supervised dimensionality reduction method. One such method is Fisher Discriminant Analysis (FDA) (Fisher [26]) which was originally defined for two-class data and then generalized by Rao [61, Sec. 9c] for multi-class data. FDA seeks a direction for which the orthogonally projected data have a minimal within-group variance and a maximal between-group variance. In general, it is possible to find  $j$ , where  $j \leq p$  for  $p$  dimensional data, mutually orthogonal directions that separate the groups best. Gnanadesikan [37] calls these directions the *discriminant coordinates* or CRIMCOORDS. Hence, dimensionality reduction is achieved by projecting the data onto the first  $j$  discriminant coordinates.

Multidimensional scaling (MDS) refers to a set of dimensionality reduction techniques that find a low-dimensional embedding or configuration of  $n$  objects in a geometric space (usually Euclidean) so that their interpoint distances correspond to the observed dissimilarities (or proximities) between the objects. Examples of proximities include correlations, similarity ratings, travel times and metric distances. Hence, MDS can be used with pairwise dissimilarity data in addition to multivariate data. There are linear and non-linear MDS variants.

Kernel methods map some  $p$ -dimensional data into a  $m$ -dimensional *feature space* with the goal of applying a linear method such as PCA, regression, or support vector machines, in this feature space. Let  $\varphi : \mathbb{R}^p \rightarrow \mathbb{R}^m$  denote the mapping into the feature space where usually  $m > p$  ( $m$  can also be infinite). The “kernel trick” allows us to determine the  $m$  optimal parameters  $\mathbf{w}$  of the linear model  $\mathbf{Y}\mathbf{w}$  in the feature space without having to explicitly evaluate the mappings  $\mathbf{y}_i = \varphi(\mathbf{x}_i)$ , but rather through the inner product evaluations  $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \langle \varphi(\mathbf{x}_i), \varphi(\mathbf{x}_j) \rangle$ , called kernel evaluations. The *kernelization* of a linear method involves first finding a dual representation of the optimal parameters  $\mathbf{w}$  so that the optimization is  $n$ - rather than  $m$ -dimensional. Next, the optimal solution has to be expressed in terms of the data variates only in the form of pairwise inner products. For example, Schölkopf et al. [64] first introduced the “kernelization” of PCA and Baudat and



Anouar [6] have shown how to kernelize the Fisher discriminant analysis for multi-class data.

Isomap (Tenenbaum et al. [74]) and locally linear embedding (Roweis and Saul [62]) are two prominent nonlinear dimensionality reduction algorithms in the manifold learning domain. In manifold learning, it is assumed that the data at hand lie on a manifold embedded in the variable space of the data. Optimally, the dimensionality of the manifold is the same as the *intrinsic dimensionality*. For example, for a set of images portraying an object from different perspectives, the intrinsic dimensionality is defined by the camera position which is the only thing that changes during the image recording. If such images have each  $p$  pixels then, even for small monochrome images of size  $50 \times 50$  pixels, the dimensionality of the data will be 2500. However, the intrinsic dimensionality is maximum 3, assuming the camera always points towards the object and maintains the same distance to the object.

Isomap has the goal to preserve the geodesic manifold distances between data points. These distances are approximated by the length of the shortest path along points that are in close proximity. To do so, a graph  $G$  is constructed with nodes representing the data points. The nodes in  $G$  get connected if their associated points lie close together. Closeness can be defined as either an absolute distance threshold (i.e.  $\epsilon$  region) or as the  $K$  nearest neighbours. The edge weights are defined as the distances between the points of their corresponding nodes. The geodesic manifold distance is then approximated by the length of the shortest path between each pair of nodes in  $G$ . This distance measure is then used to create a low-dimensional embedding using classical multidimensional scaling.

Locally linear embedding (LLE) (Roweis and Saul [62]) tries to preserve the distances between points in a small neighbourhood. That is, LLE assumes that, given enough points, small neighbourhoods such as the  $K$  nearest neighbours are well approximated by linear manifolds.

Some of these dimensionality reduction methods are interrelated, similar, or even equivalent under certain parametrization, see Maaten et al. [49, sec. 5.1].

We find that dimensionality reduction and navigation graphs have a mutually beneficial relationship. That is, oftentimes analysts conveniently choose to reduce the dimen-

sionality of the data to two or three dimensions in order to visualize them, whereas using navigation graphs easily accommodates the exploration of 5 to 20 dimensions. In addition, navigation graphs also facilitate the exploration of results from multiple dimensionality reduction methods in parallel. For example, [Figure 1.12](#) shows a 3d navigation graph with 7 “bullets” that each drive a scatterplot display in [Figure 1.12\(a\)](#) using the canonical graph semantic. The data for the projections displayed in these scatterplots come from 7 different dimensionality reduction methods that were used to reduce the olive data to 5 dimensions. The dimensionality reduction methods included in this example are: principal components analysis (pca), linear discriminant analysis (lda), a variant of Fisher’s discriminant analysis, (classical) linear multidimensional scaling (lmlds), non-linear multidimensional scaling (nlmlds), kernel principal components (kpca), isomap and locally linear embedding (lle). For lda we used the Area labels of the olive data as group classifiers, for both lmds and nlmlds we used the Euclidean inter-point distances as dissimilarity measures, for kpca we used a polynomial kernel of degree 3, for isomap we used the 6 nearest neighbors for calculating the shortest paths, and for lle we used the 6 nearest neighbors to define a small neighborhood. This setting was created with loon, a software we developed for interactive data visualization and introduced later in [Chapter 3](#). The scatterplots in [Figure 1.12\(b\)](#) are linked so that points representing the same olive oil share some visual attributes such as color and size. By working with different dimensionality reduction methods, navigation graphs and an interactive setting, one can compare the results of different dimensionality reduction methods directly and in real time. The R code to recreate this setting can be found in the `l_ng_dimred` demo of the loon R package. This demo can be run as follows:

```
library(loon)
demo('l_ng_dimred')
```

Generally, any method producing a lower dimensional embedding can be used to create a smaller navigation graph. In turn, an even smaller sub-graph can be found using the methods discussed in [Subsection 1.4.1](#).

## 1.5 Other Graphs of Possible Interest

The navigation graph framework is more general than the canonical graph semantic; that is, graphs provide a navigable infrastructure that track a real time morphing from one display of a set of variates into another. Any display on the variates associated with each node would work as long as the graph semantic of an edge transition is defined.

Hurley and Oldford [44] propose some alternate ideas for the semantic of edge transitions. For example, a 3d transition graph could also be used to control a conditioning on the common variates. This conditioning is known as slicing; that is, a transition along the edge  $(AB, AC)$  could show the scatterplot of  $C$  vs.  $B$  for the subset of the data for which  $A$  is within a certain range determined by the bullet position. Slicing controlled by a navigation graph can be extended in a straightforward manner to categorical data. The scatterplot could then be replaced by a mosaic plot or eikosogram (Cherry and Oldford [16]).

Navigation graphs can also represent more than two dimensions on each node. Navigation graphs with nodes representing a set of  $k$  variates are called  $kd$  space graphs. Let  $S(p, k)$  denote a complete  $kd$  space graph for  $p$  variates.  $S(p, k)$  can be decomposed into graphs  $S(p, k, i)$  whose edges connect nodes that share exactly  $i$  variates, and

$$S(p, k) = \sum_{i=0}^{k-1} S(p, k, i).$$

For example, edges on a 3d space graph with 4d transitions (i.e. some subgraph of  $S(p, 3, 2)$ ) can define a rotation of two disjoint variate sets. This rotation can be visualized with a 3d point cloud. Alternatively, the common variates can be used for conditioning in slicing.

If the edges in a 3d space graph define 6d transitions (i.e. their connected nodes do not share any variates) then a transition can represent a morphing of a scatterplot matrix into another using 4d transitions as described in section 1.3.1. For example, a transition from the scatterplot matrix of  $ABC$  into  $DEF$  can be visualized by transitioning  $AB$  into  $DE$ ,  $AC$  into  $DF$ , and  $BC$  into  $EF$ .

In a 4d space graph with only 6d transitions such as  $ABCD \rightarrow CDEF$ , it is possible to dynamically morph Cleveland’s conditional plot (Cleveland [17]) of  $(A,B)|(C,D)$  into that of  $(E,F)|(C,D)$ .

## 1.6 Interactive Data Visualization Software

We wrote the RnavGraph software (see Chapter 2) that implemented a user interface for navigation graphs by providing a “bullet” on a graph to drive the transitions, as proposed by Hurley and Oldford [44]. With RnavGraph our goal was to interactively analyze data in the R statistical environment [60] using navigation graphs. While working on the interactive graph we investigated various options for readily available interactive scatterplots in R; it became obvious to us that interactivity for both the graph and the scatterplot display – or any display for that matter – was important for effectively using navigation graphs.

There is a long history of the design and development of interactive visualization software for exploratory data analysis dating back to at least PRIM [27] in 1973. Other examples include Quail [43], Lisp-stat [76], Plot Windows [67], DINDE [58], DataDesk [80], Data Viewer [40], the gobi family [68, 13, 69, 47], iplots [79] and Mondrian [75]. Among other features, these systems provide a scatterplot that supports the following features: dynamic zooming and panning via mouse gestures and some form of brushing and linking (we are not completely sure about PRIM and linking).

These systems take different approaches to providing interactive data visualizations graphical user interfaces (GUI). PRIM, Data Desk, Data Viewer, the gobi family and Mondrian provide in essence an encapsulated environment to visualize and explore data. That is, they have limited or no connection to a complete statistical system with a major user community such as R. Hence, creating new plots and control widgets dynamically from a command line interface and incorporating various statistical analyses is not possible with these systems. On the other hand, Quail and Lisp-stat do support dynamic creation and incorporation of statistical analyses, but they are not integrated into a complete statistical system; adding new statistical tools to Quail or Lisp-stat involves their respective authors having to write these tools first, see for example Anglin and Oldford [4].

Finally, `iplots` was designed to bring interactive graphics to the R environment. However, `iplots` uses actions in menus that cannot be controlled via the command line.

For `RnavGraph`, we first used the interactive scatterplot display of `Ggobi` via the `rggobi` R package [47]. However, we were missing some important features such as advanced point glyphs for the scatterplot display including images, text and star glyphs. We also found that installing `rggobi` was difficult on certain operating systems which would have limited the potential users of `RnavGraph` package. We were frustrated with not having interactive tools whose value in exploratory data analysis has long been known (at least 20 or more years ago [70, 51, 7, 42, 2]) that were integrated with commonly used and statistically rich set of more formal analysis tools (as provided for example by the open source system, R). This frustration is shared by others. At a recent R users conference, Di Cook [22] shared her frustration and listed the following “challenges to the young developers”:

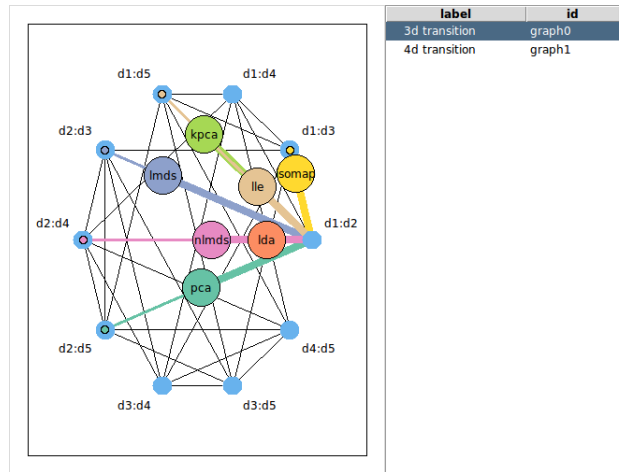
- Interactivity on the plot
- Different types of brushes
- Different kinds of linking between plots
- Programmability
- Strong connection with model fitting
- Portability, easy install, web compatible
- Large quantities of data
- Incorporating inference
- Conceptual framework

We ended up writing our own interactive scatterplot display `tk2d` as part of the `RnavGraph` R package, see [Section 2.3](#). Motivated from the results of `tk2d` we took up designing and implementing a new interactive general-purpose visualization system called `loon`. We reflect in [Chapter 8](#) on how `loon` meets the challenges set by Di Cook.

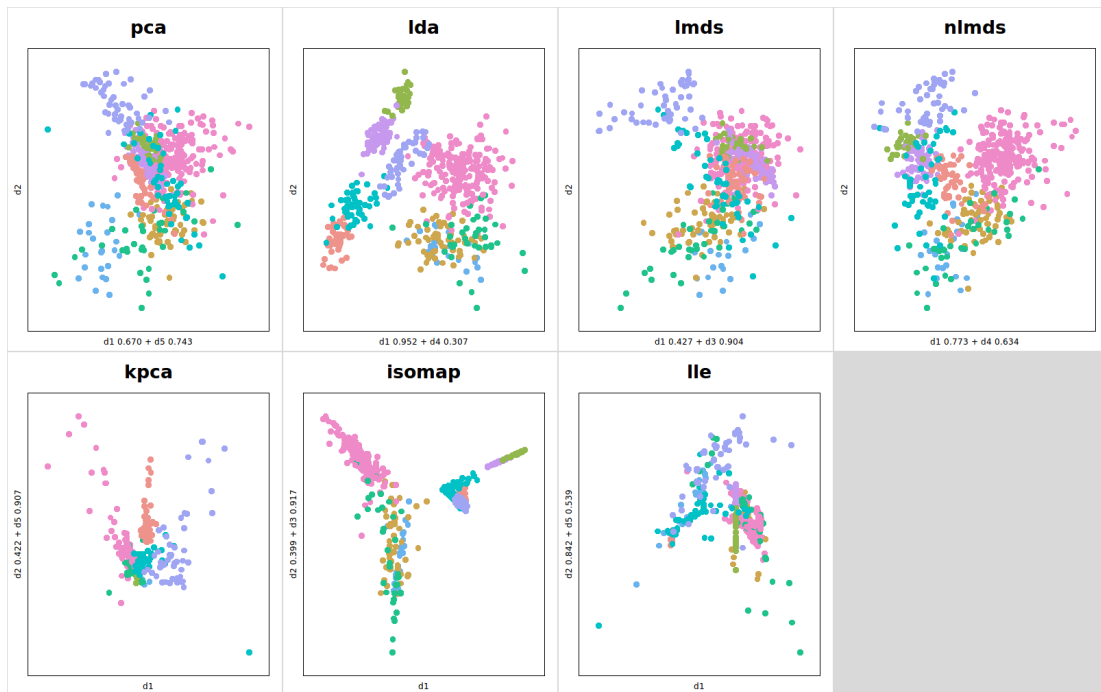
This thesis is structured as follows. In [Chapter 2](#), we discuss `RnavGraph`, a software environment for interactively exploring data using navigation graphs. In [Chapter 3](#), we

present a visual exploratory analysis of the visible minority populations distributed across major census metropolitan areas of Canada. We highlight visualization and interaction methods that are used for this analysis. We end [Chapter 3](#) with an introduction of loon and discuss how loon is used to perform the visual analysis of the minority data. To that means, we introduce the relevant conceptual aspects of the loon framework.

In [Chapter 4](#) and [Chapter 5](#), we present loon’s framework in detail. [Chapter 6](#) presents some relevant statistical applications that were enhanced with interactive visualization in loon. In [Chapter 7](#), we introduce some novel tools in loon for exploring high-dimensional data with navigation graphs. We conclude this chapter by introducing a novel high-dimensional point glyph called *spiro glyph*. [Chapter 8](#) wraps up this thesis with conclusions and a discussion of future research work.



(a) Navigation graph with canonical graph semantic and bullets representing different dimensionality reduction methods.



(b) Scatterplots driven by the location of the bullet corresponding to a particular dimensionality reduction method.

Figure 1.12: Comparing seven dimensional reduction methods using navigation graphs with the canonical semantic.

# Chapter 2

## RnavGraph

As part of our research, we have developed a software package called `RnavGraph` that provides an interactive environment to explore high-dimensional data using navigation graphs. `RnavGraph` is an open source package for the R statistical environment and hosted on the Comprehensive R Archive Network (CRAN).

`RnavGraph` is a major milestone in our research as it represents a first implementation of the concept of navigation graphs and it demonstrates that, in practice, navigation graphs are useful to explore real data. We designed `RnavGraph` to be flexible so that novel graph semantics can be applied and tested.

The design of `RnavGraph` is an important part of our research. This design includes the selection of essential features for an useful interactive navigation graph environment, the software architecture and the user experience design.

In this chapter, we discuss the functionality of the `RnavGraph` package. We first show how to initialize an `RnavGraph` session for the canonical graph semantic. We then describe the user interactions with the two main displays: the navigation graph display and the 2d scatterplot display. Next, we present part of the software architecture and show how `RnavGraph` can be extended to accommodate a new graph semantic. We end this chapter by listing some limitations of the `RnavGraph` package.

The functionality described here is implemented in version 0.1.6 of the `RnavGraph` package.



Many examples included in this thesis use the olive data first introduced in [Subsection 1.3.1](#). We therefore attach the olive data in R which allows us to refer to its variables by their names (i.e. Area vs. olive\$Area).

```
attach(olive)
```

We also create a second data set called `oliveAcids` that includes only the fatty acid variables, but not the Region and Area variables.

```
oliveAcids <- subset(olive, select=-c(Area, Region))
```


## 2.1 A Default RnavGraph Session

RnavGraph is started from within an R session. The canonical 2d scatterplot example, as described in [Subsection 1.3.1](#), is the default setting and requires the following code:

```
nav <- navGraph(ng_data(name="olive", data=oliveAcids, group=Area))
```

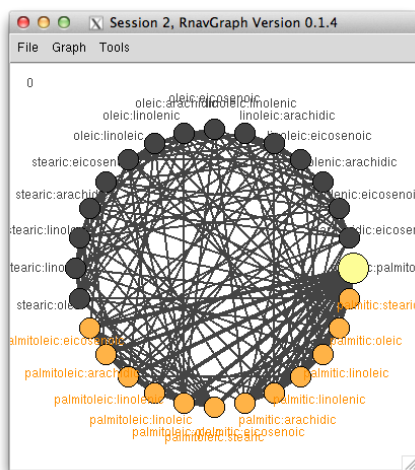
This code produces the navigation graph display as shown in [Figure 2.1\(a\)](#) and the scatterplot display called `tk2d` as shown in [Figure 2.1\(b\)](#). Note that the default group colors assigned to the points in the `tk2d` scatterplot display in [Figure 2.1\(b\)](#) do not correspond to the color key for the olive data defined in [Figure 1.1](#).

The navigation graph display and the scatterplot display in [Figure 2.1](#) form the core components of the RnavGraph package. However, the navigation graph can control any display accessible from R using any user defined graph semantic.

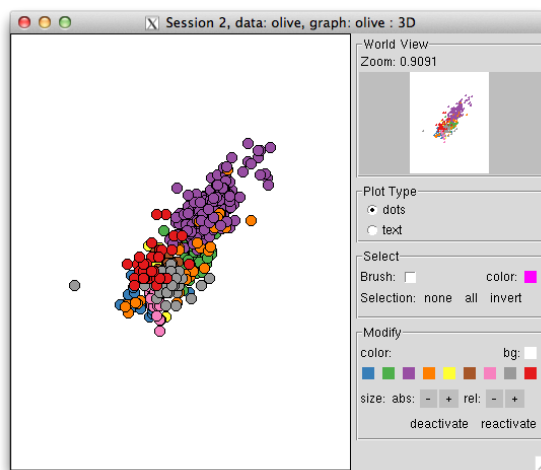
The most important user interactions with the navigation graph in [Figure 2.1\(a\)](#) are simple; the yellow “you are here” bullet  can be dragged along the graph, and controls the 3d rotation or 4d transition shown in the scatterplot display in [Figure 2.1\(b\)](#). The 4d transition graph is accessible via the Graph menu.

## 2.2 The Navigation Graph Display

We now describe the user interactions with the navigation graph display with the help of stylized drawings. These drawings follow the color scheme of the default settings of



(a) interactive 3d transition navigation graph

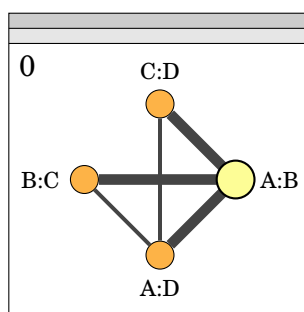



(b) interactive 2d scatterplot tk2d

Figure 2.1: Default RnavGraph session with the default group colors.

RnavGraph. However, the appearance of the navigation graph display can be customized by the user.

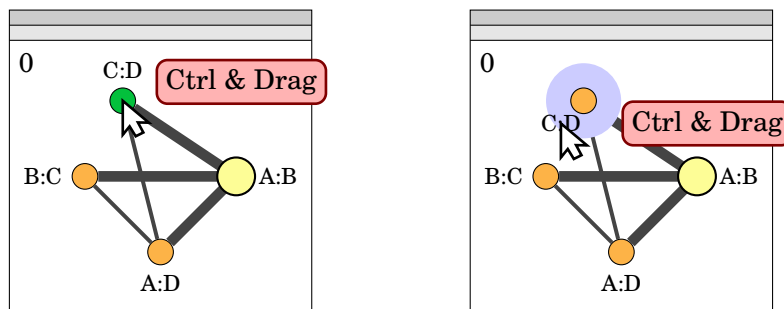
A stylized version of the navigation graph display with a 3d/4d transition graph for the four variates A, B, C, and D looks as follows:



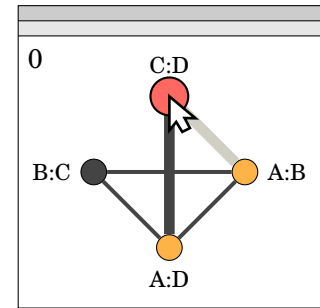
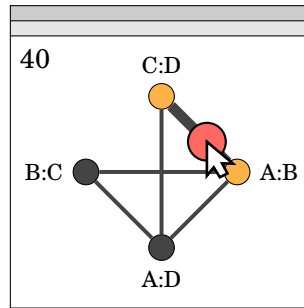
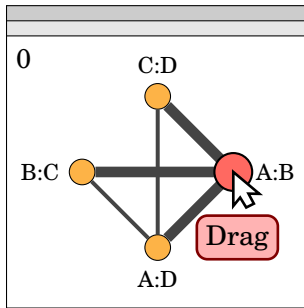
The two main elements in the navigation graph display are the navigation graph in the center and a number between 0 to 99 in the upper right corner. This number shows the percentage progression of the bullet  from the start node to the target node.

We use visual clues to guide the user interactions with the graph. Graph nodes are either orange or dark gray (i.e. ● or ●) depending on whether the node is adjacent to the bullet or not. The graph edges are wide ■ when they connect the bullet to an adjacent node or thin — otherwise. An edge that has been completely traversed by the bullet will change its color from dark to light gray —. Moving the mouse pointer over a node, edge or label will highlight that element green (i.e. ■ and ●), whereas moving the mouse over the bullet will highlight it light red ●.

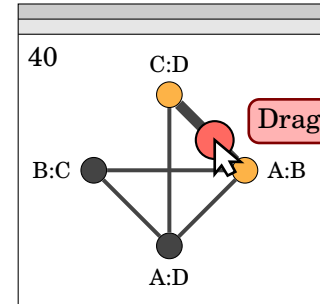
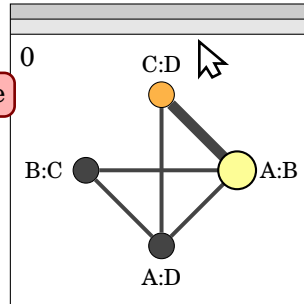
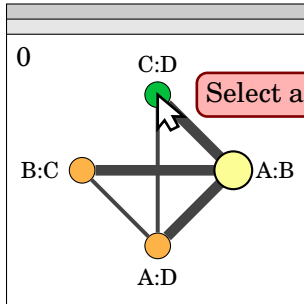
The default initial graph layout arranges the graph nodes on a circle. The user can manually move the nodes and labels on the navigation graph display by dragging the elements while pressing the `CTRL` key. The labels are restricted to lie within a certain radius of the node.



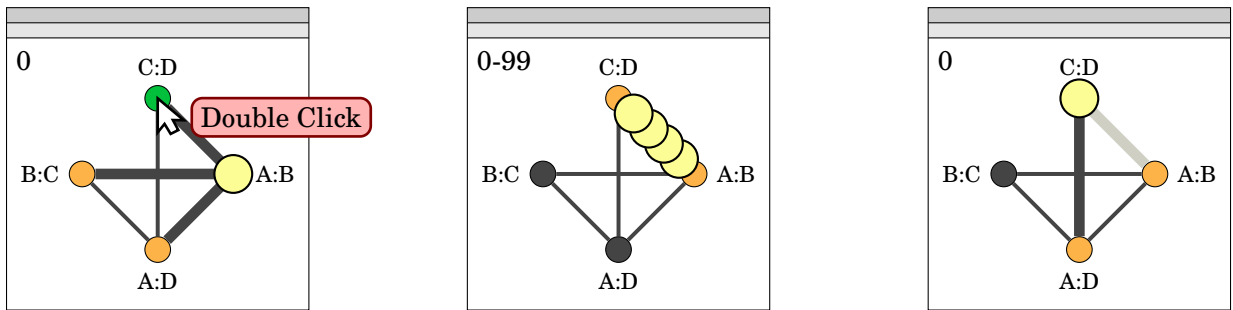
There are several ways to navigate the bullet on a graph. The most intuitive way is to drag the bullet towards an adjacent node. If the bullet is located on a node (i.e. the percentage number is 0) then dragging the bullet will select an edge once it is moved past a small circular decision boundary around the start node. The bullet will then lock onto the adjacent edge that is closest to the suggested dragging direction. The visual clues (as described above and highlighted in the diagram below) will signal that the bullet has locked on an edge and the percentage number will be greater than 0. Once the bullet locks to an edge, it can be either dragged or moved with the mouse scroll wheel.



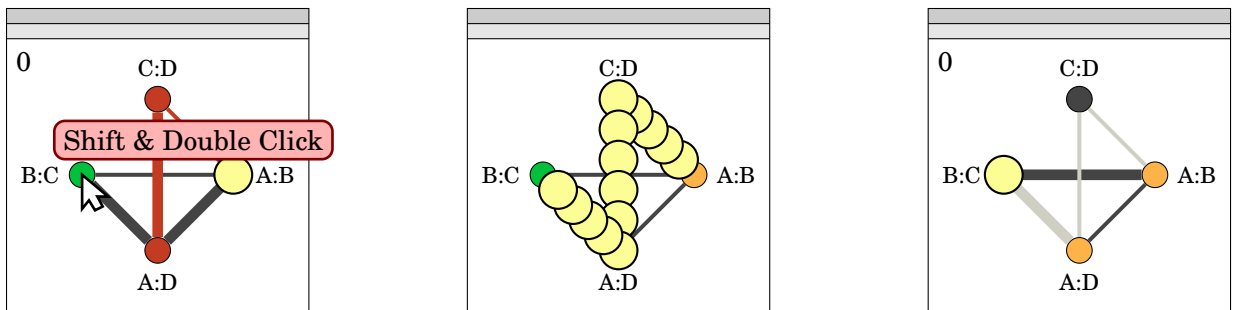
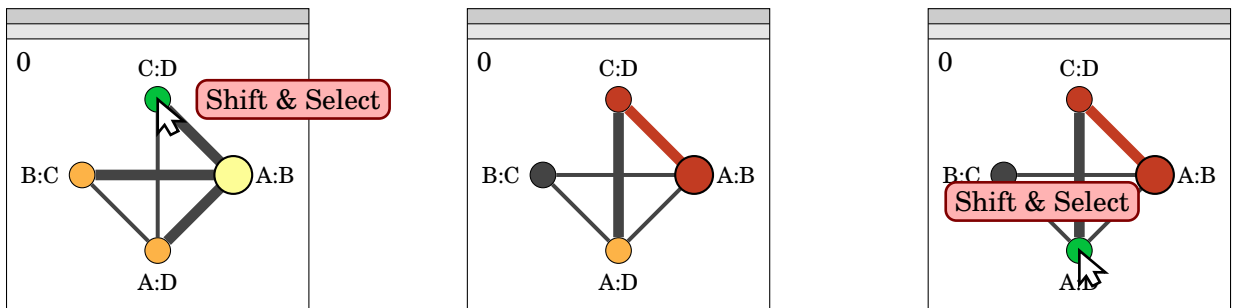
The success of selecting the intended edge for traversal depends on the user's precision in dragging the bullet, but also on the graph layout; that is, if multiple adjacent nodes are located close together it might be hard to drag the bullet in the desired angle away from the node. In this case, one can first select the desired adjacent node to lock the bullet on the corresponding edge. If a non-adjacent node (i.e not orange) is selected at any time, then the bullet will jump onto that node.



A transition can also be automatically traversed or animated by double clicking on an adjacent node. The number of steps (i.e. the rotation resolution) and the animation time can be changed in the navigation graph display settings.



A sequence of edge transitions (i.e. a path) can also be defined and animated (“walked”). Paths can be defined manually or with the `ng_walk` function. A path is defined manually by `Shift` selecting a node sequence. Note that this requires the bullet to lie on a node.



Paths can be saved, re-loaded, commented on, and re-walked with the path tool, as seen in [Figure 2.2](#). The path tool is accessible from the Tools > Paths menu.

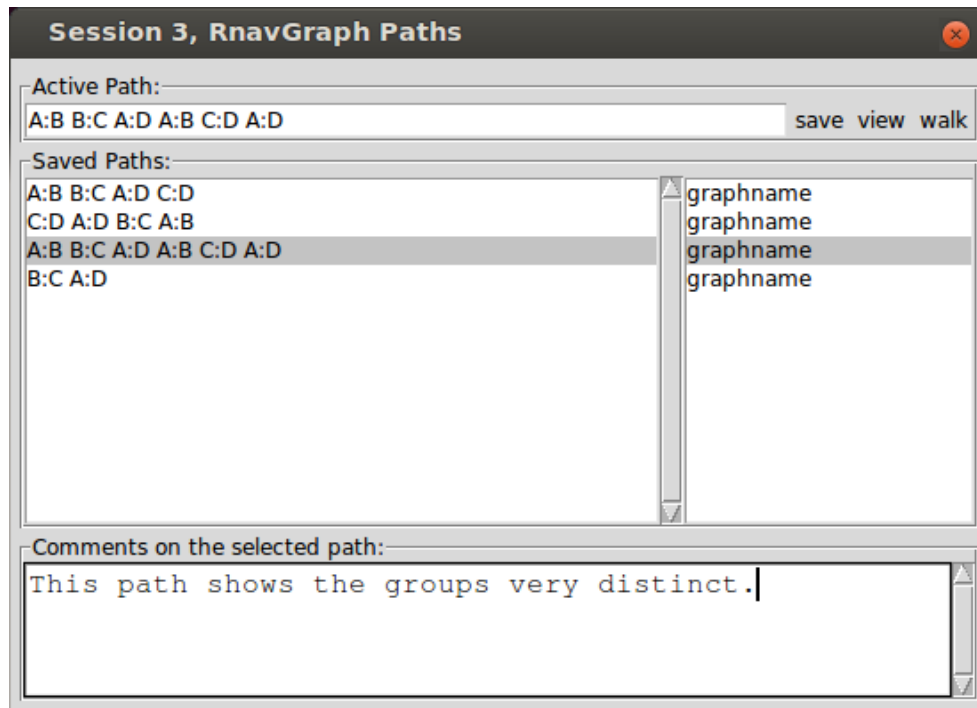


Figure 2.2: Path tool

## 2.3 The tk2d Scatterplot Display

The tk2d scatterplot display, as seen in [Figure 2.1\(b\)](#), is a powerful interactive scatterplot display that is native to RnavGraph.

We developed tk2d because there was no available interactive scatterplot display that we found adequate for testing the navigation graph framework. We initially linked the navigation graph to the scatterplot display of GGobi [69], but later on we decided to implement our own scatterplot display as GGobi has not seen any major development in recent years and we needed more features such as plotting images as point glyphs. We kept GGobi support for versions of RnavGraph up to version 0.1.5, but eventually removed it with the release of R 3.0.0 as at that time rggobi was not available on CRAN for all operating systems.

The main graphic systems in R, that is, `base`, `grid` (Murrell [54]), `lattice` (Sarkar [63]) and `ggplot2` (Wickham [81]), are designed for static graphics and they are not particularly suitable for creating smooth animations in real time. These systems often produce a flicker when they have to re-plot many times in a short time interval (depending on the operating system).

The `tk2d` display is divided vertically into the main scatterplot view to the left and a control panel to the right. All functionality is accessible from the control panel in combination with mouse gestures and the `Ctrl` and `Shift` keys.

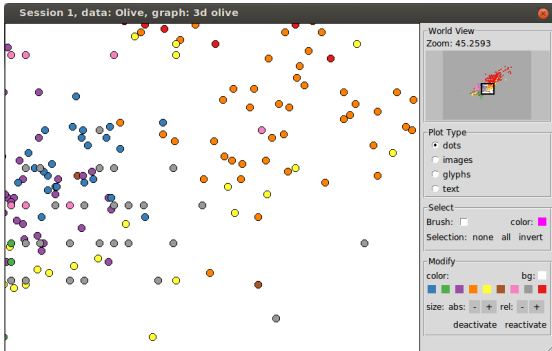
`tk2d` can display points as dots, star glyphs, images and text (see Figure 2.3). Each point has a color, size, active (i.e. visible) and selected state, and different plot types will reflect these states except for the size state for text.

The main view supports zooming and panning. For zooming, the scroll wheel can be used to zoom in and out at the location of the mouse pointer. When zoomed in, one can pan the view by dragging the mouse pointer while the right mouse button is pressed. The viewing area of the main view is highlighted by a white box with a black outline found in the *World View* in the controls panel, as seen in Figure 2.3. The World View always displays all active points, regardless of the zooming level. In addition, the World View allows for zooming with the scroll wheel and panning by dragging the view area.

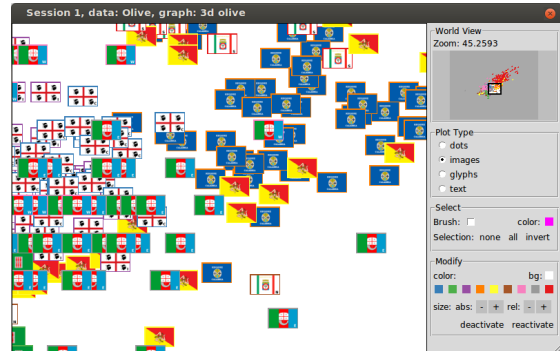
Points can be selected (i.e. highlighted) and then modified with any of the actions found in the Modify section of the control panel. The user can select or de-select a particular point with a left-click on the point or they can select multiple points by using the rectangular brush tool (see Figure 2.4). The brush tool selects points permanently when the `Shit` key is pressed. The brush rectangle can be re-sized by dragging the gray square in the lower right corner of the brush rectangle. A selection can be inverted, all points can be de-selected and all points can be selected.

Modifying a selection will not change the selection state of the points. This way multiple states (i.e. color, size, and active) can be changed for a particular selection.

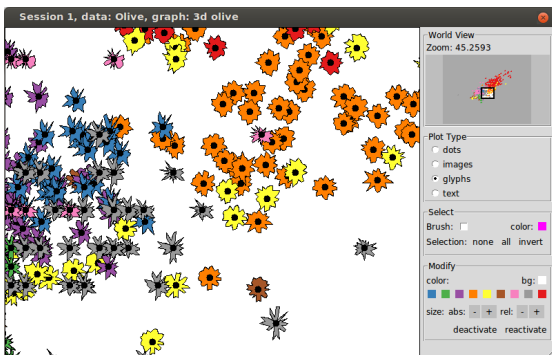
The selected points can be deactivated, that is, their point glyphs will not be visible in the main view and the World View. The *reactivate* button will reset the deactivation state for all points.



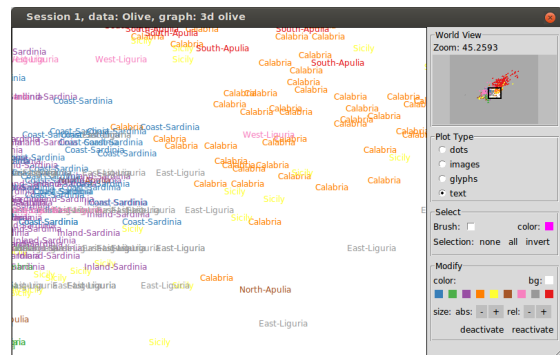
(a) dots



(b) images



(c) star glyphs



(d) text

Figure 2.3: Different plot types supported by the tk2d scatterplot display.

A selection of points can be temporarily moved within the main view by dragging the selection while holding down the `Ctrl` key. This is especially useful when points are over-plotted, as in Figure 2.3(c). Moving points does not change the data and a change in the bullet location will reset this temporary arrangement.

Finally, different tk2d displays that visualize the same data can be linked to each other. That is, all changes in the states of the points (i.e. color, size, selected and active) in one tk2d display will be propagated to the linked tk2d displays.



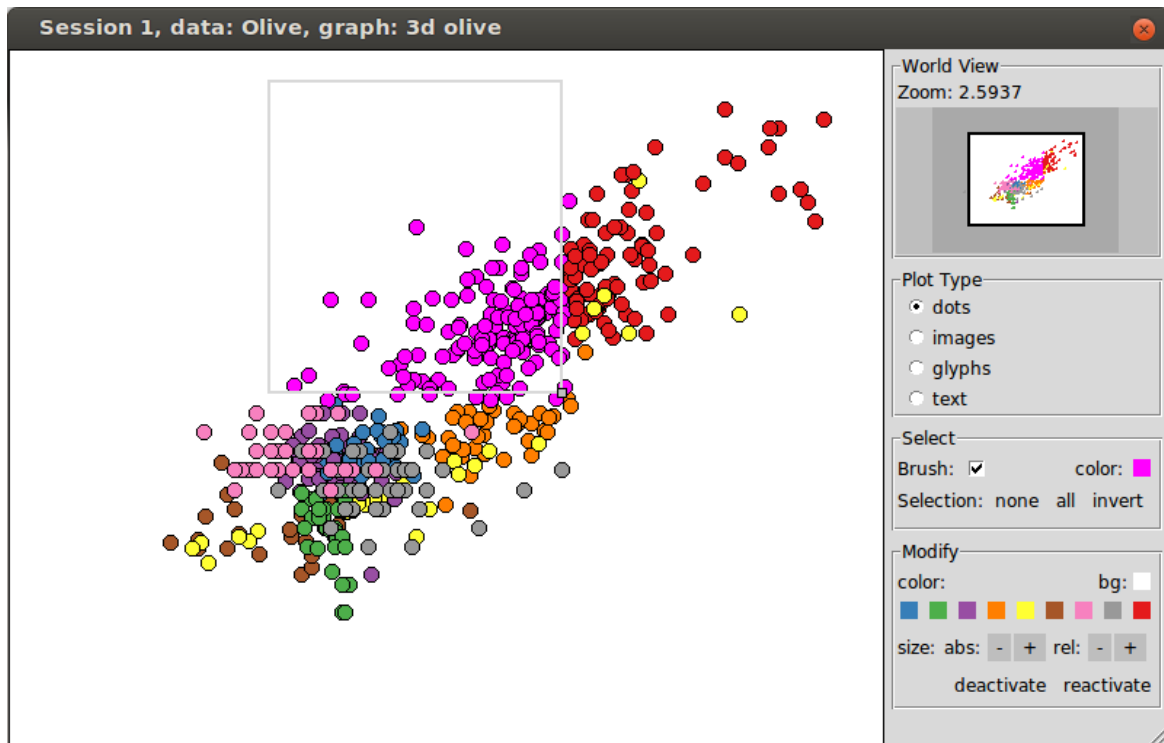


Figure 2.4: Rectangular brush tool.

## 2.4 Software Architecture

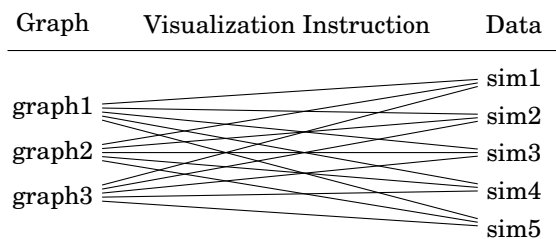
In [Section 2.1](#), we show the default setting as produced by the `navGraph` function. A `RnavGraph` session can also be initialized with much finer control over navigation graphs, graph semantics and data visualizations as described in the following sub-sections.

We first discuss the re-creation of the default `RnavGraph` session as shown in [Section 2.1](#) in more detail. Next, we explain how to create navigation graphs based on scagnostic weights. And finally, we discuss how `RnavGraph` can be extended to accommodate any semantic or visualization.

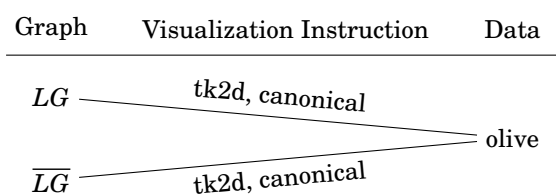
## 2.4.1 navGraph

The `navGraph` function initializes a `RnavGraph` session and takes as its arguments `data`, `graph` and `visualization instruction` objects. These objects are created as follows: the `data` and `graph` objects are instantiated with the `ng_data` and `ng_graph` functions, whereas the `visualization instructions` have different instantiation functions, depending on the navigation graph semantic and visualization.

A `visualization instruction` connects a `data` object to a `graph` object and will result in a `visualization display`. An `RnavGraph` session can link multiple navigation graphs with multiple `data` sets using multiple `views` where a `visualization instruction` corresponds to one `view`. For example, three navigation graphs could each be connected to five different `data` sets with the same variable names (e.g. from simulations) as follows:



For the olive data and the default use setting, the above diagram would look as follows:



The `data` object is created with the `ng_data` function

```
olive.ng <- ng_data(name = "olive",
  data = oliveAcids,
  shortnames = c("p1", "p2", "s", "oleic", "l1", "l2", "a", "e"),
  group = Area,
  labels = Area)
```

Data objects are identified by their names as specified in the `name` argument. The `data` argument requires a `data.frame` object with only numerical variates. If the `shortnames` are defined then they can be used as identifiers for the node labels of the navigation graph. The `group` argument is used to assign different colors to the points at initialization time. The `text` argument specifies the text used with the text point glyph (plot type) in the `tk2d` scatterplot display.

A navigation graph object is created in two steps. First, the graph must be created using the `graph` R library (Gentleman et al. [36]). Second, this graph must be passed to the `ng_graph` function. We wrote the functions `completegraph`, `newgraph` and `linegraph` to simplify the graph creation process with the `graph` R package. Many other graph theoretic algorithms such as the `complement` function and some graph layout and visualization methods are implemented in the `RBGL` and `Rgraphviz` R packages.

We continue our discussion about re-creating the default setting start by creating a complete variable graph  $G$  using the `shortnames` of the data object `olive.ng` for the node labels:

```
G <- completegraph(shortnames(olive.ng))
```

where  $G$  is an object of class `graphAM` that is defined in the `graph` package. Next, the `linegraph`  $L(G)$  is the saturated 3d transition graph:

```
LG <- linegraph(graph=G, sep="++")
```

and its complement  $\overline{L(G)}$  is the saturated 4d transition graph:

```
LGnot <- complement(LG)
```

The `sep` argument separates the node labels of  $G$  in  $L(G)$  and hence must be a unique character string among all node labels. [Figure 2.5](#) shows the visualization of the `LG` graph produced with the `Rgraphviz` package using `plot(LG, "fdp")`.

The `newgraph` function can create a graph from a matrix that defines which graph nodes are connected (i.e. an adjacency matrix) or a two-column matrix with rows defining an edge (i.e. a from-to-edge matrix). Examples that show the use of the `newgraph` function can be found in Section 4 of the `RnavGraph` package vignette. Note that `RnavGraph` only supports undirected graphs.

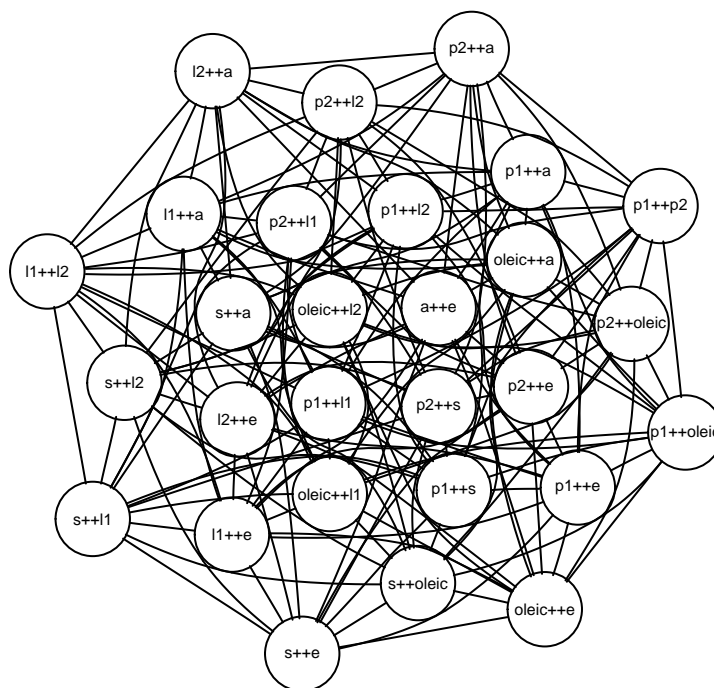


Figure 2.5: Saturated 3d transition graph for olive data.

The transition graphs LG and LGnot are objects defined in the graph package and they need to be passed to the `ng_graph` function in order to be used with the `navGraph` function:

```
LG.ng <- ng_graph(name = "3d Transition",
                  graph = LG, sep = "++", layout = "circle")
LGnot.ng <- ng_graph(name="4d Transition", graph=LGnot, sep="++")
```

As with data objects, the graph objects are uniquely identified by the name argument in a `RnavGraph` session.

Once the graph and data objects are defined they can be linked by using visualization instructions. The `ng_2d` function creates a visualization instruction for the canonical graph semantic in combination with the `tk2d` scatterplot display:

```
vizLG <- ng_2d(data = olive.ng, graph=LG.ng,
               glyphs = eulerian(as(G,"graphNEL")))
vizLGnot <- ng_2d(data=olive.ng, graph=LGnot.ng)
```

The glyph argument is used for tk2d to create star glyphs and must be a vector of the ordered variable names that define the star glyphs. Using the node names of a Eulerian path on the complete variable graph  $G$  ensures that every variable pair exists once in the star glyph, as proposed by Hurley and Oldford [39].

At this stage, a RnavGraph session can be initialized as follows:

```
nav <- navGraph(data = olive.ng,
               graph = list(LG.ng, LGnot.ng),
               viz = list(vizLG, vizLGnot))
```

where this session is shown in Figure 2.6. The navGraph function call returns a navgraph

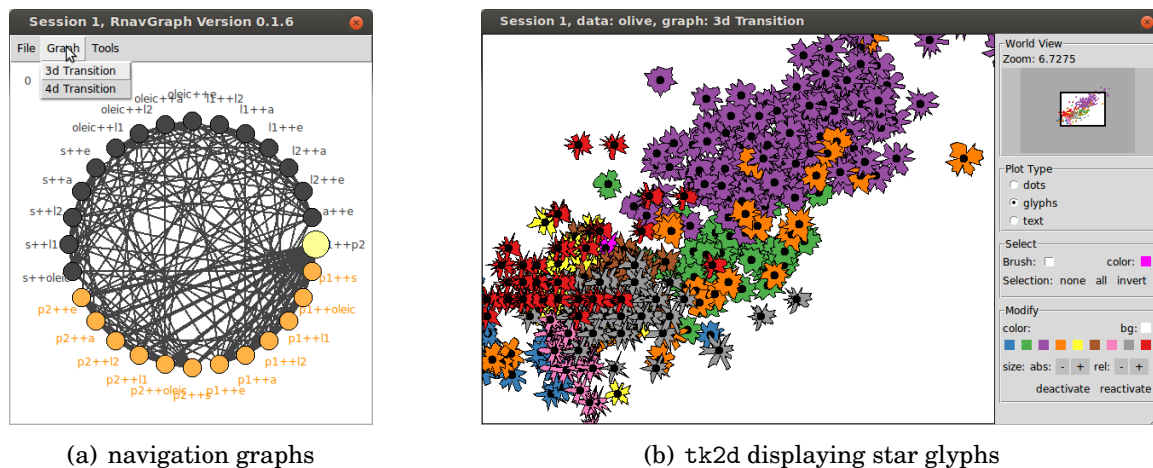


Figure 2.6: RnavGraph session.

handler. In the above example, we assigned this navgraph handler to the nav variable. The navgraph handler can be used to query and modify point sizes and colors, or to restart a RnavGraph session. The point colors can be set according to the color key defined in Figure 1.1 as follows:

```
colScheme <- data.frame(
  level = c("North-Apulia", "Calabria", "South-Apulia",
            "Sicily", "Inland-Sardinia", "Coast-Sardinia",
            "East-Liguria", "West-Liguria", "Umbria"),
  cols = c("#A65628", "#FF7F00", "#E41A1C",
           "#FFFF33", "#984EA3", "#377EB8",
```

```

        "#999999", "#F781BF", "#4DAF4A"),
    stringsAsFactors=FALSE)
oliveCols <- sapply(olive$Area, function(x){
    colScheme$cols[which(x==colScheme$level)]})
ng_set_color(nav) <- oliveCols

```

By default, every tk2d display that displays the data with the name 'olive' (as specified in `ng_data`) will have a linked selection, size, color and deactivation state for its points. To disable linking in tk2d displays one has to start the `navgraph` session with the following `settings` argument:

```

nav1 <- navGraph(data = olive.ng,
    graph = list(LG.ng, LGnot.ng),
    viz = list(vizLG, vizLGnot),
    settings = list(tk2d=list(linked = FALSE)))

```

## 2.4.2 scagGraph

`RnavGraph` comes with functionality that assists in finding small subgraphs based on scagnostics measures as discussed in [Subsection 1.4.1](#). We use the `scagnostics` R package by Wilkinson and Anand [84] for calculating the scagnostic indexes.

The `scagNav` function initializes a `RnavGraph` session with saturated navigation graphs whose nodes satisfy the scagnostic properties specified in the `scagNav` function arguments. A typical `scagNav` call is

```

navScag <- scagNav(data = olive.ng,
    scags = c("Skinny", "Sparse", "NotConvex"),
    topFrac = 0.2,
    combineFn = NULL,
    glyphs = shortnames(olive.ng)[1:8],
    sep = ':')

```

The navigation graph display for this `RnavGraph` session is shown in [Figure 2.7](#). The `scag` argument takes a vector of names of scagnostic measures with an optional prefix “Not”. A “Not” measure is defined as  $1 -$  the original measure. The `topFrac` argument specifies the fraction of nodes with the highest scagnostic measures that should be kept in a navigation graph. The `combineFn` argument specifies whether the scagnostic measures specified in `scag` argument should be used individually for a separate graph construction

or combined through a function. If the `combineFn` is `NULL` then a saturated 3d and 4d transition graph for every measure in `scag` will be created, as in [Figure 2.7](#). Alternatively, the `combineFn` argument can be a function such as `sum`, `max` or `mean`, and, in this case, the `scagNav` session will initialize only one saturated 3d transition graph and one saturated 4d transition graph.

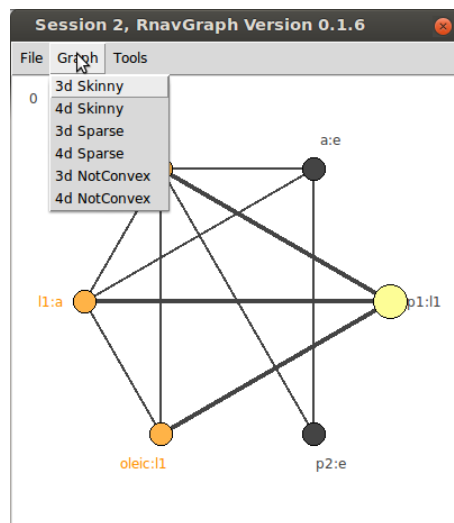


Figure 2.7: `scagNav` session

The `scagNav` function executes a lot of code before calling the `navGraph` function to initialize a `RnavGraph` session with the navigation graphs based on the `scag`nostic measures. The `scagEdgeWeights` and `scagGraph` functions simplify this process, see the vignette [Section 7.2](#) for details.

### 2.4.3 Extending `RnavGraph`

`RnavGraph` can be extended by defining custom visualization instructions. This way, the navigation graphs can drive any display accessible from R and use any user-defined graph semantic.

We provide two ways to create a custom visualization instruction. The simpler method, called “`myplot`”, requires the analyst to specify a function that is called with every bullet

location change. The second method, called “class”, is more involved and is the method that we used for defining the `ng_2d` function for a canonical graph semantic with the `tk2d` scatterplot display. Both methods are described in Section 5.3 of the `RnavGraph` package vignette. Here, we cover only the “myplot” method. Using the “class” method has the advantage of higher computational speed and of getting more information on bullet and graph changes.

The `ng_2d_myplot` function returns a visualization instruction that calls a user-specified function with every bullet location change. For example, one could run a 2d kernel density estimation and plot the scatterplot with contours overlaid:

```
userPlot <- function(x,y) {
  den <- kde2d(x,y,h = c(width.SJ(x), width.SJ(y)))
  plot(x, y, col = "steelblue", pch = 19,
       axes=FALSE, xlab = "", ylab = "")
  box()
  contour(den, add = TRUE, col = 'orange', lwd = 2)
}
```

This user-specified function can use any subset of the arguments listed in [Table 2.1](#). The visualization instructions are then defined as follows:

```
vizMyplot <- ng_2d_myplot(data=olive.ng,
  graph=LG.ng, fnName = "userPlot", device="base")
```

where the function name defined in the `fnName` argument is called with every bullet change. The `device` argument specifies the plotting device so that `RnavGraph` can do the necessary “house keeping” if multiple displays are used. One plot for this session is shown in [Figure 2.8](#).

The `ng_2d_myplot` function name contains “2d” as the  $x$  and  $y$  coordinates that are supplied to the user-defined function are from the canonical semantic using 3d rigid rotations and 4d smooth transitions along a geodesic. However, this relatively simple method can be used for any navigation graph semantic as one can only work with the `from`, `to` and `percentage` arguments.



<b>argument</b>	<b>description</b>
x	<i>x</i> -coordinate
y	<i>y</i> -coordinate
group	group field from the data object
labels	labels field from the data object
order	order of points. In 3d rigid rotation, the order increases with the distance of a the point from the viewer.
from	node name the bullet moves from
to	node name the bullet moves to
percentage	in between percentage of bullet
data	data name of the data object

Table 2.1: Possible arguments for the user-specified function used in combination with `ng_2d_myplot`.

## 2.5 Lessons Learned

RnavGraph works well for all the scenarios described in this chapter. However, RnavGraph has some limitations and the most important ones stem from architectural decisions that we made early in the design stage. As a consequence of these design decisions, it is now difficult and labour intensive to add functionality to the navigation graph display and to the tk2d scatterplot display.

For example, the navigation graphs cannot be changed in an active RnavGraph session. This would be a useful feature in a scagNav session where the navigation graph could update itself by re-calculating the scagnostic measures based on the active points in the visualization display.

Also, tk2d was programmed with the sole intent of being an interactive 2d scatterplot display used with RnavGraph. Consequently, we optimized tk2d for speed and not for general use and extensibility.

Some minor limitations of RnavGraph are due to the programming languages that we used to implement the program. The RnavGraph source is a mix of R, Tcl and C code. We

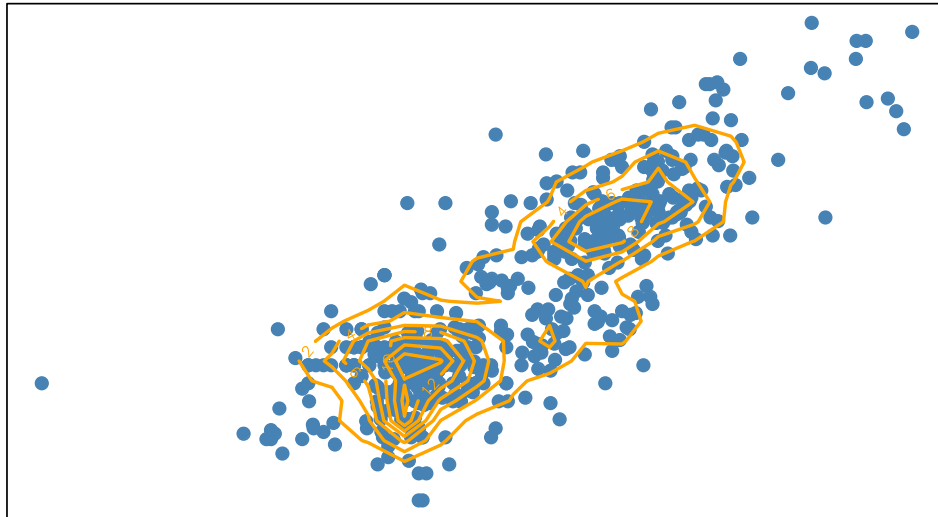


Figure 2.8: User-defined plot for an RnavGraph session.

used Tk as the graphical toolkit. The Tk canvas widget is the essential drawing surface for the navigation graph display and tk2d. The Tk canvas version available at the time of implementing RnavGraph (i.e. Tk version 8.5) did not support alpha blending or anti-aliasing. The alpha blending feature is important in dealing with over-plotting. The lack of anti-aliasing makes RnavGraph appear “pixelated”.

Further, the graphical user interfaces of RnavGraph are not concurrent, that is, only one action can be done at any time. For example, the tk2d display is blocked for further user interactions during a path animation. However, concurrency is technically possible with Tk.

The responsiveness of the tk2d display is good on our computers for up to  $n = 1000$  observations per variate. However, the user interface becomes increasingly sluggish for  $n$  between 1000 and 10000. For Microsoft Windows systems, the Tcl and Tk implementation makes the responsiveness of RnavGraph generally sluggish, even for small  $n$ . Also, displaying images in the tk2d display is dependent on the Img tcl package which is by default available in OSX, but not in Windows. Therefore, on Windows, the user has to install ActiveTcl and link it against R in order to display images with the tk2d display. The Img

tcl package can be easily installed on Linux by using some system package manager.

Another concern is related to the dependencies of the RnavGraph on other R packages. These dependencies are segmented on the CRAN and Bioconductor repositories which makes the installation process not as simple as with other R packages.

## Chapter 3

# Loon By Example

`loon` is a general-purpose interactive data visualization toolkit that we decided to develop after assessing the benefits and shortcomings of the `tk2d` scatterplot display in `RnavGraph`. That is, `tk2d` has a collection of features that we find essential in working with high-dimensional data that were not all available in other interactive visualization software integratable into a complete statistical computing environment such as R. These features include interactive zooming, panning and selection, various point glyphs, linking, and the control panel with the worldview. However, the usefulness of the `tk2d` scatterplot display is limited to navigation graph sessions in `RnavGraph` as `tk2d` is tightly coupled with the rest of `RnavGraph`. Hence, creating independent `tk2d` scatterplots is not possible.

As a consequence, we took up the development of `loon`. `loon` implements a general design for various interactive statistical graphic displays that supports layering of visual information such as point objects, lines and polygons. These displays further support zooming, panning and selection, and modification and deactivation of plot elements and layers. Interactions with plots are provided with mouse and keyboard gestures as well as via command line control and with inspectors. These inspectors provide graphical user interfaces for modifying and overseeing the plots. `loon` also implements a novel dynamic linking mechanism that can be used to assign the plots that are to be linked and the linking rules at run time. Additionally, `loon`'s design provides several different types of event bindings to add and customize functionality of `loon`'s displays.

loon is written in Tcl and Tk. We created the R package loon to show how loon can be embedded into the R environment for statistical computing. The functionality discussed in this thesis works with loon version 1.0.0.

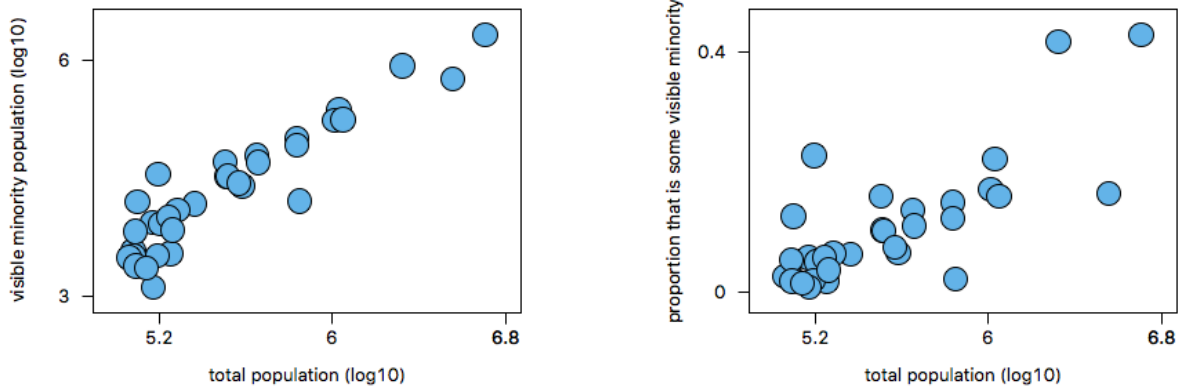
As a parallel to the diving behavior of the software’s namesake bird, “loon”, we structure this chapter and the two subsequent ones by first providing a “dip” into loon’s features, then a more detailed “dive”, and finally a “deep dive” that discusses the advanced framework and implementation of loon.

We start our discussion with a visual exploration of the Canadian visible minorities data [66]. While discussing the visual analysis, we emphasize useful visualization features and tools that can help discover interesting phenomena or “stories” in the data. Then, we introduce loon by demonstrating how it implements the desired features and tools identified during the visual analysis.

### 3.1 An Exploratory Data Analysis

We consider an exploratory visual analysis of visible minority populations distributed across major census metropolitan areas of Canada. These data are from the 2006 Canadian census, publicly available from Statistics Canada [66]. For each of the 33 Canadian census metropolitan areas, we have the total population and the population of all its “visible minorities”. These self-declared visible minorities are: “Arab”, “Black”, “Chinese”, “Filipino”, “Japanese”, “Korean”, “Latin American”, “Multiple visible minority”, “South Asian”, “Southeast Asian”, “Visible minority (not included elsewhere)”, and “West Asian”. For each metropolitan area, we also obtained the approximate latitude and longitude coordinates using the Google Maps Geocoding API and added them to the data set.

Figure 3.1(a) shows the population of all visible minorities against the total population for each metropolitan area, where both numbers are on a common logarithmic scale. For example, Canada’s largest population centers, Vancouver, Montreal and Toronto, appear on the top right of the plot in that particular order from left to right. Not surprisingly, the visible minority population increases with the total population with an approximately linear trend on this scale.



(a) Visible minority ( $y = \log_{10}(m)$ ) versus total population ( $x = \log_{10}(t)$ )

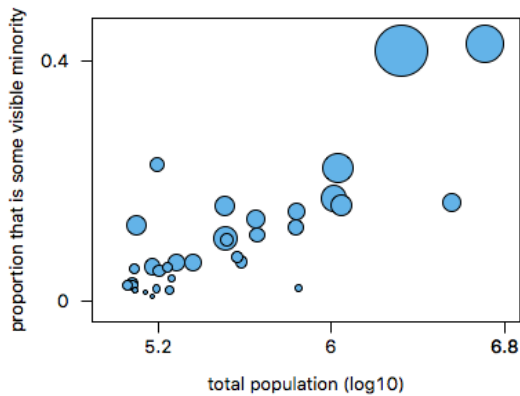
(b) Visible minority proportion ( $y = m/t$ ) versus total population ( $x = \log_{10}(t)$ )

Figure 3.1: Visible minority population versus total population for each of the 33 Canadian metropolitan census areas. Here  $t$  = total population count,  $m$  = total visible minority count.

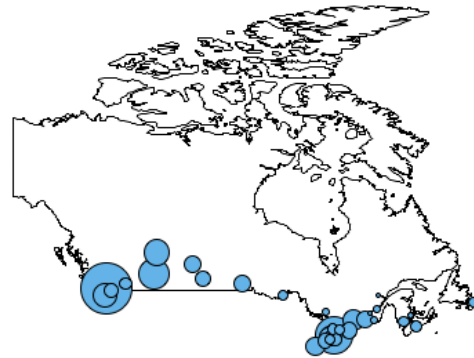
It is more interesting to notice the relationship between the proportion of visible minorities and the total (log) population as shown in Figure 3.1(b). We can see that the visible minority populations not only grow with the general population, but also that they constitute a larger proportion when populations are larger. In Vancouver and Toronto (the two top right points), the proportion of the population that is from a visible minority is higher than 40%. Montreal, on the other hand, is different in that it has a relatively low proportion of visible minorities for its population. All three of these large population areas deviate from the roughly linear relationship suggested by the remaining points of Figure 3.1(b).

Information on a particular visible minority can be encoded on these plots using the area of the circle. In Figure 3.2(a), we show the same plot as in Figure 3.1(b), but this time we use the proportion of the total population that is Chinese for each census area to determine the area of each circle.

Figure 3.2(b) locates the points on a map of Canada. The relatively large Chinese populations of Vancouver and Toronto areas are easily located on the map. The map also draws attention to two geographically close areas in Alberta that have relatively large



(a) Visible minority proportion ( $y = m/t$ ) versus total population ( $x = \log_{10}(t)$ ); circle areas are proportional to  $c/t$



(b) Census area map locations; circle areas are proportional to  $c/t$

Figure 3.2: Visible minority population versus total population for each of the 33 Canadian metropolitan census areas – range from 0.0015 for Trois-Rivières to 0.18 for Vancouver. Here  $t$  = total population count,  $m$  = total visible minority count and  $c$  = total Chinese minority count.

Chinese populations as a proportion of visible minorities (i.e. Calgary and Edmonton).

A natural question is how the composition of visible minorities varies from one area to another within Canada. In particular, it would be interesting to know whether any such variation is correlated with the geographic region. To help answer these questions, one can plot the relative proportion of each visible minority against the total minorities population on radial axes, as shown in Figure 3.3(a) and Figure 3.3(b) for Toronto and Montreal. The resulting shapes, also known as star glyphs, provide a visual representation of the minority composition for each metropolitan area.

In particular, in Figure 3.3, the values on each axis are determined as the ratio of a particular visible minority population to the total minorities population of that area. The axes are ordered so that we alternate long and short axes values (over all areas).

We note that, in Figure 3.3, the two star glyphs are quite different. Toronto’s largest self-declared visible minority is “South Asian” whereas Montreal’s is “Black”. Unlike Toronto, a significant proportion of Montreal’s visible minority population is “Arab”, “Latin

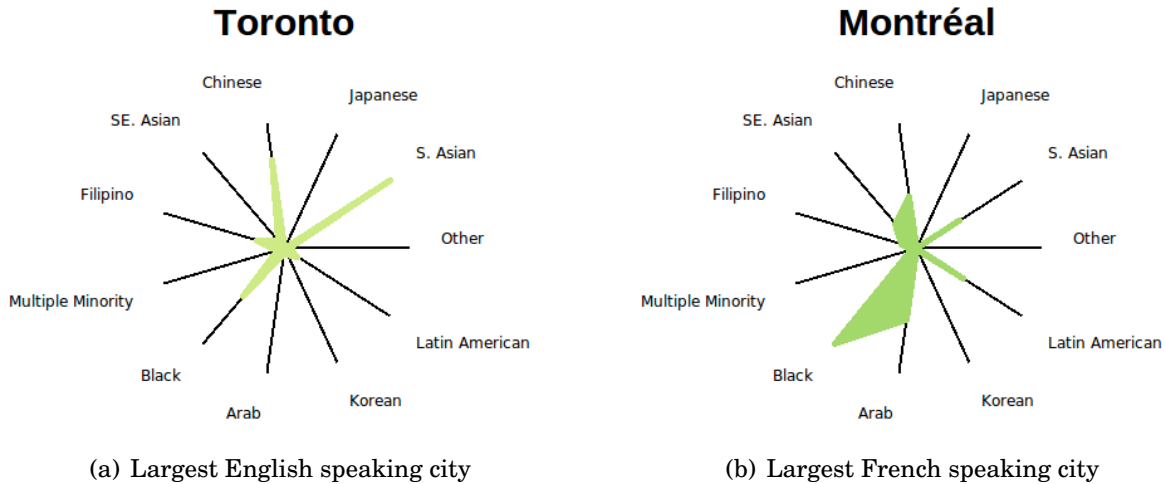


Figure 3.3: Radial axis plots for the two largest Canadian census areas.

American” and “Southeast Asian”. If visible minority populations in these major Canadian cities are largely composed by recent immigration then the difference in visible minority compositions between these two cities might be at least partially explained by linguistic preferences and sources of immigrants.

Figure 3.4 shows the star glyphs for all 33 census areas. As we can see, there is a considerable variation in the shape of the star glyphs. This indicates a variation of visible minority composition across Canadian metropolitan areas. There are also some similarities between shapes suggesting that certain metropolitan areas have a similar composition.

The colors in Figure 3.4 represent different geographic regions as shown in Figure 3.5(a). From west to east the colors are assigned using a red-yellow-green gradient as given by the RColorBrewer R library [56].

In Figure 3.5, we focus on one geographical region that includes Southwestern Canada and the Prairies. The coastal cities of Vancouver and Victoria have a very similar visible minority composition, as do the prairie cities of Edmonton, Calgary, Saskatoon and Regina. In the coastal region, Abbotsford-Mission (the first element from the left on the third row from the bottom in Figure 3.4(a)) is notably different as its visible minorities pop-



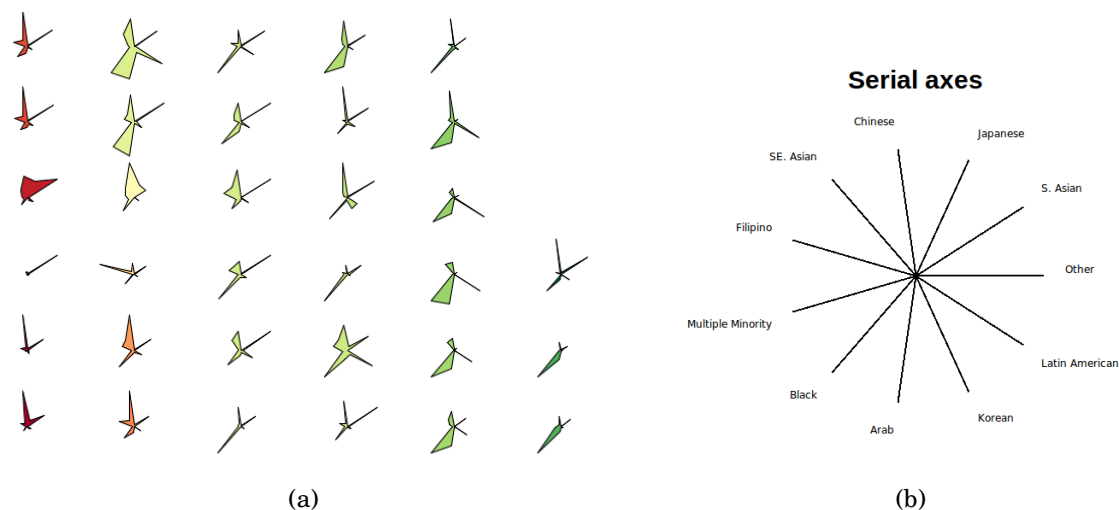
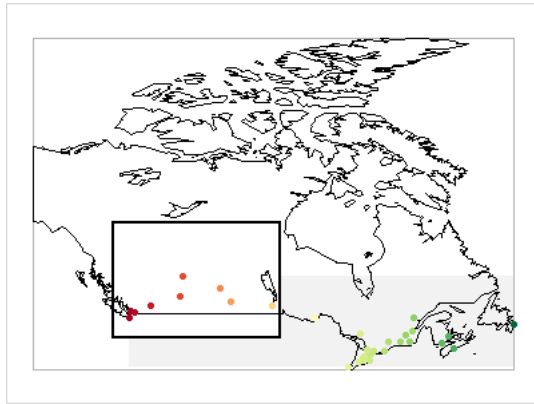


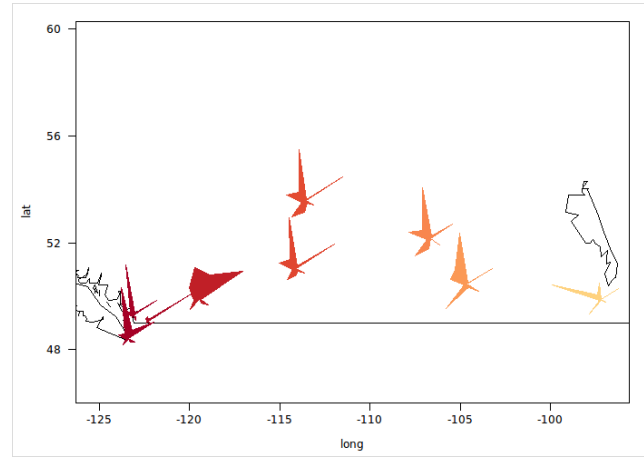
Figure 3.4: Radial axis plots, or star glyphs, for all 33 metropolitan census areas. Colors are assigned based on a red-yellow-green gradient from west to east. The radial axis order is shown at the right.

ulation is dominated by “South Asians”. Similarly, a little east of Abbotsford-Mission, is the metropolitan area of Kelowna (the first element from the left on the fourth row from the bottom in [Figure 3.4\(a\)](#)). There, an “unusually” high proportion of “Japanese” and “South-east Asian” appears in comparison to Kelowna’s neighbours. Finally, at the east end of the focus region ([Figure 3.5\(b\)](#)) lies Winnipeg which looks much like its prairie neighbours with the exception of having “Filipino” as its largest visible minority (the second element from the left on the third row from the bottom in [Figure 3.4\(a\)](#)).

Even this preliminary analysis indicates that there are similarities and interesting differences in the composition of visible minority populations within and between geographic regions in Canada. Like any good exploratory analysis, it reveals unanticipated patterns and raises new questions for further analysis and explanation. For example, why are the three major cities, Vancouver, Toronto and Montreal, so different in their composition of visible minorities? Why is there a relatively large “Filipino” population in Winnipeg, “South Asian” in Abbotsford-Mission, or “Japanese” in Kelowna?



(a) Worldview

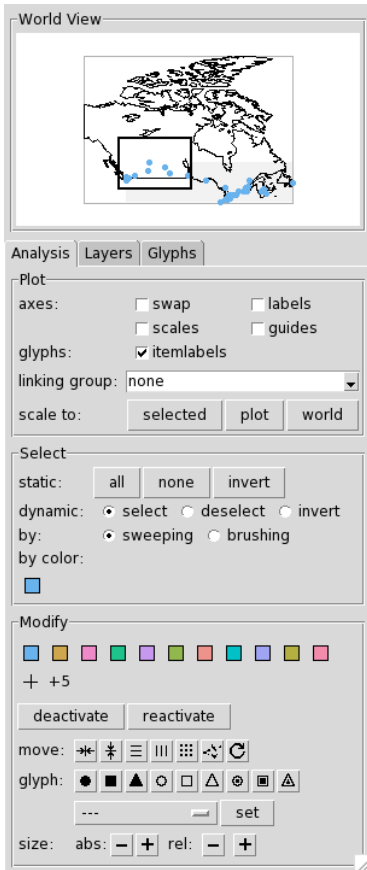


(b) Zooming in on southwestern Canada

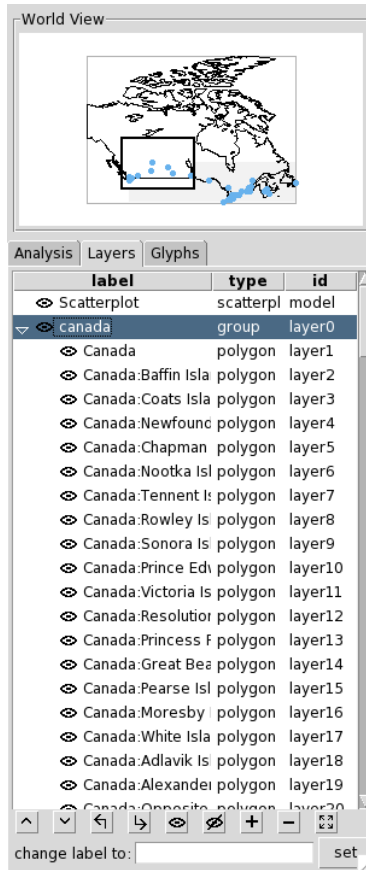
Figure 3.5: Zooming in on a region. (a) A worldview plot. (b) The area of focus – Southwestern Canada and the Prairies. The region shown in (b) is highlighted with a white rectangle in (a).

To support such exploration, software needs to be carefully designed. It needs to be a combination of interactive display-oriented methods that rely on direct manipulation through natural gestures and control panels, and of open-ended command line or programmatic interfaces that allow precise and powerful manipulation of the display.

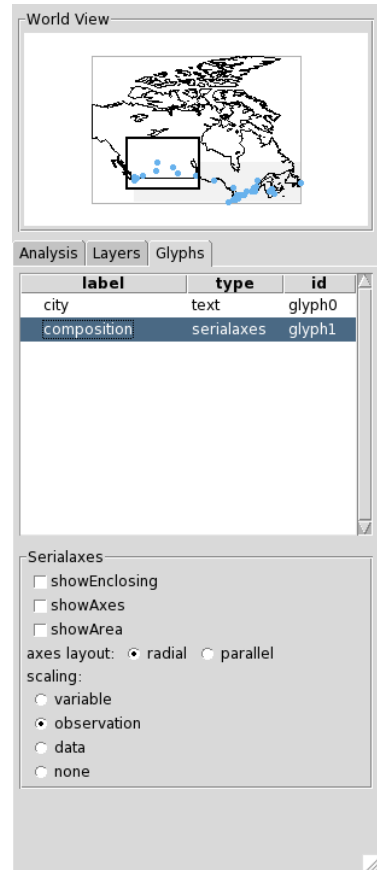
The above preliminary analysis was conducted with `loon` and made use of mouse gestures such as selection, panning and zooming, along with powerful inspectors as shown in [Figure 3.6](#). Also, of key importance was the use of the command line in R to create and modify plots using numerical, statistical and geospatial functionality available in R.



(a) analysis plot inspector



(b) layer inspector



(c) glyph inspector

Figure 3.6: The default loon inspector is context specific for the active loon plot. For a scatterplot display it shows a worldview, an analysis, layers and glyphs inspector.

## 3.2 Performing the Exploratory Analysis with the loon R package

In this section, we illustrate how the above exploratory data analysis of the minority data was performed using loon in R. Each subsection shows a main analysis step and also discusses the major loon design concept behind this step.

The geocoded minority data is part of the loon R package. The loon R package is loaded as follows

```
library(loon)
names(minority)

[1] "Arab"                "Black"
[3] "Chinese"             "Filipino"
[5] "Japanese"           "Korean"
[7] "Latin.American"     "Multiple.visible.minority"
[9] "South.Asian"        "Southeast.Asian"
[11] "Total.population"
[12] "Visible.minority.not.included.elsewhere"
[13] "Visible.minority.population"    "West.Asian"
[15] "lat"                 "long"
[17] "googleLat"           "googleLong"
```

### 3.2.1 Plot States

The R code used to create the scatterplot in [Figure 3.1\(a\)](#) with loon is

```
p <- l_plot(x = log10(minority$Total.population),
  y = log10(minority$Visible.minority.population),
  xlabel = 'total population (log10)',
  ylabel = 'visible minority population (log10)',
  showScales = TRUE,
  size = 20,
  glyph = 'ccircle')
```

The return value of `l_plot`, here assigned to `p`, is a plot handle to access and modify the scatterplot via the command line. For example, `p['color']` returns a vector with the

hexadecimal encoded color representation of each of the 33 points, and `p['size'] <- 5` sets the size of every point to 5. The quantifiers `size`, `color`, `x`, `y`, `xlabel` are called plot *states* and loon's scatterplot has over 30 states. Note that the replacement function '`[<-`' for the plot handle `p` modifies one plot state per evaluation, whereas multiple states can be modified at once with the `l_configure` function. The `l_cget` function can be used instead of the extractor function '`[`'.

The number of points in a loon scatterplot is referred to as the abstract dimension  $n$ . Some of the states of the scatterplot with dimension  $n$  are `x`, `y`, `color` and `size`. The actual value of  $n$  is set when the plot is created and it can be changed by modifying the dimension of the `x` and `y` states. Therefore, `x` and `y` are called the dominant  $n$ -dimensional states. All the other  $n$ -dimensional states are called non-dominant and take on their default values whenever  $n$  is changed. Also, when assigning a single value to an  $n$ -dimensional state, e.g. `p['size'] <- 5`, the corresponding value gets repeated  $n$  times.

Two other important  $n$ -dimensional scatterplot states are the boolean states `selected` and `active`. Active points are displayed on the scatterplot whereas inactive ones are hidden (i.e. not rendered). Being able to deactivate points allows one to focus on a certain subset of the data points. Selected points are highlighted magenta. For the selected points, attributes such as `color`, `size` and `active` states can be easily modified with the analysis inspector as shown in [Figure 3.6\(a\)](#).

### 3.2.2 Graphical User Interface

Once a scatterplot is created we can use a series of interaction gestures to further interact and explore the plot. This includes zooming towards the mouse cursor using the mouse wheel, panning by right-click dragging and various selection methods using the left mouse button such as sweeping, brushing and individual point selection. Each of these gestures modifies one or multiple plot states; panning modifies `panX` and `panY`, zooming towards the mouse cursor modifies `zoomX`, `zoomY`, `panX` and `panY`, and selecting points modifies the `selected` state. Hence, any view that can be attained by using mouse gestures can also be created – or stored and recreated – using the command line interface.

A sweep selection selects all points below a rectangular area which is defined by a left-click drag gesture. That is, the upper left corner of the sweep rectangle is at the location of the left-button press, and the lower right corner is at the current cursor location while the left button is pressed. A brush selection also selects all points below a rectangular area. However, in contrast to the sweep selection, the rectangular brush area has a fixed size and a left-button press moves the lower right corner to the current mouse location, and a left-click dragging gesture moves the brush area along the mouse pointer. To change between sweeping and brushing mode one can set the `selectBy` state to either 'sweeping' or to 'brushing'.

By default, every loon plot is accompanied by a loon inspector as seen in [Figure 3.6](#). The loon inspector for scatterplots is a composite of worldview, analysis, layer and glyph inspector. Each inspector provides a visual interface to overview and modify plot states and elements such as glyphs and layers.

The scatterplot worldview inspector shows all visual elements and the view region of the scatterplot while preserving the aspect ratio. It also supports zooming and panning with mouse gestures while updating the scatterplot accordingly.

The analysis inspector is an arrangement of graphical user interface widgets such as buttons, check boxes, radio buttons and pull-down menus that provide control over plot states. Each of these widgets performs an action that modifies one or more plot states. Additionally, some widgets display the current value of a plot state. For example, the pull-down menu for changing the linking group also displays the current linking group which can be modified either via the pull-down menu itself or in the text area of the pull-down menu. The actions on the analysis inspector are grouped into *Plot*, *Select* and *Modify* actions. The Plot actions modify various 1-dimensional plot states, whereas the Select actions control either the selected state or the interactive selection mode such as sweeping and brushing. The Modify actions change the values of their corresponding  $n$ -dimensional state for all selected points (except the “reactivate” button that sets all points to active).

The layout and presentation of the analysis inspectors are designed to be intuitive, self-explanatory and to support a broad range of statistical analyses. We describe in detail all loon inspectors in [Subsection 4.4.4](#).

### 3.2.3 Linking

All of loon's plots can be linked so that some of their  $n$ -dimensional states are synchronized. For a linked scatterplot, the states that are synchronized by default (i.e. *linked states*) are the color, size, selected and active states. Two or more plots are linked if they share the same string in their linkingGroup state. The linkingGroup state can be set via the analysis inspector, as seen in [Figure 3.6\(a\)](#), or by using the plot handle as follows

```
l_configure(p, linkingGroup='minority', sync='pull')
```

The sync argument is only necessary if there are some plots with the particular linking group 'minority'. In that case, loon needs to know whether, for the initial synchronization, the “newcomer” plot, here p, should adapt to the linked states of the other linked plots (i.e. 'pull'), or whether it should overwrite the states of the other linked plot with its own linked states (i.e. 'push'). If the linking group is set via the analysis inspector a dialog will pop up asking whether to push or pull the linked states. Once the plots are linked, every change of a linked state in one of the linked plots is pushed to the other plots.

### 3.2.4 Layers

The plot in [Figure 3.2\(b\)](#) is a scatterplot of the geographic coordinates of Canadian cities with the map of Canada layered as polygons underneath the scatterplot points. loon's scatterplot supports layering of lines, points, text, rectangles, polygons, ovals and groups. The layers are arranged in a tree structure, and a group layer can be the parent of other layers. The code to create the plot in [Figure 3.2\(b\)](#) uses the maps in the R package maps

```
p_map <- with(minority,
  l_plot(x=long, y=lat,
        size=30*Chinese/Visible.minority.population,
        showScales=FALSE, showLabels=FALSE)
)
library(maps)
canada <- map("world", "Canada", plot=FALSE, fill=TRUE)
l_layer.map(p_map, canada)
```

When plotting maps it is important to have control over the aspect ratio to deal with map distortions. In loon, the aspect ratio of the plot p\_map can be queried with

`l_aspect(p_map)` and set with `l_aspect(p_map) <- 1`. However, the aspect ratio of a loon plot changes with the ratio of the plot size, the ratio of the zoomX and zoomY or with the ratio of deltaX and deltaY plot states.

### 3.2.5 Star Glyphs

The star glyphs in [Figure 3.5\(b\)](#) were added to the scatterplot in [Figure 3.2\(b\)](#), as point glyphs using the code



```
g1 <- l_glyph_add_serialaxes(p_map,
  data = minority[, c(12,9,5,3,10,4,8,2,1,6,7)],
  axesLayout = 'radial',
  showArea = TRUE,
  scaling = 'observation',
  label = 'minority data')

p_map['glyph'] <- g1
```

The glyph state of the scatterplot `p_map` is  $n$ -dimensional, therefore it is possible to display every point with a different glyph. The serialaxes glyphs have their own states to control the glyph appearance. As with plot states, glyph states can be accessed and modified using the glyph handle `g1`. For example, `g1['axesLayout']` returns the `axesLayout` state and `g1['axesLayout'] <- 'parallel'` sets the `axesLayout` state to `'parallel'`. Alternatively, the `l_cget` and `l_configure` R functions can be used to query and modify plot states. For example,

```
l_configure(target=g1, axesLayout='radial')
```

The `axesLayout` state controls whether the point glyph `g1` gets displayed as a star glyph (`'radial'`) or as a parallel coordinate glyph (`'parallel'`).

To arrange the star glyphs in a scatterplot on a grid, as seen in [Figure 3.3\(a\)](#), one can either use the function `l_move_grid` or select all points and press the button  in the analysis inspector. Either way will assign the scatterplot temporary coordinates in the `xTemp` and `yTemp` plot states. To reset the temporary location to the initial location set in the states `x` and `y` one can either use the reset button  in the inspector or use the `l_move_reset` function.



### 3.3 Conclusions

In this chapter, we conducted an exploratory visual analysis and described desirable functionality of visualization software that would convey and expose the information in the data in an efficient manner. Next, we introduced loon and discussed how loon's design concepts implement this desirable functionality. These concepts include plot states, inspectors, linking, layers and point glyphs. The next chapter will re-visit each of these design concepts in depth and will also introduce the other plot displays available in loon such as histogram and the graph display. The more advanced design concepts such as event bindings and geometry management are discussed in [Chapter 5](#).

# Chapter 4

## Loon Framework

Our main goal in developing loon was to create a useful interactive data visualization framework that was flexible, powerful and yet simple (i.e. with a gentle learning curve) and intuitive for a broad audience. It took us several iterations to arrive at a design and syntax that balanced generality with simplicity, was extensible and integrable in R. Many software systems have influenced the design of loon, most notable are Tk's dynamic application programming interface and approach for building graphical user interfaces, R's base graphics system, Adobe's Creative Suite and Apple's Keynote with its inspector. Also, many of loon's statistical interaction features have been seen in some form in other interactive statistical visualization systems including PRIM [27], Quail [43], Lisp-stat [76], Plot Windows [67], DINDE [58], DataDesk [80], Data Viewer [40], the gobi family [68, 13, 69, 47], iplots [79] and Mondrian [75]. Among those interactive data visualization systems we see loon as being the first one that has a comparatively rich set of interaction features in addition to being integrated in a comprehensive and widely used and available statistical computing environment such as R and, further, loon also provides an extensive application programming interface that can be used to control and extend plots and to build new visualization tools.

Some of the key design features of loon include:

- it uses inspectors (i.e. control panels) to interact with displays
- loon's displays are widgets much like buttons and sliders that can be re-used and customized to dynamically create a new graphical user interface; therefore, loon is a toolkit for building graphical user interfaces
- loon's standard linking model uses a three-phase logic that is simple and fairly general, but other types of linking can also be implemented
- it supports layering information on displays such as polygons, model fits and maps
- loon's scatterplot display has interchangeable point glyph types including images, star glyphs, polygons and text
- it supports an extensive set of event bindings. Event bindings hook user-defined code to specific event types such as state change, mouse and keyboard events
- loon's software architecture is object-oriented and one can add new displays with relatively little effort
- as loon is written in pure Tcl and Tk it can be embedded in other languages such as R, Python, Perl and Ruby

We chose to use Tcl and Tk for implementing loon for several reasons. Tcl is an excellent choice for rapid prototyping; while exploring possible features and their implementation we have refactored multiple parts of the software several times. Both Tcl and Tk are mature technologies and future versions of Tcl and Tk will likely be backwards compatible and substantive issues with the framework have long been addressed by the Tcl and Tk community. These are the advantages of working with late wave technologies as discussed in [50] (page 66). Another important reason for choosing Tcl and Tk is that its binding with R (via the `tcltk` R package) is part of the base R distribution. This meant that we did not have to spend time working on the R and Tcl interface and also that the installation of loon in R is particularly simple (see our discussion “Portability, easy install, web compatible” on [page 192](#)). Important to this binding are so called “callback functions”

that allow R functions to be called from Tcl (see [Section 5.2.1](#)) and also the automatic conversion of many data structures and types between the two programming environments.

In this chapter, we discuss the full scope of loon's features introduced in the previous chapter. These features include plot states, graphical user interface interactions, linking, layers and inspectors. Additionally, we introduce all of loon's displays including the histogram, scatterplot, serialaxes plot and graph display. The graph display can be used to navigate high-dimensional data as with RnavGraph.

The next section provides an R sample session that introduces each display. In [Section 4.2](#), we describe the plot layout and the mapping between data coordinates and screen coordinates. [Sections 4.3 to 4.6](#) re-visit the design concepts introduced in [Section 3.2](#) (i.e. plot states, graphical user interface, linking and layers). [Section 4.7](#) ends this chapter with a discussion of the particularities of each display.

## 4.1 Introduction to the Displays

### 4.1.1 Scatterplot

We first introduced loon's scatterplot display in [Section 3.2](#). [Figure 4.1](#) shows another example of loon's scatterplot using the olive data that was created with the following code

```
p <- l_plot(x=stearic, y=oleic, color=Area, title='scatterplot')
```

The return value of the `l_plot` function, here assigned to the variable `p`, is a plot handle to access and modify the scatterplot states. As the elements of `Area` are not valid color names, loon maps the `Area` values to colors as explained in detail in [Subsection 5.6.3](#).

### 4.1.2 Histogram

[Figure 4.2\(a\)](#) shows a histogram of the `oleic` variable from the olive data and was created as follows

```
h <- l_hist(x=oleic, color=Area, showScales=TRUE, title='histogram')
```

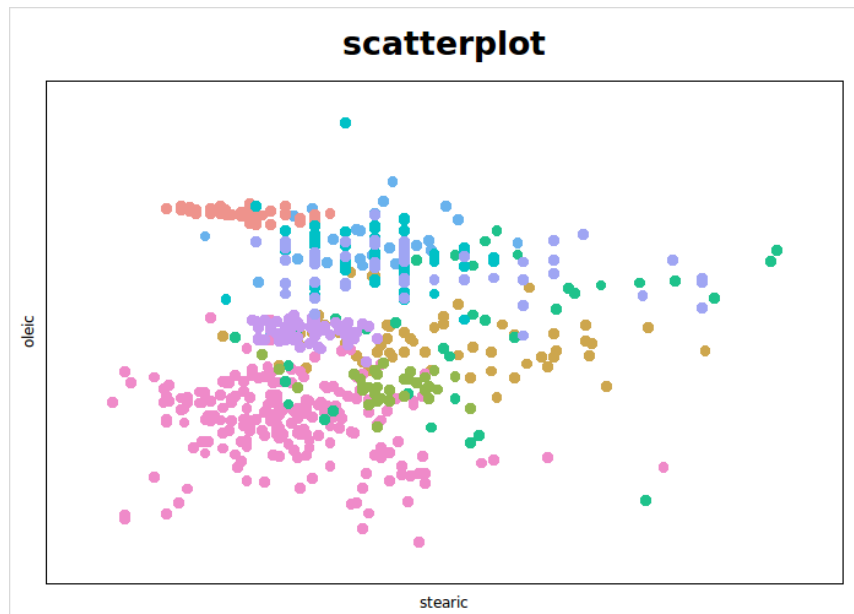


Figure 4.1: loon's scatterplot display.

Note that the histogram in [Figure 4.2\(a\)](#) is uni-colored although the color state was set to represent the areas of origin of the olive oils. This is loon's intended default behavior for histograms: the selected state is visually encoded whereas the color state is not. This way, the comparison between selected and overall distribution is based on common, aligned scales. According to the Cleveland and McGill experiments, this choice leads to good decoding of both distributions, see [Table 1.1](#). For example, in [Figure 4.2\(b\)](#) the selected points are the olive oils from the West-Liguria region.

```
h['selected'] <- Area == 'West-Liguria'
```

Setting the `showStackedColors` state to `TRUE` encodes the color state by creating stacked histograms, see [Figure 4.2\(c\)](#).


```
h['showStackedColors'] <- TRUE
```

However, to compare one of the stacked histograms to the overall distribution requires the analyst to compare positions along identical, nonaligned scales (except for the bottom histogram), which is sub-optimal according to [Table 1.1](#).

By default, the height of a histogram bar encodes the number of points represented by the bin. Setting the `yshows` state to `'density'` encodes the density instead of the counts, see [Figure 4.2\(d\)](#).

```
h['yshows'] <- 'density'  
l_scaletoworld(h)
```

Changing from counts to density changes the  $y$  values of the histogram bars. The `l_scaletoworld` function adjusts the zoom and pan such that the histogram maximizes its area in the display.

loon's histogram only supports equal bin widths. This allows us to introduce the graphical element  to interactively change the binning origin and the bin width. That is, the left side of the rectangle defines the binning origin, and the distance between the origin and the arrow tip defines the bin width.

### 4.1.3 Serialaxes Display

The serialaxes display encodes multiple variables by arranging their axes either in parallel (i.e. parallel axes) or as radii of a circle where the angles between the axes are equal (i.e. radial axes). Displays with parallel axes are called parallel coordinates plots [46]. In loon, we call the serialaxes display with radial axes a *stacked star glyph plot*. [Figure 4.3\(a\)](#) shows the stacked star glyph plot of the olive data.

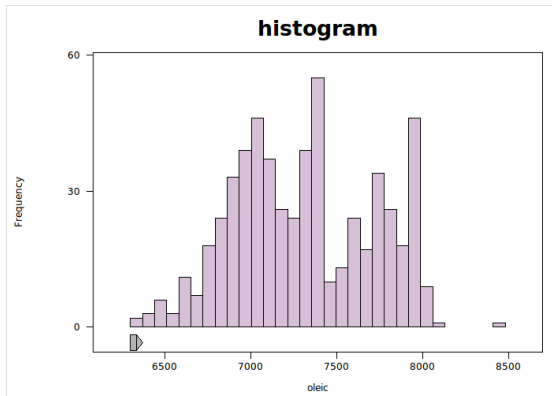
```
s <- l_serialaxes(data=oliveAcids,  
  color=Area, title='serialaxes: star glyphs')
```

Radial axes are the default display option in the serialaxes display. Switching from radial to parallel axes requires the `axesLayout` state to be set to `'parallel'`, see [Figure 4.3\(b\)](#).

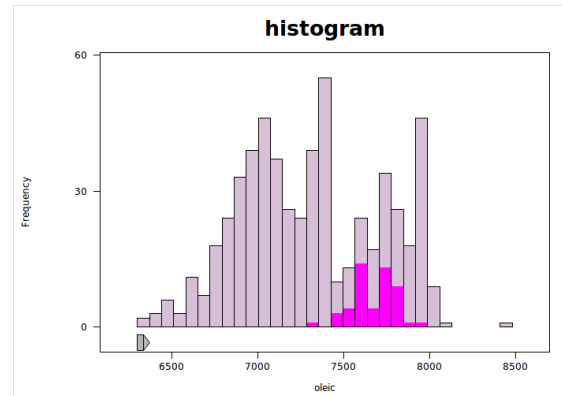
```
l_configure(s, axesLayout='parallel',  
  title='serialaxes: parallel coordinates')
```

### 4.1.4 Graph Display

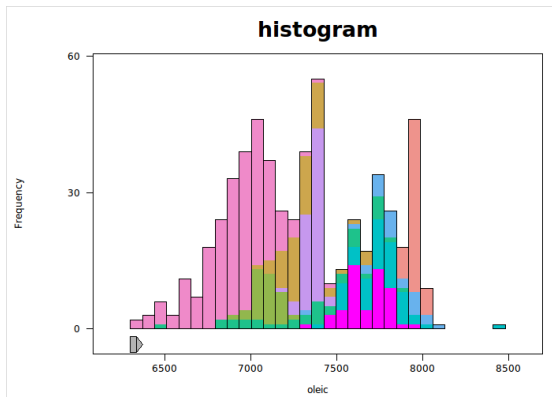
The graph display is closely related to the scatterplot display; however, the  $n$  dimensional states are now used as node attributes instead of points attributes. For example, the



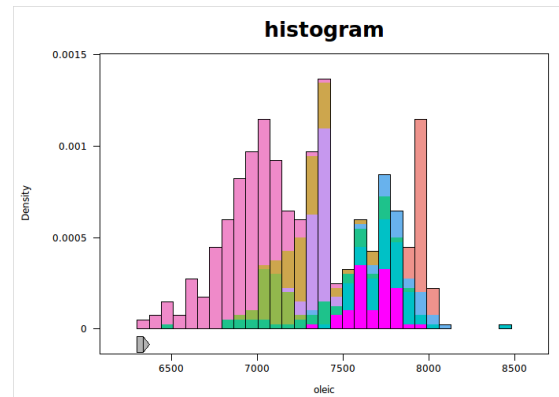
(a) default histogram



(b) selected points



(c) color stacked frequency

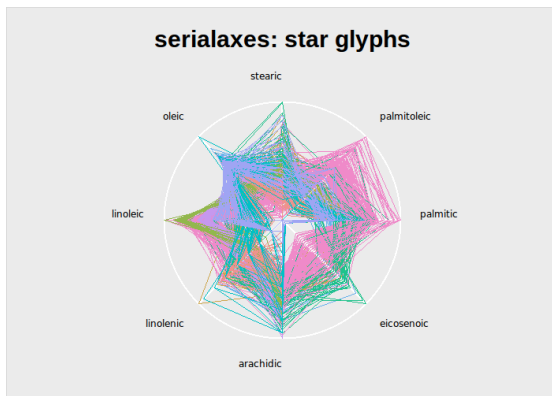


(d) color stacked density

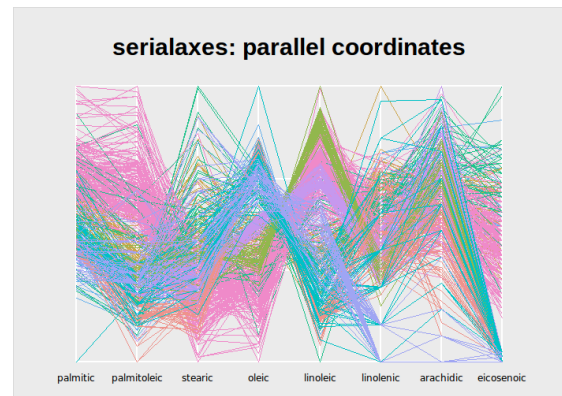
Figure 4.2: loon's histogram display.

graph's  $x$  and  $y$  states define the geometric graph layout (i.e. position of the nodes). The abstract graph is defined by the states `nodes`, `from`, `to` and `isDirected`. The `nodes` state specifies the node names, the `from` and `to` states require the node names to specify the edges, and the Boolean `isDirected` state specifies whether these edges are directed or not.

A graph display can be created by either specifying the `nodes`, `from`, `to` and `isDirected` states, or with graph objects of class `loongraph` or `graph`. The `graph` class is defined in the R package `graph` [36] and provides a more general data structure for graphs than the



(a) stacked star glyphs plot



(b) parallel coordinates plot

Figure 4.3: loon’s serialaxes display plots the data either as a stacked star glyphs plot (a) or as a parallel coordinates plot (b).

loongraph class. There are also a number of R packages that implement a variety of graph algorithms for graphs of class graph. We provide the loongraph class as a simple alternative to the graph class to create common navigation graphs without having to learn a new framework. For example, the three functions `completograph`, `linegraph` and `complement` create objects of class loongraph. The 3d transition graph for the olive data, as shown in Figure 4.4(a), is created as follows:

```
G <- completograph(nodes=names(oliveAcids))
LG <- linegraph(G)
g <- l_graph(LG)
```

The graph in Figure 4.4(a) is turned into a navigation graph by adding *navigators* (called “bullets” in RnavGraph):

```
nav1 <- l_navigator_add(g,
  from='palmitic:arachidic',
  to='palmitoleic:arachidic',
  proportion=0.28)
nav2 <- l_navigator_add(g,
  from='linoleic:linolenic',
  to='palmitoleic:linolenic',
  proportion=0.25,
  color='red')
```



Figure 4.4(b) shows the saturated 3d transition graph with two navigators. Note that loon permits multiple navigators per graph whereas RnavGraph permits only one navigator per graph. The two navigators in Figure 4.4(b) can be interactively dragged along the graph with no effect other than changing their position on the graph. In order to have a navigator driving another visualization, as we did with RnavGraph in Chapter 2, a *context* needs to be added to the navigator. For example, the “geodesic2d” context implements the canonical navigation graph semantic as described in Subsection 1.3.1.

```
l_context_add_geodesic2d(nav1, data=oliveAcids)
```

This creates a new loon scatterplot in which the projections controlled by the navigator nav1 are displayed.

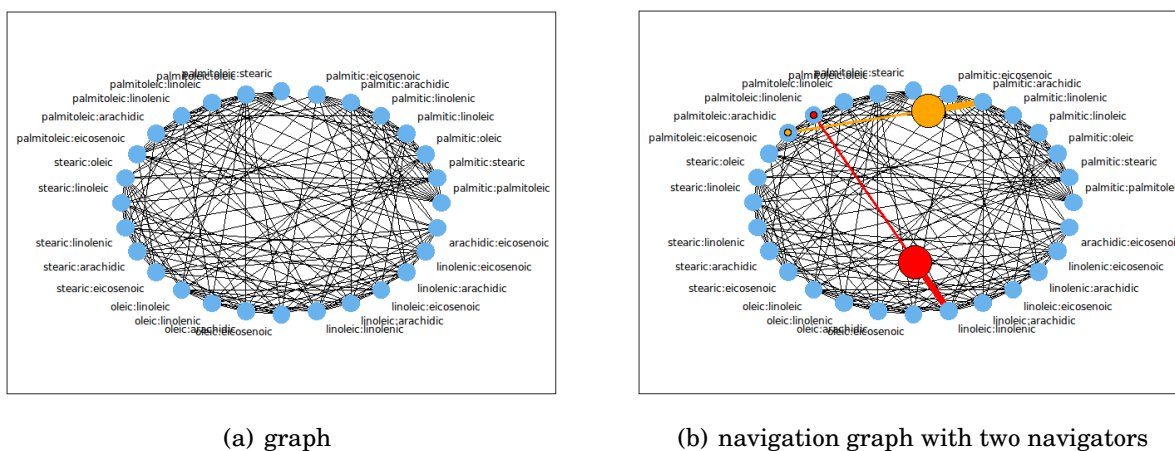


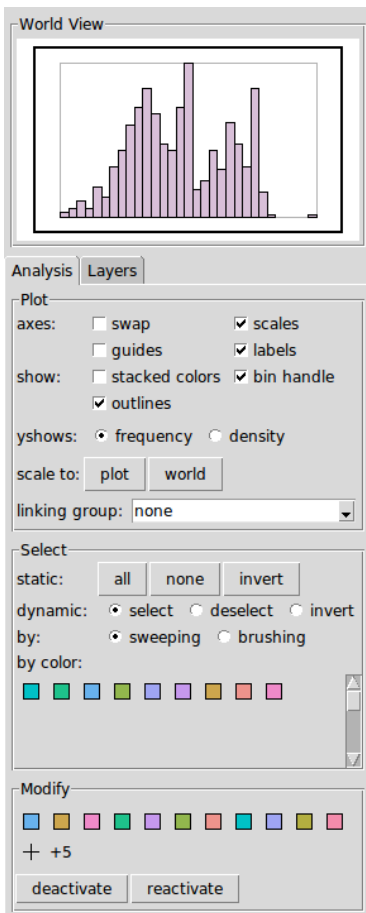
Figure 4.4: loon’s graph display.

### 4.1.5 Inspectors

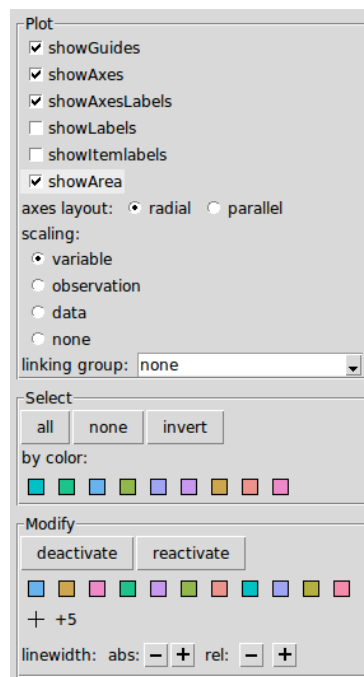
Each loon display is accompanied by an inspector. These inspectors are useful to efficiently interact with the displays; actions on the inspectors map to changes of the display states. The scatterplot inspector is shown in Figure 3.6(a), the histogram inspector in Figure 4.5(a), the serialaxes inspector in Figure 4.5(b), and the graph inspector in Figure 4.5(c). The histogram, scatterplot and graph inspectors are composed of multiple sub-

inspectors including the world view, the analysis inspector and the layers inspector. The serialaxes inspector does not contain a world view as the serialaxes display does not permit zooming and panning.

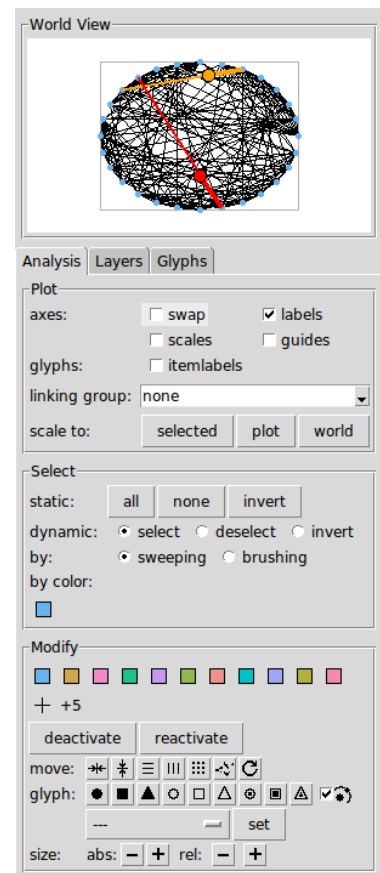
By default, these inspectors are displayed within an inspector called the loon inspector. The loon inspector is context-specific and shows the histogram, scatterplot, serialaxes or graph inspector for the display that the analyst is interacting with. The loon inspector is also a singleton, meaning there can be only one loon inspector at any time.



(a) Histogram Inspector



(b) Serialaxes Inspector



(c) Graph Inspector

Figure 4.5: loon's inspectors.

## 4.2 Main Graphics Model

The histogram, scatterplot and graph display plot the data in the Cartesian coordinate system. These displays share the same graphics model that has been abstracted in loon's software design. That is, these displays reuse code for zooming, panning, controlling the plot layout including scales and labels, and for layering visuals such as polygons and lines. We subsequently call this graphics model the *main graphics model*. As the serialaxes display is based on parallel or radial coordinates it does not use the main graphics model.

In this section, we discuss the plot layout and the mapping from data to display coordinates for the displays that are based on the main graphics model.

### 4.2.1 Plot Layout

The main graphics model splits the display area into a plot region, scales region, labels region and a minimum margins region as illustrated in [Figure 4.6](#). The three states



Figure 4.6: Main graphics model.

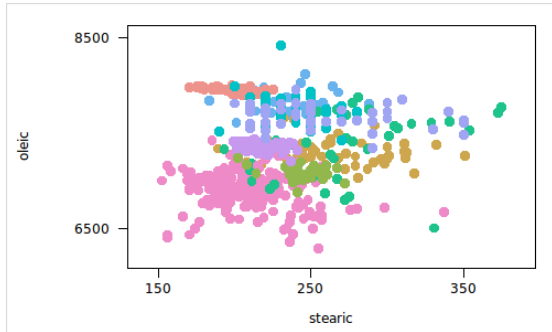
`minimumMargins`, `scalesMargins` and `labelMargins` control the size of these regions; that is, these states are 4-dimensional vectors that contain the bottom, left, top and right margins in pixel for the particular region, in the respective order. The labels and scales margins can be switched on and off by setting the boolean states `showLabels` and `showScales`

accordingly. The margins for the plot regions are not always additive, and the rules to determine the space for each region is as follows. If the `showScales` and `showLabels` states are both `FALSE` then only the plot region is shown; otherwise the margin between the plot region and the plot boundary is at least according to the `minimumMargins` state. Finally, a label or scale margin is only added if something is plotted in that region. For example, if the `title` state contains an empty string then the top margin of the labels region will not be taken into account; therefore, the top minimum margin will pad the plot region from the plot boundary. Also, as illustrated in [Figure 4.6](#), the left and top margins of the scales region, and the left margin of the labels region have end-effectively zero-width as currently there are no states for putting a label or scale on the right hand side of the plot region.

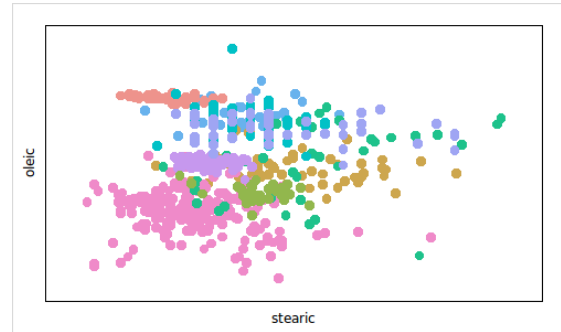
The choice for this particular plot layout is guided by practicality; the permutations of the two boolean states `showScales` and `showLabels` yield large plot regions. For example, [Figure 4.7](#) shows four scatterplots of `oleic` versus `stearic`. The R code to produce these four scatterplots is

```
p <- l_plot(x=stearic, y=oleic, color=Area,
           showScales=TRUE, showLabels=TRUE)
p['showScales'] <- FALSE
l_configure(p, showScales=TRUE, showLabels=FALSE)
p['showScales'] <- FALSE
```

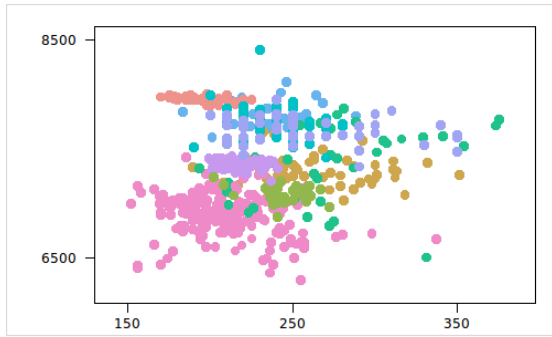
As we have not assigned a title to the `title` state of `p`, the space requested for the top label margin is 0 pixels in each plot in [Figure 4.7](#); therefore, the padding between the plot region and the plot boundary is according to the minimum margin as long as one of the states `showLabels` and `showScales` is set to `TRUE`. Hence, in [Figures 4.7\(a\) to 4.7\(c\)](#) the minimum margin is applied to the north- and east borders. [Figure 4.7\(d\)](#) omits the minimum margins altogether because neither the scales nor the labels are shown; this yields the maximum possible plot area. To display the minimum margins only, the `showLabels` state can be set to `TRUE` and the `xlabel`, `ylabel` and `title` states can be set to an empty string.



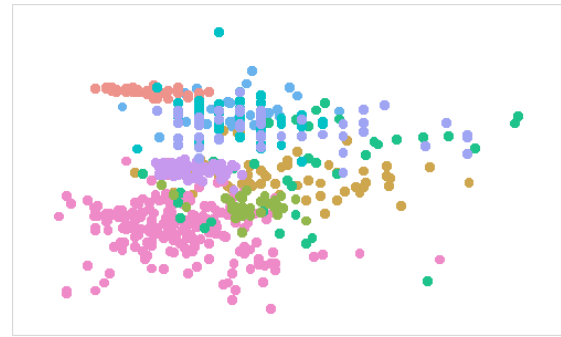
(a) scales and labels



(b) no scales



(c) no labels



(d) no scales and no labels

Figure 4.7: Plot layout

## 4.2.2 Mapping Data Onto the Plot Region

Mapping from data coordinates to plot region coordinates is controlled by the plot states `panX`, `panY`, `zoomX`, `zoomY`, `deltaX` and `deltaY` as follows. Assume the plot region is spanned by the plane with plot region coordinates (0,0) for the lower-left corner and (1,1) for the upper-right corner. Visuals such as point glyphs that fall outside the (0,0) – (1,1) plot region plane are clipped. Let  $(x_{pr}, y_{pr})$  be a point in plot region coordinates and  $(x_d, y_d)$  a point in data coordinates. Then, assuming the plot axes are not swapped, the mapping between data and plot region coordinates is

$$x_{pr} = \frac{x_d - \text{panX}}{\text{deltaX}} \cdot \text{zoomX}, \quad \text{and} \quad y_{pr} = \frac{y_d - \text{panY}}{\text{deltaY}} \cdot \text{zoomY}$$

With swapped axes, i.e. the state `swapAxes` is set to `TRUE`, the mapping is

$$y_{pr} = \frac{x_d - \text{panX}}{\text{deltaX}} \cdot \text{zoomX}, \quad \text{and} \quad x_{pr} = \frac{y_d - \text{panY}}{\text{deltaY}} \cdot \text{zoomY}$$

The `pan`, `zoom` and `delta` states define the region in the data coordinate system that is displayed in the plot region; if the axes are not swapped this region is defined by `(panX, panY)` and `(panX +  $\frac{\text{deltaX}}{\text{zoomX}}$ , panY +  $\frac{\text{deltaY}}{\text{zoomY}}$ )` for the lower-left and upper-right corner, respectively. Hence, the data range shown in the plot region is  $\frac{\text{deltaX}}{\text{zoomX}}$  and  $\frac{\text{deltaY}}{\text{zoomY}}$ . For example, the following code sets the plot states so that all the data in the (250, 7200) and (300, 7600) data coordinate region is displayed:

```
l_configure(p, panX=250, panY=7200, zoomX=p['deltaX']/50,
           zoomY=p['deltaY']/400)
```

When initializing a new plot, `loon` sets the states `deltaX` and `deltaY` based on the ranges of the data states `x` and `y` if not specified explicitly. If at plot initialization no data are supplied, e.g. by calling `l_plot()`, then `deltaX` and `deltaY` are set to 1. Further changes in the data will not affect the `deltaX` and `deltaY` states, as the aspect ratio of the plot depends on these two states. Choosing suitable `delta` values might be required to minimize computational errors in  $\frac{x_d}{\text{deltaX}}$  or  $\frac{y_d}{\text{deltaY}}$ .

The aspect ratio  $\alpha$  is defined by the ratio of the number of pixels for one data unit on the `y` axis and the number of pixels for one data unit on the `x` axes. The aspect ratio for `loon`'s plots that are based on the main graphics model can be queried and changed with the `l_aspect` and `l_aspect<-` functions, respectively. Changing the aspect ratio with `l_aspect<-` changes effectively the `zoomY` state to obtain the desired aspect ratio. Note that the aspect ratio in `loon` depends on the plot width, plot height and the states `zoomX`, `zoomY`, `deltaX`, `deltaY` and `swapAxes`. If the axes are not swapped then the aspect ratio is

$$\alpha = \frac{\text{plot height}}{\text{deltaY}/\text{zoomY}} \cdot \frac{\text{deltaX}/\text{zoomX}}{\text{plot width}}.$$

If the axes are swapped then the aspect ratio is

$$\alpha = \frac{\text{plot width}}{\text{deltaY}/\text{zoomY}} \cdot \frac{\text{deltaX}/\text{zoomX}}{\text{plot height}}.$$

The functions `l_scaletto_world`, `l_scaletto_selected`, `l_scaletto_layer`, `l_scaletto_active` and `l_scaletto_plot` modify the `zoomX`, `zoomY`, `panX` and `panY` states so that the particular region of the data (according to the function name) fills out the plot region with some padding.

The tick positions on the scales are determined using the algorithm from Talbot et al. [71]. Currently, neither the tick positions nor the tick labels can be set manually.

### 4.3 Plot States

All of loon's displays have plot states. Plot states specify what is displayed, how it is displayed and if and how the plot is linked with other loon plots. Some important plot states for loon's histogram, scatterplot, serialaxes and graph display are listed in [Table 4.1](#).

The *dimension* (column Dim in [Table 4.1](#)) of a state is either a numerical value, a letter or the keyword *any*. A numerical value specifies a plot state to be of a particular length. If the dimension is a letter such as  $n$  then the dimension is called an *abstract dimension* and all states with that dimension share the same actual value. The actual value of abstract dimensions can vary during the plot's lifetime. The keyword *any* specifies that the plot states can have any length (e.g. see the histogram state `colorStackingOrder` discussed in [Section 4.7.1](#)).

Once a plot is created, an actual value is assigned to each abstract dimension. For example, a scatterplot has  $n$  points. A scatterplot of the olive data will have  $n = 572$  and hence all of its plot states with dimension  $n$  will be vectors of length 572.

The dimension specifier  $0||n$  indicates that the vector can be of length zero or of length  $n$ . For example, for the scatterplot states `xTemp` and `yTemp` with dimension  $0||n$ , when the length of these states is according to  $n$  (as defined by `x` and `y`) then they define the  $x$  or  $y$  location of the points; when their length is 0, then the states `x` and `y` specify the location of the points.

For the histogram, scatterplot and serialaxes displays, the dimension  $n$  represents the number of data points. For the graph display, the dimension  $n$  represents the number of nodes whereas the dimension  $p$  represents the number of edges.

	Histo-gram	Scatter-plot	Graph	Serial-axes	Dim	Type	Default Value
Data	x	x, y			n	double	distributed on unit circle
		xTemp, yTemp			0    n	double	
			data		n	data.frame	
			nodes		n	string	
			from, to		p	string	
			isDirected		1	boolean	
Attributes		selected			n	boolean	FALSE
		active			n	boolean	TRUE
		color			n	color	steelblue
		size			n	double	4
			linewidth		n	pos. double	1
			pointlabel, tag		n	string	(point0,...,point<n-1>)
			activeEdge		p	boolean	TRUE
			selectedEdge		p	boolean	FALSE
		colorEdge		p	color	black	
Linking		linkingGroup			1	string	none
		linkingKey			n	string	(0,1,...,n-1)
Selection		selectBy			1	factor	<b>sweeping</b>   brushing
		selectionLogic			1	factor	<b>select</b>   deselect   invert
Plot Region		panX, panY			1	double	0
		zoomX, zoomY			1	pos. double	1
		deltaX, deltaY			1	pos. double	1
		minimumMargins			4	nneg. integer	(20, 20, 20, 20)
		labelMargins			4	nneg. integer	(30, 30, 60, 0)
		scalesMargins			4	nneg. integer	(30, 80, 0, 0)
		swapAxes			1	boolean	FALSE

Table 4.1: Important Display States. Dominant States do not have a default value. For states of type factor the default value column shows all possible factor levels and highlights the default factor level in bold.

The *type* of a state (column Type in Table 4.1) defines the data type (and sometimes the data structure) used by that state (as seen in R). All states in loon use a vector as their data structure; exceptions are states with the type `data.frame` which require an R data frame, and states with type `nested_*` which require a list of vectors. For example, a state with type `double` and dimension 1 uses a vector of length 1 to store its data. Hence, to



modify this state one needs to supply a vector of length 1 and type double.

As mentioned earlier, plot states are accessible via the '[' extract operation on a plot handle or with the `l_cget` function. For example, `p['deltaX']` and `l_cget(p, 'deltaX')` both return `deltaX` for the plot with handle `p`. The '['<-' setter operation on a plot handle and the `l_configure` function modify plot states. For example, `p['deltaX'] <-2` and `l_configure(p, deltaX=2)` both set the `deltaX` state of `p` to 2. While '['<-' can only modify one state at a time, the `l_configure` function can modify multiple states in one call. Changing multiple states with one expression is computationally more efficient than changing them sequentially. This is especially important when zooming towards the mouse pointer which usually changes `zoomX`, `zoomY`, `panX` and `panY` at the same time. If a single value is assigned to a state that has a vector data structure of length greater than one then that value is repeated accordingly.

Information about any state of a particular loon plot can be queried with the `l_info_states` function.

```
p <- l_plot()
info <- l_info_states(p)
```

The return value of `l_info_states`, here assigned to `info`, is a named list where the names are according to the plot state names

```
names(info)
 [1] "glyph"           "linkingGroup"    "linkingKey"
 [4] "zoomX"           "zoomY"           "panX"
 [7] "panY"            "deltaX"          "deltaY"
[10] "xlabel"          "ylabel"          "title"
[13] "showLabels"      "showScales"      "swapAxes"
[16] "showGuides"      "background"      "foreground"
[19] "guidesBackground" "guidelines"      "minimumMargins"
[22] "labelMargins"   "scalesMargins"  "x"
[25] "y"               "xTemp"           "yTemp"
[28] "color"           "selected"        "active"
[31] "size"            "pointlabel"     "tag"
[34] "showPointlabels" "useLoonInspector" "selectBy"
[37] "selectionLogic"
```

The elements of the `info` list are also named list with the elements type, dimension, default value and a description:

```
names(info$x)
[1] "type"          "dimension"      "defaultvalue"  "description"
```

For example, the description for `xTemp` state is extracted from `info` as follows

```
info$xTemp$description
[1] "if specified, these are x coordinates used instead those from
     the -x state"
```

### 4.3.1 Abstract Dimensions

An abstract dimension is a dimension defined by a letter, e.g.  $n$ . When creating a new plot without any arguments, all abstract dimensions have the actual value 0. For example,

```
p2 <- l_plot()
```

creates a plot with no points, hence  $n = 0$ . To change the actual value, the dominant states for that dimension have to be modified to have a common new length. The dominant states for all displays and abstract dimensions are listed in [Table 4.2](#). For example, to add points to `p2`, one can configure the dominant states `x` and `y` to have the same length

```
l_configure(p2, x=c(1,2,3), y=c(5,1,2))
```

Now  $n = 3$  and every non-dominant state with dimension  $n$  is reset to be a vector of length 3 containing their default values (see [Table 4.1](#) and the `defaultvalue` elements from `l_info_states`).

To modify states that have an abstract dimension for a subset of the data points, one can use the optional arguments `which`, `which_n` or `which_p`. The argument `which` and `which_n` are equivalent. When specified in `l_configure`, the `which_n` argument applies to all the other arguments that modify  $n$ -dimensional states.

`loon` supports three types of subsetting: logical subsetting, index subsetting and subsetting with a name of a boolean state that has the same dimension. For example, logical subsetting with `which_n` requires a boolean vector of length  $n$

Display	Dim	Dominant States
Histogram	$n$	$x$
Scatterplot	$n$	$x, y$
Serialaxes	$n$	data
Graph	$n$	nodes
Graph	$p$	from, to

Table 4.2: Dominant states for each Display.

```
p <- l_plot(oleic~stearic, color=Area)
l_configure(p, which_n=rep(c(FALSE,TRUE), each=286), color='green')
```

The `l_configure` expression above sets the color for the second half of the points to green. The color of the first half of the data points stays the same. Index subsetting requires a vector with indices of the vector elements that should be changed. Vector indices in R start with 1. The following code changes the color of the first, fifth and the last element of the olive points to green, red and blue

```
l_configure(p, which_n=c(1,5,572), color=c('green', 'red', 'blue'))
```

It is also possible to use the name of a boolean state to specify a subset of points to be modified. For example,

```
l_configure(p, which_n='selected', color='green', size=5)
```

changes the color of the selected points to green and their sizes to 5. Note that if a state name is used in `which_n` and modified in the same `l_configure` expression, as in

```
l_configure(p, which_n='selected', selected=FALSE, size=10)
```

then the `selected` state of `p` for `which_n` is used before the `configure` call; the order of the arguments is not important. Hence, the above call sets the size of all selected points to 10 and then deselects them.

Finally, the default value of `which_n` is the keyword `'all'`, meaning that all the elements of the configured  $n$ -dimensional states should be modified.

### 4.3.2 Configuration Pipeline

States are the heart of loon’s design. Every interaction with a plot revolves around states. For example, interactive selection, zooming and panning gestures result in changing one or multiple states. Also, (plot) linking is effectively a particular synchronization of certain plot states between the linked plots. Further, inspectors provide actions that modify plot states and, more importantly, the visualization (i.e. rendering) is based on the plot’s states and state changes. Hence, it is important that the states of a plot are valid and consistent at all times. We now present a top-level view of the stages of a state change request (e.g. with `l_configure`); we call this the *configuration pipeline*.

The `l_configure` function and `'[<-'` setter method pass their argument list (excluding the plot handle) to the configuration pipeline as shown in [Figure 4.8](#). If any of the elements in the argument list is invalid then the configuration pipeline will throw an appropriate error and leave the plot states unmodified.

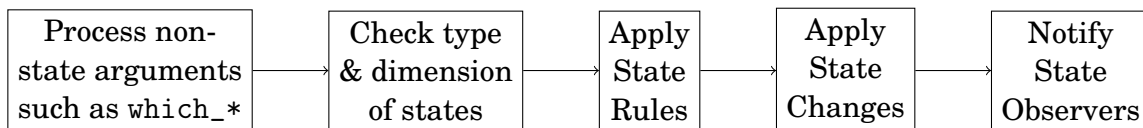


Figure 4.8: The configuration pipeline for state modifications.

In the first stage of the configuration pipeline, all the non-state arguments such as the `which` arguments are processed. If there is a `which` argument for a particular abstract dimension, say  $n$ , then all elements in the argument list that modify a state of dimension  $n$  are replaced by the appropriate  $n$  dimensional values. For example, if the argument list has the elements `which_n='selected'` and `size=10`, then the first stage of the configuration pipeline will replace the 10 in `size=10` with the an  $n$ -dimensional vector that is constructed by replacing the elements for the selected points in the current `size` state with 10. The `which_n='selected'` element is then removed from the argument list.

At the *Check type & dimension of states* stage, the argument list contains only elements that are state name and value pairs. These elements are then checked on whether they specify valid state names and whether their values are of the correct type and dimension.

If a single value is assigned to an  $n$ -dimensional state then, at this stage, the single value gets repeated  $n$  times and replaced in the argument list.

The *state rules* stage ensures that the plots states do not violate any *state consistency rules* or short *state rules*. For the histogram, scatterplot and serialaxes displays there is only one state rule: inactive points cannot be selected. One of the reasons for creating this state rule is the fact that many of the plot modification actions on the inspectors evaluate a configure call with `which_n='selected'`. Modifying only points that are visible is natural and prevents unexpected side effects. The graph display has state rules to prevent inactive nodes and edges from being selected, and to ensure that edges are only active if their adjoining nodes are active.

The *apply state changes* stage sets the plot states to the new values in the argument list. A new list that contains all the names of all states that were actually changed in the *apply state changes* stage. This list is used to notify all *observers* of the plot states that were effectively changed. Observers of plot states include loon's (plot) linking mechanism, the plots visualization (rendering), some of plot inspectors such as the "choose by color" tool, and user state bindings discussed later in [Subsection 5.2.2](#). It is possible for an observer to call the configure method of that plot while the plot is still in the configuration pipeline. In this case, a warning is thrown as unwanted side effects can happen if the next observer in line gets an outdated notification. In this case, it is recommended to use the `l_after_idle` function that evaluates some code once the processor is idle.

### 4.3.3 State Normalization

Different strings can represent the same concept in Tcl and Tk and the same is true for loon. For example, the strings TRUE, 't' and '1' are all valid Boolean Tcl values. Colors can also have different names in Tk. For example 'red', '#F00' and '#FF0000' all represent the same color in Tk.

loon accepts all valid Tcl Booleans. For example, for a plot

```
p3 <- l_plot(x=1:3, y=1:3, color='red')
```

the R expression

```
p3['selected'] <- '1'
```

will result in selecting all points as Tcl is a string-based programming language and the string '1' is a valid logical expression. Further, the boolean values for loon plot states are internally normalized (in the configuration pipeline, see [Section 4.3.2](#)) to TRUE or FALSE as in

```
p3['selected']
```

```
[1] TRUE TRUE TRUE
```

Color names in loon will be mapped to colors according to the Tk color specifications and are normalized to a 12 digit hexadecimal color representation. For example, for p3 we assigned to every point the color red but querying the color state will return '#FFFF00000000' for each point

```
p3['color']
```

```
[1] "#FFFF00000000" "#FFFF00000000" "#FFFF00000000"
```

This normalization is important when a plot state is queried and compared to a value. For example, to find out which points in p3 are currently active and red, one must use the 12 digit hexadecimal color representation for red

```
p3['active'] & (p3['color'] == '#FFFF00000000')
```

The convenience function `l_hexcolor` converts any valid Tk color string into its 12 digit hexadecimal color representation. For example, the above example is equivalent to `p3['active'] & (p3['color'] == l_hexcolor('red'))`

Note that R also maintains a list of valid color names (see the `colors` function). All R color names are also valid Tk color names but the actual color may differ (e.g. the color strings 'gray', 'grey', 'green', 'maroon' and 'purple' refer to different colors in R than in Tcl). In the case where it is important to obtain the exact color that a color name represents in R one can convert an R color name to its hexadecimal representation as follows

```
hexcolor <- function(color) {
  apply(col2rgb(color), 2, FUN=function(col) {
    rgb(col[1],col[2],col[3], maxColorValue=255)
  })
}
hexcolor(c('gray', 'green', 'maroon'))

[1] "#BEBEBE" "#00FF00" "#B03060"
```

## 4.4 Graphical User Interface

In this section, we discuss the two methods for interacting with loon's displays via mouse and keyboard gestures: the direct manipulations on the displays and loon's inspectors.

We decided to keep the keyboard gestures minimal; only the `Ctrl` and `Shift` modifier keys in combination with mouse gestures are used so that the interactions are easy to remember and the cross-platform compatibility is maximized.

### 4.4.1 Zoom & Pan

Figure 4.9 shows a chart of the keyboard-mouse gestures for zooming and panning within the histogram, scatterplot and graph display. Holding down either the `Ctrl` or the `Shift` key while zooming or panning restricts the direction of the respective action.

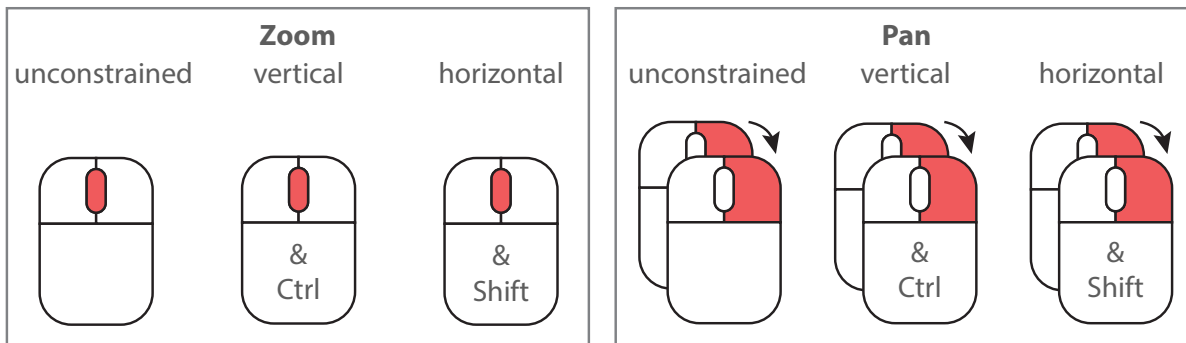


Figure 4.9: Zoom and pan gestures for the histogram, scatterplot and graph display. Zooming requires a mouse scroll gesture. Panning requires a right mouse button drag. Two superimposed mice with an arrow indicate a drag gesture.

## 4.4.2 Visual Query

There are two ways to query the data on loon’s displays. One way is to place the mouse cursor over a visual corresponding to a data point which will result in a “tool-tip” with a (point) *item label* being displayed. Another way to visually query the data is to select visuals on the display using mouse gestures and highlight the corresponding data points.

### 4.4.2.1 Item Labels

The scatterplot, graph and serialaxes displays represent each data point with a visual. With these displays, one can assign an *item label* to each data point and have these labels displayed with a “tool-tip” pop-up when resting the mouse cursor on the corresponding visuals, see [Figure 4.10](#). The item labels are stored in the  $n$ -dimensional `itemlabel` state, and the boolean `showItemlabels` state controls whether the labels are shown or not. The code to create the setting shown in [Figure 4.10](#) is

```
l_serialaxes(data=oliveAcids, color=Area,
             itemlabel=as.character(Region),
             showItemlabels=TRUE)
```

Note that visuals with no fill (such as polygons and rectangles with `color=''`) appear transparent, but they still absorb mouse events. Hence, it is possible that a point glyph





Figure 4.10: loon’s item labels that are displayed with a “tool-tip” pop-up.

can be seen but querying the item label will not work as there is a visual with no fill above that point glyph. The best way to avoid this situation is to keep the model layer on top of the rendering hierarchy (i.e. rendered last), see [Section 4.6](#).

### 4.4.2.2 Interactive Selection

Interactive selection of data points is a technique where the user selects visual objects on a display. For scatterplot displays the visual objects are point glyphs, for histograms they are bins, for graphs they are nodes and for serialaxes displays they are either star glyphs (radial axes) or lines (parallel axes). Interactive selection in loon always modifies the selected state. Selected items are highlighted magenta on the display. When it is possible for the visual items to be overplotted (e.g. point glyphs in the scatterplot display) the selected items are raised to be rendered on top of the other non-selected items. For plots based on the main graphics model (see [Section 4.2](#)) we support three ways to select visuals: individual item selection, selection by sweeping and selection by brushing with a rectangle. The serialaxes plot supports sweep selection using a line. An illustration of sweep and brush selection can be seen in [Figure 4.11](#). The keyboard mouse gestures for sweeping and brushing are outlined in [Figure 4.12](#).

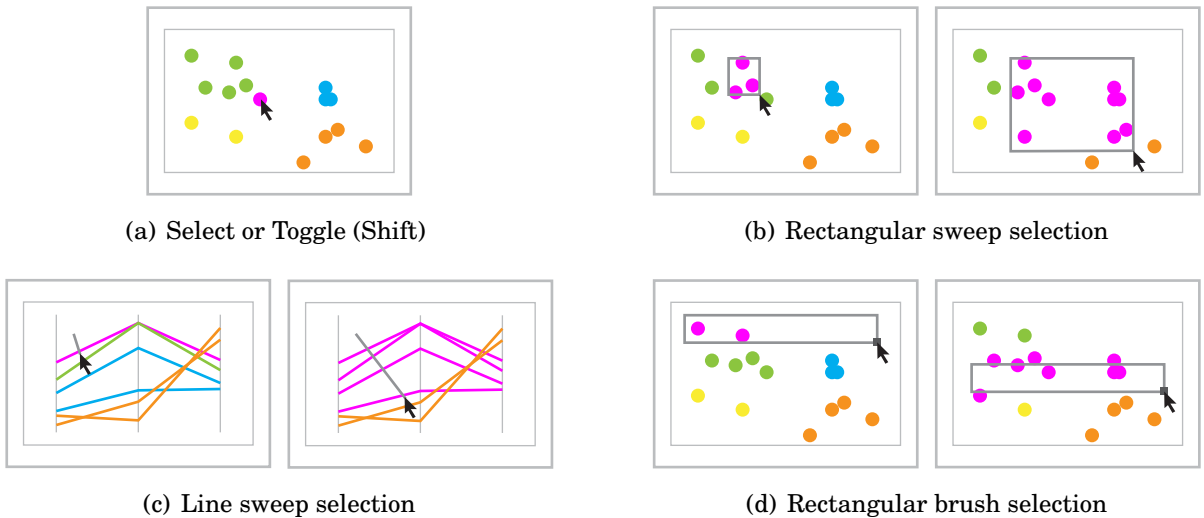
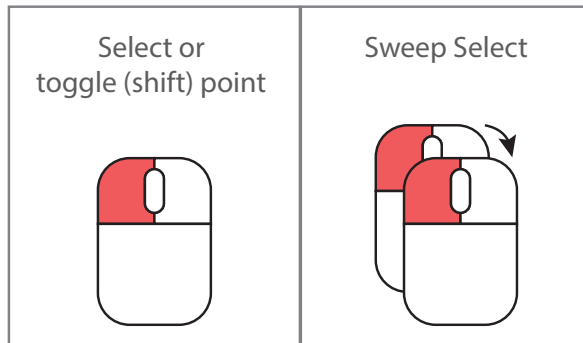


Figure 4.11: Interactive mouse/keyboard selection techniques.

In sweeping mode (i.e. `selectby='sweeping'`), pressing down the left button selects the item below the mouse cursor and deselects all the other items, hence resetting the previous selection. If there is no item below the mouse then all points are deselected.

## SWEEPING MODE

Shift will not reset previous selection



## BRUSHING MODE

Shift will make the selection permanent

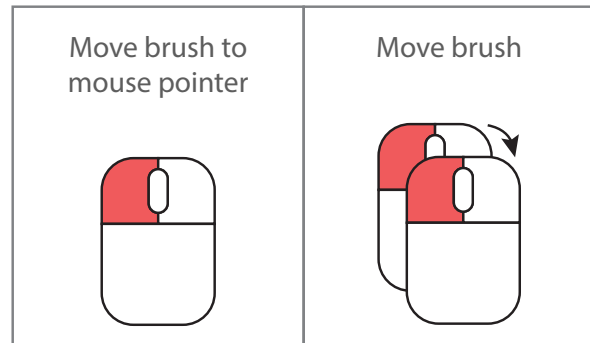


Figure 4.12: Selection gestures for the histogram, scatterplot and graph display. Two superimposed mice with an arrow indicate a drag gesture.

Holding down the `Shift` key while pressing the left button keeps the current selection and toggles the selection state for the points below the cursor. After holding down the left mouse button the first drag motion creates a sweep shape (a line or rectangle) and caches the selected state. Then, while sweeping (i.e. dragging while holding down the left mouse button), the plot's selected state is updated as follows. Let `p` be the plot handle, `sel_cached` be the cached selection and `ind_sweep` the indices of the points behind the sweep shape (e.g. rectangle or line). For the three `selectionLogic` state levels 'select', 'deselect' and 'invert', the selected state of `p` is updated while sweeping as follows:

- with `selectionLogic=select`:

```
sel <- sel_cached
sel[ind_sweep] <- TRUE
p['selected'] <- sel
```

- with `selectionLogic=deselect`:

```
sel <- sel_cached
sel[ind_sweep] <- FALSE
p['selected'] <- sel
```

- with `selectionLogic=invert`:

```
sel <- sel_cached
sel[ind_sweep] <- !sel[ind_sweep]
p['selected'] <- sel
```

Ending the sweep gesture by releasing the left button will delete the sweep shape.

In brushing mode (i.e. `selectby='brushing'`), the algorithm to update the plot's selected state is the same as the sweeping algorithm with the exception of brushing the points permanently by holding down the `Shift` key while `selectionLogic=invert`. That is, when switching to brushing the selected state is cached. Then, except for permanent invert brushing, when the brush is moved the selection logic is applied to the elements of the cached selected state. With permanent brushing, the cached selected state gets updated. With permanent invert brushing, a caching of the element indices of the points below the brush that are already toggled is necessary to avoid a constant toggling of the elements below the brush.

Finally, the brush size can be changed with the dark gray square on the lower left corner of the brush, see [Figure 4.11\(d\)](#).

### 4.4.3 Temporarily Relocating Points

Scatterplot and graph displays support interactive temporary relocation of single points (nodes for graphs) or of a group of selected points using the gestures illustrated in [Figure 4.13](#). Moving the points temporarily saves the new point coordinates to the states `xTemp` and `yTemp`. The dimension of `xTemp` and `yTemp` is either 0 or  $n$ . If `xTemp` or `yTemp` are not of length 0 then they are required to be of length  $n$ , and the scatterplot will display those coordinates instead of the coordinates in `x` or `y`. Hence, `xTemp` and `yTemp` are temporary as setting them to zero length (e.g. for a plot with handle `p`) with

```
l_configure(p, xTemp=c(), yTemp=c())
```

will cause the scatterplot to display the data stored in the `x` and `y` plot states.

In addition to using mouse motion gestures, one can also relocate data points with the inspector buttons and the functions listed in [Table 4.3](#). The `which` argument of these

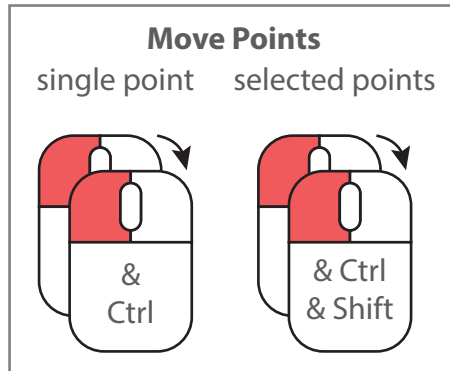


Figure 4.13: Temporary relocating points on a scatterplot.

functions specifies the points that should be temporarily rearranged. The valid subset specifications for which are the same as for the `which` argument of `l_configure` (explained in [Subsection 4.3.1](#)). The temporary rearrangement buttons on the inspector will move the selected points only.

Name	Description	Button on Inspector
<code>l_move_halign</code>	horizontally align	
<code>l_move_valign</code>	vertically align	
<code>l_move_hdist</code>	horizontally distribute	
<code>l_move_vdist</code>	vertically distribute	
<code>l_move_grid</code>	arrange on a grid	
<code>l_move_jitter</code>	jitter points	
<code>l_move_reset</code>	reset to x and y coordinates	

Table 4.3: Functions for temporarily moving points on scatterplot

When distributing points horizontally or vertically, their order remains the same. For example, when you distribute the point both horizontally and vertically, then the resulting scatterplot will be a plot of the  $y$  ranks versus the  $x$  ranks. The correlation on that plot will be Spearman's rho. When arranging points on a grid, some of the spatial ordering is

preserved by first determining a grid size (i.e.  $a \times b$  where  $a$  and  $b$  are the same or close numbers) and then by taking the  $a$  smallest values in the  $y$  direction and arrange them by their  $x$  order in the first row, then repeat for the remaining points.

#### 4.4.4 Inspectors

Table 4.4 lists all loon’s inspectors. In this section, we discuss the loon inspector, the worldview inspector and the layers inspector in detail.

Name	R creator function
loon inspector	<code>l_loon_inspector</code>
Worldview Inspector	<code>l_worldview</code>
Layers Inspector	<code>l_layers_inspector</code>
Scatterplot Inspector	<code>l_plot_inspector</code>
Scatterplot Analysis Inspector	<code>l_plot_inspector_analysis</code>
Glyphs Inspector	<code>l_glyphs_inspector</code>
Serialaxes Glyph Inspector	<code>l_glyphs_inspector_serialaxes</code>
Pointrange Glyph Inspector	<code>l_glyphs_inspector_pointrange</code>
Text Glyph Inspector	<code>l_glyphs_inspector_text</code>
Image Glyph Inspector	<code>l_glyphs_inspector_image</code>
Histogram Inspector	<code>l_hist_inspector</code>
Histogram Analysis Inspector	<code>l_hist_inspector_analysis</code>
Serialaxes (Analysis) Inspector	<code>l_serialaxes_inspector</code>
Graph Inspector	<code>l_graph_inspector</code>
Graph Analysis Inspector	<code>l_graph_inspector_analysis</code>
Graph Navigators Inspector	<code>l_graph_inspector_navigators</code>

Table 4.4: loon’s inspectors

#### 4.4.4.1 loon Inspector

The loon inspector is the default inspector and shows a histogram, scatterplot, serialaxes or graph inspector, depending on which display received the last mouse gesture input or window focus event. To detach a display from the loon inspector one can set the display state `useLoonInspector` to `FALSE`.

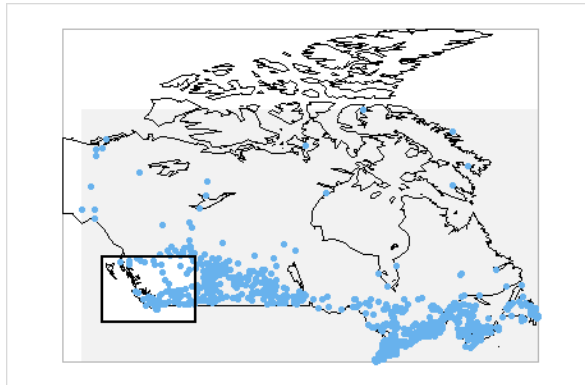
The loon inspector is a singleton, that is, there can be only one instance of it. Closing the loon inspector will result in loon creating a new loon inspector as soon as a display reporting to the loon inspector receives a mouse gesture input or window focus event.

All the other inspectors in [Table 4.4](#) are not singletons and not context specific. That is, they do not change their focus depending on which plot gets the last user input. Instead, they require that their `activewidget` state is set manually to the plot widget path name (i.e. plot handle).

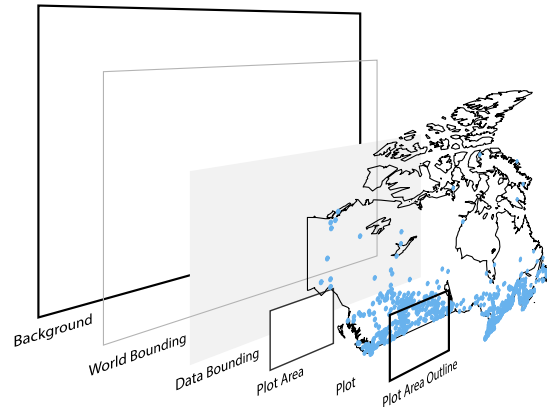
#### 4.4.4.2 Worldview

The worldview provides a view of all visual items on a display and adds visuals for the ranges of the data and for the area seen in the plot region. The worldview accepts a histogram, scatterplot or a graph display as its `activewidget`. [Figure 4.14\(a\)](#) shows the worldview for a scatterplot of Canadian city coordinates with the map of Canada layered and the focus on the British Columbia area. [Figure 4.14\(b\)](#) shows a perspective view of the composition of the worldview from [Figure 4.14\(a\)](#). In [Figure 4.14\(b\)](#), the “world bounding” outline shows the bounding box of all visual items (i.e. layers and plot data), and the “data bounding” rectangle shows the bounding box of the plot data.

The aspect ratio of the `activewidget` display is maintained in the worldview. For the scatterplot and graph displays, the point/node glyphs are always filled circles and the size state is not mapped to the glyphs. In addition to panning with a left-mouse button drag, the worldview supports the same panning and zooming gestures as the `activewidget` display.



(a) Worldview



(b) Perspective of worldview composition

Figure 4.14: Worldview inspector and its composition in perspective.

#### 4.4.4.3 Analysis Inspectors

The histogram, scatterplot, serialaxes and graph displays have their own analysis inspector. In general, the analysis inspector is composed of three sections: a Plot section with controls that can modify general plot options, a Select section with controls and actions for selecting points, and a Modify section that provides actions to modify certain plot states for the selected points.

#### 4.4.4.4 Layers Inspector

Layers are visual items that can be added to displays. The layers inspector shows the label, type and id for each added layer. Figure 4.15 show a layers inspector for a scatterplot display with three added layers. The buttons below the treeview provide the following actions for the selected layer (from left to right): move the layer down, move the layer up, move the layer below its parent, move the layer into the group below the layer, show the layer, hide the layer, add a new group layer, expunge the selected layer, and scale to the layer. The last row provides a text entry and set button to change the label of the selected layer.



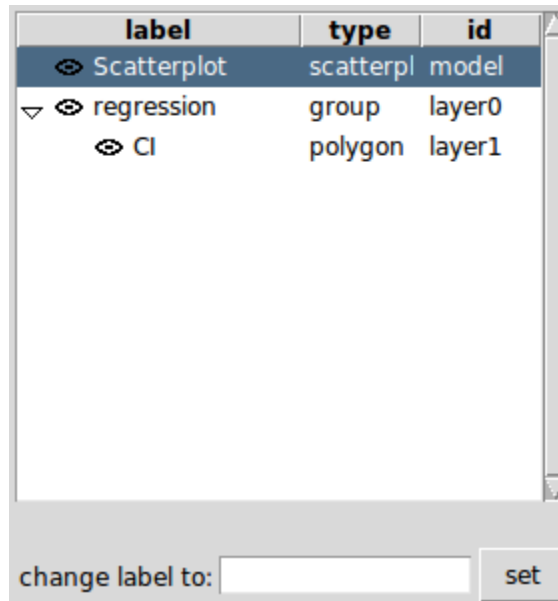


Figure 4.15: Layers inspector.

## 4.5 Standard Linking Model

In [Subsection 3.2.3](#), we discussed how to link displays with the `linkingGroup` state. This linking mechanism is actually based on two states, `linkingGroup` and `linkingKey`, and is called loon’s *standard linking model*.

The full capabilities of the standard linking model are described below. However, setting the `linkingGroup` states for two or more displays to the same string is generally all that is needed for linking displays that plot data from the same data frame. Changing the linking group of a display is also the only linking-related action available on the analysis inspectors as seen in [Figure 4.5](#); this combined pull-down menu with text-entry user-interface element lists all the currently used linking groups (and how many displays use that linking group) and allows the user to enter a new linking group or select one from the list.

The standard linking model uses three “levels” to determine how the data points are linked between displays. The first level identifies which displays are linked. The second

level defines the plot states of the displays that are linked (e.g. `selected`, `active` and `color`). The third level defines how the elements between linked states are mapped. That is, it is possible to define which plots, states and elements (points) are linked. Each of these linking levels can be configured at run time.

The first linking level is as follows: loon's displays are linked if they share the same string in their `linkingGroup` state. The default linking group `'none'` is a keyword and leaves a display un-linked.

The second linking level is as follows: all n-dimensional states can be linked between displays. We call these states *linkable*. Further, only linkable states with the *same* name can be linked between displays. One consequence of this “shared state name” rule is that, with the standard linking model, the `linewidth` state of a `serialaxes` display cannot be linked with the `size` state of a `scatterplot` display. Also, each display maintains a list that defines which of its linkable states should be used for linking; we call these states the “*used linkable*” states. The default “used linkable” states are listed in [Table 4.5](#) for each type of display. If any two displays are set to be linked (i.e. they share the same linking group) then the intersection of their “used linkable” states are actually linked.

Display	Default “used linkable” states
<code>scatterplot</code>	<code>selected</code> , <code>color</code> , <code>active</code> , <code>size</code>
<code>histogram</code>	<code>selected</code> , <code>color</code> , <code>active</code>
<code>serialaxes</code>	<code>selected</code> , <code>color</code> , <code>active</code>
<code>graph</code>	<code>selected</code> , <code>color</code> , <code>active</code> , <code>size</code>

Table 4.5: loon's default “used linkable” states.

For example, we create a `scatterplot` and a `histogram` for the olive data and link them by setting both linking groups to `'olive'`

```
p <- l_plot(stearic~oleic, linkingGroup='olive')
h <- l_hist(x=arachidic, linkingGroup='olive')
```

When creating the histogram `h`, the shared “used linkable” states of `p` and `h` are used for linking. We can query the “used linkable” states for each display with the `l_getLinkedStates` function as follows

```
l_getLinkedStates(p)
l_getLinkedStates(h)

[1] "color"      "selected" "active"    "size"
[1] "color"      "selected" "active"
```

That means that the states that are actually used for linking are `color`, `selected` and `active`.

It is possible to modify the “used linkable” states with the `l_setLinkedStates` function. For example, after the following expression, only the `selected` state is used for linking `p` and `h`.

```
l_setLinkedStates(p, c('selected', 'active', 'size'))
l_setLinkedStates(h, c('selected', 'color'))
```

Next, we create a `serialaxes` plot with the linking group `olive`

```
s <- l_serialaxes(data=oliveAcids, linkingGroup='olive')
```

For plot `s`, the “used linkable” states are the default ones as follows

```
l_getLinkedStates(s)

[1] "active"    "color"    "selected"
```

Now, the linking between the plots `p`, `h` and `s` is outlined in [Table 4.6](#).

State	linked among plots
selected	p, h, s
active	p, s
color	h, s

Table 4.6: linked states for example

The third linking level is as follows. Every display has a `n`-dimensional `linkingKey` state. Hence, every data point has an associated linking key. Data points between linked plots are linked if they share the same linking key. For example, for the two linked plots defined by

```
pA <- l_plot(x=1:5, y=1:5, linkingGroup='X',
             linkingKey=c('A','B','C','D','E'))
pB <- l_plot(x=1:3, y=1:3, linkingGroup='X',
             linkingKey=c('D','E','F'))
```

the fourth element of `pA` is linked with the first element of `pB`, and the fifth element of `pA` is linked with the second element of `pB` (for all linked states). Note that the `linkingKey` mechanism makes linking between plots with different actual values of the dimension  $n$  possible. The default `linkingKey` is the vector  $0, 1, 2, \dots, n - 1$  and `loon`'s linking (i.e. synchronization) speed is optimized for the default linking keys.

Linking keys must be unique within the `linkingKey` state of a display. Unique linking keys ensure that elements in a plot are only linked with elements of another plot and not within the same plot. This “unique linking key” constraint simplifies the default linking model. This one-to-one, zero-to-one or one-to-zero linking also facilitates the linking of seemingly many-to-one displays, such as histograms. With histograms, one bar represents one or many data points in the `x` state, and it should be noted that it is not the bar visual that is linked but the data points represented by that bar.

When changing the linking group of a display to one that is already used by other displays, `loon` needs to know in which direction the initial synchronization of the linked elements should go: from the linked displays already using that particular linking group to the display whose linking group changes or vice versa. The former option is specified with the argument `sync='pull'` in `l_configure` whereas the latter is specified with the argument `sync='push'`. Changing the linking group with an inspector will create a dialog asking whether to push or pull the linked states of the current plot. When changing the linking keys in a display that is linked, `loon` also needs to know whether to push or pull the linked states of the current display.

When `loon`'s standard linking model is too restrictive then it is possible to implement custom linking rules with state change event bindings, as described in [Section 5.3](#).

## 4.6 Layers

loon's displays that use the main graphics model (i.e. histogram, scatterplot and graph displays) support layering of visual information. [Table 4.7](#) lists the layer types and functions for layering on a display. Every layer within a display has a unique id. The visuals of

Type	Description	R creator function
group	a group can be a parent of other layers	<code>l_layer_group</code>
polygon	one polygon	<code>l_layer_polygon</code>
text	one text string	<code>l_layer_text</code>
line	one line (i.e. connected line segments)	<code>l_layer_line</code>
rectangle	one rectangle	<code>l_layer_rectangle</code>
oval	one oval	<code>l_layer_oval</code>
points	$n$ points (filled) circle	<code>l_layer_points</code>
texts	$n$ text strings	<code>l_layer_text</code>
polygons	$n$ polygons	<code>l_layer_polygons</code>
rectangles	$n$ rectangles	<code>l_layer_rectangles</code>
lines	$n$ sets of connected line segments	<code>l_layer_lines</code>

Table 4.7: loon's layer types

the data in a display present the default layer of that display and has the layer id `'model'`. For example, the `'model'` layer of a scatterplot display visualizes the scatterplot glyphs.

Layers are arranged in a tree structure with the tree root having the layer id `root`. The rendering order of the layers is according to a depth-first traversal of the layer tree. This tree also maintains a label and a visibility flag for each layer. The layer tree, layer ids, layer labels and the visibility of each layer are visualized in the layers inspector as seen in [Figure 4.15](#). If a layer is set to be invisible then it is not rendered on the display. If a group layer is set to be invisible then all its children are not rendered; however, the visibility flag of the children layers remain unchanged.

All layers have states that can be queried and modified using the same functions as the ones used for displays (i.e. `l_cget`, `l_configure`, `'['` and `'<-'`). The last group of layer types in [Table 4.7](#) have  $n$ -dimensional states, where the actual value of  $n$  can be different for every layer in a display.

The difference between the model layer and the other layers is that the model layer has a selected state, responds to selection gestures and supports linking.

For example, [Figure 4.16](#) shows a scatterplot with the fitted regression line, a 95% confidence interval and a 95% prediction interval from a simple linear regression model layered underneath the scatterplot points. The data for this example are generated as follows

```
set.seed(500)
x <- rnorm(30)
y <- 4 + 3*x + rnorm(30)
```

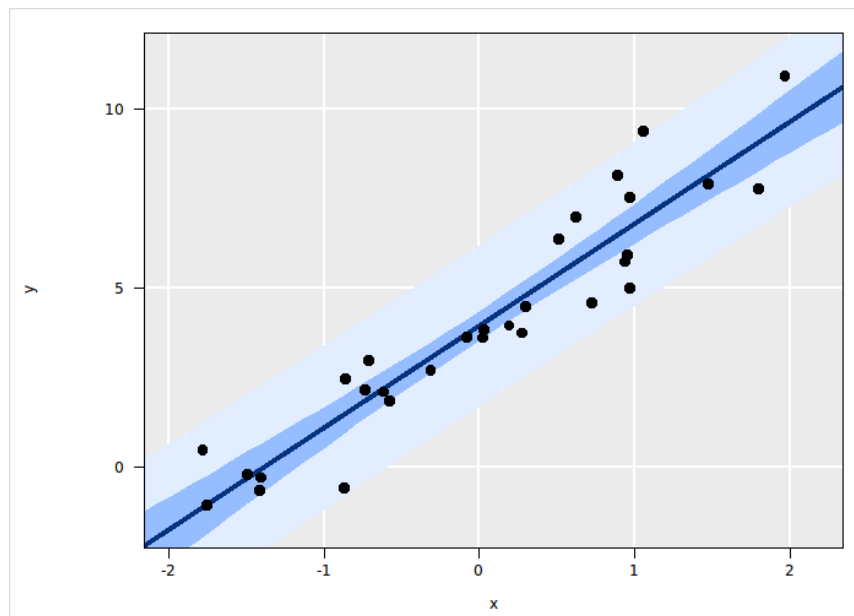


Figure 4.16: Scatterplot with a layered regression line, a 95% confidence interval and a 95% prediction interval of a simple linear fit.

Next, we fit the simple linear regression and obtain the coordinates of the regression line and the confidence and prediction intervals

```
fit <- lm(y~x)
xseq <- seq(min(x)-1, max(x)+1, length.out = 50)
fit_line <- predict(fit, data.frame(x=range(xseq)))
ci <- predict(fit, data.frame(x=xseq),
              interval="confidence", level=0.95)
pi <- predict(fit, data.frame(x=xseq),
              interval="prediction", level=0.95)
```

Finally, [Figure 4.16](#) is created by plotting  $y$  vs.  $x$ , adding a group layer and also adding the line and polygon layers that represent the regression line and confidence intervals, respectively:

```
p <- l_plot(y~x, color='black', showScales=TRUE, showGuides=TRUE)
gLayer <- l_layer_group(p, label="simple linear regression",
                       parent="root", index="end")
fitLayer <- l_layer_line(p, x=range(xseq), y=fit_line, color="#04327F",
                        linewidth=4, label="fit", parent=gLayer)
ciLayer <- l_layer_polygon(p,
                           x = c(xseq, rev(xseq)),
                           y = c(ci[, 'lwr'], rev(ci[, 'upr'])),
                           color = "#96BDFE", linecolor="",
                           label = "95 % confidence interval",
                           parent = gLayer, index='end')
piLayer <- l_layer_polygon(p,
                           x = c(xseq, rev(xseq)),
                           y = c(pi[, 'lwr'], rev(pi[, 'upr'])),
                           color = "#E2EDFE", linecolor="",
                           label = "95 % prediction interval",
                           parent = gLayer, index='end')
```

The index and parent arguments in the above code determine the location of the new layers in the layer tree. The returned layer handles can be used to access and modify layer states in the same way the plot handle `p` can be used to access and modify the plot states. For example,

```
fitLayer['dash'] <- c(10, 3, 3)
l_configure(ciLayer, linewidth=1, linecolor='black')
```

makes the regression line dashed and adds a black outline to the prediction interval polygon.

Table 4.8 lists the functions to query and modify the layer tree (used for the rendering order). Note that a layer labels and tree positions (i.e. parent and index) are not layer states but part of the display that supports layering.

### 4.6.1 Functions and Methods for Layering Data in R

With the loon R package we leverage some of R's functionality to create meaningful interactive plots. That is, we provide the `l_layer` generic function that can be used to write methods to layer information based on any R object.

We provide methods for layering geospatial data with objects from of the classes listed in Table 4.9. In Subsection 3.2.4, we discussed an example of layering an object of class `map` containing the Canadian country boundary coordinates, see Figure 3.2(b). Figure 4.17 shows a loon scatterplot display with multiple layers of 1 : 50m scale geospatial data from the Natural Earth project [55]. The information layered in the plot is: country boundaries, urban areas, rivers, lakes and international airports as scatterplot points. The methods used to layer the data in Figure 4.17 are the ones for objects of classes that are defined in the `sp` R package [10], see Table 4.9. The code to re-create Figure 4.17(b) can be found in the 'world' demo of the `naturalearth` R package, a package we will release soon.

We further provide the functions `l_layer_contourLines`, `l_layer_heatImage` and `l_layer_rasterImage` that are very similar and mostly compatible to the R functions `contourLines`, `image` and `rasterImage`, respectively. For example, Figure 4.18 shows the visual difference of an interactive loon plot and a base R plot when plotting a heat map and contour lines on a scatterplot. The 2d density estimation is for the `Sepal.Width` and `Sepal.Length` variables of the iris data from Anderson [3]:

```
kest <- with(iris, MASS::kde2d(Sepal.Width, Sepal.Length))
```

The R code to create Figure 4.18(a) is

```
p <- with(iris, l_plot(Sepal.Width, Sepal.Length, color='black', showScales=TRUE))
l_layer_contourLines(p, kest)
l_layer_heatImage(p, kest)
```



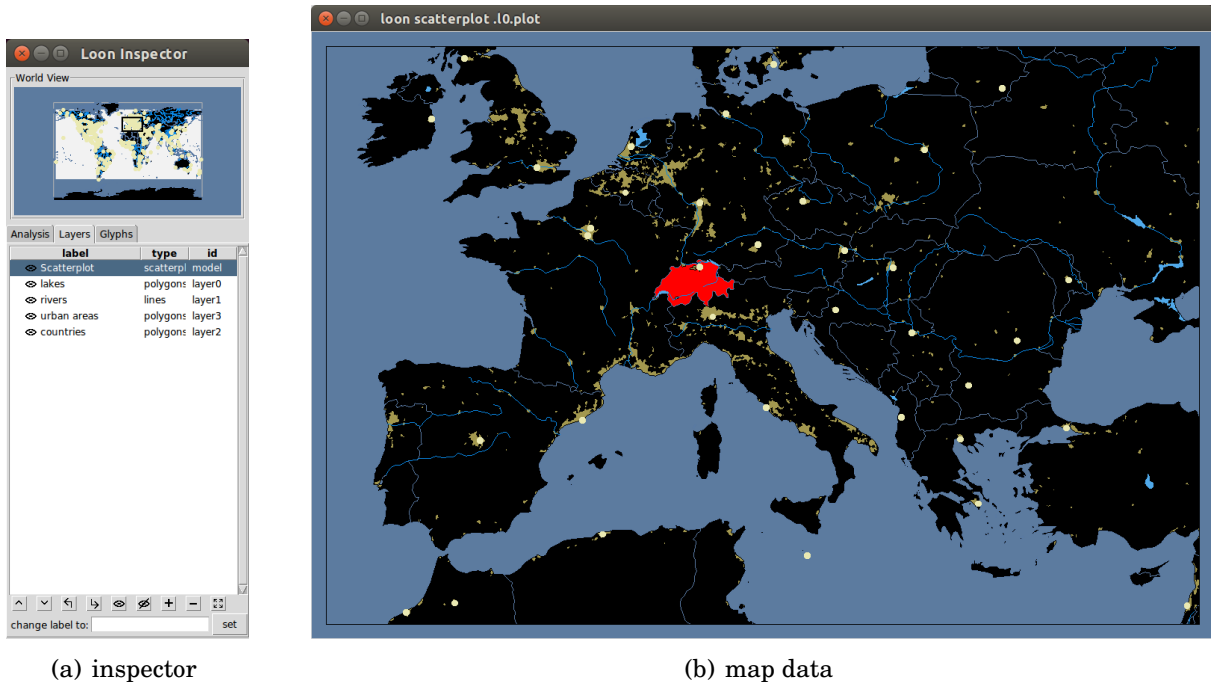


Figure 4.17: Naturalearth data displayed with loon's scatterplot display.

and the R code to create [Figure 4.18\(b\)](#) is

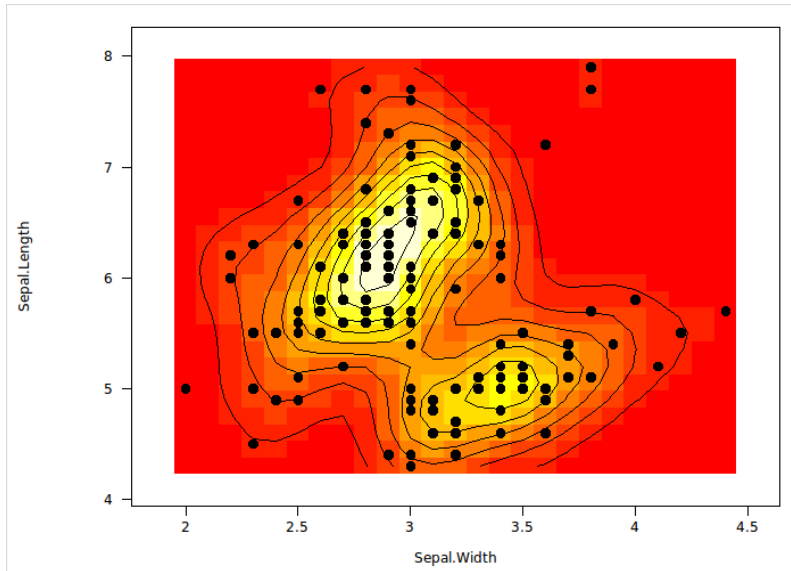
```
image(kest, xlab='Sepal.Width', ylab='Sepal.Length')
sapply(contourLines(kest), function(l) lines(l$x, l$y))
with(iris, points(Sepal.Width, Sepal.Length, pch=16))
```

Name	Description
<code>l_layer_ids</code>	List layer ids
<code>l_layer_getType</code>	Get layer type
<code>l_layer_getParent</code>	Get parent layer id of a layer
<code>l_layer_getChildren</code>	Get children of a group layer
<code>l_layer_index</code>	Get the order index of a layer among its siblings
<code>l_layer_printTree</code>	Print out the layer tree
<code>l_layer_move</code>	Move a layer
<code>l_layer_lower</code>	Switch the layer place with its sibling to the right
<code>l_layer_raise</code>	Switch the layer place with its sibling to the left
<code>l_layer_demote</code>	Moves the layer up to be a left sibling of its parent
<code>l_layer_promote</code>	Moves the layer to be a child of its right group layer sibling
<code>l_layer_hide</code>	Set the layers visibility flag to FALSE
<code>l_layer_show</code>	Set the layers visibility flag to TRUE
<code>l_layer_isVisible</code>	Return visibility flag of layer
<code>l_layer_layerVisibility</code>	Returns logical value for whether layer is actually seen
<code>l_layer_groupVisibility</code>	Returns all, part or none for expressing which part of the layers children are visible.
<code>l_layer_delete</code>	Delete a layer. If the layer is a group move all its children layers to the layers parent.
<code>l_layer_expunge</code>	Delete layer and all its children layer.
<code>l_layer_getLabel</code>	Get layer label.
<code>l_layer_relabel</code>	Change layer label.
<code>l_layer_bbox</code>	Get the bounding box of a layer.

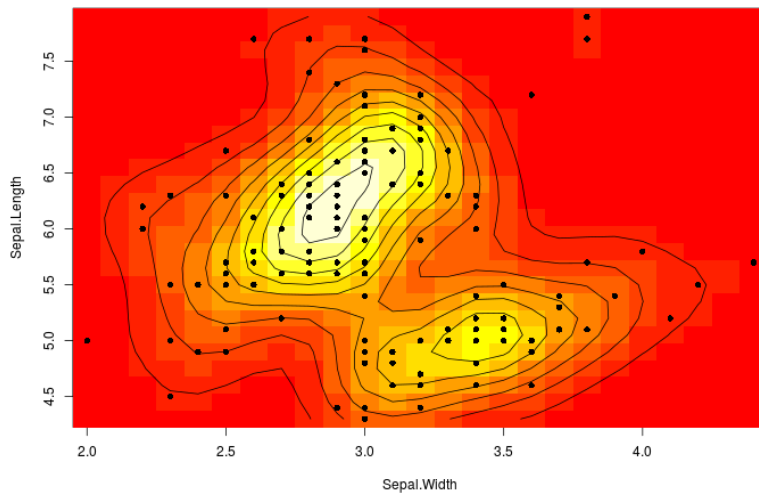
Table 4.8: Functions that work on layers

R package	Class
maps [8]	map
sp [10]	SpatialPoints SpatialPointsDataFrame Line Lines SpatialLines SpatialLinesDataFrame Polygon Polygons SpatialPolygons SpatialPolygonsDataFrame

Table 4.9: loon provides `l_layer` methods for the geospatial data classes in this table.



(a) loon version



(b) base R version

Figure 4.18: Two layers: a heat image and contour lines of a 2d density estimation.

## 4.7 Display Design Decisions

In this section, we re-visit some of loon's displays and elaborate on some relevant design decisions and functionality that have not been mentioned so far.

### 4.7.1 Histogram

The data in the  $n$ -dimensional state  $x$  of loon's histogram are binned before they are displayed. Every change in the selected, active or color states results in a re-binning of  $x$  and a change in the display accordingly.

If the boolean `showStackedColors` state is set to `TRUE` then the  $x$  data are partitioned into a "selected" group and then into a separate group for each color. Then, a histogram for every partition is stacked on the display. The default order of the stacking is as follows: first, the bins for the selected points are stacked (onto the line  $y = 0$ ); then, the remaining histograms for each color are stacked in the order of the appearance of their corresponding color in the color state (for the active points). To change this default stacking order one can specify it with the `colorStackingOrder` state; note that the keyword 'selected' is used to refer to the histogram of the selected points. Colors in the color state that do not appear in the `colorStackingOrder` are stacked last, also in the order of the appearance of their corresponding colors in color. The default value for `colorStackingOrder` is 'selected'.

Interactively selecting a bar on the histogram sets the selected state of all points that are represented by that bar to `TRUE`.

### 4.7.2 Point and Node Glyphs

The scatterplot and graph displays both have the  $n$ -dimensional state glyph that assigns each data point or graph node a glyph (i.e. a visual representation). Henceforth we only discuss point glyphs for scatterplot displays, but we keep in mind that all applies to graph displays as well.

We distinguish between primitive and non-primitive glyphs: the primitive glyphs are listed in [Table 4.10](#) and are always available for use whereas the non-primitive glyphs, listed in [Table 4.11](#), need to be first specified and added to a plot before they can be used.

Type	Name
● ○ ●	circle, ocircle, ccircle
■ □ ■	square, osquare, csquare
▲ △ ▲	triangle, otriangle, ctriangle
◆ ◇ ◆	diamond, odiamond, cdiamond

Table 4.10: Primitive point/node glyphs.

Type	R creator function
Text	<code>l_glyph_add_text</code>
Serialaxes	<code>l_glyph_add_serialaxes</code>
Pointranges	<code>l_glyph_add_pointrange</code>
Images	<code>l_glyph_add_image</code>
Polygon	<code>l_glyph_add_polygon</code>

Table 4.11: Non-primitive point/node glyphs and their creator function.

The following code creates a scatterplot with 15 points that are spread out horizontally. Then, a text glyph is added assigning the first 15 lower-case letters to the data points. Next, the glyph state is changed so that the first 12 points are displayed as primitive glyphs whereas the remaining three points are displayed as the non-primitive text glyphs added previously, see [Figure 4.19](#).

```
p <- l_plot(x=1:15, y=rep(0, 15), size=10, showLabels=FALSE)
text_glyph <- l_glyph_add_text(p, text=letters[1:15])
p['glyph'] <- c('circle', 'ocircle', 'ccircle',
               'square', 'osquare', 'csquare',
               'triangle', 'otriangle', 'ctriangle',
               'diamond', 'odiamond', 'cdiamond',
               rep(text_glyph, 3))
```



Figure 4.19: Point glyph examples.

All non-primitive glyphs have states that can be queried and modified using the same functions as the ones used for displays and layers (i.e. `l_cget`, `l_configure`, `'['` and `'[<- '`). All non-primitive glyphs have  $n$ -dimensional states (e.g. `text` for the text glyph). The dimension  $n$  of a point glyph is bound to the dimension  $n$  of a scatterplot display. Hence, non-primitive glyphs need to be defined for each data point in order to be added to a scatterplot display. If the actual value  $n$  of a scatterplot display changes then all its non-primitive glyphs are automatically deleted.

Every non-primitive glyph gets a unique glyph id that is returned by the glyph creator functions listed in Table 4.11. The functions listed in Table 4.12 can be used to list, delete, re-label, query the label and the glyph type. The glyph labels are used for naming glyphs in the glyph inspector and in the pull-down menu of the analysis inspector.

Function	Description
<code>l_glyph_ids</code>	List all glyph ids
<code>l_glyph_delete</code>	Delete a Glyph
<code>l_glyph_getLabel</code>	Get glyph label
<code>l_glyph_relabel</code>	relabel glyph
<code>l_glyph_getType</code>	get glyph type

Table 4.12: Functions for working with glyphs

The `size` state of the scatterplot display assigns a size attribute to the glyph corresponding to each data point. The mapping from size to the area of the glyph (in `pixel2`) is listed in Table 4.13 and shown in Figure 4.20. For the polygon glyph, it is the user's

responsibility to center the polygons at (0,0) and to determine an appropriate size of the polygons with the polygon coordinates.

We chose the size mappings in [Table 4.12](#) such that the glyph area grows proportionally with the size (except for polygon and text glyphs), with proportionality factors that seemed satisfactory to us. The image, serialaxes and polygon glyphs grow faster with the size of a data point than the primitive glyphs as they usually represent a lot of information. This information should be accessible by only a few button clicks via the size change actions on the analysis inspector.

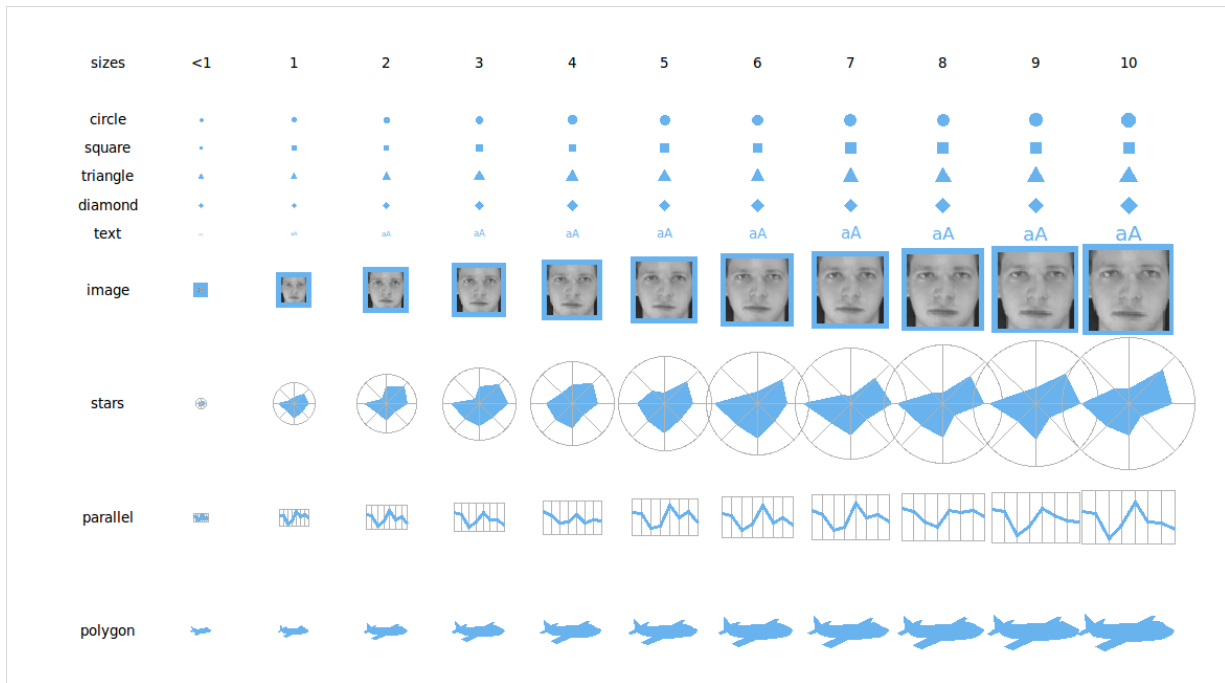


Figure 4.20: Point glyph size mapping.

The glyphs inspector as seen in [Figure 4.21](#) provides lists all non-primitive glyphs for a scatterplot display and has a glyph specific control panel. The glyphs control panel presently provides actions for modifying the serialaxes glyphs only.



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

Table 4.13: The mapping of size to point glyph area.

### 4.7.3 Serialaxes Display and Serialaxes Glyphs

The data state for the serialaxes display and serialaxes glyphs contains the data frame that is mapped onto the axes. The data value to axis position mapping is from  $[0, 1]$  to the axes range. We provide a number of data scaling options with the scaling state:

- `scaling='variable'`: every column of data gets scaled to the range  $[0, 1]$ .
- `scaling='observation'`: every row of data gets scaled to the range  $[0, 1]$ .

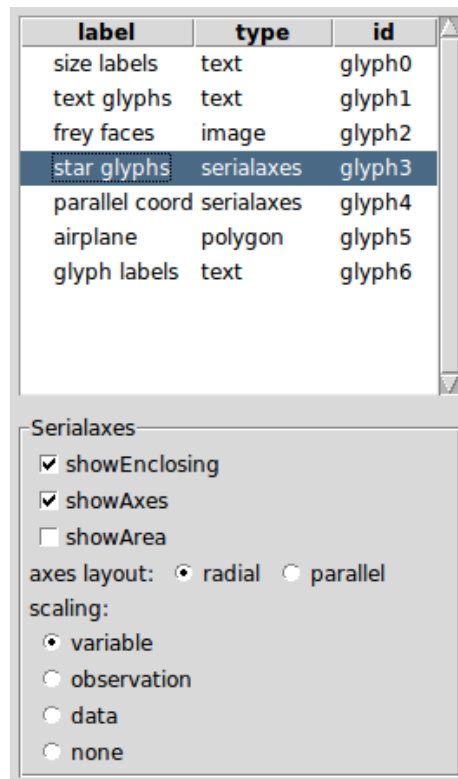


Figure 4.21: Glyphs inspector.

- `scaling='data'`: the data as a whole gets scaled to the range [0,1].
- `scaling='none'`: the data is not scaled before it is mapped to the serial axes axes. Values smaller than 0 or greater than one will lead to visual inconsistencies with the axes visuals.

The sequence state defines the axes sequence of the variables. Any length for the sequence is possible. The `axesLayout` state specifies whether to display the serial axes glyphs as parallel coordinates (i.e. 'parallel'), or as star glyphs (i.e. 'radial').

## 4.7.4 Graph Display

We now discuss graph display related topics that are useful for working with navigation graphs.

### 4.7.4.1 Graphswitch

The graphswitch provides a graphical user interface element that is used for changing the graph in a graph display interactively. The functions to create and work with the graphswitch are listed in [Table 4.14](#). Selecting a graph on the graphswitch replaces the graph in the display (defined in the activewidget state of the graphswitch) with the selected graph on the graphswitch.

Function	Description
<code>l_graphswitch</code>	Widget creator function
<code>l_graphswitch_add</code>	add a graph
<code>l_graphswitch_delete</code>	delete a graph
<code>l_graphswitch_get</code>	return a graph as a loongraph object
<code>l_graphswitch_getLabel</code>	get a graph label
<code>l_graphswitch_relabel</code>	change a graph label
<code>l_graphswitch_ids</code>	list all graphs in the graph switch
<code>l_graphswitch_move</code>	move a graph in the list
<code>l_graphswitch_reorder</code>	set a new graph order
<code>l_graphswitch_set</code>	change the graph of the activewidget to the currently selected one

Table 4.14: loon's graphswitch widget.

For example, the following code creates a graph display and a graphswitch as seen in [Figure 4.22](#), and adds three graphs to the graph widget. Selecting a graph on the graphswitch will update the nodes, from, to and isDirected states of the graph display correspondingly.

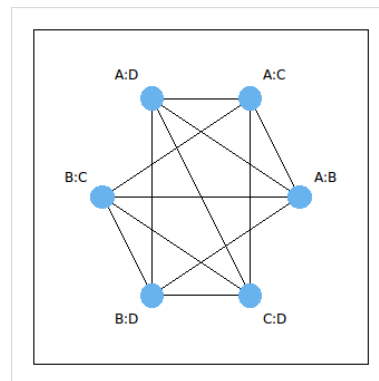
```
g <- l_graph()
gs <- l_graphswitch(activewidget=g)

G <- completegraph(nodes=c('A','B','C','D'))
LG <- linegraph(G)
LGnot <- complement(LG)

l_graphswitch_add(gs, G, label='variable graph')
l_graphswitch_add(gs, LG, label='3d transition graph')
l_graphswitch_add(gs, LGnot, label='4d transition graph')
```

label	id
variable graph	graph0
3d transition graph	graph1
4d transition graph	graph2

(a) Graphswitch



(b) Graph display

Figure 4.22: Graphswitch and graph display.

#### 4.7.4.2 Navigators

Navigators turn a graph into a navigation graph; they represent a “you are here” bullet that can be dragged along the graph. Navigators are constrained to be positioned on the graph and any number of navigators can be added to a graph display. [Table 4.15](#) lists all navigator-related functions. Navigators have states that can be queried and modified. The navigator states from and to define a path on the graph using node names. The

Name	Description
<code>l_navigator_add</code>	add a navigator
<code>l_navigator_ids</code>	list navigator ids
<code>l_navigator_delete</code>	delete a navigator
<code>l_navigator_getLabel</code>	query the label of a navigator
<code>l_navigator_relabel</code>	modify the label of a navigator
<code>l_navigator_walk_path</code>	have to navigator walk a path
<code>l_navigator_walk_backward</code>	walk on the current path forward to a node
<code>l_navigator_walk_forward</code>	backwards to a node

Table 4.15: Working with navigators.

navigator is located between the last node given by its from state and the first node given by its the to state at a relative distance equal to its proportion state. If to is empty then the navigator is located on the last node in from. For example, the navigation graph in [Figure 4.23](#) was created with the following code

```

1 G <- completegraph(LETTERS[1:4])
2 LG <- linegraph(G)
3 g <- l_graph(LG, showLabels=FALSE)
4 nav <- l_navigator_add(g)
5 l_configure(nav, from = c('A:B', 'B:C', 'A:C'),
6     to = c('C:D', 'B:D'),
7     proportion= 0.3)
8 g['activeNavigator'] <- nav

```

Lines 5–7 set the path and the navigator position. The path is highlighted with the color of the navigator, and the from path is represented by a thicker line than the to part of the path. The path end node has a small filled circle with the color of the path. Line 8 sets the navigator nav as the active navigator of g. A graph display can have at most one active navigator which is outlined magenta. When the activeNavigator graph state is set to a navigator id instead of '' then the user interactions with the graph display change as follows:

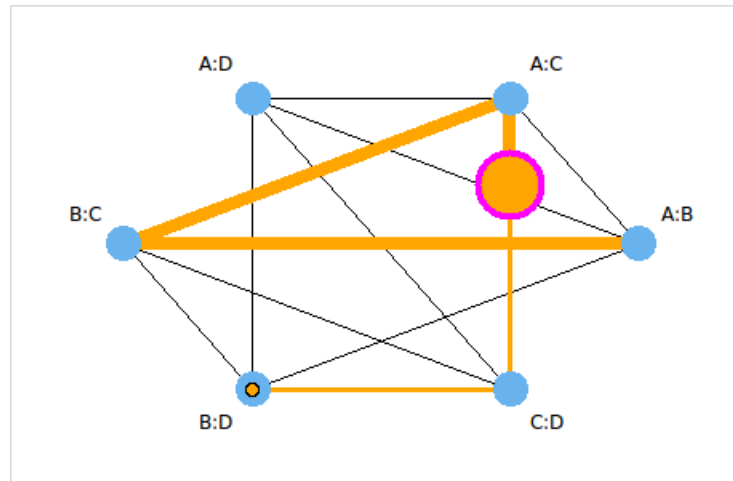


Figure 4.23: Navigator example.

- A scroll mouse event moves the navigator along its path rather than zooming.
- A left button press on a node will move the navigator to that node.
- A `Shift` key press will highlight all adjoining nodes to the path end highlighted with a small circle in the color of the navigator, as seen in [Figure 4.23](#).
- A `Shift` left button press on adjoining node to the path end will add that node to the path end, i.e. append it to the to navigator state.
- A `Ctrl` left button double press on a node that lies on the path will cause the navigator to “walk” towards that node (i.e. the navigator is animated).
- A left button press on a visual that is not a node will change the activeNavigator state to either another navigator (if a navigator was selected) or to ' '.
- If the navigator is on the path end (i.e. `to=' '`) then dragging the navigator towards a new node will extend the path accordingly.

During interactions with the navigator the graph nodes and edges get temporarily highlighted for visual guidance.

It is also possible to “walk” a path with the last three functions listed in [Table 4.15](#). A path “walk” is an animation of the navigator along a path. The navigator’s “walking” speed can be modified with the `animationProportionIncrement` and `animationPause` navigator states.

### 4.7.4.3 Navigator Contexts

A navigator context sets up an environment that gives a meaning to the navigator’s position on the graph; that is, a navigator context implements a graph semantic. [Table 4.16](#) lists the context-related R functions. We currently provide three contexts: the `context2d`, `geodesic2d` and `slicing2d`. Next, we discuss each of these contexts.

Name	Description
<code>l_context_add_context2d</code>	Add a <code>context2d</code> context
<code>l_context_add_geodesic2d</code>	Add a <code>geodesic2d</code> context
<code>l_context_add_slicing2d</code>	Add a <code>slicing2d</code> context
<code>l_context_ids</code>	List all contexts of a navigator
<code>l_context_delete</code>	Delete a navigator

Table 4.16: Navigator context-related functions.

### Context2d

The `context2d` maps every location on a 2d space graph to a list of `xvars` and a list of `yvars` such that, while moving the navigator along the graph, as few changes as possible take place in `xvars` and `yvars`, see [Figure 4.24](#). One of the cases where this functionality is useful is with the canonical graph semantic; there, using the `context2d` helps avoid an abrupt swapping of the axes in the scatterplot display when arriving at a node. The `context2d` uses its separator state to split the graph node names into 2d spaces. The `interchange4d` context state switches the combination of variables in `xvars` and `yvars`. For example, when transitioning from node `A:B` to `C:D` and assuming that `interchange4d` is `FALSE`, `xvars` is `(A,C)` and `yvars` is `(B,D)`, then switching `interchange4d` to `TRUE` will

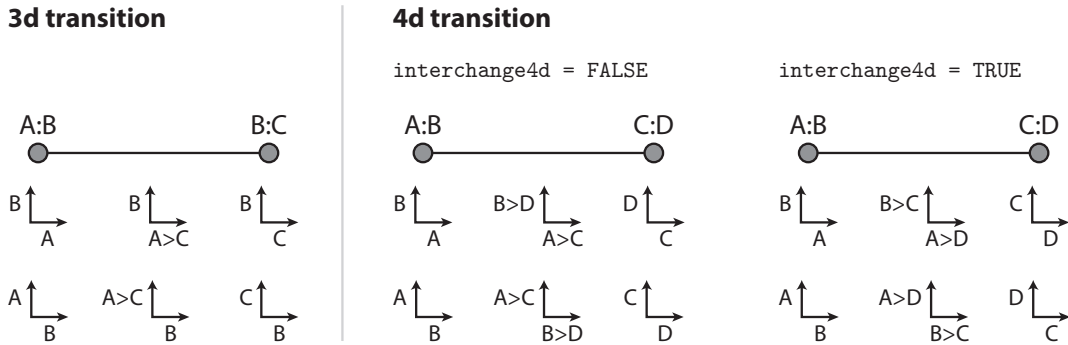


Figure 4.24: Context2d mapping scheme to xvars and yvars.

result in xvars being (A,D) and yvars being (B,C). The `interchange4d` state can be toggled with a double click on the navigator.

We now show how to print the xvars and yvars lists of a `context2d` to the R prompt while moving the navigator. Note that every navigator change will evaluate the callback code in the 'command' state of the `context2d`.

```
G <- completograph(nodes=c('A','B','C','D'))
LG <- linegraph(G)
g <- l_graph(LG)
nav <- l_navigator_add(g)
l_context_add_context2d(nav, command=function(xvars, yvars) {
  cat(paste0('xvars=', xvars, ', and yvars=', yvars, '\n'))
})
```

We will further discuss the use of callback functions in [Subsection 5.2.1](#).

## Geodesic2d

The `geodesic2d` context implements the canonical navigation graph semantic as discussed in [Subsection 1.3.1](#). The `geodesic2d` context is an extension of `context2d` and uses its xvars and yvars together with the proportion state of the navigator to determine the corresponding projections (recall that every location on the graph corresponds to a 2d projection).



Similarly to the `context2d`, the `geodesic2d` context has a `command` state that is evaluated with every change of the navigator's location. If the `command` state is not specified when the `geodesic2d` context is created then the default behavior of the `geodesic2d` context is to create a loon scatterplot whose coordinates are updated with the projection coordinates as specified in the `command` state.

The following example demonstrates how the `geodesic2d` context works for a 3d transition graph.

```

1 G <- completegraph(names(oliveAcids))
2 LG <- linegraph(G)
3 g <- l_graph(LG)
4 nav <- l_navigator_add(g)
5 cg2d <- l_context_add_geodesic2d(nav, data=oliveAcids)
6 attr(cg2d, 'plot')['color'] <- Area

```

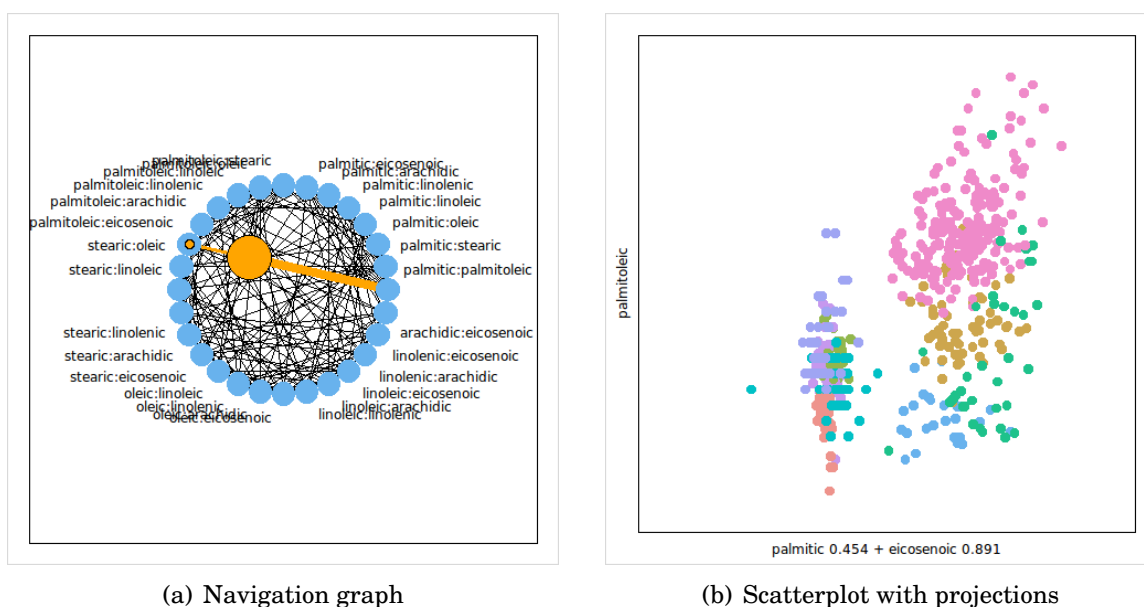


Figure 4.25: Context2d example session.

Figure 4.25 shows the setup created with the above code. Line 6 sets the color of the scatterplot points according to the olive `Area` variable. As we did not specify a `command` callback when creating the `geodesic2d` context on line 5, the `geodesic2d` function will create

a loon scatterplot for the corresponding projections with the plot handle attached as a 'plot' attribute to the context handle.

The default command contains the code to update the scatterplot with the projection coordinates and is

```
cg2d['command']
```

```
[1] ".l2.plot configure -x %x -y %y -xlabel %xlabel -ylabel %ylabel"
```

This is Tcl code and the %x and %y are substituted with the projection coordinates, and the %xlabel and %ylabel are substituted with suitable axes labels, see [Figure 4.25](#). These are advanced topics that are further discussed in the next chapter. However, it should be noted that this design allows one to use any R functionality and R graphic device to work with and/or display the projections. For example, to display the projections with an base R plot one could use:

```
cg2d['command'] <- function(x, y, xlabel, ylabel) {  
  plot(l_toR(x), l_toR(y), xlab=xlabel, ylab=ylabel)  
}
```

The data state of a geodesic2d context contains the data used for projections. The variable names of the data state need to match the node names of the graph. That is, the geodesic2d context uses its separator state to split each graph node name into two variable names. Then the geodesic2d looks up these variable names in the data contained by its data state.

The context2d state scaling determines the scaling performed on the data before projecting them. The different scaling methods are listed in [Table 4.17](#). By default the geodesic2d context has variable as the default scaling state.

## Slicing2d

A slice of a data set is a subset of its observations. The slicing2d context implements slicing using navigation graphs and a scatterplot to condition on one or two variables as detailed below. Slicing2d implements, modifies and extends a graph semantic proposed by Hurley and Oldford [44]. They propose to use a 3d transition graph with the edge semantic

Name	Description
none	no scaling
observation	every row of data gets scaled to mean 0 and variance 1
variable	every column of data gets scaled to mean 0 and variance 1
observation01	every row of data gets scaled to the range [0,1]
variable01	every column of data gets scaled to the range [0,1]
data01	the data as a whole gets scaled to the range [0,1]

Table 4.17: Scaling methods for context2d.

as follows: an edge from  $(X, Y)$  to  $(X, Z)$  is a conditioning on the common variable (i.e.  $X$  here) and moving a navigator on that edge displays a scatterplot of  $Y$  vs.  $Z$  for the subset of the data points that lie in some neighborhood of  $x$ , where  $x$  moves with the navigator position from the minimum observed value of  $X$  to the maximum value. We modify and generalize this semantic to include a conditioning for 4d transitions as well.

Our slicing semantic is as follows. Assume we are interested in displaying the data with a scatterplot. If the navigator is on a 2d space node (i.e. representing two variables) then the scatterplot shows these two variables. If the navigator is on a 3d transition edge, for example, from  $(A, B)$  to  $(B, C)$ , then the scatterplot shows  $B$  vs.  $A$  for a subset of data points that are conditioned on  $C$ . Note that we condition on the variable that is not shared between the two nodes whereas Hurley and Oldford [44] condition on the common variable; this allows us to generalize our semantic to 4d transitions, as discussed below. The conditioning on  $C$  is the same as the one discussed above for  $X$ . We now introduce some notation to simplify the explanations that follow. For a variable  $C$  let  $C(p)$ , where  $p \in [0, 1]$ , represent the value that lies  $100 \cdot p$  percent between the minimum and maximum of  $C$ ; that is,  $C(p) = \min(C) + p \cdot (\max(C) - \min(C))$  for  $p \in [0, 1]$ . Then, the 3d transition from  $(A, B)$  to  $(B, C)$  plots  $B$  vs.  $A$  for all points with  $C$  in a neighborhood of  $C(p)$ , where  $p = nav_p$  is the proportion of the navigator location along the edge to the total length of the edge. One possible neighborhood could be defined as  $[C(p) - d, C(p) + d]$  for some  $d > 0$ . Another possible neighborhood could be defined as the  $k$  nearest neighbors of  $C(p)$  in the

$C$  variable. If the navigator is on a 4d edge, for example, from  $(A,B)$  to  $(C,D)$ , then the scatterplot shows  $B$  vs.  $A$  for a subset of data points that are conditioned on  $C$  and  $D$ . We propose three different methods for conditioning with  $C$ ,  $D$  and the navigator proportion  $nav_p$

- *Union conditioning*: we use data points for which  $C$  lies in a neighborhood of  $C(nav_p)$  or  $D$  lies in a neighborhood of  $D(nav_p)$ .
- *Intersection conditioning*: we use data points for which  $C$  lies in a neighborhood of  $C(nav_p)$  and  $D$  lies in a neighborhood of  $D(nav_p)$ .
- *Sequential conditioning*: if  $nav_p < 0.5$  then we use data points for which  $C$  lies in a neighborhood of  $C(nav_p * 2)$ . If  $nav_p \geq 0.5$  then we use data points for which  $D$  lies in a neighborhood of  $C((nav_p - 0.5) * 2)$ . Note that the fact that the first interval for  $nav_p$  is an open interval (i.e.  $[0,0.5)$ ) and the second one is a closed interval (i.e.  $[0.5,1]$ ) is not an issue in practice as long as the neighborhood is chosen large enough.

We implement this slicing semantic with the `slicing2d` context as follows. We use a scatterplot of the conditioning variables to visualize how the points were sliced. That is, we use the selected state of the scatterplot of the conditioning variables to select the subset of data points displayed in the plot that visualizes the sliced data. If only one variable is used for conditioning then we use random numbers generated from the `uniform(0,1)` distribution for the second axis. With no conditioning variables (i.e. the navigator is located on a node) we use random numbers for both axes. Moving the navigator on a graph updates the conditioning and the “sliced” scatterplot accordingly. Then the conditioning neighborhood is visualized with rectangles, see [Figure 4.26](#) and [Figure 4.27](#). The setting shown in [Figure 4.26](#) and [Figure 4.27](#) was created with the following code

```
oa <- oliveAcids
names(oa) <- c('p','p1','s','o','l','l1','a','e')
nodes <- apply(combn(c('s','o','a','e'),2),2, function(x)paste(x, collapse=':'))

G <- ndtransitiongraph(nodes=nodes, n=c(3,4))
g <- l_graph(G)
nav <- l_navigator_add(g)
```

```

con <- l_context_add_slicing2d(nav, data=oa)
attr(con, 'plot_xy')['color'] <- Area
attr(con, 'plot_uv')['color'] <- Area

```

The conditioning variable plot is called 'plot\_uv' and the plot with the sliced data is called 'plot\_xy'. Their respective handles are attached to the context handle as an attribute. The slicing2d context has the proportion state that defines the neighborhood of a conditioning as a proportion of the range of that variable. The conditioning4d specifies the conditioning method with 4d edge transitions and has to be either 'union', 'intersection' or 'sequential'. If the analyst interactively selects points in the conditioning variable plot 'plot\_uv' (e.g. with sweeping or brushing) then the rectangles showing the conditioning neighborhood are hidden until the navigator changes its location and updates the conditioning scatterplot accordingly.

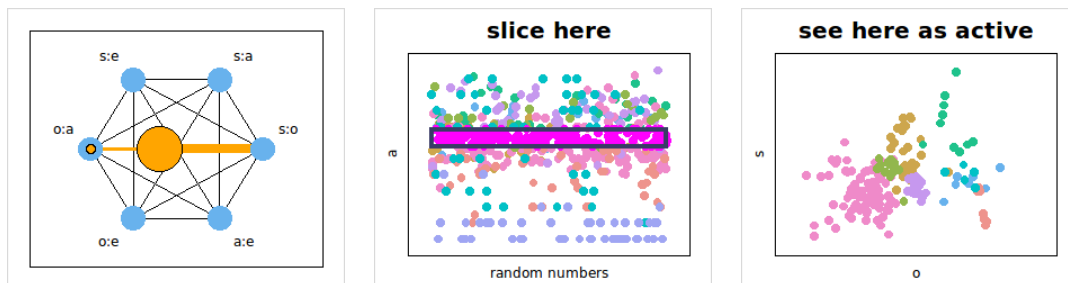
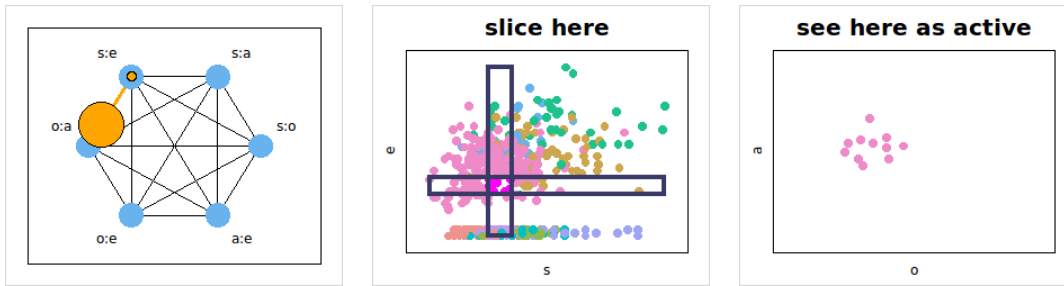
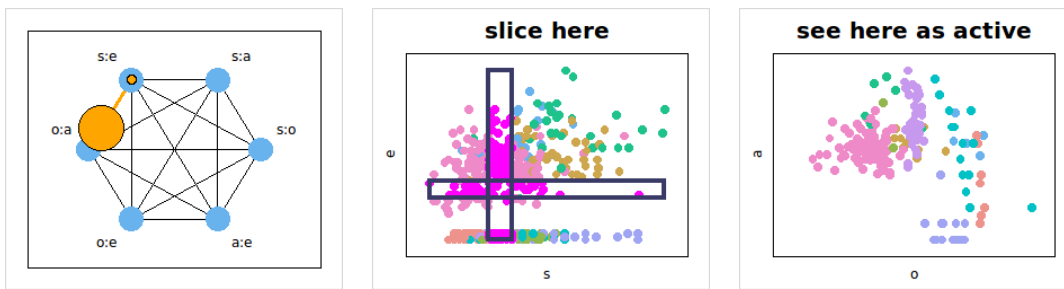


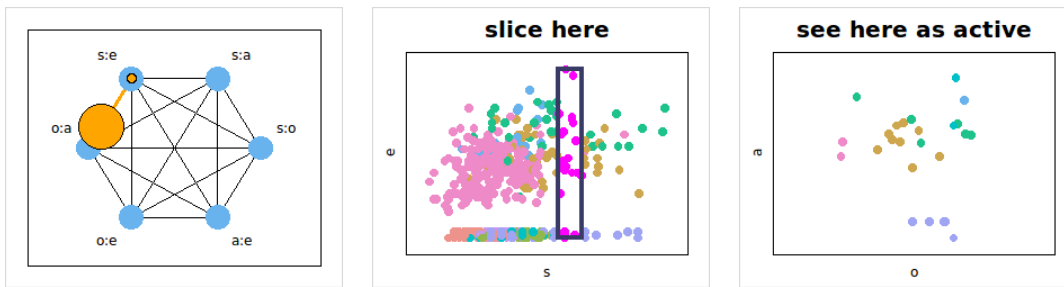
Figure 4.26: loon's slicing2d context setting for 3d transitions.



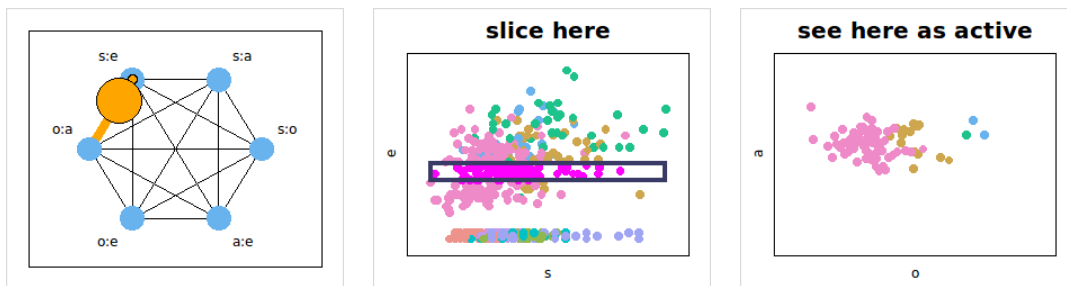
(a) 4d transition with intersection conditioning



(b) 4d transition with union conditioning



(c) 4d transition with sequential conditioning:  $nav_p = 0.3$



(d) 4d transition with sequential conditioning:  $nav_p = 0.7$

Figure 4.27: loon's slicing2d context setting for 4d transitions.

# Chapter 5

## Advanced Loon Framework

### 5.1 Implementation

loon is implemented with Tcl and Tk. Tcl is a dynamic scripting language and Tk is a cross-platform widget toolkit for Tcl. loon can be used with any programming language that has bindings with Tcl and Tk. There are a number of programming languages that provide such bindings including R, Python, Perl and Ruby. In these languages, it is possible to access a Tcl interpreter that can also evaluate code in the host language with callbacks. This is the functionality needed to use or to embed loon with a host programming language.

We have embedded loon into R with the R package loon as a working example of adapting the Tcl's loon to a new programming environment. The R binding with Tcl and Tk is provided with the tcltk package originally developed by Dalgaard [24] which is now part of the core R distribution [60]. In this thesis, we explain loon's framework with the R syntax only. A comparison of loon's syntax in R and Tcl can be found in loon's web documentation (with `l_help()`). As a general rule, we chose the names for the R functions based on loon's Tcl application programming interface (API). For example, the following code creates the same loon session once in R and once in Tcl:

in R

```
p <- l_plot(x=c(1,3), y=c(3,2))
l_configure(p, color='red')
l_scaletto_world(p)
```

in Tcl

```
set p [plot -x {1 3} -y {3 2}]
$p configure -color red
$p scaletto world
```

In Tcl, the plot handle `p` refers to an object and `configure` and `scaletto` are methods of `p`. loon's R function names usually have a `l_` prefix to avoid masking other R functions (e.g. `plot` and `hist`) and to unify loon's R API. Functions without the `l_` prefix are not graphical user interface related (e.g. `loongraph` and `completograph`).

Graphical user interface (GUI) elements in Tk are called widgets. loon's displays and inspectors are Tk megawidgets; that is, they are compositions of standard Tk widgets that are packed into a Tk frame widget. When designing the loon (mega)widget library we followed the Tk conventions as is suggested by Flynt [28, sec. 14.2]. Hence, people familiar with Tk should feel familiar with loon's widgets – we took the liberty to call Tk's *configurable options* states in loon. loon is designed to be a toolkit; loon's widgets can be used, customized and integrated in new GUIs the same way the ordinary Tk widgets can be used to create new GUIs. For example, it would be possible to use loon's widgets and the standard Tk widgets to re-create environments like Mondrian, GGobi or Conditional Choropleth Maps without modifying loon's source – something that is not possible the other way around.

With the loon R package, we sometimes deviate from the tcltk R package conventions to simplify certain tasks with loon. For example, we provide the `l_configure` and `l_cget` functions that should be used instead of the `tkconfigure` and `tkcget` functions from the tcltk package. One reason to do so is that the tcltk functions do not support the conversion of R's `data.frame` and `list` data structures to a data structure in Tcl. Another reason is that `l_configure` and `l_cget` can be used with all of loon's objects that have states such as custom point glyphs, layers, navigators and contexts. Another deviation from the conventions used in the tcltk R package is that widget handles for loon widgets are of class `loon` and not of class `tkwin` as for the normal Tk widgets. We do this for sake of simplicity: objects of class `tkwin` involve an environment whereas objects of class `loon` do not. A loon widget path handle in R is a string with the *widget path name* that has a 'loon' class attribute. In Tk every widget has a unique widget path name. For



the histogram, scatterplot, serialaxes and graph displays the widget path name can be found in the window decoration as long as these widgets are created without specifying a parent widget explicitly (as discussed in [Section 5.4](#)). For example, the scatterplot in [Figure 4.17\(b\)](#) shows “loon scatterplot .l0.plot” as its window title; here, '.l0.plot' is widget path name of the display.

loon object handles are returned by the object creator functions such as `l_plot`, `l_glyph_add_text` and `l_layer_polygons`. It is possible to re-create any loon object handle in an R session. That is, handles for displays, layers, glyphs, navigators and context handles can be created with the widget path name and the appropriate ids. The object handle can then be used with the methods '[' and '[<-' to access and modify states. For example, for a display with the widget path name '.l1.hist' one can create a loon plot handle as follows

```
h <- '.l1.hist'
class(h) <- 'loon'
```

For a layer with the layer id 'layer23' of that display the layer handle can be created as follows

```
l <- 'layer23'
class(l) <- c('loon', 'l_layer')
attr(l, 'widget') <- '.l1.hist'
```

For a context with the id 'context0' of a navigator with id 'navigator1' of a graph with widget path name '.l4.graph', the context handle is created as follows:

```
con <- 'context0'
class(con) <- c('loon', 'l_context')
attr(con, 'widget') <- '.l4.graph'
attr(con, 'navigator') <- 'navigator1'
```

and so on. The `l_create_handle` function can be used to create the loon object handles from a vector of the widget path name and the object ids (in the order of the parent-child relationships). For example, the above three object handles can also be created with

```
h <- l_create_handle('.l1.hist')
l <- l_create_handle(c('.l1.hist', 'layer23'))
con <- l_create_handle(c('.l4.graph', 'navigator1', 'context0'))
```

The `l_cget` and `l_configure` have `target` as their first argument which either accepts a loon object handle or a vector with the widget path name and the object ids as used for `l_create_handle`. For example

```
l_configure(h, color='red')
```

is equivalent to

```
l_configure('.l1.hist', color='red')
```

and

```
l_configure(con, command='')
```

is equivalent to

```
l_configure(c('.l4.graph', 'navigator1', 'context0'), command='')
```

The re-creation of object handles is useful when, for example, an object handle is lost or overwritten.

## 5.2 Event Bindings

Event bindings provide the functionality of binding code to specific event types. The bound code is called a callback. In loon, we distinguish between four classes of events: state change events, item events, canvas events and content events. Examples of each type of event (in the same order as mentioned before) include: a selected state modification of a plot, moving the mouse cursor over a point glyph, re-sizing the plot window and adding a layer.

The callback code gets evaluated by the Tcl interpreter, hence R users of loon need to be aware of R function callbacks in order to use loon's binding functionality. Before the callback gets evaluated loon substitutes certain expressions in the callback code with relevant information. For example, `%W` is substituted with the widget path name of the widget that causes the callback evaluation. The substitution expressions will be listed for every binding type.

Bindings are the driving force of interactivity. We provide the user with every binding type used for implementing loon's interactivity. There are two classes of bindings: system bindings and user bindings. System bindings are meant for developing new displays whereas user bindings are meant to be used by the analyst. Only user bindings are accessible through the R API. System bindings are always evaluated before user bindings and errors in the system binding callbacks are not caught with exception handling whereas errors in the callbacks of the user bindings are caught.

Every binding has a binding id that can be used to delete the binding or to reorder the callback evaluation when multiple bindings are signed up to a particular event. To query the binding ids for a particular binding type one should use the `l_bind_<type>_ids` function, where `<type>` is substituted with either `state`, `item`, `canvas`, `layer`, `glyph`, `navigator` or `context`. Except for `item` and `canvas` bindings, the order of the binding ids returned by `l_bind_<type>_ids` is also the evaluation order of the callback functions. To change the evaluation order one should use the `l_bind_<type>_reorder` function. For `item` and `canvas` bindings, the evaluation order is according to the order they were added. A particular binding can be deleted using the `l_bind_<type>_delete` function.

### 5.2.1 R function callbacks

The Tcl interpreter can call R functions via the Tcl command `R_Call` that takes as its first argument the hex-encoded address of an R function. Further arguments to the `R_call` command get passed to the R function as arguments of type character.

The R function `.Tcl.callback`, as defined in the `tcltk` R package, takes another R function as an argument and returns a Tcl script (i.e. string) that calls that R function. The formal arguments of the R function, except for the ellipsis (i.e. `...`), get a prefix `%` and then are added to the Tcl callback script. For example,

```
foo <- function(x,y,z) {
  print(paste(x,y,z, sep='|'))
}
.Tcl.callback(foo)

[1] "R_call 0x20cff20 %x %y %z"
```

The % arguments are meant to be substituted before the R\_call evaluation. The Tcl interpreter can then call the R function foo as follows

```
.Tcl('R_call 0x20cff20 Hello 2 World')
[1] "Hello|2|World"
<Tcl>
```

Note that, in the above code, foo is a variable that points to the function with the hex-encoded address 0x20cff20. Assigning a new function (with a different hex-encoded address) to foo will garbage-collect the function previously assigned to foo (if no other variable points to that function), and the above .Tcl call will cause an error. If the callback should evaluate whatever function is assigned to foo one can wrap the foo function call into another function as follows

```
.Tcl.callback(function(x,y,z){foo(x,y,z)})
```

Sometimes it is useful to pass arguments to the R function

```
.Tcl.callback(function(x,y){bar(x,y,add=TRUE)})
```

loon assigns a variable to every R callback function used in loon's bindings in order to prevent them from getting garbage-collected.

## 5.2.2 State Change Bindings

The callback of a state change binding is evaluated when certain states change, as specified at binding creation. loon's plot widgets, inspector widgets, plot layers, point glyphs, navigators and contexts support state change bindings. These objects implement the configuration pipeline as described in [Subsection 4.3.2](#) and illustrated in [Figure 4.8](#). In the following example we create a scatterplot with handle p, define a function foo that outputs the number of selected points in p on the prompt, and finally we add a state binding for a selected state change to p:

```
p <- l_plot(stearic~oleic, color=Area)
foo <- function() {
  cat(paste('Plot', p, 'has',
           sum(p['selected']), 'points selected.\n'))
}
bid <- l_bind_state(p, event='selected', callback=foo)
```

The return value of `l_bind_state` is a binding id that can be used to change the callback evaluation order or to delete the binding. The event argument of `l_bind_state` specifies which state changes in `p` trigger the binding's callback evaluation. If a vector of state names is passed to the event argument then the binding's callback is evaluated when any of the specified states is changed in `p`. The keyword `all` specifies that any state change in `p` should trigger the evaluation of the binding's callback. Changing the selected state in the example above will cause the binding `bid` to print a message on the R prompt as follows

```
p['selected'] <- Area=="West-Liguria"
Plot .l2.plot has 50 points selected.
```

State binding evaluations are the last step in the configuration pipeline, see [Figure 4.8](#). The `configure` method checks which states have been changed and then loops through its state change bindings and evaluates the callbacks that have at least one state in the events list that has been modified. Hence, when all points are already selected, evaluating `p['selected']<-TRUE` will not result in evaluating any callbacks of state change bindings for the event `selected`.

[Table 5.1](#) lists the callback substitutions of state change bindings. For R users, a `%W` substitution is achieved by defining the R callback function with an argument named `W`. The callback function evaluation then passes the substituted information as arguments of type string to the R callback function. Some expressions such as `%e` are substituted by (Tcl) lists. In the R callback function this list arrives as a single string with the list elements separated by spaces. We provide the `l_toR` function to convert substituted lists to R vectors.

String	Description
<code>%W</code>	widget path name
<code>%e</code>	list of state changes of the configure call
<code>%b</code>	binding id.

Table 5.1: State change binding substitutions.

The following example demonstrates callback substitutions.

```
1 h <- l_hist(x=palmitoleic, color=Area)
2 bar <- function(W, e) {
3   class(W) <- 'loon'
4   events <- l_toR(e)
5   cat(paste('The states', paste0(events, collapse=', ') ,
6     'of the plot', W, '\nhave changed. Now',
7     sum(W['selected']), 'points are selected.\n'))
8 }
9 bid <- l_bind_state(h, event='all', callback=bar)
10 l_configure(h, showScales=TRUE, swapAxes=TRUE, selected=TRUE)
```

The states showScales, swapAxes, selected of the plot .l3.hist have changed. Now 572 points are selected.

In this example, we scoped for the plot handle p in the function foo. Here, we substitute the widget path name and, on line 3, we assign the class loon to widget path in order to query and modify the states with the '[' and '['<-' generics (see line 7). The events that were responsible for the callback evaluation are stored as a single string in e and, on line 4, we split them into an R character vector of event names using l\_toR.

Finally, aside from the events that are named according to their corresponding state names, there is also the destroy event that occurs when the display gets destroyed, for example, by closing the window.

### 5.2.3 Item Bindings

Item bindings are used for evaluating callbacks at certain mouse and/or keyboard gestures events (i.e. X events) on visual items on the canvas. Items on the canvas can have tags and item bindings are specified to be evaluated at certain X events for items with specific tags. Item bindings are Tk canvas bindings with one level of indirection in order to support loon's context specific substitutions in [Table 5.2](#) instead of the standard Tk substitutions. Therefore, the X events specifiers and valid tag expressions for item bindings are documented on the Tk canvas manual page [73]. loon's histogram, scatterplot and graph displays presently support item bindings. The latter three displays have an *n*-dimensional state tag to give each visual a canvas tag. In addition, layers also have a tag state. Tags have to be non-numerical.

String	Description
%W	widget path name
%b	binding id.
%x	$x$ coordinate of mouse at event
%y	$y$ coordinate of mouse at event

Table 5.2: Item binding substitutions.

In the following example, we create an item binding that prints out the index of a point when pressing with the left mouse button on a point glyph

```

1 p <- l_plot(stearic~oleic, color=Area)
2 foo <- function(W) {
3   i <- l_currentindex(W)
4   cat(paste('pressed on point', i, 'which is currently colored',
5           l_cget(W, 'color')[i], '\n'))
6 }
7 l_bind_item(p, tags='layer&&model',
8           event='<ButtonPress-1>', callback=foo)

```

For example, when pressing on point 298 on the scatterplot display the following message is printed in R:

```
pressed on point 298 which is currently colored #EFEF8A8AC9C9
```

The `l_currentindex` function, as used on line 3, checks if there is a visual item below the mouse cursor and if there is, it returns the index of the visual item's position in the corresponding variable dimension of its layer. That is, in the above example,  $i$  represents the element index of the point in the  $n$ -dimensional states of `p`. The `tags` argument on line 7 accepts logical expressions of tags using the operators `&&`, `||`, `^` and `!` (i.e. logical and, or, xor and negation), and also parenthesized subexpressions. Therefore, for the item binding on line 7, the callback is evaluated at a left button press event on visual items that have both 'model' and 'layer' tags. We also provide the `l_currenttags` function to retrieve the tags of the visual item that at the time of the function evaluation is below the mouse cursor. The tagging scheme of loon's visual items is outlined in [Table 5.3](#).

Visual Item	Tags
Histogram model layer	layer, model, bin<id>, <bin quantifier>
Scatterplot model layer	layer, model, point, item<i-1>, <tag state>[i]
Graph model layer	
nodes	layer, model, point, item<i-1>, <tag state>[i]
edges	layer, model, edge, item<i-1>, <from node>, <to node>
orbits	layer, model, orbit, item<i-1>
navigators	layer, model, navigator, <navigator id>
navigator path from	layer, model, navigatorProgression, <navigator id>
navigator path to	layer, model, navigatorEdge, <navigator id>
navigator path end	layer, model, navigatorPathEnd, <navigator id>
Non-Model layers	
polygon, text, line, rectangle, oval	layer, <layer id>, <type>, item0, <tag state>
points, texts, poly- gons, rectangles, lines	layer, <layer id>, <type>, item<i-1>, <tag state>[i]
x y labels	loon, label, <x y>label
title	loon, label, title
scales	
x y scale label	loon, scale, <x y>scale, label
x y scale tick	loon, scale, <x y>scale, tick
x y scale guidelines	loon, scale, <x y>scale, guideline
clipping borders	loon, border

Table 5.3: Item tags for visuals for plots based on the main graphics model.



It is possible to query the tags of visual items on a display interactively as follows

```
p <- l_plot(oleic~stearic, color=Area)
l_bind_item(p, 'all', '<ButtonPress-1>',
           function(W)print(l_currenttags(W)))
```

When working with the tag state of a plot or layer, it is important to omit any collision with the tags that are used by loon. Therefore, we suggest adding an underscore as a prefix for each user-defined tag.

Note that the interior of visuals with no fill (such as polygons and rectangles with the fill color ' '), although it appears transparent, it still absorbs X events. This is a Tk canvas limitation and a possible solution to work around it is to draw those visuals with lines instead of shapes that have no fill.

## 5.2.4 Canvas Bindings

Canvas bindings are used to evaluate callbacks at certain X events on the canvas widget. Such X events include re-sizing of the canvas and entering the canvas with the mouse. Canvas bindings are Tk bindings for the canvas widget with one level of indirection to support loon's context-specific substitutions seen in [Table 5.4](#) instead of the default Tk substitutions. Valid event specifiers can be looked up on the Tk bind manual [\[72\]](#). The canvas bindings are supported by the histogram, scatterplot, graph and serialaxes displays.

String	Description
%W	widget path name
%b	binding id.
%x	<i>x</i> coordinate of mouse at event
%y	<i>y</i> coordinate of mouse at event
%w	plot width in pixels
%h	plot height in pixels

Table 5.4: Canvas binding substitutions.

The following example prints the size of the plot canvas (i.e. Tk canvas) to the R prompt

```
h <- l_hist(x=palmitoleic, color=Area)
foo <- function(W, w, h) {
  cat(paste('The area of', W, 'is',
           l_toR(w, as.numeric)*l_toR(h, as.numeric),
           'pixel^2\n'))
}
l_bind_canvas(h, event='<Configure>', callback=foo)
# resize window using the mouse or
# via the command line as follows
l_resize(h, 300, 500)
```

The area of .l0.hist is 150000 pixel^2

## 5.2.5 Content Bindings

The functions `l_bind_layer`, `l_bind_glyph` and `l_bind_navigator` create bindings that evaluate callbacks whenever the collection of layers, glyphs or navigators, respectively, change for a display. The `l_bind_context` function adds a context binding to a navigator to have the callback evaluated when a context gets added, deleted or relabeled for a navigator. We collectively call these bindings content bindings. The basic substitutions of content bindings are listed in [Table 5.5](#). In addition to the basic substitutions, loon also provides substitutions to obtain the id of the element that is responsible for the callback evaluation. That is, layer bindings substitute `%l` with the layer id, the glyph bindings substitute `%g` with the glyph id, the navigator bindings substitute `%nav` with the navigator id, and the context bindings substitute `%con` with the context id. [Table 5.6](#) lists all possible events of a particular binding type.

String	Description
<code>%W</code>	widget path name
<code>%e</code>	event that cause callback evaluation
<code>%b</code>	binding id

Table 5.5: Content binding substitutions.

Binding Type	Events
layer	add, delete, move, hide, show, relabel
glyph	add, delete, relabel
navigator	add, delete, relabel
context	add, delete, relabel

Table 5.6: Overview binding events.

The following example adds a layer binding to a scatterplot and then layers the contour lines of a density estimate on the display

```
p <- l_plot(stearic~oleic, color=Area)
foo <- function(W,e,l) {
  cat(paste0('Plot ', W, ' had event "',
            e, '" for layer ', l, '\n'))
}
b_id <- l_bind_layer(p, event='all', callback=foo)

library(MASS)
lc <- l_layer_contourLines(p, MASS::kde2d(oleic,stearic))

Plot .l0.plot had event "add" for layer layer0
```

## 5.3 Custom Linking

In the case where the standard linking model described in [Section 4.5](#) is not flexible enough for a particular situation, one can use state change bindings to implement custom linking rules. For example, it is not possible to perform any of the following linking setups with the standard linking model: one-to-many linking, one directional linking of states, linking states with different names, linking points within a plot or linking model layers with non-model layers. In this section, we give an example for each of these linking setups and one that shows how to avoid circularities (i.e. infinite loops). To keep the code for these examples short, we only link the selected state of two displays if not mentioned otherwise. These examples illustrate the simplicity of adding a particular linking rule and of defining new linking mechanisms that are more general.

### 5.3.1 One Directional And One-To-Many Linking

In the following example, we create two scatterplots `pa` and `pb` and we link the selected state of `pa` with the selected state of `pb` using a state change binding. The first point of `pb` is selected when any of the first three points in `pa` are selected; otherwise the first point in `pb` is not selected. The second and the third point in `pb` take on the same selected state as the fifth point in `pa`. The selected state of the fourth point in `pb` is independent of the selected state of `pa`. Changing the selected state in `pa` updates the selected state in `pb`, but not the other way around; therefore, this example illustrates a case of one directional, one-to-many, and many-to-one linking.

```
local({
  pa <- l_plot(x=1:5, y=1:5, title="One to Many: plot A")
  pb <- l_plot(x=1:4, y=1:4, title="One to Many: plot B")

  pa2pb <- function() {
    sa <- pa['selected']
    sb <- pb['selected']
    pb['selected'] <- c(any(sa[1:3]), sa[5], sa[5], sb[4])
  }

  l_bind_state(pa, 'selected', pa2pb)
})
```

The callback function `pa2pb` scopes for the plot handles `pa` and `pb` in its parent environment. Therefore, one has to make sure that the variables `pa` and `pb` are not overwritten or erased within the life-time of these plots. In the above example, this is not an issue as we evaluated the code in a local environment. However, if the plot variables are stored in the global environment where overwriting the variables `pa` and `pb` could easily happen, then we recommend wrapping the callback function in another function such that the local environment of the function call keeps a copy of the plot handles.

```
linkSelected <- function(pa, pb) {
  force(pb)
  pa2pb <- function() {
    sa <- pa['selected']
    sb <- pb['selected']
    pb['selected'] <- c(any(sa[1:3]), sa[5], sa[5], sb[4])
  }
  l_bind_state(pa, 'selected', pa2pb)
}

plotA <- l_plot(x=1:5, y=1:5, title="One to Many: plot A")
plotB <- l_plot(x=1:4, y=1:4, title="One to Many: plot B")
linkSelected(plotA, plotB)
```

Now, the variables `plotA` and `plotB` could be deleted in the global environment and the linking would still work. Also, this allows the custom linking rule to be applied to different plots. Note that the environment of the first function call is not garbage-collected as the `'l_bind_state'` function keeps a variable link to the `'pa2pb'` function.

A real world example of one-to-many linking can be found in the `l_us_and_them_choropleth` demo that comes with the R `loon` package. There, we highlight magenta the polygons of a country whenever the corresponding point (to that country) is selected in a different scatterplot. For example, selecting the point corresponding to the USA highlights all polygons that represent the USA, see [Figure 5.1](#). To run the demo use

```
demo('l_us_and_them_choropleth')
```

### 5.3.2 Linking States with Different Names

The following example creates two plots `pa` and `pb` and links the selected state of `pa` to the active state of `pb` bidirectionally. That is, changing the selected state of `pa` updates the active state of `pb` and the other way around.

```
linkSelectedActive <- function(plotA, plotB) {
  select2active <- function(W) {
    if (W == plotA) {
      plotB['active'] <- plotA['selected']
    } else {
      plotA['selected'] <- plotB['active']
    }
  }
  c(l_bind_state(plotA, 'selected', select2active),
    l_bind_state(plotB, 'active', select2active))
}

pb <- l_plot(olive[,4:5], title="Different State Names: Selected")
pa <- l_plot(olive[,6:7], title="Different State Names: Active")
linkSelectedActive(pa, pb)
```

For the above example, we now change the selected state of `pa`

```
pa['selected'] <- sample(c(TRUE,FALSE), size=pa['n'], replace=TRUE)
```

Then, the state change binding of `pa` evaluates the `select2active` function which in turn modifies the active state of `pb`. Next, the state change binding of `pb` evaluates the `select2active` which will evaluate the configure call to `pa` to change the selected state. Since the selected state does not change with this configure call, the state change binding of `pa` will not be evaluated and that will close the linking cycle. The last configure call is unnecessary and might slow down the interactions with the plots. We show in [section 5.3.4](#) how this unnecessary last configure call can be avoided with an additional variable (e.g. `inLinking`).

A more involved example of linking the selected state of a plot with the active state of another plot can be found in the `l_selectToActive` demo that comes with the R `loon` package.

### 5.3.3 Linking Items Within a Plot

The following example creates a scatterplot `p` and links the selected state for the first three points such that if one of them is selected all three points are selected. One of the cases where this linking mechanism is useful is when selecting a polygon of a country on a map and have all the polygons associated to that country selected as in [Figure 5.1\(b\)](#).

```
linkPoints <- function(w) {  
  foo <- function() {  
    l_configure(w,  
      selected=rep(any(w['selected'] [1:3]),3), which=1:3)  
  }  
  l_bind_state(w, 'selected', foo)  
}  
  
p <- l_plot(1:4, 1:4)  
linkPoints(p)
```

### 5.3.4 Avoiding Circularity

Assume the following linking scenario: we have three plots `pa`, `pb` and `pc` and when we interactively select points in one of them the other two plots show the inverted selected state of that plot. That is, if a point is selected in the first plot it is not selected the other two; on the other hand, if a point is not selected in the first plot then it is selected in the other two. Without special care, implementing this linking scenario results in circularity (i.e. infinite loop) when using state change bindings for the selected state of each plot. In order to avoid circularity it is necessary to use a “busy“ variable (e.g. `inLinking` here). The following code implements the linking scenario of the selected state given above.

```
linkNegate <- function(plotA, plotB, plotC) {
  inLinking <- FALSE
  foo <- function(W) {
    if (!inLinking) {
      inLinking <<- TRUE
      if (W == plotA) {
        plotB['selected'] <- !plotA['selected']
        plotC['selected'] <- !plotA['selected']
      } else if (W == plotB) {
        plotA['selected'] <- !plotB['selected']
        plotC['selected'] <- !plotB['selected']
      } else {
        plotA['selected'] <- !plotC['selected']
        plotB['selected'] <- !plotC['selected']
      }
      inLinking <<- FALSE
    }
  }
  c(l_bind_state(plotA, 'selected', foo),
    l_bind_state(plotB, 'selected', foo),
    l_bind_state(plotC, 'selected', foo))
}

pa <- l_plot(olive[,4:5])
pb <- l_plot(olive[,5:6])
pc <- l_plot(olive[,6:7])

linkNegate(pa, pb, pc)
```

Note the use of the argument `W` in the callback function `foo`. `W` will be substituted with the widget path name of the plot that evaluates the function `foo` in its state change binding.



### 5.3.5 Linking Model with Non-Model Layers

Layers do not have a selected state; hence, by default, it is not possible to “select” them with the mouse cursor and have them highlighted magenta. However, it takes little effort to interactively select a layer item and have it highlighted magenta. This example will create an empty plot, layer a heat image and make the rectangles “selectable” with left-click and `Shift`-left click for multiple or toggle selection. That is, `Shift`-clicking on a rectangle that is already highlighted will toggle the color of the rectangle back to the original one.

```
layerSelect <- function(layer, hcol='magenta') {
  col <- layer['color'] # cache color
  widget <- attr(layer, 'widget')
  foo <- function(add=FALSE) {
    i <- l_currentindex(widget)
    if (i == -1) return()
    if (add) {
      if (layer['color'][i] == l_hexcolor(hcol)) {
        l_configure(layer, color=col[i], which=i)
      } else {
        l_configure(layer, color=hcol, which=i)
      }
    } else {
      l_configure(layer, color=replace(col, i, hcol))
    }
  }
  tag <- paste0('layer&&', layer)
  c(l_bind_item(widget, tag, '<ButtonPress-1>',
    function()foo(FALSE)),
    l_bind_item(widget, tag, '<Shift-ButtonPress-1>',
    function()foo(TRUE)))
}

p <- l_plot()
mat <- matrix(gray(seq(0,1, length.out = 24)), ncol=4)
l <- l_layer_rasterImage(p, mat, 0, 0, 1, 1)
layerSelect(l)
```

A more involved example of linking model with non-model layers is the `'l_us_and_them_choropleth'` demo in the `loon` R package as, discussed in [Subsection 5.3.2](#) and shown in [Figure 5.1](#). There, the polygons in [Figure 5.1\(b\)](#) can be “selected” as described above.

## 5.4 Geometry Management

When a loon widget is created it is placed into a new window by default. However, it is possible to use one of Tk's geometry manager (i.e. `place`, `grid` and `pack`) to manually position a loon widget as with a standard Tk widget. All of loon's widget creator functions in R (e.g. for plots and inspectors) have an argument named `parent` that accepts a valid Tk widget as a parent object (e.g. a toplevel window or a Tk frame). The default parent argument is `NULL` which creates a new toplevel window with the loon widget packed into it.

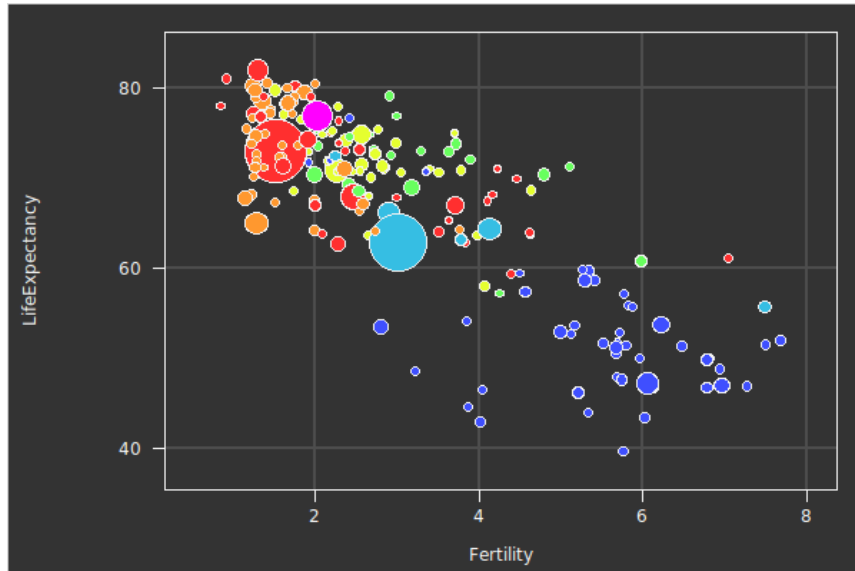
For the following example, a histogram widget is packed next to a histogram inspector, see [Figure 5.2](#).

```
tt <- tktoplevel()
tktitle(tt) <- 'Histogram Layout Example'
h <- l_hist(parent=tt, x=olive$oleic, useLoonInspector=FALSE)
hi <- l_hist_inspector(parent=tt, activewidget=h)
tkpack(hi, side='right', fill='y')
tkpack(h, side='right', fill='both', expand=TRUE)
```

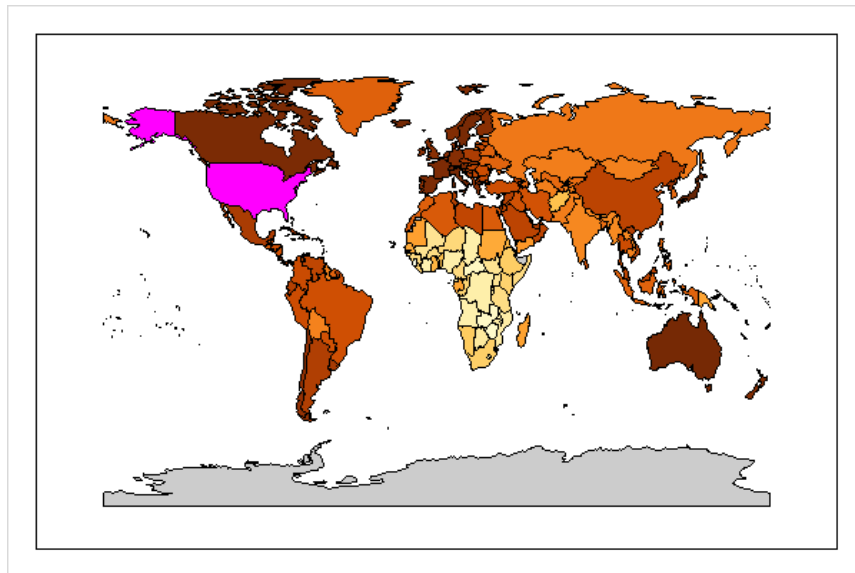
With the grid geometry manager it is possible to create a scatterplot matrix using loon's scatterplot displays. We do this in the `l_pairs` function. For example, the following code creates the scatterplot shown in [Figure 5.3](#).

```
p <- l_pairs(oliveAcids, color=Area)
```

The `l_pairs` function also adds a state binding to every scatterplot such that, when we zoom and pan in one scatterplot, the displayed  $x$  and  $y$  ranges in the plot areas of the scatterplots on the same row and column of the scatterplot matrix will be the same.



(a) scatterplot of life expectancy vs. fertility



(b) Choropleth map

Figure 5.1: Life expectancy (in years) vs. fertility (number of children per women) for different countries in 2002. The data is from the Gapminder data project [35]. The choropleth plot (right panel) encodes life expectancy as color.

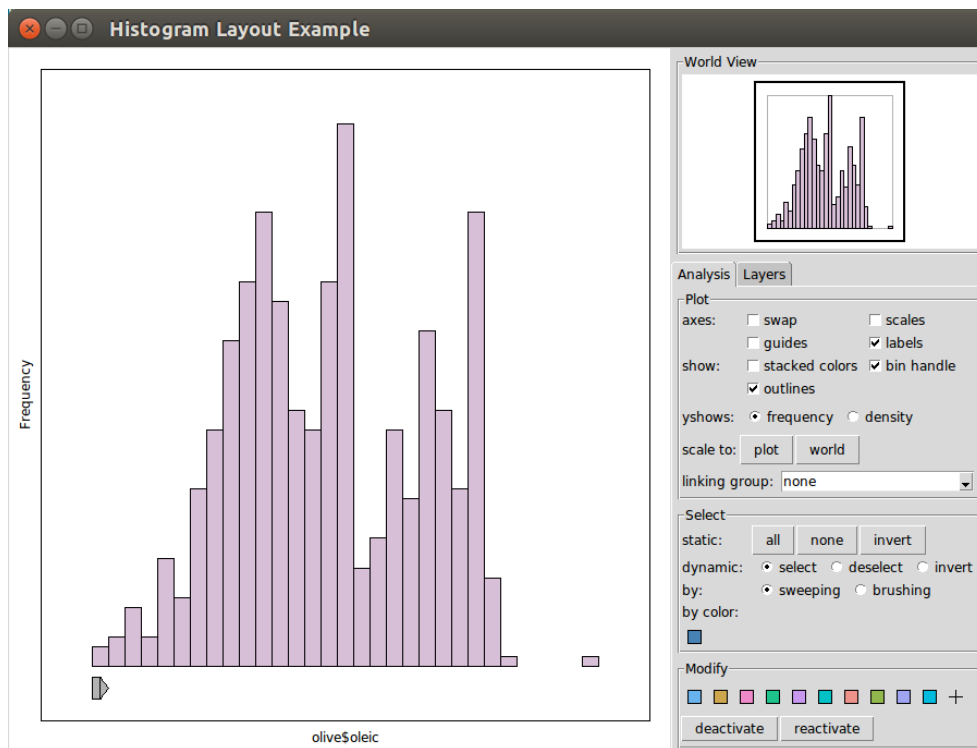


Figure 5.2: Geometry management.

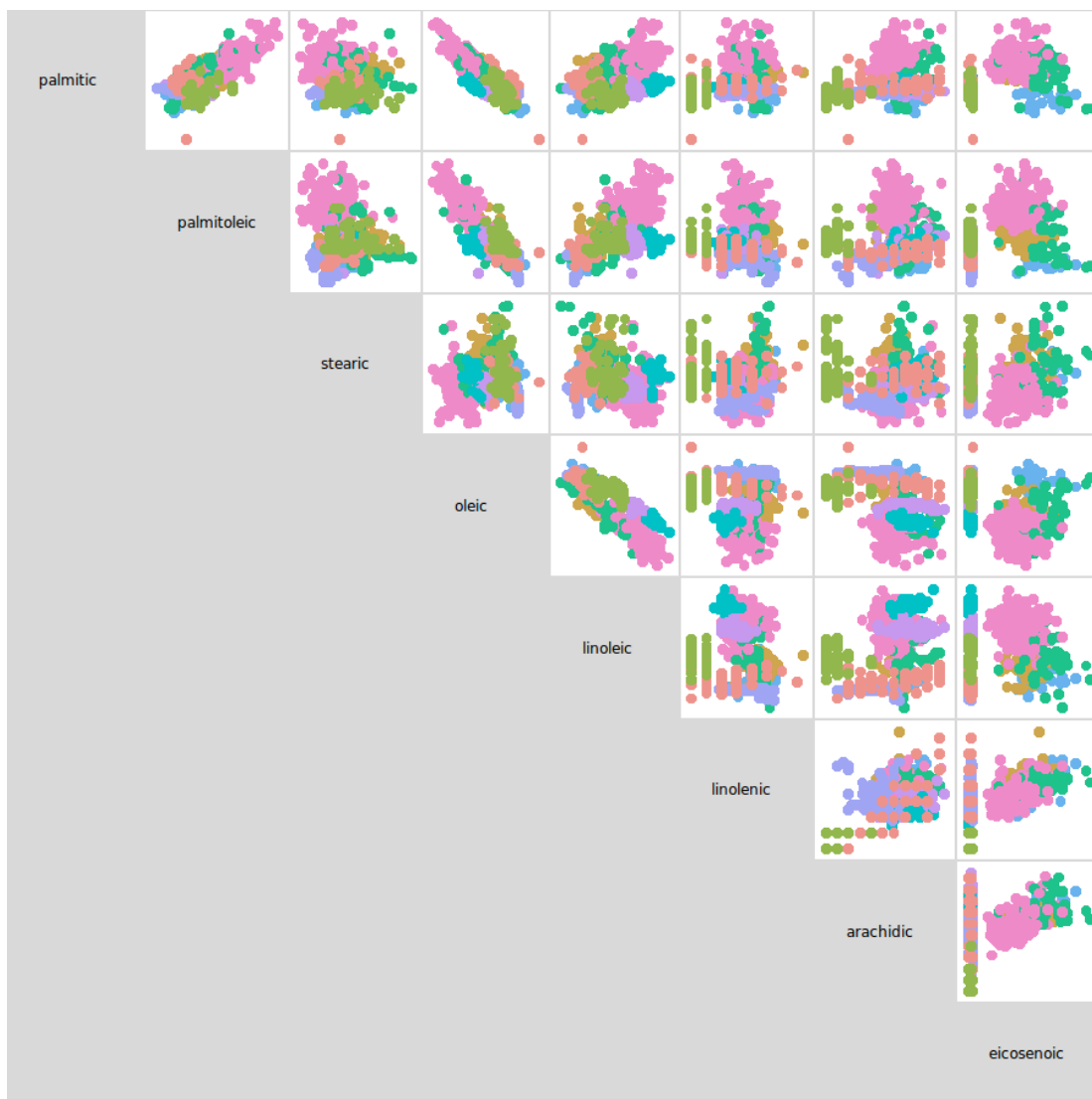


Figure 5.3: Scatterplot matrix using grid geometry manager.

## 5.5 Writing an Inspector

The inspectors provided with loon may not cover all the actions that are useful for a particular analysis. In this section, we show how to create a custom inspector for a display. That is, we create a new graphical user interface and have its widgets modify the states of a display and also show information about that display.

In the following example, we define a function `aspectRatioInspector` which creates a custom inspector for displaying and modifying the aspect ratio of a plot that is passed as an argument to this function.

```
1 aspectRatioInspector <- function(p) {
2
3     curAspect <- tclVar('1')
4     entryVal <- tclVar('1')
5
6     tt <- tkoplevel()
7     tktitle(tt) <- 'Aspect Ratio Inspector'
8     tkgrid(tklabel(tt, text=paste0('widget: ', p)),
9            sticky='w', padx=5, pady=5)
10    l <- tklabel(tt, textvariable=curAspect)
11    e <- tkentry(tt, textvariable=entryVal, width=5)
12    b <- tkbutton(tt, text='set', command=function()setAspect())
13    tkgrid(tklabel(tt, text='Current Aspect Ratio:'), l,
14           sticky='sw', padx=5, pady=5)
15    tkgrid(tklabel(tt, text='New Aspect Ratio:'), e, b,
16           sticky='w', padx=5, pady=5)
17
18
19    setAspect <- function() {
20        `l_aspect<-`(p, as.numeric(tclvalue(entryVal)))
21    }
22    getAspect <- function() {
23        tclvalue(curAspect) <- round(l_aspect(p), 3)
24    }
25    getAspect()
26
27    l_bind_canvas(p, '<Configure>', getAspect)
28    l_bind_state(p, 'all', getAspect)
29    tt
30 }
```

Lines 6 to 16 in the above code create the graphical user interface with standard Tk wid-

gets, see [Figure 5.4](#). Lines 19 to 21 define the function `setAspect` that takes the value from the entry box and updates the aspect ratio of the plot. Lines 22 to 24 define the `getAspect` function that updates the label on the inspector with the current aspect ratio of the plot. On line 27 to 29 we add the bindings necessary to keep the current aspect ratio label updated; the aspect ratio can change when the plot is resized or with some plot state changes (e.g. `zoomX`, `showLabels` and `labelMargins`). The aspect ratio inspector, as seen in [Figure 5.4](#), can be added to a histogram, scatterplot and graph widget as in the following example.

```
p <- l_plot(oleic~stearic, color=Area)
aspectRatioInspector(p)
```

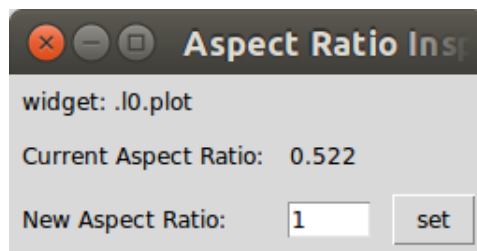


Figure 5.4: Custom inspector for aspect ratio.

## 5.6 Other Topics

### 5.6.1 Export as an Image

l<sub>oon</sub>'s plots can be exported as images with the `l_export` function. The supported image formats are dependent on the system environment. Plots can always be exported to the Postscript format. Exporting displays as `.pdfs` is only possible when the command line tool `epstopdf` is installed. Finally, exporting to either `png`, `jpg`, `bmp`, `tiff` or `gif` requires the `Img Tcl` extension. When choosing one of the formats that depend on the `Img` extension, it is possible to export any Tk widget as an image including inspectors.

When a plot is created with the default `parent=NULL` argument then the shortcut `CTRL-p` opens a dialog to export the plot as an image.

## 5.6.2 Animations

The Tk canvas is double buffered and therefore loon's plots are double buffered too. Creating an animation is possible by successively changing plot states or adding layers. For example, projecting points in 3 dimensions onto a plane that rotates from two variables to two other variable (i.e. a rigid rotation) can be done as follows.

```
X <- as.matrix(olive[,3:5])
p <- l_plot(X[,1], X[,2], color=Area, showLabels=FALSE)

for (alpha in seq(0, pi/2, length.out=60)) {
  A <- matrix(c(0,0,1,cos(alpha),sin(alpha),0),
              byrow=FALSE, ncol=2)
  coords <- t(A) %*% t(X)
  l_configure(p, x=coords[1,], y=coords[2,])
  l_scaletto_world(p)
  tcl('update', 'idletasks')
  Sys.sleep(0.01)
}
```

The Tk canvas does not refresh while loon is busy (or the Tcl interpreter in general). A `tcl('update', 'idletasks')` will force the canvas to refresh itself.

## 5.6.3 Color Mapping

There are two commonly used mapping schemes of data values to colors: one scheme maps numeric values to colors on a color gradient and the other maps nominal data to colors that can be well differentiated visually (e.g. to highlight the different groups). Presently, loon always uses the latter approach for its color mappings.

When assigning values to a display state of type color loon checks if these values are valid Tk color specifications. If one or more of the values are not valid then loon maps all the data values to colors from an internal list; that is, with loon's color mapping, every unique data value is assigned a different color. loon's default color list is composed of the first 11 colors from the *hcl* color wheel in [Figure 5.5](#). The letters in *hcl* stand for *hue*, *chroma* and *luminance*, and the *hcl* wheel is useful for finding “balanced colors” with the same chroma (radius) and luminance but with different hues (angles), see Ihaka [\[45\]](#).



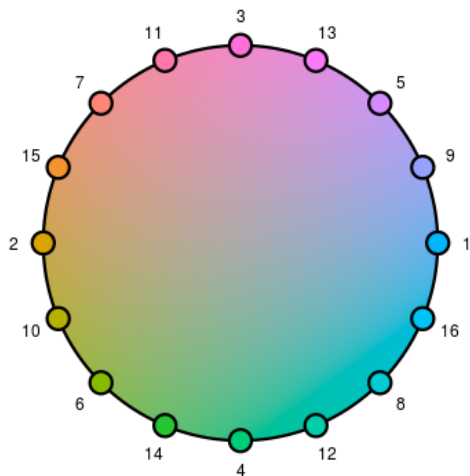
The colors in loon’s internal color list are also the default ones listed as the “modify color actions” in the analysis inspectors, see [Figure 4.5](#). To query and modify loon’s color list use `l_getColorList` and `l_setColorList`.

In the case where there are more unique data values than colors in loon’s color list then the colors for the mapping are taken from different locations distributed on the hcl color wheel as seen in [Figure 5.5](#). One of the advantages of using the hcl color wheel is that one can obtain any number of “balanced colors” with distinct hues. This is useful in encoding data with colors for a large number of groups; however, it should be noted that the more groups we have the closer the colors sampled from the wheel become and, therefore, the more similar in appearance.

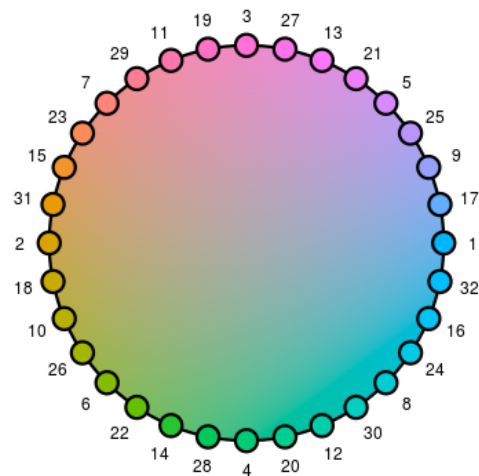
A common way to sample distinct “balanced colors” on the hcl wheel is to choose evenly spaced hues distributed on the wheel [\[45\]](#) (i.e. angles on the wheel). However, this approach leads to color sets where most colors change when the sample size (i.e. the number of sampled colors from the wheel) increases by one. For loon, it is desirable to have the first  $m$  colors of a color sample of size  $m + 1$  to be the same as the colors in a color sample of size  $m$ , for all positive natural numbers  $m$ . Hence, we prefer to have a sequence of colors. This way, the colors on the inspectors stay relevant (i.e. they match with the colors of the data points) when creating plots that encode with color a data variable with different number of groups. If there are more unique colors in the data points than there are on the inspectors then it is possible to add the next five colors in the sequence of the colors with the “+5” button. Alternatively, the “+” button on the modify color part of the analysis inspectors allows the user to pick any additional color with a color menu.

We implemented such a color sampling scheme (or color sequence generator) that also makes sure that neighboring colors in the sequence have different hues. [Figure 5.5](#) shows the color generating sequence twice, once for 16 colors and once for 32 colors.

When other color mappings of data values are required (e.g. numerical data to a color gradient) then the functions in the R package `scales` [\[82\]](#) provide various mappings including mappings for qualitative, diverging and sequential values. For example, the following code creates two plots with different color encoding of the point glyphs: once with loon’s default mapping for the olive Area variable and once by encoding the olive palmitic



(a) sampling 16 colors

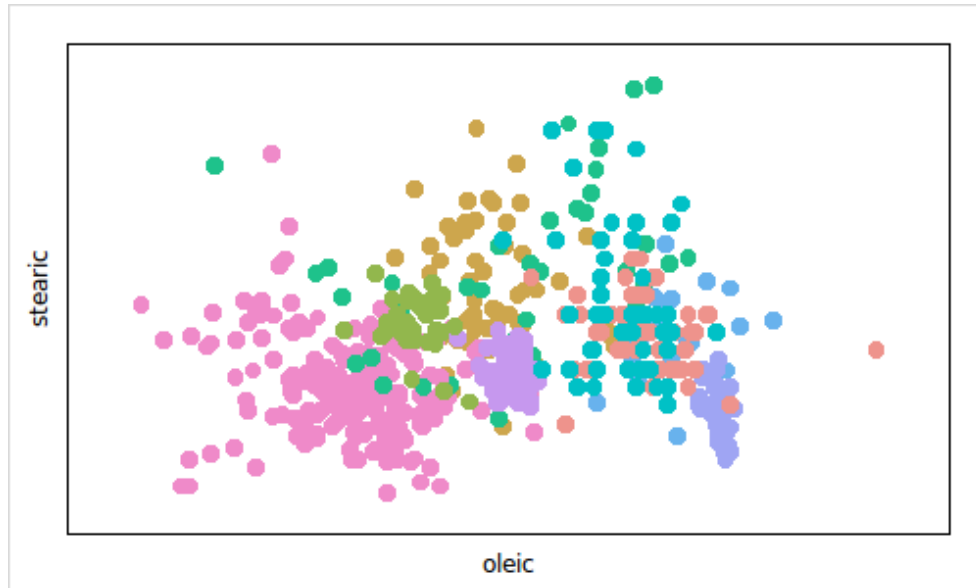


(b) sampling 32 colors

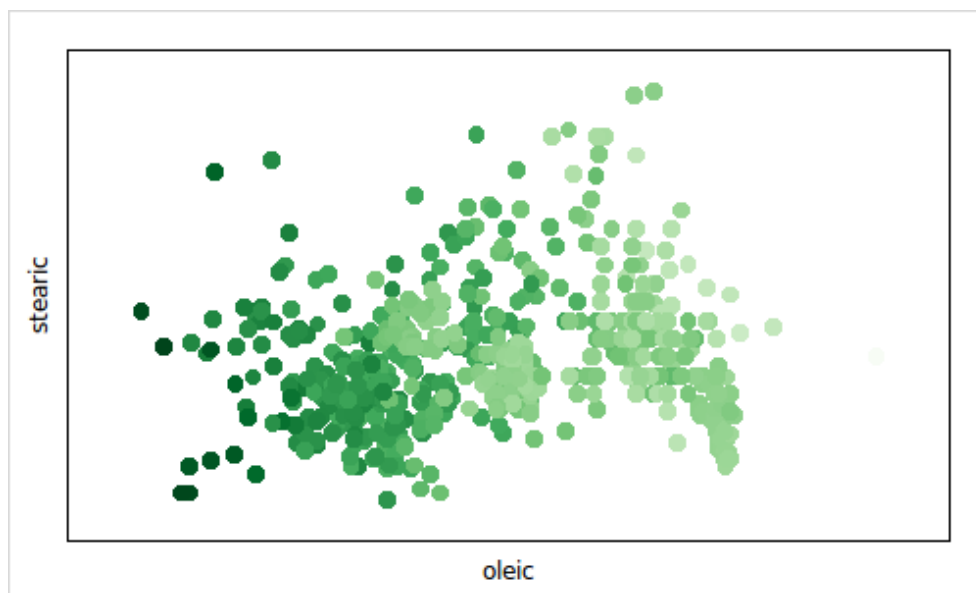
Figure 5.5: Hcl colors: luminance is 70, chroma on circle is 66. The numbers indicate loon's default color mapping order.

variable with a sequential color gradient of light green to dark green, see [Figure 5.6](#).

```
p_loon <- l_plot(stearic~oleic, color=Area)
library(scales)
p_custom <- l_plot(stearic~oleic,
  color = col_numeric("Greens", domain = NULL)(palmitic))
```



(a) Area encoded with loon's default color mapping



(b) palmitic encoded with shades of green

Figure 5.6: Examples of mapping data values to colors.

# Chapter 6

## General Statistical Interaction

### Examples

In this chapter, we illustrate how we can enhance four common statistical methods by using interactive visualization with `loon`. In all of these examples, a user interface action (on a plot or control panel) evaluates R code which in turn updates the plot(s) to visualize the desired results, hence providing a “real-time” exploration of the data and the applied model. These examples are exploratory in terms of looking at the data and/or models. Also, we use these examples to demonstrate that `loon` is a useful tool for exploratory programming [65]; that is, `loon` is very efficient in implementing advanced interaction settings and, hence, it can be used to quickly try and compare different interactive ways to look at the data.

In [Section 6.1](#), we use sliders to control the parameters of a Box-Cox transformation for data on both  $x$  and  $y$  axes. This is our “Hello World” example of interactive statistical visualization with `loon` as power transformations have been used previously to demonstrate statistically meaningful interactivity with software such as DINDE [58] and Lisp-Stat [76] pp. 60-62.

[Section 6.2](#) presents a setting where a new control panel is created to fit regression lines of a certain degree to points selected on a scatterplot. The example shows how control panels and functionality can be dynamically created and added ad-hoc to a `loon` plot.

In [Section 6.3](#), we introduce a setup where a simple linear regression is enhanced with linked residual and leverage plots. Points on the plot can be interactively removed from the OLS, or temporarily moved around to get an insight of the sensitivity of the OLS fit to certain observations. This example serves as a teaching and exploration tool where the user can get insight into the nature of an OLS by modifying the data interactively.

Finally, in [Section 6.4](#), we show a tool to highlight the  $k$  nearest neighbors of the selected points in a particular sub-space. This an example of an ad-hoc tool to explore high dimensional data.

The code for these examples is distributed in the `loon` R package as package demos and also attached in the [Appendix A](#). The functionality of most examples discussed here is encapsulated into a function and could be applied to other data or plots. However, when writing the code for these examples, we chose simplicity over generality to give the reader an idea of how much R code is required to implement the desired functionality for a particular well-formatted data set. Hence, each example code could be further developed to be more robust and with more generalized functionality.

## 6.1 Power Transformations

[Figure 6.1](#) shows a scatterplot and two sliders (i.e. Tk scale widgets) that control the  $\lambda$  parameter of a Box-Cox power transformation [\[11\]](#) for both  $x$  and  $y$  coordinates as shown in [Equation 6.1](#).

$$y^{(\lambda)} = \begin{cases} \frac{y^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \log y & \lambda = 0 \end{cases}, \quad y > 0 \quad (6.1)$$

The data used for [Figure 6.1](#) are those of brain vs. body weight for 62 mammals [\[1\]](#). The code for this example can be found in the `'l_power'` demo of the `loon` R package and in [Section A.1](#). With this code, we create a function `power` with 31 non-empty lines of R code that generates the scatterplot, scales and bindings for the desired functionality. Then, [Figure 6.1](#) was created as follows

```

library(MASS)
p <- with(mammals, power(body, brain,
  xlabel="body weight in kg",
  ylabel="brain weight in g",
  title="Brain and Body Weights for 62 Species of Land Mammals",
  itemlabel=rownames(mammals),
  showItemlabels=TRUE))

```

Moving the two sliders will apply the power transformation in [Equation 6.1](#) to the corresponding data and updates the scatterplot accordingly. Every change will end with a *scale to world* operation (i.e. using the `l_scaleto_world` function) in order to display all the data.

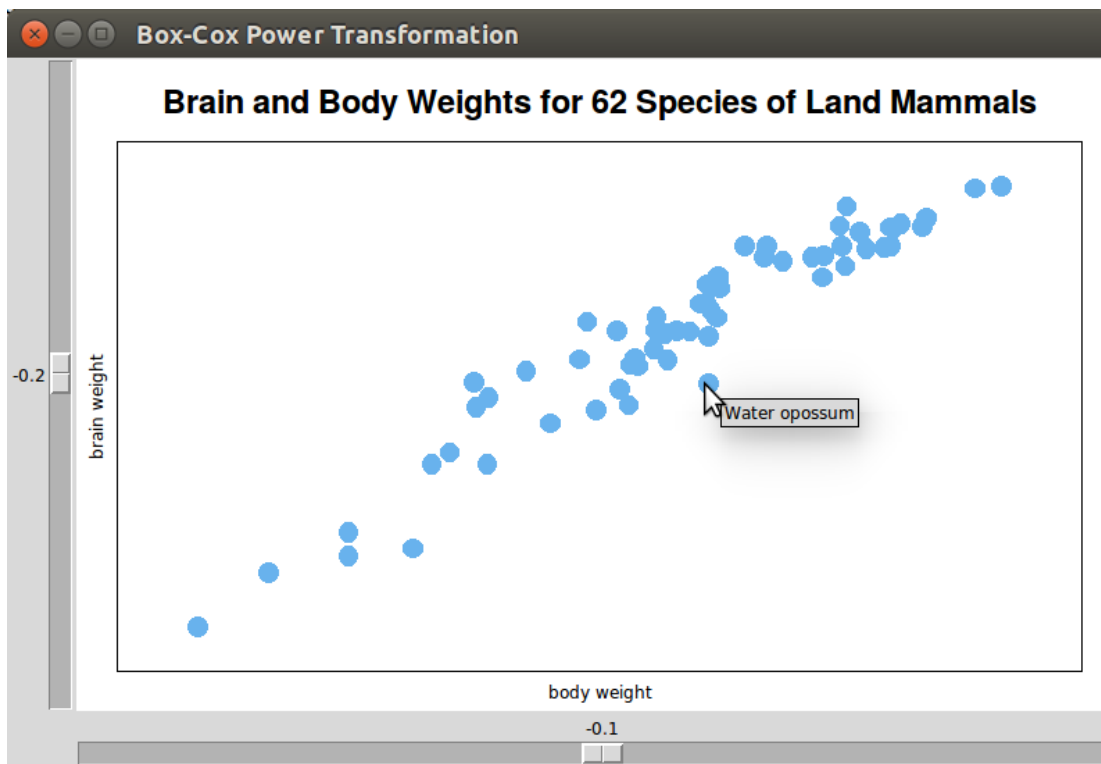


Figure 6.1: Power transformation example.

## 6.2 Interactively Adding Regression Lines

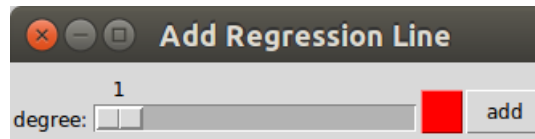
In this example, we demonstrate a tool to fit linear regression models of user-defined degrees to the selected points in a scatterplot. The code for this tool can be found in the 'l\_add\_regressions' demo and in [Section A.2](#). With this code, we first define the function `addRegressionLinesGUI` in 33 non-empty lines of code. This function takes the scatterplot handle as an argument and creates the GUI shown in [Figure 6.2\(a\)](#). This control panel provides a slider to choose the degree of the regression, a button to choose the color of the layered regression line, and an “add” button that layers the regression line for the selected points onto the scatterplot. In addition to layering the line, the point glyphs of the selected points are changed to have the glyph type `ocircle` (i.e. outline only) and to have the color of the regression line. [Figure 6.2\(b\)](#) shows a scatterplot with simulated data and three regression lines with degree 1, 3 and 5 from the left to the right. The code to create the setup in [Figure 6.2](#) is

```
x <- runif(500)*7
y <- sapply(x, function(x) {
  if (0 <= x && x < 2) {
    5*x + rnorm(1,0,1)
  } else if (2 <= x && x < 5) {
    8.6 + 2*x - .6*x^2 + rnorm(1,0,.5)
  } else {
    8.5 - log(x) + rnorm(1,0,.8)
  }
})

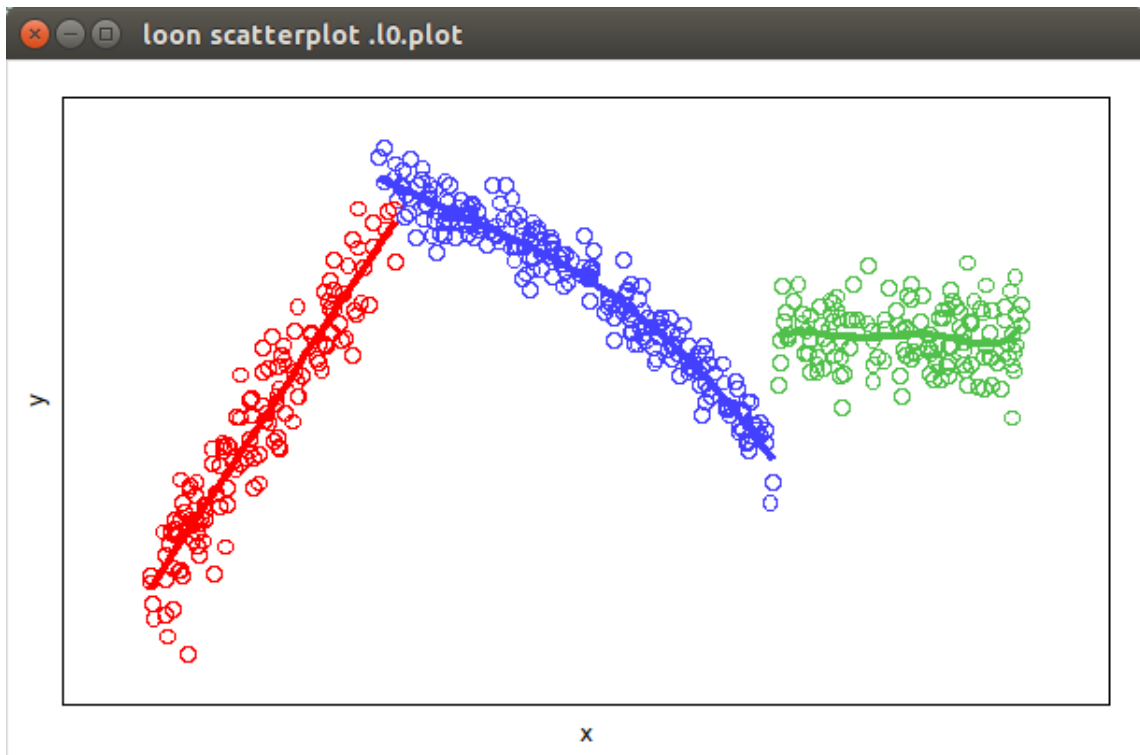
p <- l_plot(x,y)
addRegressionLinesGUI(p)
```

The second last line creates the scatterplot whereas the last line creates the control panel to fit and layer the regression lines onto the scatterplot.

The `addRegressionLines` tool can be used for every loon scatterplot as the plot handle argument is sufficient to retrieve the  $x$  and  $y$  coordinates of the selected points and to layer the regression lines.



(a) control panel



(b) scatterplot with regression lines

Figure 6.2: Interactively adding regressions lines.



## 6.3 Sensitivity Analysis of a Simple Linear Regression

In this example, we use the `swiss` data [53] and fit a simple linear regression on the fertility measure against the infant mortality counts for 47 French-speaking provinces of Switzerland at about 1888. The `swiss` data set is distributed with the `datasets` R package and the code for this example can be found in the `l_regression_influential` demo of the `loon` R package and also in [Section A.3](#). [Figure 6.3\(a\)](#) shows the data, regression line and the 95% and 99% confidence intervals (blue shades) and prediction intervals (pink shades). [Figure 6.3\(b\)](#) shows the residuals vs. fitted plot, [Figure 6.3\(c\)](#) shows the leverage plot and [Figure 6.3\(d\)](#) shows the associated spatial location for each province relative to the border of Switzerland. All these plots are linked such that the analyst can interactively look for patterns. For example, in [Figure 6.3](#), we select the provinces with positive residuals and from the map plot we note that these provinces tend to be located in the eastern region of the provinces.

In the `l_regression_influential` demo, we also add two state bindings to the scatterplot with the least-squares fit, see [Figure 6.3\(a\)](#). One state binding is bound to `color` state changes and re-calculates the OLS for the blue points only (the blue shown in the first color modify box of the plot inspector) and updates the regression related plots accordingly, see [Figure 6.4](#). In [Figure 6.4](#), the points colored brown have been removed from the OLS and, as a result the prediction and confidence intervals have become narrower. The red line represents the new fit and the black line the original fit. This functionality is very useful for interactively removing outliers.

The second state binding is bound to `xTemp` and `yTemp` state changes and updates the OLS using the temporary coordinates. For example, in [Figure 6.5](#), the outliers are still removed as in previous example and, additionally, the data point with the highest leverage value has been selected. Next, we moved this point around and eventually placed it as shown in [Figure 6.5](#). We note that the new regression line is not substantially affected by removing some outliers and by moving the most influential point.

Adding functionality to a simple linear regression setting as demonstrated with this example could have a big impact on teaching regression analysis and can also be helpful for experienced statisticians.

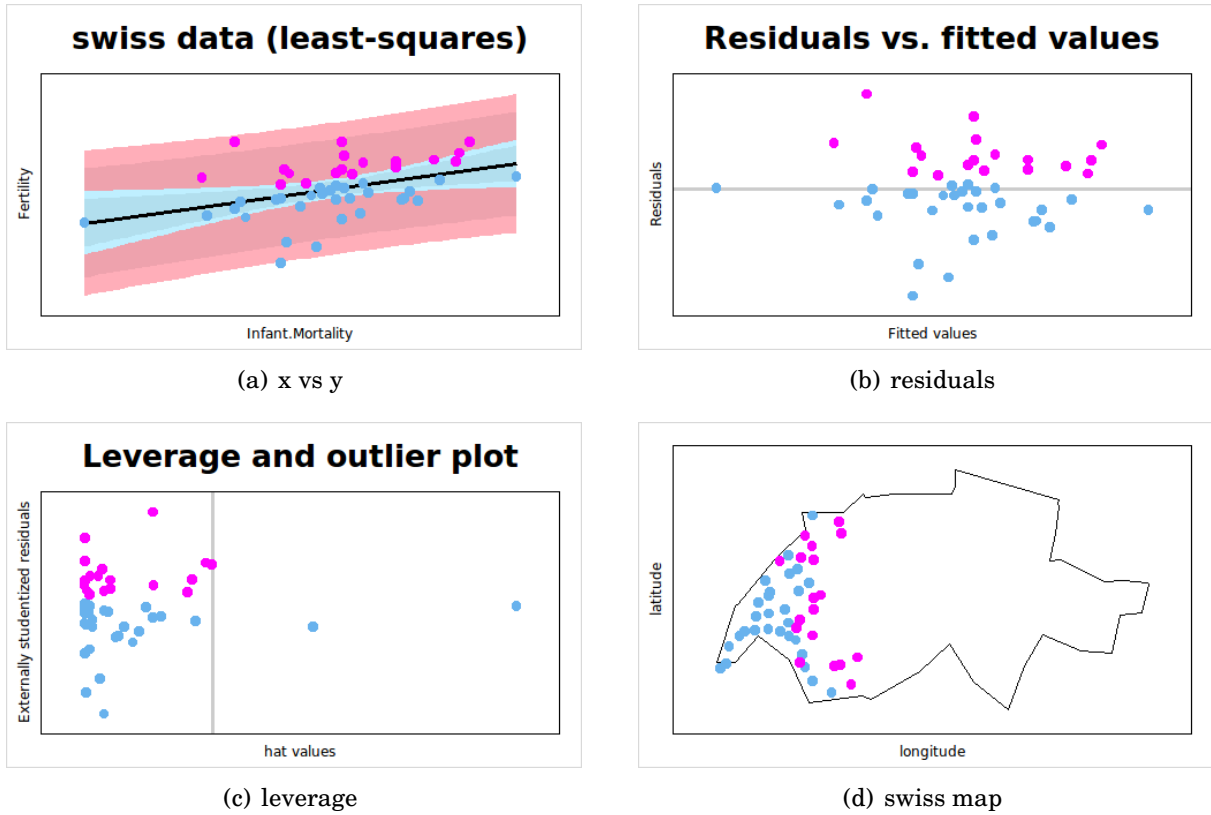


Figure 6.3: Influential points in regression analysis.

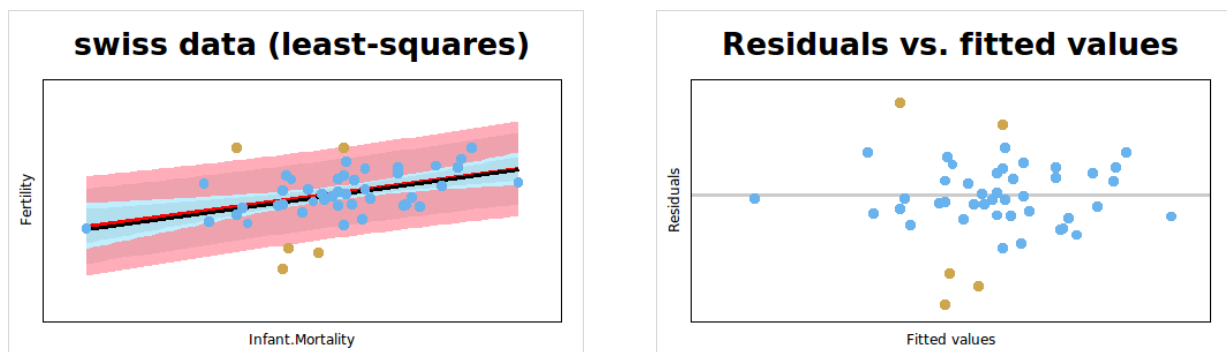


Figure 6.4: Influential points in regression: recolor points to remove outliers.

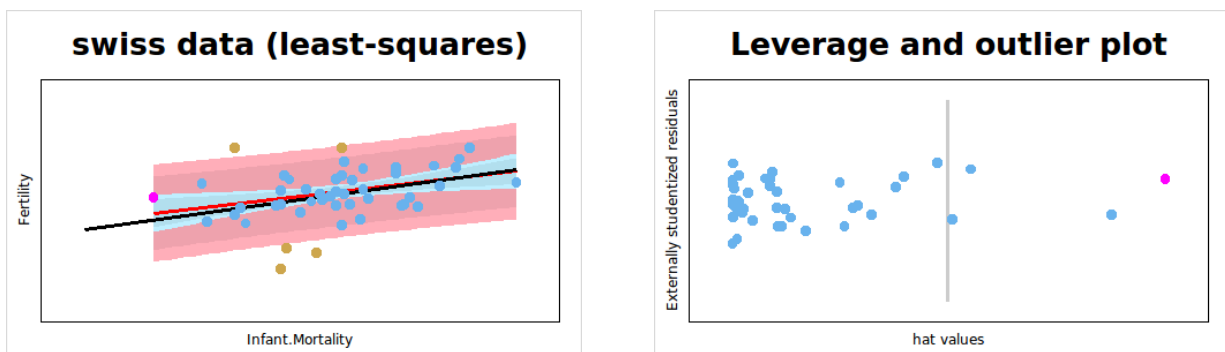


Figure 6.5: Influential points in regression: sensitivity analysis.

## 6.4 Interactive K Nearest Neighbor highlighting

In this section, we provide R code to highlight the  $k$  nearest neighbors of the points selected on a scatterplot. The section is divided into two parts to describe the approach we took to solve this problem. That is, we first wrote code to quickly get to the desired functionality: highlight the  $k$  nearest neighbors of the selected points. Next, we refined the  $k$  nearest neighbor highlighting idea to a more involved setting and built a feature-rich new tool.

### 6.4.1 A Quick Solution

Executing the following code results in the  $k = 3$  nearest neighbors of the selected points being highlighted in the scatterplot with handle `p`. We highlight the neighbors by increasing their point size and coloring them orange. For multiple selected points, the measure of nearness is defined by the minimum distance to any of the selected points.

```
1 n <- 100
2 x <- runif(n)
3 y <- runif(n)
4
5 k <- 3
6 D <- as.matrix(dist(cbind(x,y), method="euclidian"))
7 I <- matrix(rep(1:n, n), ncol=n, byrow=TRUE)
8
9 hnn <- function(W) {
10     l_configure(W, color='steelblue', size=5)
11     isel <- which(l_cget(W, 'selected'))
12     if(length(isel) != 0) {
13         ind_close <- unique(c(I[isel, -isel])[order(c(D[isel, -isel]))])
14         ind <- ind_close[seq(1, min(k, length(ind_close)))]
15         l_configure(W, color='orange', size=15, which=na.omit(ind))
16     }
17 }
18 p <- l_plot(y~x, showScales=TRUE)
19 l_bind_state(p, 'selected', function(W)hnn(W))
20 l_aspect(p) <- 1
```

Line 2 and 3 generate the data vectors `x` and `y` and line 5 generates the pairwise euclidean distance matrix `D` for the elements in `x` and `y`. The matrix `I` has the same dimension as `D` and its elements contain the corresponding column indices for `D`. The matrix `I` is used to get the point indices of the nearest neighbors. The function `hnn` which is defined

on lines 9 – 17 first resets the size and color of all points and assigns the point indices of the selected points to the variable `isel`. If any points are selected then the sub-matrix `D[isel, -isel]` contains all the distances to the non-selected points and `I[isel, -isel]` contains their point indices. Both sub-matrices are then vectorized and used to find the closest points. Line 18 creates the scatterplot of `x` and `y` and line 19 adds a state binding to this scatterplot that evaluates the `hnn` function at every selected state change. We wrapped the function call to `hnn` in an anonymous function in order to be able to assign new functions to the `hnn` variable for debugging purposes. On line 20, we set the aspect ratio to 1 so that the inter-point distances on the screen are proportional to the data inter-point distances. Figure 6.6 shows a plot produced with the above code and with 4 selected points.

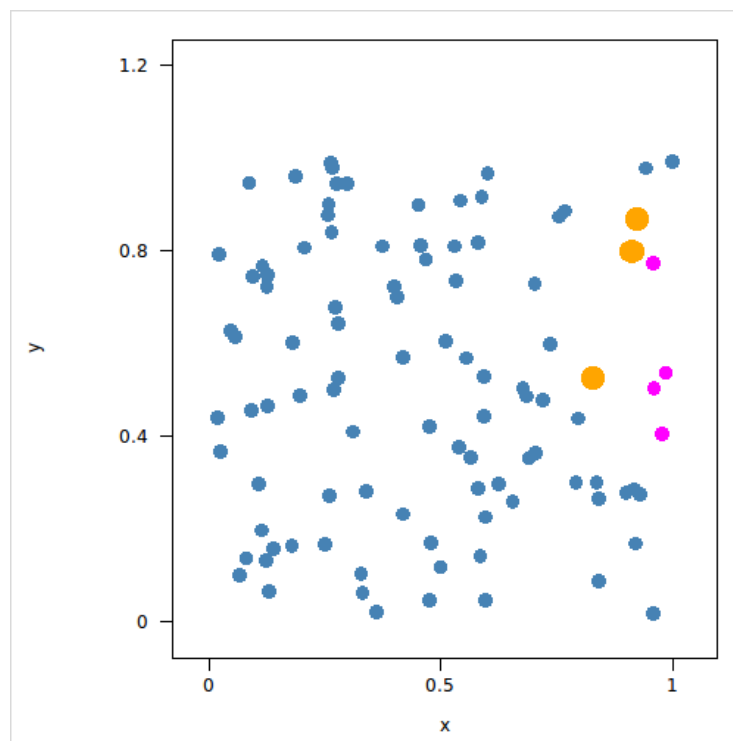


Figure 6.6: 3 nearest neighbors highlighted.

This example shows how loon and R can be used to quickly get to the desired functionality. In the above code, the `hnn` function scopes for `D` and `I` in the global environment and, hence, it is not safe to reuse this function. To create a reusable version we could wrap the functionality into a function as follows

```
highlight_knn_quick <- function(p, D, k=3) {
  n <- dim(D)[1]
  I <- matrix(rep(1:n, n), ncol=n, byrow=TRUE)
  hnn <- function() {
    l_configure(p, color='steelblue', size=5)
    isel <- which(l_cget(p, 'selected'))
    if(length(isel) != 0) {
      ind_close <- unique(c(I[isel, -isel][order(c(D[isel, -isel]))])
      ind <- ind_close[seq(1, min(k, length(ind_close)))]
      l_configure(p, color='orange', size=15, which=na.omit(ind))
    }
  }
  l_bind_state(p, 'selected', hnn)
}
```

The `highlight_knn_quick` function can then be used for any scatterplot without any scoping issues. Also, the distance matrix argument `D` accepts any valid distance matrix including inter-point distances in more than two dimensions. For example, the following code creates a scatterplot using the olive data and then uses the `knn_simple` function to have the 5 nearest neighbours highlighted in the complete `oliveAcids` data space.

```
p <- l_plot(oleic~stearic)
highlight_knn_quick(p, D=as.matrix(dist(scale(oliveAcids))), k=5)
```

In the next subsection we present a more capable  $k$  nearest neighbor highlighting tool.

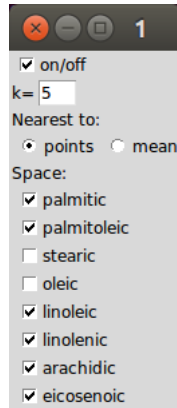
## 6.4.2 A Solution With Control Panel

The code included in the `l_knn` demo and [Section A.4](#) defines the `highlight_knn` function that provides more features than the `highlight_knn_quick` function defined in the previous subsection. For example, the call

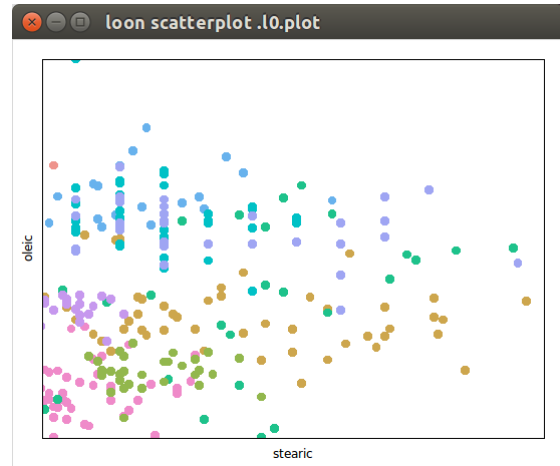
```
sOliveAcids <- data.frame(scale(oliveAcids))
p <- with(sOliveAcids, l_plot(oleic~stearic, color=Area))
highlight_knn(p, data=sOliveAcids, k=5)
```

creates the scatterplot with handle `p` and the control panel shown in [Figure 6.7\(a\)](#). This control panel can be used to switch the nearest neighbor highlighting on and off, choose the  $k$ , select between two nearest distance measures and choose a subspace for which the point inter-distances are calculated. When selecting the “mean” option for the “nearest to” radiobutton, the center of the selected points in the euclidean space is used to get the distances of the non-selected points. The `highlight_knn` highlights the  $k$  nearest neighbors by changing the glyph type to rectangles and rank their sizes according to their distance measures; that is, the biggest square is closest to the selected points. Also, when the selected state changes, the size and glyph states of the  $k$  nearest neighbors get cached so that these states can be reset once the points are no longer highlighted.

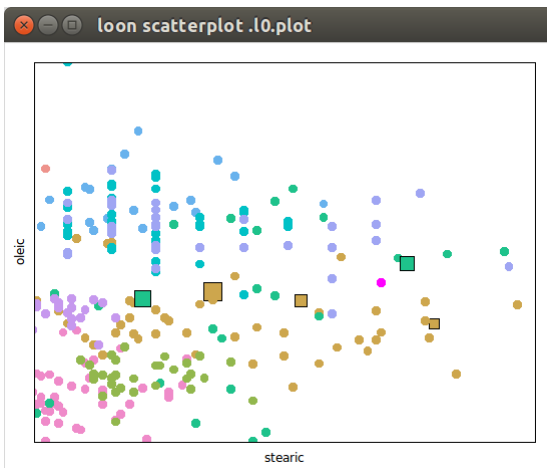
[Figure 6.7\(b\)](#), [Figure 6.7\(c\)](#) and [Figure 6.7\(d\)](#) show the same plot region on a scatterplot display once with no selected point, once with one selected point and once with a group of selected points for the settings shown in the control panel in [Figure 6.7\(a\)](#).



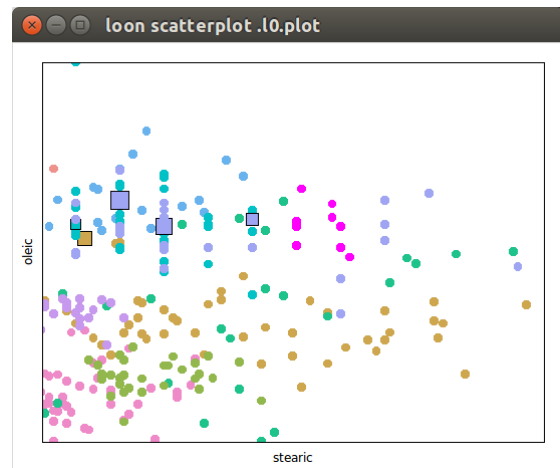
(a) control panel



(b) no points selected



(c) single point selected



(d) group of points selected

Figure 6.7:  $K$  nearest neighbors highlighting for subspaces.



# Chapter 7

## Exploring High-Dimensional Data

In this chapter, we present new tools for exploring high-dimensional data. In [Section 7.1](#), these tools are centered around navigation graphs. The building blocks for working with navigation graphs in `loon` were introduced in [Subsection 4.7.4](#) and include the graph display, navigators and contexts. Now, we use these building blocks to construct data exploration settings with navigation graphs that use the canonical graph semantic (i.e. with the `geodesic2d` context). In [Subsection 7.1.1](#) we present the `l_navgraph` function that sets up a navigation graph setting based on a data set alone. In [sections 7.1.2 to 7.1.4](#) we present tools that create settings for dealing with large graphs by finding interesting sub-graphs based on measures of interestingness for variables or variable pairs, as we discussed in [Subsection 1.4.1](#). Next, in [Subsection 7.1.5](#), we discuss the implementation of a new navigation graph semantic with `loon`. The novel graph semantic that we implement provides another way to deal with large graphs by slicing navigation graphs.

[Section 7.2](#) ends this chapter with the introduction of a novel point glyph called *spiro glyph*. A spiro glyph is the trajectory of a point that is projected onto a moving plane on that plane. That is, spiro glyphs capture the trajectories of the points in a smooth movie of different projections, for example, from traversing a navigation graph or from a grand tour.

## 7.1 Navigation Graphs

### 7.1.1 Canonical Navigation Graph Setup

The `l_navgraph` function provides a quick and convenient way to construct a navigation graph setup using the canonical graph semantic (i.e. the `geodesic2d` context) for a data set. For example, the following code creates the setup shown in [Figure 7.1](#):

```
nav <- l_navgraph(data=oliveAcids, color=Area)
```

That is, for the above code, the `l_navgraph` function creates three transition graphs, a graph widget and a graphswitch widget, and it further adds a navigator to the graph widget and a `geodesic2d` context to the navigator. The `geodesic2d` context creates, in turn, the scatterplot widget that displays the projected data based on the navigator's position. The three automatically created transition graphs are the saturated  $3d$  and  $4d$  transition graphs and a complete  $2d$  space graph for all variable pairs of `oliveAcids`. The return value of the `l_navgraph` function, here assigned to `nav`, is a list with the graph, graphswitch, plot, navigator and context handles.

The `l_navgraph` function also allows for some customization of the default graph setting with the optional named arguments `separator` and `graph`. The `separator` argument specifies a string (without spaces) that is used to separate the variable names in the nodes of the transition graphs. The optional `graph` argument allows the user to specify a custom transition graph for the navigation graph setting.

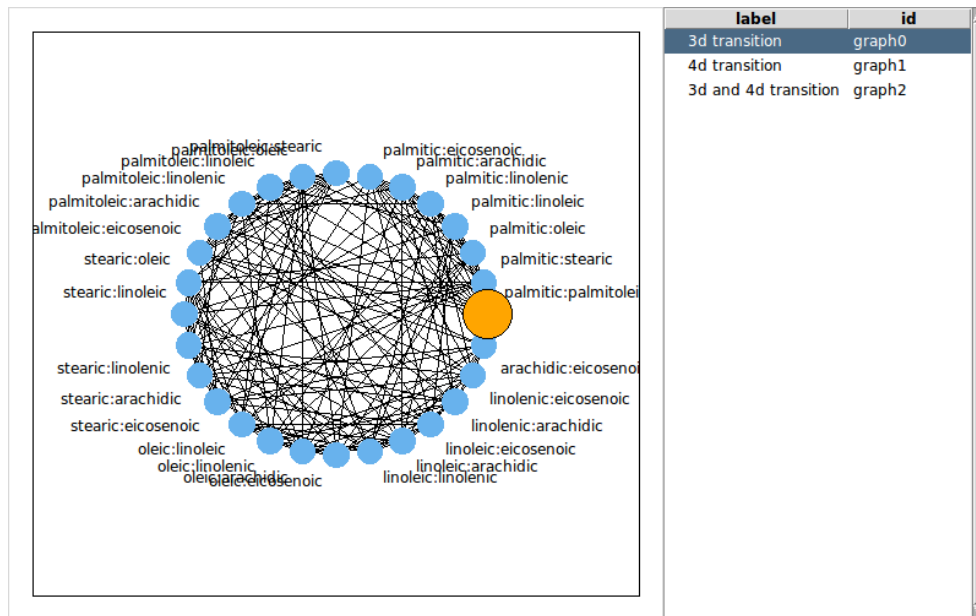
The setting created by `l_navgraph` can be used as a starting point for a more involved data exploration setting. This is possible due to `loon`'s modular framework; the `l_navgraph` function returns all widget and object handles for the navigation graph setting it created. Hence, with these handles one could, for example, add more transition graphs to the graphswitch widget, add the  $k$  nearest neighbor highlighting from [Section 6.4](#) to the scatterplot or add new point glyphs to the scatterplot. Note that these kind of additions and modifications are not possible for a running navigation graph session with `RnavGraph`. For example, the following code adds serialaxes glyphs to the scatterplot in [Figure 7.1](#):

```
library(PairViz)

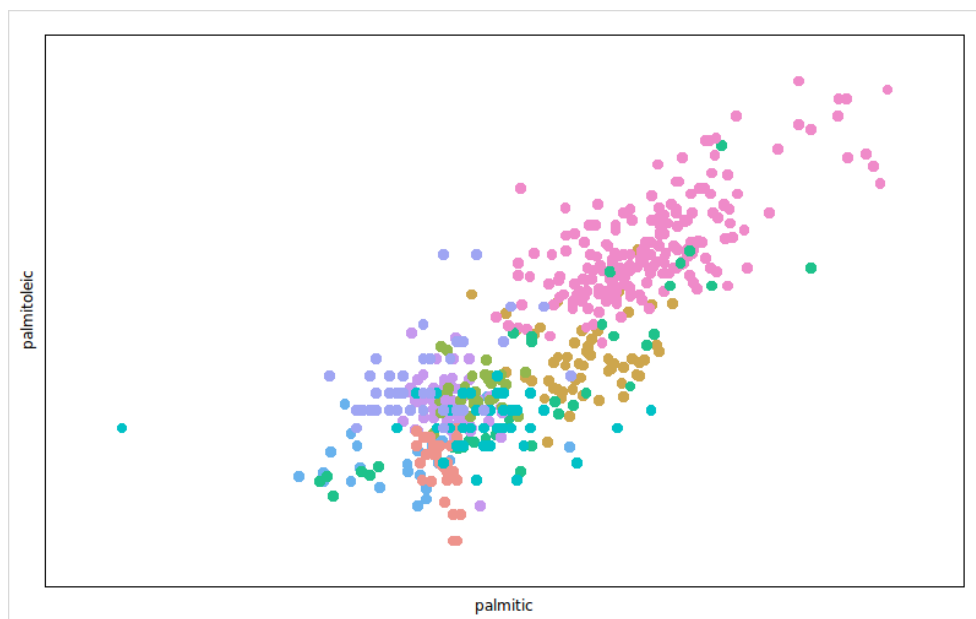
gl <- l_glyph_add_serialaxes(nav$plot,
  data=oliveAcids,
  sequence=c(t(hpaths(names(oliveAcids))))),
  showArea=FALSE)

l_configure(nav$plot, glyph = gl, size=1)
```

The variable sequence for the serialaxes glyphs is chosen to have every variable pair neighbouring at least once, as proposed in Hurley and Oldford [39]. That is, the variable sequence is the concatenated Hamiltonian decompositions of the complete variable graph for the oliveAcids data. The scatterplot showing these serialaxes glyphs is seen in [Figure 7.2](#).



(a) Navigation graph and Graphswitch



(b) Scatterplot display showing the projection according to the navigator position

Figure 7.1: 1\_navgraph setup.

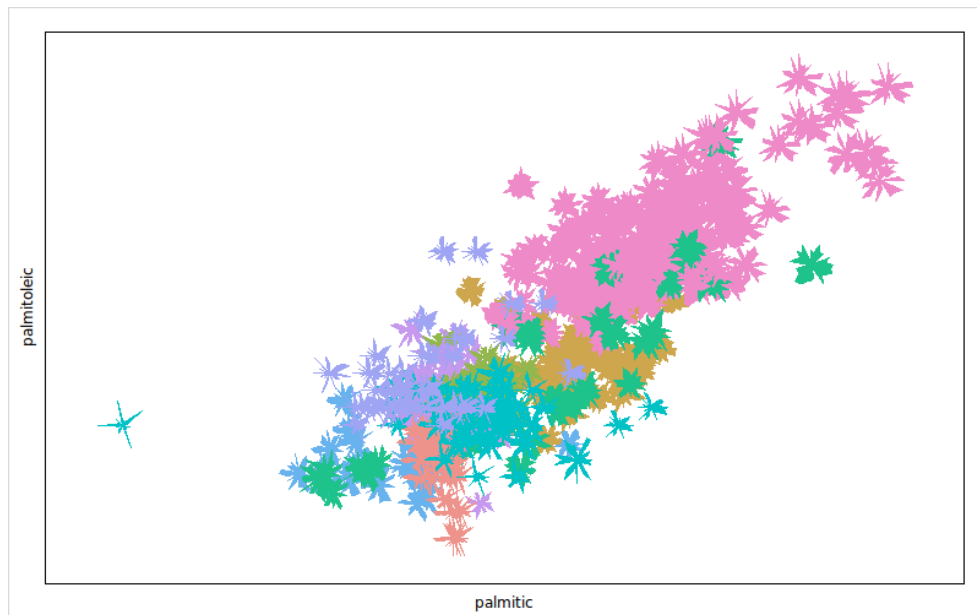


Figure 7.2: Star glyphs.

## 7.1.2 Dynamic Navigation Graphs Based on Measure Ranges

In [Subsection 1.4.1](#), we discussed the possibility of associating a measure of “interestingness” to either the nodes or the edges of a transition graph to determine an interesting sub-graph. The `l_ng_ranges` function creates such a setting (i.e. for finding a sub-graph based on measures) by providing a min-max-slider to filter measures associated with the nodes of some saturated transition graph.

We now explain the details of `l_ng_ranges` function based on an example setting for the following manageable 5-dimensional randomly generated data:

```
n <- 100
dat <- data.frame(
  A = rnorm(n), B = rnorm(n), C = rnorm(n),
  D = rnorm(n), E = rnorm(n)
)
```

Assume that we are interested in a transition graph for these data that has at most the following five 2d-spaces represented by its nodes:  $(A,B)$ ,  $(A,C)$ ,  $(B,D)$ ,  $(D,E)$  and  $(A,E)$ . Also, assume that, for these data, one measure of interest is the correlation of the variable pairs and another measure is assigned by us based on some judgment-based criteria. The `l_ng_ranges` function requires the measure values for the variable pairs to be arranged in a data frame as follows:

```
m2d <- data.frame(
  cor = with(dat, c(cor(A,B), cor(A,C), cor(B,D), cor(D,E), cor(A,E))),
  my_measure = c(1, 3, 2, 1, 4),
  row.names = c('A:B', 'A:C', 'B:D', 'D:E', 'A:E')
)
```

The `m2d` data set looks as follows:

```
m2d
      cor my_measure
A:B 0.06452377      1
A:C 0.13580683      3
B:D 0.10066350      2
D:E -0.02725931      1
A:E 0.18043338      4
```

Note that the row names of `m2d` represent the 2d variable spaces with a string of the variable names concatenated by the colon character. Hence, `m2d` defines a measure named `cor` and one named `my_measure` for five 2d sub-spaces of the data `dat`. The following `l_ng_ranges` call creates the setup shown in [Figure 7.3](#):

```
nav <- l_ng_ranges(measures=m2d, data=dat, separator=':')
```

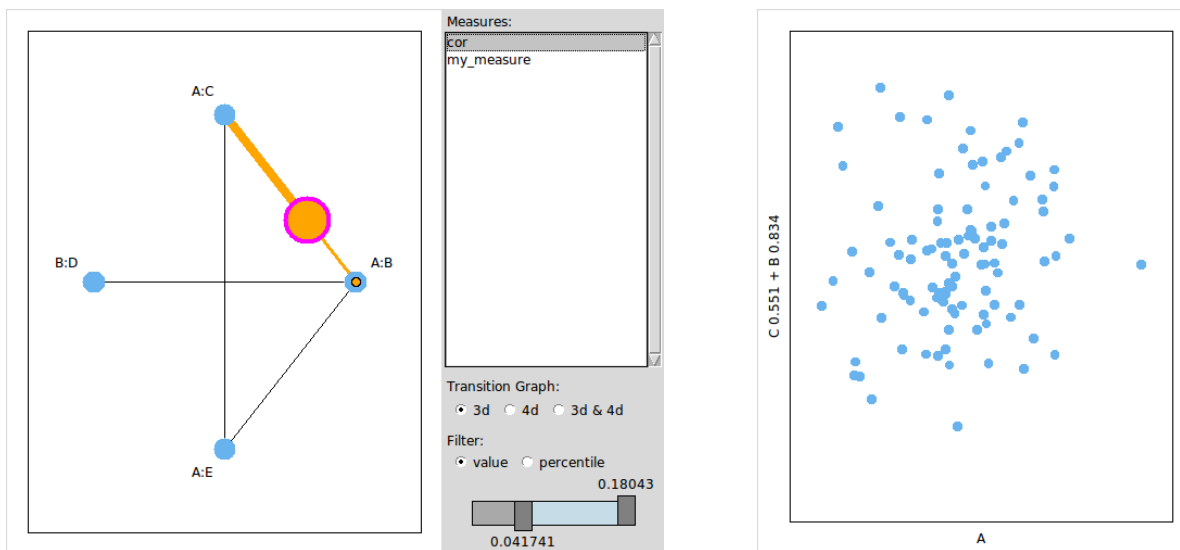


Figure 7.3: `l_ng_ranges` setup with 2d measures.

[Figure 7.3](#) shows a graph widget with a custom control panel packed next to it and a separate scatterplot widget that is driven by the navigator in the graph widget. The control panel has a list of all measure names and provides a min-max-slider to filter the variable pairs based on their associated selected measure to create a transition graph. For example, in [Figure 7.3](#), the control panel is set to use the `cor` measure to create a 3d transition graph with nodes whose corresponding `cor` measure lies in the interval  $[0.041741, 0.1843]$ . Only one measure can be used to filter the variable pairs of the transition graph. The return value of `l_ng_ranges` is a list with graph, navigator, context and plot handles.

With `l_ng_ranges` it is also possible to dynamically build transition graphs that are based on 1d measures for single variables. With 1d measures, the min-max slider filters variables, and the transition graph will have nodes for all variable pairs in the subset of filtered variables. For example, assume we are interested in the following robust measures: median, inter-quartile range, kurtosis and skewness. We can create an `l_ng_ranges` session for 1d measures that works as in the case of 2d measures with the exception that the measures are now for variables in the data and not for variable pairs:

```
iqr <- function(x) { diff(quantile(x, probs=c(0.75, 0.25))) }
kurtosis <- function(x) { mean((x-mean(x))^4)/mean((x-mean(x))^2)^2 - 3 }
skewness <- function(x) { mean((x-mean(x))^3)/sd(x)^3 }

m1d <- data.frame(
  median = sapply(dat, median),
  irq = sapply(dat, iqr),
  kurtosis = sapply(dat, kurtosis),
  skewness = sapply(dat, skewness),
  row.names = names(dat)
)
```

The `m1d` data set looks as follows:

```
m1d

      median      irq  kurtosis  skewness
A -0.02178844 -1.152425  0.2259010  0.03878397
B -0.12519883 -1.354264 -0.3003271 -0.05734108
C  0.10453212 -1.332907 -0.1755559 -0.17443438
D -0.13740205 -1.428953 -0.1196611  0.11410720
E  0.09957466 -1.141733  0.3152615 -0.30971828
```

The `l_ng_ranges` function with the `m1d` measures is invoked as follows:

```
nav <- l_ng_ranges(measures=m1d, data=dat, separator=':')
```

**Figure 7.4** shows the setup created by this code. The `l_ng_ranges` function knows whether the data in the `measures` argument is for single variables or for variable pairs by checking whether the row names of the measures data separate into one or two variables using the separator string.



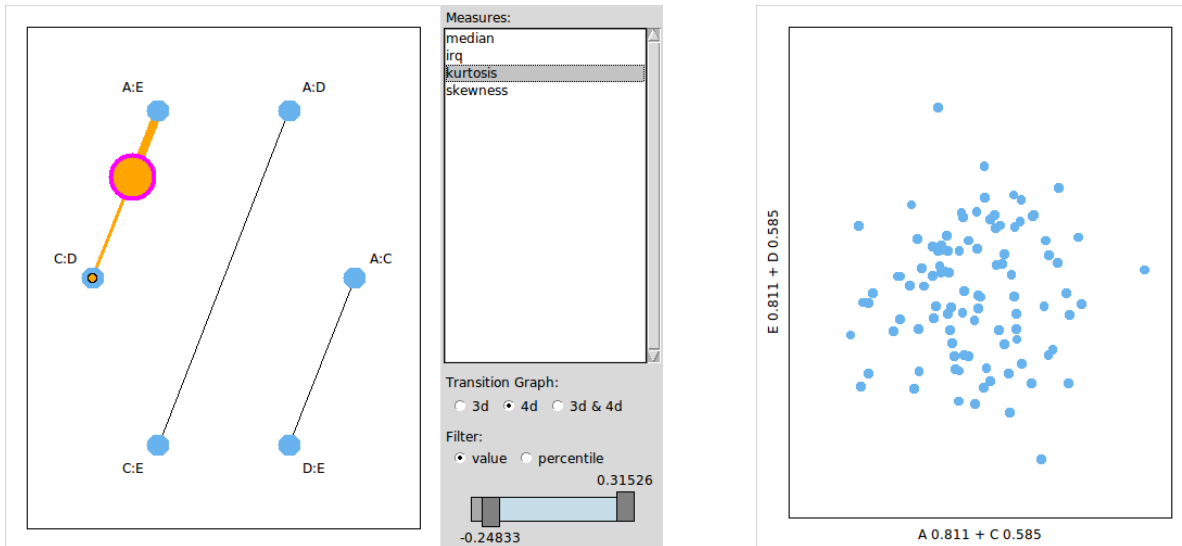


Figure 7.4: `l_ng_ranges` setup with 1d measures.

We now provide an example of `l_ng_ranges` for the olive data and 2d scagnostics measures [83]. We start by calculating the scagnostics measures for the variable pairs in `oliveAcids`:

```
library(scagnostics)
scags <- scagnostics(oliveAcids)
```

To make the return value of the `scagnostics` function a valid `measures` argument for `l_ng_ranges`, it is necessary to strip the class attribute from `scags`, transpose the matrix and replace the spaces in the row names of the matrix:

```
m2d_scags <- t(unclass(scags))
row.names(m2d_scags) <- gsub(' ', '', row.names(m2d_scags), fixed=TRUE)
```

Finally, the `l_ng_ranges` session is created in the following code and shown in Figure 7.5.

```
nav <- l_ng_ranges(measures=m2d_scags, data=oliveAcids,
                  separator='*', color=Area)
```

Note that the `l_ng_ranges` is a generic function with a method for objects of class `scagnostics`. Hence, the following call creates the same setting as obtained with the above code.

```
nav <- l_ng_ranges(measures=scagnostics(oliveAcids), data=oliveAcids, color=Area)
```

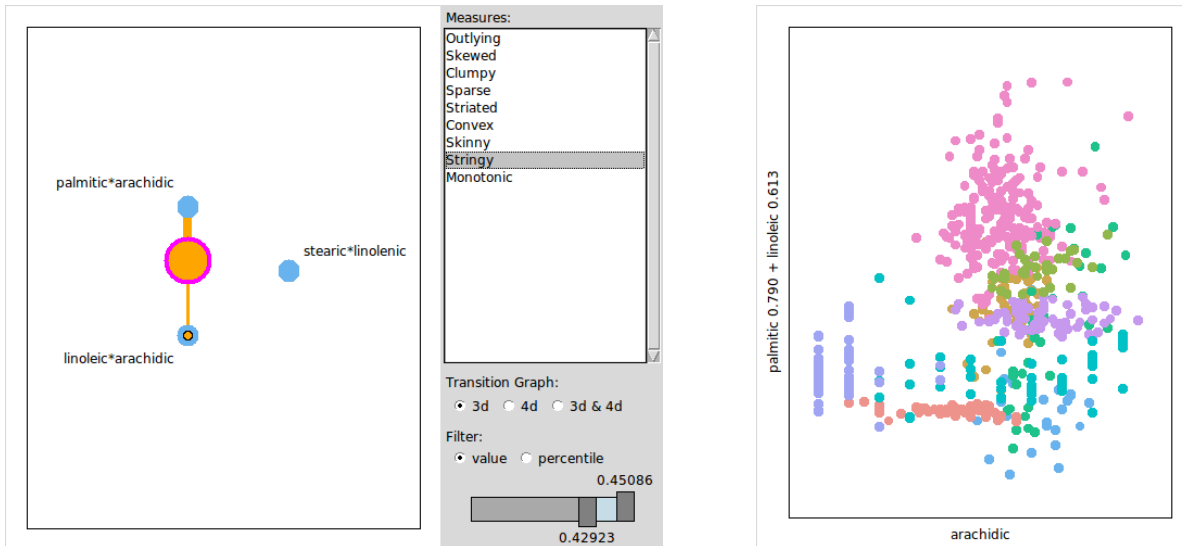


Figure 7.5: `l_ng_ranges` using scganostics measures and the olive data.

### 7.1.3 Dynamic Navigation Graph based on Plots

The `l_ng_plots` function differs from the `l_ng_ranges` function in that it creates a scatterplot matrix of the measures instead of a control panel in order to filter measures. That is, it uses the selected state of the scatterplots in the scatterplot matrix to choose which sub-spaces should be included in the navigation graph. If a single measure is specified a histogram is created instead of a scatterplot matrix.

The `l_ng_plots` function is used exactly the same as the `l_ng_ranges` function. In fact, for the `l_ng_ranges` and `l_ng_plots` functions calls are interchangeable (though resulting with different settings). For the following example we use the *Frey faces* data that contains 1965 greyscale images of Brendan Frey’s face, taken from sequential frames of a small video. The size of each image is  $20 \times 28$  pixels and the data can be found as the `frey` data frame in the `RnavGraphImageData` R package. We reduce the  $20 \cdot 28 = 560$  dimensional data to 15 dimensions using locally linear embedding [62] with 12 nearest neighbors. We then calculate the scganostics measures on the variable pairs of the reduced data and create a `l_ng_plots` setting with these data.

```

library('RnavGraphImageData')
library(RDRToolbox)
library(scagnostics)
data(frey)

frey.lle <- as.data.frame(LLE(t(frey), dim=15, k=12))

nav <- l_ng_plots(scagnostics(frey.lle), frey.lle)

```

Figure 7.6 shows the setting created with the above code. We selected the nodes with high striated and low skewed scagnostics measures. The return value of `l_ng_plots` is a list with the graph, navigator and scatterplot handles. We continue the above example by adding image glyphs with the Frey faces to the scatterplot. Figure 7.7(a) shows the scatterplot with the face image glyphs scaled to the points in the lower “spike” in the scatterplot in Figure 7.6. Figure 7.7(b) shows the same faces arranged on a grid.

```

frey.imgs <- l_image_import_array(frey, 28, 20, img_in_row=FALSE, rotate=90)
gl <- l_glyph_add_image(nav$plot, images=frey.imgs, label="frey faces")

```

## 7.1.4 Closures of Measures

When the functional form of the measures used for `l_ng_ranges` and `l_ng_plots` is known then it is possible to focus on a subset of the data points and have the measures recalculated for that subset. To illustrate the usefulness of recalculating measures for a subset we look at the olive data and the clumpy scagnostics measure. The scatterplots with a high clumpy scagnostics measure (relative to the other scatterplots) for the olive data are those that are separated into the two point clouds as seen in Figure 7.8(a). When the points in the smaller point cloud on the left hand side in Figure 7.8(a) are deactivated and the clumpy scagnostics measure is recalculated for all scatterplots of the active points then the scatterplot in Figure 7.8(b) has a high clumpy measure relative to the other scatterplots.

The `measures1d` and `measures2d` functions can be used to specify the functional form of 1d or 2d measures for the `l_ng_ranges` and `l_ng_plots` functions. That is, the `measures1d` and `measures2d` functions encapsulate the measure functions, data and separator, and return a closure of class measures. The `l_ng_ranges` and `l_ng_plots` are generic functions and have a method for objects of class measures. This method creates the same setup as

described in the previous sections and adds an additional panel with buttons for recalculating the measures based on the active or selected points in the scatterplot that shows the projections.

We start with an example of closures for 1d measures for the scaled olive data.

```
s_oliveAcids <- scale(oliveAcids)

m1dc <- measures1d(data=s_oliveAcids, separator='+',
                  median = median,
                  irq = iqr,
                  kurtosis = kurtosis,
                  skewness = skewness)
```

The `m1dc` object is a closure (i.e. a function with access to the defined `s_oliveAcids`, measures and separator) and returns the measures for all data points if no argument is used when evaluating the function

```
m1dc()

      median      irq  kurtosis  skewness
palmitic  -0.18234086 -1.5718396 -0.1863765  0.3422686
palmitoleic -0.30659302 -1.5525476 -0.5786855  0.4540451
stearic   -0.15962430 -1.1974439  1.5300760  0.9847540
oleic     -0.02278960 -1.6756601 -0.8835291  0.0762623
linoleic   0.20375695 -1.6886380 -1.1970407 -0.2087071
linolenic  0.08573630 -1.0987996  0.4774809 -0.5485431
arachidic  0.13173241 -0.9078426  0.9716094 -0.9785804
eicosenoic 0.05102012 -1.8461589 -1.1503986  0.3402334
```

The `m1dc` closure also accepts a logical vector to define the subset of data points that should be used to calculate the measures:

```
m1dc(Area == 'North-Apulia')

      median      irq    kurtosis    skewness
palmitic  -1.0720614 -0.7236394  0.7242277  0.5267205
palmitoleic -1.2590762 -0.3428939  0.2039190  0.5311321
stearic    0.1941660 -0.7620098 -0.2249252 -0.1174545
oleic      1.2524370 -0.5322685  1.7274993 -0.7850680
linoleic   -1.1553907 -0.3336090  1.8787465  0.8522058
linolenic  1.0110413 -1.0795225 -0.6024142 -0.5472931
arachidic  0.7672223 -0.7262741 -1.0290224 -0.2372025
eicosenoic 1.1871179 -0.7810672 -0.5531112  0.6626874
```

The `m1dc` measures object can be used for the `measures` argument of `l_ng_ranges`. The `data` argument is not necessary as the data is already accessible through the `m1dc` object (with `m1dc('data')`). Hence, the following code creates the `l_ng_ranges` setup shown in [Figure 7.9](#).

```
nav <- l_ng_ranges(measures=m1dc, color=Area)
```

Note that the control panel in [Figure 7.9](#) has the added “Recalculate Measures based on” section with the two buttons labeled “active” and “selected”. These buttons recalculate the measures for all active or selected points, respectively, in the scatterplot in [Figure 7.9](#).

Finally, we also provide the `scagnostics2d` function that creates a `measures` object for the scagnostic measures. For example, the following code shows the setups in [Figure 7.10](#).

```
scags <- scagnostics2d(oliveAcids, separator=':')
nav <- l_ng_plots(scags, color=Area)
```

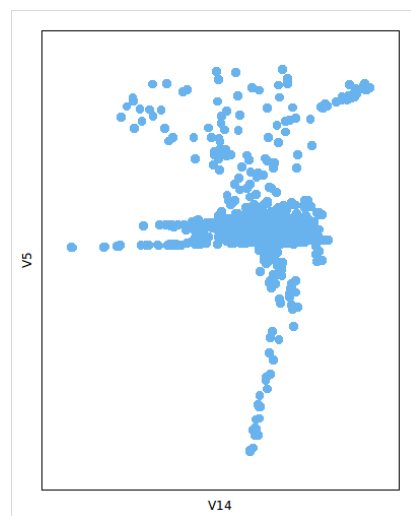
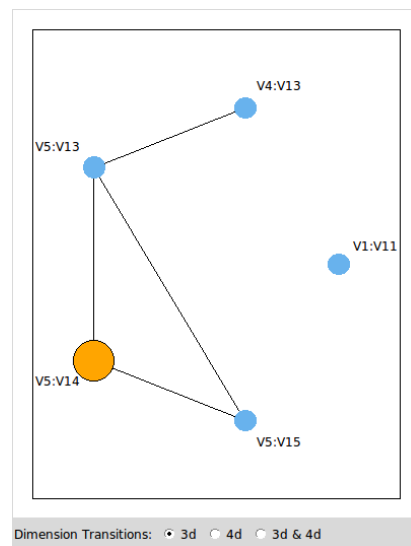
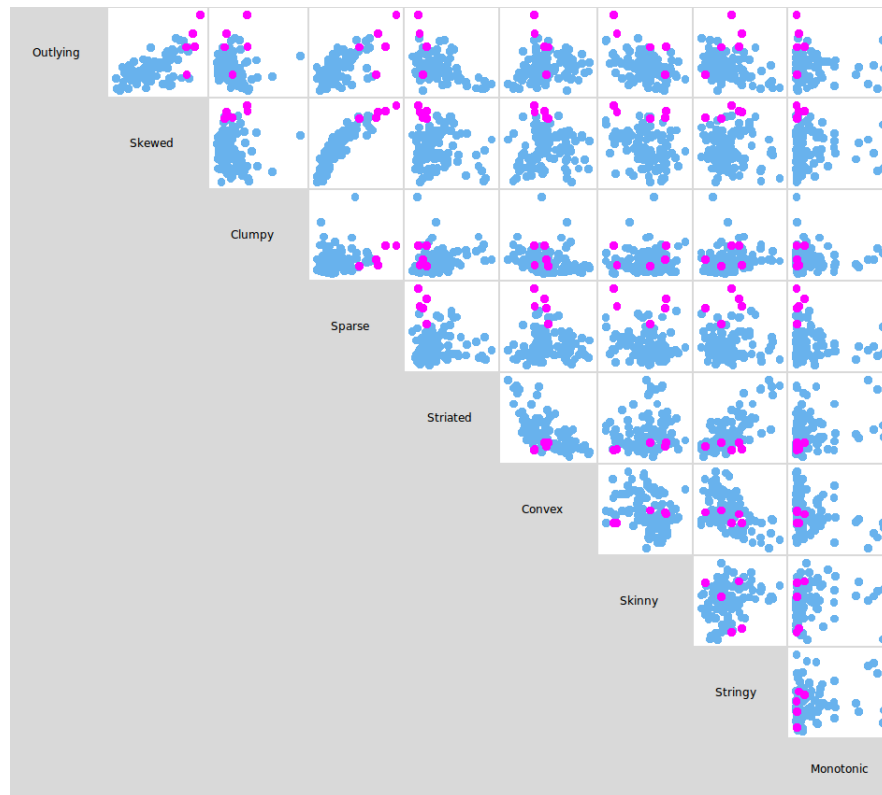
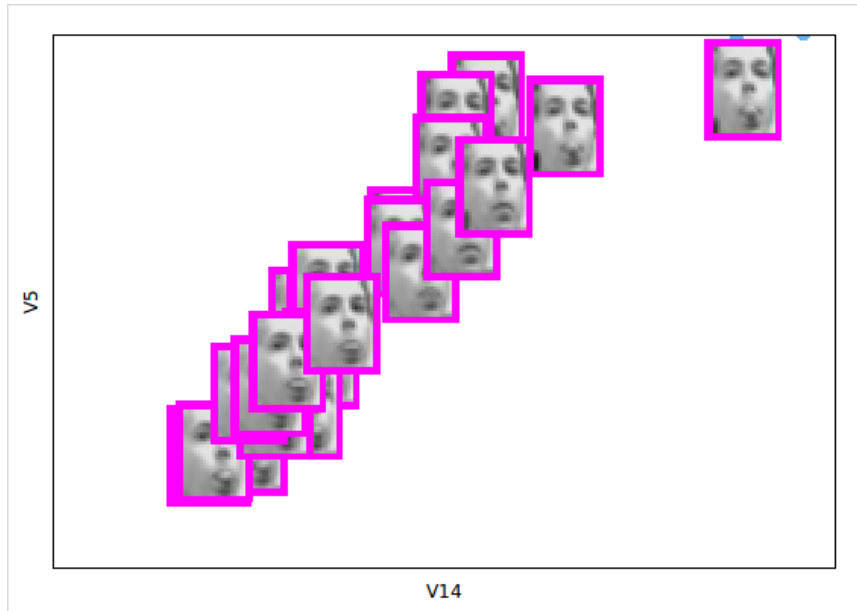
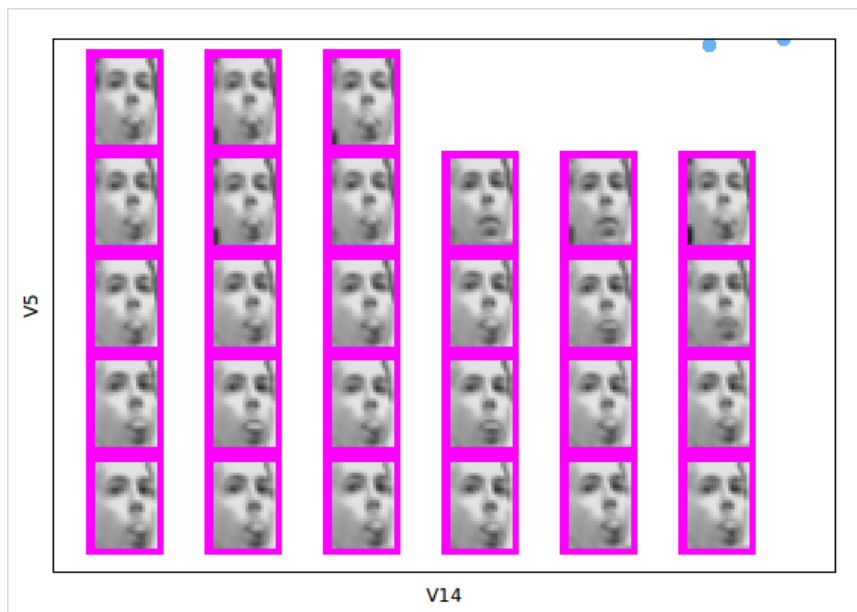


Figure 7.6: 1\_ng\_plots setup.

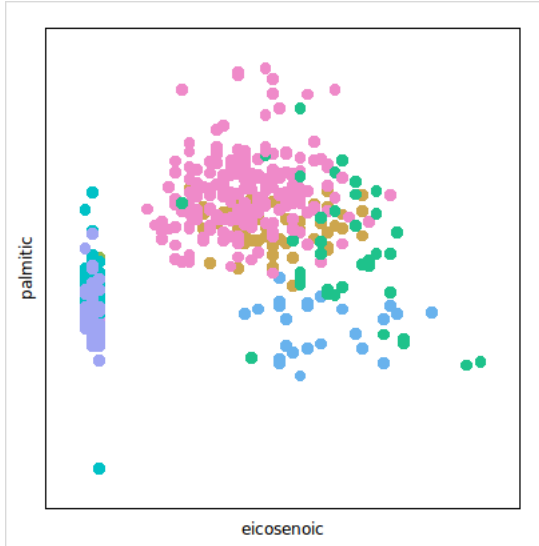


(a) from a “spike” of the LLE data

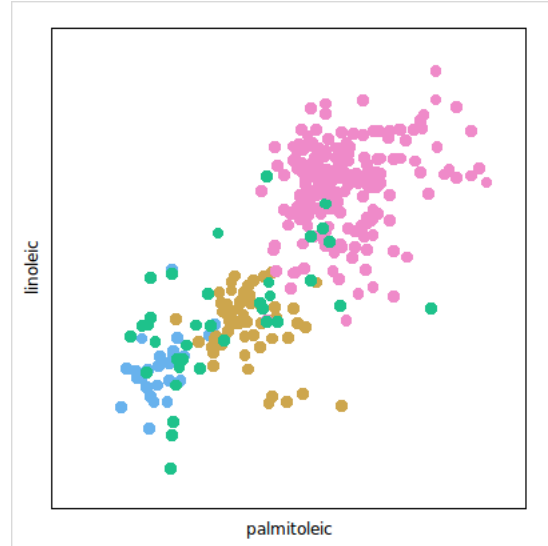


(b) arranged on a grid

Figure 7.7: Frey faces image glyphs.



(a) High clumpy for all points



(b) High clumpy for subset

Figure 7.8: Scatterplot of “clumpy” olive data.

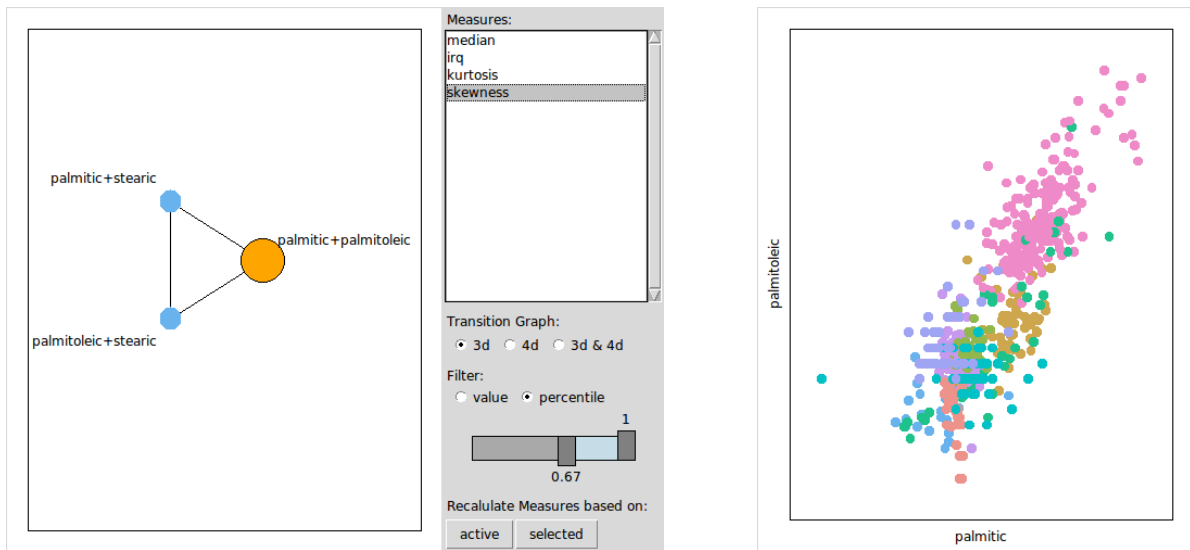


Figure 7.9: l\_ng\_ranges setup with measures1d.



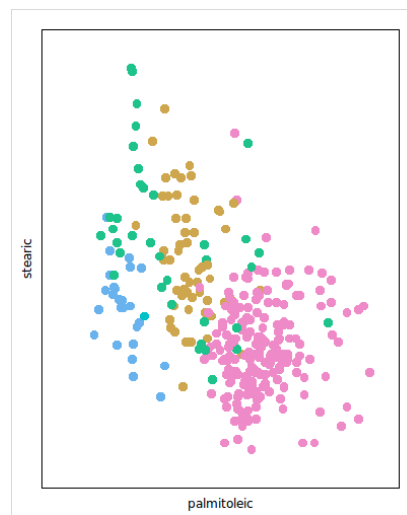
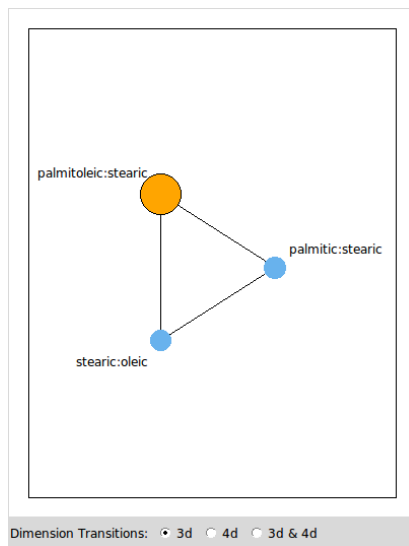


Figure 7.10: `l_ng_plots` for `scagnostics2d` and high sparse and low outlying points selected.

## 7.1.5 Exploring New Graph Semantics

In this section, we discuss the implementation of a new graph semantic with loon. We first give a high-level description for implementing a novel graph semantic and then provide a concrete example.

In general, to implement a new graph semantic, a graph widget with a navigation graph and navigator is needed to start. The following code creates a 3d transition graph named `G`, a graph widget `g3d` that visualizes `G` and finally adds a navigator to the graph widget.

```
G <- loongraph(nodes=c('A:B:C', 'A:B:E', 'A:C:D'), from = c('A:B:C', 'A:B:E'),
              to = c('A:B:E', 'A:C:D'), isDirected = FALSE)

g3d <- l_graph(G, x = c(0, 4, 1), y = c(3, 2, 1))
nav <- l_navigator_add(g3d)
```

The setting created by the above code is shown in [Figure 7.11\(a\)](#). The new semantic needs to be defined in terms of navigator changes or position on the transition graph. To do so, a function, for example, `mySemantic`, needs to be defined which is evaluated with every navigator state change (i.e. by using state change bindings, see [Subsection 5.2.2](#)). For now, assume that our navigation graph semantic of interest only prints the navigator location to the prompt:

```
mySemantic <- function() {
  cat(paste0('navigator is at ', tail(nav['from'], 1),
           ' --(', round(nav['proportion'], 2), ')-- ',
           nav['to'], '\n'))
}
l_bind_state(nav, 'all', function() mySemantic())
```

Note that the `mySemantic` function scopes for the navigator handle `nav` to print out the current navigator location. For the navigator location in [Figure 7.11\(a\)](#), the last message on the R prompt is

```
navigator is at A:B:C --(0.2)-- A:B:E
```

The above example can be used as a starting point to implement a new graph semantic. However, here are some further considerations. It should be possible to parse the spaces

from the graph nodes and, hence, also from the `from` and `to` navigator states. If the history of the navigator positions is required for the graph semantic then the `mySemantic` function (or whatever function is bound to the navigator state changes) needs to keep track of the previous navigator positions.

We now change the above setup (by overwriting the `mySemantic` function) to provide a more involved and novel graph semantic. The new graph semantic for the transition graph `g3d` drives a 2d space graph in a new graph widget `g2d` as follows. The nodes of `g3d` represent 3d spaces and the spaces are separated with the colon character. If the navigator is on an edge then the space of the navigator is defined to be the union of the spaces of the adjoining nodes. If the navigator is on a node then the space of the navigator is the space of the node. Then, our novel graph semantic creates a complete 2d space graph in `g2d` with nodes that representing all variable pairs of the navigator space. In addition, the colors of the nodes of `g2d` are controlled during an edge transition as follows

- in blue: the spaces in `g2d` that are present before and after the edge transition
- in white-gray-black: the spaces that fade in and out in `g2d` during the navigator transition on the edge from one node into another
- in yellow: the spaces that exist in `g2d` only when the navigator is on the edge

It is possible to extend this setup by adding a navigator and a `context2d` to the `g2d` graph to create a setting that shows the projections using the canonical graph semantic as in the other examples in this section.

```
g2d <- l_graph()
current_vars <- character()

mySemantic <- function() {
  space <- tail(nav['from'], 1)
  if (as.numeric(nav['proportion']) != 0)
    space <- c(space, head(nav['to'], 1))
  vars <- unique(unlist(strsplit(space, ':', fixed=TRUE)))

  if (!identical(vars, current_vars)) {
    # create new graph for g2d
    nodes <- apply(combn(vars, 2), 2, function(x) paste(x, collapse = ':'))
    G <- ndtransitiongraph(nodes, 3, ':')
```

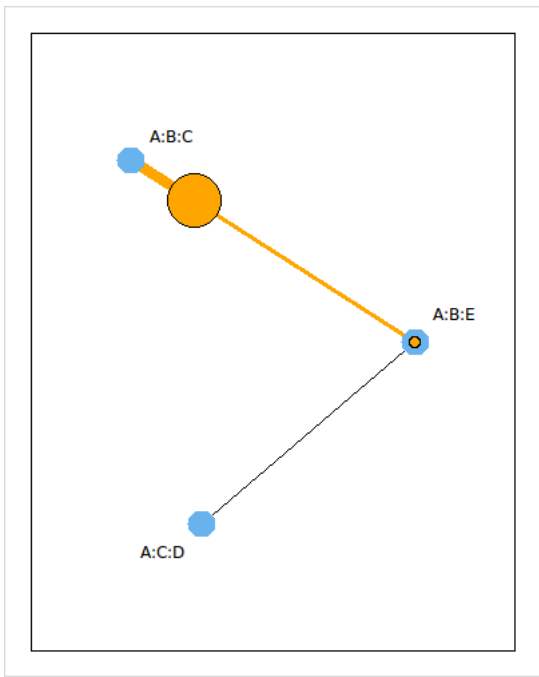
```

    l_configure(g2d, nodes=G$nodes, from=G$from, to=G$to)
    l_scaleto_world(g2d)
    l_zoom(g2d, 0.8)
    current_vars <- vars
}

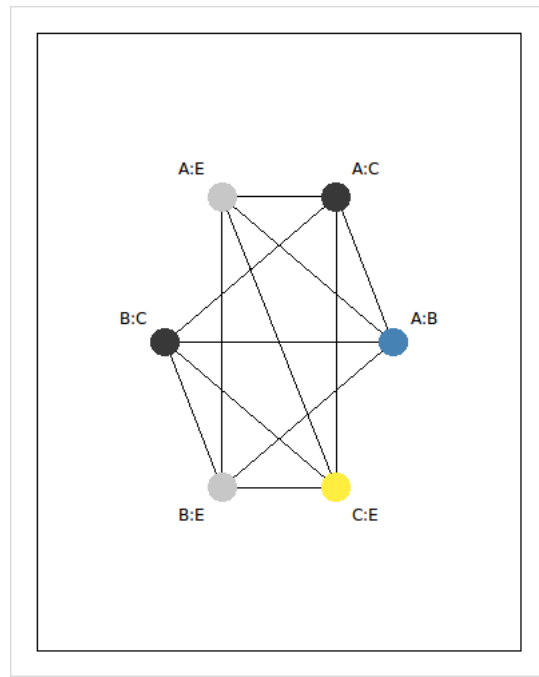
## Recolor recolor nodes of g2d
if(length(space) == 2) {
  node_spaces <- strsplit(g2d['nodes'], ':', fixed=TRUE)
  from_spaces <- unlist(strsplit(space[1], ':', fixed=TRUE))
  to_spaces <- unlist(strsplit(space[2], ':', fixed=TRUE))

  g2d['color'] <- sapply(node_spaces, function(x) {
    if(all(x %in% from_spaces) && all(x %in% to_spaces))
      'steelblue'
    else if(all(x %in% from_spaces))
      gray(nav['proportion'])
    else if(all(x %in% to_spaces))
      gray(1-nav['proportion'])
    else
      '#FFEE40' # pale yellow
  })
} else {
  g2d['color'] <- 'steelblue'
}
}
mySemantic()

```



(a) 3d space graph



(b) 2d space graph

Figure 7.11: Implementing a custom graph semantic.

## 7.2 Spiro Glyphs

Many of the methods discussed in this thesis involve an analyst tracking moving points on a scatterplot display. These methods include the grand tour [5], projection pursuit guided tour [23] and navigation graphs with the canonical graph semantic [44]. We propose to capture such point trajectories and to turn them into a point glyph called a *spiro glyph*. We now introduce two examples of spiro glyphs for the olive data.

Figure 7.12 shows the spiro glyphs that are obtained by recording the point locations on the projections when “walking” along a path on a 3d transition graph (using the canonical semantic). The particular path in Figure 7.12 is (a:l, l:s, s:p1, p1:a, a:l1, a:s, s:p, p:l1), where each element represents a 2d space defined by the concatenated variable names separated by a colon; here, we use the abbreviated variable names listed in Figure 1.1. We chose this path by randomly generating paths of length 8 on a saturated 3d transition graph and visualizing the spiro glyphs until we arrived at a plot that we found satisfying for visually differentiating the areas of origin (at least for some of the areas). Note that these glyphs are composed of lines that are either perpendicular or parallel to each other as, in a rigid 3d rotation, one variate stays the same.

Figure 7.13 shows the spiro glyphs that are obtained from “walking” along a path on the saturated 4d transitions graph for the olive oils. The path is (p1:p, a:s, p1:l, l1:s, a:e, p:l1, s:l, o:a) and was generated the same way as the path along the 3d transition graph above.

Spiro glyphs make for an interesting alternative to star glyphs and Chernoff faces [15]. They seem to be a promising tool for interactive visual clustering of high-dimensional data when combined with the navigation graph framework and with loon’s tools for interactive visualization. For example, Figure 7.14 shows a scatterplot of linolenic vs. linoleic with the spiro glyphs from Figure 7.13. With the two added dimension for  $x$  and  $y$  locations of the glyphs, it seems that the spiro glyphs shapes are able to help distinguish between the area groups better than when located on a grid. Note here that, for illustration purposes, we encoded the Area region with colors. Now, we could further explore the olive data space by controlling the scatterplot to show the projections according to a navigator position on a graph. Whenever we are not sure whether a point belongs to one group

or another, we could use the spiro glyphs as a visual aid. For example, in [Figure 7.15](#), we zoom in on the area of [Figure 7.14](#) where the gray glyphs are on a vertical line and intersect the brown glyphs arranged on the fourth horizontal line from the bottom. Now, loon's temporary arrangement tools are handy to move the glyphs from the center of [Figure 7.15](#) on a grid in order to compare them better, see [Figure 7.16](#). Alternatively, one could also stack the spiro glyphs onto the same  $x$  and  $y$  location for visual comparison.

Spiro glyphs deserve further investigation. For example, it would be interesting to work out the following research questions:

- how long should a trajectory be in order to obtain interesting shapes?
- what are the automated methods to generate interesting glyphs?
- how do spiro glyphs look for dimensions obtained from dimensionality reduction methods?
- how much visual information should be encoded in the spiro glyph visuals; for example, where do the trajectories start, where do they end, what is the geometric origin to distinguish the origin from the start?
- how do spiro glyphs change with different scalings, and why do some glyphs look like a “shrunk” version of another glyph (what does this mean geometrically in the high-dimensional space)?

spiro glyphs for 3d path: a:l l:s s:p1 p1:a a:l1 a:s s:p p:l1

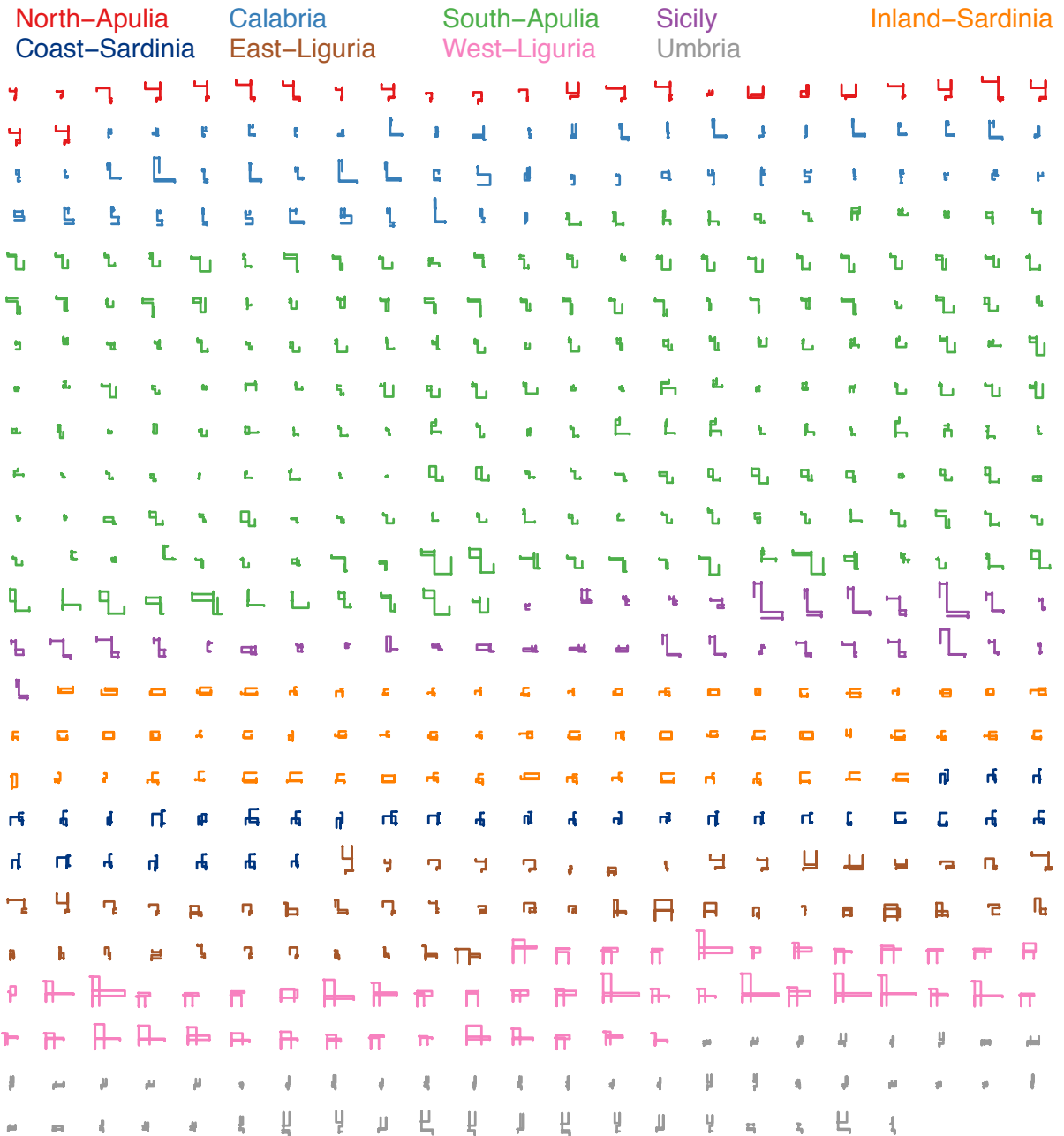


Figure 7.12: Spiro glyphs from 3d transitions arranged on a grid.



spiro glyphs for 4d path: p1:p a:s p1:l l1:s a:e p:l1 s:l o:a

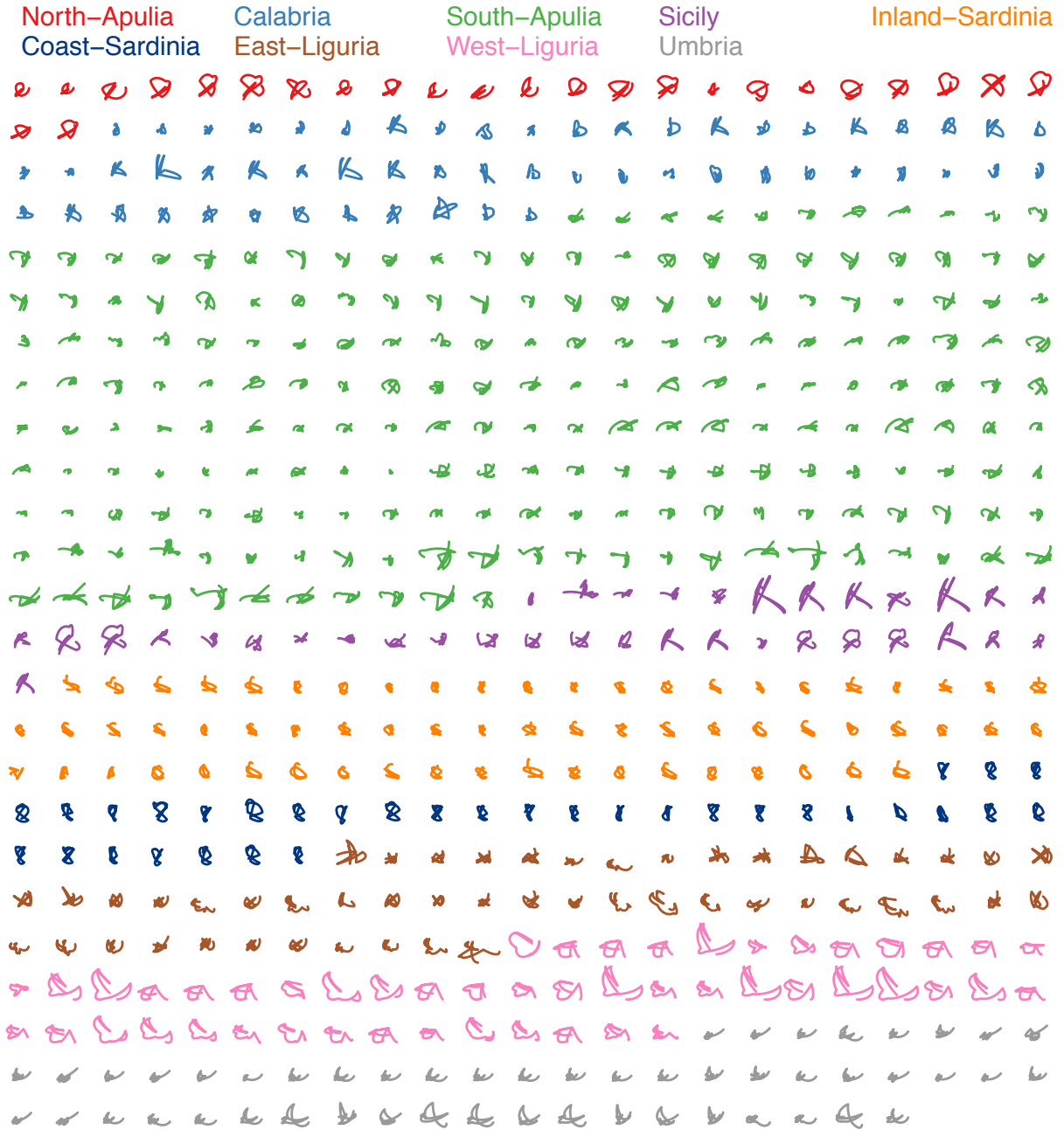


Figure 7.13: Spiro glyphs from 4d transitions arranged on a grid.

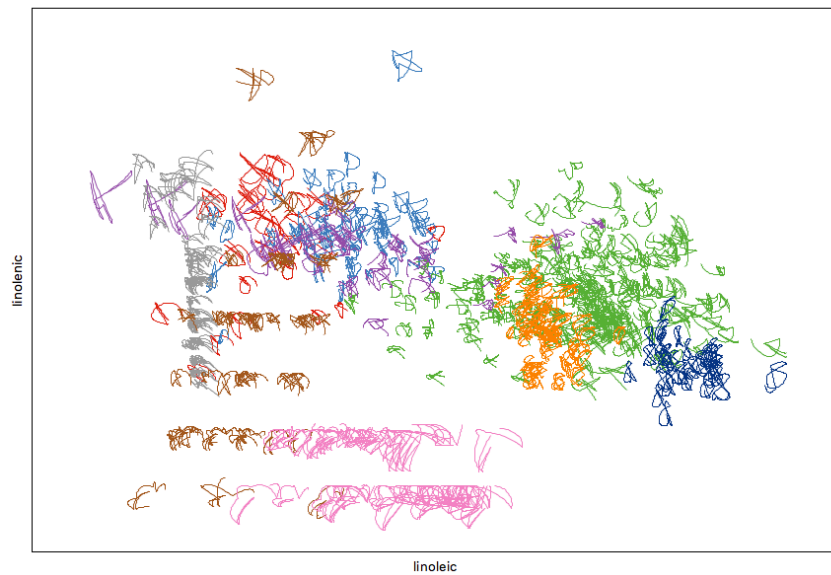


Figure 7.14: Spiro glyphs from 4d transitions on a scatterplot.

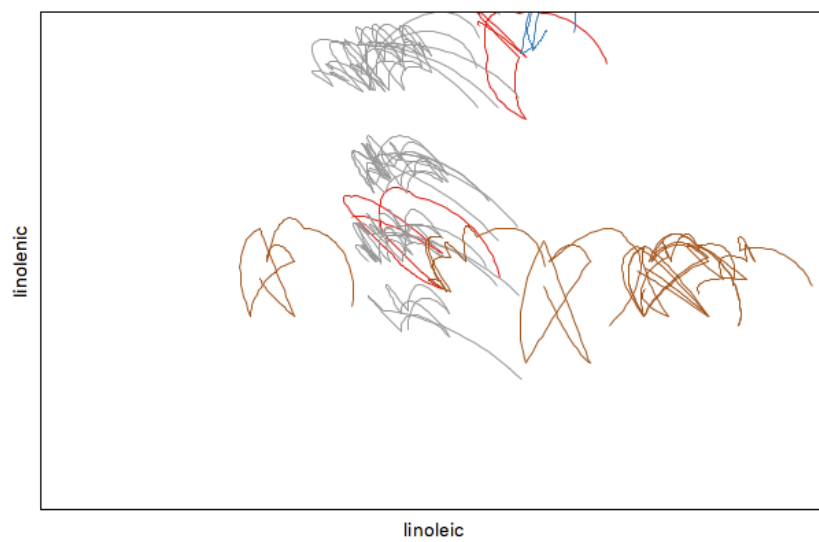


Figure 7.15: Zoomed in on spiro glyphs from [Figure 7.14](#).

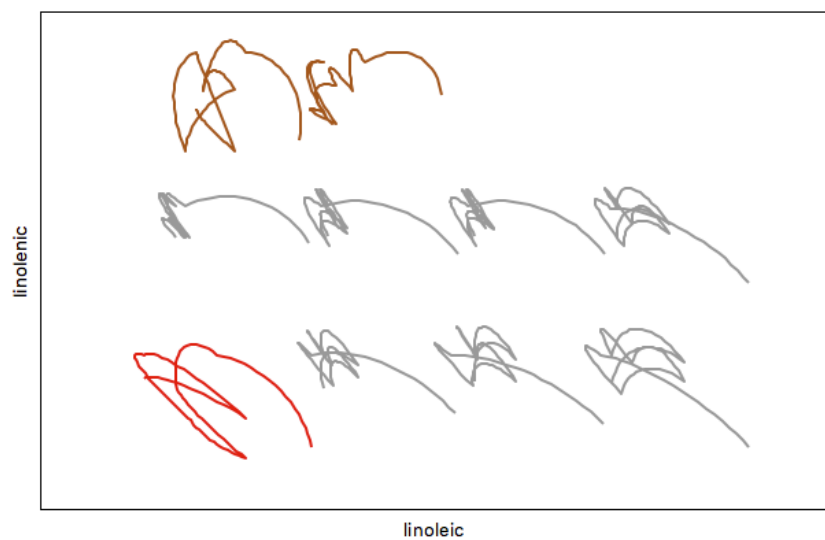


Figure 7.16: Temporarily arranged spiro glyphs in [Figure 7.15](#) on a grid with 100.

# Chapter 8

## Conclusions and Future Work

This thesis focuses on tools, methods and applied examples for interactive data visualization and exploration. With `RnavGraph`, we provide the first interactive environment to use navigation graphs to explore high-dimensional data. With `loon`, we provide a general-purpose interactive data visualization toolkit. We first introduce `loon` in [Chapter 3](#) by performing a visual exploratory analysis of Canadian census data. `loon`'s framework is discussed in [Chapter 4](#) and [Chapter 5](#). In [Chapter 6](#), we provide a number of examples where we enhance statistical methods with interactivity using `loon`. In [Chapter 7](#), we present new tools for efficiently exploring high-dimensional data using navigation graphs with `loon` in R. [Chapter 7](#) also discusses an implementation of a new navigation graph semantic with `loon` and proposes spiro glyphs as a novel high-dimensional point glyph.

In this chapter, we discuss `loon`'s place in the visualization landscape and propose some future work for expanding `loon`.

### 8.1 Conclusions for `loon`

With `loon`, we have created a comprehensive interactive data visualization environment that is useful for visualization novices as well as for visualization experts and researchers. For novices, is it easy to create an interactive scatterplot, histogram, serialaxes and graph display with little code and familiar syntax in both R and Tcl. Interactivity is made ac-

cessible with inspectors and common mouse gestures for zooming, panning and selection. The standard linking model is simple and powerful. For visualization experts, the loon framework, including layers and event bindings, can be used to customize and extend loon's functionality. This includes the creation of custom inspectors for particular visualization problems (e.g. time series and clustering). For visualization researchers, loon opens opportunities to test and implement new interactive displays and techniques with relatively little work, as we have shown with numerous examples in this thesis. For some problems it might be necessary to work with loon's Tcl source instead of loon's API. For example, to create a new widget with states and custom interactions we advise researchers to investigate loon's object oriented design. loon is open source with a GPL license (see <http://waddella.github.io/loon>), so everyone can contribute or make modifications to the source. In R, we made some functions such as `l_plot`, `l_graph` and `l_layer` generic function so that others can write specialized methods.

Although Di Cook's "challenges to the young developers", see [Section 1.6](#), were announced three years after we started working on loon, it is valuable to discuss how loon meets these challenges:

- **Interactivity on the plot:** All loon's displays support interactive selection. The displays based on the main graphic model (i.e. the scatterplot, histogram and graph display) further allow for zooming and panning. Custom interactions can be defined with canvas and item bindings.
- **Different types of brushes:** Depending on the display, loon provides rectangular sweep, rectangular brush, line sweep and point selections. We have also demonstrated how to implement a nearest neighbor highlighting.
- **Different kinds of linking between plots:** The standard linking model is simple and works for many linking situations. Any linking rule can be defined by using state change bindings.
- **Programmability:** loon is a toolkit, that is, its widgets can be arranged and modified as standard Tk widgets (e.g. buttons and sliders). loon's framework provides many ways to customize these widgets. If that is not sufficient then loon's Tcl source code can be studied and modified to add further features.

- **Strong connection with model fitting:** We have embedded loon in R which is a complete statistical computing environment. This makes the step from fitting a model to visualizing the fit a small one. With event bindings and callbacks it is further possible to connect a model with a visualization and the other way around. If the model fitting algorithm is reasonably fast then it is possible to create a real-time interaction with the model and visualization via mouse gestures, as we did with the simple linear regression setup in [Section 6.2](#).
- **Portability, easy install, web compatible:** loon is interpreted Tcl code that relies on Tcl and Tk version 8.6 only. That is, loon requires no compiling and no external libraries. This was a design decision that we made and that required additional effort. The benefit of this design decision is that it avoids complications when porting loon to other programming environments such as R and Python. We do provide an optional Tcl C extension for fast image resizing. Also, importing and exporting images of certain formats requires the Tcl Img extension.

As for portability, Tcl and Tk are platform independent and so is loon.

The loon R package is currently easy to install on OS X and Linux. Installing the loon R package on Windows presently requires an extra installation step. That is, R on Windows currently ships with Tcl and Tk version 8.5; hence, R needs to be linked against Tcl interpreter of version 8.6. We are working on compiling Tcl and Tk version 8.6 on Windows to ship it with future R Windows binaries. We plan to put the loon R package on CRAN once R on Windows ships with Tcl and Tk version 8.6. This will make loon easy to install for most R users. One reason that installing the loon R package is so easy is that the `tcltk` package is part of the R core distribution and the R binaries for Windows and OS X embed a Tcl and Tk binaries. Hence, creating a graphical user interface with Tcl and Tk in R often requires no extra setup efforts. This is not the case for other graphical user interface toolkits including `RGtk2` and `rJava` which require extra setup steps.

However, we have encountered issues with the default user interfaces for R in Windows and OS X. We recommend using loon with RStudio or by starting R from the Terminal. We hope that these issues are eventually resolved by the authors of the

Rgui app and the standard Windows R user interface.

loon's web compatibility depends on whether there is a web browser plugin for Tcl and Tk. There have been several approaches to do this, but to our knowledge there is presently no browser plugin that enables the evaluation of Tcl and Tk version 8.6 scripts that works on a variety of browsers and operating systems.

Another approach to bring loon's interactive displays to the web is to split loon into server-side and client-side code. loon's displays that are based on the main graphic model are based on a Model-View-Controller design. Hence, the model part of these displays could be run on the server-side and the view and controller part would have to be re-implemented on the client-side, for example, using the HTML5 canvas.

- **Large quantities of data:** Unwin et al. [78] state that for visualizing large data “the number 'a million' is a useful symbolic target”. 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. While designing loon, we often chose interaction features and user friendliness over speed. For example, maintaining a worldview of a plot requires the plot to be rendered twice. Also, the standard linking model is computationally expensive.

For us, working on a Linux computer with an Intel Core i5 4590 processor and 16 GB of RAM having two linked scatterplots each having 10,000 points was at the boundary of having a satisfactory experience while interactively zooming, panning and selecting. Working with two linked scatterplots each having 2,000 points results in a fluid interaction.

Non-interactive layers are less expensive to work with. For example, the plot with the world map, lakes, oceans, urban areas and international airports in [Figure 4.17](#) visualizes elements with 181,378 coordinates in total, with a satisfactory speed.

- **Incorporating inference:** loon's displays can be individually created, controlled and arranged using geometry managers. Hence, settings that are useful for visual inference can be easily created by the users (e.g. line up plots). Also, the scatterplot display supports temporarily moving point glyphs. This can be used to arrange the

point glyphs (e.g. spiro glyphs and star glyphs) on a grid or to stack them in order to compare them.

- **Conceptual framework:** loon has a conceptual framework that includes dimensional states, layers and event bindings, see [Chapter 4](#) and [Chapter 5](#).

In the end, we are happy to provide loon as the first comprehensive interactive visualization environment for R that makes interactive data visualization accessible and useful to a large community of data analysts that use R.

## 8.2 Future Work

Open ended systems like loon provide the opportunity for growth and development well into the future. With the completion of this thesis we make loon open source (i.e. public domain, see <http://waddella.github.io/loon>) and encourage others to participate in and contribute to the loon project. Others will have new ideas for interactive graphics that can be prototyped in loon and possibly incorporated into loon. In this section, we first focus on possible future directions of loon. We end this section by discussing interesting research directions for the navigation graphs paradigm.

### 8.2.1 loon in General

- **Tk desired improvements:**
  - **Alpha blending:** The Tk canvas widget, which is the foundation for all of loon’s displays, as of Tk version 8.6, does not support transparency for most canvas item types (the exception is image items). Transparency would be a useful feature for dealing with overplotting and visualizing uncertainty [57]. Hopefully, a future Tk release will add transparency to all canvas item types. Otherwise, there is a project called tkpath that provides a canvas alternative that supports alpha blending. However, the original author of tkpath apparently passed away. We have tried to use the tkpath canvas in its current state



but we were not satisfied. Also, using tkpath would make loon dependent on software other than Tcl and Tk.

- **Anti Aliasing:** The Tk canvas widget does not anti alias its items (e.g. visuals) except for some Tcl and Tk distributions for OS X (not the one that ships with R). This results in having the plots look pixelated. This is an aesthetic issue only. As with alpha blending, anti aliasing could become a future feature of the Tk canvas widget. Otherwise, the tkpath project uses anti aliasing for its visuals.
- **Clipping:** Clipping is a method to render visuals in a certain region only. For example, we are clipping the points and other layer visuals in a scatterplot that would otherwise fall into the scales and labels region. We clip by drawing filled rectangles at the borders. More sophisticated clipping such as clipping a group of canvas items that fall outside an arbitrary shape (e.g. defined by another canvas item) would be useful for a richer plot design. Clipping would be also useful to cut “holes” into visuals. This would be useful, for example, when plotting maps (e.g. when subtracting a lake from a polygon of a country).
- **Make every layer type interactive:** In loon, the model layer has a great deal of interactivity including selection, moving points in the scatterplot and graph display, interacting with the bin handle in the histogram and moving the navigator on a graph. It would be interesting to make every layer type as interactive as the model layer. We have designed loon’s object model with interactivity for every layer in mind. The widget could then get an `activelayer` state to indicate which layer should receive mouse and keyboard gestures. This would also make it possible to create a general 2d plot container without a model layer. The layers for a scatterplot, histogram and graph would then become ordinary layers and could be stacked on the same display. This would also avoid mixing model layer states (e.g. `x`, `y` and `selected`) with container states (e.g. `zoomX`, `panX` and `showScales`). Note that making all layers interactive also requires an analysis inspector for every layer type.
- **Linkable layers and linkable arbitrary dimensional states:** Currently the standard linking model applies to  $n$ -dimensional states of model layers only. Other

abstract dimensions such as  $p$ , see [Subsection 4.3.1](#), are presently not supported in the standard linking model. However, assuming that the  $p$  dimensional states share the linking group (i.e. using the `linkingGroup` state) with the  $n$  dimensional states, then for any other dimensional states, e.g.  $p$  dimensional states, an addition of linking key state for that dimension, for example `linkingKey_p`, would fit into the loon framework.

Adding linking to  $n$  dimensional states for layers is also a possible modification of the loon framework. loon's object design is such that layer objects could be made to inherit from the `Linkable` class in order to add the states `linkingGroup` and `linkingKey` to make the layer part of the standard linking model. However, it is unclear to us whether the added functionality is worth the added complexity and possible performance consequences.

It should be noted that any linking for layers and arbitrary abstract dimensions can be implemented by the user with state change bindings, see [Section 5.3](#).

- **More sophisticated event patterns for state change bindings:** The canvas and item bindings support logic in their event patterns. This is due to the fact that we delegate the event pattern to the Tk canvas which supports this logic. For the state change bindings, we currently only support a vector with state names and, if any of the states in that vector get modified in the configure pipeline, then the callback for that binding will be evaluated. Supporting logic expressions for state change event patterns would be a good further feature. For example,

```
l_bind_state(p, event='(selected&!active)|color', callback=foo)
```

would evaluate the callback function `foo` if the `color` state in `p` is modified or the `selected` state is modified but the `active` state is not modified in a configure call. Note that is not supported in the current loon model but could be added. Again, the functionality would have to be weighed against any performance consequences.

- **Deal with missing values:** One improvement would be to accommodate missing values (e.g. NA in R) for plot states. The obvious way to deal with missing values is to not plot them. Alternatively, an interesting research problem is to figure out how missing values might actually be displayed.

- **Embedding loon in Python:** Python is an interesting environment for conducting data analysis [52]. Embedding loon in Python would require similar work as we demonstrated with the loon R package.
- **Context specific menus:** We chose to work with inspectors rather than context specific menus. However, having both might be better. One possible first step towards implementing context specific menus would be to re-use loon’s modular inspectors and have pieces of the inspectors pop up as context menus.
- **Annotate Tab for Inspectors:** Annotation is used to add information to a plot (e.g. text, rectangle, drawings). In loon, annotation could be implemented by interactively adding layers to a display. To that end, it would be useful to add an “Annotate” tab to inspectors of displays that support layering.

## 8.2.2 Current Displays

- **Histogram:** The analysis inspector lacks a widget for interactively choosing the color stacking order.
- **Serialaxes Display:** The serialaxes display would benefit from zoom and pan support.
- **Scatterplot:** Next to layers and glyphs, the scatterplot could also use *segments* to connect an ordered sequence of points.
- **Compound Glyphs:** Providing layered glyphs would allow for more sophisticated point glyphs such as Chernoff faces. However, it should be noted that since image glyphs support transparency, any point glyph can be implemented in loon with image glyphs.
- **Graph Display:** If linking support is added for  $p$  dimensional states then it would be desirable to add interactive selection and modifications of graph edges. This would also require an analysis inspector for graph edges.

- **Graph Inspector:** The graph inspector could use a navigator and context inspector. The navigator inspector could benefit from a path tool such as in [Figure 2.2](#).

### 8.2.3 New Displays

- **Scatterplot Matrix:** `loon` in R currently implements a scatterplot matrix with the `l_pairs` function by laying out individual scatterplot widgets, see [Section 5.4](#). A special purpose scatterplot matrix widget addition to `loon` would be a worthwhile effort. There are a number of drawbacks when using individual scatterplot widgets for a scatterplot matrix. For example, a scatterplot matrix with  $p$  variables has presently  $\binom{p}{2}$  scatterplots widgets and every widget has its own color, selected, active, etc. states which need to be synchronized among the scatterplots in the scatterplot matrix. This is computationally expensive and also prone to inconsistency; for example, in the case where the analyst changes the linking group of a plot manually. A custom scatterplot matrix widget would have only one color, selected, active, etc. state. It would also be interesting to design a custom scatterplot matrix inspector. Finally, changing the data shown in the scatterplot matrix currently requires one to change the individual scatterplots widgets manually. If the number of variables changes then one has to use the geometry manager to add or remove scatterplots.
- **Cross tables:** We have experimented with a color cross table widget that has the two  $n$  dimensional states color and group. The widget then visualizes the cross table of color vs. group as seen and explained in detail in [Figure 8.1](#). Another possibility would be to create a cross table widget for a cross table of two group states, for example, `group_x` and `group_y`, and visualize the color state in the table cells.
- **Other displays:** There are popular and useful data displays that are presently not implemented in `loon`. These include the boxplot, dotplot, bar chart, mosaic plot and eikosogram display [16]. `loon` would benefit from these displays being carefully designed and added to the toolkit.

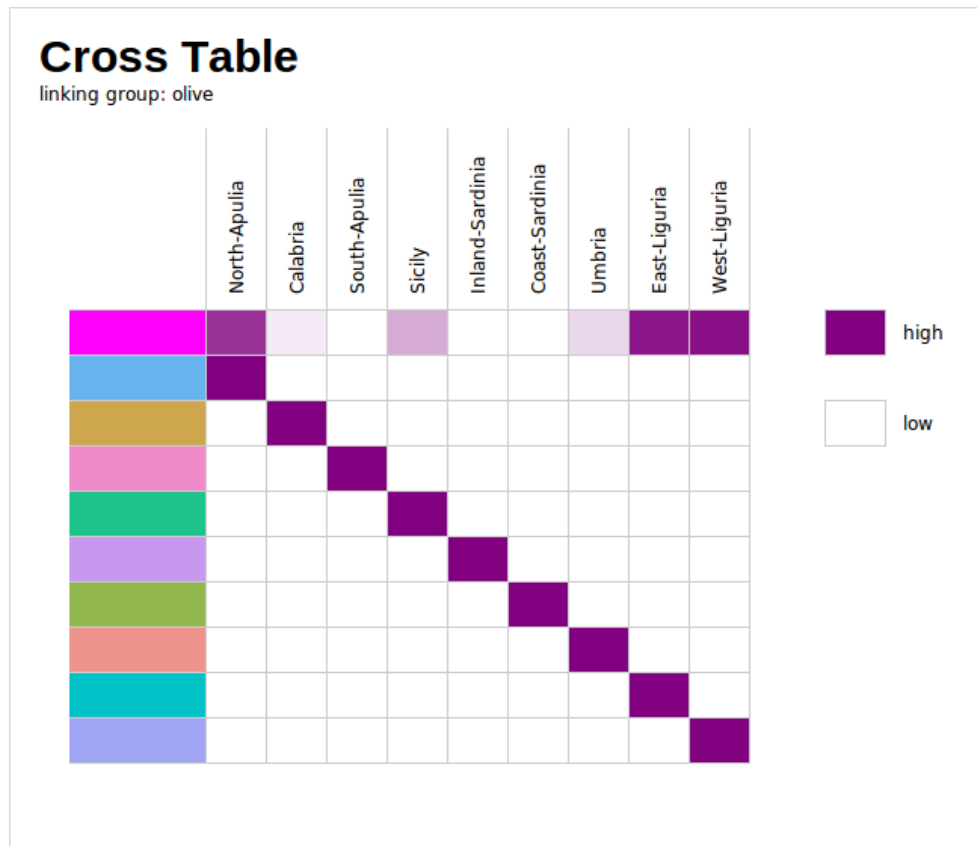


Figure 8.1: Example of a color cross table of the plot states group vs. color. Note that the first row represents the selected state and not the color state. The data used here is the olive data and its Area variable is assigned to the group state. Also, the color state has a different color for each area in Area. Here, the color of a cell  $(i, j)$  is according to the fraction of points that have the color from row  $i$  out of the points that are from the group in column  $j$ . For the first row the color of a cell represents the fraction of points that are selected for a given group.

## 8.2.4 Navigation Graphs

- The two functions `l_ng_ranges` and `l_ng_plots` could be generalized to work with contexts other than the `geodesic2d` context.
- The closures of measures in [Subsection 7.1.4](#) update some measures for subsets

of the data. It would be interesting to create a closure for recalculating both the measures and data. That is, if the data are derived from a dimensionality reduction method then one could re-calculate the embedding of the dimensionality reduction method for a subset of the points and embed the remaining out-of-sample points if possible (e.g. for a projection based methods such as principal components). Re-calculating the embedding would trigger a recalculation of the measures which would update the graph based on the measures. For dimensionality reduction methods, where embedding of out-of-sample points is not possible (e.g. LLE and isomap), one would have to continue working with the subset of the data only.

# References

- [1] Allison, T. and Cicchetti, D. V. (1976). “[Sleep in mammals: ecological and constitutional correlates](#)”. *Science* 194.4266, pp. 732–734.
- [2] American Statistical Association (2015). “[Video Library Statistics Computing and Graphics](#)”.
- [3] Anderson, E. (1935). “The irises of the Gaspé Peninsula”. *Bulletin of the American Iris society* 59, pp. 2–5.
- [4] Anglin, D. and Oldford, R. (1994). “[Modelling response models in software](#)”. *Selecting Models from Data*. Springer, pp. 413–424.
- [5] Asimov, D. (1985). “[The Grand Tour: A Tool for Viewing Multidimensional Data](#)”. *SIAM Journal of Scientific and Statistical Computing* 6.4, pp. 128–143.
- [6] Baudat, G. and Anouar, F. (2000). “[Generalized discriminant analysis using a kernel approach](#)”. *Neural computation* 12.10, pp. 2385–2404.
- [7] Becker, R. A., Cleveland, W. S., and Wilks, A. R. (1987). “[Dynamic graphics for data analysis](#)”. *Statistical Science*, pp. 355–383.
- [8] Becker, R. A., Wilks, A. R., Brownrigg, R., and Minka, T. P. (2015). *maps: Draw Geographical Maps*. R package version 2.3-11.
- [9] Bellman, R. (1961). *Adaptive Control Processes: A Guided Tour*. Rand Corporation Research studies. Princeton University Press.
- [10] Bivand, R. S., Pebesma, E. J., Gomez-Rubio, V., and Pebesma, E. J. (2008). *Applied spatial data analysis with R*. Vol. 747248717. Springer.
- [11] Box, G. E. and Cox, D. R. (1964). “[An analysis of transformations](#)”. *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 211–252.

- [12] Buja, A. and Asimov, D. (1986). “Grand Tour Methods: An Outline”. *Computing Science and Statistics* 17, pp. 63–67.
- [13] Buja, A., Swayne, D. F., Littman, M. L., Dean, N., and Hofmann, H. (2001). “XGvis: Interactive Data Visualization with Multidimensional Scaling”.
- [14] Carr, D. B., Littlefield, R. J., Nicholson, W. L., and Littlefield, J. S. (1987). “Scatterplot Matrix Techniques for Large N”. *Journal of the American Statistical Association* 82.398, pp. 424–436.
- [15] Chernoff, H. (1973). “The use of faces to represent points in k-dimensional space graphically”. *Journal of the American Statistical Association* 68.342, pp. 361–368.
- [16] Cherry, W. and Oldford, R. (2006). “Picturing Probability: the poverty of Venn diagrams, the richness of Eikosograms”.
- [17] Cleveland, W. S. (1993). *Visualizing data*. Hobart Press.
- [18] Cleveland, W. S. and McGill, R. (1984). “Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods”. English. *Journal of the American Statistical Association* 79.387, pp. 531–554.
- [19] Cleveland, W. S. and McGill, R. (1985). “Graphical Perception and Graphical Methods for Analyzing Scientific Data”. English. *Science*. New Series 229.4716, pp. 828–833.
- [20] Cleveland, W. S. and McGill, R. (1986). “An experiment in graphical perception”. *International Journal of Man-Machine Studies* 25.5, pp. 491–500.
- [21] Cleveland, W. S. and McGill, R. (1987). “Graphical Perception: The Visual Decoding of Quantitative Information on Graphical Displays of Data”. English. *Journal of the Royal Statistical Society. Series A (General)* 150.3, pp. 192–229.
- [22] Cook, D. (2015). “Stories of Two Decades of Efforts to Build Interactive Graphics Capacity into R.” The useR! Conference 2015.
- [23] Cook, D., Buja, A., Cabrera, J., and Hurley, C. (1995). “Grand Tour and Projection Pursuit”. *Journal of Computational and Graphical Statistics* 4.3, pp. 155–172.
- [24] Dalgaard, P. (2001). “The R-Tcl/Tk interface”. *Proceedings of DSC*.
- [25] Elmqvist, N., Dragicevic, P., and Fekete, J.-D. (2008). “Rolling the Dice: Multidimensional Visual Exploration using Scatterplot Matrix Navigation”. *Visualization and Computer Graphics, IEEE Transactions on* 14.6, pp. 1539–1148.



- [26] Fisher, R. A. (1936). “The Use of Multiple Measurements in Taxonomic Problems”. *Annals of Eugenics* 7.2, pp. 179–188.
- [27] Fisher, M., Friedman, J., and Tukey, J. (1974). “Prim-9: An Interactive Multidimensional Data Display and Analysis System”. Stanford Linear Accelerator Center.
- [28] Flynt, C. (2012). *Tcl/Tk: A Developer’s Guide*. Morgan Kaufmann.
- [29] Forina, M., Armanino, C., Lanteri, S., and Tiscornia, E. (1983). “Classification of olive oils from their fatty acid composition”. *Food Research and Data Analysis*, pp. 189–214.
- [30] Friedman, J. (1987). “Exploratory Projection Pursuit”. English. *Journal of the American Statistical Association* 82.397, pp. 249–266.
- [31] Friedman, J. and Stuetzle, W. (1981). “Projection pursuit regression”. *Journal of the American statistical Association*, pp. 817–823.
- [32] Friedman, J. and Tukey, J. (1974). “A Projection Pursuit Algorithm for Exploratory Data Analysis”. *Computers, IEEE Transactions on C-23.9*, pp. 881 –890.
- [33] Fu, L. and Oldford, W. (2009). “Implementation of Three-dimensional Scagnostics”. MA thesis. University of Waterloo.
- [34] Gabriel, K. (1971). “The biplot graphic display of matrices with application to principal component analysis”. *Biometrika* 58.3, pp. 453–467.
- [35] Gapminder Foundation (2015). “Gapminder data”.
- [36] Gentleman, R., Whalen, E., Huber, W., and Falcon, S. (2015). *graph: A package to handle graph data structures*. R package version 1.46.0.
- [37] Gnanadesikan, R. (2011). *Methods for Statistical Data Analysis of Multivariate Observations*. Wiley Series in Probability and Statistics. Wiley.
- [38] Huber, P. J. (1985). “Projection Pursuit”. *The Annals of Statistics* 13.2, pp. 435–475.
- [39] Hurley, C. B. and Oldford, R. W. (2010). “Pairwise Display of High-Dimensional Information via Eulerian Tours and Hamiltonian Decompositions”. *Journal of Computational and Graphical Statistics* 19.4, pp. 861–886.
- [40] Hurley, C. (1987). *The Data Viewer: A Program for Graphical Data Analysis*. Tech. rep. 115. Seattle, Washington 98195 USA: Department of Statistics, GN-22, University of Washington.

- [41] Hurley, C. and Buja, A. (1990). “Analyzing high-dimensional data with motion graphics”. *SIAM Journal on Scientific and Statistical Computing* 11.6, pp. 1193–1211.
- [42] Hurley, C. and Oldford, R. (1988). “Higher Hierarchical Views of Statistical Objects”. Video.
- [43] Hurley, C. and Oldford, R. (1999). “Statistical Graphics in QUAIL: An Overview”. Vol. 58. ISI, pp. 113–116.
- [44] Hurley, C. and Oldford, R. (2011). “Graphs as navigational infrastructure for high dimensional data spaces”. English. *Computational Statistics* 26.4, pp. 585–612.
- [45] Ihaka, R. (2003). “Colour for presentation graphics”. *Proceedings of DSC*, p. 2.
- [46] Inselberg, A. and Dimsdale, B. (1991). “Parallel Coordinates”. *Human-Machine Interactive Systems*. Ed. by A. Klinger. Languages and Information Systems. Springer US, pp. 199–233.
- [47] Lang, D. T., Swayne, D., Wickham, H., and Lawrence, M. (2014). *rggobi: Interface between R and GGobi*. R package version 2.1.20.
- [48] Lee, J. and Verleysen, M. (2007). *Nonlinear dimensionality reduction*. Springer Verlag.
- [49] Maaten, L. Van der, Postma, E., and Herik, J. van den (2009). *Dimensionality reduction: A comparative review*. Tech. rep. 005. Tilburg center for Cognition and Communication.
- [50] McConnell, S. (2004). *Code Complete*. Microsoft Press.
- [51] McDonald, J. A. and Pedersen, J. (1985). “Computing Environments for Data Analysis I. Introduction”. *SIAM Journal on Scientific and Statistical Computing* 6.4, pp. 1004–1012.
- [52] McKinney, W. (2012). *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O’Reilly Media, Inc.
- [53] Mosteller, F. and Tukey, J. W. (1977). “Data analysis and regression: a second course in statistics.” *Addison-Wesley Series in Behavioral Science: Quantitative Methods*.
- [54] Murrell, P. (2005). *R graphics*. Chapman and Hall/CRC.
- [55] Natural Earth (2015). “Free vector and raster map data at 1:10m, 1:50m, and 1:110m scales”.
- [56] Neuwirth, E. (2014). *RColorBrewer: ColorBrewer Palettes*. R package version 1.1-2.

- [57] Oldford, R. (2015). “[Visualizing uncertainty: problems and solutions](#)”. *Data Meets Viz Workshop*. Kloster Holzen, Augsburg.
- [58] Oldford, R. and Peters, S. (1988). “[DINDE: Towards more sophisticated software environments for statistics](#)”. *SIAM Journal on Scientific and Statistical Computing* 9.1, pp. 191–211.
- [59] Pearson, K. (1901). “On lines and planes of closest fit to systems of points in space”. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11, pp. 559–572.
- [60] R Core Team (2015). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria.
- [61] Rao, C. (1952). *Advanced statistical methods in biometric research*. Wiley publications in statistics. Wiley.
- [62] Roweis, S. T. and Saul, L. K. (2000). “[Nonlinear Dimensionality Reduction by Locally Linear Embedding](#)”. *Science* 290.5500, pp. 2323–2326.
- [63] Sarkar, D. (2008). *Lattice: multivariate data visualization with R*. Springer-Verlag New York.
- [64] Schölkopf, B., Smola, A., and Müller, K.-R. (1997). “[Kernel principal component analysis](#)”. *Artificial Neural Networks - ICANN'97*. Ed. by W. Gerstner, A. Germond, M. Hasler, and J.-D. Nicoud. Vol. 1327. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 583–588.
- [65] Sheil, B. (1983). *Power Tools for Programmers*. Palo Alto Research Center.
- [66] Statistics Canada (2006). “[Ethnic diversity and immigration: Visible minorities](#)”. Address.
- [67] Stuetzle, W. (1987). “[Plot Windows](#)”. *Journal of the American Statistical Association* 82.398, pp. 466–475.
- [68] Swayne, D. F., Cook, D., and Buja, A. (1991). “Xgobi: Interactive Dynamic Graphics In The X Window System With A Link To S”.
- [69] Swayne, D. F., Lang, D. T., Buja, A., and Cook, D. (2003). “[GGobi: evolving from XGobi into an extensible framework for interactive data visualization](#)”. *Computational Statistics & Data Analysis* 43.4, pp. 423 –444.

- [70] Switz, D., Asimov, D., Donoho, D., Zelen, M., and Huber, P. (1982). “[Evaluation of large clinical datasets using an interactive graphics system \(PRIM/H\)](#)”. *Controlled Clinical Trials* 3.2, pp. 148–149.
- [71] Talbot, J., Lin, S., and Hanrahan, P. (2010). “[An Extension of Wilkinson’s Algorithm for Positioning Tick Labels on Axes](#)”. *Visualization and Computer Graphics, IEEE Transactions on* 16.6, pp. 1036–1043.
- [72] Tcl/Tk core team (2015a). “[Bind manual page](#)”.
- [73] Tcl/Tk core team (2015b). “[Canvas manual page](#)”.
- [74] Tenenbaum, J. B., Silva, V. d., and Langford, J. C. (2000). “[A Global Geometric Framework for Nonlinear Dimensionality Reduction](#)”. *Science* 290.5500, pp. 2319–2323.
- [75] Theus, M. (2002). “[Interactive Data Visualization using Mondrian](#)”. *Journal of Statistical Software* 7.11, pp. 1–9.
- [76] Tierney, L. (2009). *Lisp-stat: an object-oriented environment for statistical computing and dynamic graphics*. Vol. 353. John Wiley & Sons.
- [77] Tukey, J. W. and Tukey, P. A. (1988). “[Computer graphics and exploratory data analysis: An introduction](#)”. *The Collected Works of John W. Tukey: Graphics: 1965-1985* 5, p. 419.
- [78] Unwin, A., Theus, M., and Hofmann, H. (2006). *Graphics of large datasets: visualizing a million*. Springer Science & Business Media.
- [79] Urbanek, S. and Theus, M. (2003). “[iPlots: high interaction graphics for R](#)”. *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*.
- [80] Velleman, P. F. and Velleman, A. Y. (1988). “[Data Desk](#)”.
- [81] Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.
- [82] Wickham, H. (2015). *scales: Scale Functions for Visualization*. R package version 0.2.5.
- [83] Wilkinson, L., Anand, A., and Grossman, R. (2005). “[Graph-theoretic scagnostics](#)”. *Information Visualization, 2005. INFOVIS 2005. IEEE Symposium on*, pp. 157–164.

- [84] Wilkinson, L. and Anand, A. (2012). *scagnostics: Compute scagnostics - scatterplot diagnostics*. R package version 0.2-4.
- [85] Wilkinson, L. and Wills, G. (2008). “[Scagnostics Distributions](#)”. *Journal of Computational and Graphical Statistics* 17.2, pp. 473–491.

# **APPENDICES**

# Appendix A

## R Code for Chapter 6 Examples

### A.1 Power Transformations

```
1 power <- function(x, y, from=-5, to=5, ...) {
2
3   tt <- tktoplevel()
4   tktitle(tt) <- "Box-Cox Power Transformation"
5   p <- l_plot(x=x, y=y, parent=tt, ...)
6   lambda_x <- tclVar('1')
7   lambda_y <- tclVar('1')
8   sx <- tkyscale(tt, orient='horizontal',
9                 variable=lambda_x, from=from, to=to, resolution=0.1)
10  sy <- tkyscale(tt, orient='vertical',
11                variable=lambda_y, from=to, to=from, resolution=0.1)
12
13  tkgrid(sy, row=0, column=0, sticky="ns")
14  tkgrid(p, row=0, column=1, sticky="nswe")
15  tkgrid(sx, row=1, column=1, sticky="we")
16  tkgrid.columnconfigure(tt, 1, weight=1)
17  tkgrid.rowconfigure(tt, 0, weight=1)
18
19  powerfun <- function(x, lambda) {
20    if (lambda == 0)
21      log(x)
22    else
23      (x^lambda-1)/lambda
24  }
25
26  update <- function(...) {
```

```

27     l_configure(p,
28         x = powerfun(x, as.numeric(tclvalue(lambda_x))),
29         y = powerfun(y, as.numeric(tclvalue(lambda_y))))
30     l_scaletoworld(p)
31 }
32
33 tkconfigure(sx, command=update)
34 tkconfigure(sy, command=update)
35
36 invisible(p)
37 }

```

## A.2 Interactively Adding Regression Lines

```

1 addRegressionLinesGUI <- function(p) {
2     force(p)
3     addRegressionLine <- function() {
4         sel <- p['selected']
5         if (sum(sel)==0) return()
6         xs <- p['x'][sel]; ys <- p['y'][sel]
7         fit <- lm(ys ~ poly(xs, as.numeric(tclvalue(degree))))
8         xrng <- seq(min(xs), max(xs), length.out = 20)
9         ypred <- predict(fit, newdata=data.frame(xs = xrng))
10        l_layer_line(p, x=xrng, y=ypred, color=as.character(color),
11                    linewidth = 4, index=0,
12                    label=paste("degree", tclvalue(degree)))
13        l_configure(p, color=color, glyph='ocircle', which=sel)
14    }
15
16    updateColor <- function() {
17        col <- as.character(tcl('tk_chooseColor', initialcolor=color))
18        if (col!='') {
19            tkconfigure(b_col, bg=col, activebackground=col)
20            color <- col
21        }
22    }
23
24    tt <- tkoplevel()
25    tktitle(tt) <- 'Add Regression Line'
26    degree <- tclVar('1')
27    color <- 'red'
28    s <- tkScale(tt, orient='horizontal', variable=degree,
29                from=1, to=8, resolution=1)
30    b_col <- tkbutton(tt, bg=color, activebackground=color, command=updateColor)

```



```

31     b_add <- tkbutton(tt, text='add', command=addRegressionLine)
32     tkgrid(tklabel(tt, text='degree:'), s, b_col, b_add, sticky='s', pady=5)
33     tkgrid.columnconfigure(tt, 1, weight=1)
34     tkgrid.configure(s, sticky='ew')
35 }

```

## A.3 Sensitivity Analysis Simple Linear Regression

```

1  ## Fit simple linear regression
2  fit <- lm(Fertility~Infant.Mortality, data=swiss)
3
4  ## Data
5  p <- with(swiss, l_plot(Fertility~Infant.Mortality,
6                          title='swiss data (least-squares)',
7                          linkingGroup='swiss',
8                          itemlabel=rownames(swiss)))
9
10 ## layer fit
11 xrng <- range(swiss$Infant.Mortality)
12 yhat <- predict(fit, data.frame(Infant.Mortality=xrng))
13 l_layer_line(p, x=xrng, y=yhat, linewidth=3, index="end")
14
15 ## Fitted vs. Residuals
16 pr <- l_plot(x=fit$fitted, y=fit$resid,
17             xlabel="Fitted values",
18             ylabel="Residuals",
19             title="Residuals vs. fitted values",
20             linkingGroup='swiss',
21             itemlabel=rownames(swiss))
22
23 l_layer_line(pr, x=c(-25,100), y=c(0,0),
24             linewidth=3, color="gray80",
25             index="end")
26
27 ## Influential Points
28 plev <- l_plot(x=hatvalues(fit), y=rstudent(fit),
29             title="Leverage and outlier plot",
30             ylabel="Externally studentized residuals",
31             xlabel="hat values",
32             linkingGroup="swiss",
33             itemlabel=rownames(swiss))
34
35 llev <- l_layer_line(plev, index='end',
36             x=rep(4/plev['n'],2), y=c(-5,5),

```

```

37         linewidth=3, color='gray80')
38
39
40 ## Layer Confidence Intervals
41 xpvals <- with(swiss, seq(from=min(Infant.Mortality),to=max(Infant.Mortality),
42                          length.out=60))
43
44 conf95 <- predict(fit, data.frame(Infant.Mortality=xpvals),
45                  interval="confidence", level=0.95)
46
47 conf99 <- predict(fit, data.frame(Infant.Mortality=xpvals),
48                  interval="confidence", level=0.99)
49
50 pred95 <- predict(fit, data.frame(Infant.Mortality=xpvals),
51                  interval="prediction", level=0.95)
52
53 pred99 <- predict(fit, data.frame(Infant.Mortality=xpvals),
54                  interval="prediction", level=0.99)
55
56
57 ## Interactively remove points from OLS
58 use_color <- p['color'][1]
59
60 l.sel <- l_layer_line(p, x=xrng, y=yhat, color='red', linewidth=3,
61                      index="end")
62
63 confg <- l_layer_group(p, label="Confidence intervals", index="end")
64 predg <- l_layer_group(p, label="Prediction intervals", index="end")
65
66
67 polyc95 <- l_layer_polygon(p,
68                            x=c(xpvals, rev(xpvals)),
69                            y=c(conf95[,2], rev(conf95[,3])),
70                            color="lightblue2",
71                            linecolor="",
72                            parent=confg,
73                            label="95% confidence",
74                            index="end")
75
76 polyc99 <- l_layer_polygon(p,
77                            x=c(xpvals, rev(xpvals)),
78                            y=c(conf99[,2], rev(conf99[,3])),
79                            color="lightblue1",
80                            linecolor="",
81                            parent=confg,

```

```

82             label="99% confidence",
83             index="end")
84
85 polyp95 <- l_layer_polygon(p,
86             x=c(xpvals,rev(xpvals)),
87             y=c(pred95[,2],rev(pred95[,3])),
88             color="lightpink2",
89             linecolor="",
90             parent=predg,
91             label="95% prediction",
92             index="end")
93
94 polyp99 <- l_layer_polygon(p,
95             x=c(xpvals,rev(xpvals)),
96             y=c(pred99[,2],rev(pred99[,3])),
97             color="lightpink1",
98             linecolor="",
99             parent=predg,
100            label="99% prediction",
101            index="end")
102
103 l_scaletto_world(p)
104
105
106 updateRegression <- function() {
107     ## which points to use for regression
108     sel <- p['color'] == use_color
109     sel <- sel & p['active']
110     ## which coordinates to use for regression
111     xnew <- p['xTemp']
112     if (length(xnew) == 0) {
113         xnew <- p['x']
114     }
115
116     ynew <- p['yTemp']
117     if (length(ynew) == 0) {
118         ynew <- p['y']
119     }
120
121     fit.temp <- lm(y~x, subset(data.frame(x=xnew, y=ynew), sel))
122
123     xrng <- range(xnew)
124
125     ## the fitted line
126     yhat <- predict(fit.temp, data.frame(x=xrng))

```

```

127     l_configure(l.sel, y=yhat, x=xrng)
128
129     ## the intervals
130     ##
131     xpvals.temp <- seq(from=min(xrng),to=max(xrng),
132                       length.out=60)
133
134     conf95.temp <- predict(fit.temp, data.frame(x=xpvals.temp),
135                          interval="confidence", level=0.95)
136
137     conf99.temp <- predict(fit.temp, data.frame(x=xpvals.temp),
138                          interval="confidence", level=0.99)
139
140     pred95.temp <- predict(fit.temp, data.frame(x=xpvals.temp),
141                          interval="prediction", level=0.95)
142
143     pred99.temp <- predict(fit.temp, data.frame(x=xpvals.temp),
144                          interval="prediction", level=0.99)
145
146     ## update the prediction intervals
147     ##
148     l_configure(polyp99,
149               x=c(xpvals.temp,rev(xpvals.temp)),
150               y=c(pred99.temp[,2],rev(pred99.temp[,3])))
151
152     l_configure(polyp95,
153               x=c(xpvals.temp,rev(xpvals.temp)),
154               y=c(pred95.temp[,2],rev(pred95.temp[,3])))
155
156     ## update the confidence intervals
157     ##
158     l_configure(polyc99,
159               x=c(xpvals.temp,rev(xpvals.temp)),
160               y=c(conf99.temp[,2],rev(conf99.temp[,3])))
161
162     l_configure(polyc95,
163               x=c(xpvals.temp,rev(xpvals.temp)),
164               y=c(conf95.temp[,2],rev(conf95.temp[,3])))
165
166
167     ## residvs versus fitted plot
168     fitted <- predict(fit.temp, data.frame(x=xnew))
169     l_configure(pr, x=fitted, y=ynew - fitted)
170
171     ## leverage plot

```

```

172     l_configure(plev, x=hatvalues(fit.temp), y=rstudent(fit.temp),
173                 linkingKey=which(sel)-1, sync="pull")
174     l_scaletto_world(plev)
175     l_configure(llev, x=rep(4/sum(sel),2))
176     tcl('update', 'idletasks')
177 }
178
179 bnd <- l_bind_state(p, c("color","active","xTemp","yTemp"),
180                    function()updateRegression())
181
182 ## Map
183 library(maps)
184 m <- map("world", 'Switzerland', fill=TRUE, plot=FALSE)
185
186 ## Coordinates Obtained With Google Maps
187 swissCoords <- structure(list(
188     latitude = c(47.1783274, 47.365837, 47.254872,
189                 47.2782749, 47.0632023, 47.416647, 46.5892626,
190                 46.761285, 46.6757485, 46.7866673, 46.5280339,
191                 46.3190253, 46.4953291, 46.8806009, 46.6140944,
192                 46.6410996, 46.8092091, 46.5196535, 46.4312213,
193                 46.5, 46.5088127, 46.6698891, 46.3832683, 46.7293301,
194                 46.5666667, 46.8220266, 46.4666667, 46.4612971,
195                 46.4628333, 46.7784736, 46.2244777, 46.0163423,
196                 46.0801475, 46.1049798, 46.2521873, 46.214941,
197                 46.2941311, 46.2331221, 46.9542921, 47.1034892,
198                 47.0577195, 46.9899874, 47.0833333, 46.9, 46.2043907,
199                 46.2083126, 46.2455233),
200     longitude = c(7.0729547, 7.3451555, 7.0028421, 7.3716656, 7.0912628,
201                 7.0765657, 6.9555376, 7.0901001, 7.095521, 7.1621113,
202                 6.9175828, 6.970566, 6.3918325, 7.0427075, 6.507171,
203                 6.6344508, 6.6457678, 6.6322734, 6.9106799, 6.75, 6.4961301,
204                 6.7975224, 6.2347852, 6.5323588, 6.8333333, 6.9405663, 7.0833333,
205                 6.3397549, 6.8419192, 6.641183, 7.303512, 7.2706464, 7.4698932,
206                 7.0755334, 6.9469598, 7.0047948, 7.5335362, 7.360626, 6.8478409,
207                 6.8327838, 6.7487354, 6.9292732, 6.9666667, 6.6, 6.1431577,
208                 6.1458643, 6.2090779)),
209     .Names = c("latitude", "longitude"),
210     row.names = c("Courtelary Switzerland",
211                 "Delemont Switzerland", "Franches-Mnt Switzerland",
212                 "Moutier Switzerland", "Neuveville Switzerland",
213                 "Porrentruy Switzerland", "Broye Switzerland",
214                 "Glane Switzerland", "Gruyere Switzerland",
215                 "Sarine Switzerland", "Veveyse Switzerland",
216                 "Aigle Switzerland", "Aubonne Switzerland",

```

```

217         "Avenches Switzerland", "Cossonay Switzerland",
218         "Echallens Switzerland", "Grandson Switzerland",
219         "Lausanne Switzerland", "La Vallee Switzerland",
220         "Lavaux Switzerland", "Morges Switzerland",
221         "Moudon Switzerland", "Nyone Switzerland",
222         "Orbe Switzerland", "Oron Switzerland",
223         "Payerne Switzerland", "Paysd'enhaut Switzerland",
224         "Rolle Switzerland", "Vevey Switzerland",
225         "Yverdon Switzerland", "Conthey Switzerland",
226         "Entremont Switzerland", "Herens Switzerland",
227         "Martigwy Switzerland", "Monthey Switzerland",
228         "St Maurice Switzerland", "Sierre Switzerland",
229         "Sion Switzerland", "Boudry Switzerland",
230         "La Chauxfdnd Switzerland", "Le Locle Switzerland",
231         "Neuchatel Switzerland", "Val de Ruz Switzerland",
232         "ValdeTravers Switzerland", "V. De Geneve Switzerland",
233         "Rive Droite Switzerland", "Rive Gauche Switzerland"),
234     class = "data.frame")
235
236
237 p_map <- with(swissCoords, l_plot(longitude,latitude,
238                                 itemlabel=rownames(swissCoords),
239                                 showItemlabels=TRUE,
240                                 linkingGroup='swiss'))
241
242 l <- l_layer(p_map, m, index=1)
243 l_layer_lower(p_map, l)
244 l_scaletto_world(p_map)

```

## A.4 Interactive K nearest neighbour highlighting

```

1 highlight_knn <- function(p, data, k=5, method='euclidean') {
2
3     if(!is(data, 'data.frame'))
4         data <- as.data.frame(data)
5
6     ## Create Custom Control Panel
7     tt <- tktoplevel()
8
9     onOff <- tclVar('1')
10    tkgrid(tkcheckboxbutton(tt, text='on/off', variable=onOff), sticky='w')
11
12    k <- tclVar(k)
13    f1 <- tkframe(tt)

```

```

14
15 e <- tkentry(f1, width=3, textvariable=k)
16 tkbind(e, '<Return>', function()hNN())
17 tkgrid(f1, sticky='w')
18 tkpack(tklabel(f1, text='k='), e, side='left')
19
20 tkgrid(tklabel(tt, text='Nearest to:'), sticky='w')
21 distFrom <- tclVar('points')
22 f2 <- tkframe(tt)
23 tkgrid(f2, sticky='w')
24 tkpack(tkradiobutton(f2, text='points', variable=distFrom,
25                 value='points', command=function()hNN()),
26         tkradiobutton(f2, text='mean', variable=distFrom,
27                 value='mean', command=function()hNN()),
28         side='left')
29
30 tkgrid(tklabel(tt, text='Space:'), sticky='w')
31 chbtns <- lapply(names(data), function(name) {
32     bvar <- tclVar('1')
33     b <- tkcheckboxbutton(tt, text=name, variable=bvar,
34                 command=function()hNN())
35     tkgrid(b, sticky='w', padx=2)
36     return(bvar)
37 })
38
39
40 ## Create Nearest neighbour highlighting Functionality
41
42 if(!is(p, 'loon'))
43     class(p) <- "loon"
44
45 n <- nrow(data)
46 ## Which variables are used for D
47 cachedSpaceSelection <- rep(TRUE, ncol(data))
48 D <- as.matrix(dist(data, method = method))
49 I <- matrix(rep(1:n, n), ncol=n, byrow=TRUE)
50
51 inds <- 1:n # used for subsetting
52
53 ## Cache point glyph attributes that are used for highlighting
54 glyphCache <- character(0)
55 whichCache <- integer(0)
56 sizeCache <- integer(0)
57
58 ## Function that highlights nearest neighbours

```

```

59 hNN <- function() {
60
61     ## reset cached point glyphs attributes
62     if (length(whichCache) > 0) {
63         l_configure(p, glyph=glyphCache, size=sizeCache, which=whichCache)
64         whichCache <- integer(0)
65     }
66
67     if (tclvalue(onOff) == '0') return()
68
69     isel <- which(p['selected'])
70     if (length(isel) == 0 || length(isel) == n) return()
71
72     spaceSelection <- vapply(chbtns,
73                             function(b)as.logical(as.numeric(tclvalue(b))),
74                             logical(1))
75
76     if(tclvalue(distFrom)=='points') {
77         if(!identical(cachedSpaceSelection, spaceSelection)) {
78             D <- as.matrix(dist(data[, spaceSelection]))
79             cachedSpaceSelection <- spaceSelection
80         }
81         chng_which <- unique(c(I[isel, -isel][order(c(D[isel, -isel]))])
82     } else {
83         p_mean <- apply(data[isel, spaceSelection], 2, mean)
84         d <- apply(data[-isel, spaceSelection], 1,
85                   function(row) dist(rbind(row, p_mean)))
86
87         chng_which <- (inds[-isel])[order(d)]
88     }
89
90     kval <- tclvalue(k)
91     if (grepl('[:digit:]+', kval)) {
92         kval <- as.numeric(kval)
93     } else {
94         kval <- 5
95     }
96
97     ksel <- min(length(chng_which),kval)
98
99     whichCache <- chng_which[1:ksel]
100    glyphCache <- p['glyph'][whichCache]
101    sizeCache <- p['size'][whichCache]
102    l_configure(p, glyph='csquare', size=seq(25, 8, length.out = ksel),
103              which=whichCache)

```



```
104     }  
105  
106     l_bind_state(p, 'selected', hNN)  
107 }
```