Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand). This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.

- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.

- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage. [http://researchspace.auckland.ac.nz/feedback](http://researchspace.auckland.ac.nz/feedback)

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library [Thesis Consent Form](http://researchspace.auckland.ac.nz/feedback)
Investigations in Graphical Statistics

Paul R. Murrell
A thesis submitted in partial fulfilment of the requirements for
the degree of Doctor of Philosophy in Statistics,
The University of Auckland, 1998

July 31, 1998
This thesis is concerned with the design and development of statistical graphics software—programs to help draw graphs. Graphs serve two major functions in statistics. Firstly, graphs are used for exploratory data analysis—for detecting the message in a set of data—and secondly, graphs are used for data display—for presenting the message in a set of data.

The most important feature of software for exploratory data analysis is extensibility. This is the ability to quickly and easily develop new graphical images and is vital for being able to explore a data set in many different ways. The most important feature of software for data display is customisation. This is the ability to fine-tune a graphical image in great detail and is vital for the production of presentation-quality graphics. In both cases it is important that a graphical image should be constructed to best explore or show-off the peculiarities of a specific data set.

A pervading theme of this thesis is that statistical graphics software should be flexible. The software tools described herein allow graphical images to be modified in arbitrary ways; the structure of graphical images is also arbitrary and not restricted to standard graph formats; a simple, coherent method, based on a general constraint system, for developing novel graphical images is explored; and a mechanism for specifying the arrangement of the components of a graphical image is introduced. Some of these ideas are incorporated within an existing statistical analysis package.
Acknowledgements

I would like to thank the following for their contributions to this thesis:

The University of Auckland for the provision of a University of Auckland Doctoral Scholarship.

The Department of Statistics for the provision of generous space, computing, printing, and photocopying resources.

Ross Ihaka for his supervision. In particular, thanks for always being available, always having an answer, and always seeming to place more importance on my career than your own.

Julia Murrell née Clark for proofreading, advice, support, and for teaching me the importance of “time off.”
## Contents

1 Introduction ........................................... 1
   1.1 What is Graphical Statistics ? .................. 1
   1.2 Issues in Graphical Statistics ................. 2
      1.2.1 Customisation ............................. 2
      1.2.2 Extensibility ............................. 2
      1.2.3 Presentation-Quality Output vs. Flexibility and Interaction 3
   1.3 Modern Graphical Statistics Software .......... 4
      1.3.1 S ........................................... 4
      1.3.2 R ........................................... 4
      1.3.3 Qual ......................................... 5
      1.3.4 Pictor ....................................... 6
      1.3.5 XLispStat ................................... 7
      1.3.6 XGobi ....................................... 8
   1.4 The Structure of this Thesis .................... 9
   1.5 References ......................................... 10

2 Simplisp ................................................. 12
   2.1 Aims ............................................... 15
      2.1.1 Fully incorporating 3D graphs ............. 15
      2.1.2 Accessing every detail of a graph ......... 15
      2.1.3 Sharing elements between graphs .......... 16
      2.1.4 3D selection ................................ 17
      2.1.5 Arbitrary Plot Structures .................. 19
   2.2 Design ............................................ 20
      2.2.1 Constructing a Scene ....................... 20
      2.2.2 Viewing a Scene ............................. 20
      2.2.3 Automatic updating .......................... 21
      2.2.4 Selecting with a mouse ..................... 21
   2.3 Implementation ................................... 23
      2.3.1 Simplisp Classes ............................ 23
   2.4 Simplisp Reference ................................ 29
   2.5 User’s Guide ...................................... 37
      2.5.1 Getting Started ............................. 37
      2.5.2 Creating a plot ............................. 37
      2.5.3 Selecting Elements of a Plot ................ 37
      2.5.4 Sharing elements between graphs .......... 42
      2.5.5 Creating Arbitrary Plots .................... 43
      2.5.6 Getting stopped ............................. 46
      2.5.7 An Advanced Simplisp Example .............. 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Results and Conclusions</td>
<td>48</td>
</tr>
<tr>
<td>2.6.1</td>
<td>3D-based graphics</td>
<td>48</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Accessing every detail of a graph</td>
<td>49</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Sharing elements between graphs</td>
<td>50</td>
</tr>
<tr>
<td>2.6.4</td>
<td>3D selection</td>
<td>51</td>
</tr>
<tr>
<td>2.6.5</td>
<td>Arbitrary graph structure</td>
<td>52</td>
</tr>
<tr>
<td>2.6.6</td>
<td>General problems</td>
<td>52</td>
</tr>
<tr>
<td>2.7</td>
<td>References</td>
<td>55</td>
</tr>
<tr>
<td>3</td>
<td>Xtend</td>
<td>56</td>
</tr>
<tr>
<td>3.1</td>
<td>Aims</td>
<td>62</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Extensibility</td>
<td>62</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Incremental Graphics</td>
<td>62</td>
</tr>
<tr>
<td>3.1.3</td>
<td>A Stand-Alone Graphics System</td>
<td>63</td>
</tr>
<tr>
<td>3.2</td>
<td>Design</td>
<td>65</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Constructing a graph</td>
<td>65</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Developing new plotting elements</td>
<td>66</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Combining plotting elements</td>
<td>66</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Constraints between plotting elements</td>
<td>66</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Communication with R</td>
<td>67</td>
</tr>
<tr>
<td>3.3</td>
<td>Implementation</td>
<td>68</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Xtend Prototypes</td>
<td>70</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Inheritance</td>
<td>74</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Edit-slot</td>
<td>74</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Rslave</td>
<td>75</td>
</tr>
<tr>
<td>3.4</td>
<td>Xtend Reference</td>
<td>77</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Functions</td>
<td>77</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Prototypes</td>
<td>78</td>
</tr>
<tr>
<td>3.5</td>
<td>User's Guide</td>
<td>90</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Getting started</td>
<td>90</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Creating a plot</td>
<td>90</td>
</tr>
<tr>
<td>3.5.3</td>
<td>The edit-slot function</td>
<td>90</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Controlling plot layout</td>
<td>90</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Adding plotting elements to each other</td>
<td>91</td>
</tr>
<tr>
<td>3.5.6</td>
<td>Communicating with R</td>
<td>91</td>
</tr>
<tr>
<td>3.5.7</td>
<td>Constraints between plotting elements</td>
<td>92</td>
</tr>
<tr>
<td>3.5.8</td>
<td>Creating new plotting elements</td>
<td>93</td>
</tr>
<tr>
<td>3.5.9</td>
<td>The Motif interface</td>
<td>95</td>
</tr>
<tr>
<td>3.5.10</td>
<td>An Advanced Xtend Example</td>
<td>96</td>
</tr>
<tr>
<td>3.6</td>
<td>Results and Conclusions</td>
<td>101</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Extensibility</td>
<td>101</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Incremental Graphics</td>
<td>102</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Stand-Alone Graphics</td>
<td>106</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Problems</td>
<td>106</td>
</tr>
<tr>
<td>3.7</td>
<td>References</td>
<td>108</td>
</tr>
</tbody>
</table>
A An introduction to
Object-Oriented Programming
in Common Lisp  179
A.1 Programming in Lisp  179
  A.1.1 Lisp Syntax  181
  A.1.2 Lisp and Lists  181
  A.1.3 Symbols  182
  A.1.4 Special forms  182
  A.1.5 Keyword Arguments  182
A.2 CLOS: Object-Oriented Programming in
Common Lisp  183
  A.2.1 Classes and Instances  183
  A.2.2 Generic Functions and Methods  184
List of Figures

1  The common components of a graph .......................................................... ix
1.1 A hierarchy of Pictor groDs ................................................................. 7

2.1 A simple scatterplot. ................................................................................ 12
2.2 A scatterplot and a 3D plot together on the same page. ....................... 13
2.3 Modifying a shared y-axis ..................................................................... 13
2.4 Modifying individual data symbols and tick-mark labels .................... 13
2.5 Relocating an axis label in 3D ............................................................... 14
2.6 Customising a “temperature” axis ......................................................... 15
2.7 Overlapping graphical objects ............................................................... 18
2.8 Back-to-front ordering of graphical objects ........................................ 18
2.9 The viewing elements of the Simplisp system ......................................... 22
2.10 The resource classes in Simplisp. .......................................................... 24
2.11 The viewing classes in Simplisp. ............................................................ 25
2.12 The input/output classes in Simplisp. ..................................................... 26
2.13 The graphical classes in Simplisp. .......................................................... 26
2.14 The collection classes in Simplisp. ........................................................ 27
2.15 A Simplisp plot viewed in an X11 window ........................................... 38
2.16 A Simplisp object hierarchy ................................................................. 39
2.17 Customisation and 3D views in Simplisp ............................................. 40
2.18 Customisation in 3D ............................................................................. 41
2.19 An object hierarchy showing shared elements ....................................... 42
2.20 Two plots with shared elements ............................................................ 43
2.21 Two plots after modifying a shared element ......................................... 44
2.22 A scatterplot of height vs. age ............................................................. 44
2.23 Adding new elements to a plot ............................................................. 45
2.24 The Simplisp 3D picking algorithm ...................................................... 51

3.1 A graph with a dot-plot per group ........................................................... 57
3.2 A graph with dot-plots and boxes ........................................................... 58
3.3 A graph for performing an analysis of variance by eye ......................... 59
3.4 Another graph for performing an analysis of variance by eye ............... 60
3.5 The components of the viewing system in Xtend ................................... 65
3.6 An example of inheritance between graphic objects ............................. 80
3.7 A Motif-style dialog ............................................................................. 96
3.8 Two possible implementations of a boxplot plotting element ............... 102
3.9 A novel data symbol plotting element .................................................. 105

4.1 Specifying the location of a plot on a page ............................................ 110
LIST OF FIGURES

4.2 Coordinate systems in S ........................................ 111
4.3 The S margin coordinate systems. .............................. 111
4.4 Allocating rows and columns from a layout specification 120
4.5 Allocating a region to a plot .................................. 120
4.6 Allocating absolute rows and columns ........................ 121
4.7 Allocating rows and columns with full respect .............. 121
4.8 Allocating rows and columns with partial respect ......... 122
4.9 Allocating a mixture of relative and absolute rows and columns ........ 123
4.10 A simple layout example ...................................... 129
4.11 A layout with full respect .................................... 130
4.12 A layout with row-heights and column-widths specified .... 131
4.13 A layout with absolute row-heights and column-widths .... 132
4.14 A layout with margins ......................................... 133
4.15 Arranging components within a plot with layouts ......... 134

5.1 A diagram of the bounding box of a character. .............. 139
5.2 Typesetting for the text "hey there" .......................... 139
5.3 Typesetting for the expression \( \text{over}(x[i], y + 2) \) ....... 140
5.4 A plot of the Normal probability density function .......... 143
5.5 A periodogram with mathematical annotation ............... 143
5.6 Specifying figure regions with \( \text{par(mfrow=}... \) .......... 147
5.7 Specifying the figure region with \( \text{par(fin=}... \) ........ 148
5.8 Specifying the plot region with \( \text{par(mar=}... \) ........ 149
5.9 Specifying the plot region with \( \text{par(pin=}... \) ........ 149
Terminology

In this thesis, a graph or plot is simply considered to be a collection of graphical elements. The graphical elements may be anything from basic graphical primitives such as lines, rectangles, and text to complex statistical objects such as axes, boxplots, and scatterplots. The latter are often referred to as plotting elements. Plotting elements are considered to be made up of a number of components. For example, the components of an axis are a major line, a number of tick-marks, and a label. Figure 1 shows the terminology used in this thesis for the standard components of a graph.

![Diagram of graph components](image)

Figure 1: The common components of a graph.
Chapter 1

Introduction

This thesis describes a body of research in the area of graphical statistics. The research involved developing experimental graphical software and enhancing the graphics capabilities of an existing statistical software package.

1.1 What is Graphical Statistics?

Graphs, statistical graphics, or “pictures of numbers” (Tufte, 1983) provide a powerful and efficient means of communicating quantitative information. The birth of graphs is often traced to the work of William Playfair (1759-1823), but the proliferation of graphs has only occurred since the mid-1970’s (Chambers, Cleveland, Kleiner, & Tukey, 1983). The increased use of graphs has depended upon the development of computer hardware and software which can largely automate the production of complex graphical images.

This thesis is concerned with the design and implementation of statistical graphics software.

There are an enormous number of software packages available which produce graphs of one sort or another. There are packages which primarily provide a full complement of statistical analysis tools, including graphics (e.g., S-Plus, SAS, SPSS, Systat, Minitab, Statistica, and DataDesk), packages which primarily provide graphical tools, with some statistical analysis included (e.g., SigmaPlot, Axum), and software which has no specific statistical orientation, but includes some analysis and graphing capabilities (e.g., Microsoft Excel). In addition, there are a number of packages which represent ongoing research into statistical graphics software, for example, XLispStat (Tierney, 1990), the R system (Ihaka & Gentleman, 1996), Quail (Hurley & Oldford, 1991), Pictor (Wilks, 1996), and XGobi (Swayne and Cook, 1990).

What are the common features of these software packages? It is not just that they all allow the user to create a graph. It would be possible to create a graph using any graphics package (e.g., MacDraw, Microsoft Draw, or xfig) albeit with an enormous amount of time and effort. What is important about a statistical graphics package is that it allows the user to create a graph and it does most of the work.
1.2 Issues in Graphical Statistics

This section introduces a number of general issues in statistical graphics which are addressed in this thesis.

1.2.1 Customisation

Statistical graphics software is convenient because it is able to produce a very complex graph from relatively simple instructions. The software arranges many lines, points, and pieces of text, which would be incredibly laborious to arrange by hand, in an appropriate and pleasing manner. The potential danger with this approach lies in determining what sort of arrangement is "appropriate and pleasing"—as evidenced by the number of different theories and the greater number of opinions concerning the layout of the elements of a graph (see, for example, Chambers et al., 1983; Cleveland, 1993, 1994; Henry, 1995; Schmid, 1992; Tufts, 1983, 1990, Tukey & Wilk, 1965).

All statistical graphics packages provide a default arrangement for a graph and a range of parameters for customising the default arrangement for specific needs or preferences. For example, a graph axis might have five tick-marks by default and the range on the axis might encompass the range of the plotted data. The user can override these defaults by explicitly specifying the number of tick-marks and/or the range of values on the axis.

The extent to which the defaults provided by the software can be overridden varies between packages. For example, XGobi does not allow any control over the axis scales on graphs whereas Systat and Axum allow very fine control of the graph (e.g., it is possible in Systat to control the size of each individual tick-mark in a scatterplot).

A high degree of customisation is a very useful feature. The convenience of having design decisions made by the software should not come at the expense of being constricted by those decisions; there is nothing convenient about an ugly or obstructive graphical arrangement no matter how easy it is to create.

Chapter 2 investigates a software design which allows for arbitrary customisations of graphs.

1.2.2 Extensibility

To a greater or lesser extent, most statistical graphics software provides some way for the user to extend the available functionality of the software. This ranges from simple macro or scripting facilities (e.g., SigmaPlot), to special command languages (e.g., Axum), and even to complete environments for "programming with data" (e.g., S-Plus, XLispStat).

In one sense, the ability to extend a system provides the ultimate in customisation—if the default arrangement is not sufficiently appealing, program a new one—however, this is only really true of systems which provide a statistical programming environment.

The real power of extensibility is in the development of new graphical techniques. Luke Tierney's X LispStat system (see Section 1.3.5 below) is currently the leading-edge in this regard, particularly for the investigation of new dynamic techniques.

One obstacle to extensibility, from the point-of-view of the user, is that it generally involves programming. Here the goal of extensibility appears incompatible with the desire for ease-of-use. A programming environment which is specifically
designed for the development of statistical applications can provide a friendlier interface than a general-purpose programming language, but the user-interface of an extensible system can be a major hurdle for the user.

A new approach to extensibility, based on a general constraint mechanism is described in Chapter 3.

1.2.3 Presentation-Quality Output vs. Flexibility and Interaction

The uses of graphs can be divided into two major types—finding the message in data (data analysis) and communicating the message in data (presentation graphics). Typical requirements of the first type of graph are that it must be easy and quick to generate and modify (at least in simple ways), while the requirements of the second type are that the graph must be well-designed, the user must be able to extensively customise the graph, and it must be possible to produce the graph in a recognised output format (e.g., PostScript).

Most systems allow the user to generate very high-quality 2D graphs, but provide less support for 3D graphs. In some cases, this is deliberate (e.g., the axes on a 3D rotating plot are very basic, or missing altogether, in order to avoid cluttering up the plot), however, there is still a requirement for high-quality 3D graphs (e.g., in order to publish the “best” view obtained from a rotating plot). Chapter 2 looks at the production of high-quality 3D graphs.

In general, more flexible and interactive systems provide less support for high-quality output (e.g., XGobi and XLispStat). Chapters 3 and 4 address the problem of creating a flexible system which still allows for fine control of the graphics and for the production of high-quality output.
1.3 Modern Graphical Statistics Software

This section describes a number of statistical graphics packages in more detail. These packages have been developed by statistical graphics researchers and are designed to provide the user with the most up-to-date statistical graphics tools and/or allow users to develop new tools of their own.

1.3.1 S

The S system (Becker, Chambers, and Wilks, 1988) was developed by Richard A. Becker, John M. Chambers, and Allan R. Wilks of AT&T Bell Laboratories Statistics Research Department. S-Plus is a commercial version of S with extra functionality added, especially for performing specific sorts of statistical analyses.

S has a command line interface. The standard graphs are created by entering the appropriate commands. For example, two vectors of random normal values are created and plotted with the following commands:

```r
x <- rnorm(10)
y <- rnorm(10)
plot(x, y)
```

The power of S is that it is a programming environment; users are able to create new commands (functions) to perform new statistical analyses or to generate new graphical displays. For example, the following code creates a new function which takes a vector of y-values and two vectors of x-values and generates two scatterplots side by side—y versus z1 and y versus z2:

```r
plot.two <- function(y, x1, x2)
{
  par(mfrow=c(1,2))
  plot(x1, y)
  plot(x2, y)
}
```

S graphics are static. This means that once a graphical object has been drawn on an output it cannot be modified; the overall graph can be modified by adding further elements (e.g., annotating with text or adding lines between points or further series of points), but once each element has been drawn, it is fixed. Modifications to elements which have been drawn require redrawing the entire graph (reentering the commands with the appropriate changes). Also, the S graphics commands are relatively high-level, which makes it difficult to generate arbitrary novel graphical images.

The graphics systems described in Chapters 2, 3, and 4 are dynamic; elements of a graph may be modified after they have been drawn (and the software will take care of redrawing the image to reflect the change). Chapter 3 describes a more powerful approach to providing users with the ability to program novel graphical images.

1.3.2 R

The R system (Ihaka and Gentleman, 1996) is "a language which is not entirely unlike the S language". R began as an experiment to investigate whether an S-like system could be created which did not suffer from the memory and performance
deficiencies of $S$ ($S$ is particularly greedy with respect to computer memory and performs certain sorts of computations inefficiently). The $R$ interface and the behaviour of $R$ commands is almost identical to those of $S$.

Because the system is relatively new and because its source code is freely available, $R$ provides a technologically up-to-date test-bed for experimenting with new graphical ideas. Chapter 5 reports on work within the graphics system of $R$.

1.3.3 Quail

Quail (QUantitative Analysis In Lisp; Hurley and Oldford, 1991) is a general statistical analysis package which includes graphics capabilities. Quail was developed in MCL (Macintosh Common Lisp).

The graphics system of Quail (called vloews) provides a number of standard graphs. For example, the following code creates vectors, $x$ and $y$ each containing 10 random normal values and produces a scatterplot of these values:

```
(<- x (random-gaussian :n 10))
(<- y (random-gaussian :n 10))
(scatterplot :x x :y y)
```

The range of graphs provided by vloews is relatively small, but the system is noteworthy for its dynamic capabilities and its potential for extensibility. It is not only possible to link plots in vloews, but also to link all sorts of graphical objects. For example, a scatterplot with a linked histogram in the right margin is created by:

```
(scatterplot :x x :y y :right-view 'histogram :link t)
```

Clicking with the mouse on data points in the scatterplot highlights the appropriate data points and the appropriate regions of the bars of the histogram. The following code creates a needle-slider (a rectangle with a line that can be moved left and right within the rectangle using the mouse) and a linked text label which describes the current level represented by the line on the slider:

```
(let ((n (needle-slider))
    (l (label :text
      #'(lambda ()
         (format nil "A" (slider-level-of n)))))
    (p (plot :left-view 'axis
             :interior-view n
             :right-view l))
    (text-link (interior-view-of p) l))
```

When the line on the slider is dragged with the mouse, the text in the label is updated for the new position of the line.

VIEWS allows graphical objects to be combined together to form new objects. In the example above, the needle-slider and the text label were combined together within a plot object (by the command (plot :left-view 'axis ...)), which has predefined regions for objects to fit into (e.g., an :interior-view region and a :right-view region). The following code demonstrates that objects can also be combined in arbitrary ways; the new object combines a pair of lines so that they form a cross occupying the object's top-right quarter:
(setq newl
  (view-layout :subviews
    (list (line)
      (line :slope -1 :intercept 1))
    :positions
    (list (make-region .5 1 .5 1)
      (make-region .5 1 .5 1))
    :draw? t))

The disadvantages of Quail are the fact that it is based on the MCL environment and its lack of documentation. MCL is not a familiar environment for many users and Lisp is arguably one of the more difficult to come to grips with. The fact that Quail is only available for the Macintosh further decreases its potential audience.

The lack of documentation makes it very difficult to take advantage of the power of the system or even to determine what is possible. While it is a simple matter to dumbly perform the provided examples, there is not enough explanation of how these examples work for users to be able to develop ideas of their own.

There is a subtlety to this problem. In any software package, there is an internal implementation which the user does not see and does not need to know anything about to use the software, and there is an external interface which the user must know as much as possible about in order to use the software. In a software package which has a GUI, the distinction is quite obvious; the implementation is a body of code written in a programming language (e.g., C) and the interface is a set of windows, menus and dialogs. In Quail, the implementation is in Lisp and there is a lisp-based command-line interface (in addition to a GUI). The advantage of this is that the user has access to all of the power of general Lisp expressions for forming commands to send to Quail. The disadvantage is that it is difficult to distinguish the boundary between the implementation and the interface. I am not suggesting that documentation is required for all of the Lisp code in Quail (implementation and interface), just that more explanation is required of the Lisp code that makes up the Quail command-line interface.

This problem has the greatest impact on users who wish to use Quail to experiment with new graphical images. An associated problem is the fact that there appears to be several different ways of performing any one task. For example, linking plots requires a quite different approach than linking a needle-slider and a piece of text, and combining objects within a plot is quite different from combining objects within a view-layout. This difficulty is greatly exacerbated by the fact that neither method is clearly explained, but it is easier for the user to work with a system in which there is a single, straightforward way of performing a particular task (e.g., linking objects or combining objects together). This important aspect of extensible software is addressed in Chapter 3.

1.3.4 Pictor

Pictor (Wilks, 1994a, 1994b, 1996) is a statistical graphics system implemented in S. In Pictor, graphs are hierarchical collections of graphical objects called grobs. For example, a simple scatterplot consists of a number of points, two axes, and a box to go around the points. Each axis consists of a number of tick-marks and a label (see Figure 1.1).

The different components are combined together in sensible ways by specifying constraints between them. Each grob has a coordinate system and a bounding box.
It is possible to specify a limited number of constraints between the coordinate systems of two grobs and between the bounding boxes of any two grobs. For example, the constraint,

\[
\text{constraint("xexact", i, j)}
\]

specifies that, in the horizontal direction, the coordinate systems of \(i\) and \(j\) exactly match (this can be used to make sure that the x-axis and the points in a simple scatterplot use the same coordinate system). The following constraint specifies that the right side of the bounding box of \(i\) is horizontally aligned with the left side of the bounding box of \(j\):

\[
\text{constraint("bxmatch", i, j, 1, 0)}
\]

The arrangement of graphical objects in a hierarchy turns out to have some other useful applications (especially in relation to customisation and extensibility); these are discussed in Chapters 2 and 3. Chapter 3 also considers the use of constraints within statistical graphics and demonstrates the usefulness of completely general constraints. An alternative way of expressing the arrangement of the components of a plot is outlined in Chapter 4.

### 1.3.5 XLispStat

XLispStat (Tierney, 1990) is a statistical package with a particular emphasis on the development of new, especially interactive, graphical techniques.

XLispStat is based on the XLISP dialect of Lisp (Betz, 1985) and includes an object system. Plotting commands create a window containing a graph and return an object representing the graph window. For example, the following commands generate a scatterplot of some random normal values and store the object representing the scatterplot in the symbol called \texttt{my-plot}:

\[
\text{(setf x (normal-rand 10))}
\]
(setq y (normal-rand 10))
(setq my-plot (plot-points x y))

The object representing a graph can be sent a message to perform modifications to the graph. For example,

(send my-plot :abline 0 1)

adds the line \( y = x \) to the plot. The following message tells the plot to change the number of tick-marks on the x-axis to 7:

(send my-plot :x-axis t nil 7)

The object-oriented nature of XLispStat means that new objects can be derived from existing objects. This means that the behaviour of the new object in response to messages is exactly the same as the object it is derived from. For example, the standard graph objects (scatterplots, histograms, and boxplots) are all derived from an object called graph-proto which defines all sorts of useful behaviour that most graphs need (such as drawing an axis when the object is sent a :x-axis message). The new object can override the behaviour of the object it is derived from by specifying its own behaviour for a particular message (this is called writing a method for the new object). For example, the scatterplot, the histogram, and the boxplot all behave differently to the :add-points message because they each have their own method for this message.

The amount of work required to produce new sorts of plots depends upon how different the new plot is from the existing XLispStat objects—a larger number of differences will require writing more special methods. A reasonable amount of programming sophistication is required to fully exploit the power of XLispStat. An alternative paradigm for extensible graphics, which requires less programmer sophistication, is described in Chapter 3.

1.3.6 XGobi

XGobi (Swayne and Cook, 1990) is a stand-alone statistical graphics package available for the X Window environment. XGobi provides standard 1- and 2-dimensional graphs, but its speciality is visualisations of high-dimensional data. These include automatic animated visualisations such as grand tours (Cook, Buja, and Cabrera, 1995) and interactive methods where the user manually controls the projection of each variable into two dimensions (Swayne, Cook, and Buja, in press).

XGobi also implements brushing (selecting points by clicking the mouse or dragging a rectangular region with the mouse) and linked graphs (e.g., brushing points in one graph also selects corresponding points in another graph which shows the same data). The package is very up-to-date, especially with respect to high-dimensional data-visualisation and certain dynamic graphical techniques, but it is primarily a tool for exploratory data analysis rather than for producing presentation-quality output and it is not extensible (i.e., users cannot use it to further experiment with new visualisation techniques).

The most important aspect of XGobi with respect to this thesis is the fact that it is a stand-alone graphics package. The advantage of this design is that the sophisticated graphical capabilities of XGobi can be made available to any statistical package (and the sophisticated statistical analysis capabilities of other packages become available to XGobi).

There are a number of ways in which XGobi can communicate with other software packages:
• XGobi can import and export ASCII files, allowing it to use data exported in this format from other software and provide data for other software which can import this format.

• XGobi can be run from $S$, in which case data is automatically sent from $S$ and XGobi exports data in the $S$ format directly into the current $S$ data directory (Swayne, Cook, and Buja, 1992).

• A small number of programs (control panels) exist which run $S$ (invisibly apart from the $S$ graphics window) and one or more XGobi processes. These communicate with $S$ using UNIX inter-process communication (pressing a button on a control panel sends a specific command to $S$ and the result from $S$ is captured by the control panel) and run XGobi processes using the data obtained from $S$ (Swayne, Buja, and Hubble, 1991).

• XGobi can also be used in conjunction with ArcView (Symanzik, Majure, and Cook, 1997) and XploRe (Symanzik, Klinke, Cook, and Lewin, in press) using RPC (Remote Procedure Call) technology. This allows the different software packages to send complicated requests to each other as well as raw data.

Communication between statistical graphics software and other software packages becomes more problematic if the graphics software is extensible (i.e., new graphics capabilities can be added at run-time). This problem is addressed in Chapter 3.

1.4 The Structure of this Thesis

The research reported in this thesis is mainly concerned with the following issues:

• the ease-of-use of statistical software—in what ways can statistical software be made even more convenient?

• the customisation of graphs—is there a limit and should we go there?

• extensibility—how can this be provided and how can it be made less scary?

• the compromise between power and convenience—can we have both and if so how?

During the course of the research various new ideas were experimented with in three separate prototype software modules. Chapters 2, 3, and 4 describe these software modules. These chapters describe work which involved a good deal of programming. The thousands of lines of raw code are not included (for obvious reasons), but every attempt is made to provide sufficient explanation for each system by describing each programming task at several levels of complexity: the design of the software is given to convey the intended behaviour of the system; details of the implementation are provided to demonstrate how the design goals were achieved; reference material is included to describe the user interface of the software; and a user's guide provides a brief tutorial of the system and provides an opportunity to show off the important features.

Chapter 5 describes further work, which has a more practical aspect, implementing various graphical features in the $R$ system and the final chapter discusses the overall themes of the research and suggests further directions.
1.5 References


CHAPTER 1. INTRODUCTION


Chapter 2

Simplisp

Suppose that a number of children are given a test and their scores are graphed on a scatterplot as a function of their ages (see Figure 2.1).

There is a suggestion of more than one trend in the data so a 3D plot is created to plot the data against gender as well (see Figure 2.2).

The number of tick-marks on the y-axes is reduced to three to reduce the clutter on those axes (see Figure 2.3).

The 3D plot shows that there are two distinct trends: the scores for boys improve slowly at first and then rapidly, while the scores for girls do the reverse. To emphasise this distinction, the symbols for boys are coloured blue and the symbols for girls are coloured pink. The appropriate tick-mark labels are also coloured. (see Figure 2.4).

Finally, the label on the gender axis is reoriented and repositioned in 3D so that it can be read more easily (see Figure 2.5).

Figure 2.1: A simple scatterplot.
CHAPTER 2. SIMPLISP

Figure 2.2: A scatterplot and a 3D plot together on the same page.

Figure 2.3: The plots in Figure 2.2 with the y-axis modified.

Figure 2.4: The plots in Figure 2.3 with individual data symbols and tick-mark labels modified.
CHAPTER 2. SIMPLISP

This example of a statistical graphics session may appear quite straightforward (and it should), but it illustrates several basic features that are not available in most statistical graphics software. First of all, it is often not possible to freely mix 2D and 3D graphs—they are treated as quite separate entities. Secondly, it is usually not possible to perform the same modification on several graphs at once. For example, if a page contains several graphs, all of which have the same y-axis, it would be useful to be able to modify a feature of the y-axis for all graphs at once, rather than having to repeat the modification for each graph. Thirdly, it is often not possible to modify the characteristics of individual components within a plotting element (e.g., the individual symbols within a series and the individual tick-marks within an axis). Fourthly, interaction with a 3D graph is often inferior to the interaction possible with a 2D graph. For example, it is not usually possible to select the elements of a 3D graph with the mouse. Similarly, the amount of customisation is typically inferior for 3D graphs. For example, there is often less control over the appearance of the axes on a 3D plot. Finally, although modern statistical graphics software often boasts about how much graphs can be customised, there is not a complete freedom to modify all of the components of a graph in any possible way.

The experimental software described in this chapter is designed to provide the features mentioned above.

This chapter describes the Simplisp statistical graphics package. There are six sections: the first section introduces the issues that motivated the development of Simplisp and describes the aims of the project; the second and third sections give an outline of the overall design of Simplisp and how that design was implemented; the fourth section provides a reference for all of the functions available to the user in Simplisp; the fifth section provides a brief tutorial on how to use the system and shows examples of what can be achieved with Simplisp; the sixth section discusses how well the aims of the project were met.
2.1 Aims

2.1.1 Fully incorporating 3D graphs

All modern statistical graphics software provides some sort of three-dimensional (3D) graphing facility.

In many cases, a 3D plot is provided for data inspection. For example, by rotating a 3D plot it is sometimes possible to identify simple relationships between three variables. In these cases, the graph tends to have a very basic structure, for example, the axes may have labels, but no indication of scale. However, when the graph is a static (non-interactive) image (for example, when the user has interactively discovered an interesting view of the data and wants to generate a presentation-quality graph of that view) there is more use for the normal trappings of a graphical image (e.g., ticks and labels, text annotations, legends, and so on).

The capabilities for modifying and annotating a 3D graph are typically inferior to those for a 2D graph. For example, the control of axes, positioning of labels, or the selection of data-symbol type may be less flexible or even missing for a 3D graph. Also, there is typically no facility for placing new objects within the 3D scene. For example, a text annotation might be positioned with respect to the 2D projection plane, but not with respect to the original 3D scene.

Part of the motivation for the Simplisp package was the desire to allow the same amount of control over a 3D graph as a 2D graph and to investigate the potential for making use of the extra dimension to provide an even greater amount of control.

2.1.2 Accessing every detail of a graph

The developers of early statistical graphics software were able to provide users with convenient and (compared to pen and paper) fast tools for producing simple, standard graphs. However, they were constrained by the speed and size (memory-wise) of their technology (and the technology that the user was likely to have) so that they could not provide the user with a great deal of control over the structure of the final graph (e.g., old character-based SAS graphics). As software and hardware have improved, the amount of control that the user can exert over the final appearance of a graph has increased dramatically. How much control does the user require?

![Figure 2.6: A "temperature" axis in standard form and with useful customisations.](image-url)
CHAPTER 2. SIMPLISP

Consider a graph of the temperature of a body of water over the course of an experiment. The graph consists of an x-axis, which indicates the time scale, a y-axis which indicates the temperature scale, and a series of data points which represent the various temperature readings during the experiment. The values of 0°C and 100°C (the freezing and boiling points of water) are significant values in this context. It would certainly be useful to be able to adjust the scale on the y-axis so that tick-marks occur at these temperature values. It would also be useful to be able to increase the size of the tick-marks at these two temperature values in order to emphasise them. Even further, it would be nice if the label on the tick-mark at 0°C could be coloured blue and the label on the tick-mark at 100°C could be coloured red (see Figure 2.6). The point is that it is not very difficult to think of a situation in which there are good reasons for being able to control the look of a graph right down to the level of graphical primitives (lines and text).

Some packages do allow, in one way or another, the user to access the very lowest level of graphic primitives from which a graph is constructed. For example, SAS allows the user to invoke a graph editor which converts the graph into lines and text. Similarly, S-Plus graphs can be exported as lines and text, in a format that allows them to be loaded into the xfig program so that an S-Plus graph can be edited, as lines and text, in an arbitrary manner within xfig. One problem with most of these systems is that the low-level editing facility is not properly integrated with the high-level graphing facility; the graph is constructed from a high-level description and then must be converted into some other format for low-level editing and/or annotation. A major disadvantage of this approach is that there is typically no way to reverse the process and convert a graph from the low-level format back to the original high-level format. This means that further modifications of the high-level description of the graph are not possible (unless the graph is created again, but that will lose all of the low-level modifications!). A goal for the Simplisp package is to have a single integrated system for constructing and editing graphs at any level of description.

Another problem with many systems is that control is available at the high-level (e.g., the number of tick-marks and the shape of data-points) and at the low-level (i.e., the modification of graphic primitives), but not at intermediate levels of description. For example, it is not usually possible to specify the characteristics of an individual tick-mark. This is not a matter of modifying the low-level primitives (the line and text that make up the tick-mark), rather it involves modifying the description of the tick-mark as a whole. Another motivation for the Simplisp package is to investigate how to provide control over intermediate-level features of the graph, such as the direction and size of each individual tick-mark.

2.1.3 Sharing elements between graphs

It is common to generate a number of quite similar graphs at once. For example, in the presentation of the results of an experiment involving a small number of subjects, the results of each subject may be plotted on a separate graph. For the purposes of comparison, the graphs may be identical in all respects (axes, legends, and so on) except the data (series of points and lines). In such a situation, it is tedious to have to specify the same details such as numbers of tick-marks and ranges on axes for each graph.

In a command-line system, the problem is relatively easily overcome by retaining the text of the command which produces the appropriate plot and simply modifying the data to be plotted each time. If the system allows it, the appropriate graph
format can even be included in a function. For example, in S-Plus, the following function creates a graph with constant axes, but different data points depending on the values of x and y that are passed to it:

```r
std.plot <- function(x, y)
{
  plot(x, y, xlim=c(0, 10), ylim=c(1, 100), xlab="Session", ylab="Score")
}
```

In an event-driven system, a new graph is created with default settings, which are modified via dialogs at creation time or subsequently by double-clicking appropriate areas of the graph. For the purpose of creating several similar graphs, some systems (e.g., SigmaPlot) allow a graph to be saved as a template so that further new graphs created from this template have the template settings as their default settings.

A more serious problem arises when, having set up a number of graphs which all have similar settings, the user decides to change one of the settings that the graphs share (e.g., the number of tick-marks on the x-axis). Again, modifying each separate plot in the same way is very tedious.

Another option is to start again from the beginning. In a command-line system, the user can modify the recorded commands (or function) and re-enter them, which is relatively quick and easy. However, in an event-driven system, the user has to modify the template and recreate each instance. This is a less desirable option because templates will not necessarily include all of the information about each graph. For example, text annotations may need to be recreated.

Another aim of the Simplisp system is to investigate a more convenient way of sharing information between separate graphs.

### 2.1.4 3D selection

A convenient feature of many event-driven statistical graphics systems is the ability to select elements of a graph with clicks of a mouse, either to highlight data points on a graph or to pop-up a dialog for editing a graph element. A problem that arises in this sort of selection process is that sometimes a mouse click can be ambiguous; it is not always clear which graph element should be selected. For example, if two graph elements overlap (e.g., a text annotation overlaps a data series) and a mouse click occurs in the overlapping region, it is unclear which element should be selected (see Figure 2.7).

One solution which is often adopted is to specify a back-to-front ordering of the graph elements. A mouse click will select an element which is closer to the front in preference to one that is further to the back. This does mean that it is possible for an element to be hidden, in the sense that it cannot be selected with the mouse, if it is completely overlapped by another element which is closer to the front. This problem is easily overcome by allowing the user to send an element to the back (i.e., manipulate the back-to-front ordering; see Figure 2.8).

Unfortunately, the problem is considerably worse in a 3D graph, mainly because there is much more opportunity for overlaps to occur. This means that it is not usually possible to select the elements of a 3D graph using mouse clicks.

As part of the full integration of 3D graphs, Simplisp provides a mechanism which allows the user to select elements using mouse clicks in a situation where many elements overlap each other.
Figure 2.7: Two rectangular graphical objects which overlap (the dark-shaded region). A mouse click in the region of overlap is ambiguous with respect to which rectangle should be selected.

Figure 2.8: Two rectangular graphical objects with a back-to-front ordering. In (a) the white rectangle is behind the shaded rectangle. In (b) the white rectangle is in front of the shaded rectangle.
2.1.5 Arbitrary Plot Structures

Most statistical graphics software treats a graph as a fairly rigid structure. For example, a scatterplot typically consists of an x-axis, a y-axis, one or more data series (symbols and lines), plus possibly a secondary y-axis. The software usually supplies some way of annotating the graph with graphical primitives (text and lines), but the basic structure is fixed. This rigidity of structure is typically offset by the provision of a large range of different structures for the user to choose from. However, there are inevitably inconvenient gaps in such a selection and there is no capability for experimentation with new structures.

The S-Plus system provides some flexibility of structure by allowing the addition of axes, as well as lines and text, to an existing graph. However, this system still has a quite rigid structuring of its coordinate systems, which makes it inconvenient to, for example, add a secondary y-axis with a different scale (e.g., a plot of height which shows height in centimetres on the primary y-axis and shows height in inches on the secondary y-axis).

A goal of the Simplisp system is to allow the user to create arbitrary plot structures. In particular, the user should be able to add and remove not only graphical primitives, but also complex statistical elements. Further, the user should be able to specify arbitrary coordinate systems for different elements.

SUMMARY Simplisp aims to provide an integrated environment for the production and modification of 2D and 3D graphs. In particular, all elements of both 2D and 3D graphs will be modifiable and will be able to be selected using a mouse. Graphs will be able to share elements so that they can be modified simultaneously and graphs will be able to be constructed from arbitrary combinations of plotting elements.
2.2 Design

The Simplisp system consists of two major coordinate systems: a 3D system for constructing graphical scenes and a 2D system for viewing the scenes.

2.2.1 Constructing a Scene

The units for constructing a scene are called *collections*. The collections available are tick-marks, axes (x-axes, y-axes, and z-axes), data-symbols (xy-squares, xz-squares, yz-squares and cubes), data-series, runes (individual characters), and scripts (pieces of text).

Every collection occupies space in the 3D coordinate system, which is known as Object World Coordinates (OWC). For example, a script might be centred at the location (1.4, 2.3, 0) in OWC (i.e., to the right and above the origin of OWC) with a width of 0.5 (in the x-direction), a height of 0.1 (in the y-direction) and zero depth (in the z-direction; i.e., oriented parallel to the xy-plane of OWC).

The location and size of some collections are not specified directly in OWC (e.g., the location of a data-symbol). Instead, the user supplies one or more data values (a *case* or a *variable*), which are used to determine the location or size of the collection. A case represents a single numerical data value in Data Coordinates (DC) and a variable represents a list of numerical data values in DC. For example, a data-series must be given an x variable and a y variable to determine the locations of its data-symbols.

A *data scale* describes a mapping between a range in one dimension of OWC and a range in DC. For example, a data scale might map the range 0 to 50 in DC to the range 0 to 1 in the x-dimension of OWC. Data scales are used to transform a data value into a location (or dimension) in OWC. For example, given the above scale, the data value 25 is converted to the x-location 0.5 in OWC. Axes use data scales to determine the labels on their tick-marks. Data-symbols use data scales to convert the values in their cases into locations in OWC.

2.2.2 Viewing a Scene

The computational methods for presenting a 3D scene on a two-dimensional (2D) viewing device are well-established (Hearn & Baker, 1986; Foley, van Dam, Feiner, & Hughes, 1990); given an eye-point from which to view the scene and a viewing direction (both specified in 3D), points in the scene are projected onto a plane which is located at (or just in front of) the eye-point and orthogonal to the viewing direction. The projection can be parallel (so that parallel lines in the scene end up as parallel lines in the projection) or it can involve perspective (so that objects which are further away in the scene appear smaller in the projection and lines which are parallel in the scene tend to converge in the projection). The visible portion of the scene is determined by the viewing direction and the dimensions of the projection plane.

In Simplisp, a scene is viewed via an *output*, which is either a window on the screen or a postscript file (these are 2D surfaces). The link between a scene in 3D and a view of that scene on an output is provided by an *image*. An image consists of a list of collections to view, a *view* (see below), a *projection* (see below), and a viewport (a rectangular region on the output surface). A view defines a viewing plane, which is a bounded plane within the 3D coordinate system. A projection
defines how 3D collections are projected onto the viewing plane. Figure 2.9 shows how the different viewing elements fit together.

There can be multiple outputs open at once. There can be multiple images in each output and the same image may appear in more than one output (for example, an image can be viewed in a window on the screen and in a postscript file at the same time). Similarly, views and projections can be used in more than one image (for example, there is a default projection which performs an orthogonal projection—parallel lines in the scene end up as parallel lines on the output—and most images will just use this). There can be arbitrarily many collections in an image and each collection may appear in more than one image (for example, two images can present two different views of the same scene).

2.2.3 Automatic updating

The Simplisp system is designed so that the outputs are automatically updated whenever an image, view, projection, collection, data, or data-scale is modified. For example, suppose that an image consists of an x-axis plus a data-series, both with the same x-scale. If the x-scale is modified, the locations of data-symbols in the data-series are updated and the locations and labels of the tick-marks in the x-axis are updated and the image is redrawn in any outputs that it appears in. Similarly, if the x-variable of the data-series is modified, then the locations of the data-symbols are updated and the image is redrawn.

In this way, the collections in a scene always reflect the current state of the data, and the views that appear in the outputs always reflect the current state of the collections.

2.2.4 Selecting with a mouse

If a scene is being viewed in a window on screen, an element of the scene can be selected by clicking on it with the left mouse button. In addition, the components of an element can be selected by clicking on the appropriate component when the element is already selected. For example, if a scene includes an x-axis and the user has clicked on the axis to select it, then a click on one of the tick-marks of the axis will select just that tick-mark. This refinement of the current selection is effective down to the level of graphical primitives. For example, having selected a tick-mark on an axis, the user can click again on the label of the tick-mark in order to select the label.

It is possible to navigate from a component of an element back up to the element by clicking anywhere with the right mouse button. For example, having selected one tick-mark on an axis, if the user clicks with the right mouse button, the entire axis is selected.

**SUMMARY** A graph is constructed by creating a number of 3D plotting elements (called collections) and grouping them together within an image. The graph is viewed by creating a 2D output and specifying a view and a projection which describe a mapping of the plotting elements in 3D onto the 2D surface of the output. Plotting elements can be selected with mouse clicks within an output. Modifications of data are automatically propagated to dependant plotting elements and modifications to plotting elements automatically trigger a redraw on appropriate outputs.
Figure 2.9: The viewing elements of the Simplisp system.
2.3 Implementation

Simplisp was implemented in Common Lisp (Steele, 1990; Tatar, 1987; in particular, CMU Common Lisp, hereafter CMUCL; MacLachlan, 1992), using CLOS (the Common Lisp Object System; Steele, 1990; Keene, 1989). CMUCL was chosen because it provides an interpreted environment which has many advantages for the development of software; it is easy to develop small code-fragments, it provides a command-line interface for free (even if the Lisp notation takes a little getting used to), and it allows the system to be extended at run-time.

The object-oriented programming paradigm was also chosen for the advantages it provides in the development, maintenance, and extensibility of code (Rankin, 1995), especially in a graphics system (Hurley and Oldford, 1991; Stuetzle, 1987; Tierney, 1990).

Appendix A provides an introduction to programming in Common Lisp and to the features of object-oriented programming. The following sections assume a familiarity with the material contained in Appendix A.

2.3.1 Simplisp Classes

Simplisp is an object-oriented system consisting of a number of classes (types of objects) and generic functions (messages that the objects send to each other). This section describes the most important classes and functions within the Simplisp system.

Fundamental Classes

There are six fundamental classes within the Simplisp system: the input class, the output class, the ancestor class, the descendant class, the provider class, and the updatable class.

The input and output classes are the basis for the output devices on which scenes are ultimately viewed (X11 windows and Postscript files). An output has a slot containing a list of images. When an output is required to redraw its contents, it draws each image in this list. When an image is added to an output (i.e., appended to the output's list of images), that output is also added to the image's list of outputs (see below). This allows the image to let the output know when it is time to redraw. Similarly, when an image is removed from an output, the output is removed from the image so that the image will not cause unnecessary redraws. The output class defines a method for the generic ping-output function (see below), which draws all of the outputs image's. Objects which want to signal to an output that it is time for a redraw call this function. The output class also defines a number of generic functions for performing graphics operations, such as drawing lines and text (e.g., move-to and line-to). The different classes that are derived from the output class define different methods for these functions (e.g., the x11window class defines methods for sending drawing commands to the X server so that drawing occurs on the screen and the postscript-file class defines methods for writing postscript commands to a file). Objects which need to draw on an output call these functions. The input class defines generic functions for handling user input (such as activity with a mouse or the keyboard).

The ancestor and descendant classes provide the basis for arranging objects into hierarchies (see below). An ancestor has a slot containing a list of children and a descendant has a slot for its parent. When a descendant becomes a child of an
ancestor, the descendant is added to the parent’s list of children and the ancestor is made the parent of the descendant.

The provider and updatable classes provide the basis for the automatic updating of output devices. The provider class has a slot containing a list of dependants; a provider object represents something that provides information or resources for other objects, which are therefore dependant on the provider. The updatable class has a slot containing a list of outputs; an updatable object represents something that somehow influences what is drawn in an output and is able to alert the output when a redraw is required. When a provider needs to update its dependants, it calls a generic function ping-dependant on each of its dependants. This provides each dependant with an opportunity to make changes in response to changes in the provider. When outputs are added to an updatable (i.e., appended to the updatable’s list of outputs), the updatable calls the generic function add-outputs-to-children. This allows the updatable to maintain its children’s output lists (if it has any children) to ensure that, for example, if object A has output O in its output list then all of the children of object A also have O in their output lists. The function is generic because different types of updateables have different numbers of children, arranged in different ways. Similarly, if outputs are removed from an updatable, the updatable calls the generic function remove-outputs-from-children. When an updatable is required to update its outputs it calls the generic function ping-output for each output. This tells the outputs to redraw their contents.

Resource, Viewing, Inputoutput, and Graphical Classes

Classes which are derived from the fundamental classes can be divided into four major groups: inputoutput classes, viewing classes, graphical classes, and resource classes.

The resource classes include the the data-case class, the variable class, and the font class. All of these are derived from the provider and updatable classes (see Figure 2.10), which means that they provide information for other objects and they alert their outputs when a redraw is required. A data-case is an object which consists of a data value and a flag to indicate whether the case has been selected; a variable consists of a list of data-cases. Instances of these classes are used to represent the data to be plotted. A font is an object which contains a description of how characters are to be drawn. When a data-case or variable or font is modified, it updates its dependants (by calling ping-dependant) and updates its outputs (by calling ping-output). The first action provides an opportunity for the dependants to make modifications of their own in response to the changes in the data-case, variable, or font, and the second action ensures that every relevant output device redraws its contents to reflect the changes.

The viewing classes include the projection class, the view class, and the image
class. All of these are derived from the updatable class (see Figure 2.11), which means that they alert their outputs when a redraw is required. A view object consists of a description of a viewing plane for viewing a scene (the location, dimensions, and orientation of a plane in 3D). A projection object consists of a description of how a 3D scene will be projected onto a viewing plane (orthogonal or perspective). An image object consists of a viewport (a rectangular region on the output device), a view, a projection, and a list of objects to view; this defines the scene which is to be viewed and the transformation from the 3D scene onto the 2D output device. When any viewing object is modified, it updates its outputs (by calling ping-output). This ensures that, if the composition of a scene or the way in which it is viewed is modified, the relevant outputs will redraw their contents to display the new scene or the new view of a scene. The image class is different from the other viewing classes because it has slots which contain other objects (the view, the projection, and the list of objects in the scene). This means that an image object has some work to do to help maintain the list of outputs for each of these objects. When an object is added to an image, the image instructs the arriving object to add the image's current list of outputs to its list of outputs. Similarly, if an object is removed from an image, the image instructs the departing object to remove the image's list of outputs from its list of outputs. In addition, the image class defines methods for the add-outputs-to-children and remove-outputs-from-children functions, which pass on changes in the image's list of outputs to the view, the projection, and the objects in the scene. The upshot of all this is that every output in the image's list of outputs also appears (at least once) in the list of outputs for the view, for the projection, and for each object in the scene. Finally, when an image is drawn it calls the generic function project-object3D on each object in the image's list of objects.

The inputoutput classes include the inputoutput class (derived from the input and output classes), the postscript-file class (derived from the output class), the x11-window class, and the motif-window class (both of which are derived from the inputoutput class; see Figure 2.12). A postscript-file object represents a postscript output file on disk. The postscript-file class defines a method for the ping-output function (which overrides the method defined by the output class) which does nothing—this means that postscript-file outputs are not constantly updated for the current state of the images being viewed (the file is actually only created when the postscript-file object is killed). The inputoutput class has a slot containing a list of shadows (see below); the shadows represent a mapping between the objects in a 3D scene and their projections on the inputoutput and are used to relate mouse-clicks on the inputoutput to objects in a 3D scene that is being viewed on the inputoutput (by calling the generic function pick-object2D). When a shadow has been picked, the inputoutput calls the generic function select-object2D to draw the selection. An x11-window object represents an X11 window on screen and
a motif-window object represents a Motif window on screen.

The graphical classes include the collection class, the shadow class, the graphic-primitive class, and the imprint class. Two further classes are derived from the graphic-primitive class—the line-primitive and the point-primitive—and two from the imprint class—the line-imprint and the point-imprint (see Figure 2.13).

The graphic-primitive class is derived from the descendant class (i.e., a graphic-primitive can have a parent, but no children). Graphic-primitive objects represent the basic building blocks for objects in a 3D scene; a point-primitive consists of a 3D location and a line-primitive consists of a list of 3D locations. The point-primitive and line-primitive classes define methods for the project-object3D generic function which perform appropriate drawing operations on the output device specified in the function call (by calling the appropriate generic functions, such as move-to and line-to).

Collection objects represent more complex objects in a 3D scene. The collection class is derived from the ancestor, descendant, and updatable classes (i.e., a collection can have a parent, it can have children, and it can alert its outputs when a redraw is required). The children of a collection are other collections and/or graphic-primitives so that a complex object in a 3D scene is actually a hierarchy or tree-like structure of objects, with collections at the nodes and graphic-primitives at the leaves. The collection class defines a method for the project-object3D function which simply calls the same function for each of the collection's children. When an image is drawn, the project-object3D call filters down until it hits a graphic-primitive which actually performs the necessary drawing. In this sense, a collection is just a group of graphic-primitives. The collection class also defines methods for the add/remove-outputs-to/from-children functions, which pass on changes to the collection's list of outputs to the collection's children. This ensures that the children of a collection know what outputs the collection is in (so that redrawing

---

**Figure 2.12:** The input/output classes in Simplisp.

**Figure 2.13:** The graphical classes in Simplisp.
The collection classes in Simplisp.

Figure 2.14: The collection classes in Simplisp.

can occur whenever any part of a collection hierarchy is modified).

The imprint class is derived from the descendant class (i.e., an imprint can have a parent, but no children). An imprint represents a memory of the 2D projection of a graphic-primitive onto an output device. A point-imprint consists of a 2D location and a line-imprint consists of a list of 2D locations. The imprint class defines a method for the generic function pick-object2D which does nothing (i.e., it is not possible for the user to pick a graphic-primitive). The imprint class also defines a method for the select-object2D generic function which draws the imprint in a special selection-colour (by calling the appropriate generic drawing functions).

The shadow class is derived from the ancestor and descendant classes (i.e., a shadow can have both a parent and children). When a 3D object is projected onto an output device, a tree-like structure consisting of shadows and imprints is created which mirrors the tree-like structure of the 3D object; where there is a collection in the 3D tree there is a shadow in the 2D tree, and where there is a graphic-primitive in the 3D tree, there is an imprint. A shadow consists of a rectangular boundary and represents a memory of the extent of the 2D projection of a collection onto an output device. The shadow class defines a method for the pick-object2D function which determines whether the supplied location is within the shadow’s rectangular boundary. The shadow class also defines a method for the select-object2D function which simply calls the same function for each of the shadow’s children. When a selection is drawn, the select-object2D call filters down through the shadows until it hits an imprint which actually performs the necessary drawing. Note that there may be more than one tree consisting of shadows and imprints for each object in a 3D scene (for each collection/graphic-primitive tree); there will be one such tree for each input/output device that the object is being viewed on.

Collections

The classes which are derived from the collection class represent the objects from which statistical images can be composed: tick-marks, axes, symbols, series, and text.
A collection consists of both a description of the structure of a statistical object—for example, an axis has slots which describe the number of ticks on the axis, the text of the axis label, the size of the ticks, and so on—and a number of objects which represent that structure—for example, an axis has a number of tick-mark objects to represent the tick-marks and a text-primitive to represent the label, and a tick-mark has a line-primitive to represent the tick and a text-primitive to represent the label.

By having objects which represent the structure of a collection, it is possible to deal directly with a component of a complex plotting element (e.g., a single tick-mark on an axis), because the tick-mark is represented by a separate object, but it is still possible to perform useful, convenient, high-level operations on the structure by dealing with the description of the top-level collection (which then deals directly with the lower-level objects; for example, modify the number of tick-marks on an axis).

A collection is more than just a group of graphic-primitives because the code for editing a collection (i.e., the code for editing the high-level description of a collection) causes changes in the children of the collection as well.

Simplisp makes use of vector fonts for drawing text. This means that text is just composed of lines and can be transformed like any other element of a graph; in particular, text can be part of a 3D scene which is viewed from an arbitrary location. This also means that Simplisp text is actually a collection rather than a graphic-primitive.

**SUMMARY** Simplisp is implemented in Common Lisp using CLOS. The collection and primitive classes, and the project-object3D generic function provide a basis for creating an object in OWC and projecting it onto a 2D output. The provider and updatable classes, and the ping-output and ping-dependent generic functions provide a basis for automatic updating of collections when data is modified and automatic redrawing of outputs when collections are modified. The shadow and imprint classes, and the pick-object2D generic function support the selection of 3D objects with a mouse.
2.4 Simplisp Reference

--- geometric functions ---

make-vector3D(x y z)

make-rectangle(left bottom right top)

--- data functions ---

edit-variable(value-list)
   Creates new data cases

make-variable(value-list)

modify-variable(value-list)
   Modifies the values in the existing data cases

variable-max(variable)

variable-min(variable)

--- coordinate functions ---

convert-data-to-image(value scale)

convert-image-to-data(value scale)

edit-scale-image-range(scale
   &key min max)

edit-scale-data-range(scale
   &key min max)

make-data-scale(data-min data-max
   &key (image-min 0) (image-max 1))
CHAPTER 2. SIMPLISP

---

image functions

add-object-to-image(object image)
add-objects-to-image(object-list image)
edit-image-view(image view)
edit-image-viewport(image viewport)
edit-image-projection(image projection)
make-image(&key viewport view projection objects)
remove-all-objects-from-image(image)
remove-object-from-image(object image)
remove-objects-from-image(object-list image)

---

input/output functions

add-image-to-file(image file)
add-image-to-window(image window)
kill-postscript-file(file)
kill-x-window(window)
make-postscript-file(name
 &key (width 595) (height 842))
make-x-window(&key (width 400) (height 400))
pick-object(window)
remove-image-from-file(image file)
remove-image-from-window(image window)
CHAPTER 2. SIMPLISP

---

rune functions

edit-rune-char(rune char)
edit-rune-colour(rune colour)
edit-rune-font(rune font)
edit-rune-height(rune height)
edit-rune-horizontal-alignment(rune hjust)
edit-rune-location(rune location)
   Location is a vector3D
edit-rune-orientation(rune horizontal vertical)
   Horizontal and vertical are vector3Ds
edit-rune-vertical-alignment(rune vjust)
edit-rune-width(rune width)
make-rune(char location height
   &key font hsize hjust vjust horizontal
   vertical colour)
CHAPTER 2. SIMPLISP

__________________________
script functions
__________________________

edit-script-chars(script chars)
edit-script-colour(script colour)
edit-script-font(script font)
edit-script-height(script height)
edit-script-horizontal-alignment(script hjust)
edit-script-location(script location)
    Location is a vector3D
edit-script-orientation(script horizontal vertical)
    Horizontal and vertical are vector3Ds
edit-script-vertical-alignment(script vjust)
edit-script-width(script width)
make-script(chars location height
    &key font hjust vjust horizontal
    vertical hsize colour)
tick-mark functions

\begin{itemize}
  \item \texttt{edit-tick-colour(tick colour)}
  \item \texttt{edit-tick-direction(tick direction)}
    Direction is a vector3D
  \item \texttt{edit-tick-label-chars(tick chars)}
  \item \texttt{edit-tick-label-distance(tick distance)}
  \item \texttt{edit-tick-label-font(tick font)}
  \item \texttt{edit-tick-label-height(tick height)}
  \item \texttt{edit-tick-label-horizontal-alignment(tick h-just)}
  \item \texttt{edit-tick-label-orientation(tick horizontal vertical)}
    Horizontal and vertical are vector3Ds
  \item \texttt{edit-tick-label-vertical-alignment(tick v-just)}
  \item \texttt{edit-tick-label-width(tick width)}
  \item \texttt{edit-tick-length(tick length)}
  \item \texttt{edit-tick-origin(tick origin)}
    Origin is a vector3D
  \item \texttt{make-tick-mark(length origin direction label)}
    \&key label-font label-distance
    label-hjust label-vjust
    label-vertical label-horizontal
    label-height label-width)
\end{itemize}
axis functions

edit-axis-colour(\textit{axis colour})

edit-axis-first-tick(\textit{axis first-tick})
  First-tick is a vector3D

edit-axis-label-chars(\textit{axis chars})

edit-axis-label-direction(\textit{axis direction})
  Direction is a vector3D

edit-axis-label-distance(\textit{axis distance})

edit-axis-label-font(\textit{axis font})

edit-axis-label-height(\textit{axis height})

edit-axis-label-orientation(\textit{axis hmultiplier vmultiplier})
  Hmultiplier and vmultiplier must be + or - 1. These multipliers specify
  whether the label orientation is the same as or the opposite of the orienta-
  tion of the axis major

edit-axis-label-position(\textit{axis position})
  Position gives the location of the label as a proportion of the distance along
  the axis major

edit-axis-last-tick(\textit{axis last-tick})
  Last-tick is a vector3D

edit-axis-min(\textit{axis min})

edit-axis-max(\textit{axis max})

edit-axis-num-ticks(\textit{axis num-ticks})

edit-axis-scale(\textit{axis scale})

edit-axis-tick-colour(\textit{axis colour})

edit-axis-tick-direction(\textit{axis direction})
  Direction is a vector3D
edit-axis-tick-label-chars(\textit{axis chars-list})

edit-axis-tick-label-distance(\textit{axis distance})

edit-axis-tick-label-font(\textit{axis font})

edit-axis-tick-label-height(\textit{axis height})

edit-axis-tick-label-horizontal-alignment(\textit{axis hjust})

edit-axis-tick-label-vertical-alignment(\textit{axis vjust})

edit-axis-tick-label-orientation(\textit{axis horizontal vertical})

Horizontal and vertical are vector3Ds

edit-axis-tick-length(\textit{axis length})

edit-axis-x-offset(\textit{axis offset})

edit-axis-y-offset(\textit{axis offset})

edit-axis-z-offset(\textit{axis offset})

\textbf{make-x-axis} (\texttt{&key min max scale num-ticks})
\begin{verbatim}
  tick-length tick-direction
  first-tick last-tick
  tick-label-chars tick-label-font
  tick-label-distance
  tick-label-hjust tick-label-vjust
  tick-label-vertical
  tick-label-horizontal
  tick-label-height label-chars
  label-font label-direction
  label-distance label-position
  label-hmultiplier
  label-vmultiplier label-height
  label-width colour tick-colour
  y-offset z-offset)
\end{verbatim}

\textbf{make-y-axis}() see \textbf{make-x-axis}

\textbf{make-z-axis}() see \textbf{make-x-axis}
data-symbol functions

edit-symbol-colour(symbol colour)

edit-symbol-size(symbol
&key xsize ysize zsize)

make-data-symbol(symbol-type x-case y-case z-case xsize ysize zsize
&key x-scale y-scale z-scale colour)

series functions

edit-series-colour(series colour)

edit-series-symbol(series symbol-type)

edit-series-symbol-size(series
&key zsize ysize zsize)

edit-series-x(series scale)

edit-series-y(series scale)

edit-series-z(series scale)

edit-series-x-scale(series variable)

edit-series-y-scale(series variable)

edit-series-z-scale(series variable)

make-data-series(&key x y z x-scale y-scale z-scale
xsize ysize zsize data-symbol
colour)
2.5 User’s Guide

Simplisp is not in a state to be distributed publicly so the instructions given in this section assume that the user is sitting at the author’s machines in the appropriate directory, and the instructions are unabashedly UNIX- and x11-centric. The purpose of this section is to give an idea of what the user has to do to create a plot and to demonstrate some of the useful things the user can do; it is also to provide the ammunition for a critical analysis of Simplisp’s weaknesses.

2.5.1 Getting Started

The first step is to start CMUCL by typing cmucl at the system prompt. (All further typing should occur at the Lisp prompt.) Next, Simplisp must be loaded by typing (load "init.lisp").

2.5.2 Creating a plot

The first step is to generate the data to plot:

(setf x (make-variable (list 1 3 5 6)))

Next, create the objects in the scene. This involves creating a scale for the x-dimension, creating an x-axis, and creating a data-series:

(setf x-scale (make-data-scale 0 8))
(setf x-axis (make-x-axis :scale x-scale :y-offset -.1))
(setf series (make-data-series :x x :x-scale x-scale))

Now, describe how the scene will be viewed (in this case, via an orthogonal projection onto a viewing plane that is parallel to the x-y plane [the default] and is bounded by the specified rectangle):

(setf view
     (make-view :window (make-rectangle -.5 -1 1.5 1)))
(setf proj (make-projection nil))

Now, assemble the objects into a scene:

(setf image (make-image :view view :projection proj))
(add-objects-to-image (list x-axis series) image)

Finally, create an output device (in this case an x11-window) for viewing the scene (see Figure 2.15):

(setf my-window (make-x-window))
(add-image-to-window image my-window)

2.5.3 Selecting Elements of a Plot

It is possible to access some elements of a plot through Lisp symbols. For example, various attributes of the x-axis can be modified by typing

(edit-axis-label-distance x-axis 0.35)
(edit-series-symbol-size series :xsize 0.03)
Figure 2.15: A Simplisp plot viewed in an X11 window.

The first expressions moves the label of the x-axis down further and the second expression makes all of the data symbols narrower (in the x-dimension). Some elements of the plot do not have a corresponding Lisp symbol (e.g., the tick-marks and the individual symbols in the data series). These elements may be accessed by typing

\[(\text{pick-object my-window})\]

and using judicious clicks of the mouse.

As discussed in Section 2.3.1, the elements of the plot are arranged in a hierarchy (see Figure 2.16). Left mouse clicks (LMCs) select down the hierarchy and right mouse clicks (RMCs) select up the hierarchy. A middle mouse click (MMC) returns the current selection.

For example, an LMC on the x-axis selects the entire axis, and an LMC on the series selects the series (all of the symbols). Having selected the axis, a further LMC on one of the tick-marks selects the appropriate tick-mark, and a click on the label selects that (this is called picking a child). Having selected one of the tick-marks, a further LMC on the label of the tick-mark selects just the label. With a tick-mark label selected, an RMC selects the whole tick-mark again (this is called picking the parent). With either a tick-mark or the label selected, an RMC selects the entire axis again. With the series selected, an LMC on one of the symbols selects that symbol; an RMC selects the series again. With a tick-mark selected, an LMC on one of the other tick-marks selects that tick-mark instead (this is called picking a sibling).
An element can be modified by embedding the `pick-object` call within an editing expression. For example, by typing

```lisp
(edit-tick-length (pick-object my-window) .1)
```
then selecting the middle tick-mark (with an LMC on the x-axis followed by another LMC on the middle tick-mark), then returning the middle tick-mark (with an MMC), the length of the tick on the middle tick-mark will be doubled. The tick-mark label is moved down and made larger by typing

```lisp
(edit-tick-label-distance (pick-object my-window) .1)  
(edit-tick-label-height (pick-object my-window) .1)
```

Recall that the plot is actually a 3D scene. To demonstrate this, the plot can be viewed from a different angle by typing

```lisp
(edit-view view :horizontal (/ pi 4) :vertical (/ pi 6))
```
(The plot is now being viewed from a location above and to the right of the plot—originally the plot had been viewed from directly in front; Figure 2.17 shows the new view)

The elements of the graph can still be selected with the mouse. For example, by typing

```lisp
(edit-tick-label-orientation (pick-object my-window)
  (make-vector3D 0 0 -1)
  (make-vector3D 0 1 0))
```
(The label of the middle tick-mark is now in the y-z plane; the other labels are still in the x-y plane.) Figure 2.18 shows the final image.
The plot in Figure 2.15 after modifications to the axis label and central tick-mark, viewed from a different 3D location.
Figure 2.18: The plot in Figure 2.17 with the label of the central tick-mark rotated in 3D.
2.5.4 Sharing elements between graphs

A single plot element may be used within more than one plot. Suppose that the data from two different subjects in an experiment are to be plotted. The variables are session-number and score. The data are entered by typing

```lisp
(setf session (make-variable (list 1 2 3 4 5 6 7)))
(setf score-1 (make-variable (list 4.5 1.5 3 3.5 1.5 2 3)))
(setf score-2 (make-variable (list 4.5 1.5 4 3.5 4.5 4 2)))
```

Two separate plots can be created by typing

```lisp
(setf plot-1 (make-image :viewport (make-rectangle 0 .25 .5 .75)))
(setf plot-2 (make-image :viewport (make-rectangle .5 .25 1 .75)))
```

Each plot will have its own data series, but the plots will share the same x-scale and y-scale, and the same x-axis and y-axis (see Figure 2.19).

```lisp
(setf x-scale (make-data-scale 0 8))
(setf y-scale (make-data-scale 1 5))
(setf series-1 (make-data-series :x session :y score-1
:scale x-scale
:scale y-scale))
(setf series-2 (make-data-series :x session :y score-2
:scale x-scale
:scale y-scale))
(setf x-axis (make-x-axis :scale x-scale
:label-chars "session number")
(setf y-axis (make-y-axis :scale y-scale
:label-chars "score")
```
CHAPTER 2. SIMPLISP

Figure 2.20: Two plots with shared elements.

plot-1)
( ADD-OBJECTS-TO-IMAGE (LIST X-AXIS Y-AXIS SERIES-2)
plot-2)
(setf another-window (make-x-window))
(ADD-IMAGE-TO-WINDOW PLOT-1 ANOTHER-WINDOW)
(ADD-IMAGE-TO-WINDOW PLOT-2 ANOTHER-WINDOW)

Figure 2.20 shows the resulting plots.

Now it is possible to edit the shared aspects of both plots at once. Suppose that there are too many tick-marks on the x-axes, so the number of tick-marks is reduced to three by typing

(edit-axis-num-ticks x-axis 3)

and the scores are actually out of 12 (with a minimum score of zero) so the scale on the y-axes (also used by the series) is modified to reflect the range of possible scores by typing

(edit-scale-data-range y-scale :min 0 :max 12)

Figure 2.21 shows the results of these modifications.

2.5.5 Creating Arbitrary Plots

There is no set structure to a plot in Simplisp; Statistical elements can be combined in an arbitrary fashion. Consider a simple scatterplot of height (in centimetres) against age (see Figure 2.22).

(setf age (make-variable (list 1.3 2 3.7)))
(setf height (make-variable (list 50 55 75)))
(setf x-scale (make-data-scale 0 4))
(setf y-scale (make-data-scale 40 80))
(setf x-axis (make-x-axis :scale x-scale
:label-chars "AGE (YRS)"
:num-ticks 3))
(setf y-axis (make-y-axis :scale y-scale
:label-chars "HEIGHT (CM)"
)
CHAPTER 2. SIMPLISP

Figure 2.21: The plots in Figure 2.20 after modification of the shared elements.

Figure 2.22: A scatterplot of height (in centimetres) against age (in years).

:(num-ticks 3))
(setf series (make-data-series :x age :y height
:x-scale x-scale
:y-scale y-scale))

(setf plot (make-image))
(add-objects-to-image (list x-axis y-axis series) plot)
(setf window (make-x-window))
(add-image-to-window plot window)

It would be useful to have a secondary y-axis which indicated the height in inches. Such an axis will have its own scale and rely on its location to be coherent with the existing elements of the plot.

(setf y-scale-2 (make-data-scale 15.75 31.50))
(setf y-axis-2
(make-y-axis :scale y-scale-2

44
It would also be useful to have a tick-mark to indicate the average age at which a child begins to walk.

The first expression above calculates where the tick-mark should be located to correspond to an age of one year. The second expression above moves the axis label away to make room for the new tick-mark. Figure 2.23 shows the modified plot.

In addition to allowing new elements to be added to a plot, Simplisp allows the user to remove any element. For example,

```
(remove-object-from-image extra-tick plot)
```
removes the tick-mark that was just added. The secondary y-axis and any of the original objects in the plot could also be removed in a similar fashion.

2.5.6 Getting stopped
To exit Simplisp, simply exit CMUCL by typing (quit).

2.5.7 An Advanced Simplisp Example
This section presents the Simplisp code that will produce the graphs demonstrated at the beginning of the chapter (Figures 2.1 to 2.5).

;;; generate simple scatterplot
(setf win (make-x-window))
(setf v1 (make-view))
(setf i1 (make-image :view v1))
(setf x (make-variable (list 2 2 3 3 4 4 5 5 6 6)))
(setf y (make-variable
          (list 1.1 1.3 3.1 1.3 4.3 1.9 4.4 2.3 4.7 4.5)))
(setf xs (make-data-scale 1 7))
(setf ys (make-data-scale 1 5))
(setf x-axis (make-x-axis :scale xs
                          :num-ticks 4
                          :z-offset 1
                          :label-chars "age"))
(setf y-axis (make-y-axis :scale ys
                          :label-chars "score"))
(setf series (make-data-series :x x :y y
                                :x-scale xs :y-scale ys))
(add-objects-to-image (list x-axis y-axis series) i1)
(add-image-to-window i1 win)

;;; reposition scatterplot and add 3D plot
(edit-image-viewport i1 (make-rectangle 0 .25 .5 .75))
(setf v2 (make-view :window (make-rectangle -1 -1 1 1)
                     :horizontal (* pi .67)
                     :vertical (/ pi 8)
                     :look-at-point
                     (make-vector3D .5 .5 .5)))
(setf i2 (make-image :view v2
                     :viewport
                     (make-rectangle .5 .25 1 .75)))
(setf z (make-variable (list 1 2 1 2 1 2 1 2 1 2)))
(setf zs (make-data-scale .5 2.5))
(setf z-axis
     (make-z-axis :scale zs :num-ticks 2
                  :first-tick
                  (convert-data-to-image 1 zs)
                  :last-tick
                  (convert-data-to-image 2 zs)
                  :label-chars "gender"
:label-hmultiplier 1
:tick-label-chars
  (list "girls" "boys")

(setf series2
  (make-data-series :x x :y y :z z
    :x-scale xs
    :y-scale ys
    :z-scale zs
    :data-symbol
      'xy-square-symbol))

(add-objects-to-image (list x-axis y-axis
                         z-axis series2)
   i2)

(add-image-to-window i2 win)

;; modify common y-axis
(edit-axis-num-ticks y-axis 3)

;; colour symbols and tick-marks
(edit-symbol-colour (pick-object win) _blue_)
(edit-symbol-colour (pick-object win) _pink_)
(edit-script-colour (pick-object win) _blue_)
(edit-script-colour (pick-object win) _pink_)

;; reposition z-axis label in 3D
(edit-script-orientation (pick-object win)
    (make-vector3D 1 0 0)
    (make-vector3D 0 1 0))

(edit-script-horizontal-alignment (pick-object win)
  _right_)
2.6 Results and Conclusions

This section discusses how well the design and implementation of the Simplisp system met the aims that were set out in Section 2.1. As well as pointing out the useful and successful features of the system, there will be a critical analysis of the system's weaknesses.

2.6.1 3D-based graphics

All plots in Simplisp are created in 3D. With this approach, a standard 2D plot is just a special (simple) sort of 3D plot; a standard 2D plot is created by assembling a scene with components in the x-y plane and viewing the scene (via an orthogonal projection) through a viewing plane which is parallel to the x-y plane. Because there is only one type of plot, with 2D and 3D merely representing different ways of viewing the same type of scene, anything that can be done with a 2D plot can also be done with a 3D plot.

The major advantages of this approach are:

- 3D axes have all of the flexibility of 2D axes. For example, it is possible to specify the range and number of tick-marks on 3D axes.

- the elements of a 3D plot can all be selected with the mouse. For example, it is possible to select and modify the label on a 3D axis.

- it is possible to annotate a graph in 3D. For example, a label can be positioned at the 3D location of a data-symbol.

One problem that was encountered with this approach was that, by forcing all plots to be 3D, the set of possible operations on a plot was constrained. In particular, whereas normally it is possible to click-and-drag an object in a 2D image, this is not possible (or is at least very complicated) in a 3D image. The problem is that a drag in 2D converts to an ambiguous motion in 3D. One solution would be to convert the 2D drag into a 3D motion that is parallel to the viewing plane; this would make sense in a 2D plot (where the viewing plane is parallel to the plane in which all of the plot elements are located), but could produce quite bizarre and frightening results in a general 3D plot where the viewing plane is at an angle to all of the major axes and there may be a perspective projection operating. In some special cases, where an obvious plane of reference exists (e.g., the author has implemented dragging with the mouse in a 3D chess program, where the chess board provides a convenient reference plane), there may be a sensible conversion from 2D dragging to 3D motion, but there appears to be no obvious general solution.

Another problem with this approach is that the creation of a 2D graph becomes as complicated as the creation of a 3D graph (see Section 2.5). For example, it is necessary to specify a viewing plane in 3D in order to view a simple 2D plot. This problem is mostly a matter of there being a deficit of user-support in terms of high-level functions. At the moment, Simplisp is like a workshop full of weird and wonderful tools; it could be made to be a lot more like a friendly, helpful mechanic. If it is easy to create a 3D plot then the user is unlikely to complain that it is "as difficult" (as easy !) to create a 2D plot.

Finally, there is an efficiency price to pay for the everything-is-3D approach. It is slower to perform the projections of 3D objects onto a region of a 2D device than it is to simply transform an inherently 2D image into a particular region of a device.
(although more and more computers now include support for 3D transformations in hardware).

Overall, Simplisp was a success in terms of providing an equal treatment of 2D and 3D graphs. Unfortunately, this equality not only involved raising the stature of 3D graphs (e.g., allowing mouse access to elements of a 3D graph), but also diminishing the stature of 2D graphs (i.e., not allowing elements to be dragged with the mouse). Perhaps the best solution would be a separate approach for 2D and 3D graphs, although Simplisp demonstrates that the functionality of 3D graphs can be much closer to that of 2D graphs than is often the case. In the software described in the next two chapters, only 2D graphs are developed; 3D graphs would be developed as a separate case.

2.6.2 Accessing every detail of a graph

A plot in Simplisp is composed of an arbitrary combination of plotting elements (e.g., axes and data series). The fact that plotting elements (right down to individual tick-marks and individual data symbols) can be created separately from any overall plot structure and added directly to a graph (image) introduces a high degree of accessibility. For example, if a plot is constructed by creating an x-axis and a data series separately from each other (but with shared scales for coherence) and adding them to the same graph, the axis and the series are both accessible to the user (through Lisp symbols). This is in contrast to a system which only allows the user to create an entire plot, which contains a high-level description, including a description of the axis and the series, and the user can only modify the axis or series through the high-level description contained in the plot. The difference is like the difference between doing something yourself (Simplisp) and asking someone else to do something for you (with the risk that he or she may decide that you actually wanted something other than what you asked for).

Also, more complex plotting elements in Simplisp (such as axes) are not just elements with a more complex description. They also consist of components which represent the element's underlying structure. For example, an axis consists of a description which specifies the number of tick-marks on the axis, the label to be used and the relative positions of these things; an axis consists of a number of tick-mark objects, plus a text object (for the label). This allows the user to interact directly with lower-level elements which are part of a higher-level element (e.g., modify the colour of a single tick-mark on an axis). This is in contrast to a system in which plotting elements only consist of the description of the element's structure. In such a system, the user is restricted to the modifications which are allowed through the high-level description.

Some systems (e.g., SAS, Splus) do provide a quite detailed description of plotting elements (e.g., it is possible to specify a separate colour for each tick-mark on an axis), however, that sort of approach can become unwieldy quite quickly; the amount of description grows rapidly for more complex plotting elements (e.g., a scatterplot must have such a list for each axis); this becomes manifest in a large number of complicated dialog windows or a long list of complicated parameters for creating and editing complex plotting elements. By having a separate object to store the relevant description of each component of a plotting element, the detail is available, but it is spread out and located where it is most useful rather than all being squashed into the high-level description (although see below).

One problem is that of accessing plotting elements which are not top-level objects. A top-level object is one which has a Lisp symbol. For example, consider the
plotting elements created by the following code:

\[
\text{(setf my-axis (make-x-axis))}
\]

This command creates an x-axis object which is associated with the Lisp symbol my-axis. The command also creates a number of tick-mark objects, and a number of script objects (the tick labels) none of which are associated with any Lisp symbols. It is possible to access the x-axis object via the command-line using the Lisp symbol, for example:

\[
\text{(edit-axis-num-ticks my-axis 7)}
\]

However, it is not possible to access any of the tick-mark objects or the script objects because there are no corresponding Lisp symbols. These objects are accessible via the mouse, but it would also be useful to be able to access them via the command-line.

Another problem is that the user is unable to access the very lowest level of graphical primitives. This is a deliberate decision in Simplisp which is designed to protect the inherent structure of a plotting element; the user is not allowed to manipulate the line-primitive component of a tick-mark separately from the text-primitive component (except via the high-level description of the tick-mark). This sort of protection—deciding what is best for the user—is, unfortunately, exactly the sort of thing that Simplisp was originally designed to discourage. The user should be allowed access to whatever detail he or she decides is appropriate.

Simplisp is successful in providing access to all elements of a graph by representing the graph as a hierarchy of plotting components, each with its own identity. Ideally it would go even further and allow access right down to the graphical primitives. Also, the system needs to provide a mechanism for accessing the elements of a graph which are not top-level objects. The software described in the next chapter retains the hierarchy of components for representing a graph and provides access to all elements, including graphical primitives, via the command-line.

### 2.6.3 Sharing elements between graphs

Any plotting element in Simplisp can be used in several images (scenes) at once. For example, two separate graphs can contain the same x-axis. The advantage of this is that both graphs can be simultaneously updated, at any time, by modifying the shared x-axis object (see, Section 2.5.4). Another use of this feature would be to generate two different views of the same scene, by having two separate images, each with its own view, which contain the same set of plotting elements.

A disadvantage of this approach is that the graphs which share a plotting element must be constructed in a piecemeal fashion. For example, if two scatter-plots are to share an x-axis then both plots have to be constructed by adding an x-axis (the same one for both plots), a y-axis, and a data-series (as opposed to constructing each scatterplot as a single complete entity). In the current state of Simplisp, this is how any graph has to be created, but it would be possible and preferable to be able to create graphs in their entirety (see Section 2.6.6 below for further discussion). It is not clear that the advantages obtained from creating graphs with shared elements is worth the extra effort involved in creating them piecemeal.

The sharing of objects does not only apply to plotting elements. For example, a number of graphs (images) may share the same view. This is useful for producing the same layout in different plots (see Section 2.6.6 below for a discussion of the
if (left mouse click)
  if (no current selection)
    if (click in top-level object)
      select top-level object
  else if (click in current selection)
    if (click in child of current selection)
      select child of current selection
    else if (click in sibling of current selection)
      select sibling of current selection
  else if (click in top-level object)
    select top-level object
else if (right mouse click)
  if (there is a current selection &
      current selection has parent)
    select parent of current selection
else
  no current selection

Figure 2.24: The Simplisp 3D picking algorithm for selecting objects with the mouse.

2.6.4 3D selection

The fact that Simplisp graphs are inherently 3D, plus the fact that each plotting element consists of a hierarchy of objects, each of which the user needs to be able to select, contributes to a situation where an enormous number of objects in a view overlap each other. The solution, for selecting these objects with a mouse, is to select only at one level of the hierarchies at a time. For example, initially, mouse clicks only select objects at the top of the hierarchies. With a top-level object selected, objects at the next level within that hierarchy can then be selected. This approach dramatically reduces the amount of overlap between plotting elements and provides a convenient, quick way to access any element of a graph. Figure 2.24 shows the general selection algorithm (assuming at least a two-button mouse or equivalent).

The selection mechanism in Simplisp does work well for navigating within a tree-like structure of objects where higher-level objects completely obscure lower-level objects. There is, however, still a need for being able to perform some sort of "push-to-back" or "bring-to-front" operation to handle the case where objects at

limitations of Simplisp's layout mechanism). Also, data scales, fonts, and variables can be shared by different plotting elements. These features make it easy to maintain coherent coordinate systems between graphs, modify the style of text in many places at once with a single editing command, and implement features like linking data symbols (so that, for example, points which depend on shared data can be selected in multiple plots at once).

The most important aspect of this general sharing approach is that, having composed a graph (or several graphs), the user is able to modify the graph (or graphs) quickly and efficiently, without having to repeat modifications many times and without having to reconstruct the graphs from scratch.
the same level—objects which are not part of the same hierarchy or objects which are siblings in a hierarchy—obscure each other. This occurs when, for example, a data series obscures an axis in a 3D plot.

2.6.5 Arbitrary graph structure

There is a deliberate attempt in the design of Simplisp to try to separate the elements of a graph so that they can be created in isolation and therefore combined in arbitrary ways. For example, a data scale is a separate object from an axis and a data series; an axis has a data scale and a data series has a data scale, but there is no requirement that they share the same data scale.

Simplisp succeeds quite well in separating the various elements of a plot, but it does not provide enough of the standard combinations of these elements. There is a lot of potential for combining elements in novel and interesting ways, but there is not enough convenience for creating the standard combinations.

By way of analogy, consider a graphics system as providing moulds for graphs. The user pours data into the mould and, after some careful "baking", out comes a graph. Some systems only provide moulds for complete graphs; a new sort of graph requires a new mould. Simplisp, on the other hand, provides moulds for the many different elements of a graph so that a new sort of graph does not require a new mould, it only requires assembling the existing moulds in a new way.

The software described in the next two chapters retain the idea of a flexible graph structure, but provide much more support for different plotting elements to share default information (such as data scales).

2.6.6 General problems

This section discusses the issues that arose during the development of Simplisp that are not directly related to the initial aims of the project.

Specifying graph layout

It is not very easy to control the layout of a graph in Simplisp. A graph is composed in a 3D world (OWC) and is projected onto a 2D world (an output surface). This projection is relatively complicated, involving a projection of the 3D graph onto a rectangular window in the 3D world and then a mapping of the window into a rectangular viewport on the output surface. The software does all of the number crunching to produce the final image, but the user needs to be able to have some idea of what the final image will look like in order to specify and modify the projection, window, and viewport for an image. The default window is parallel to the xy-plane in OWC and is centred on the unit square and the default viewport is the entire output surface. This means that a graph which is composed around the unit square in OWC will nicely fill the output device. It is relatively easy to position the graph anywhere on the output surface by specifying an appropriate rectangle for the viewport (although Chapter 4 describes a much more convenient and powerful mechanism for doing this). However, it is much more difficult to specify the location of the graph within the viewport. For example, in a standard scatterplot, it is common not to centre the data region of the graph within the viewport. This is because there are axes below and to the left of the data region, but not above or to the right. Consequently, there is usually more space below and to the left of the data region (to accommodate the axes). Similarly, extra space is often provided
above the data region for a title for the graph. This sort of positioning of the graph within the viewport requires an appropriate specification of the window in OWC. This is difficult because it is not easy to intuitively grasp how modifications to a rectangle in 3D will affect the projection of 3D objects onto that window and it is even harder to intuit how these modifications will affect the location of the projected image within the viewport. Chapter 4 describes a system which allows objects to be positioned in a much more simple and intuitive manner.

**High-level plot elements**

One of the deficiencies of the Simplisp system is the fact that quite simple plots are not simple to create. The lack of convenient utility functions, essentially a lack of a user-friendly interface, is part of this problem, but another part is the lack of high-level plotting elements, such as scatterplots and histograms. Simplisp is designed to allow the user to escape the restrictions of standard plotting structures (e.g., a scatterplot can consist of more than just an x-axis, a y-axis, and a data series). However, the reason such standard structures exist at all is because they are so convenient and appropriate for many situations. The lack of high-level plotting elements in Simplisp is not due to a design restriction (a scatterplot is easily conceived of as a combination of an x-axis, a y-axis, and a data-series), but rather it is due to the implementation becoming too unwieldy.

The first problem is that the description of a high-level plotting element is very complex. The description of a high-level plotting element often includes a repetition of some of the description of the components of the plotting element. For example, the description of an axis includes a tick-length for all of the tick-marks on the axis. Higher-level plotting elements (such as scatterplots and histograms) tend to have a greater number of components and therefore tend to have larger and larger descriptions (as they include repetition for each type of component). This is not an insurmountable difficulty, but in Simplisp it leads to a larger problem.

In Simplisp, each plotting element has a suite of editing functions; one for each slot in its high-level description. For example, a tick-mark has an `edit-tick-length` function corresponding to its `:length` slot. These functions are responsible for making sure that the components of a plotting element continue to obey the high-level description of the element. For example, the `edit-tick-length` must modify the length of the line-primitive which represents the tick and modify the location of the text object which represents the tick-mark’s label to ensure that the label remains the correct distance away from the end of the tick\(^1\). For higher-level plotting elements, the consequences of editing a description slot can become quite complicated, especially if editing the component causes it to modify its own components. For example, if the user modifies the `:tick-length` slot of an axis, the axis must then modify the `:length` slot of each of its tick-marks and each tick-mark must then modify the length of its line-primitive and the location of its label. It becomes very difficult for the designer of a new plotting element to keep track of all of the possible consequences of a single edit.

The software described in the next chapter simplifies the construction of simple graphs by providing higher-level plotting elements and solves the problem of exponential growth in number and complexity of editing functions by having only

---

\(^1\)This is necessary to support automatic redrawing of outputs. When a plotting element is modified it must perform appropriate modifications on all of its components then force a redraw of all relevant outputs.
one editing function for all plotting elements and by automatically handling the consequences of an editing action.

Lack of extensibility

The previous section described how the increasing complexity of higher-level plotting elements leads to an increasing burden on the developer of the Simplisp system. If the cost for the Simplisp developer becomes prohibitive, what chance is there that a user will be able to extend the system? The Simplisp system does not provide very good opportunities for extensibility.

The major focus of the software described in the next chapter is to investigate an extensible graphical system.

SUMMARY Simplisp successfully incorporates 3D graphs with 2D graphs in terms of customisation and selection with a mouse. Simplisp is also successful in allowing a very high degree of customisation of graphs and allowing a great deal of flexibility in the structure of a graph. The following features were considered worthy of retaining in further software experiments: automatic updating of outputs, representing graphs with hierarchies of plotting elements, and flexible graph structure. The following features will be addressed by further software experiments: improved accessibility to components of plotting elements from the command-line, improved mechanisms for sharing default information between plotting elements, and greater opportunities for extensibility.
2.7 References


Suppose that an experiment is to be conducted which will measure the wool production of sheep. There will be $N$ sheep, divided into $k$ groups of size $n$ and each group will be given a different feed. The experimenters hope to determine whether the different feeds result in differences in wool production. The statistical consultant wishes to design a graphical analysis for the experimental data.

First of all, a graph is created which takes two variables—the wool production for each sheep and a group number for each sheep—and creates a dot-plot of wool production for each group (see Figure 3.1). This graph allows a simple comparison of the wool production for each group.

Next, the graph is enhanced by adding a special box to each dot-plot. Each box is centred on the mean of its dot-plot (indicated by a horizontal line) and spans one standard-deviation either side of the mean (see Figure 3.2). The graph now displays crude measures of location and spread to assist in the comparison of the groups (especially with respect to hypothesis testing of differences between the mean wool production of each group).

Now, a further dot-plot is added, which shows the group means together and has a box to indicate the overall mean of the data and a derived measure of spread between the group means\(^1\). The boxes indicating the mean and spread of each group are coloured red, as are the symbols which plot the group means in the additional dot-plot, and the box indicating the mean and spread of the group means is coloured green.

The resulting graph allows for a quick visual analysis of variance of the data. If the boxes for each group are similar to each other and to the box for the group means then there is no significant difference between the groups (see Figure 3.3). If, on the other hand, the boxes are similar for each group, but the box for the group means is much bigger, then there is likely to be a significant difference between the groups (see Figure 3.4).

The statistical graphics sessions described above may not seem very familiar. This is because the tasks described above could not be achieved in most statistical graphics packages. The following points are of particular interest:

1. There is a very loose structure to the graph; an arbitrary number of plotting

\[^{1}\text{The F-test statistic is } f_0 = \frac{s^2_B}{s^2_W}, \text{ where } s^2_B = n s^2_X \text{ and } s^2_W = \frac{s^2_1 + \cdots + s^2_k}{k}. \text{ If there is no difference between the groups, } s^2_B \text{ is approximately equal to } s^2_W \text{ which means that the standard-deviation of each group should be approximately the same as } \sqrt{ns^2_X}.\]
elements are combined to create the final graph.

2. At the same time, there is a sensible sharing of useful information. For example, the new plotting elements automatically conform to the coordinate system of the graph (so that the new plotting elements are consistent with the original plotting elements).

3. The final graph is a coherent, "intelligent" whole (rather than a collection of "dumb" components which know nothing about each other). For example, when the variables being plotted are changed, all of the plotting elements in the graph are updated for the new data (see Figures 3.3 and 3.4). Similarly, if the window is resized, the elements of the graph are resized in a coherent manner.

4. The graph is constructed incrementally via a series of cumulative additions of plotting elements, where the results of each addition are immediately available in the graph window.

The goal of the software that is described in this chapter (Xtend) is to create a system that allows the graphical tasks described above to be performed.

The chapter is divided into six sections. The first section describes the aims of the Xtend project, the second section outlines the overall design of the software, and
Figure 3.2: A graph with a dot-plot plus a box indicating mean and spread per group.
Figure 3.3: A graph for performing an analysis of variance by eye (no difference between the groups).
Figure 3.4: A graph for performing an analysis of variance by eye (significant difference between groups).
the third section describes the implementation of that design. The fourth section provides reference material for the system, the fifth section is a simple user’s guide, and the final section discusses how well the project succeeded and where it failed.
3.1 Aims

The Xtend software package was motivated by a desire to build on and improve the Simplisp package. Xtend has no facility for 3D graphs, but it maintains the accessibility of plotting elements, the sharing of plotting elements, and the free-structure of statistical images that were introduced in Simplisp. Xtend focuses on allowing the user to define new plotting elements to expand the system (extensibility) and on providing a more sophisticated method for defining new graph structures.

3.1.1 Extensibility

Extensibility—the ability to extend a system to perform new analyses—is an important feature of modern statistical software. An extensible system provides researchers with a basis for experimentation with new statistical analyses and graphics and means that such experimentation is immediately available to practising statisticians (Tierney, 1996). Both the $S$ system (Chambers, Becker, and Wilks, 1988) and the XLispStat system (Tierney, 1990) provide some degree of extensibility, although all $S$ graphics are static and can only be constructed from relatively high-level commands and XLispStat has a fairly rigid graphics structure which only allows certain graphical operations to be modified easily (ultimately anything could be achieved because XLispStat includes a complete Lisp implementation, but the effort and expertise required for major modifications is substantial).

The Simplisp statistical graphics system, described in the previous chapter, adopted a kitset approach to statistical plotting elements. That is, in Simplisp, a number of basic components are provided from which plotting elements are constructed; for example, a tick-mark is a kitset made up of a label and a line, an axis is made up of a number of tick-marks, a line, and a label, and a scatterplot is made up of two axes, and a data series. The purpose of this approach in Simplisp was to provide a structure for plotting elements that allowed the user to access and modify plotting elements at many levels. Unfortunately, the creation of new plotting elements (e.g., different kinds of plots such as histograms and boxplots) was complicated and required a great deal of effort to ensure that modifications to a high-level description had the desired effects on the components of a plotting element (e.g., changing the location of an axis required changing the locations of all of the tick-marks, the line, and the label that make up the axis).

One aim for the Xtend system was to provide a simpler mechanism for creating new plotting elements. This would allow the user to expand the system to include new plotting elements or even to create new implementations of existing plotting elements.

3.1.2 Incremental Graphics

The primary usefulness of statistical graphics software is the graphical dirty work that it performs. For example, the $S$ system (Chambers, Becker, and Wilks, 1988) creates a complete scatterplot full of appropriately positioned lines, points and text from the simple command `plot(x, y).

Many modern statistical software packages also provide the flexibility to construct a statistical image in a piecemeal fashion. For example, the $S$ system creates a complete plot if given the command `plot(x, y), but the same image can instead be constructed in several separate steps as follows:

\[
\text{par(usr=c(range(x)*1.05, range(y)*1.05))}
\]
CHAPTER 3. XTEND

points(x, y)
axis(1)
axis(2)
box()

There is little to be gained from creating the above plot in this incremental fashion, but the finer control it provides is very useful for creating images which deviate from the default layout or structure (e.g., adding a secondary y-axis to a plot).

Many packages also allow an image to be annotated with further lines, and text. For example, in the S system, the command text(2, 2, "hi there") places the specified text at the location (2, 2) in the current plot.

In these ways, the construction of a statistical image is similar to the construction of an image in a standard graphics editor; the difference is that, whereas both types of image can be constructed from simple graphical elements like lines, points, and text, statistical images can also be constructed from very complex graphical elements like axes and plots.

The Similisp package allowed novel statistical images to be created by assembling plotting elements in novel ways. However, Similisp took a very naive approach to combining elements, forcing the user to perform all of the work in order to make the different elements in an image consistent (e.g., the user had to specify that an axis and a data series in a scatterplot used the same scale).

Another aim of Xtend is to provide much more sophisticated support for combining plotting elements within an image in order to facilitate the incremental construction of statistical graphs.

3.1.3 A Stand-Alone Graphics System

In most cases, a statistical graphics system is included within a larger software package for statistical analysis. The advantages of this approach are obvious (consistency of interface, total sharing of information, and so on). However, there are also good reasons for producing a stand-alone statistical graphics package. The advantage for a developer is that it is possible to focus entirely on the graphics features and not be distracted by analysis issues. The advantage for the user is that the developer has been able to focus on the graphics so that the best possible graphics system is the result.

The major problem to be solved in order to create a useful stand-alone graphics package is: how does information get from the statistical software to the graphics software and back again?

A simple solution is to write data to disk in a format that both applications can read (e.g., ASCII text). The XGobi system can be run from S-Plus in this manner, with both programs communicating by writing data to disk (Swayne, Cook, and Buja, 1991). However, there are difficulties with this approach. In particular, it is not easy to synchronise the activity of the two systems. This is a problem if, for example, the statistical system attempts to modify a data file at the same time that the graphics system is using the data file to draw a graph. More sophisticated solutions exist (e.g., XGobi can also communicate with a number of other systems using UNIX inter-process communication techniques and remote procedure call technology), however, these tend to be hardwired solutions to provide for specific functionality. In an extensible system, it is desirable for the user to be able to communicate arbitrary commands to a statistical package and to receive arbitrary results.
The third aim of the Xtend package was to investigate an alternative method for communicating between a stand-alone graphics system and a statistical analysis package which specifically allowed for arbitrary commands and results to be exchanged.

SUMMARY Xtend aims to build on the useful features of Simplisp and overcome some of its deficiencies. In particular, Xtend will provide an environment which supports incremental construction of graphs and the development of novel graphical structures. Xtend will be a stand-alone graphics package, which means that it will depend upon communication with other software packages in order to perform statistical analyses on data.
3.2 Design

The Xtend system includes many of the features of the Simplisp system, including automatic updating of outputs and selecting the elements in a graph with the mouse. In addition, Xtend provides the features outlined below.

3.2.1 Constructing a graph

The Xtend system distinguishes between two sorts of components for constructing statistical images: simple graphical primitives (such as lines and text) and more complex plotting elements (such as tick-marks, symbols, axes, series, and plots). Xtend provides a number of predefined primitives and plotting elements; the user can also develop new plotting elements.

All of these graphical objects exist in a 2D system called World Coordinates (WC). The locations and dimensions of some plotting elements rely on data values; scales provide the mapping between data values, which specify locations and dimensions in Data Coordinates (DC) and the corresponding locations and dimensions in WC (e.g., a scale is necessary for converting a pair of data values into the location of a data-symbol plotting element in WC).

A graph is constructed by combining a number of primitives and/or plotting elements together into an image. The image defines a rectangular region of WC (a window) which will be visible. This defaults to a rectangle that is slightly larger than the unit square (i.e., the square with its bottom-left corner at (0,0) and its top-right corner at (1,1)). This means that graphical objects which are located in the unit square will be visible by default.

The graph is viewed by adding the image to an output device (e.g., a window on screen or a postscript file). The image defines a rectangular region of the output (a viewport) within which the visible region of WC will be drawn (i.e., an image provides a mapping between a rectangular region in WC and a rectangular region on the output; see Figure 3.5).

![Figure 3.5: The components of the viewing system in Xtend.](image-url)
3.2.2 Developing new plotting elements

Xtend differs from Simplisp in that it explicitly supports the development of new plotting elements. In particular, Xtend makes public the method used for developing the predefined Xtend plotting elements so that the same method can be used to develop new plotting elements.

The description of a plotting element in Xtend consists of three parts: a high-level description of the element, a list of raw materials for creating the element, and the rules for combining these raw materials (the relationship between the high-level description and the raw materials). For example, the high-level description of a tick-mark includes the size of the tick-mark and the direction that the tick-mark is pointing; the raw materials for a tick-mark are a line-primitive (for the tick) and a text-object (for the label); the rules for a tick-mark specify the layout of the line-primitive and the text object (e.g., the direction and length of the line and the position of the text relative to the line).

Xtend allows the user to specify a description for a new plotting element and, given such a description, the Xtend system can create examples of the plotting element and automatically maintain the arrangement of the raw materials based on the current high-level description of the plotting element (i.e., enforce the rules of the plotting element).

3.2.3 Combining plotting elements

Xtend allows for an arbitrary graph structure. Any plotting element or graphical primitive can be added directly to an image (e.g., a graph can be created by adding an x-axis, a y-axis, a data series, and another data series to an image). In addition, any plotting element or graphical primitive can be added to another plotting element. This means that, for example, a graph could be created by adding a scatterplot to an image, then another data series could be added to the scatterplot rather than directly to the image.

Xtend provides sophisticated support for arbitrary graph structures by providing a mechanism which allows plotting elements to share certain information automatically. When a graphical object is added to a plotting element, the plotting element may provide useful information for the graphical object so that it will be consistent with existing components of the plotting element. For example, when a series is added to a plot, the plot provides the series with an x-scale and a y-scale so that the symbols plotted for the new series are consistent with existing series and axes in the plot.

3.2.4 Constraints between plotting elements

The Xtend system allows the user to specify arbitrary constraints between plotting elements or within a single plotting element, which are then automatically maintained by the system. This means that, for example, the y-location of a line can be constrained to be the average value of a given variable so that if the variable values change, the y-location of the line will change.

The specification of constraints within a plotting element are important in Xtend for specifying the rules for combining the raw materials of a plotting element.
3.2.5 Communication with R

The Xtend system allows the user to send commands to the R statistical environment and receive the results back in an appropriate format. This means that, for example, in order to add a regression line to a scatterplot of two variables, a command can be sent to R to perform a regression analysis on the appropriate values and the result can be used to locate the regression line.

**SUMMARY**  
Xtend supports flexible plotting structures by allowing plotting elements to be combined in arbitrary ways and by providing a mechanism for sharing information between plotting elements. Xtend supports extensibility by providing a set of guidelines for the development of new plotting elements and by providing a general constraint mechanism. Xtend provides statistical-analysis functions by allowing communication with the R statistical environment.
3.3 Implementation

The Xtend system was implemented in Common Lisp (see Section 2.3) and was based on the KR (Knowledge Representation) module of the Garnet system (Myers et al., 1990).

The KR module was used because it provides a prototype-instance object system and a constraint system. The prototype-instance object system differs from the object system provided by CLOS, which is a class-instance system. In a class-instance system, classes provide templates from which instances are created and only the instances are objects. In a prototype-instance system, the prototype is also an object and may, but does not have to, be used as a template for creating instances. The prototype-instance system is therefore somewhat more flexible and arguably more convenient for developing software (Tierney, 1990).

A constraint is a relationship between two objects consisting of two parts: the declaration of the constraint and the maintenance of the constraint (Hoole and Blake 1996). Constraints can be implemented in any standard programming environment, but in a constraint-based environment, the specification of constraints is more explicit and the maintenance of the constraints occurs automatically, thereby removing this burden from the programmer (Borning and Duisberg, 1986).

A KR object consists of a set of slots, which contain information about the object. For example,

```
(create-instance 'prototype-object ()
  (:slot-1 1)
  (:slot-2 "slot-2")
```

creates an object called prototype-object with two slots; the slot :slot-1 contains the numeric value 1, and the slot :slot-2 contains the string value "slot-2".

The value of the slot is accessed using the g-value function and set using the s-value function. For example,

```
(s-value prototype-object :slot-1 "slot-1")
(g-value prototype-object :slot-1)
```

There is no need to specify all of the slots in an object when the object is first created. If a value is specified for a slot which does not yet exist then the slot is created. For example,

```
(s-value prototype-object :unknown-slot 5)
(g-value prototype-object :unknown-slot)
```

Of course, an attempt to access a slot which does not exist will not return a value:

```
(g-value prototype-object :another-unknown-slot)
```

A KR object (an instance) can be derived from another KR object the prototype, which means that the instance will inherit slots from the prototype if it does not provide a slot itself. For example,

```
(create-instance 'instance-object (prototype-object)
  (:slot-2 2)
  (:slot-3 3))
```
creates an object called instance-object, which is an instance of the object called prototype-object (prototype-object is the prototype for instance-object). The new object inherits the slot :slot-1 from its prototype and defines its own slots: :slot-2 with the numeric value 2 (it does not inherit this slot from its prototype) and :slot-3 with the numeric value 3.

Methods can be defined for a KR object and these are also inherited by instances of an object. For example,

```
(define-method :show-slot-2 prototype-object (object)
  (print (g-value object :slot-2)))
```

defines a method called :show-slot-2 for the object prototype-object which prints out the value of :slot-2. This method is inherited by the object instance-object because it is an instance of prototype-object.

A method is run by sending the appropriate message to an object. For example,

```
(kr-send prototype-object :show-slot-2 prototype-object)
>> "slot-2"
```

```
(kr-send instance-object :show-slot-2 instance-object)
>> 2
```

KR does not have generic functions; methods are implemented as slots in objects and if a message is sent to an object which has no appropriate slot then the object does nothing.

If an object specifies a method called :initialize, that function will automatically be run when instances of the object are first created. For example,

```
(create-instance 'some-object nil
  (:slot-1 1)
  (:slot-2 2))
(define-method :initialize some-object (the-object)
  (s-value the-object :slot-3
    (+ (g-value the-object :slot-1)
      (g-value the-object :slot-2))))
(create-instance 'instance-of-some-object some-object)
(g-value instance-of-some-object :slot-3)
>> 3
```

A constraint is created by defining a formula in a slot of an object. For example,

```
(create-instance 'another-object ()
  (:constrained-slot
    (c-formula (* (gv instance-object :slot-3) 2))))
```

defines a new object called another-object with one slot called :constrained-slot which contains a formula that gets the value of :slot-3 of instance-object and multiplies it by 2.

The value of :constrained-slot is calculated from the formula. If the value of :slot-3 in instance-object changes, KR automatically updates the value of :constrained-slot. For example,

2Strictly speaking, when the value of :slot-3 changes, KR sets a flag in all of the slots that depend on :slot-3, which in this case is just :constrained-slot. The value of :constrained-slot is only updated the next time that :constrained-slot is accessed. This approach is called lazy evaluation (as opposed to eager evaluation)
(g-value another-object :constrained-slot)  
>> 6

(s-value instance-object :slot-3 5)  
(g-value another-object :constrained-slot)  
>> 10

The use of the gv function is important. This function returns the value of a slot just like g-value, but it also sets up the constraint information required to determine when :slot-3 changes. If the g-value function was used in the constraint, this information would not be set up and the constraint would not be maintained. A related function gv1 works like gv except that it accesses a slot in the object within the object which owns the constrained slot. For example,

(s-value another-object :another-slot  
(o-formula (* 2 (gv1 :constrained-slot))))  
(g-value another-object :another-slot)  
>> 20

Another way of automatically performing actions when a slot is modified, is to define a KR demon. The standard demon is called the *pre-set-demon*. This is a function that will be called whenever certain slots in an object are modified. The slots which the demon works on are specified in a special :update-slots slot in the object. For example, the following code defines a function for the demon which will print out the old value and the new value of the slot which is being modified.

(defun my-demon (object slot new-value)  
(print (g-value object slot))  
(print new-value))  
(setf *pre-set-demon* #'my-demon)

Now, if any object has one of its :update-slots modified, the demon will be run. For example,

(create-instance 'any-old-object nil  
(:slot-1 1)  
(:slot-2 "hi")  
(:update-slots '(:slot-1)))  
(s-value any-old-object :slot-2 "bye")  
(s-value any-old-object :slot-1 2)  
>> 1  
>> 2

(Note that modifying :slot-2 of the object did not cause the demon to be run).

3.3.1 Xtend Prototypes

This section describes the main KR prototypes in the Xtend system.

Fundamental Prototypes

There are two fundamental prototypes within the Xtend system: the proto-drawable and the proto-updatable.
A proto-drawable object has a slot containing a list of drawable children. The :draw method for proto-drawable objects sends the :draw message on to each of the drawable children. This means that when an instance of proto-drawable is drawn, each of the drawable children of that instance will also be drawn. Similarly, the :select and :unselect methods for proto-drawable objects simply pass these messages on to the drawable children of the proto-drawable. This means that when a proto-drawable is (un)selected, all of its children will also be (un)selected.

A proto-updatable object has a slot containing a list of updatable children and a slot containing a list of outputs. When an output is added to an proto-updatable (usually when the proto-updatable object is added to the output in order to be viewed), the output is appended to the proto-updatable's list of outputs and the output is added to each of the proto-updatable's updatable children. This is to ensure that all of the updatable children of a proto-updatable object (i.e., all of the children which have a list of outputs) know which outputs they appear in; this is useful for being able to redraw the outputs when the proto-updatable is modified. For example, suppose that an image, I, consists of a rectangle, R (both I and R are proto-updatable objects—they know which outputs they are in—and R is an updatable child of I). When I is added to an output, O, the output must be added to the output lists for both I and R (both I and R “appear in” O and if either I or R is modified, the contents of O should be redrawn).

When a new updatable child is added to a proto-updatable, the proto-updatable's current list of outputs is added to the new child (and when an updatable child is removed from a proto-updatable, the proto-updatable's current list of outputs is removed from the child). For example, suppose that another image, I₂, has been added to another output, O₂ (so the current list of outputs for I₂ is just O₂). If the rectangle R is added to I₂, the output O₂ should be added to the list of outputs for R (R now “appears in” both O and O₂ and if R is modified, the contents of both O and O₂ should be modified).

In Xtend, all slots are modified using the same function—edit-slot (see Section 3.3.3) —rather than having a special edit function for every slot of every object so it is not always possible to detect when an updatable child is being added to or removed from a proto-updatable (i.e., when object A has slot c which contains an updatable child and the user modifies the value of slot c so that it now contains a new updatable child). To account for this sort of occurrence, the KR *pre-set-demon* is defined to perform the correct adding and removing of updatable children. For this to work, every instance of proto-updatable must specify all of its updatable children as :update-slots (so that the demon will be run); this will be described for each case below. For example, the rectangle R has a :colour slot, which contains a colour, C. C is a proto-updatable (it knows which outputs it appears in) and an updatable child of R. From the examples above, the output list for C contains both O and O₂ (when these outputs were added to R, they were automatically passed on to C as well). Suppose that the :colour slot of R is modified so that it now contains the colour C₂. The outputs O and O₂ must be added to the output list of C₂ (C₂ now “appears in” both of these outputs) and these outputs must be removed from the output list of C (C no longer “appears in” either output). These actions are carried out by the KR *pre-set-demon* because the :colour slot is one of the :update-slots of R.

The proto-updatable prototype is the basis of all objects which influence the appearance of the contents of an output (images, graphic objects, scales, fonts, and colours). The list of outputs in a proto-updatable represents the union of all
outputs that the proto-updatable can currently influence. When a proto-updatable is modified, it is able to direct these outputs to redraw their contents; this is how automatic updating of outputs occurs.

**Outputs**

The proto-output prototype is an instance of proto-drawable. A proto-output has a list of images (which are collections of things to draw; see below). The drawable children of a proto-output are the images in the proto-output’s list of images (a separate list of images is kept so that the proto-output can sever links with its images if it is destroyed). When an image is added to a proto-output (so that the objects in the image can be viewed), the image is added to both the list of images and the list of drawable children and the proto-output is added to the image (so that the image can tell the proto-output when to redraw).

The proto-input/output prototype is an instance of proto-output. A proto-input/output has a slot containing a list of shadows (which represent where objects in the proto-input/output’s images were drawn so that the objects can be picked with a mouse; see below). The :draw method for proto-input/outputs (which overrides the method that it would otherwise inherit through proto-output from proto-drawable) constructs its list of shadows if required (i.e., if there has been a change in any of the images since the last draw), otherwise it just passes the message to each of its drawable children (just like any normal drawable).

The X11 object is an instance of proto-input/output. This represents an X11 window on screen. There are a number of methods defined for X11 objects to actually produce lines and text on the screen (e.g., :move-to, :line-to). Objects which need to produce output call these methods.

**Images**

The proto-image prototype is an instance of proto-drawable and proto-updatable. An image has a viewport, which specifies a rectangle on the output(s) that the image appears in, a window, which specifies a rectangle in WC, and a list of graphical objects to view. The graphical objects are all drawable children and updatable children of the proto-image. This means that, when a proto-image is drawn, it draws all of the graphical objects, and a proto-image is responsible for maintaining the output lists for all of its graphical objects (i.e., every graphical object in a proto-image knows which outputs the proto-image has been added to). Graphical objects are added to and removed from a proto-image via special functions so the proto-image has no :update-slots (i.e., the proto-image does not require the *pre-set-demon* to help maintain the outputs for its updatable children).

**Scales, Fonts, and Colours**

The proto-scale, proto-font, and proto-colour are all instances of proto-updatable. None of these objects has any updatable children (so they have no :update-slots), but they all maintain a list of outputs. Scales represent a mapping between DC and WC, fonts represent a style for drawing text, and colours represent RGB codings of colours. All of these objects serve as sources of information for the graphical objects that make up an image (see below). They are reliant on the objects that use them to maintain their lists of outputs. As updatables, whenever one of these objects is modified, it will instruct each of its outputs to redraw.
CHAPTER 3. XTEND

Graphic objects

The graphic-object prototype is an instance of proto-updatable. All graphic-objects have a list of images (so that they can sever links with their images if they are destroyed), a colour, a font, an x-scale, a y-scale, and a line-thickness. A graphic-object defines the colour, font, x-scale, and y-scale slots to be :update-slots. This means that, if any one of these slots is modified (e.g., the colour of the graphic object is changed), the KR demon will be run in order to maintain the output lists of the old value and the new value (e.g., the outputs of the graphic-object will be removed from the old colour and added to the new colour; this ensures that, if the old colour is modified, it will not unnecessarily update the outputs that the graphic-object appears in, and if the new colour is modified the outputs will be updated).

The primitive prototype is an instance of graphic-object. This object adds slots for x, y, width, and height values. The graphic primitive objects—line, rectangle, arc, and char—are all instances of primitive. Each graphic primitive defines a :draw method which makes calls to the appropriate output to draw lines and text (recall that the drawing message gets passed down by drawable objects; this is where the message ends up and drawing actually occurs). Each graphic primitive also defines a :bound method for calculating a bounding rectangle for itself. These methods are called in the production of shadows to record where the graphic primitives were drawn on an output.

Groups

By far the most important prototype in the Xtend system is the proto-group. This object forms the basis of all plotting elements.

Most of the prototypes in the Xtend system are only for internal use. Specifically, most prototypes are designed so that the user can create instances of them and can modify the values in their existing slots, but the user is not expected to modify their structure (i.e., add new slots or remove existing ones; the user is not prevented from doing this, but the consequences are highly unpredictable). In contrast, the group prototype is designed so that the user can create instances and modify not only the values of existing slots, but also the overall structure of the object.

The proto-group object is an instance of the graphic-object and the proto-drawable objects. This means that: a proto-group has a list of drawable children (so that when a proto-group object is drawn, all of its children will be drawn); a proto-group has a list of outputs (so that, when a proto-group object is modified, it will ask all of these outputs to redraw); a proto-group has a list of updatable-children (so that the proto-group will maintain the lists of outputs for those of its children which are instances of proto-updatable).

A group object has a slot containing a list of inherited slots (this list specifies slots in the group which can inherit values from the group’s parent; see 3.3.2), a slot containing a list of private slots (this list specifies which slots cannot be inherited by components of the group), and a slot containing a list of data slots (this list specifies which slots will contain variables).

The :x-scale, :y-scale, :colour, :font and data slots are all :update-slots for a group object.

A group can have any number of slots containing components or lists of items (components of the same type stored in a list). By default, a group has no components or lists of items, but they can be added and removed at any time.

Component slots are added to a group using the add-component function (and
removed using the `remove-component` function). This function creates a slot in the
group to contain the component and a `:parent` slot in the component (to provide
a link from the component to the group), and installs constraints in the inherited
slots of the component to enforce inheritance (see 3.3.2).

Slots containing lists of items can be added to a group using the `add-list`
function. This function creates a slot in the group to contain the list of items, a slot
to contain the number of items in the list, and a slot to contain the prototype object
for the items in the list (all of the items in the list are instances of this prototype).
Also, a constraint is installed to ensure that if the item prototype or the number
of items changes, the list of items will be updated. As each item is added to the
list of items, a `:parent` slot is created in the item (to provide a link from the item
to the group), a `:rank` slot is created in the item (to indicate the location of the
item within the list) and constraints are installed in the inherited slots of the item
to enforce inheritance.

### 3.3.2 Inheritance

This section applies only to graphic-objects and groups.

If a graphic-object, \( C \), is a component of a group, \( P \), and \( C \) occupies slot \( :c \) in
\( P \), and \( C \) has an inheritable slot, \( :i \):

1. slot \( :i \) will use the value in slot \( :c-i \) of \( P \) if it exists, otherwise
2. slot \( :i \) will use the value in slot \( :i \) of \( P \) if it exists and it is not a private slot,
   otherwise
3. slot \( :i \) will use its own value.

If \( C \) is a list-item of \( P \), and the list occupies slot \( :cs \) in \( P \) and \( C \) is the \( n \)th item in
the list and \( C \) has an inheritable slot, \( :i \):

1. slot \( :i \) will use the \( n \)th value in slot \( :cs-i-list \) of \( P \) if it exists, otherwise
2. slot \( :i \) will use the value in slot \( :cs-i \) of \( P \) if it exists, otherwise
3. slot \( :i \) will use the \( n \)th value in slot \( :i-list \) of \( P \) if it exists and it is not a private slot, otherwise
4. slot \( :i \) will use the value in slot \( :i \) of \( P \) if it exists and it is not a private slot,
   otherwise
5. slot \( :i \) will use its own value.

The inheritance of slots provides a mechanism by which it is possible for plotting
elements to be added to each other in an "intelligent" way (e.g., when a data series
is added to a plot, it inherits the x-scale and y-scale from the plot so that it is
automatically consistent with the existing elements of the plot).

### 3.3.3 Edit-slot

The `edit-slot` macro is essential to the automatic updating of outputs in Xtend.
This provides a wrapper for the standard KR `s-value` function and ensures that,
when the value of a slot is modified, the object whose slot it was is instructed to
update its outputs (which forces any redrawing that may be necessary). If a slot

---

**CHAPTER 3. XTEND**

74
is modified just using s-value the updating of the appropriate outputs will be
unpredictable.
Any changes to the plotting elements themselves (e.g., if the :length slot in a
tick-mark object is modified then the :height and :width slots of the line-primitive
component of the tick-mark have to be modified as well) are automatically per-
formed by the constraint system (which ensures that what actually gets drawn is
up to date).
This arrangement means that there is a single function for editing any slot in any
Xtend object, rather than an enormous number of edit functions for each different
object.

3.3.4 Rslave

The Xtend system uses R as a statistical analysis engine (as opposed to being a
graphics engine for the R system). Xtend allows the user to spawn a slave R process
and communicates with this process via UNIX pipes (it is not possible to run Xtend
from the R system).

The basis for the creation of an R process and communication between this
process and Xtend are three C functions. The c_start function creates two pipes
(a pipe is a character buffer with two file descriptors—one for reading the buffer
and one for writing the buffer, Wang, 1993) and forks the child R process (the fork
system call creates a copy of a process, including all open file descriptors; after the
copy, there are two processes, both of which have file descriptors for reading from
and writing to the two pipes). The parent process (Xtend, or more accurately the
Lisp process into which Xtend has been loaded) closes its file descriptor for writing
into pipe1 and its file descriptor for reading from pipe2. The child R process does
the reverse so that the R writes via pipe1 and reads via pipe2 and Xtend reads
via pipe1 and writes via pipe2. Also, R is fooled into thinking that its stdin and
stdout are pipe2 and pipe1 respectively. this means that the child R acts pretty
much like any normal R process (reading commands from stdin and writing the
results to stdout), but the input can only come from Xtend and the output can
only go to Xtend.

The c_send function takes a string message and writes it into pipe2. Xtend uses
this to send commands to R. The c_receive function reads pipe1 and returns the
characters it reads as a string. Xtend uses this command to receive results from R.
As part of its normal activity, R reads pipe2 (which it thinks is stdin) and writes
to pipe1 (which it thinks is stdout).

There are also a number of Lisp functions which prepare messages for sending
them to R and evaluate the results which are returned from R. When the R process
is first started (using the startR function), several R commands are automatically
sent to set up R functions that will do the formatting of results to be sent back
to Xtend. The results can be returned verbatim (i.e., in the standard R output
format) or as a Lisp list.

The format function is defined to print an expression which will evaluate as a
Lisp list (recall that printing, or writing to stdout, for R means writing to pipe1).
This function also precedes the result with a control-B character to signal the start
of the formatted result. The reply function is defined to print a value (if the results
are verbatim) or call format. The reply.parse function is defined to parse a value
and, only if the resulting expression is valid, evaluate the expression and print the
result (if the results are verbatim) or call format on the result. Both reply functions
append a control-G character to the result to signal the end of the result.
A message can be sent to \textit{R} as a valid \textit{R} command or as a command which should be parsed. The \texttt{Rquest} function takes a string message and wraps it in one of the predefined \textit{R} functions (so that it will not only be evaluated by \textit{R}, but the result will be returned with the appropriate formatting). If the message is a valid \textit{R} expression it is wrapped in the \texttt{Reply} function, otherwise it is wrapped in \texttt{Reply.parse}. If necessary, \texttt{Rquest} splits the \textit{R} command into several strings before sending it (to avoid exceeding the length of the buffer in pipe2 and to avoid exceeding limits in \textit{R} on the length of a single command-line).

Having sent the message to \textit{R}, \texttt{Rquest} reads pipel for the result (it will automatically wait until something is written into pipe1). Anything prior to a control-B character is just written to stdout (this takes care of error messages and other unwanted by-products of \textit{R} output). Whatever is read between the control-B character and a control-G character is evaluated as a Lisp expression (which will result in a Lisp list) and returned.

\textbf{SUMMARY} \textit{Xtend} is implemented in Common Lisp and is based on the \texttt{KR} module of Garnet. Automatic updating of outputs is supported by the proto-updatable object. All plotting elements are derived from the group object. Sharing information between plotting elements is supported by an inheritance mechanism. Access to all components of a plotting element is provided by the syntax of the edit-slot command. Communication with \textit{R} is via UNIX pipes.
CHAPTER 3. XTEND

3.4 Xtend Reference

3.4.1 Functions

--- useful KR functions ---

g-value(object list-of-slot-specifiers)

create-instance(symbol prototype)
creates an instance of prototype and creates new symbol to store it in

--- utility functions ---

cs(list-of-strings)

edit-slot(object list-of-slot-specifiers value)

get-data-dimension(variable rank scale)
returns the rank'th value in the variable, converted to WC (as a dimension)

get-data-location(variable rank scale)
returns the rank'th value in the variable, converted to WC (as a location)

radians(degrees)

show(object)
adds the object to the *default-image*, which has automatically been added to the *default-window*

text(whatever)
converts the argument into a string

--- functions for creating new groups ---

add-component(component slot-name parent)

add-list(item-prototype slot-name parent)

define-group(group-name
    &key inherited-slots private-slots
data-slots default-components
default-lists)

remove-component(component)
remove-list(slot-name parent)

conversion functions

convert(value scale
   &key (to :owc) (locn-p T))

convert-list(value-list scale
   &key (to :owc) (locn-p T))

communicating with R

R(message-string
   &key (evaluate T) verbatim (parse T))

Rvector(variable)

3.4.2 Prototypes

This section provides a detailed reference to all of the prototypes in Xtend that the user will encounter. In particular it describes all of the slots in the prototypes that the user is able to modify.

The description of each prototype consists of a listing of the prototype's slots. Slots contain numeric values in WC unless otherwise specified. A justification is one of _left_, _centre_, or _right_ if it is a horizontal justification; _bottom_, _centre_, or _top_ if it is a vertical justification.
viewing objects

The slot descriptions in this section are of the form:

:slot-name () description

proto-window

proto-image

:viewport () a vector (left, bottom, right, top); the location of the image on its output(s)

:window () a vector (left, bottom, right, top); the region of WC visible in the image

resource objects

The slot descriptions in this section are of the form:

:slot-name () description

proto-scale

:min () the minimum value in DC

:max () the maximum value in DC

:ow-min () the minimum value in WC

:ow-max () the maximum value in WC

proto-font

:filename () the name of a font description file (e.g., "../../../simpflex/Fonts/font.str")

proto-colour

:red ()

:green ()

:blue ()
CHAPTER 3. XTEND

The slot descriptions in this section are of the form:

:slot-name (list of characteristics) description

The characteristics indicate whether the slot is inherited (see Section 3.3.2; defaulted indicates a special sort of inheritance where the value of the slot may be defaulted if its inherited value is nil), whether it is private (see Section 3.3.2), and whether it is a data slot.

To determine the inheritance of values between any two prototypes (say A and B), first find the inherited slots of A; these represent the slots which A will try to find values for in B. Next, for each inherited slot in A, use the rules of inheritance in Section 3.3.2 to find a matching slot in B, which is not a private slot.

For example, the text plotting element has the following inherited slots: :string, :location, :height, :angle, :h-just, and :v-just. A text plotting element is a component of a tick-mark, occupying slot :label. The tick-mark plotting element has neither a :label-string slot nor a :string slot so the text does not inherit a value for its :string slot from the tick-mark. The tick-mark does, however, have a :label-location slot so the text inherits a value for its :location slot from the tick-mark (by inheritance rule 1. for components). The tick-mark has
a :label-height slot so the text inherits a value for its :height slot from the
tick-mark as well (by inheritance rule 1. for components). Figure 3.6 shows the
complete inheritance of values between a tick-mark plotting element and its text
plotting element (the inheritance between a tick-mark and its line primitive is also
shown).

All graphic primitives and plotting elements have the following slots:

:x-scale (defaulted) a proto-scale object or nil
:y-scale (defaulted) a proto-scale object or nil

In addition, all plotting elements have the following slots:

:colour (defaulted) a colour object or nil
:font (defaulted) a font object or nil
### line
- :x (inherited)
- :y (inherited)
- :width (inherited) the horizontal distance to the other end-point of the line
- :height (inherited) the vertical distance to the other end-point of the line
- :colour (defaulted) a colour object or nil
- :line-thickness (inherited)

### rectangle
- :x (inherited)
- :y (inherited)
- :width (inherited)
- :height (inherited)
- :colour (defaulted) a colour object or nil
- :line-thickness (inherited)
- :filled () a logical flag

### arc
The :x, :y, :width, and :height slots describe a rectangle which bounds the arc.
- :x (inherited)
- :y (inherited)
- :width (inherited)
- :height (inherited)
- :colour (defaulted) a colour object or nil
- :line-thickness (inherited)
- :filled () a logical flag

### char
- :x (inherited)
- :y (inherited)
- :width () defaults to the :height of the char; modifying this slot directly will distort the character
- :height (inherited)
- :line-thickness (inherited)
- :h-just () a horizontal justification
- :v-just (inherited) a vertical justification
- :angle (inherited) angle of rotation, anti-clockwise from the positive x-axis, in degrees
- :font (defaulted) a font object or nil
- :char (inherited) a character
text

lists
:chars () a list of char primitives to represent the characters in the text

slots for list items
:num-chars () the number of characters in the text string
:chars-char-list () a character for each char primitive
:chars-x-list () an x-location for each char primitive
:chars-y-list () a y-location for each char primitive

high-level description
:string (inherited, private) a string
:location (inherited, private) a vector (x, y)
:height (inherited)
:width (private) defaults from the text height, depending on the length of the text string; modifying this slot directly will distort the text
:angle (inherited)
:h-just (inherited, private) a horizontal justification
:v-just (inherited, private) a vertical justification

square-symbol

components
:rect () a rectangle primitive to represent the symbol

slots for components
:rect-x () the rectangle primitive is centred on the location specified by the rank'th value in the symbol's x- and y-vbles (converted into WC)
:rect-y () see :rect-x

high-level description
:rank (private) the order statistic of the symbol
:width (inherited)
;height (inherited)
:x-vble (inherited, private, data)
:y-vble (inherited, private, data)
round-symbol

components
:arc () an arc primitive to represent the symbol

slots for components
:arc-x () the arc primitive is centred on the location specified by the rank'th value in the symbol's x- and y-vbles (converted into WC)
:arc-y () see :rect-x

high-level description
:rank (private) the order statistic of the symbol
:width (inherited)
:height (inherited)
:x-vble (inherited, private, data)
:y-vble (inherited, private, data)

plus-symbol

components
:line-up () a line primitive to represent the horizontal line in the plus
:line-across () a line primitive to represent the vertical line in the plus

slots for components
:line-up-x () the vertical line primitive is the height of the symbol and is centred on the location specified by the rank'th value in the symbol's x- and y-vbles (converted into WC)
:line-up-y () see :line-up-x
:line-up-width () see :line-up-x
:line-up-height () see :line-up-x
:line-across-x () the horizontal line primitive is the width of the symbol and is centred on location specified by the rank'th value in the symbol's x- and y-vbles (converted into WC)
:line-across-y () see :line-across-x
:line-across-width () see :line-across-x
:line-across-height () see :line-across-x

high-level description
:rank (private) the order statistic of the symbol
:width (inherited, private)
:height (inherited, private)
:x-vble (inherited, private, data)
:y-vble (inherited, private, data)
symbol-series

lists
: symbols () a list of symbol objects

slots for list items
: num-symbols () the minimum of the number of values in the x- and y-vbles
: symbols-width () a width for all of the symbols
: symbols-height () a width for all of the symbols

high-level description
: x-vble (inherited, data)
: y-vble (inherited, data)

tick-mark

components
: line () a line primitive representing the tick line
: label () a text object representing the tick label

slots for components
: line-x () anchors one end of the tick line to the tick origin
: line-y () see : line-x
: line-width () locates the other end of the tick line the line-length away from the tick origin (making the tick-angle with the positive x-axis)
: line-height () see : line-width
: label-location () locates the tick label the correct distance off the end of the tick line
: label-angle () orientates the tick label orthogonal to the tick line
: label-h-just () the tick label is centred on the label-location
: label-v-just () see : label-h-just

high-level description
: origin (inherited, private)
: angle (inherited, private)
: line-length (inherited, private)
: label-distance (inherited, private) the distance from the end of the tick line to the location of the tick label
: label-height (inherited)
Note that the :left, :right, :y, :first-tick, and :last-tick slots are all in WC components and lists:
- :line () a line primitive representing the axis line
- :label () a text plotting element representing the axis label
- :ticks () a list of tick-mark plotting elements representing the axis tick-marks

Slots for components and list items:
- :line-x ()
- :line-y ()
- :line-width ()
- :line-height ()
- :label-h-just () the label is centred on the label location
- :label-v-just ()
- :label-height ()
- :label-location () the label is located below the axis
- :num-ticks () a “pretty” number of tick-marks are generated by default, based on the range of the x-scale
- :ticks-line-length ()
- :ticks-label-height ()
- :ticks-origin-list () a location for each tick-mark (“pretty” locations are chosen by default)
- :ticks-label-string-list () a string for each tick-mark label; based on the location of the tick-marks and the x-scale

High-level description:
- :left (inherited, private) the x-location of the left end of the axis line, in WC (distinct from the minimum value on the x-scale)
- :right (inherited, private) the x-location of the right end of the axis line, in WC (distinct from the maximum value on the x-scale)
- :y (inherited, private) the y-location of the axis
- :first-tick (inherited, private) the x-location of the first tick-mark
- :last-tick (inherited, private) the x-location of the last tick-mark
- :label-location-propn (inherited, private) the x-location of the axis label as a proportion of the width of the axis line
- :label-height-propn (inherited, private) the height of the axis label as a proportion of the width of the axis line
- :label-distance-mult (inherited, private) the distance of the label away from the axis, in terms of multiples of the tick line-length
- :ticks-line-length-propn (inherited, private) the length of the tick lines as a proportion of the width of the axis line
Note that the :bottom, :top, :x, :first-tick, and :last-tick slots are all in WC components and lists

:line () a line primitive representing the axis line
:label () a text plotting element representing the axis label
:ticks () a list of tick-mark plotting elements representing the axis tick-marks

slots for components and list items

:line-x ()
:line-y ()
:line-width ()
:line-height ()
:label-angle ()
:label-h-just () the label is centred on the label location
:label-v-just ()
:label-height ()
:label-location () the label is located to the left of the axis
:num-ticks () a "pretty" number of tick-marks are generated by default, based on the range of the x-scale
:ticks-angle ()
:ticks-line-length ()
:ticks-label-height ()
:ticks-origin-list () a location for each tick-mark ("pretty" locations are chosen by default)
:ticks-label-string-list () a string for each tick-mark label; based on the location of the tick-marks and the x-scale

high-level description

:bottom (inherited, private) the y-location of the bottom the axis line
:top (inherited, private) the y-location of the top of the axis line
:x (inherited, private) the x-location of the axis
:first-tick (inherited, private) the y-location of the first tick-mark
:last-tick (inherited, private) the y-location of the last tick-mark
:label-location-propn (inherited, private) the y-location of the axis label as a proportion of the width of the axis line
:label-height-propn (inherited, private) the height of the axis label as a proportion of the height of the axis line
:label-distance-mult (inherited, private) the distance of the label away from the axis, in terms of multiples of the :tick-line-length
:ticks-line-length-propn (inherited, private) the length of the tick lines as a proportion of the height of the axis line
CHAPTER 3. XTEND

**dot-plot**

components
  :x-axis () an x-axis plotting element
  :series () a symbol-series plotting element

slots for components
  :label-height ()
  :x-axis-y () the x-axis is placed along the bottom edge of the plot border
  :x-axis-left ()
  :x-axis-right ()
  :series-x-vble ()
  :series-y-vble ()
  :symbols-width ()
  :symbols-height ()

high-level description
  :border (inherited, private) a vector describing the rectangular region occupied by the plot
  :x-vble (inherited, private, data)
  :axis-label-height-propn (private) the height of the axis label as a proportion of the size of the plot
  :axis-tick-length-propn (private) the length of the axis tick-marks as a proportion of the size of the plot
  :series-symbol-size-propn (private) the size of the series symbols as a proportion of the size of the plot
scattered-plot
components
:x-axis () an x-axis plotting element
:y-axis () a y-axis plotting element
:series () a symbol-series plotting element

slots for components
:label-height ()
:x-axis-y () the x-axis is placed a distance equal to the bottom-margin below the bottom of the plot border
:x-axis-left ()
:x-axis-right ()
:y-axis-x () the y-axis is placed a distance equal to the left margin to the left of the left side of the plot border
:y-axis-bottom ()
:y-axis-top ()
:series-x-vble ()
:series-y-vble ()
:symbols-width ()
:symbols-height ()

high-level description
:border (inherited, private) a vector describing the rectangular region occupied by the plot
:x-vble (inherited, private, data)
y-vble (inherited, private, data)
:left-margin (inherited, private)
:bottom-margin (inherited, private)
:axis-label-height-propn (private) the height of the axis label as a proportion of the size of the plot
:axis-tick-length-propn (private) the length of the axis tick-marks as a proportion of the size of the plot
:series-symbol-size-propn (private) the size of the series symbols as a proportion of the size of the plot
3.5 User's Guide

3.5.1 Getting started
First of all, start cmucl by typing cmucl at the system prompt (all subsequent typing will occur at the Lisp prompt). Now, load Xtend by typing:

(load "init.lisp")

An x11 window will automatically appear.

3.5.2 Creating a plot
The first step is to generate the data to plot:

(create-instance 'x proto-variable (:data-list (list 1 3 5 6)))

Next, create the plot object and view it in the window:

(create-instance 'my-plot dot-plot (:x-vble x))
(show my-plot)

3.5.3 The edit-slot function
The edit-slot function provides a convenient way to access any component of a graph from the command-line. The function takes as arguments an object to edit, one or more slot specifiers and a new value for the specified slot. The value of the appropriate slot is modified, relevant changes are made to the slots of other objects which are constrained in some way to the modified slot, and any redrawing that is necessary is performed. For example,

(edit-slot my-plot :axis-label-height-propn .15)

modifies the height of the axis label (as a proportion of the size of the plot) and redraws the plot.

A more complicated example shows how to edit the text of the axis label:

(edit-slot my-plot :x-axis :label :string "score")

This command modifies the value of the :string slot in the text object which occupies the :label slot in the x-axis object which occupies the :x-axis slot of my-plot.

3.5.4 Controlling plot layout
The show function adds a plotting element to the *default-image* (which has been added to the *default-window*). This image occupies the entire window (i.e., its viewport is from (0,0) to (1,1)). The image can be positioned in a different part of the window by modifying the viewport of the *default-image*. For example, the following expression places the image in the top-right corner of the window:

(edit-slot *default-image* :viewport (vector .5 .5 1 1))
CHAPTER 3. XTEND

Another default setting of the *default-image* is the region of WC that is visible. This is called the image's window. This defaults to a rectangle that is slightly larger than the unit square. Most plotting elements default their location and size to be arranged on the unit square, which is why my-plot is visible without any work being done. A different region of WC can be made visible (e.g., to generate larger margins around a plot) by modifying the window of the *default-image*. For example,

```lisp
(edit-slot *default-image*
:window (vector -.5 -.5 1.5 1.5))
```

If several plots are to be displayed in a single window, it becomes necessary to create a new image. The following image occupies the bottom-left corner of the *default-window* and shows the original view of my-plot (i.e., the view that the *default-image* provided before its window was modified):

```lisp
(create-instance 'my-image proto-image
 (:viewport (vector 0 0 .5 .5)))
(add-image my-image *default-window*)
(add-object my-plot my-image)
```

### 3.5.5 Adding plotting elements to each other

A new data series can be added to the plot as follows:

```lisp
(create-instance 'series-2 symbol-series
 (:x-vble (create-instance nil proto-variable
 (:data-list (list 2 4 6 8))))
 (:y-vble (create-instance nil proto-variable
 (:data-list (list .5 .5 .5 .5)))))
(edit-slot series-2 :symbols-prototype round-symbol)
(add-component series-2 :series-2 my-plot)
```

The new series automatically inherits the :x-scale of my-plot so that it is consistent with the axes and the existing series. The new series also inherits the size of its symbols from my-plot so that modifications to the high-level description of my-plot affect both the existing series and the new one. For example,

```lisp
(edit-slot my-plot :series-symbol-size-propn .03)
```

affects the symbol-sizes for both series. In order to only modify the size of the symbols in the existing series, the user would just modify that series. For example,

```lisp
(edit-slot my-plot :series :symbols-height .02)
(edit-slot my-plot :series :symbols-width .02)
```

### 3.5.6 Communicating with R

The first step is to start the R system by typing (start). The R system runs invisibly. That is, the user does not see the normal R command prompt; all interaction with R is through the Xtend interface.

Having started R, a request is sent to R using the R function and supplying an R command as a text parameter. For example, the following command sends a command to R to calculate the mean of a vector of numbers (Note that the value is returned as a Lisp list):

```lisp
```
There are a number of optional keyword parameters to the R function. The :parse keyword indicates whether or not R should parse the command before executing it. This should only be set to NIL if the user is certain that the command is syntactically correct. The :evaluate parameter determines whether the value of the command will be returned from R. If the value of a command is large and of no interest, this should be set to nil to prevent unwanted output being displayed. This option is useful for sending commands which only have side-effects in R, such as generating a set of random normal values and storing them in an R variable (see the example below). The :verbatim parameter indicates whether the value returned by R is evaluated as a Lisp list or whether it is just displayed as standard R output. The user can set this to T in order to determine the nature of the value returned from a command. The following series of commands illustrates the use of these parameters.

(R "y \(-\) rnorm(10)" :evaluate NIL)
(R "lowess(y)" :verbatim T)
>> $x
>> [1] 1 2 3 4 5 6 7 8 9 10
>> $y
>> [1] -0.57918453 -0.51945490 -0.42919324 -0.20704203
>> [5] 0.05636042 0.26439522 0.48715271 0.64745579
>> [9] 0.76919582 0.88606347

(R "lowess(y)$y")
>> (-0.57918453 -0.51945490 -0.42919324 -0.20704203
>> 0.05636042 0.26439522 0.48715271 0.64745579
>> 0.76919582 0.88606347)

Utility functions are also provided for creating R vector expressions from Xtend variables and for combining strings together into a single message (cs stands for "concatenate strings"). For example,

(create-instance 'y proto-variable
 (:data-list '(1 2 3 4 5))
(Rvector y)
>> "c(1,2,3,4,5)"

(R (cs "mean" (Rvector y) "))
>> (3)

3.5.7 Constraints between plotting elements

In addition to the coordination of plotting elements through inheritance, it is possible to specify explicit constraints between plotting elements. In the example of Section 3.5.5, a line to indicate the mean value in the first series could be added to the dot-plot as follows:

(create-instance 'mean line
 (:x

(R "mean(c(1,2,3,4,5))")
>> (3)
(o-formula
  (convert (car (R (cs "mean(" (Rvector x) ")")))))
  (:width 0))
(add-component mean :mean my-plot)

By adding the mean line to my-plot, it automatically inherits the :x-scale from my-plot so that it is consistent with the other plotting elements. The constraint makes sure that the line will be repositioned if either the :x-scale or the variable x is modified.

My-plot does not provide any specific rules for the mean line by default, however, these can be added now and they will be automatically detected by the mean line. For example, in order to make the line only span the vertical dimensions of the plot, the following rules could be added to my-plot:

    (edit-slot my-plot :mean-y
      (o-formula (elt (gvl :border) 1)))
    (edit-slot my-plot :mean-height
      (o-formula (- (elt (gvl :border) 3)
                    (elt (gvl :border) 1))))

The constraints ensure that if, for example, the size or location of the plot is modified, not only the original components of the plot (the x-axis and the original series) but also the new series and the mean line will be relocated and resized appropriately.

3.5.8 Creating new plotting elements

A new plotting element, or group, is created in two steps. First of all, a definition of the group is created using the the define-group function. This function requires a name for the new group plus a number of optional arguments:

- a list of inherited slots. This list specifies which slots in the group can be inherited from the group’s parent (if it has one).
- a list of private slots. This list specifies which slots in the group cannot be inherited by the group’s children.
- a list of data slots. This list specifies which slots in the group will contain variable objects.
- a list of default components. This is a list of two-element lists. Each sub-list specifies a graphic-object plus a slot-name, n. Instances of the group will be created with a slot called n containing an instance of the graphic-object.
- a list of default lists. This is also a list of two-element lists. Each sub-list specifies a graphic-object plus a slot-name, n. Instances of the group will be created with: a slot named num-n, a slot named n-prototype containing an instance of the graphic-object, and a slot named n which will contain a list of num-n instances of the object in the n-prototype slot.

For example, a boxplot group might be defined as follows:
An instance of boxplot-group will consist of four graphic-objects, a rectangle and three lines. Instances of boxplot-group can inherit values for the slots \( x \), \( y\)-vble, and \( \text{width} \) from their parents, but slot \( y\)-vble cannot be inherited by their children (i.e., the four graphic-object instances). The slot \( y\)-vble will contain a variable.

The second step in creating a new group is to create an object which contains the high-level description of the group and specifies the rules relating this description to the components of the group.

For example, the following creates an object containing the description and rules for a boxplot:

\[
\begin{align*}
\text{(create-instance 'boxplot boxplot-group} & \quad \text{(:x 0.5)} \\
& \quad \text{(:width 0.2)} \\
& \quad \text{(:y-scale} \\
& \quad \quad \text{(o-formula (make-default-scale (gvl :y-vble))))} \\
& \quad \text{(:q} \\
& \quad \quad \text{(o-formula (convert-list} \\
& \quad \quad \quad \text{(R (cs "quantile"} \\
& \quad \quad \quad \quad \text{(Rvector (gvl :y-vble))} \\
& \quad \quad \quad \quad \quad \text "{})"))} \\
& \quad \quad \quad \text{(gvl :y-scale))})} \\
& \quad \text{(:upper-whisker-y (o-formula (elt (gvl :q) 3))} \\
& \quad \text{(:upper-whisker-height} \\
& \quad \quad \text{(o-formula (elt (gvl :q) 4)} \\
& \quad \quad \quad \text{ (elt (gvl :q) 3))}) \\
& \quad \text{(:lower-whisker-width 0)} \\
& \quad \text{(:lower-whisker-y (o-formula (elt (gvl :q) 0))} \\
& \quad \text{(:lower-whisker-height} \\
& \quad \quad \text{(o-formula (elt (gvl :q) 1)} \\
& \quad \quad \quad \text{ (elt (gvl :q) 0))}) \\
& \quad \text{(:lower-whisker-width 0)} \\
& \quad \text{(:median-x (o-formula (elt (gvl :x} \\
& \quad \quad \quad \text{ (/ (gvl :width) 2))))} \\
& \quad \text{(:median-y (o-formula (elt (gvl :q) 2))} \\
& \quad \text{(:median-height 0)} \\
& \quad \text{(:box-x (o-formula (elt (gvl :x} \\
& \quad \quad \quad \text{ (/ (gvl :width) 2))))} \\
& \quad \text{(:box-y (o-formula (elt (gvl :q) 1))} \\
& \quad \text{(:box-height (o-formula (elt (gvl :q) 3)} \\
& \quad \quad \quad \text{ (elt (gvl :q) 1))))})
\end{align*}
\]
A boxplot is described by its x-location (the :x slot), its width (the :width slot), and the variable it is plotting (the :y-vble slot). The slot :q contains a constraint which calculates the quantiles of the variable in slot :y-vble, in WC (if the value of slot :y-vble changes, the value of slot :q will change). The slot :upper-whisker-y contains a constraint which gets the fourth (indexing of lists is zero-based) element of the list in slot :q (i.e., the upper-quartile in WC; if slot :q changes, this slot will change). The slot :upper-whisker-height contains a constraint which calculates the difference between the maximum of the variable and the upper-quartile. The value of these two slots, along with the value in slot :upper-whisker-width, will be inherited by the line object in slot :upper-whisker. The line object will also inherit the value of the slot :x from the boxplot object. In this way, the rules of inheritance plus the automatic constraint maintenance will ensure that the line object in the slot :upper-whisker will behave like the upper-whisker line in a boxplot.

The remaining slots specify constraints which position the lower whisker of the boxplot, the median line and the box which indicates the inter-quartile range (both of the last two objects inherit the value of the :width slot from the boxplot object).

Instances of the boxplot object can now be used as plotting elements in an image. For example,

```
(create-instance 'y proto-variable
 (:data-list (R "rnorm(20)")))
(create-instance 'bp boxplot
 (:y-vble y))
(show bp)
```

Because the new group is created in the same manner as all of the predefined plotting elements, it has all of the same standard features. For example, the user can select any component of the boxplot with the mouse, any other plotting element can be added to a boxplot, and the boxplot can be added to any other plotting element.

3.5.9 The Motif interface

A Motif-style interface is provided for editing the descriptions of some Xtend objects. This interface is incomplete; there are only dialogs for the graphical primitives.

The Motif-style interface is not loaded by default, because of the size of the full Garnet system that is required. In order to use this interface the "Getting started" routine must be modified as follows: first, type `cmucl at the system prompt, then, at the Lisp prompt, type

```
(defvar *load-motif* T)
(load "init.lisp")
```

As with Simplisp, it is possible to select elements of a graph with the mouse (left mouse clicks select down the hierarchy of plotting elements, and right mouse clicks select up; see Section 2.5.3). With the Motif-style interface loaded, when the middle mouse button is pressed, a Motif-style dialog box is presented to allow editing of the whatever was the current selection.

The most useful aspect of this interface (which is otherwise a set of quite typical GUI dialog boxes) is the capability for specifying arbitrary units for each location and dimension field in the dialogs. For example, the dialog for a line (Figure 3.7) allows the user to edit the location, x and y, and dimensions, width and height,
of the line in any one of five units or coordinate systems. It is possible to view the locations and dimensions in a particular set of units by clicking on the menu button beside each of these fields and selecting the desired units.

Figure 3.7 demonstrates a number of the possible units available. The x location is in screen pixels. The y location is in normalised device coordinates, where (0, 0) is at the bottom left of the window and (1, 1) is at the top-right. The width dimension is in data coordinates, which are taken from the x-scale of the line and the height dimension is in object-world coordinates which are taken from the image’s window in WC. The fifth coordinate system is normalised image coordinates, where (0, 0) is at the bottom-left of the image window or viewport and (1, 1) is at the top-right.

In order to specify a value in a particular set of units, select the appropriate units via the menu button, enter the value in the field and press the “OK” button (the value is converted back to WC, which is the coordinate system used for the values of locations and dimensions in the slots of graphical objects).

### 3.5.10 An Advanced Xtend Example

This section provides the code from the graphics session described at the beginning of this chapter. Recall that the data to be plotted are two variables representing measures on a number of sheep. One variable represents the wool production for each sheep and the other variable represents the group membership for each sheep.

First of all, a plotting element, called an anova-series is defined. This plotting element will generate dot-plots of wool production for each sheep group. This has a y-variable containing values to plot and an x-variable which indicates the group that each value belongs to. The plotting element consists of dot-plots of the values in each group.

```
(define-group 'anova-series-group
```
CHAPTER 3. XTEND

:default-lists '(:symbols round-symbol))
:data-slots '(:x-vble :y-vble))

(create-instance 'anova-series anova-series-group
 (:x-scale
  (o-formula (make-default-scale (gvl :x-vble)))))
(:y-scale
  (o-formula (make-default-scale (gvl :y-vble)))))
(:num-symbols
  (o-formula (min (num-data (gvl :x-vble))
                 (num-data (gvl :y-vble))))))
 (:width .04)
 (:height .04))

Figure 3.1 was created by generating an x-variable and a y-variable and an anova-series based on these variables.

(create-instance 'x proto-variable
 (:data-list (r "sort(rep(seq(5),10))"))
(create-instance 'y proto-variable
 (:data-list (r "c(rnorm(10),rnorm(10),rnorm(10),
             rnorm(10),rnorm(10))")))

(create-instance 'as anova-series
 (:x-vble x)
 (:y-vble y))

(show as)

Next, a new plotting element, called a box-with-spread, is defined. This consists of a rectangle and a line and will generate a box which shows the mean and standard deviation for a single dot-plot. The y-location of the line is determined by the :middle of the box-with-spread; the height of the rectangle is twice the :spread of the box-with-spread and the rectangle is centred vertically on the line. The width of the line and the rectangle are determined by the :width of the box-with-spread.

(define-group 'box-with-spread-group
 :inherited-slots '(:x :middle :
                   :width :spread)
 :private-slots '(:x)
 :default-components '((:midline line)
                       (:box rectangle))))

(create-instance 'box-with-spread box-with-spread-group
 (:x .5)
 (:middle .5)
 (:width .2)
 (:spread .5)
 ;; slots for components
 (:midline-x (o-formula (- (gvl :x)
             (/ (gvl :width) 2)))))
 (:midline-y (o-formula (gvl :middle)))
 (:midline-height 0)
 (:box-x (o-formula (- (gvl :x)
             (/ (gvl :width) 2)))))

97
A couple of useful functions are defined to make the formulae which will follow easier to read.

\begin{verbatim}
(defun levels (group-vble)
  (r (cs "levels(factor(" (Rvector group-vble) ")))))

(defun means (data-vble group-vble)
  (r (cs "as.vector(tapply(" (Rvector data-vble) "," (Rvector group-vble) ", mean)")))

(defun sds (data-vble group-vble)
  (r (cs "as.vector(tapply(" (Rvector data-vble) "," (Rvector group-vble) ", sd)")))
\end{verbatim}

The dot-plots are now prepared for incorporating the new box-with-spread plotting elements. New slots are added to the anova-series plotting element in preparation for the addition of a list of box-with-spread elements to represent the mean and spread of each group. The values in these slots will be inherited by the box-with-spread elements.

\begin{verbatim}
(edit-slot as :num-boxes
  (o-formula (length (levels (gvl :x-vble)))))

(edit-slot as :boxes-middle-list
  (o-formula
    (convert-list (means (gvl :y-vble)
      (gvl :x-vble))
      (gvl :y-scale)))))

(edit-slot as :boxes-spread-list
  (o-formula
    (convert-list (sds (gvl :y-vble)
      (gvl :x-vble))
      (gvl :y-scale) :locn-p nil)))

(edit-slot as :boxes-x-list
  (o-formula
    (convert-list (levels (gvl :x-vble))
      (gvl :x-scale)))))

(edit-slot as :boxes-width .05)

(add-list box-with-spread :boxes as)
\end{verbatim}

Figure 3.2 was created by adding a list of box-with-spread elements to the anova-series element.

Now, further slots are added to the anova-series plotting element in preparation for the addition of another dot-plot to represent the means of all of the groups. A list of round-symbols is added to implement the dot-plot.
(edit-slot as :num-mean-symbols
  (o-formula (gvl :num-boxes)))
(edit-slot as :mean-symbols-y-vble
  (o-formula
    (create-instance nil proto-variable
      (:data-list (means (gvl :y-vble)
                     (gvl :x-vble))))))
(edit-slot as :mean-symbols-x-vble
  (o-formula
    (create-instance nil proto-variable
      (:data-list
        (make-sequence
         'list (gvl :num-boxes)
         :initial-element
         (gvl :x-scale :max))))))
(edit-slot as :mean-symbols-colour red)

(add-list round-symbol :mean-symbols as)

Some more useful functions are defined to simplify formulae.

(defun mean (vble)
  (car (r (cs "mean(" (rvector vble) ")"))))
(defun sd (vble)
  (car (r (cs "sd(" (rvector vble) ")"))))

Further slots are added to prepare for the final box-with-spread which represents
the derived spread of the group means and a box-with-spread component is added
which inherits the values in the new slots. This completes Figure 3.3.

(edit-slot as :mean-box-middle
  (o-formula (convert (mean (gvl :y-vble)))
             (gvl :y-scale)))
(edit-slot as :mean-box-spread
  (o-formula
    (convert
     (* (sd (gvl :mean-symbols-y-vble))
        (sqrt (/ (num-data (gvl :y-vble))
                (gvl :num-boxes)))
        (gvl :y-scale) :locn-p nil))))
(edit-slot as :mean-box-x
  (o-formula (gvl :x-scale :ow-max)))
(edit-slot as :mean-box-width .05)
(edit-slot as :mean-box-colour green)

(add-component box-with-spread :mean-box as)

Figure 3.4 was created by generating a new y-variable and making this the y-
variable in the anova-series.

(create-instance 'y2 proto-variable
  (:data-list (r "c(rnorm(10,5),rnorm(10),rnorm(10),

---
rnorm(10),rnorm(10))

(edit-slot as :y-vble y2)
3.6 Results and Conclusions

3.6.1 Extensibility

In Simplisp, there are two major obstacles to creating new plotting elements: firstly, there are a large number of slots associated with the high-level description of an object; secondly, there is the difficulty of propagating modifications of the description of an object to the description slots of components of the object and to other slots in the object itself. Xtend solves the first problem by allowing the user to easily access the components of an object directly via the edit-slot syntax. This means that higher-level objects do not need to have copies of the description-slots of their components because the user can easily directly access the description of the components. Xtend solves the second problem by being built on top of KR, which provides automatic maintenance of arbitrary constraints.

Consider the simple boxplot described in Section 3.5.7. The high-level description of the boxplot consists of a variable to plot, the width of the boxplot and the x-location of the boxplot. The y-scale and the quantiles of the variable determine the y-location and dimensions of the boxplot. Suppose that a boxplot is created by typing,

\[
\text{(create-instance 'my-boxplot boxplot)}
\]

Basic modifications to the high-level description of the boxplot are simple to specify because there is a top-level symbol for the boxplot (i.e., my-boxplot). For example,

\[
\text{(edit-slot my-boxplot :width .3)}
\]

can be used to modify the width of my-boxplot.

The boxplot has three line-primitives and a rectangle-primitive as its components. It might be useful to be able to modify the descriptions of these components as well (e.g., modify the line type in the lines which represent the whiskers so that the whiskers are dashed lines, or shade the inside of the rectangle). One way to allow this is to add to the high-level description of the boxplot (e.g., add :upper-whisker-line-type, :lower-whisker-line-type, and :box-shading slots). The problem with that approach is that the high-level description of the boxplot then becomes very cluttered and it is very difficult, if not impossible, to provide slots in the boxplot for all possible modifications to its components. The approach in Xtend is to make it easy to access the components themselves. For example,

\[
\begin{align*}
\text{(edit-slot my-boxplot :upper-whisker :line-thickness 2)} \\
\text{(edit-slot my-boxplot :lower-whisker :line-thickness 2)} \\
\text{(edit-slot my-boxplot :median :colour red)} \\
\text{(edit-slot my-boxplot :box :filled T)}
\end{align*}
\]

The user is able to directly access the slots of a component so there is no need to have a repetition of the slots of each component in the description of the boxplot.

The propagation of modifications of the high-level description of a plotting element is essentially the problem of maintaining a constraint. There is a relationship between the high-level description of an object and the descriptions of the components of the object. The usefulness of Xtend and KR is that this relationship only needs to be declared; the system takes care of maintaining the relationship. In practice, this means that the creator of a new plotting element is not required to
write an edit function for each high-level description slot in order to preform the necessary propagation of changes to the components. Instead, all that is required is a statement of how the value of one slot is calculated from the value of another slot; all slots are modified in the same way (with edit-slot) and the system ensures that changes propagate correctly.

Because it is so simple to create new plotting elements, the user is provided with the ultimate powers of customisation. Suppose that the user wants to colour the top-half of the box (which represents the inter-quartile range) in a boxplot differently from the bottom-half. This is not possible with the boxplot created in Section 3.5.7 (either through high-level description of the boxplot or through the descriptions of the components of the boxplot); what is required is a structural change. The easy extensibility of Xtend means that the user is able to create a new boxplot plotting element with two line-primitives (for the whiskers) and two rectangle-primitives (for the top-half and bottom-half of the inter-quartile range; see Figure 3.8).

3.6.2 Incremental Graphics

The design of graphical primitives and the plotting elements in the Xtend system are based upon a sort of pins-and-sockets model. Each object has a set of pins (inherited slots) and sockets (slots for its components or list items plus possibly some high-level description slot which are not private slots). When graphical object $A$ is added to graphical object $B$, some of $A$'s pins find sockets in $B$. In this way, when graphical objects are added together they automatically share some information. This allows, for example, a plot to provide an x-scale socket and a y-scale socket so that any object with an x-scale pin or a y-scale pin will automatically share the plot's scales and the components of a plot will be consistent (see also Figure 3.6).
In addition, this approach makes graphical objects very autonomous. A graphical object does not have to have its pins plugged into any other object, nor does it require any of its sockets to be filled. The potential for sharing exists, but it can remain unfulfilled. The advantage of this is that the designer of a new graphical object can just focus on the structure of that graphical object—the pins that it has and the sockets that it provides; the system takes care of what happens when other objects are actually added to the new object (i.e., fitting the right pins into the right sockets).

Adding further to the interactive nature of the system, sockets can be created at any time. This means that a graphical object can be customised to accept any sort of pin as and when required and this makes it very easy to experiment with new graphical structures.

Another major advantage to this design is the freedom it brings to the structure of graphical objects which are traditionally quite restricted. For example, in a traditional system, there are a quite limited set of plotting symbols available. In Xtend, a symbol-series is a graphical object with a list of symbols. It provides sockets for an x-variable, a y-variable, a symbol-width and a symbol-height. Any graphical object could be made the prototype for this list of symbols and make use of these sockets. For example, the following code creates a new sort of plotting symbol which would work immediately with the existing symbol-series:

```lisp
(define-group 'new-symbol-group
  :default-components '(:box rectangle)
                     (line line))
  :inherited-slots '(:x-vble :y-vble
                      :width :height
                      :line-y-offset)
  :data-slots '(:x-vble :y-vble))

(create-instance 'new-symbol new-symbol-group
  (:rank 0)
  (:width 0.05)
  (:height 0.05)
  (:line-y-offset 0.5)
  (:box-x
    (o-formula
     (- (get-data-location (gvl :x-vble) (gvl :rank)
         (gvl :x-scale))
        (/ (gvl :width) 2))))
  (:box-y
    (o-formula
     (- (get-data-location (gvl :y-vble) (gvl :rank)
         (gvl :y-scale))
        (/ (gvl :height) 2))))
  (:line-x (o-formula (gvl :box-x)))
  (:line-height 0)
  (:line-y (o-formula (* (gvl :box-y)
                        (gvl :line-y-offset))))))
```

This symbol draws a square centred at the appropriate data location with a horizontal line across the middle.
New sockets could even be created to accommodate novel pins in the prototype. For example, the symbol described above has a pin for the vertical position of the horizontal line within the rectangle (i.e., the :line-y-offset slot is inherited). A new socket could be added to a series to provide a value for the new symbols to inherit as follows:

\[
\text{create-instance 'ss symbol-series}
\text{ (;:symbols-prototype new-symbol))}
\text{edit-slot ss :symbols-line-y-offset-list}
\text{(o-formula}
\text{(R (cs "seq{1:"}
\text{ (text (gv1 :num-symbols}))
\text{") /"}
\text{(text (gv1 :num-symbols))))))}
\]

This new socket provides a value for each new symbol in the series; each symbol will draw its horizontal line at a different height within the symbol square (corresponding to the order statistic of the symbol—i.e., the symbol representing the first data value will draw its line at the bottom of the square and the symbol representing the last data value will draw its line at the top of the square). Figure 3.9 shows an example data series using the new symbols.

Of course, if this is still too restrictive, the user can always create a completely new sort of series object to meet their requirements (for example, a series which plotted more than two variables).

It may seem that this design is too complex for creating simple, static graphical objects (like a square symbol). There is certainly no distinction made between objects which (typically) have a stable structure (like a data symbol) and objects which have a somewhat flexible structure (like a scatter-plot which can have additional data series added to it). However, as long as the additional flexibility does not come at a cost, then it can only be a bonus. The creation of objects with an initial structure is (mostly\textsuperscript{3}) only a convenience; for example, a square-symbol could be created incrementally as follows:

\[
\text{define-group 'square-symbol-group}
\text{:default-components NIL}
\text{:inherited-slots '(:x-vble :y-vble)}
\text{:data-slots '(:x-vble :y-vble))}
\text{create-instance 'square-symbol square-symbol-group}
\text{(:width .05)}
\text{(:height .05))}
\text{create-instance 'rect rectangle)}
\text{add-component rect :rect square-symbol)}
\text{edit-slot square-symbol :rect-x}
\text{(o-formula}
\text{(- (get-data-location (gv1 :x-vble) (gv1 :rank)}
\text{(gv1 :x-scale))}
\text{(/(gv1 :width) 2))))}
\text{edit-slot square-symbol :rect-y}
\text{(o-formula}
\text{(- (get-data-location (gv1 :y-vble) (gv1 :rank))}
\text{\textsuperscript{3}the only problem is that it is not, currently, possible to add or remove inherited, private, or data slots other than through define-group

104
Figure 3.9: A data-series using square symbols which have a horizontal line to indicate the symbol's order statistic.
(gvl :y-scale))
   (/ (gvl :height) 2))]

One problem with the inheritance mechanism in Xtend is the possibility that a graphical object may inherit a value that was not intended for it; I call this accidental inheritance. Consider the x-axis plotting element, which provides a :label-height slot for its :label component (a text plotting element) to inherit and a :ticks-label-height slot for the items in its :ticks list (which are tick-mark plotting elements) to inherit. If the user destroys the :ticks-label-height slot in order to stop the tick-marks inheriting a label height from the axis the tick-marks in the :ticks list will instead inherit the :label-height slot (which is meant only for the text which represents the axis label). This is essentially a name-space problem. Unfortunately, the solution is for the user to be a bit careful when choosing the names of components (for example, the conflict above could be resolved by naming the slot which contains the text representing the axis label :axis-label); this is undesirable because it complicates the creation of new plotting elements.

3.6.3 Stand-Alone Graphics

The communication with the R system is extremely beneficial. Data can be manipulated, analysed, even loaded using R. This not only simplifies and focuses the initial writing of Xtend (because these features do not have to be implemented in Xtend), but also the ongoing development of Xtend (because Xtend automatically keeps pace with the development of the features in R).

The generation of commands to send to R is a little inconvenient, but this could be easily fixed by providing wrapper functions, at least for the most common statistical functions. For example,

(defun mean (x)
   (car (R (cs "mean(" (Rvector x ")")"))))

(defun median (x)
   (elt (R (cs "quantile(" (Rvector x ")")")) 2))

The user is unable to interact directly with R—the only contact with R is through the Xtend interface—but this is a reasonable price to pay in order to have well-synchronised communication between R and Xtend; R does nothing until Xtend sends a message then R executes the message, returns a result and goes back to waiting.

3.6.4 Problems

The major and, at this stage, fatal problem with the Xtend system is the speed at which it runs and the memory it consumes. The very simple examples given in this chapter should run quickly or at least with only a small pause, but the creation of a scatterplot, without any data symbols, takes more than a minute and involves sufficient memory allocations (consing) to trigger five or six garbage collections (with a collection limit of 10 MB!). This sort of inefficiency is totally unacceptable.

\footnote{\textsuperscript{4}It is necessary to delete a slot in order to stop components from inheriting a value; this might be done in order to preserve a customisation in a component that would otherwise be overridden, or to move between inheritance rules - for example, so that the items in a list all inherit a single value rather than each inheriting their own value from a list.}
The problem is the large number of KR constraints within the system, particularly in the implementation of the inheritance mechanism. An early version of the system, which did not include inheritance (but still used constraints to specify relationships between the high-level description of a graphical object and the arrangement of the components of the object), ran acceptably. With the introduction of inheritance, the number of constraints more than doubled and the system slowed enormously.

A possible solution would be to implement inheritance without constraints. It is important to retain constraints in the specification of a new plotting element so that the system is easily extensible, however, the inheritance mechanism is completely internal (i.e., the user never has to deal with the implementation of inheritance) so this could be implemented by hand (i.e., without constraints) in order to gain efficiency.

Another problem with Xtend is that it is not very useful for creating plotting elements that involve a large number of graphical primitives (e.g., a 3D mesh). The specification of rules for each primitive would quite quickly become tedious. A possible solution to this would be to provide more complex graphical primitives—complex objects which do all their own drawing (rather than owning other objects which do the drawing). This is essentially a return to the more conventional statistical graphics system, but only in the sense of incorporating the useful features of such systems; the main usefulness of Xtend is for creating relatively simple, highly flexible plotting components, but this could be incorporated with more traditional capabilities as well.

Finally, the Xtend interface is an improvement on Simplisp in terms of user-friendliness (e.g., the default output window and image, plus the show function), but there are still some quite major inconveniences. In particular, positioning plotting elements within an image and an image within an output is still quite an awkward task. The next chapter describes an attempt to address these issues.

**SUMMARY** Xtend provides good support for extensible statistical graphics by defining a single coherent framework for developing new plotting elements. Xtend’s inheritance mechanism allows graphs to be constructed incrementally which promotes experimentation, particularly with new graph structures. Xtend provides flexible and detailed communication with R which gives important support for graphical extensibility. Unfortunately, Xtend is too inefficient in terms of memory and speed for anything other than simple demonstrations of its potential uses.

---

5 This does not mean that a good solution would be to ditch inheritance altogether! The early version without inheritance had very poor support for sharing information between plotting elements.
3.7 References


Chapter 4

SPG

4.1 Aims

The previous two chapters describe experimental software packages which investigate ways of increasing the flexibility, extensibility, and accessibility of statistical graphics software. Those software packages were constructed in a bottom-up fashion, creating powerful foundations upon which to build generic plotting elements; the top-level mechanism for viewing the plotting elements was treated as a necessity rather than as a focus of the design of the systems.

The software described in this chapter focuses more upon top-level mechanisms; the way in which the user views and interacts with plotting elements.

4.1.1 Arranging graphs on a page

By default, most systems generate a graph which fills the entire region of whatever output it is being viewed on. In many cases, there is a need for the user to control the region occupied by the graph. For example, the user might want the graph to occupy a region of a fixed size (e.g., 4 inches square). If several graphs are being viewed on the same output, there is a need to allocate them non-overlapping regions (e.g., divide the output into two halves and show each graph in one half).

Any arrangement of plots\textsuperscript{1} can be achieved by specifying the left, bottom, width, and height of each plot (see Figure 4.1). In many cases, however, this level of description is unnecessarily detailed and the determination of the appropriate specifications for each plot can be time-consuming and inconvenient.

Many existing software packages provide higher-level descriptions for the arrangement of plots. For example, in the S system (Chambers, Becker, and Wilks 1988), the statement

\[
\text{par(pty="s")}
\]

specifies that all figure regions (within which the plots are drawn) must be square, and the statement

\[
\text{par(mfrow=c(2,2))}
\]

specifies that the next four plots will be arranged together on a page in a $2 \times 2$ matrix.

\textsuperscript{1}This assumes that plots are rectangles which cannot be rotated
More recently, the Trellis Graphics package (Becker and Cleveland 1996) provides a command to subdivide a page into an array of subregions and specify which subregion a plot should occupy. For example, the statements,

```r
plot1 <- xyplot(y ~ x)
print(plot1, split=c(1,2,2,3))
```

divides the page into an array with two columns and three rows (based on the third and fourth values in the `split` argument) and draws `plot1` in the first column of the second row (based on the first and second values in the `split` argument).

An aim of the SPG package is to provide a mechanism for arranging plots on a page which is powerful and flexible, but also intuitive.

### 4.1.2 Coordinate systems and units

Much of the drawing involved in the creation of a graph occurs within the data coordinate system (i.e., the scale that is used to position data symbols and to determine the labels on tick-marks). Most statistical graphics systems provide a number of other coordinate systems, which are more appropriate for specific tasks, such as the positioning of a graph on an output. For example, the S system provides the following set of coordinate systems (among others; see Figure 4.2): an outer margin for positioning outer margin text, such as page titles; a figure margin per figure for positioning figure margin text, such as axis labels; and a plot area per figure for positioning data elements, such as data symbols.

Locations within the plot area are in terms of the standard \((x, y)\) cartesian coordinates, but locations within the margins are in terms of distance "out" from the inner edge of the margin (i.e., up from bottom of top margin, left from right of left margin and so on; see Figure 4.3).

This sort of approach provides excellent support for certain layout tasks (e.g., positioning text in the margins), but it is somewhat restrictive and incomplete. The approach is restrictive because the different coordinate systems are only available
Figure 4.2: Coordinate systems in S (adapted from a figure in S-Plus User's Manual, Version 3.2).

Figure 4.3: The S margin coordinate systems.
to certain drawing operations. For example, it is only possible to draw text in the margins (not lines and points). The approach is incomplete because there are several other coordinate systems that would be useful for different tasks. For example, in order to draw some text in the middle of a plot would require knowing the scales on each axis (because text within a plot has to be located in terms of data coordinates); it would be useful to have some sort of normalised plot coordinates so that the centre of the plot, \((0.5, 0.5)\), could be specified regardless of the axis scales.

The other aim of SPG is to provide a comprehensive set of coordinate systems and make them available for all drawing operations.

**SUMMARY** The first aim of SPG is to provide a convenient, intuitive, and powerful layout mechanism for arranging plots on a page. The second aim of SPG is to provide a number of different coordinate systems for specifying locations and dimensions and to make these coordinate systems available for all drawing operations.
4.2 Design

SPG retains some of the features of Simplisp and Xtend (such as automatic redrawing of outputs and access to lower-level plotting elements), but focuses more on the user-interface, a mechanism for arranging graphs, and the provision of arbitrary coordinate systems.

4.2.1 Constructing a graph

A graph is constructed by combining a number of gobs (graphical objects—not to be confused with Pictor’s grobs; Wilks, 1996) and primitives together within a picture. Primitives are simple graphical objects such as lines and text and gobs are complex plotting elements such as axes and plots. The more complex gobs can own other gobs (e.g., a plot owns an x-axis, a y-axis, and some points).

A picture is viewed by adding it to an output.

Outputs, pictures, and gobs have a direction (left, right, up, or down), which determines the “up” direction for their children. For example, if the direction of an output is left, then for each picture in the output, “up” is to the left (i.e., the picture is rotated 90 degrees anticlockwise). If the direction of an output is left and the direction of a picture in the output is also left then “up” for any gobs in the picture is down (i.e., directions are cumulative).

4.2.2 Arranging plots

Pictures and gobs have a left, bottom, width, and height so that a picture may be positioned anywhere on an output, and a gob can be positioned anywhere within a picture (or within another gob).

The arrangements of pictures within an output, of gobs within a picture, and of components within a gob can also be controlled by layouts.

A layout is a matrix of names (picture names, gob names, or component names) which specifies how a region (the output, the picture, or the gob) is divided up into rows and columns and determines the subregion that is allocated to each named object.

4.2.3 Coordinate systems

The location and dimensions of a picture within an output, a gob within a picture, and a component or primitive within a gob can be specified in any one of three coordinate systems or units:

:cm this is an absolute coordinate system with (0, 0) at the bottom-left corner of the parent (the parent of a picture is an output, the parent of a gob is a picture or another gob, and the parent of a primitive is a picture or a gob) and (width_in_cm, height_in_cm) at the top-right.

:npc this is a relative coordinate system with (0, 0) at the bottom-left corner of the parent and (1, 1) at the top-right.

nil this is the “native” coordinate system of the parent with (x.min, y.min) at the bottom-left corner and (x.max, y.max) at the top-right (i.e., it depends on the x-scale and y-scale of the parent).
SUMMARY A layout mechanism is provided for arranging plots on a page and for arranging the components of a plot within the plot. All locations and dimensions in SPG are specified with a unit.
4.3 Implementation

SPG was implemented in Common Lisp (CMUCL) using CLOS (see Section 2.3).

4.3.1 SPG classes

Fundamental classes

The three fundamental classes in SPG are the updatable class, the vp class, and the viewport class.

An updatable object has a list of children and a list of pictures. The list of pictures is used to perform redrawing when an updatable is modified. Not all instances of the updatable class make use of the list of pictures. The list of children can be used to maintain the list of pictures in the children of an updatable object. For example, when a picture is added to an updatable, usually when a gob or primitive is added to a picture, the picture is also added to all of the children of updatable. Also, when a new child is added to an updatable, all of the updatable’s pictures are added to the child. The list of children is also used to draw all of the children of an updatable when the updatable is drawn (e.g., when an output is drawn it draws all of its pictures and when a picture is drawn it draws all of its gobs).

A vp has a location (an x and a y), a size (a width and a height), a direction, an x-scale, a y-scale, and a width in cm and a height in cm. A vp represents a rectangular region within its parent (i.e., a location and size) and specifies coordinate systems for that region (in terms of absolute size in cm and “native” scale).

The viewport class is derived from the updatable and vp classes (i.e., a viewport is a rectangular region with its own coordinate system and a number of children which will be drawn within the region). A viewport has a layout which specifies the arrangement of its children.

primitives

The primitive classes (the line, the rectangle, and the text classes) are all derived from the updatable class (i.e., they are objects which, directly or indirectly, appear in pictures).

A line or rectangle consists of a location (x and y) and size (width and height). A text object has a location, a height, an angle of rotation, horizontal- and vertical-justification, and a string (the characters in the text).

gobs

The gob classes (the x-axis, y-axis, points, and plot classes) are all derived from the viewport class (i.e., they are objects which, directly or indirectly, appear in pictures, they specify a region with coordinates, and they may have children to draw).

For example, a points object specifies a rectangular region and a coordinate system within that region which is used to locate the data symbols (and any other primitive or gob which is added to the points object).

pictures

The picture class is derived from the viewport class (i.e., a picture defines a region with coordinates and has children to draw; the children of a picture are gobs).
outputs

The output classes (the xll-window and postscript-file classes) are derived from the viewport class (i.e., they are objects which specify a region with coordinates and they have children to draw; the children of an output are pictures).

4.3.2 Drawing

When the contents of an output need to be drawn, the output calls the generic draw function on all of its children (i.e., for each picture in the output).

The draw method for viewports calls the generic draw-self function (to do any drawing for the viewport itself), determines regions for the viewport's children, based on the viewport's layout, and then calls draw for each of the viewport's children. The draw method for each primitive performs the appropriate drawing operation (e.g., the draw method for the line primitive draws a line).

The draw-self method for pictures performs clipping to ensure that no drawing occurs outside the picture region. The draw-self methods for gobs with no components (i.e., x-axis, y-axis, and points) perform the appropriate drawing operations (e.g., the draw-self method for points draws a square centred at each data location).

This means that the drawing command is issued from the output to each of its pictures. Each picture performs clipping, arranges its children (gobs and primitives), then passes the command onto each of its children. Each primitive performs the appropriate drawing. A gob with components arranges the components then passes on the drawing command to each component. Each component which has no components of its own performs the appropriate drawing commands.

4.3.3 Getting and setting slots

The standard way of accessing a slot in CLOS is to use the following sort of statement:

```
;; to get the value in a slot
(slot-value my-object 'my-slot)
;; to set the value in a slot
(setf (slot-value my-object 'my-slot) a-value)
```

When objects are arranged in a hierarchy (e.g., gobs within pictures or components within gobs), this can become cumbersome. For example, ...

```
(slot-value (slot-value parent-object 'child) 'child-slot)
```

SPG has functions for accessing slots which allow concatenation of slot specifiers (similar to KR's g-value function; see Section 3.3). For example,

```
(get-slot my-object 'my-slot)
(get-slot parent-object 'child 'child-slot)
(set-slot my-object 'my-slot a-value)
(set-slot parent-object 'child 'child-slot a-value)
```

When a slot is modified, there is often a need for subsequent changes to slots within the same object or to slots in other objects. SPG provides a general mechanism for propagating changes, which is designed to allow easy extensibility.
There are three different sorts of updates that can occur; an update just prior to a slot being set (a pre-update), an update just after a slot has been set (a post-update), and an update of the output (a draw-update). This allows for the same sort of updating mechanisms that were implemented in Xtend by providing similar service to the automatic updating mechanisms that were provided by KR (i.e., automatic constraint maintenance and the *pre-set-demon*).

A slot can be set without any updating occurring by using the *sets* (pronounced "set s") function. The *set-slot* function forces a pre-update and a post-update for the slot being modified and the *edit-slot* function forces a pre-update, a post-update, and a draw-update to occur.

The *edit-slot* function is designed for the user. Whenever the user modifies a slot there should be a propagation of necessary changes and then a redraw.

The *set-slot* function is designed for internal use, when a slot is to be modified and this change should be propagated, but no drawing should occur (a good example is the setting of slots within pre- or post-update code).

The *sets* function is also intended for internal use. In many cases, several slots need to be set at once (e.g., the x, y, width, and height of a viewport). Further, the consequences of modifying each of these slots may be identical or at least very similar (e.g., setting any of the x, y, width, or height of a viewport will probably cause the viewport to modify the x, y, width, and height of its children). In this sort of situation, the programmer does not want the change to propagate when each slot is modified (effectively, it is possible to end up with the same sort of updates occurring several times - i.e., massively redundant). The *sets* function allows a slot to be modified without the change propagating. However, this is not the whole story. What is typically desired is that the propagation should occur only once. The *sets* function does not cause any propagation so the programmer must force updates explicitly. One solution is to choose one of the set of slots being modified and call the update code for that slot. This is not very safe, however, because if the changes that are propagated by modifying each slot are not identical then some, possibly vital, propagated changes will not occur. Fortunately, the format of the update functions provides a better solution. The pre- and post-update functions each allow an arbitrary number of slot parameters. Within the update code the programmer should use the *intersection* function to determine whether a particular update activity should occur. This allows the programmer to modify several slots at once using the *sets* function and bracket these with calls to pre- and post-update, passing the complete set of slots being modified. For example,

```
(defmethod post-update ((my-object my-class) &rest slots)
  (if (intersection slots '(slot-1 slot-2 slot-3))
    (update-action-1 my-object))
  (if (intersection slots '(slot-1))
    (update-action-2 my-object))
  (if (intersection slots '(slot-2 slot-3))
    (update-action-3 my-object))
  (if (intersection slots '(slot-3))
    (update-action-4 my-object)))

(defun whatever (my-object ...)
  ...
  (pre-update my-object 'slot-1 'slot-2)
  (sets my-object 'slot-1 some-value)
```
(sets my-object 'slot-2 some-other-value)
(post-update my-object 'slot-1 'slot-2)

The pre-update call does nothing because there is no pre-update method defined for my-class. The set calls set slot-1 and slot-2 with the new values and the post-update call runs update-action-1, update-action-2, and update-action-3. If set-slot was used instead of set, then update-action-1 would be run twice (unnecessarily) and if post-update was called with only one of 'slot-1 and 'slot-2 then either update-action-2 or update-action-3 would not be run.

4.3.4 Layouts

All viewport objects (outputs, pictures, and gobs) have a layout. A layout is a structure which describes how the children of a viewport are arranged within the viewport’s rectangular region. For example, layouts are used to describe how a number of pictures are arranged within an output (e.g., specify a matrix of scatterplots), how gobs are arranged within a picture (e.g., specify that the plot region is the largest square possible in the picture region), and how gobs are arranged within other gobs (e.g., specify the arrangement of the axes and points within a scatterplot).

A layout is a structure consisting of:

1. a matrix of symbols (which specify children of the viewport object)
2. a list of rowheights (which default to one)
3. a list of columnwidths (which default to one)
4. a logical flag or a matrix to indicate whether the height of row i and the width of column j are to be respected (see below)
5. a vector of margins
6. a logical flag or matrix to indicate whether the marginrespect (basically whether all of the margins are the same physical width; see below)

The row heights, the column widths, and the margins may be specified in centimetres (cm).

For example, suppose that six pictures are to be arranged on an output so that there are three rows and two columns of pictures, all of the same size, and together they completely occupy the output surface (an arrangement that would be useful for generating a scatterplot matrix). A layout for this arrangement would look like:

1. matrix = \[
\begin{bmatrix}
p1 & p2 \\
p3 & p4 \\
p5 & p6 \\
\end{bmatrix}
\]
2. rowheights = \[
\begin{bmatrix}
1 \\
1 \\
1 \\
\end{bmatrix}
\]
3. columnwidths = [1 1 1]
4. \( \text{respect} = \text{false} \)

5. \( \text{margins} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \)

6. \( \text{marginrespect} = \text{true} \)

A layout structure is created using the `make-layout` function. The following expression creates a layout for the example above (all of the keyword arguments could actually be left out because the default values will be correct in this case):

\[
\text{(make-layout 3 2 '((p1 p2) (p3 p4) (p5 p6)) :heights '(1 1 1) :widths '(1 1) :respect nil :margins '(0 0 0 0) :m-respect T)}
\]

The next section describes how the layout structure is converted into the regions that should be allocated to each of the children of the viewport.

generating layout regions

Consider the following layout:

\[
\text{(make-layout 2 2 '((p1 p2) (p3 p3)))}
\]  

(4.1)

This layout describes three distinct components: p1, p2, and p3. The function `layout-info` takes a layout and the dimensions of a rectangular region and returns the portions of the overall region which are to be allocated to the components.

The general algorithm involves dividing the overall region into a number of rows and columns (corresponding to the number of rows and columns in the layout matrix; in the example above, the overall region would be divided into two rows and two columns). Next, each row is allocated a portion of the height of the overall region and each column is allocated a portion of the width of the overall region. The portions allocated to each row and column depend on the row heights, the column widths, and the respect of the layout (specific cases are described in detail below; in the simple example above, each row is allocated half of the height of the overall region and each column is allocated half of the width of the overall region). Figure 4.4 shows how the overall region is divided up for the example layout given above.

The region allocated to each region is described by a vector (left, bottom, width, height). Component p1 occupies the top-left cell of the layout matrix so it is allocated the top-left cell of the output, which is described by \((0.0, 0.5, 0.5, 0.5)\) (see Figure 4.5).

Similarly, the regions allocated to components p2 and p3 are \((0.5, 0.5, 0.5, 0.5)\) and \((0.0, 1.0, 0.5)\) respectively.
Figure 4.4: The division of the overall region into rows and columns for the layout described in expression 4.1.

Figure 4.5: The region allocated to component p1 for the layout in expression 4.1.

Calculating row-height and column-width proportions

Once the overall region has been divided into a number of rows and columns, the heights of the rows and the widths of the columns are calculated depending on the row-heights, the column-widths, and the respect of the layout.

Consider a layout with row heights and column widths all expressed in cm:

\[
\text{(make-layout 1 2 '((p1 p2)))}
\]

\[
\text{:heights '((4 :cm))}
\]

\[
\text{:widths '((2 :cm) (4 :cm))}
\]

The proportional row-heights and column-widths (heights/widths hereafter) depend on the absolute size of the overall region. For example, if the overall region is 8 cm high and 6 cm wide then the heights are

\[
\frac{1}{2} = \frac{4cm}{8cm}
\]

and the widths are

\[
\frac{1}{3} = \frac{2cm}{6cm}
\]

and

\[
\frac{2}{3} = \frac{4cm}{6cm}
\]

(see Figure 4.6).
CHAPTER 4. SPG

Figure 4.6: The layout in expression 4.2 arranged within a region which is 8cm high and 6cm wide.

Figure 4.7: The region allocated to p₁ by the layout in expression 4.1 with the layout aspect ratio respected.

If all of the column-widths and row-heights in the layout are not in cm, then the conversion to proportional column-widths and row-heights depends on the layout respect.

If there is no respect (i.e., the respect is nil) then the proportional widths/heights are found by dividing the original widths/heights by the sum of all widths/heights. This was the case for the layout in expression 4.1 (see Figure 4.4).

If there is full respect (i.e., the respect is T), intermediate widths/heights are calculated as for no respect, then final widths/heights are found by multiplying the intermediate widths/heights by the width/height of the largest rectangle that can fit into the overall region that respects the layout aspect ratio. The aspect ratio of a layout is the sum of the row heights divided by the sum of the column widths. The “largest region” is expressed as a width and height which are proportions of the parent region—one of them will always be one and the other will always be less than one. For example, if the layout aspect ratio is 1 (i.e., a square) and the parent region has an aspect ratio of \( \frac{3}{2} \) (i.e., it is a rectangle that is higher than it is wide) then the full width of the overall region is used, but only three quarters of the height is used (i.e., the multipliers are 1 and 3/4). Figure 4.7 shows the region allocated to p₁ by the example layout with the layout aspect ratio respected.

If there is partial respect (i.e., the respect is a matrix) then the conversion depends on whether the aspect ratio of the layout is “taller” than the aspect ratio of the overall region.

If the overall region is taller, the row heights which are not to be respected are increased so that the layout aspect ratio with the new heights is the same as the aspect ratio of the overall region. The converted widths and heights are then
calculated from the original widths and the adjusted heights, treating the layout with total respect. Consider a variation on the layout in expression 4.1 where there is partial respect:

\[
\text{(make-layout 2 2 '((p1 p2) (p3 p3)) :respect '((0 0) (1 0))) (4.3)}
\]

Figure 4.8 shows the region allocated to p1 by this layout (compare this with Figure 4.4). The arrow indicates that the height of the top row has been stretched so that the layout aspect ratio is the same as the aspect ratio of the overall region. The effect of the partial respect in this case is to ensure that the bottom-left allocated region is square (i.e., has an aspect ratio of 1).

If the layout is taller, the column widths which are not to be respected are increased so that the layout aspect ratio with the new widths is the same as the aspect ratio of the parent region. The converted widths and heights are then calculated (from the adjusted widths and the original heights) treating the layout with total respect.

If some but not all of the widths are in cm and some but not all of the heights are in cm, then the widths and heights are split into cm-widths/heights and non-cm-widths/heights. The cm-widths/heights are calculated using the subregion of the overall region which is occupied by cm-rows and cm-columns and the non-cm-widths/heights are calculated using the subregion of the overall region which is not occupied by cm-rows and cm-columns. Consider the following, rather complicated layout:

\[
\text{(make-layout 3 3 '((p1 p2 p3) (p4 p2 p3) (p5 p5 p5)) :heights '(2 1 (2 :cm)) :widths '(3 (2 :cm) 1)) (4.4)}
\]

There is one cm-row (the bottom row) and one cm-column (the middle column). Figure 4.9 shows the subregion of the overall region which is occupied by cm-rows and cm-columns (the hatched area) for an overall region which is 8cm high and 6cm wide. The non-cm rows and columns have been allocated using what is left over from the overall region (the first row is twice as high as the second row and the first column is three times as wide as the last column).
The separate widths/heights are then recombined; adjustments are made for the fact that the widths/heights were calculated using only a subregion of the overall region.

If some but not all of the widths are in cm and all of the heights are in cm, then the widths are converted as for some but not all widths in cm and the heights are converted as for all-cm heights.

If all of the widths are in cm and some but not all of the heights are in cm, then convert the widths as for all-cm widths and convert the heights as for some but not all heights in cm.

A layout with margins is converted to a layout without margins—extra rows and columns of the appropriate heights and widths are added to the layout matrix and the layout respect is augmented for the margin respect—then the regions are calculated as above.

### 4.3.5 Specifying Units

Some slots in SPG objects contain a `nu` structure (number-with-unit) rather than a normal Lisp value. A `nu` structure consists of a number plus a unit (nil, :cm, or :nnc).

Rather than force the user to create a `nu` structure when setting the value of a slot, for example,

```
(edit-slot my-object 'my-slot
  (make-nu :value a-value :unit :cm))
```

the `edit-slot` function automatically detects whether the slot contains a `nu` structure and interprets the new value as a `nu` structure rather than a normal value. The following code illustrates the allowable syntax:

```
;; set the value of a normal slot
(edit-slot my-object 'my-normal-slot a-value)
;; set the value of a nu-slot
(edit-slot my-object 'my-nu-slot '(a-value :cm))
;; set the value of a nu-slot
;; (unit implicitly set to NIL)
(edit-slot my-object 'my-nu-slot a-value)
```

---

Figure 4.9: The allocation of rows and columns for the layout described by expression 4.4.
SUMMARY SPG is implemented in Common Lisp using CLOS. SPG allows access to the components of a plot with the syntax of the get-slot and set-slot commands. Relationships between graphical objects are specified using pre-update and post-update functions. The layout mechanism is based upon a matrix of names of graphical objects. Units are conveniently incorporated with the syntax of the edit-slot command.
4.4 SPG Reference

This section describes all of the editable slots of the SPG objects that the user should interact with, and describes the functions available to the user.

4.4.1 Objects

**X11-Window**

**Postscript-File**

 filament - string

**Picture**

 x
 y
 width
 height
 direction

**Line**

 x
 y
 width
 height

**Rectangle**

 x
 y
 width
 height

**Text**

 string
 x
 y
 height
 angle
 h-just
 v-just

**X-Axis**

 min - location of left end of axis within axis parent
 max - location of right end of axis within axis parent
 tick-length
 x-scale

**Y-Axis**

 min - location of bottom end of axis within axis parent
 max - location of top end of axis within axis parent
 tick-length
 y-scale

**Points**

 x-vble
 y-vble
 symbol - can only be :square
 symbol-size
 x-scale
 y-scale

**Plot**

 x-vble
 y-vble
CHAPTER 4. SPG

4.4.2 Functions

add-picture(\textit{picture output})
Add the picture to the output.

edit-slot(\textit{object slots value})
Set the value of the specified slot in the object, propagate any required changes, and perform any redrawing that is necessary.

get-slot(\textit{object slots})
Get the value of the specified slot in the object.

line(\textit{parent})
\hspace{1em} \&key x y width height
Create a line primitive and add it to the parent.

make-layout(\textit{nrows ncols names})
\hspace{1em} \&key heights widths respect margins m-respect
Create a layout structure. \textit{nrows} and \textit{ncols} should be integers, \textit{names} should be a matrix, \textit{row-heights} and \textit{col-widths} should be lists of numbers, \textit{respect} should be logical or a matrix, \textit{margins} should be a single number or a vector of numbers, and \textit{m-margin-respect} should be logical or a \(2 \times 2\) matrix.

picture(output)
Create a picture and add it to the output.

plot(\textit{x y picture})
\hspace{1em} \&key x-axis-p y-axis-p
Create a scatter-plot, with points, an x-axis (unless x-axis-p is false), and a y-axis (unless y-axis-p is false), and add it to the picture.

points(\textit{x y parent})
\hspace{1em} \&key symbol size name
Create a set of points and add them to the parent.

postscript(\textit{filename})
\hspace{1em} \&key layout
Create a postscript file.

R(\textit{message})
\hspace{1em} \&key evaluate verbatim parse
Send a (string) message to the slave \textit{R} process. If evaluate is false then no value is returned. If verbatim is true then the value is returned as an \textit{R} value, otherwise the value is returned as a Lisp list. If parse is true (the default) then the message is parsed before being evaluated.

rect(\textit{parent})
\hspace{1em} \&key x y width height
Create a rectangle primitive and add it to the parent.

\texttt{remove-picture}(*\textit{picture output}*)
\begin{itemize}
\item Remove the picture from the output.
\end{itemize}

\texttt{start()}
\begin{itemize}
\item Start a slave $R$ process for performing statistical analyses.
\end{itemize}

\texttt{text( \textit{string parent} &\textit{x y height h-just v-just angle})}
\begin{itemize}
\item Create a text primitive and add it to the parent.
\end{itemize}

\texttt{x-axis( \textit{parent} &\textit{min max name})}
\begin{itemize}
\item Create an x-axis and add it to the parent, in a slot with the specified name.
\end{itemize}

\texttt{y-axis( \textit{parent} &\textit{min max name})}
\begin{itemize}
\item Create a y-axis and add it to the parent, in a slot with the specified name.
4.5 User's Guide

4.5.1 Getting Started
Type CMUCL at the system prompt then type (load "spg-init.lisp") at the Lisp prompt (all other typing is at the Lisp prompt).

4.5.2 Creating a Plot
First, create a window to view the plot by typing:

(setf xw (x11-window :layout (make-layout 1 1 '((p)))))

This automatically creates a picture called p (since a picture of that name did not already exist) which occupies the entire window. Next, generate the data to plot:

(setf x (make-variable :data (list 1 2 3)))
(setf y (make-variable :data (list 2 4 6)))

Finally, create a plot within the picture p:

(setf pl (plot x y p))

4.5.3 Annotating a Plot
Text can be added to the plot using the text function. The location and size of the text can be specified in whatever units are most convenient. For example, the following adds a piece of text with a fixed height at a data location:

(text "point #2" (get-slot pl 'points)
  :x 2 :y 4 :height '(.3 :cm))

By specifying no units for the :x and :y values, these values are implicitly interpreted as being in the native coordinate system of the parent (i.e., :x 2 is the same as :x '(2 nil)). By adding the text to the set of points (rather than, for example, to the picture), this native coordinate system is the data scale used to position the points.

In contrast the following code adds a title to the top of the plot:

(text "My Plot" pl
  :x '(.5 :_npc) :y '(.9 :npc)
  :height '(.5 :cm) :h-just :centre)

4.5.4 Layouts
A layout primarily consists of a matrix (m rows and n columns) of plot names. For example, the Lisp expression,

(make-layout 2 2 '((p1 p2)
  (p3 p4)))

creates the layout matrix,

\[
\begin{bmatrix}
  p1 & p2 \\
  p3 & p4
\end{bmatrix}
\]
Figure 4.10: The regions allocated to two plots on a page (dashed rectangle), in both portrait and landscape orientations, based on the layout created by expression (4.5).

Each plot named in the layout matrix is allocated a region on the page using the following algorithm. The page is divided into \( m \) rows and \( n \) columns and each plot occupies the same cells on the page as it does in the layout matrix. In the above example, the page is divided into two rows and two columns (thereby creating four equal-sized cells) and each plot occupies one of these cells. In particular, plot \( p_1 \) occupies the top-left cell, plot \( p_2 \) occupies the top-right cell, and so on.

If a plot name appears in more than one cell of the layout matrix, the plot is allocated more than one cell on the page. For example, consider the layout created by,

\[
\text{make-layout} \ 2 \ 2 \ ((p_1 \ p_1) \\
(0 \ p_2))
\]

which has the layout matrix,

\[
\begin{bmatrix}
p_1 & p_1 \\
0 & p_2
\end{bmatrix}
\]

The page is again divided into two rows and two columns, but the plot \( p_1 \) is allocated the entire top row and the plot \( p_2 \) is allocated only the bottom-right cell. Figure 4.10 shows the regions that the two plots would be allocated on a printed page in portrait and landscape orientations.

The aspect ratio of a layout is the number of rows in the layout matrix \( (m) \) divided by the number of columns in the layout matrix \( (n) \). In the example, the layout aspect ratio is one (i.e., the layout matrix is square). The aspect ratio of a page is the physical height of the page divided by the physical width of the page (e.g., the aspect ratio of an A4 page with a portrait orientation is \( 297 \text{mm} / 210 \text{mm} = 1.414 \)).

When the aspect ratio of the page differs from the aspect ratio of the layout, the height of one row on the page will not be equal to the width of one column on the page.

The aspect ratio of the layout is ignored by default, however, it is possible to specify that the aspect ratio of the layout should be respected. This means that
the height of one row on the page will be the same as the width of one column on the page, no matter what aspect ratio the page has. The effect of this is that only the largest region on the page with the appropriate aspect ratio is used for arranging the plots. For example, the following Lisp code creates the same layout matrix as the previous example, but specifies that the aspect ratio of the layout is to be respected:

\[
\text{Figure 4.11: The regions allocated to two plots on a page (dashed rectangle), in both portrait and landscape orientations, based on the layout created by expression (4.6).}
\]

\[
\text{(make-layout 2 2 '((p1 p1) (0 p2))) :respect T)}
\] (4.6)

Figure 4.11 shows the regions that would be allocated to each plot on a printed page. Now only the largest square region (i.e., the largest region with an aspect ratio of one) on the page is actually used for the plots, \( p2 \) is allocated a square region, and \( p1 \) is allocated a region which is twice as wide as it is high.

It is also possible to specify the heights of the rows and the widths of the columns of the layout matrix. For example,

\[
\text{(make-layout 2 2 '((p1 p1) (0 p2))) :row-heights '(2 1) :col-widths '(2 2) :respect T)}
\] (4.7)

produces the following layout matrix and height and width vectors,

\[
\begin{bmatrix}
2 & p1 \\
1 & p1 \\
0 & p2
\end{bmatrix}
\]

The aspect ratio of a layout is actually the sum of the row heights divided by the sum of the column widths (which is the number of rows divided by the number
Figure 4.12: The regions allocated to two plots on a page (dashed rectangle), in both portrait and landscape orientations, based on the layout created by expression (4.7).

of columns when the row heights and column widths are all one; if heights or widths are not specified then they default to one). This means that it is possible to specify arbitrary aspect ratios simply by controlling the heights of rows and the widths of columns (and respecting the layout aspect ratio).

Figure 4.12 shows examples of the regions that each plot would be allocated on a printed page for the layout described above. The first point of interest is that only the largest region on the page with an aspect ratio of 0.75 (3/4) is used. Also, plot p1 is always twice as high as plot p2 and twice as wide. Finally, both plots are twice as wide as they are high.

The row-heights and column-widths can be specified in absolute units rather than relative units. The Lisp expression,

\[
\text{(make-layout 1 1 '((p))
:row-heights '((16 :cm))
:col-widths '((10 :cm)))}
\]

produces the following layout matrix and height and width vectors,

\[
\begin{bmatrix}
16\text{cm} \\
10\text{cm}
\end{bmatrix}
\]

Figure 4.13 demonstrates that the dimensions of the region allocated to the plot p by this layout is constant regardless of the page size and orientation.

A margin can be specified for the layout. A margin may be thought of as extra, empty rows and columns in the layout matrix. To avoid specifying these empty rows and columns by hand, which would be tedious and inefficient, a syntax is provided so that the rows and columns are built into the layout matrix automatically. For
example, the Lisp expression,

\[
\text{(make-layout 2 2 '( (p1 0) (0 p2)) :margins '(3 :cm))}
\]

produces the following layout matrix and height and width vectors,

\[
\begin{bmatrix}
3cm & 0 & 0 & 0 & 0 \\
1 & 0 & p1 & 0 & 0 \\
1 & 0 & 0 & p2 & 0 \\
3cm & 0 & 0 & 0 & 0
\end{bmatrix}
\]

\[
[3cm \ 1 \ 1 \ 3cm]
\]

Notice that the row heights and column widths have defaulted to a value of one.

Figure 4.14 shows examples of the regions allocated to plots \( p1 \) and \( p2 \) on different-sized pages. This layout demonstrates that absolute and relative units can be combined. In effect, the absolute margins are first allocated around the perimeter of the page and then the relative rows and columns are allocated using the remainder of the page. The regions for \( p1 \) and \( p2 \) are not square, even though each plot occupies exactly one row of height one and one column of width one, because the layout aspect ratio is not respected.

It is also possible to specify layouts for the arrangement of components within a plot. A useful feature of layouts within this context is the ability to specify partial respect of the layout matrix. This is achieved by specifying a matrix of 0's and 1's for the layout respect. If a 1 appears in cell \((i, j)\) of the respect matrix and the width of column \( j \) is \( x \) times the height of row \( i \), then the width allocated to column \( j \) by the layout will be \( x \) times the height allocated to row \( i \). This allows, for example, the user to specify that the width of column \( j \) be equal to the height
Figure 4.14: The regions allocated to two plots on a page (dashed rectangle), in both portrait and landscape orientations, based on the layout created by expression (4.9). The dotted rectangle indicates the extent of the layout margins.

of row i. An example should clarify things. The Lisp code,

\[
\text{\texttt{(make-layout 2 2 '\((y-axis\, data-points)\)}}
\begin{array}{l}
\text{\texttt{(0 x-axis))}} \\
\text{\texttt{:row-heights '\((4\, 1)\)}} \\
\text{\texttt{:col-widths '\((1\, 4)\)}} \\
\text{\texttt{:respect '\((0\, 0)\)}} \\
\text{\texttt{(1\, 0))}} \\
\text{\texttt{:margins .5}} \\
\text{\texttt{:margin-respect T)}}
\end{array}
\]

produces the following layout matrix and height and width vectors,

\[
\begin{bmatrix}
.5 & 0 & 0 & 0 & 0 \\
4 & 0 & y-axis & data-points & 0 \\
1 & 0 & 0 & x-axis & 0 \\
.5 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

which can be used to arrange the three components within a plot region.

Figure 4.15 shows the subregions that are allocated to each component within an example plot region. The height of the region allocated to the x-axis is the same as the width of the region allocated to the y-axis. The margin provides an area of blank space at the edges of the plot region. The margin aspect ratios are respected (:margin-respect T), which means that the relative heights of margin rows and the relative widths of the margin columns are respected. In this case the margin rows are both 0.5 high and the margin columns are both 0.5 wide so the margins are the same on all sides of the plot.
4.6 Results and Conclusions

4.6.1 Layouts

The layouts mechanism provides an intuitive and very powerful method for specifying the arrangement of graphs on an output.

The primary feature of the layouts mechanism is that it allows the user to specify a high-level description of an arrangement of plots, which is then automatically converted into a low-level specification of the locations and dimensions of the plots. The advantages of this are twofold. First of all, a low-level specification of a simple arrangement of plots is not difficult to calculate, but it is tedious and inconvenient. The specification of a matrix of plot names, possibly with row heights, column widths and margins is a convenient and intuitive alternative.

Secondly, a low-level specification of a complex arrangement of plots can be very difficult to calculate. The ability to combine relative and absolute units and the ability to respect the aspect ratio of only some rows and columns within a layout matrix provide a much simpler description of more complex plot arrangements.

It is significant that layouts can also be used within plots to arrange the components of the plots. This provides a great deal of flexibility for the layout of plotting elements. Most statistical graphics systems provide a very rigid plot structure and severely restrict how much the user is able to modify the layout of this structure. By providing layouts for the arrangement of plot components, SPG introduces a great deal more power for the user to specify the layout of the plot structure.

A disadvantage of layouts is that it is not convenient for specifying a repetitive layout such as a scatterplot matrix. For that sort of arrangement, it is much more efficient and simple to specify just the number of rows and columns (as in, for example, S-Plus and R).
4.6.2 Coordinate Systems

There are two ways in which SPG provides improved support for coordinate systems in statistical graphics. First of all, there is a selection of coordinate systems for locating and sizing all objects; there is an absolute coordinate system based on centimetres, a relative system based on normalised coordinates, and an arbitrary "native" system based on an x-scale and y-scale. This is in contrast to, for example, text which is drawn in the margins of an S graph, where there is only one coordinate system available.

Furthermore, the hierarchical structure of some plotting elements (e.g., the scatterplot) implicitly introduces further coordinate systems because the plotting element, which has a rectangular bounding region, describes a new normalised coordinate system (with (0, 0) at the bottom-left corner of the bounding region and (1, 1) at the top-right corner) and a new native coordinate system (via the x-scale and y-scale of the plotting element).

Secondly, there is increased flexibility: all graphical objects (primitives and plotting elements) can be located within any of the available coordinate systems (the absolute system of centimetres, the relative system of normalised parent coordinates, and the native system of the x-scale and y-scale).

There are no coordinate systems which correspond completely to those in the S-Plus system for locating text in margins (regions can be given different directions and arbitrary x- and y-scales, but there is no unit defined in terms of the height of a line of text), however, the framework could easily be extended to support any sort of unit for positioning and sizing all graphical objects.

**SUMMARY** SPG's layout mechanism provides benefits for specifying both simple and complex arrangements of plots. The layout mechanism also supports specifying the arrangements of components within a plot, which promotes flexible plot structures. SPG provides two absolute coordinate systems for specifying the location and dimensions of graphical objects. A third coordinate system is also available for locating and sizing components of a plot relative to the bounding region of the plot.

---

2On the other hand, SPG does not provide a coordinate system corresponding to S's margin coordinate systems (where either the x- or y-dimension is measured in lines of text), but this and any other coordinate system could be added relatively easily. The important point is that all of the existing coordinate systems are available all of the time.
4.7 References


Chapter 5

R Graphics

This chapter describes work involved with the development of the graphics engine of the R statistical programming environment (Ihaka and Gentleman, 1996). Some of this work represents an implementation of some of the ideas that were investigated in the experimental software that was described in the previous three chapters. R is being developed in the C programming language.

5.1 Mathematical Annotation

Providing mathematical annotation in plots can be important when those plots are to be used to summarise and present the results of a statistical analysis.

In most systems, mathematical annotations must be added via some sort of third-party software. For example, a chart from Systat may be combined with a mathematical expression from MathType in a Microsoft Word document. This sort of approach makes it very difficult to, for example, include mathematical annotation in an axis label or locate a mathematical annotation at a particular data location on a graph (e.g., when providing a label for a data symbol). Some systems provide rudimentary mathematical facilities within normal text annotation (e.g., SigmaPlot allows for superscripts and subscripts), but there are no sophisticated solutions available.

The approach described here uses a programmatic description of mathematical annotation and uses software to render the equation into the plot. This is the same approach that is taken in typesetting systems such as TeX.

This approach partitions the problem into two subproblems. The first is that of describing the structure of the mathematical annotation, and the second of rendering that description into graphics on the page.

5.1.1 Describing Mathematical Annotation

Computations in R are carried out by typing expressions to an interpreter which parses and executes them. The expressions accepted by the interpreter are specified in infix notation. For example, the expression \( f(a, b) \) indicates that the computation should apply the procedure \( f \) to the arguments \( a \) and \( b \). In addition, there is a certain amount of syntactic sugar which permits binary operators to be typed in the form \( a + b \), in addition to their infix form \( + (a, b) \).
CHAPTER 5. R GRAPHICS

The expressions used to specify computations in R are similar to traditional mathematical notation. Because of this, they can also be used to describe the structure of mathematical expressions to be used in graphical annotation. The following table shows a number of simple mathematical expressions and how they can be represented with R expressions.

<table>
<thead>
<tr>
<th>Mathematics</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x + y)/2)</td>
<td>((x + y)/2)</td>
</tr>
<tr>
<td>(\sin x)</td>
<td>(\sin(x))</td>
</tr>
<tr>
<td>(x \leq y)</td>
<td>(x \leq y)</td>
</tr>
<tr>
<td>(\sum x_i)</td>
<td>(\text{sum}(x[i]))</td>
</tr>
</tbody>
</table>

In order to use R expressions to describe the layout of mathematical expressions, it is necessary to have a means of indicating that the expression is not to be evaluated, but instead treated as a symbolic description. This can be achieved in R by using the *quoting* mechanism provided by the `expression()` function. The R expression

```r
expression(expr1, expr2, ...)
```

returns its unevaluated arguments `expr1`, `expr2`, ... in a *vector* of mode `expression`.

The R graphics code has been modified so that in any place where a vector of text annotation strings could be passed as an argument to a graphics function, it is also possible to pass a vector of expressions. When this happens, the expressions are processed by the underlying equation renderer rather than by the graphics code which draws text strings.

When the renderer receives an expression, it draws a representation of it on the graphics device using its built-in set of layout rules. Some of these rules specify special actions for mathematical operators such as summation and integration, and others the way in which accents and radicals should be handled. Additional rules provide simple text translations such as the translation of \(\alpha\) into the corresponding greek symbol \(\alpha\).

### 5.1.2 The Equation Renderer

The process of producing a mathematical expression on a printed page or a computer screen involves two steps. First of all, the expression must be described in some way that the computer software can recognise and, secondly, the software must convert that description into the appropriate drawing operations - what to draw and where to draw it. This section describes the second step, typesetting the mathematical expression.

In order to introduce the basic concepts behind the typesetting process, first consider what happens when a normal piece of text is typeset.

#### Typesetting basics

A piece of text consists of a number of characters. For example, the text "hello" consists of the characters "h", "e", "l", "l", and "o". The basic text-drawing operations automatically draw these characters in the correct position relative to
CHAPTER 5. R GRAPHICS

Figure 5.1: A diagram of the bounding box of a character.

![Figure 5.1: A diagram of the bounding box of a character.](image)

Figure 5.2: Typesetting for the text “hey there”.

![Figure 5.2: Typesetting for the text “hey there”.](image)

each other. For example, the bases of the characters are vertically aligned, and the "e" is drawn immediately to the right of the "h", and so on.

The information necessary for positioning the characters is contained in each character’s bounding box. The bounding box for a character roughly corresponds to a rectangle which circumscribes the region in which the character will be drawn and is described by a width, height, depth, and baseline (see Figure 5.1). The bounding-box information is dependent upon the text’s font (e.g., Times-Roman 10pt, Times-Italic 12pt).

It is not necessary that all of the character fits inside the bounding box (e.g., slanted or italic characters often “stick out” the sides of their bounding boxes), but the dimensions of the box are designed so that characters may be positioned correctly relative to each other simply by aligning their bounding boxes. For example, Figure 5.2 shows how the text "hey there" is typeset by placing the bounding boxes of the characters side by side and vertically aligning the baselines of the boxes.

Typesetting mathematical expressions

There are two differences between typesetting a mathematical expression and typesetting a piece of text: first of all, a mathematical expression can be made up of a greater variety of elements (not just characters) and, secondly, the rules for combining the elements of a mathematical expression vary depending on the type of the expression.

Although there are considerably more rules for typesetting mathematical expressions, the rules are still based upon the positioning of the bounding boxes for the elements of the expression. For example, the rules for typesetting a fraction (e.g., \( \frac{1}{2} \)) require placing the baseline of the numerator a sufficient distance above the horizontal line, placing the baseline of the denominator a sufficient distance
below the horizontal line and aligning the bounding boxes of the numerator and denominator so that they are centred on the middle of the horizontal line. The rules used in this implementation were loosely based upon the rules described in The \TeX\book (Knuth, 1984); the rules are less general in \proglang{R} because \proglang{R} expressions contain much more structure than the stream of tokens that \TeX\ rules are designed for.

Much of the variety in the elements of a mathematical expression arises from the fact that the operands in a mathematical expression can themselves be mathematical expressions. This means that a mathematical expression must be able to be considered as a typesetting unit itself. In particular, it must be possible to generate a bounding box for a mathematical expression. This is achieved simply by generating a box which completely encapsulates the typeset expression. For example, Figure 5.3 shows how the expression over(x[i], y + 2) is typeset; the numerator and denominator are themselves mathematical expressions with their own bounding boxes.

**Parsing mathematical expressions**

The previous section describes how bounding boxes are combined in the typesetting process. This section describes how bounding boxes are obtained from the original mathematical expression.

Expressions in \proglang{R} are internally represented as lists of the form:

```
(operator operand operand ...)
```

For example, consider again the expression over(x[i], y + 2). This is stored internally as:

```
(over ([ x i] (+ y 2))
```

The operands are either further lists (expressions) or *atoms*. For example, ([ x i]) is another list, but x and i are atoms.

Atoms can be converted into an individual characters, which have bounding boxes. For example, x converts to the character "x" and 2 converts to the character "2". Some special atoms are defined to allow for special sorts of characters. For example, alpha converts to the character "α". So the bounding box for an atom is just the bounding box for the character that corresponds to that atom.

Lists are a bit more complicated. Consider first just a list which has atoms for its operands; for example, ([ x i]) or (+ y 2). These lists can be converted into a set of characters; the characters from the atoms plus possibly a character for the operator as well. For example, ([ x i]) converts to just "x" and "i", but (+ y 2)
converts to "y", "+", and "2". The list of characters can be converted into a list
of bounding boxes which are combined in the typesetting process according to the
rules specified for the particular operator.

When an operand is itself a list, the operation of generating bounding boxes,
parsing the expression, and arranging bounding boxes, typesetting the expression,
become intertwined. Consider the list (over ([ x i ] (+ y 2))) in which both
operands are lists. In order to generate a bounding box for the operand ([ x i ]),
the operand must be typeset (i.e., the list must be converted into bounding boxes
and those bounding boxes must be arranged appropriately). Once the operand has
been typeset, a bounding box for the operand can be generated by drawing a box
around the typeset operand.

The complete process for this example looks something like:

```
    generate bounding box for "x"
    generate bounding box for "i"
    typeset ([ x i ]) i.e., arrange bounding boxes
    generate bounding box for ([ x i ]) i.e., draw box around outside
    generate bounding box for "y"
    generate bounding box for "+"
    generate bounding box for "2"
    typeset (+ y 2) i.e., arrange bounding boxes
    generate bounding box for (+ y 2) i.e., draw box around outside
    typeset (over ([ x i ] (+ y 2)) i.e., arrange bounding boxes
```

Controlling fonts

The bounding box for a character depends upon the character's font. It is possible
to explicitly specify the font style in mathematical expressions (e.g., \texttt{bold(x)}), but
the renderer also does some setting of fonts automatically. In particular:

- The font is automatically set to \texttt{symbol} for certain characters. This is used to
  obtain greek characters (e.g., \( \alpha \)) and mathematical operators (e.g., 
  \(+\), \( \sum \), \( =\),
  \( ...\)).

- The font style is automatically set to \texttt{italic} for atoms (unless the font has
  been set to \texttt{symbol}). This is because most normal characters in mathematical
  expressions are variables, which are traditionally typeset in italics (e.g., \( y = x + 2 \)).

- The font size is automatically reduced for superscripts, subscripts, and fractions.
  This just follows the standard typesetting conventions (e.g., \( x_i \), \( \frac{1}{2} \), ...
  ). This effect is cumulative so that subscripts are smaller than subscripts and
  so on down to the smallest available font.

Device-specific typesetting

If a device is to be used to provide mathematical annotation it must possess a
minimal level of functionality. Most importantly, it should provide access to a
special symbol font and also have accurate font metric information available (i.e.,
information about the bounding boxes of the characters in the font). Ideally, the font
information should include width, height, and depth measures for each individual
character.
5.1.3 Capabilities

The mathematical operators currently supported (for mathematical typesetting) by R are shown below.

<table>
<thead>
<tr>
<th>type</th>
<th>operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>+, -, /, *</td>
</tr>
<tr>
<td>unary</td>
<td>+, -</td>
</tr>
<tr>
<td>super/subscripts</td>
<td>^, [</td>
</tr>
<tr>
<td>accents</td>
<td>hat, bar</td>
</tr>
<tr>
<td>fractions</td>
<td>over, frac</td>
</tr>
<tr>
<td>relations</td>
<td>==</td>
</tr>
<tr>
<td>grouping (visible)</td>
<td>(</td>
</tr>
<tr>
<td>grouping (invisible)</td>
<td>{</td>
</tr>
<tr>
<td>big operators</td>
<td>sum, product, integral</td>
</tr>
<tr>
<td>radicals</td>
<td>sqrt, root</td>
</tr>
<tr>
<td>absolute value</td>
<td>|, abs</td>
</tr>
<tr>
<td>juxtaposition</td>
<td>paste</td>
</tr>
<tr>
<td>typeface</td>
<td>bold, italic, plain, bolditalic</td>
</tr>
</tbody>
</table>

5.1.4 Examples

Consider a graph of the Normal probability density function (see Figure 5.4). It is obviously useful to be able to generate text annotations within a graph that use mathematical formulae, but this graph shows that mathematical annotation on the tick-mark labels can also be extremely useful.

The R code used to produce this graph is as follows:

```r
plot(-300:300/100, dnorm(-300:300/100),
     type="l", xlab="", ylab="", axes=F)
axis(2)
axis(1, at=c(-3,-2,-1,0,1,2,3),
     labels=expression(-3*sigma,-2*sigma,-sigma,0,
                        sigma,2*sigma,3*sigma))
text(-3,.25, expression(paste(frac(1,sigma*sqrt(2*pi)),
                           \[, plain(e)^{-frac(-(x-mu)^2,
                           2*sigma^2)}])))
```

Figure 5.5 shows the usefulness of mathematical annotation in graph titles and axis labels. The code to produce this graph is as follows:

```r
par(mar=c(5.1,5.1,4.1,2.1))
x <- seq(0,0.5,length=501)
y <- rchisq(501,2)
plot(x, log(y), type="l",
     xlab=expression(paste("Angular Frequency ",lambda)),
     ylab=expression(plain(log)[10]*
                     "^\{I[XX]^{-1}(T)\}(lambda)),
     main=expression(plain(Log)[10]*
                     "^\{"Periodogram\}"))
```
Figure 5.4: A plot of the Normal probability density function.

Figure 5.5: A periodogram with mathematical annotation in the title and in the axis labels.
5.1.5 Conclusions

Using R language expressions to describe mathematical expressions together with a mathematical expression renderer which can use such expressions as a description of what is to appear in a plot, provides a simple way to produce mathematical annotations in graphs. While the capabilities of the renderer are limited (compared to a system such as \TeX) it provides most of the capabilities commonly required when producing statistical graphics.

Versions of R containing the capabilities described herein can be obtained from CRAN sites\(^1\) and in particular from STATLIB.

---

\(^1\)The Comprehensive R Archive Network maintained by Kurt Hornik and Friedrich Leisch of the Technical University of Wien in Austria.
5.2 Rewriting the \( R \) Graphics Engine

The rewrite of the \( R \) graphics engine was motivated by the desire for a number of additional features. These were: multiple devices (e.g., more than one window open on the screen at one time), the detection of user events (such as clicking the mouse button and resizing windows), automatic redrawing of window devices when they are resized, and extra layout facilities for positioning graphs on a device.

5.2.1 The old graphics engine

This section describes the state of the \( R \) graphics engine prior to the rewrite.

Only one device

There was only one device (or zero devices) in existence at one time. It was an error to perform a graphical operation if there was no device active, but otherwise all graphical operations occurred within the unique active device.

There were a large number of graphical parameters stored for each active device. These can be divided into three groups:

- **basic device-driver properties** provided a basic description of the device (e.g., dimensions, whether the device provided automatic clipping, ...). Most of these parameters were set when the device was created and did not change (the exceptions were the dimensions of a window which could be resized).

- **general parameters** provided a description of the current graphical state of the device (e.g., line type, font, text justification, ...). These parameters were accessed and modified via the `par` command. Many of these parameters could also be set as additional arguments to graphics commands (e.g., `plot`, `text`, ...).

- **mapping parameters** described the transformations between different coordinate systems on the device (e.g., inches, normalised device coordinates, data or plot coordinates, ...). These were only intended for internal use.

The graphics parameters were stored in two global structures, \( \text{GP} \) and \( \text{DP} \). \( \text{GP} \) represented the current or temporary parameter settings and \( \text{DP} \) represented the default or permanent parameter settings.

The general graphics parameters could be accessed and modified via the `par` command (e.g., `par("lty")` returned the current line type and `par(col="red")` set the default colour to be red). Some of the general parameters could also be set as additional arguments to graphics commands (e.g., `plot(1:10, pch=".")`) set the plotting symbol to be a decimal point for that plot and `text(1, 1, "hi", adj=0)` set the justification of text to be left-justified for that piece of text). A few of the basic device-driver properties could be accessed via `par` (e.g., `par("din")` returned the dimensions of the device in inches), but the user could not modify any of these properties. The mapping parameters were not accessible to the user.

Setting a parameter with `par` set the parameter in both \( \text{GP} \) and \( \text{DP} \), but setting a parameter via a graphical command (e.g., `plot`, `text`, ...) only set the parameter in \( \text{GP} \) and only for the duration of that command.

The \( R \) graphics system was static. This means that, once something was drawn on a device, it could not be modified. It was possible to add further graphical
elements to a graph, but in order to modify an existing element, the entire graph had to be redrawn. For example, suppose that a plot was created with the command

\[
x \leftarrow \text{rnorm}(10)\\
\text{plot}(x)
\]

This created a scatterplot with points and axes. It was possible to add, for example, text annotation to the plot, or even additional axes:

\[
\text{text}(5.5, 0, "\text{Some random points"})\\
\text{axis}(4)
\]

However, it was not possible to change, for example, the label on the x-axis or to modify the margins around the plot (to accommodate the extra axis) without generating the entire graph again. For example,

\[
\text{par(mar=c(5.1, 5.1, 4.1, 3.1))}\\
\text{plot}(x, xlab="\text{Observation Number"})\\
\text{text}(5.5, 0, "\text{Some random points"})\\
\text{axis}(4)
\]

With such a system, there was a need for a function (or functions) to indicate that it was time to start a new graph rather than just add to the current graph. In R, a new graph was started when the user entered one of the “plot” commands: barplot, boxplot, contour, coplot, dotplot, hist, piechart, plot, or ts.

Another important graphical event was the erasure of the device surface. This happened when the user entered one of the plot commands and there was no room allocated for the new plot on the device or when the user specified a new layout for the device. The first situation depended on the current layout of plots on the device. For example, if the current layout specified a \(2 \times 2\) matrix of plots (i.e., a total of four plots) the fifth plot would clear the device (and start a new plot). The user specified a new layout by entering one of the “layout” commands: \text{par(mfrow=}\ldots\), \text{par(mfcol=}\ldots\), or \text{par(mfgr=}\ldots\).

When a new graph was started, the parameter values in GP were reset to the default values in DP (hence the “temporary” nature of values in GP).

All graphical operations used the parameter values in GP (hence the “current” nature of the values in GP).

When a new device was created, the old device was destroyed and the graphics parameter settings in GP and DP were initialised for the new device, completely overwriting the settings from the previous device.

**Limited event-handling**

\(^2\)The old system was command-line driven. This meant that nothing much happened until the user entered a command at the command-line. For example, if the user resized a window, R would take no action until the next user command was entered (at which point the change in window size was detected and the new window size taken into account in subsequent graphics operations).

When R was ready to accept a new command, it entered a loop which would only return once a complete command had been entered. Once a complete command had been obtained, R would check for other user events (e.g., a window resize) and

\(^2\)The details described here are applicable to the UNIX version of R, but there may be some differences in Microsoft Windows and Macintosh versions
handle these before looking for the next command. In a system with a command-line interface, the system typically spends most of the time waiting for the user to enter a command. This meant that R spent most of its time in a state where it could not handle user events.

The exceptions to this situation were the `locate()` and `identify(...)` commands. These commands monitored mouse clicks in a window. `locate()` returned the location of the clicks in data or plot coordinates and `identify(...)` labeled data points (when the clicks are sufficiently close to data points) and returned the indices of the labeled data points. Both commands entered a local event-handling loop in order to immediately respond to mouse clicks within a window. Although these commands did provide immediate responses to user events, only mouse clicks were handled and the command-line was disabled until the user had finished locating or identifying points.

**Plot layout facilities**

The arrangement of plots on a device was controlled by a number of layout parameters (these are a subset of the general graphical parameters).

The device surface was divided into the outer margins and an inner region (which was the device surface minus the outer margins). The outer margins were controlled by the `oma`, `omd`, and `omi` parameters.

The inner region contained one or more figures. Specifying `par(mfrow=c(nr, nc))`, `par(mfcol=c(nr, nc))`, or `par(mfg=c(i, j, nr, nc))` created a matrix of \( nr \times nc \) equally-sized figures in \( nr \) rows and \( nc \) columns. Specifying `par(fig=c(left, right, bottom, top))` created a single figure within the inner region where `left`, `right`, etc... were proportions of the inner region. For example, Figure 5.6 shows the six figure regions resulting from the following layout command:

```
par(oma=c(4, 4, 8, 4), mfrow=c(3,2))
```

Specifying `par(fin=c(width, height))` created a single figure centered within the inner region, where `width` and `height` were in inches. For example, Figure 5.7
Figure 5.7: A figure layout produced by `par(oma=c(4, 4, 8, 4), fin=c(4,4)).` shows the single figure region resulting from the following layout command (the dashed line indicates the inner region):

```
par(oma=c(4, 4, 8, 4), fin=c(4,4))
```

The figure region was divided into figure margins and a plot region. The figure margins were controlled by the `mar` and `mai` parameters. The plot region was the figure region minus the figure margins unless the user specified a plot region using the `plt` parameter or the `pin` parameter. For example, Figure 5.8 shows the plot region (the inner dashed rectangle) resulting from the following command (the outer margin and figure region settings are the same as for Figure 5.7):

```
par(mar=c(3, 3, 5, 1))
```

Figure 5.9 shows the result of an explicit specification of the plot region as follows:

```
par(pin=c(2, 2))
```

This layout mechanism was modeled on the S-Plus layout facility (see Figure 4.2).

The approach was good for producing simple arrangements of plots (arrange-
ments that involved multiple plots of the same size). It was also possible to produce arbitrary arrangements of plots by specifying the `fig` or `fin` parameters, to position a plot anywhere within the inner region, and fooling R into thinking that a plot had not yet been drawn by specifying `par(new=T)` before each new plot that was to appear in the same window.

The different layout parameters could be specified in a number of different units, but they were always stored in GP and DP in a fixed unit. In particular:

- the outer margins could be specified in terms of the height of a line of text (e.g., `par(oma=c(0, 0, 3, 0)))`, as a proportion of the device surface (e.g., `par(omd=c(0, .2, .2, 0)))`, or in inches (e.g., `par(omi=rep(1, 4)))`, but the outer margins were always stored in terms of lines of text.
Figure 5.8: The plot region produced by `par(mar=c(3, 3, 5, 1))`.

Figure 5.9: The plot region produced by `par(pin=c(2, 2))`. 
• the figure region could be specified in terms of a proportion of the inner region (the device surface minus the outer margins; e.g., `par(fig=c(.5, 1, .5, 1))`) or in inches (e.g., `par(fin=c(4,4)))`, but the figure region was always stored as a proportion of the inner region

• the figure margins could be specified in terms of the height of a line of text (e.g., `par(mar=c(5.1, 5.1, 4.1, 2.1)))` or in inches (e.g., `par(mai=c(1, 1, .5, .2)))`, but the figure margins were always stored in terms of lines of text

• the plot region could be specified as a proportion of the figure region (e.g., `par(plt=c(.25, .75, .25, .75)))` or in inches (e.g., `par(pin=c(2, 3)))`, but the plot region was always stored as a proportion of the figure region.

### 5.2.2 Motivations for the new graphics engine

#### Multiple devices

The advantages of having multiple devices are: it is possible to create a **PostScript** version of a picture in an x11 window without having to destroy the x11 window (so that, for example, the user can continue modifying the picture); it is possible to compare different plots without having to use small multiples.

#### Improved event-handling

The advantage of moving to a more event-driven system is that user events such as resizing a window are detected and acted upon immediately (rather than having to wait for the next command to be entered at the command-line).

Also, by making the system event-driven, it becomes possible to develop interactive graphical features such as linked brushing and spinning plots.

#### Display lists

In the old system, if a device was resized, the contents of the device were lost. The command or commands that produced the contents had to be re-entered in order to restore the contents.

A display list is essentially a record of the most recent graphics operations that have occurred on a device. Display lists are useful for redrawing the contents of a device if, for example, the device is resized. Display lists are also useful for copying the contents of one device to another device (e.g., in order to make a **PostScript** version of a picture in an x11 window).

#### Layout facilities

A number of high-level graphics commands in R are needed to perform quite difficult contortions in order to produce a desirable plot layout (notably the `coplot` command where a matrix of plots is flanked by one or two marginal plots of quite different size). A more powerful layout facility would simplify these commands and provide a useful tool for creating new commands.

Also, given that a device can be resized, it can be useful to record the units in which a layout parameter was set. For example, suppose that the figure region is specified in inches, as in `par(fin)`. Under the old system, this was stored as a proportion of the inner region (the total device surface less the outer margins) so
CHAPTER 5.  \textit{R} GRAPHICS

that, if the device is resized, the figure would no longer have the dimensions that were specified by the user. It would be preferable if the layout parameters retained the units that they were specified in.

5.2.3 The new graphics engine

Multiple devices

There can be multiple devices in existence at once (up to 64 devices). Each device is represented by a device descriptor (DD). A DD consists of two structures containing the graphics parameters (i.e., instead of global GP and DP structures, each device has its own GP and DP), a third structure for graphics parameters which is a saved copy of the DP structure for use with the display list, a structure containing device-specific information (e.g., the x11 resources associated with an x11 window), and a display list (see below).

One of the devices is denoted the \textit{active} device (all other devices are \textit{inactive}). Graphical operations always occur in the active device. All devices are associated with a device number and functions are provided to switch between devices:

\texttt{dev.list} returns the numbers and names (x11 or PostScript) of all of the existing devices

\texttt{dev.next, dev.prev} return the \textit{number} of the next (previous) device in the list of devices (the next device is found by looking for a device with the lowest device-number higher than the device-number of the active device; if no such device exists then the device with the lowest device-number is returned—which will be the active device if only one device is active)

\texttt{dev.set} makes the device with the specified device-number the active device

\texttt{dev.off} closes the device with the specified device-number (or the active device if no device-number is specified)

The GP and DP within each device play the same role as in the old system (i.e., \texttt{par} sets the DP values, \texttt{plot} and others set the GP values, and the GP values are reset from the DP values for each new plot).

All graphics operations are now given a DD (typically that of the active device) and use the values of the graphics parameters in the GP of that DD.

Handling user events

The code which handles user activity now monitors the command-line and other user events (such as window resizes) simultaneously. This means that, for example, if the user resizes a window, \texttt{R} will immediately respond by redrawing the window contents (see below).

The entry of commands is monitored at the same time as other user events through the \texttt{select} function. This function is used to wait until \textit{either} there is activity on the command-line \textit{or} there is some other user event. If the activity was on the command-line, the current command is updated (if the command is complete then it is executed). If there was a user event then that is handled (e.g., when a window is resized, the contents of the window are redrawn).
CHAPTER 5. R GRAPHICS

Display lists

A display list is not usually needed for a static device such as a PostScript device. The following discussion only concerns dynamic devices such as X11 windows.

A display list is a recording of the recent graphical activity on a device. There are two major design issues: what constitutes "recent" (i.e., when should the display list be reinitialised?) and at what level should graphics operations be recorded—user commands (e.g., plot and hist) or primitive graphical operations (e.g., GMoveTo and GLineTo).

A natural place to initialise the display list is when the device is cleared; in the R system this occurs when a plot command is issued, but there is no room for the plot. The number of plots that can appear on a device depends upon the current plot layout for the device, which is set by par(mfrow=...), par(mfcol=...), par(mfg=...), or layout. For example, if the user specifies par(mfrow=c(1,1)) then the device is cleared for every new plot because there is only room for one plot at a time on the device.

There are (at least) three different levels of graphics operations in the R system. There are the primitive graphics operations provided by the graphics engine (e.g., GMoveTo, GLineTo, and GText; these are implemented in C and are used internally by R); there are the basic R graphics commands (e.g., plot.new, plot.window, and plot.xy; these are implemented in C and are used in higher level R graphics commands, such as plot and hist, and by users writing their own R functions), and there are high-level R graphics commands (e.g., plot, hist, and piechart; these are implemented in interpreted R code—they are essentially predefined R functions).

The problem with recording primitive graphics operations is that all knowledge of the relationships between the different graphical primitives is lost (e.g., the line and the piece of text on a tick-mark become completely independent entities). This makes it difficult to maintain the relationships between the graphical primitives when, for example, a device is resized and the graphical primitives have to be relocated and/or resized. A simple option is just to scale every graphical primitive so that it retains its relative location and size on the device, but this is by no means guaranteed to generate a desirable result. For example, the tick-marks on different axes may become very different in length, text may distort, and so on. This problem is solved by recording higher-level graphics operations because these operations embody the relationships between the graphical primitives that they generate. For example, the axis command ensures the correct relative locations of the lines and text on an axis.

There are two problems associated with recording high-level R commands; first of all, this would require creating R commands to start and stop recording. This would be dangerous because it would mean that the user could take control of when recording of the display list will start and stop. This would greatly increase the chances of quite nasty situations arising (e.g., the user starts recording, performs some drawing operations, resizes a window, then performs more drawing before stopping recording). If the recording of graphics operations is kept internally in the C code, then the task becomes much simpler; the implementation of display lists can be controlled and fewer situations need to be considered.

The second problem with recording high-level R graphics commands is that it becomes possible to have nested recording (i.e., an R graphics command begins recording then calls another R graphics command which also tries to begin recording). Again, this adds to the complexity of the recording mechanism.
Both of these problems are solved by recording only basic \( R \) graphics commands; the recording occurs within the C implementation of these commands so is invisible to the user. There is no possibility of nested recording because none of the basic \( R \) graphics commands call any of the other basic \( R \) graphics commands. Unfortunately, this option has problems of its own. Just as recording primitive graphics operations loses information that is implicit in \( R \) graphics commands regarding the relationships between different graphical primitives, recording basic \( R \) graphics commands loses information that is implicit in higher-level \( R \) graphics commands. For example, suppose that a high-level command queries the current size of a device and uses that information as a parameter to a basic command (e.g., to determine where to position plots). By recording only the basic command, the querying of the device size is lost and the size becomes a fixed parameter in the recorded command. If the device is resized this information becomes outdated, with respect to its original intention. Despite this drawback, recording currently occurs at the level of basic \( R \) graphics commands.

The display list is part of the DD for a device. It consists of a list of basic \( R \) graphics commands—the command itself, plus a copy of the evaluated arguments given to the command. The fact that the arguments have been evaluated means that the display list records graphics commands in their original context. For example, suppose the user types the following:

\[
x \leftarrow 1:10 \\
\text{plot}(x) \\
x \leftarrow \text{rnorm}(10)
\]

If the user now resizes the window, the plot will be redrawn with the original value of \( x \) (i.e., \( 1:10 \)).

Whenever a device is cleared, or when it is first created, the display list is erased and the current state of the device’s DP is saved. The latter action captures the current graphical state of the device just prior to the point when the recording of graphical operations on the device was started.

All basic \( R \) graphical operations are appended to the display list. This builds up a record of all graphical operations on the device since the last time the device was cleared.

When the display list is replayed (e.g., when a device is resized), the saved DP values are restored (this restores the graphical state of the device to what it was just before the display list began recording) and the commands in the display list are performed.

A command is recorded as the last action of the C implementation of the basic \( R \) graphics command in order to avoid recording commands that will generate errors. Commands that fail—i.e., generate an error—are not recorded.

Functions are provided to allow copying of display lists between devices and, because the maintenance of a display list is fairly expensive in terms of memory, a function is provided to turn the display list feature off.

\texttt{dev.copy(device, ..., which=dev.next())} copies the display list from the active device to either: the device specified by the \texttt{which} parameter (a device-number, which defaults to \texttt{dev.next()}); or to the device specified by the device parameter (which should be the name of a device function; e.g., \texttt{x11}, or \texttt{postscript}). In the latter case a new device is created.
dev.print(device = postscript) copies the display list from the current device to the device specified by the device parameter (which should be the name of a device function and defaults to postscript)

dev.control("inhibit") turns off the display list recording in the active device

Layouts

The layout facilities of the old system were retained. In particular, the par(mfrow=...) mechanism is very efficient for simple arrangements of plots of the same size. The system was augmented by implementing the layouts mechanism which was described in the previous chapter (see Section 4.3.4).

Layouts have been implemented only for the arrangement of figure regions on a device (i.e., not for the layout of components within a plot) and there are no layout margins (because par oma already takes care of that). Also, the syntax of the layout command differs somewhat from the implementation in SPG. The R help file for the layout command is given below:

```
layout: Specifying complex plot arrangements

usage :

layout(mat,
       widths=rep(1, dim(mat)[2]),
       heights=rep(1, dim(mat)[1]),
       respect=F)

arguments :

mat a matrix object specifying the location of the next N figures on the output device. Each value in the matrix must be 0 or a positive integer. If N is the largest positive integer in the matrix, then the integers 1..N-1 must also appear at least once in the matrix.

widths a vector of values for the widths of columns on the device. Relative widths are specified with numeric values. Absolute widths (in centimetres) are specified with the lcm() function (see examples).

heights a vector of values for the heights of rows on the device. Relative heights are specified with numeric values. Absolute heights (in centimetres) are specified with the lcm() function (see examples).

respect either a logical value or a matrix object. If this is a matrix then it should have the same dimensions as mat and each value in the matrix must be either 0 or 1.

description : layout divides the device up into as many rows and columns as there are in mat, with the column-widths and the row-heights specified in the respective arguments. Figure i is allocated a region composed from a subset of these rows and columns, based on the rows and columns in which i occurs in mat. The respect argument controls whether a unit column-width is the same physical measurement on the device as a unit row-height.

examples :
```
# divide the device into two rows and two columns
# allocate figure 1 all of row 1
# allocate figure 2 the intersection of column 2 and row 2
layout(matrix(c(1,1,0,2), 2, 2, byrow=T))
# show the regions that have been allocated to each plot
layout.show(2)

# divide device into two rows and two columns
# allocate figure 1 and figure 2 as above
# respect relations between widths and heights
layout(matrix(c(1,1,0,2), 2, 2, byrow=T), respect=T)
layout.show(2)

# create single figure which is 5cm square
layout(matrix(1), widths=lcm(5), heights=lcm(5))
layout.show(1)

# create a scatterplot with marginal histograms
x <- rnorm(50)
y <- rnorm(50)
xhist <- hist(x, breaks=seq(-3,3,0.5), plot=F)
yhist <- hist(y, breaks=seq(-3,3,0.5), plot=F)
top <- max(c(xhist$counts, yhist$counts))
xrange <- c(-3,3)
yrange <- c(-3,3)
layout(matrix(c(2,0,1,3),2,2,T), c(3,1), c(1,3), T)
par(mar=c(3,3,1,1))
plot(x, y, xlim=xrange, ylim=yrange, xlab="", ylab="")
par(mar=c(0,3,1,1))
barplot(xhist$counts, axes=F, ylim=c(0, top), space=0)
par(mar=c(3,0,1,1))
barplot(yhist$counts, axes=F, xlim=c(0, top), space=0, horiz=T)

The addition of layouts required adding a number of extra parameters to the GP and DP structures. These additional parameters replace those previously used to store the old layout specification (mfrow, mfcol, and mfg). The old layout specification are still valid, but they are now stored within the new parameters in the GP and DP structures.

Thus, the new layouts are an alternative to the old layout parameters. This means that only one sort of layout specification can be enforced at once; a new layout (from the layout command) or an old layout (from the par(mfrow=...), par(mfcol=...), or par(mfg=...) commands). For example, the following command sets the layout of the graphs as shown. The command par(mfrow=c(2,2)) sets the layout structure to:
The command `layout(matrix(c(2,1)), heights=c(2,1))` changes the layout to:

```
1  2
3  4
```

Finally, the command `par(mfcol=c(2,2))` sets the layout to:

```
2
1
```

Memory for units in layout parameters

In the old static graphics system, when a window was resized, the contents of the window were lost. The drawing commands had to be reissued by the user in order to redraw the graph. In the new system, drawing commands are recorded by the system so that when a window is resized the contents of the window are redrawn automatically. This introduces a problem for the old style of recording layout parameters. Suppose that the user has specified that the figure region is to be four inches square (i.e., `par(fin=c(4,4))`). When the window is resized, the graph should be redrawn in a figure region which is four inches square. However, the figure region was only stored as a proportion of the inner region so it was not possible to redraw the graph correctly because the units in which the figure region was specified were lost\(^3\). This problem was resolved by storing the layout parameters in all possible units and by recording the units that were last specified.

The GP and DP structures in each DD have been expanded to record each layout parameter in all possible units. In particular:

- the outer margins are now stored in terms of lines of text \textit{and} as a proportion of the total device surface \textit{and} in inches
- the figure region is stored as a proportion of the inner region \textit{and} in inches
- the figure margins are stored in terms of lines of text \textit{and} in inches

\(^3\)This is not a problem in the old system because the user had to reissue all of the drawing commands, \textit{including the par(fin=c(4,4)) command.}
the plot region is stored as a proportion of the figure region and in inches

In addition, there are extra parameters for storing which units were last used to specify each layout parameter:

- oUnits stores the outer margin units
- fUnits stores the figure region units
- mUnits stores the figure margin units
- pUnits stores the plot region units

The figure region is defaulted from the outer margins and the current plot layout unless the user specifies par(fig) or par(fin). Defaulting of the figure region is restored by specifying a new plot layout. If the figure region is defaulted, it is specified as a proportion of the inner region.

The plot region is defaulted from the figure region and the figure margins unless the user specifies par(plt) or par(pin). Defaulting of the plot region is restored by specifying the figure margins. If the plot region is defaulted, it is specified as a proportion of the figure region.

5.2.4 Internal details

This section is for developers of the internal C code of the R system.

Multiple outputs

The device descriptions are held in a global array. The following functions are provided for modifying the global array:

addDevice () The device-driver code which creates a new device description should use this function to add the new description to the global array.

CurrentDevice () Returns a pointer to the description of the current device.

deviceNumber (deviceDescription) Returns the index of the specified deviceDescription in the global array.

GetDevice (deviceNumber) Returns a pointer to the description stored in the deviceNumber element of the global array.

NoDevices () Returns whether the current device is the null device.

NumDevices () Returns the number of device descriptions in the global array.
Display lists

The following functions are provided for manipulating a device's display list:

- `initDisplayList(deviceDescription)` Clears the display list for the specified device and saves the current state of the device's graphical parameters.

- `recordGraphicOperation(operation arguments deviceDescription)` Adds the specified command in the display list of the specified device. This is called within all of the basic R graphics commands (e.g., `plot.new`, `axis`, ...). In other words, the display list consists of a number of basic R graphics commands.

- `playDisplayList(deviceDescription)` Performs each of the commands in the display list of the specified device. The saved state of the device's graphical parameters (see `initDisplayList`) are restored first.

- `copyDisplayList(deviceDescription)` Copies the display list from the specified device to the current device and copies the saved state of the specified device's graphical parameters to the current device and calls `playDisplayList` on the current device.

- `inhibitDisplayList(deviceDescription)` Turns off the display-list feature for the specified device and clears the device's display list. This means that basic R graphical commands will not be recorded on this device.

Layouts

The layout specification is stored as part of the GPar structure which contains the graphical parameters for a device. The following fields are defined:

- `layout (integer)` records whether a layout has been specified (i.e., whether the most recent layout command issued by the user was `layout` rather than `mfrow`, `mfcol`, or `mfg`).

- `numrows/numcols (integer)` record the number of rows/columns in the layout matrix. They are also used to record the number of rows/columns specified by `mfrow`, `mfcol`, and `mfg`.

- `currentFigure (integer)` records the current figure number.

- `lastFigure (integer)` records the highest possible value for `currentFigure`. If a layout has been specified then this equals the largest value specified in the order matrix, otherwise it is equal to `numrows` times `numcols`.

- `heights/widths (double array)` record the row-heights and column-widths for the layout.

- `cmHeights/cmWidths (integer array)` record which of the row-heights and column-widths were specified in absolute units.

- `order (integer array)` records the layout matrix.

- `rspct (integer)` records the layout respect: 0 = no respect, 1 = full respect, 2 = partial respect—see `rspct`. 
The current figure region is calculated in the function `mapFigureRegion`. If a layout has been specified, the figure region is calculated as follows: the `layoutRegions` function is used to calculate a set of row-heights and column-widths which represent portions of the inner region that have been allocated to each row and column in the layout matrix (see Section 4.3.4 for a description of these calculations); the `figureExtent` function is used to calculate which rows and columns the current figure occupies in the layout matrix; finally, the `subRegion` function is used to generate a figure region which is a proportion of the inner region.

**Coordinate systems**

The following coordinate systems are available for graphical operations:

- **device**: these are the native device coordinates (rasters), the origin depends on the device type (bottom, left for postscript; top-left for x11)
- **ndc**: (normalised device coordinates) this has an origin at the bottom, left of the device and (1,1) at the top, right of the device
- **nic**: (normalised inner coordinates) this has its origin at the bottom, left of the inner region and (1,1) at the top, right of the inner region
- **nfc**: (normalised figure coordinates) this has its origin at the bottom, left of the figure region and (1, 1) at the top, right of the figure region
- **npc**: (normalised plot coordinates) this has its origin at the bottom, left of the plot region and (1,1) at the top.right of the plot region
- **user**: this maps data coordinates onto the plot region (the axes reflect the range of values in the plot region); this has (xmin, ymin) at the bottom, left of the plot region and (xmax, ymax) at the top, right of the plot region
- **inches**: this has its origin at the bottom, left of the device and (device-width-in-inches, device-height-in-inches) at the top-right of the device (this is mostly used for unit conversions)
- **lines**: this has its origin at the bottom, left of the device and (device-width-in-lines, device-height-in-lines) at the top-right of the device (this is mostly used for unit conversions)
- **chars**: this has its origin at the bottom, left of the device and (device-width-in-chars, device-height-in-chars) at the top-right of the device (this is mostly used for unit conversions)
- **oma1**: this has its origin at the bottom, left of the inner region and (1, oma[0]) at the right of the inner region and the bottom of the device (x-values are in nic and increase to the right, y-values are in lines and increase down)
- **oma2**: this has its origin at the bottom, left of the inner region and (1, oma[1]) at the top of the inner region and the left of the device (x-values are in nic and increase up, y-values are in lines and increase to the left)
omn3: this has its origin at the top, left of the inner region and (1, oma[2]) at the right of the inner region and the top of the device (x-values are in nic and increase to the right, y-values are in lines and increase up)

omn4: this has its origin at the bottom, right of the inner region and (1, oma[3]) at the top of the inner region and the right of the device (x-values are in nic and increase up, y-values are in lines and increase to the right)

mar1: this has (xmin, 0) at the bottom, left of the plot region and (xmax, mar[0]) at the right of the plot region and at the bottom of the plot region minus mar[0] (x-values are in user(x) and increase to the right, y-values are in lines and increase down)

mar2: this has (xmin, 0) at the bottom, left of the plot region and (xmax, mar[1]) at the top of the plot region and at the left of the plot region minus mar[1] (x-values are in user(y) and increase up, y-values are in lines and increase to the left)

mar3: this has (xmin, 0) at the top, left of the plot region and (xmax, mar[2]) at the right of the plot region and at the top of the plot region plus mar[2] (x-values are in user(x) and increase to the right, y-values are in lines and increase up)

mar4: this has (xmin, 0) at the bottom, right of the plot region and (xmax, mar[3]) at the bottom of the plot region and at the right of the plot region plus mar[3] (x-values are in user(y) and increase up, y-values are in lines and increase to the right)

All of the primitive graphical operations (GCircle, GLine, ... ) take a coordinate system as an argument. This means that a programmer can work in whichever coordinate system is most convenient. For example, in the code to draw an axis, the location of the tick-marks along the axis can be calculated in user coordinates and the length of the tick-marks can be calculated in lines. The functions GConvert, GConvertXUnits, and GConvertYUnits are provided to make it easy to transform between arbitrary pairs of coordinate systems. These functions are used in the device drivers to perform the final transformation from an arbitrary coordinate system to device coordinates.

Event-handling

The function ProcessEvents is called whenever a user event is detected. This function currently handles resizing windows and closing windows via the window manager.

5.2.5 Conclusions

The new R graphics engine has a number of useful features: multiple output devices, user event handling, automatic redrawing, and a powerful layout mechanism. In addition to the immediate benefit from these features, the changes provide a strong basis for further development of R's graphics capabilities. In particular, the more event-driven user interface makes it possible to contemplate developing interactive graphics features such as brushing plots (McDonald, 1982; Becker and Cleveland,

4 This only applies to the UNIX version of R
1987) and spinning 3D plots (Fisher, Keller, Friedman, and Tukey, 1974), the layout mechanism provides improved support for developing new interpreted R plotting functions, and the freedom to work in an arbitrary coordinate system makes it easier to develop new basic R graphics operations.
5.3 References


Chapter 6

Discussion

6.1 Static vs. Automatically-Updating Graphics

In a static graphics system (e.g., S-Plus), once a graphical element has been created it cannot be modified. Further drawing elements can be added to the same picture (e.g., add an additional y-axis, add text annotation, ...), however, if existing elements require modification, the complete graph must be recreated. For example, suppose that a user creates a scatterplot and then decides that the number of tick-marks on the x-axis is wrong. In order to modify the number of tick-marks, the user must discard the existing scatterplot and run the function again to draw a new one with the correct number of tick-marks.

A system which provides automatic updating allows the user to modify an existing graphical element. This makes it possible to edit a graph by simply entering a command to edit the appropriate feature as opposed to re-entering all of the original commands, with the appropriate modification included.

In Simplisp, Xtend, and SPG, automatic updating is provided as follows. A graph (or an axis, or a piece of text, ...) is represented by an object. The object stores the description of the graph and this description can be modified by the user. The object knows where it is being viewed (which windows it appears in) and forces a redraw whenever it is modified.

In Simplisp, a separate command is provided for every possible editing operation. For example, there is a function for editing the text of the label on an axis, another function to edit the distance of the label from the axis, a further function to edit the location of the label along the axis, and so on. This has the disadvantage that there are an enormous number of functions to learn. In Xtend and SPG this problem is overcome by having a single editing function, edit-slot, which has parameters to specify which object to edit as well as what aspect of the object is being modified. For example, the expression (edit-slot x-axis :num-ticks 7) specifies that the num-ticks slot of the x-axis object should be set to the value 7.

Simplisp also has a problem with allowing editing of objects which are components of another object (i.e., objects which are embedded within an object hierar-

\[\text{1}^{1}\] differentiate here between creating a graphical element and rendering a graphical element, which means drawing the element on an output. Typically, both static and automatically-updating systems automatically render a graphical element when it is created; the difference is that in an automatically-updating system, the system also automatically renders a graphical element whenever it is modified. This difference is sometimes referred to as immediate-mode versus retained-mode graphics.
CHAPTER 6. DISCUSSION

It is possible to select such objects with the mouse, but they are inaccessible from the command-line. A partial solution involves having slots in the top-level object (the object which owns the components and is accessible from the command-line) which are copies of the slots in the components. For example, an axis object, which owns a text object to represent its axis label can have a :label-height slot which copies the :height slot of the text object. When the user edits the :label-height slot of the axis, the axis modifies the :height slot of the text object accordingly. The problem with this approach is that higher-level objects tend to get a rather silly number of slots (copies of all of the slots of all of their components, which will include copies of the slots of their components, and so on). Xtend and SPG solve this problem also by providing a special syntax which allows slot specifiers to be concatenated. For example, the expression (edit-slot x-axis :label :height .05) specifies that the :height slot of the object in the :label slot of the x-axis object should be set to the value .05.

The work on display lists in R is a somewhat different form of updating. To explain this it is necessary to make a distinction between the graphics engine of a system and the output devices of the system. The graphics engine accepts commands from the user to create and modify graphical elements and it sends commands on to the output devices to render those graphical elements. In an automatically-updating system, the graphics engine retains some sort of representation of the graphical elements that have been created. This allows the user to modify these elements and allows the graphics engine to render the modified elements. Separately from this, the output devices may retain some representation of the rendering that they have performed. This allows a device to perform the same rendering again when, for example, a window is resized. The difference is that, when a device uses a display list it is performing the same rendering (possible within a different device context; e.g., a differently-sized window), whereas when a system performs automatic updating it performs a new rendering.

The difference between the display lists in R and the behaviour of output devices in S is the level at which rendering activity is represented. In the R display lists, the original R commands (including the original parameter values) are recorded. In S the rendering is recorded at a lower level which means that some of the "intelligent" features of a graph are not kept. For example, when a window containing a scatterplot is resized, the dimensions of the scatterplot margins and the locations of axis and tick labels are simply scaled for the new window size rather than being recalculated using the original plotting code.

6.2 Customisation

Customisation refers to modifying the fine details of a graph. Customisation was achieved in Simplisp and Xtend by fragmenting a graph into many components, each with its own description. In a traditional graphics package, a graph consists of a rigid structure. For example, a scatterplot consists of an x-axis, a y-axis, a set of points, plus possibly a secondary y-axis. Each element of this structure has a number of parameters that can be modified in order to change the element’s appearance. For example, an axis has parameters which describe the number of tick-marks on the axis, the range of values on the axis, and so on. In Simplisp and Xtend, each of these elements is further broken down into individual components (e.g., an axis consists of a label and some tick-marks) and each of these components
CHAPTER 6. DISCUSSION

may even be broken down into further components (e.g., a tick-mark consists of a line and a label). The idea behind this is that, rather than having a single description of entire plotting elements in one place, there are a multitude of small plotting elements, each with their own description. Each element can be interacted with directly—for example, the colour and size of an individual tick-mark can be modified independently of the axis that the tick-mark is contained in by modifying the tick-mark’s own description—or indirectly via its parent—for example, all of the tick-marks on an axis can be resized at once by modifying the axis description.

Arranging the components of statistical graphs in a hierarchy has been suggested previously by Hurley and Oldford (1991) and Wilks (1996). Hurley and Oldford describe two useful reasons for using hierarchies of objects:

1. the localisation of information within components of a hierarchy means that the description of each component is relatively simple—each component needs to know that it may have a parent and/or children, but it needs to know very little about its parent or children. This advantage is particularly apparent in Xtend and SPG because the edit-slot function allows the user to access the specific component of an object so that the information for each component only needs to reside with that component. Also, Xtend and SPG allow objects to be combined in a very generic manner, which further allows the designer of a new component to concentrate solely on the description of the new component with little regard for potential parents or children of the component.

2. different graphs can use the same object prototypes for components that they have in common (e.g., a scatterplot and a histogram can use the same axis prototype for their axes).

The object hierarchies in Simplisp, Xtend, and SPG are different from those described by Hurley and Oldford (1991) because they are more fine-grained. The original motivation for object hierarchies in Simplisp was to provide access to every detail of a graph, right down to individual graphical primitives. Consequently, the object hierarchies in Simplisp, Xtend, and SPG include more components and each component is smaller and simpler. This does allow a far greater degree of customisation and a greater potential for reusing prototypes (see Section 6.4).

Pictor (Wilks, 1996), also represents graphical elements in a hierarchical “tree” structure. The motivation for hierarchies within this system is to allow the components of graphical elements to be arranged relative to each other in a more flexible and efficient manner. The hierarchies are very fine-grained, going down to the level of graphical primitives, but this system is aimed more at the designers of new graphical elements rather than at allowing a high degree of customisation; Pictor graphics are static so are not designed for being manipulated in fine detail by the user. Pictor is discussed in more detail in Section 6.5.

An alternative approach to allowing unlimited customisation would be to simply make the high-level description more and more detailed. For example, the description of an axis could include a length for each tick-mark instead of, or as well as, a single length for all tick-marks. The disadvantage of this approach is that it produces clutter and complexity in the high-level description, which translates to clutter and complexity in whatever interface the user encounters—command-line parameters or dialog boxes. In the hierarchical approach, the description of a plotting element is not concerned with the details of the components of the element—those details can be found in the components themselves.
CHAPTER 6. DISCUSSION

Some packages only allow for customisation to a certain level of detail. This can be annoyingly restrictive, but it can be argued that some sort of restriction is required.

The approach taken in this thesis is that the user should be allowed as much freedom of expression as possible; the role of the statistical graphics package is not to frustrate, but to facilitate. On the other hand, what wisdom there is on the correct design of graphical displays espouses simplicity and clarity and castigates the scourge of gratuitous decoration or chartjunk (e.g., Tufte, 1983, 1990). Should these design recommendations moderate a drive toward ever-greater powers of customisation?

Consider the graphics facilities of S-Plus (version 4). In this system, the user is allowed a high degree of interaction with the graphs that the system produces. For example, the user is able to select and drag separate elements of a graph axis and relocate them independently of the other elements of the axis. This capability can be used to produce the most hideous results possible. Does this mean that these interactive facilities should be removed?

The situation is analogous to that of a researcher developing a technology with a potentially dangerous application; who is responsible for the damage caused by such technology: the researcher or the person who applies the technology?

This is a difficult question (although the outcomes are not usually as life-threatening in graphical statistics as in, say, nuclear physics). A possible safety feature in a graphical statistics package would be a way to get back from whatever private hell users manage to immerse themselves in. This might be in the form of an “undo” facility or some way of restoring defaults that the system normally provides. For example, if a user manages to scatter the various components of an axis to the four corners of a plot and realises the folly of this sort of manipulation, normality can be restored by undoing each of the unwanted modifications one at a time, or even all at once by re-enforcing the default axis layout. This sort of facility may be required to make arbitrary customisation a safe feature, but at least until restrictions which are motivated by principles of correct graphical design have been formalised and agreed-upon there should be no restrictions enforced.

6.3 Incremental Graphics

Incremental graphics is concerned with allowing the user control over the structure of a graph. The structure of a graph is the type, number, and layout of the components which make up the graph (e.g., a scatterplot might be allowed up to four axes which are arranged around the outside of the region containing the points).

In a standard graphics package which provides a set of predefined graph types, the structure of a graph is largely immutable. For example, a scatterplot is composed of a y-axis, an x-axis, and some data points. It may be possible to annotate the graph with additional text or lines, but these additional elements know nothing about the location or coordinate systems of the graph (e.g., it is not possible to locate a piece of text at a specific data location other than by eye).

The S system provides somewhat more freedom for the structure of a plot. Graphs come with a standard structure (a standard set of components and a standard layout for them). It is, however, possible to add further graphical elements to a graph which can be coordinated with the existing graph, including lines and text and axes and data points. This is possible because the coordinate system implicit in a graph in S is available to a number of other drawing operations. For example, the
text command positions text within the coordinate system of the data in a graph, the mtext command positions text within the margins of a graph, and so on.

Simplisp takes the approach that a graph is an arbitrary collection of graphical elements. This has the advantage that any graphical elements can be combined together. Unfortunately, although Simplisp removes the traditional restrictions on the structure of graphs, it does not provide enough glue to automatically generate the standard ties between graphical elements (e.g., the sharing of coordinate systems between axes and data points). This means that it is easy to lump a number of graphical elements together, but it is not easy to get them to behave in concert.

Xtend persists with the idea that graphical structure should be fully modifiable, but provides much more support for allowing the components of a graph to share information (e.g., sharing scales between axes and data points). This is achieved by providing an inheritance mechanism. For any graphical element in Xtend, it is possible to specify that the element will look for certain information in its parent (if it has one). For example, an axis object will look for an x-scale and a y-scale in its parent (which is likely to be some sort of plot; e.g., a scatterplot or a histogram). The information is searched for by looking for slots in the parent with a specific name (see Section 3.3.2). This means that a parent can provide information for its children simply by having the information in a slot with the appropriate name. This sharing of information allows graphical elements to behave in a coherent manner.

A weakness in both Simplisp and Xtend is the lack of support for specifying the layout of the elements of a graph; both packages make it easy to combine an arbitrary number of graphical elements together in a single graph and Xtend makes it easy to coordinate the behaviour of the elements, but the placement of these elements still requires effort and coordination from the user. SPG provides a layout mechanism which allows the elements of a graph to be arranged easily and intuitively.

Pictor (Wilks, 1996) represents graphs as hierarchies of graphical components and allows the user to specify graphs as arbitrary combinations of graphical components, so how is it different? The difference is that Pictor’s graphs are not incremental. In Pictor, a graph or graphical element, such as an axis, is specified by a function. For example, the following code defines a function for a bottom axis (from Wilks, 1994a):

```r
baxis <- function(Rx)
{
  xpos <- nice(Rx, out=F)
  nx <- length(xpos)
  gpaste(axisline = segments(xpos[1], 0, xpos[nx], 0),
         ticks = segments(xpos, rep(0, nx), xposm rep(1,nx)),
         ticklabels = text(xpos, 1, format(xpos),
                            textadj="top"),
         constraints = list(constraint("yscale", "/", "", -0.2)))
}
```

The significant feature of this function is the gpaste command. This command is where the components of the axis are specified along with how the components are to be arranged and the relationships between their coordinate systems. Suppose an axis is constructed from this function. The structure of the axis is fixed by the description given in the function. It is not possible to alter the structure of that axis. In order to modify the structure of a graphical element, it is necessary
CHAPTER 6. DISCUSSION

168

to edit the function which produced the argument (or write a new function) and create a new graphical element. It is not possible to add components to (or remove components from) an existing graphical element as in Xtend and SPG.

The significance of incremental graphics can be seen by considering the different ways that a graphical object, for example a piece of text, could be added to a graph:

1. add text as annotation (e.g., add a title at the top of a page). This requires absolutely no knowledge of the graph or its coordinate systems; the text is located and sized with respect to the device surface only. Most graphics packages allow this sort of annotation.

2. add text to a graph (e.g., add a label to a point or line in a graph). The text is located and sized relative to the graph so that if the graph is modified (e.g., different data are plotted or the graph is resized), the text will be relocated and/or resized appropriately. This requires knowledge of the coordinate systems associated with the graph. For example, S provides the text function for positioning text relative to locations within the data region of a graph and the mtext function for positioning text in the margins of a graph.

3. add text to a component of a graph (e.g., add a label to the internal region of a boxplot or to the inside of a data symbol). The text is located and sized relative to the component so that if the component is modified (e.g., resized or even deleted) the text will be relocated and/or resized (or removed) as is appropriate. This requires that the structure of the graph is represented somehow (e.g., as a hierarchy of graphical components) and that the user has access to the components of the graph.

4. add (or remove) text to (or from) a component of a graph interactively (e.g., create a graph and then add a label to a data symbol within the plot). The text is added independently of the definition or creation of the graph and its components. This requires that the structure of the graph can be modified interactively (i.e., not just when the graph is created or defined).

An important requirement for the latter two items in the above list is that information about the graph must be retained. In S, limited information about a graph is retained in the form of coordinate systems for the current figure (e.g., where is the current figure in the window and what are the current range of values in the data region). However, S does not retain information about what has been drawn so it is not possible for subsequent drawing operations to be coordinated with previous drawing operations (other than through the shared coordinate systems). In contrast, the software described in this thesis retains a representation of graphical elements—plotting elements are represented by objects. This means that it is possible for new plotting elements to get information about existing plotting elements, which means that the new plotting elements can be sensitive to or be coordinated with the existing plotting elements in arbitrary ways. In other words, incremental graphics, as they are defined in this thesis, are dependent upon some sort of retained graphics.

6.4 Extensibility

Extensibility is the ability to create completely new graphical components; to extend the capabilities of the graphics system. The Xtend project (Chapter 3) takes a new
approach to providing extensibility (for graphical statistics) which is based on the use of general constraints.

Constraints are used in Pictor (Wilks, 1996) to specify the arrangement of plotting elements relative to each other and to specify relationships between the coordinate systems of different plotting elements, however, only a limited set of constraints are available and they are available only for these specific purposes.

Hurley and Oldford (1991) discuss links between objects in a statistical graphics system within the context of constraints. In the Quail system, a number of different links are possible (e.g., between data points in separate graphs and between a text object and the value of a slider), however, there are only a limited number of such linking constraints available. Hurley and Oldford discuss the usefulness of general constraints for being able to create links between arbitrary pairs of objects.

The earliest example of the explicit use of constraints in a statistical graphics system that I am aware of is in the Antelope system (McDonald, 1986). This system provides simple constraints with the following features:

- constraints are separate objects
- constraints are between two objects; one is the leader and one is the follower. The constraint object has a pointer to each of these
- each constraint has a strategy to determine when the leader should send the follower an update-message
- if the constraint strategy is immediate then the constraint sends the follower the update-message whenever the leader receives a state-change-message
- if the constraint strategy is on-request then the constraint sends the follower the update-message whenever the follower receives a query-message

A constraint is created by specifying a leader object, a follower object, a strategy, an update-message, a state-change-message, and a query-message.

Antelope does not provide a complete constraint system. In particular, only simple dependency constraints are allowed (e.g., the state of object A can depend on the state of object B, but the state of object B cannot also depend on the state of object A at the same time; this is called circular dependency). However, the system does provide some support for general constraints, which is the requirement for being able to generate arbitrary relationships between objects.

Xtend is based upon the KR module of Garnet (Myers et al., 1990), which provides a complete constraint system. This allows the user to specify arbitrary relationships between arbitrary objects within the system, which is a significant boon for extensible software. The advantage of such a constraint system over a generic programming environment is that the user only has to declare the relationship between two objects and the constraint system takes care of maintaining the relationship (which involves determining when to update the relationship as well as actually performing the updating).

This is particularly pertinent in a system which allows the user to modify existing graphical elements. For example, consider the task of designing a scatterplot which consists of an x-axis, a y-axis, and a set of data points. The positioning of the components of the scatterplot depends on the location and size of the scatterplot. It is a simple matter to specify how to calculate the positions of the components from the location and size of the scatterplot (this is the declaration of the relationship between the scatterplot and its components). What the constraint system does is
automatically maintain the relationship if the location and/or size of the scatterplot are modified. Without a constraint system, the designer must specify when the relationship should be updated, which requires some way of detecting when the size or location of the scatterplot has been modified (this is one reason why there are so many specific editing functions in Simplisp).

The situation is significantly more complicated if components can be added to an existing graphical element. For example, suppose a boxplot is added to a scatterplot to indicate the marginal distribution of the y-variable. In this case, a new relationship must be specified between the new component and the existing graphical element. For example, the location and size of the boxplot will depend on the size and location of the scatterplot. It is extremely useful to have the new relationship maintained automatically instead of having to determine how changes in the existing graphical element may affect the new component. For example, without a constraint system, it would be necessary to modify the code which handled changes to the location and size of the scatterplot in order to make sure that such changes caused an update of the location and size of the boxplot.

However, Xtend does not simply ride upon the coat-tails of KR's constraint system. All graphical elements in Xtend are derived from the group object. This means that a number of behaviours are automatically inherited by all new graphical elements. In particular, no new code is required for rendering a new graphical element and no new code is required for selecting the new graphical element and its components with the mouse. Also, all new graphical elements inherit the behaviour which allows them to share information with their components and any other objects which they are a component of (see the previous section on Incremental Graphics).

There are three important aspects to the environment that Xtend provides for developing novel graphical elements. First of all, Xtend provides a single, simple, coherent framework for extensible statistical graphics. In Xtend, there are three parts to a graphical element: a high-level description of the element; a description of the components of the element; and a description of the relationship between the element and its components. Secondly, Xtend provides significant support for the difficult part of describing a graphical element—the relationship between an element and its components. This support consists of a general constraint system, which takes care of maintaining relationships, and an inheritance mechanism, which provides a way for an element to provide information for its components and for the components to find information in the element without either the element or the component knowing anything about each other. For example, an axis can provide information for a tick-mark without knowing whether the tick-mark is there or not. Similarly, the tick-mark can find information in the axis without knowing that it is looking in an axis. Finally, Xtend encourages experimentation with graphical elements by allowing components and relationships to be added and removed interactively.

The extensible environments provided by XLispStat and Quail are also based upon deriving new graphical elements from a template provided by the system, however there are differences in the nature of the templates in the support provided for fine-tuning the new graphical element.

XLispStat is extensible because it is an object-oriented environment which allows the existing graphics functionality to be modified for new purposes. This is achieved by deriving new graphical objects from existing ones in order to inherit useful graphical features and defining new methods (behaviour) to specialise the new graphical object. The fact that a new graphical element can be created by
modifying an existing one means that the user does not have to start from scratch. However, users are pretty much on their own after that. There is no support for generating or maintaining sophisticated relationships between graphical elements. Furthermore, the choice of graphical objects to build on is not appropriate for developing arbitrary statistical images. There are two primary objects on which to base a graphical element, the graph-window-proto and the graph-proto. The graphical structure provided by the graph-proto is quite rigid; the graph is divided into a plot region consisting of a content rectangle, outside which axes are drawn, and a margin (which goes outside the plot). This sort of rigid structure is by no means appropriate for arbitrary statistical images. In order to avoid this structure it is necessary to use the graph-window-proto instead. This object provides basic support for graphical operations, but no sophisticated data-handling (e.g., there is no concept of scales or separate coordinate systems). In summary then, the environment allows the user to create new graphical objects easily as long as they are similar to graphical primitives or scatterplots; otherwise a considerable amount of work is involved.

Quail also allows extensibility through deriving new graphical elements from existing ones. Quite a range of graphical elements are available and there is support for arbitrarily combining elements together and for creating links between certain types of elements. The major difference with Quail is that there is no single, coherent method for creating new graphical elements from existing ones; there appears to be a distinct approach for each different graphical element. This makes it difficult to determine what approach is required for a new graphical element.

Constraints are not the only feature which contribute to the extensible nature of Xtend. The fact that graphical elements are represented by object hierarchies also provides an excellent basis for extensibility. Section 6.2 mentioned that one advantage of object hierarchies is that the components in a hierarchy become relatively simple to design. The significance for extensibility is that new graphical elements can be created as a collection of quite simple components. This means that each component is easier to design and each component can be used in other new graphical elements.

In the discussion of customisation (Section 6.2), I considered the dangers of providing the user with too much power for customisation. An extensible graphics system by comparison is positively lethal. Should extensible systems be outlawed? We can begin to see how ridiculous this objection really is. There is no limit to how much trouble a programmer can get into with a conventional programming language like C or Lisp, but this doesn't mean that all the potentially dangerous commands are removed from these languages (there would be nothing left!). Instead, the user is responsible for understanding the tools in order to expect to use them properly. The same principle should apply to statistical graphics.

6.5 Presentation-Quality and Flexibility

Many statistical graphics packages produce very high-quality graphical output, but are relatively inflexible in terms of the graph structure and cannot be extended. On the other hand, some packages provide a lot of flexibility and extensibility, but only poor output quality.

Typically, a statistical graphics system will provide a very flexible 3D graph which is of very poor quality or a high-quality 3D graph which the user cannot
CHAPTER 6. DISCUSSION

interact with much. Simplisp (Chapter 2) provides high-quality 3D graphs which can be accessed as much as 2D graphs.

A major problem in an extensible (i.e., very flexible) system is that of specifying the arrangement of plotting elements. In a graphics system which only provides a choice of predefined plot structures, the layout of the graph is predefined (with possibly some parameters to make slight modifications; e.g., change the direction of the tick-marks on an axis). In Simplisp, Xtend, and SPG, a graph could be constructed by combining an arbitrary number of different plotting elements together. In Xtend and SPG, plotting elements could be arbitrarily combined within another plotting element (e.g., a data-series could be added to a scatter-plot, a piece of text could be added to a data symbol, ...). In these situations, it is not possible to know a priori what the components of a graph (or a plotting element) will be so it is not possible to determine an arrangement of components which will be guaranteed to satisfy. It is possible to provide a default arrangement in most cases (e.g., a scatterplot has a default set of components for which a default arrangement can be defined), however, there is a need for being able to modify the complete arrangement if further components are added (or a new plotting element is created). The layout mechanism described in Chapter 4 provides the required power.

Pictor (Wilks, 1996) approaches the layout of graphical elements in a somewhat different way. Like Simplisp, Xtend, and SPG, graphical elements in Pictor are broken down into components, which are arranged in a hierarchy. The designer of a new graphical element expresses the relationships between the components of the graphical element (the relative positions of the components and the associations between the coordinate systems of the components) by a set of constraints. The total set of constraints are then "solved" to determine where each component is placed on an output and what the coordinate systems are for each component. This differs from SPG's layout mechanism in that the specification of a layout is more of a suggestion in Pictor and more of a directive in SPG. In Pictor, the user specifies a set of constraints and the software produces a solution to those constraints which maximises the sizes of the components (so as to use the space available most efficiently). In SPG layouts, the user specifies how the components are to be arranged and the software simply does the number-crunching to achieve that layout. Another difference arises from the fact that Pictor is a procedural package dealing with static graphics, while SPG is an object-oriented package dealing with automatically-updating graphics. The difference is that SPG has to be able to deal with graphical elements whose structure can be modified interactively. Recall that in SPG components may be added to or removed from an existing graphical element (e.g., the user can create a graph, then add an axis, then remove it again). Accordingly, it is possible in SPG to simply specify a new layout for a graphical element interactively (e.g., when a new axis is added, a new layout is also specified which allows room for the new axis). In contrast, the layout mechanism in Pictor is expressed as part of the procedural description of a graphical element. For example, the function to draw a scatterplot includes descriptions of the axes and constraints which specify the locations of the axes. It is not possible to add a further axis to this scatterplot without writing a new function which includes a description of the new axis and constraints to specify where the new axis should be located.
6.6 Stand-Alone Graphics

Simplisp, Xtend, and SPG are all designed to be stand-alone statistical graphics packages. This means that they do not include any statistical analysis features and rely on other software to provide them. This means that they have to communicate somehow with other software. The Xtend and SPG packages include a facility for communicating with the R system (Ihaka and Gentleman, 1996).

XGobi (Swayne and Cook, 1990) is a stand-alone statistical graphics package which has solved the communication problem in a number of ways (see Section 1.3.6). However, XGobi differs significantly from Xtend and SPG. XGobi is designed to provide a number of predefined graphical features for the analysis of data; Xtend and (to some extent) SPG are extensible graphics systems designed to provide a basis for experimentation with new graphical features. Not suprisingly then, the communication facilities in XGobi are quite different from those implemented in Xtend and SPG. The emphasis for XGobi is communicating either raw data or a finite set of commands (corresponding to the finite set of graphical features) from software which performs statistical analyses to XGobi (i.e., XGobi is used as a graphics engine for a statistical analysis package). In particular, the communication is implemented by the developers of XGobi (and/or the analysis software) and is hidden from the user.

In Xtend, the emphasis is on communicating commands from Xtend to R (i.e., R is used as a statistical engine for XGobi). This is because the analysis features of R are required as support for developing new graphical images. In particular, the user has to be able to send arbitrary commands with all of the detail of normal R commands. This is achieved by sending R commands as text via UNIX pipes and receiving results from R in the same way. The communication between R and Xtend is therefore more complete and more dynamic because new R features are instantly accessible.

6.7 Lessons Learned

This section provides advice for designers of new statistical graphics systems.

Development platform. The Lisp environment is excellent for the rapid development of code. It is not a popular environment for users, but the significance of this would depend on the potential audience for the software; research statisticians might be willing to overcome the hurdle of Lisp syntax in order to use a powerful statistical graphics research tool, but applied statisticians who require an easy-to-use ready-made graphics package are unlikely to be tempted.

Programming paradigm. The object-oriented programming paradigm is excellent for developing and maintaining code, especially for graphics applications. This approach is also useful as a basis for extensibility as in XLispStat (Tierney, 1990) and Quail (Hurley and Oldford, 1991).

User interface. The choice between a command-line interface and a GUI depends largely on the intended audience for the software. At this time, a command-line is really the only option for software which allows the user to program new features. As demonstrated by XLispStat (Tierney, 1990), both types of interface can coexist.
CHAPTER 6. DISCUSSION

Retained graphics. A system which just draws to the screen and does not remember anything about what has been drawn is quicker and easier to develop initially. However, there are many benefits to be gained from having a system which retains information about what has been drawn. For example, it is easier to redraw an image when windows are resized and it is easier to add GUI features which rely on interaction between the user and the image (e.g., the system has information to identify objects within an image on the basis of a mouse click). Also, a system must have retained information about an image in order to provide incremental graphics capabilities.

3D graphics. 3D graphs need to be handled as a separate case from 2D graphs—viewing a 3D graph (see Section 2.2.2) involves much more complexity than viewing a 2D graph (see Section 3.2.1) and user-interaction with 3D graphs is different from that with 2D graphs (e.g., it is only really sensible to drag objects within a 2D graph and it is only really worthwhile spinning a 3D scatterplot). On the other hand, Simplisp demonstrates that many of the standard features of 2D graphs, such as the ability to select the components of a graph with the mouse and the ability to customise any of those components, can also be provided for 3D graphs.

Coordinate systems. It is very useful if coordinate systems are available for locating and sizing any graphical object (i.e., do not have coordinate systems that only work for certain types of graphical object) anywhere within a graph. The margin coordinate systems in \( S \) and \( R \) are particularly useful for positioning text relative to axes.

Extensibility. An important part of providing extensible graphics is providing a clear and simple method for creating new graphical objects. A constraint system may be difficult to incorporate, depending upon the development platform, but it does provide an excellent basis for defining the behaviour of new graphical objects and the relationships between them. The ability to create graphs incrementally makes it easier to experiment with new graphical objects.

Customisation. The basic idea behind allowing customisation is not to make any arbitrary decisions about the structure or appearance of a graph that the user cannot override. In practice, this is quite difficult to avoid. The best solution is to make the system extensible so that, if users really cannot modify an existing object to meet their needs then they can create a new object.

Graph layout. Many systems lack a simple, but powerful mechanism for arranging plots on a page. The layouts mechanism described in Chapter 4 could be implemented in any system to provide this functionality.

6.8 Future Directions

Supporting Customisation

It has been argued throughout that users should not be protected from themselves. The user should be assisted in the creation of graphs by providing sensible defaults for the structure and layout of a graph. However, the user should be able to override
all of the defaults. Providing ultimate powers of customisation has been the motivation behind many of the issues described in this thesis. Several possible topics for future investigation arise from this approach.

First of all, given a system which allows the user to change virtually anything, it would be useful to provide some sort of "undo" or "reset defaults" mechanism.

Secondly, it was argued that users of an extensible graphics package should be given all of the freedom and responsibilities of users of conventional programming languages. This suggests that the users of extensible graphics software would benefit from the sorts of tools available to programmers. In a conventional programming language, there are many diagnostic tools for detecting problems in a program (e.g., compiler warnings and run-time debuggers). Possibly some sort of similar facility could be developed for graphical statistics. Parsers and debuggers already exist for R and S expressions, but perhaps some sort of graphical parser could be developed which evaluates a graphical composition and warns against any horrible errors and suggests improvements.

**Fine Tuning Xtend**

One of the virtues of the Xtend system is the ability to modify the structure of graphs interactively. This is achieved by allowing the user to add components to a graph interactively and define new slots in a graph which are automatically detected and inherited by the children of the graph. The current limitation of this feature is that the slots that children look for in their parents (the inherited slots of an object) can only be specified in the definition of a graphical element. It would be worthwhile allowing these to be edited interactively too so that the user could not only add or remove slots in a parent for the components of the parent to detect, but also modify which slots the components will look for. Similarly, the private slots (slots which components cannot detect) and the data slots (slots which will contain variables) can only be specified in the definition of a graphical element. If these could also be specified interactively then the inheritance of values between objects would become still more flexible and the functionality of graphical elements would be further enhanced (e.g., an extra variable could be added to a data series which could then represent the extra variable by the size of its symbols).

The other major goal would be to investigate how to solve the speed and memory problems of Xtend. As mentioned in Section 3.6.4, the immediate aim would be to reduce the number of constraints in the system by implementing the inheritance mechanism by hand rather than using KR constraints. It is important to stress that this only involves removing constraints from the internal implementation of the Xtend system. Constraints would retain an integral role in specifying the behaviour of new graphical objects and in specifying the relationships between objects.

**Solving the Accidental Inheritance Problem**

The inheritance model implemented in Xtend to provide support for incremental graphics is based upon the use of slot names. That is, object A can provide information for object B to inherit by having the information in a slot with an appropriate name. Unfortunately, object B has no way of knowing whether a slot in object A has been given a certain name for the purpose of allowing object B to inherit the value in that slot or whether the slot has that name just because it seemed like a good name for the slot. This is essentially an example of what is known as the *namespace* problem. Consider the following Lisp function:
(defun double (x) (* 2 x))

This function doubles the value in the argument by multiplying it by two. Suppose that this function is part of a program and that, later in the program, another function is written which doubles the value in the argument by returning a list which contains the elements twice:

(defun double (x) (list x x))

Now, when the double function is called, which action should be performed? What will happen is that whichever function definition was most recent will be in effect, however, this is not desirable because sometimes the function will be called in order to multiply a value by two and other times the function will be called in order to generate a list with a value repeated twice. The obvious solution is to change the name of one of the functions, but ideally it would be nice if the situation could never arise; it is not always a simple matter to determine that a program is performing incorrectly because of this sort of problem. Also, it is convenient for a programmer to use a name which is most appropriate for a particular function rather than choosing a name which avoids conflict with other existing names.

The Lisp namespace problem described above is only really relevant in large programs (where the number of function names gets large so it is difficult for the programmer to remember which names have already been used). The solution to this problem in Common Lisp is provided by the *packages* feature. This allows names to be assigned to a package so that, for example, the name double in the package numeric-package can be differentiated from the name double in the package list-package. In Xtend, a similar approach is taken by allowing slot names to be specified as private. This splits names into two categories so that an object can only inherit a value from a slot with the appropriate name which is not private.

Another solution to the namespace problem involves using a special prefix or suffix for certain names. For example, in C it is possible to define macros which allow a value or a series of commands to be associated with a name (they are like functions only faster). There are a number of predefined macros with most C systems which would be lost if the C programmer defines a new macro with the same name in a program. To avoid this, the names of the predefined macros are all of the form:

```c
__name__
```

This means that all C programmers just have to know not to use this sort of format for their macro names in order to avoid conflicts with predefined macros. It may be worthwhile investigating a similar sort of naming convention to try to solve Xtend's namespace problem.

**Putting It All Together**

There are a number of desirable features in each of the Simplisp, Xtend, and SPG packages. The most exciting application of these features is to the development of an extensible statistical graphics system.

The framework for creating new graphical elements in Xtend, supported by general constraints and an inheritance mechanism provides a good basis for extensible
graphics. Xtend already includes a number of useful features first developed in Simplisp: statistical elements are represented by hierarchies of components; statistical elements are represented as persistent objects which allows them to be interactively modified; and outputs are automatically updated when statistical elements are modified. It has already been noted that Xtend's performance would be improved by implementing some of the internal mechanisms, such as inheritance, directly rather than relying on constraints. The Xtend environment would be further enhanced by incorporating the following features from SPG:

- Many of the relationships between a graphical element and its components are concerned with the location and size of the components relative to the element (e.g., the positions of axes within a scatterplot, the positions of tick-marks and labels within an axis, ...). Such relationships are typically not difficult to specify, but the specification is often tedious (e.g., to position a rectangular data symbol relative to a data location requires specifying four constraints for the left, bottom, width, and height of the rectangle). The layout mechanism developed in SPG (Chapter 4) provides a much more convenient and efficient means for describing such relationships.

- A very useful feature of SPG was the ability to specify a unit for values so that, for example, the location of a tick-mark could be specified in data coordinates and its length could be specified in inches. This feature should be included in an ideal system.

Such a system would provide a very powerful and fertile environment for data analysis. The ability to easily create statistical graphics which are not constrained by traditional graphical structures would promote the development of new graphical techniques. It would also encourage the choice of data representation to be driven by the data; rather than having to choose the most appropriate type of graph from a predefined set of options, the data analyst could construct customised data representations which are ideally suited to the exploration and presentation of specific data sets.
6.9 References


Appendix A

An introduction to
Object-Oriented
Programming
in Common Lisp

A.1 Programming in Lisp

Programming typically involves three steps. First of all, a source file is created containing the program commands in a high-level language (such as C, Pascal, or Lisp). Secondly, the source file is compiled into an executable file, which means that it is translated into a language-independent low-level representation that the computer will understand. Finally the executable file is run, which means that the computer executes the program commands.

The archetypal programming challenge is the "hello world" program, which prints the message "hello world" on the computer screen. In a typical C programming environment in UNIX, this program would be created as follows:

1. create a source file, say mysource.c, which contains the code:

```c
#include <stdio.h>
main()
{
    printf("hello world\n");
}
```

2. compile the source file to produce an executable file, say myexec, by typing:

```bash
gcc -o myexec mysource.c
```

3. run the program by typing:

```bash
myexec
```
In this case, the C environment only provides a compiler gcc. There are much more sophisticated environments (especially in the Macintosh or Microsoft Windows environments) which provide, for example, powerful text editors for creating the source file in addition to the compiler, but this only changes how the three steps are performed.

The source file for a program consists of a number of commands (e.g., `printf("hello world\n");`). These are typically organised into `functions` or `procedures`, which are just collections of commands. For example, the following C code defines a function to generate the mean of a set of values:

```c
double mean(double values[], int n)
{
    double sum = 0;
    for (i=0; i<n; i++)
    {
        sum += values[i];
    }
    return sum / n;
}
```

Within a function, commands are executed in the order they appear. For example, the mean function sets the value of `sum` to zero, performs a loop `n` times to accumulate the sum of the values, and returns the sum of the values divided by the number of values.

Programs have an `entry point` to determine which function should be executed first (e.g., in C, the entry point for a program is the `main` function).

Most Lisp environments work a little differently to this. For a start, there is usually an interactive interpreter, which allows the programmer to evaluate individual commands without having to write a complete source file and compile it into an executable file. Secondly, every function is a complete program, which can be executed on its own or called from another function; there is no special entry-point function.

For example, the `cmucl` environment provides a command line for entering Lisp commands to be evaluated. The "hello world" program can be run simply by typing the following command at the CMUCL command-line:

```lisp
(print "hello world")
```

The command is interpreted (i.e., the programmer does not need to compile the command in order to run it) and the `print` command is run as a program in itself (without having to be embedded within a special `main` function). This makes it very easy and quick to experiment with and test code. It also promotes the development of programs via small, simple code fragments which can be properly tested and made stable before incorporating them into larger, more complex programs.

The problem with interpreting commands is that it is slower than running compiled commands. Consequently, it is possible (usually once the code has been made to work) to perform the normal three-step process to generate compiled executable code. For example, the "hello world" program could be created as follows:

1. create a source file, say `mysource.lisp`, which contains the code:

```lisp
(defun my-hello-world ()
    (print "hello world")
)
```
2. within the Lisp environment, compile the source file to produce a compiled file, say mysource.sparcf, by typing:

```
(compile-file "mysource.lisp")
```

3. within the Lisp environment, load the compiled file and run the program by typing:

```
(load "mysource.sparcf")
(my-hello-world)
```

### A.1.1 Lisp Syntax

Lisp commands are generally of the form:

```scheme
(function-name arg-1 arg-2 ...)
```

In the "hello world" example, print is the function-name and "hello world" is the argument. This syntax can take some getting used to, particularly when the arguments have an implicit ordering or when the arguments are further Lisp expressions. For example, the difference between four and two is obtained from:

```scheme
(- 4 2)
```

and the command to calculate the mean from a set of data containing three 1's, four 2's, and one 3 requires:

```scheme
(/ (+ (* 1 3) (* 2 4) (* 3 1)) (+ 3 4 1))
```

In standard form, this would be somewhat more familiar:

```scheme
((1*3 + 2*4 + 3*1) / (3 + 4 + 1))
```

### A.1.2 Lisp and Lists

The list is a fundamental data type in Lisp (lisp actually stands for LISt Processing). Lisp provides many commands which make it very easy to create and manipulate lists. The most important of these commands are:

- **list**: this command creates a list from the arguments. For example:

  ```scheme
  (list 1 2 3)
  >> (1 2 3)
  ```

- **car**: this command returns the first element in a list. For example:

  ```scheme
  (car (list 1 2 3))
  >> 1
  ```

- **cdr**: this command returns all of the elements in the list (as a list) except the first one. For example:

  ```scheme
  (cdr (list 1 2 3))
  >> (2 3)
  ```
A.1.3 Symbols

Most Lisp commands return a value. Symbols provide places to store these values between several commands. The value of a symbol is set using `setf` and the value of a symbol is returned if the symbol itself is evaluated. For example:

```
(setf my-symbol 1)
my-symbol
>> 1
```

A.1.4 Special forms

Some Lisp commands have a special syntax. A couple of important examples are given below.

- **dotimes** this command repeats a command or series of commands n times:

  ```lisp
  (dotimes (i n)
    some-lisp-commands)
  ```

- **if** this command performs one set of actions if the `condition` is true and another set of commands if it is false:

  ```lisp
  (if condition
    commands-if-true
    commands-if-false)
  ```

- **let** this command performs a set of actions with a set of symbols locally defined:

  ```lisp
  (let ((local-symbol-1 value-of-local-symbol-1)
        (local-symbol-2 value-of-local-symbol-2)
         ...
    some-lisp-commands)
  ```

- **defun** this command allows the programmer to define a new command:

  ```lisp
  (defun function-name (argument-list)
    some-lisp-commands)
  ```

A.1.5 Keyword Arguments

A Lisp function has zero or more required arguments. When the function is called, a value must be provided for each required argument. In addition, a Lisp function can have a number of keyword arguments. A value does not have to be supplied for a keyword argument; if no value is supplied, the keyword argument takes a default value (specified when the function was defined) or the value `nil`. A value is supplied for a keyword argument by specifying the name of the keyword argument (preceded by a `:`) and a value. The following example demonstrates how a new command is defined with keyword arguments and how it can be called:
A.2 CLOS: Object-Oriented Programming in Common Lisp

The four basic elements of the CLOS implementation of OOP are classes, instances, generic functions, and methods.

A.2.1 Classes and Instances

A class is a description of a data structure. The description consists of a name for the class plus descriptions for a number of slots. Each slot description consists of a name for the slot plus information about an initial value for the slot to contain and functions to access this value. For example,

```
(defun my-func (arg-1 arg-2 &key arg-3 (arg-4 4))
  (print (list arg-1 arg-2 arg-3 arg-4)))

(my-func 1 2)
  >> (1 2 nil 4)
(my-func 1 2 :arg-3 3)
  >> (1 2 3 4)
(my-func 1 2 :arg-4 5)
  >> (1 2 nil 5)
(my-func 1 2 :arg-3 6 :arg-4 7)
  >> (1 2 6 7)
(my-func 1 2 :arg-5 8)
  >> error: unknown keyword
(my-func :arg-4 8)
  >> error: no value supplied for required args
```

A class serves as a template for creating instances. Every instance of a class is created with the slots that are described in the class. Each slot in the instance is a place where a value can be stored (in this sense, an instance is like a collection of symbols). The class can specify an initial value for a slot in an instance. For example, the expression :initform 2 in the definition of my-class means that every instance of my-class is created with a slot called slot-2 containing the value 2. Alternatively, the class can specify a keyword argument to be used to specify a value for a slot when an instance is created. For example, the expression :initarg :slot-1 in the definition of my-class means that every instance of my-class is created with a slot called slot-1 and a value for this slot must be specified using the keyword argument :slot-1 when the instance is created.

The function make-instance is used to create an instance from a class. For example,
APPENDIX A. AN INTRODUCTION TO
OBJECT-ORIENTED PROGRAMMING
IN COMMON LISP

(setf my-instance (make-instance 'my-class :slot-1 1))
creates an instance of my-class and stores it in the symbol my-instance. The
object thus created has a slot called slot-1 containing the value 1 and a slot called
slot-2 containing the value 2.
The class definition also describes functions which may be used to access the
values of slots in an instance. The expression :reader my-slot-2 in the definition
of my-class means that the function my-slot-2 can be used to get the value
contained in slot-2 of any instance of my-class. For example,

(my-slot-2 my-instance)
>> 2

The expression :accessor my-slot-1 in the definition of my-class means that
the function my-slot-1 can be used to get and set the value contained in slot-1
of any instance of my-class. For example,

(my-slot-1 my-instance)
>> 1

(setf (my-slot-1 my-instance) "1")
(my-slot-1 my-instance)
>> "1"

Subclasses, Superclasses, and Inheritance

When a new class is defined, it can be derived from an existing class. The new class
is called a subclass of the existing class, which in turn is known as the superclass of
the new class. The subclass inherits the slots of the superclass in addition to any
slots that it defines itself. For example,

(defclass my-other-class (my-class)
  ((slot-3 :initarg :slot-3
            :accessor my-slot-3)))
defines a new class called my-other-class which is derived from my-class. In-
stances of my-other-class will have three slots: slot-1 for which a value must be
provided (inherited from my-class); slot-2 which always has the value 2 (inher-
ited from my-class); and slot-3 for which a value must be provided (specific to
my-other-class). For example,

(setf instance-of-my-other-class
  (make-instance 'my-other-class :slot-1 1 :slot-3 3))

(my-slot-2 instance-of-my-other-class)
>> 2

A.2.2 Generic Functions and Methods

Just as a class is a description of a data structure, a generic function is a description
of a function. The definition of a generic function consists of the name of the generic
function plus a specification of the function's parameters. For example,

(defun generic identify-yourself (the-object))
defines a generic function called \texttt{identify-yourself} with a single required parameter (the object that should identify itself).

Just as classes are used as templates to create instances, generic functions are used as templates for methods. A method is a function with the same name and parameter list as a generic function. For example,

\begin{verbatim}
(defmethod identify-yourself ((the-object my-class))
  (print "I'm an instance of my-class")
\end{verbatim}

creates a method for the generic function \texttt{identify-yourself}. The expression \texttt{(the-object my-class)} means that this method will only work for objects which are instances of \texttt{my-class} (this method is \textit{specialised} for \texttt{my-class}).

Methods are inherited in a similar way to slots. This means that, if a method is defined for \texttt{my-class} then it will work for all instances of \texttt{my-class} and for all instances of any class that is derived from \texttt{my-class} (e.g., instances of \texttt{my-other-class}). For example,

\begin{verbatim}
(identify-yourself my-instance)
  >> "I'm an instance of my-class"

(identify-yourself instance-of-my-other-class)
  >> "I'm an instance of my-class"
\end{verbatim}

Generic functions are useful for obtaining different results from the same function call, based upon the nature of the object that the function is called on. For example,

\begin{verbatim}
(defclass another-class ()
  (setf another-instance (make-instance 'another-class))
  (defmethod identify-yourself ((the-object another-class))
    (print "I'm an instance of another-class")
  (identify-yourself another-instance)
    >> "I'm an instance of another-class"
\end{verbatim}

It is an error to call a function with a parameter for which there is no method.

\textbf{keyword arguments to methods}

The argument list for a method has to have the same required arguments as its generic function. A method also must have every keyword argument that its generic function has, however, the method may also have additional keyword arguments that the generic function does not have. For example,

\begin{verbatim}
(defgeneric a-func (required &key key-1))
(defmethod a-func ((required my-class) &key key-1)
  ...)
(defmethod a-func ((required another-class)
  &key key-1 key-2)
  ...)
\end{verbatim}
Bibliography


Index

3D graphs, 12, 39, 48
   selecting, see picking
   viewing, 20
   vs. 2D graphs, 14, 15, 48–49
accessing plot components
   from command-line, 50, 90, 164
   with mouse, see picking
accidental inheritance, 106, 175–176
Antelope, 169
arbitrary graph structure, 19, 52, 66, 167
   in Quail, 5
arbitrary graph structure, 43
arranging graphs, see layouts
automatically updating, 21, 74, 117
CLOS, 23, 115, 183
CML, see Common Lisp
combining objects, 52, 171, 172, see
   also incremental graphics
Common Lisp, 23, 68, 115
Common Lisp Object System,
   see CLOS
components
   of graphs, see hierarchies
constraints, 68
   and extensibility, 92
   and inheritance, 74
   in Antelope, 169
   in KR, 69
   in Pictor, 6, 169
coordinate systems
   in R, 159
   in S, 110
   in Simplisp, 20
   in SPG, 113, 135
   in Xtend, 65, 96
customisation, 2, 164
   and extensibility, 102
   limiting, 166, 171
display lists, 150, 152, 158

vs. automatic updating, 164
dragging
   in 3D, 48
editing graphs, see customisation
   equations, see mathematical annotation
event-handling, 150, 151
exploratory data analysis, 8, 56, 177
extensibility, 2, 62, 93, 101, 168
   in Quail, 5, 171
   in XlispStat, 8, 170
formulas, see mathematical annotation
Garnet, 68
graphical statistics, 1
hierarchies, 26, 165
   and extensibility, 171
   and picking, 38, 51
   in Pictor, 6
images
   in Simplisp, 20
   in Xtend, 65
incremental graphics, 62, 102
   and retained graphics, 168
inheritance
   accidental, see accidental inheritance
   in object-oriented programming, 184
   in Xtend, 80
   rules of, 74
interactive graphics, 103, 168, 172, 175,
   see also incremental graphics
KR, 68–70
layouts, 109, 118, 128
INDEX

and extensibility, 172
in Pictor, 172
in R, 154
in S, 109
in Trellis, 110
linking, 5, 51, 169
Lisp, 179, see also Common Lisp

mathematical annotation, 137
modifying graphs, see customisation

object hierarchies, see hierarchies
object-oriented programming
advantages of, 23
and extensibility, 8, 170
in CLOS, 183
in KR, 68
OOP, see object-oriented programming

picking, 17, 37, 51–52
Pictor, 6
constraints, 169
layout of plots, 172
object hierarchies, 165, 167
pictures
in SPG, 113
presentation-quality output, 15
and flexibility, 3, 171

Quail, 5
extensibility, 171

R, 4
graphics engine, 145
mathematical annotation, 137
redrawing, see automatic updating
relationships, see constraints
resizing windows, see automatic updating, see display lists
retained graphics, 168, see also automatic updating

S, 4
layout of plots, 110
selecting, see picking
sharing information
between components, 66, 102
between graphs, 16, 50
S-Plus, see S
stand-alone graphics, 63, 173,
see also UNIX, pipes

XGobi, 8
static graphics, 4, 145
vs. automatic updating, 163
three-dimensional graphs,
see 3D graphs
trees, see hierarchies
units, 123, 156,
see coordinate systems
UNIX, 37
inter-process communication, 9,
63
pipes, 75

viewports
in Simplisp, 20
in SPG, 115
in Xtend, 65

XGobi, 8
stand-alone graphics, 173
XLispStat, 7
extensibility, 170