

PBSmodelling 2.68.4: User's Guide

Jon T. Schnute, Alex Couture-Beil, Rowan Haigh, and A.R. Kronlund

Fisheries and Oceans Canada
Science Branch, Pacific Region
Pacific Biological Station
3190 Hammond Bay Road
Nanaimo, British Columbia
V9T 6N7

2017

**User's Guide Revised from
Canadian Technical Report of
Fisheries and Aquatic Sciences 2674 (2006)**



Fisheries and Oceans
Canada

Pêches et Océans
Canada

Canada¹³¹¹

© Her Majesty the Queen in Right of Canada, 2017

Revised from Cat. No. Fs97-6/2674E ISSN 0706-6457

Last update: Jul 12, 2017

Correct citation for this publication:

Schnute, J.T., Couture-Beil, A., Haigh, R., and Kronlund, A.R. 2017. PBSmodelling 2.68.4:
user's guide revised from Canadian Technical Report of Fisheries and Aquatic Sciences
2674: viii + 82 p. Last updated Jul 12, 2017

TABLE OF CONTENTS

Abstract.....	iv
Preface.....	v
Update for Version 2.50	vi
Update for Version 2.60	vii
Update for Version 2.65	vii
Additional Notes	viii
1. Introduction	1
2. GUI Tools for Model Exploration	3
2.1. Example: Lissajous Curves	4
2.2. Window Description File.....	6
2.3. Window Support Functions	8
2.4. Internal Data	11
3. Functions for Data Exchange.....	12
4. Support Functions for Graphics and Analysis.....	14
4.1. Graphics Utilities	14
4.2. Data Management	15
4.3. Function Minimization and Maximum Likelihood.....	15
4.4. Handy Utilities	16
5. Functions for Project Management	17
5.1. Project Options	17
5.2. Project Management Utilities	18
6. Support for Lectures and Workshops	19
7. Examples	23
7.1. Random Variables	24
7.1.1. RanVars – random variables	24
7.1.2. RanProp – random proportions.....	25
7.1.3. SineNorm – sine normal	26
7.1.4. CalcVor – calculate Voronoi tessellations	27
7.2. Statistical Analyses	28
7.2.1. LinReg – linear regression	28
7.2.2. MarkRec – mark-recovery	29
7.2.3. CCA – catch-curve analysis	30
7.3. Other Applications.....	31
7.3.1. FishRes – fishery reserve	31
7.3.2. FishTows – fishery tows	32
References.....	33
Appendix A. Widget descriptions	35
Window	35
Grid	36
Menu	37
MenuItem	37
Button.....	38
Check.....	39
Data	40
Droplist.....	41
Entry.....	43
History	44
Image	45
Include	46

Label	46
Matrix	47
Notebook.....	49
Null	51
Object.....	51
Progressbar	53
Radio.....	55
Slide	56
SlidePlus	57
Spinbox	58
Table	59
Text	60
Vector.....	61
Appendix B. Talk Description Files	63
<talk> ... </talk>	63
<section> ... </section>	64
<text> ... </text>	64
<file> ... </file>	64
<code> ... </code>	65
Appendix C. Building PBSmodelling and Other Packages	67
C.1. Installing Required Software.....	67
C.2. Building PBSmodelling	69
C.3. Creating a New R Package	70
C.4. Embedding C Code	72
Appendix D. PBSmodelling Functions and Data.....	77
D.1. Objects in PBSmodelling	77
D.2. PBSmodelling Manual	79

LIST OF TABLES

Table 1. Lissajous project files	4
Table 2. R source code with GUI definition strings	8
Table 3. Data file in PBS format.....	12
Table 4. Talk description file swisstalk.xml	21
Table C1. C representations of R data types.....	73
Table C2. .C() example in PBSty	74
Table C3. .Call() example adapted from PBSty	75

LIST OF FIGURES

Figure 1. Tangled relationships among computer model components	2
Figure 2. GUI organization of computer model components	2
Figure 3. Lissajous GUI	5
Figure 4. Lissajous graph.....	5
Figure 5. GUI generated by presentTalk from swisstalk.xml	22
Figure 6. RanVars GUI and density plot.....	24
Figure 7. RanProp GUI and pairs plot for Dirichlet	25
Figure 8. SineNorm GUI and plot	26
Figure 9. CalcVor GUI and tessellation plot	27
Figure 10. LinReg GUI and regression plot	28
Figure 11. MarkRec GUI and density plots	29
Figure 12. CCA GUI and parameter pairs plot	30

Figure 13. FishRes GUI and population time series31
Figure 14. FishTows GUI and simulated tow tracks32

Abstract

Schnute, J.T., Couture-Beil, A., Haigh, R., and Kronlund, A.R. 2017. PBSmodelling 2.68.4: user's guide revised from Canadian Technical Report of Fisheries and Aquatic Sciences 2674: viii + 82 p. Last updated Jul 12, 2017.

This report describes the R package `PBSmodelling`, which contains software to facilitate the design, testing, and operation of computer models. The initials PBS refer to the Pacific Biological Station, a major fisheries laboratory on Canada's Pacific coast in Nanaimo, British Columbia. Initially designed for fisheries scientists, this package has broad potential application in many scientific fields. `PBSmodelling` focuses particularly on tools that make it easy to construct and edit a customized graphical user interface (GUI) appropriate for a particular problem. Although our package depends heavily on the R interface to Tcl/Tk, a user does not need to know Tcl/Tk. In addition to GUI design tools, `PBSmodelling` provides utilities to manage projects with multiple files, write lectures that use R interactively, support data exchange among model components, conduct specialized statistical analyses, and produce graphs useful in fisheries modelling and data analysis. Examples implement classical ideas from fishery literature, as well as our own published papers. The examples also provide templates for designing customized analyses using other R packages, such as `PBSmapping`, `PBSddesolve`, `odesolve` (now `deSolve`), and `BRugs`. Users interested in building new packages can use `PBSmodelling` and a simpler enclosed package `PBStry` as prototypes. An appendix describes this process completely, including the use of C code for efficient calculation.

Preface

After working with fishery models for more than 30 years, I've used a great variety of computer software and hardware. Currently, the free distribution of R (R Development Core Team 2011a) provides an excellent platform for software development in an environment designed to support multiple computers and operating systems. Furthermore, an associated network of contributed packages on the Comprehensive R Archive Network ([CRAN](http://CRAN.R-project.org/)) gives access to a wealth of algorithms from many users in various fields. This disciplined system allows users, like the authors of this package, to distribute software that extends the utility of R in new directions.

Previously I've used software in Basic (Schnute 1982), Fortran (Mittertreiner and Schnute 1985), Pascal, C, and C++ to implement ideas in published papers. Usually this software goes stale in time, due to minimal documentation, changing operating systems, the lack of portable libraries, and many other factors. Because R includes a rich library of statistical software that operates on multiple platforms, my colleagues and I can now distribute software that actually works when other people try it. The user community includes us, because we often find that we can't remember how to operate our own software after a few weeks or months, let alone years. Although writing a good R package requires considerable effort, the result often pays off in portability, communication, and long term usage.

PBSmodelling tries to accomplish several goals. First, it anticipates the need for model exploration with a graphical user interface, a so-called GUI (pronounced *gooey*). We make this easy by encapsulating key features of Tcl/Tk into convenient tools fully documented here. A user need not learn Tcl/Tk to use this package. Everything required appears in Appendix A. You might want to start by running the function `testWidgets()`. Co-author Rowan Haigh likes the subtitle: "modelling the world with gooey substances."

Second, we want to demonstrate interesting analyses related to our work in fishery management and other fields. The function `runExamples()` illustrates some of these, as described further in Section 7. The code for all of them appears in the R library directory `PBSmodelling/examples`. We demonstrate the power of other R packages, such as `BRugs` (to perform Bayesian posterior sample with the application `WinBUGS`), `odesolve` (now `deSolve`, to solve differential equations numerically), `PBSddesolve` (to solve delay differential equations), and `PBSmapping` (to draw maps and perform spatial analyses).

Third, `PBSmodelling` serves as a prototype for building a new R package, as summarized in Appendix B. We illustrate two methods of calling C code (`.C` and `.Call`), and discuss many other technical issues encountered while building this package. The functions `compileC` and `loadC` (added in 2008) give direct support for dynamically adding C functions to the working R environment.

Finally, to use R effectively, we've found it convenient to devise a number of "helper" functions that facilitate data exchange, graphics, function minimization, and other analyses. We include these here for the benefit of our users, who may choose to ignore them. We hope that `PBSmodelling` inspires interest in interactive models that demonstrate applications in many fields.

As with our earlier package `PBSmapping`, Rowan and I employed a bright student who could learn quickly and implement creative ideas. Dr. Jim Uhl (Computing Science) and Dr. Lev Idels (Mathematics), both from Malaspina University-College (MUC) here in Nanaimo, drew my

attention to the student Alex Couture-Beil, who has strong credentials in both fields. Rowan and I gave him a few initial specifications, and he quickly got ahead of us by extending our ideas in new and useful directions. This process continued in 2008, when we employed Anisa Egeli, another bright student from MUC. The current version of `PBSmodelling` represents the result of an evolutionary process, as we experimented with design concepts that would support our modelling goals. Users familiar with the earlier versions (starting with 0.60, posted on CRAN in August, 2006) may need to revise their code slightly to make it work with this version.

Since 1998, I have maintained a formal relationship with the Computing Science Department at MUC (now named Vancouver Island University – VIU), where I find kindred spirits in developing projects like this one. I particularly want to thank Dr. Jim Uhl for his suggestions and support on this project. Conversations with Dr. Peter Walsh have also stimulated my interest in the theory and application of computing science.

Fishery management depends on models with a great range of complexity, starting from some fairly simple ideas. Unfortunately from a coding perspective, “industrial strength” models can’t run exclusively in R. Algorithms with high computational requirements don’t run fast enough in R for practical application, due to interpretive code and other technical limitations. Examples in `PBSmodelling` often illustrate ideas at the simple end of the spectrum, although the package can certainly be used to manage external software designed to deal with greater complexity. The current version assists users in writing C code that can dramatically speed model performance.

Scientifically, I like to work from both ends of the spectrum. The behaviour of a complex model sometimes mimics a much simpler model, and it helps to become well versed in some of the simpler cases. I appreciate the motto of Canadian storyteller and humorist Stuart McLean, who hosts a CBC radio broadcast [The Vinyl Cafe](#), “We may not be big, but we’re small.”

Jon Schnute, December 2006; revised October 2008.

Update for Version 2.50

Our colleagues Rob Kronlund, Sean Cox, and Jaclyn Cleary used this package extensively for research on Management Strategy Evaluation. Their experiences led them to suggest a number of significant improvements. We thank Rob for providing written specifications and financial resources to implement their ideas. `PBSmodelling` now includes new widgets (`droplist`, `table`, `spinbox`, `include`), bug fixes, and other improvements that give users even greater control over GUIs designed for exploring and demonstrating analyses with R. Alex Couture-Beil, who now pursues graduate studies at Simon Fraser University, added the new programming code that contributes to this significant upgrade.

This update also includes greatly enhanced versions of our functions to support project development (Section 5) and interactive lectures (Section 6). Our colleague Andrew Edwards at PBS assisted this work with funding that allowed us, once again, to engage our intrepid graduate student Alex. Our code now includes modest use of S4 classes, such as the new `PBSoptions` class (Section 5.1). Furthermore, we now use XML scripts in the *talk description files* that enable users to give dynamic presentations about analyses in R (Section 6). Users who employed our function `presentTalk` in the past will need to revise their description files to operate with this update.

The scope of our R packages has grown considerably over the last few years. Thanks to prodding from Alex, we now use Google Code web sites for all our [PBS projects](#). In particular, this update supports our new package [PBSadmb](#). It allows R users to tap into an open source package that can handle “industrial strength” assessment problems. We have a version that should soon be ready for posting on CRAN.

Jon Schnute, October 2009

Update for Version 2.60

Open source software often benefits greatly from unanticipated suggestions and participation by the user community. John Chambers (2008, p. 10), who designed the S language that underlies R, describes this phenomenon as “a cause for much gratitude and not a little amazement.” Eric Raymond (2000), speaking from his own experience with open source projects, puts it this way: “Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.” This is Raymond’s sixth lesson in a series of fundamental observations. The tenth is: “If you treat your beta-testers as if they’re your most valuable resource, they will respond by becoming your most valuable resource.”

In the previous update for version 2.50, I mentioned my long-time friend, colleague, and co-author Rob Kronlund. I was delighted by his interest in applying `PBSmodelling` to evaluate fishery management strategies (Kronlund et al. 2010). He quickly identified problems and helped design extensions that have greatly benefited the current version. In addition to many small changes and bug fixes, it includes three new widgets: notebook, image, and progressbar. Very significantly, the new notebook widget implements tabbed windows, in which distinct tabs correspond to different aspects of the GUI. This can help organize material for user interaction, and it potentially reduces the required size of a GUI so that it can fit on the small screens now popular on ultraportable computers.

We are delighted and grateful to welcome Rob as a co-author of `PBSmodelling`. He has certainly become a valuable resource, although I’m not sure we always gave him the quality treatment suggested by Eric Raymond.

Jon Schnute, March 2010

Update for Version 2.65

Version 2.60 fixes an unfortunate bug in the grid widget. In earlier versions, the arguments `nrow` and `ncol` were reversed when `byrow=FALSE`. We realized that we had programmed around this bug in our previous window description files. Consequently, after the bug was fixed, we needed to make changes to every grid widget with `byrow=FALSE`. (These were relatively uncommon.) We encourage users of version 2.60 to check their own historical description files for this potential problem.

Starting in version 2.65, we implement accessor functions to comply with [CRAN Repository Policy](#), specifically “packages should not modify the global environment (user’s workspace).” This change should largely be invisible to users; however, be aware that control objects like `.PBSmod` are now located in a temporary environment called `.PBSmodEnv`. The accessor functions `tget`, `tput`, `tcall`, and `tprint` communicate with this environment, and we

recommend that users become familiar with these functions (see the `tget` help file in R). Additionally, the accessor functions are implemented in all of the PBS packages that depend on `PBSmodelling`, invariably as wrappers to the original functions. For example, in `PBSadmb` the temporary workspace is called `.PBSadmbEnv` and the accessor functions are called `atget`, `atput`, `atcall`, and `atprint`, using the prefix “a” to designate `PBSadmb` wrappers to the `PBSmodelling` accessor functions. Similarly, we employ the wrapper prefixes “t” for accessing `.PBStoolEnv` in `PBStools`, “x” for `.PBSmapxEnv` in `PBSmapx`, and “d” for `.PBSdataEnv` in `PBSdata`.

Rowan Haigh, March 2013

Additional Notes

We maintain development and distribution sites for a suite of PBS software packages on [GitHub](#) (Google Code is now defunct), including one for `PBSmodelling`. Four of these packages appear on CRAN while the others are perhaps too specialised for mass appeal. The latter packages, however, contain interesting and complex functionality that individuals are welcome to explore.

Rowan Haigh, March 2013

This User’s Guide is based on a version originally published in 2006 (Schnute et al. 2006). Since that time, the software has undergone numerous changes due to shifting CRAN requirements, but the functionality has remained largely unchanged. The original report appended complete technical documentation for every `PBSmodelling` object (compiled from `.Rd` files written for the R documentation system) at the end of the final Appendix. This User’s Guide no longer reports these details as they are readily available using R’s help menu system.

Some sections of the report have not been reviewed thoroughly for a while (e.g., Creating a new package). Please let the authors know if you wish to see revisions for specific instructions, or suggest revisions based on your experience.

Throughout the document, we use the delimiter symbols « » to designate a GUI menu item selection or a GUI action such as pushing a button or selecting a check box or radio button. These symbols are not to be confused with *talk description file* tags (<>) used later (Section 6).

Rowan Haigh, July 2017

Version of R at time of User Guide update: R-3.4.1

1. Introduction

This report describes software to facilitate the design, testing, and operation of computer models. The package `PBSmodelling` is distributed as a freely available package for the popular statistical program R (R Development Core Team 2011a). The initials PBS refer to the Pacific Biological Station, a major fisheries laboratory on Canada's Pacific coast in Nanaimo, British Columbia. Previously, we produced the R package `PBSmapping` (Schnute et al. 2004), which draws maps and performs various spatial operations. Although both packages (which can run separately or together) include examples relevant to fishery models and data analysis, they have broad potential application in many scientific fields.

Computer models allow us to speculate about reality, based on mathematical assumptions and available data. The full implications of a model usually require numerous runs with varying parameter values, data sets, and hypotheses. A customized graphical user interface (or GUI, pronounced “gooey”) facilitates this exploratory process. `PBSmodelling` focuses particularly on tools that make it easy to construct and edit a GUI appropriate for a particular problem. Some users may wish to use this package only for that purpose. Other users may want to explore the examples included, which demonstrate applications of likelihood inference, Bayesian analysis, differential equations, computational geometry, and other modern technologies. In constructing these examples, we take advantage of the diversity of algorithms available in other R packages.

In addition to GUI design tools, `PBSmodelling` provides utilities to support data exchange among model components, conduct specialized statistical analyses, and produce graphs useful in fisheries modelling and data analysis. Examples implement classical ideas from fishery literature, as well as our own published papers. The examples also provide templates for designing customized analyses using the R packages discussed here. In part, `PBSmodelling` provides a (very incomplete) guide to the variety of analyses possible with the R framework. We anticipate many revisions, as we find time to include more examples.

`PBSmodelling` depends heavily on Peter Dalgaard's (2001, 2002) R interface to the Tcl/Tk package (Ousterhout 1994). This combines a scripting language (Tcl) with an associated GUI toolkit (~~Tk~~). We simplify GUI design with the aid of a “window description file” that specifies the layout of all GUI components and their relationship with variables in R. We support only a subset of the possibilities available in Tcl/Tk, but we customize them in ways intended specifically for model design and exploration (Appendix A). A user of `PBSmodelling` does not need to know Tcl/Tk.

Computer models typically involve a variety of components, such as code, data, documentation, and a user interface. Figure 1 illustrates the tangled relationships that sometimes accompany computer model design. `PBSmodelling` allows the GUI to become a device for organizing components, as well as running and testing software (Figure 2). The project might involve other applications, as well as R itself. In addition to its interactive role, the GUI becomes an archival tool that reminds the developer how components, functions, and data tie together. Consequently, it facilitates the process of restarting a project at a future date, when details of the design may have been forgotten.

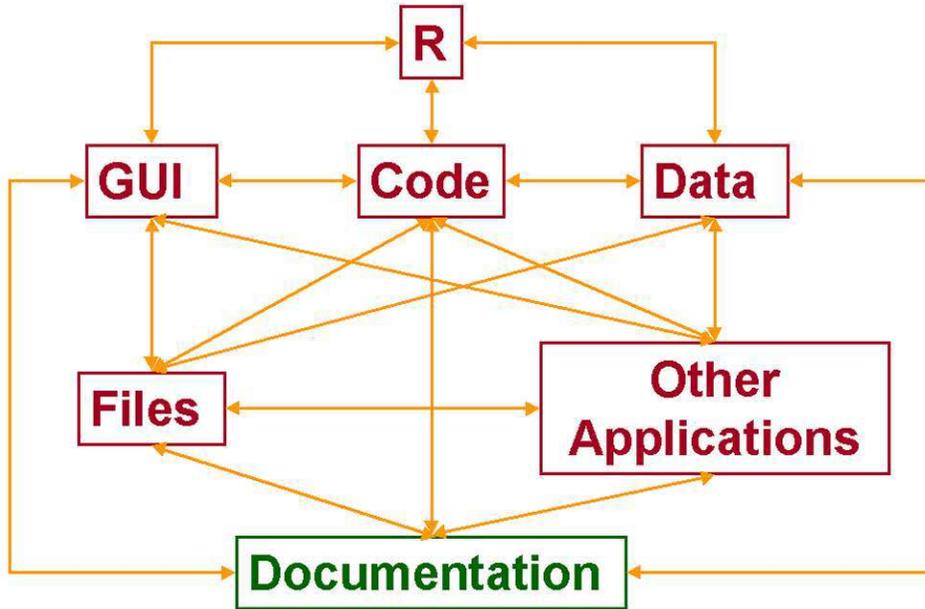


Figure 1. Tangled relationships among computer model components.

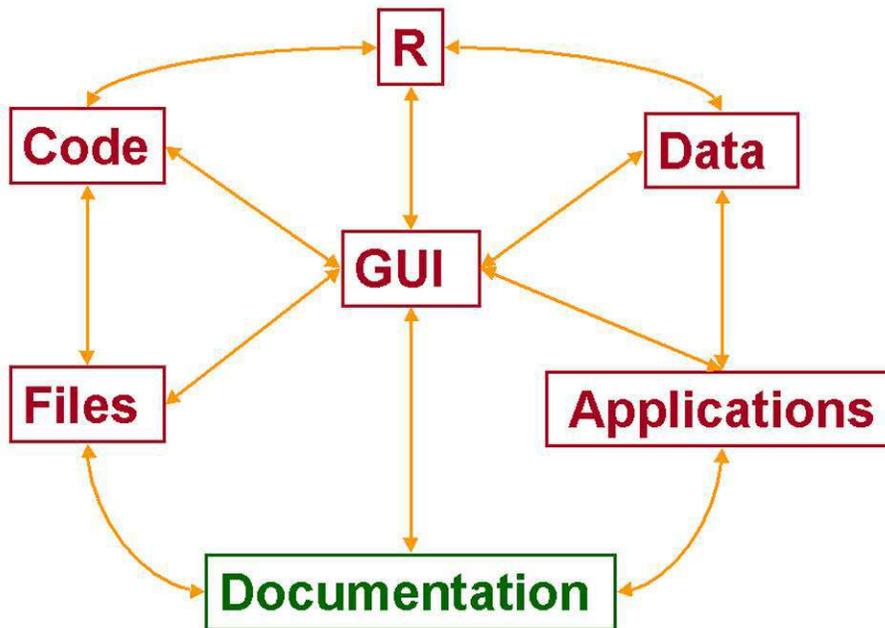


Figure 2. Computer model components organized with a graphical user interface (GUI).

In PBSmodelling, project design normally begins with a text file that describes the GUI. Additional files may contain code for R and other applications, which sometimes require languages other than R. For example, the R BRugs package (to perform Bayesian inference using Gibbs sampling) requires a file with the intended statistical model, written in the language of a separate program WinBUGS. In other contexts, a user might write C code to get acceptable performance from model components that require extensive computer calculations. This code might be compiled as a separate program or linked directly into a customized R package.

Section 2 of this report describes the process of designing a GUI to operate a computer model. Components can share data through text files in a specialized “PBS format” presented in Section 3. These correspond naturally to `list` objects within R. Section 4 describes additional tools for customized graphics and data analysis. Sections 5 and 6 discuss tools developed in 2008 for managing projects (like C code development) and writing lectures that use R interactively. In Section 7, we highlight briefly some of the examples in our initial release, although we expect the list to expand in future versions. This guide explains the context and general purpose of all functions in `PBSmodelling`. Consult the help files for complete technical details.

Appendix A gives the complete syntax for all visual components (called *widgets*) available for writing a window description file to construct a customized GUI. Appendix B provides syntax detail for talk description files. Appendix C describes the process of building `PBSmodelling` in a Windows environment. A simple enclosed package `PBstry` gives a prototype for building any R package, including the use of C code to speed calculations. Appendix D shows the help files included with the package.

To use `PBSmodelling`, run R and install the package from the R GUI (click «Packages», «Install package(s)... », select a mirror, and choose `PBSmodelling` from the list of packages). Windows users can also obtain an appropriate compressed file from the authors of this report or directly from the CRAN web site.

The R GUI normally runs as a Multiple Document Interface (MDI), in which child windows like the R console and graphics screens all appear within the GUI itself and a menu item can be used to tile the sub-windows. Unfortunately, in this configuration, windows generated by Tcl/Tk sometimes disappear mysteriously when an application runs. They can be recovered by clicking the appropriate “TK” icon on the taskbar. You can avoid this problem by using the Single Document Interface (SDI), in which the operating system manages all R windows (console, graphics, Tcl/Tk, etc.) independently on the desktop. Set this configuration by running the R GUI, choosing the menu items «Edit» and «GUI Preferences», and then selecting and saving the SDI option. Alternatively, go to the master configuration file `Rconsole` in the `\etc` subdirectory of the R installation, and use a text editor to select the option `MDI = no`.

2. GUI Tools for Model Exploration

The practical task of writing appropriate code for the R Tcl/Tk package can sometimes become daunting, particularly if the GUI window requires extensive design and change. For a restricted set of TK components (called *widgets*), `PBSmodelling` makes it much easier to design and use GUIs for exploring models in R. A user needs to supply two key parts of a GUI-driven analysis:

- a *window description file* (an ordinary text file) that completely specifies the desired layout of widgets and their relationship with functions and variables in R, and
- R code that defines relevant functions, variables, and data.

This section begins with an example to illustrate the main ideas, and then gives complete details for constructing window description files that can be used to generate GUIs.

2.1. Example: Lissajous Curves

A [Lissajous curve](#), named after one of its inventors Jules-Antoine Lissajous, represents the dynamics of the system

$$x = \sin(2\pi mt), \quad y = \sin[2\pi(nt + \phi)], \quad (1)$$

where time t varies from 0 to 1. During this time interval, the variables x and y go through m and n sinusoidal oscillations, respectively. The constant ϕ , which lies between 0 and 2π , represents a cycle fraction of phase shift in y relative to x . We want to design a GUI that allows us to explore this model by plotting Lissajous curves (y vs. x) for various choices of the parameters (m, n, ϕ) . We also want to vary the number of time steps k and choose a plot that is either lines or points.

Table 1. Two text files associated with the “Lissajous Curve” project. The first gives a description of the GUI window used to manage the graphics. The second contains R code to draw a Lissajous curve.

File 1: LissajousCurve.txt

```
window title="Lissajous Curve"
vector length=4 names="m n phi k" \
  labels="'x cycles' 'y cycles' 'y phase' points" \
  values="2 3 0 1000"
radio name=ptype text=lines value="l" mode=character
radio name=ptype text=points value="p" mode=character
button text=Plot function=drawLiss
```

File 2: LissajousCurve.r

```
drawLiss <- function() {
  getWinVal(scope="L");
  tt <- 2*pi*(0:k)/k;
  x <- sin(2*pi*m*tt); y <- sin(2*pi*(n*tt+phi));
  plot(x,y,type=ptype);
  invisible(NULL); }
```

This analysis can be accomplished with the R code and window description file shown in Table 1. Assume that these two files reside in the current working directory and that PBSmodelling has been installed in R. Start an R session from this directory, and type the following three lines of code in the R command window:

```
> require(PBSmodelling)
> source("LissajousCurve.r")
> createWin("LissajousCurve.txt")
```

The first line assures that PBSmodelling is loaded, the second defines the function drawLiss for drawing Lissajous curves, and the third creates a window that can be used to draw curves corresponding to any choice of parameters. Figure 3 shows the resulting GUI window interface. When the «Plot» button is clicked, the curve in Figure 4 appears in the R graphics window. This corresponds to the default parameter values:

$$m = 2, n = 3, \phi = 0, k = 1000. \quad (2)$$

The GUI allows different Lissajous figures to be drawn easily. Simply change parameter values in any of the four entry boxes, and click «Plot».

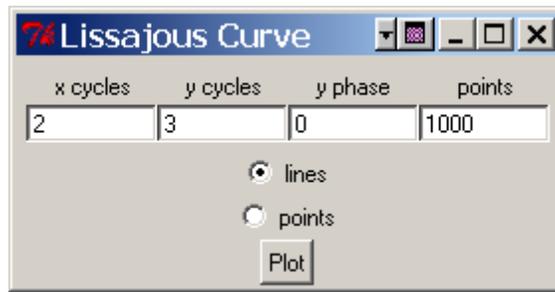


Figure 3. GUI generated by the description file *LissajousCurve.txt* in Table 1. It contains five widgets: the window titled “Lissajous Curve”, a vector of four entries, two linked radio buttons («lines» and «points»), and a «Plot» button.

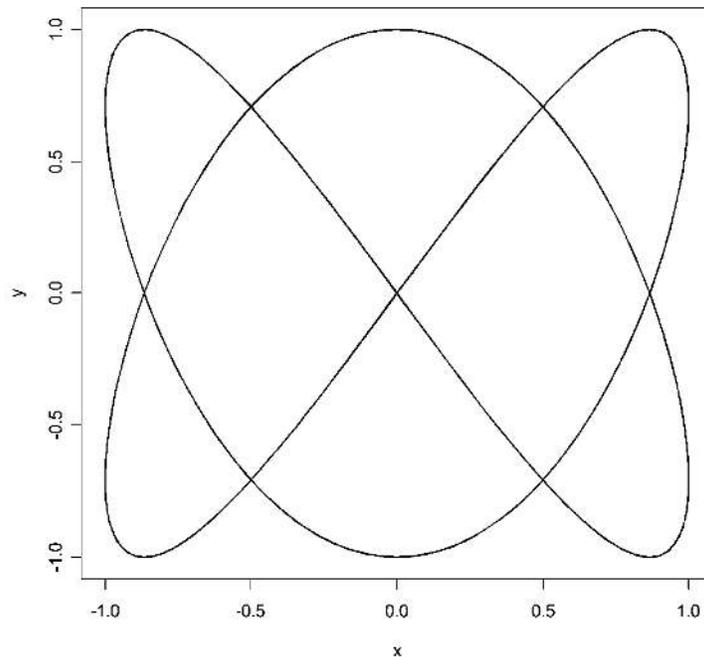


Figure 4. Default graph for the “Lissajous Curve” project, obtained by clicking the «Plot» button in Figure 3. The x variable goes through two cycles while the y variable goes through 3 cycles. A line graph is drawn through 1,000 points generated by the algorithm (1).

The window description file (Table 1) specifies a window titled “Lissajous Curve” with a vector of four entries. These correspond to quantities with the R variable names m , n , ϕ , and k . The corresponding window (Figure 3) will contain four entry boxes that allow these quantities to be changed. A label for each quantity emphasizes its conceptual role: the number of cycles for x or y , the phase shift for y , and the number of points plotted. Initial values correspond to those listed in (2). The backslash (\backslash) character indicates that a widget description (in this case, a vector) continues on the next line. A pair of radio buttons, both corresponding to an R variable named $ptype$, allow selection between «lines» and «points» when drawing the plot. The graph (Figure 4) is actually drawn (i.e., the R function `drawLiss` is called) when the user presses a button that contains the text “Plot”.

The file of R code (Table 1) implements the algorithm (1) for computing k points on a Lissajous curve. The function `drawLiss` has no arguments, but gets values of the R variables `m`, `n`, `phi`, `k`, and `ptype` from the GUI window via a call to the `PBSmodelling` function `getWinVal`. The argument `scope="L"` implies that these variables have local scope within this function only. (Another choice `scope="G"` would give the variables global scope by writing them to the user's global environment `.GlobalEnv`.)

2.2. Window Description File

A window description file currently supports the following widgets:

1. `window` – an entire new window;
2. `grid` – a rectangular block for placing widgets;
3. `menu` – a menu grouping;
4. `menuItem` – an item in a menu;
5. `button` – a button linked to an R function that runs a particular analysis and generates a desired output, perhaps including graphics;
6. `check` – a check box used to turn a variable on or off, with corresponding values `TRUE` or `FALSE`;
7. `data` – an aligned set of entry fields for all components of a data frame, where columns can have different modes;
8. `droplist` – an entry widget with a drop down list of values;
9. `entry` – a field in which a scalar variable (number or string) can be altered;
10. `history` – a device for archiving parameter values corresponding to different model choices, so that a “slide show” of interesting choices can be preserved;
11. `image` – a graphical widget that displays a GIF image file;
12. `include` – a pseudo widget which embeds a specified window description file within the current window description file;
13. `label` – a text label;
14. `matrix` – an aligned set of entry fields for all components of a matrix;
15. `notebook` – a widget comprised of pages that can be selected by tabs, where each page is visible when the corresponding tab is selected;
16. `null` – a blank widget that can occupy an empty space in a grid;
17. `object` – an aligned set of entry fields defined by an existing R-object (vector, matrix, or data frame);
18. `progressbar` – a progress indicator widget;
19. `radio` – one of a set of mutually exclusive radio buttons for making a particular choice;
20. `slide` – a slide bar that sets the value of a variable;
21. `slideplus` – an extended slide bar that also displays a minimum, maximum, and current value;
22. `spinbox` – an entry widget for a numeric value within a given range which can be changed with the up and down arrows;
23. `table` – a spreadsheet widget with scrollbars for large tabular data;
24. `text` – an entry box that supports multiple lines of text;
25. `vector` – an aligned set of entry fields for all components of a vector.

The description file is an ordinary text file that specifies each widget on a separate line. However, any one widget description can span multiple lines by using a backslash character (\) to indicate the end of an incomplete line. For example, the single line:

```
label text="Hello World!"
```

is equivalent to:

```
label \  
  text="Hello World!"
```

Meaningful indentation is highly recommended, but not compulsory. The three-line description of a vector widget in Table 1 illustrates a readable style.

Each widget has named arguments that control its behaviour, analogous to the named arguments of a function in R. Some (required) arguments must be specified in the widget description. Others (not required) can take default values. All widgets have a `type` argument equal to one of the 25 names above, although the word `type` can be omitted in the description file. Appendix A gives an alphabetic list of all these widgets, along with detailed descriptions of all arguments. As in calls to R functions, argument names can be omitted as long as they conform to the order specified in the detailed widget descriptions given below. Nevertheless, we recommend that all argument names be specified, except possibly the name `type`, which is always the first argument for each widget. Unlike R functions, where commas separate arguments, the arguments in a widget description are separated by white space.

In a description file, all argument values are treated initially as strings. In addition to specifying a line break, the backslash can be used to indicate five special characters: single quote `\'`, double quote `\"`, tab `\t`, newline `\n`, and backslash `\\`. If an argument value does not include spaces or special characters, then quotes around the string are not required. Otherwise, double quotes must be used to delineate the value of an argument. Some arguments can take a NULL argument value; quotes are used to differentiate between a NULL object, and the text value "NULL". Single quotes indicate strings nested within strings. For example, the vector in Table 1 has four labels specified by the string argument

```
labels="'x cycles' 'y cycles' 'y phase' points"
```

A hash mark (#) that is not within a string begins a comment, where everything on a line after the hash mark is ignored. As mentioned above, an isolated backslash (not part of a special character) indicates continuation onto the next line. A break can even occur in the middle of a string, such as the long label

```
label text="This long label with spaces \  
  spans two lines in the description file"
```

In this case, leading spaces in the second line are ignored, to allow meaningful formatting in the description file. Intentional spaces in a long string should appear prior to the backslash on the first line.

Although the `type` argument (like `vector`) for a widget can never be abbreviated, other arguments follow the convention used with named arguments in R function calls. For a given widget type, the available arguments can be abbreviated, as long as the abbreviations uniquely identify each argument. For example, the vector in Table 1 could be specified as:

```
vector len=4 nam="m n phi k" \  
  lab="'x cycles' 'y cycles' 'y phase' points" \  
  val="2 3 0 1000"
```

Unlike variable names in R, widget names and their arguments are not case sensitive. Some users may prefer to write all type variables in upper case or with an initial capital letter. For example, the names WINDOW, VECTOR, RADIO, and BUTTON could be used to emphasize the widgets in Table 1.

2.3. Window Support Functions

PBSmodelling includes functions designed to connect R code with GUI windows. Every window has a name argument (with default name=window), and windows with different names can coexist. Window names must use only letters and numbers; they cannot contain a period (dot) or any other punctuation. When running a program with multiple windows, only one window will be current (i.e., selected by the user) at any particular time. Normally, a user selects a window by clicking on it, but the function `focusWin` allows program control of the window currently in focus. Thus, activity in one window might be used to shift the focus to another.

The function `createWin` uses a description file to generate one or more windows, where each window has a distinct name (perhaps the default) taken from the file. If a window with the specified name already exists, it will be closed before the new window is opened. When designing and testing a GUI, this feature ensures that a new version automatically replaces the previous one. The function `closeWin`, which takes a vector of window names, closes all windows named in the vector. With no arguments, `closeWin()` closes all windows that are currently open.

Although `createWin` normally builds a GUI from a description file, it will also accept a vector of strings equivalent to such a file. Thus, a file of R source code can define a GUI directly, without the need for a separate description file. Table 2 illustrates how this can be done in a simple case.

Table 2. A simple file of R source code with character strings that define a GUI. No separate window description file is required.

File: Simple.r

```
# window description strings
winStr=c(
  "window title=Simple",
  "entry name=n value=5",
  "button function=myPlot text=\"Plot sinusoid\"");

# function to plot a sinusoid
myPlot <- function() {
  getWinVal(scope="L");
  x <- seq(0,500)*2*n*pi/500;
  plot(x,sin(x),type="l"); };

# commands to create the window
require(PBSmodelling); createWin(winStr,astext=TRUE)
```

To see the character vectors equivalent to a given description file (say, `winDesc.txt`), type the R command:

```
scan("winDesc.txt",what=character(),sep="\n")
```

In particular, if the description file includes a backslash or double quote character, the corresponding R string must represent it as `\\` or `\"`, respectively. Despite this alternative of

embedding window descriptions in R source files, we recommend writing separate files to define GUIs, except perhaps for very simple models.

Internally, `PBSmodelling` converts a description file into a `list` object that is used to generate the corresponding GUI. The functions `compileDescription` and `parseWinFile` give lists that correspond to description files. Just as `createWin` can act directly on a character vector, it can also act on a suitably defined list, rather than a file. This feature makes it possible to replace a description file with R code that defines the corresponding list, although we recommend against this practice in most cases.

R programs need to share data with a GUI window. `PBSmodelling` provides seven functions that deal with values of R variables named in a description file:

```
getWinVal ..... returns values from the current window;
setWinVal ..... sets values in the current window;
getWinAct ..... returns all actions (to a maximum of 50) invoked in the current window;
setWinAct ..... adds an action to the action vector for the current window;
getWinFun ..... returns the names of all R functions referenced in the current window;
clearWinVal ..... clears global values associated with the current window;
updateGUI ..... updates the currently active GUI with values from R's memory.
```

Some models make use of a single parameter vector. In such cases the function `createVector` generates a GUI directly, without the need for a corresponding description file. We also offer a few “choosing” functions – `getChoice` and `chooseWinVal` – that invoke a prompting GUI offering string choices. The latter writes the choice to a variable in a GUI specified by the user.

After using `createWin` to produce a GUI, the functions `getWinVal` and `getWinFun` provide useful summaries of names declared in the current project. Furthermore, the function `getWinAct` provides a record of GUI actions taken by the user, starting with the most recent and working backwards. By default, the action associated with a widget is its type; for example a button has default `action=button`. In general, however, the description file could give a unique action name to each potential action, so that the vector would give an unambiguous record of user actions.

Alternatively, GUI widgets that support function arguments can take the `PBSmodelling` function `doAction`, which evaluates code specified as a string in the widget’s action argument. This code string can be a simple expression or a multi-line set of R-code. In essence, `doAction` allows the user to implement subroutines by clicking a widget (such as a button):

```
winStr=c("window title=\"doAction Demo\"",
"button text=\"See attached libraries\" width=30 pady=10 \
function=doAction action=\"x=search(); N=length(x); mess=paste(paste(pad0(1:N,2),x),
collapse=\"\n\"); resetGraph(); addLabel(.2,.5,mess,adj=0)\")
createWin(winStr,astext=TRUE)
```

Within the action string, substitute double quotation marks “...” with backtick characters `...`, and the function `doAction` will replace them with interpretable quotation marks. In most cases (not all), escaping the quotation marks “...” will also work.

The package provides a function called `selectFile` for opening and saving files to directories using a GUI menu. Earlier functions (`promptOpenFile`, `promptSaveFile`) remain

available, but should be deprecated in favour of `selectFile`. Files opened using programs external to R depend on file name extensions:

`openFile` opens a file using the default program for the file extension;
`setPBSext` overrides the default program associated with an extension;
`getPBSext` shows the overridden file extension and associated program.
`clearPBSext` clears file extensions added by `setPBSext`.

If a widget invokes the function `openFile`, the associated action should be the file name. By definition, `openFile` has the default argument `getWinAct()[1]`.

On a Windows platform, the native R function `shell.exec` (called by `openFile`) automatically chooses a default from the registry. For this reason, our distribution specifies an empty list: `getPBSext()` returns `list()`. The default can, however, be overwritten by specifying explicit list components, such as:

```
setPBSext('html',  
  "'c:/Program Files/Mozilla Firefox/firefox.exe" file://%f')
```

where `%f` denotes the file name in the string passed to the operating system. Unix platforms typically lack such generic file associations, and thus require a user to specify defaults this way.

`PBSmodelling` includes a history widget designed to collect interesting choices of GUI variables so that they can be redisplayed later, rather like a slide show. This widget has buttons to add and remove GUI settings from the current collection, to scroll backward and forward, and to clear all entries from the collection. Other buttons allow entire history files to be saved or loaded. The history widget defines and uses the list `PBS.history` in the global environment to store a saved history.

Normally, a user would invoke a history widget simply by including a reference to it in the description file. However, `PBSmodelling` includes some support functions for customized applications:

`initHistory` initializes data structures for holding a collection of history data;
`addHistory` saves the current window settings to the current history record;
`rmHistory` removes the current record from the history;
`backHistory` and
`forwHistory` move backward and forward between successive history records;
`firstHistory` and
`lastHistory` move to the first and last records in the history;
`jumpHistory` moves to a specified record in the history;
`exporthHistory` and
`importHistory` save and load histories from files;
`clearHistory` removes all records from the current collection.

The help file for `initHistory` shows an example that uses these functions directly.

Since version 2.50, we have incorporated additional functionality for GUIs stemming from experience using this package in multi-stakeholder workshops. In particular, a new widget called `notebook` now allows tabbed pages within one GUI, where each page can contain whatever grid and widget combination the programmer wishes to present. This removes the

problem of having too many GUIs on screen when situations require multiple inputs and outputs.

Other new widgets include drop lists and spin boxes (`droplist`, `spinbox`), scrollable objects for data with dimensions too large to fit on screen (`object`), a spreadsheet-like widget that can display and edit data in tabular format (`table`), an animated progress indicator (`progressbar`), and an image widget to add illustrations, logos, and other visual cues (GIF format only).

2.4. Internal Data

PBSmodelling uses the hidden list variable `.PBSmod` in a temporary working environment called `.PBSmodEnv` to store current settings and internal information needed to communicate with the `tcl/tk` interface. This variable is intended for exclusive use by PBSmodelling, and users should not alter or delete it while PBSmodelling is active. We include the material in this section for advanced users and developers interested in further details about the internal data used to manage GUI windows.

The list `.PBSmod` contains a named component for each open window, where the component name matches the window name. Recall that, if a window is not named explicitly, it receives the default `name=window`. In addition to window names, `.PBSmod` contains two other named components: `$.activeWin` and `$.options`. These names do not conflict with the window names, because the latter cannot include a dot (`.`). The `$.activeWin` component stores the name of the window that has most recently received user input. The `$.options` component saves key values of interest to PBSmodelling, such as a component `$openfile` with information that links programs to file extensions for the function `openFile`. See Section 2.3 for further information.

Any named component of `.PBSmod` that does not start with a dot stores information related to the corresponding window. Each window uses a list with the following named components:

- `widgetPtrs`
A list containing widget pointers. Each component has a name that matches widget name. Only widgets with a name argument and a corresponding `tk` widget will appear in this list.
- `widgets`
A list containing information from the window description file relevant to each widget. This list includes every widget that has a name or names argument. Widgets without names will never be referenced again after the window has been created; consequently, information about them is not stored for later usage.
- `tkwindow`
A pointer to the window created by `tkoplevel()`.
- `functions`
A vector of all function names referenced in the window description.
- `actions`
A vector containing action strings corresponding to the most recent user actions in the window, up to a maximum of 50. (The internal constant `.maxActionSize` sets this upper limit. See the file `defs.R` in the distribution source code.)

Users can explore the contents of `.PBSmod` with the R structure command `str`. Remember that an accessor function is also needed to “see” this object. For example, from the R console, type `runExamples()` and select the example “CalcVor”. Then type the command

`str(tcall(.PBSmod), 2)` to shows the list structure to a depth of 2. This reveals all the list components discussed above. Further details appear by exploring the structure to depths 3, 4, or more. Notice also how the contents change as different examples are selected.

The functions `getWinVal`, `setWinVal`, `getWinAct`, `setWinAct`, `getWinFun`, `getPBSext`, and `setPBSext` (discussed in Section 2.3) provide methods for manipulating and retrieving variables stored in `.PBSmod`. Use these, rather than direct access, to alter the internal data. Future design modifications to `PBSmodelling` might change the architecture for storing the data components, but the methods functions will continue to have their current effect.

Table 3. Sample data file for `PBSmodelling`. The function `readList` converts this file to a list object with six components: a scalar x , a logical vector y , two matrices (z , a), and two data frames ($b1$, $b2$). The matrix a is read by column, and $b1=b2$.

```
$x
0

$y
T F TRUE FALSE

$z
11.1 12.2 13.3 14.4
15.5 16.6 17.7 1.88e+01

$a
$$matrix ncol=2 byrow=FALSE colnames="a b"
5 1 2 3

$b1
$$data ncol=3 modes="numeric logical character" \
  byrow=TRUE colnames="N L C"
5 T aa
3 F bb
8 T cc
10.5 F dd

$b2
$$data ncol=3 modes="numeric logical character" \
  byrow=FALSE colnames="a b c"
5 3 8 10.5
T F T F
aa bb cc dd
```

3. Functions for Data Exchange

Computer models usually require data exchange between model components. For example, as described above, the functions `getWinVal` and `setWinVal` move data between an R program and the GUI. Other applications, such as those written separately in C, may have the ability to write data to files that R can read. In cases like this, it would be convenient to have variable names in the C code correspond to variables with the same names in R. `PBSmodelling` can facilitate this process with the functions `readList` and `writeList`, which convert a text file to an R list and vice-versa. Another function `unpackList` creates local or global variables with

names that match the list components. Similarly a global or local list can be populated with objects by name through the function `packList`.

Table 3 illustrates a data file in PBS format, legible by `readList`. The file contains lines with an initial dollar sign (like `$x` in Table 3) that specify a list component name in R, followed by one or more lines of data. Data items are separated by white space. A single item of data corresponds to a scalar in R, multiple items on a single line correspond to a vector, and multiple lines of data correspond to a matrix with the number of columns determined by the first line of data. Thus, in Table 3, `$x` is a scalar, `$y` is a vector of length 4, and `$z` is a 2×4 matrix. The format also supports four possible data type definitions on a line preceded by `$$`:

```
$$ vector mode=numeric names=""
$$ matrix mode=numeric ncol rownames="" colnames="" byrow=TRUE
$$ data modes=numeric ncol rownames="" colnames byrow=TRUE
$$ array mode=numeric dim fromright=TRUE dimnames
```

Table 3 illustrates their use in specifying `$a`, `$b1`, and `$b2`. Matrices and data frames can be read by row or column. This choice determines the order of reading the data, and white space (including line breaks) merely signifies breaks between data items. Array objects with three or more dimensions can be read in two ways, with indices varying first from the right or from the left. For example, data for an array indexed by `[i, j, k]` are read by varying `i` first with fixed `j` and `k` if `fromright=TRUE`. Similarly, `k` varies first if `fromright=FALSE`.

As in widget descriptions, arguments may be omitted in favour of their defaults, and the `$$` line may be continued across multiple lines by using a backslash character `\`. For a matrix, the argument `ncol` is required. Similarly, a data object (i.e., a data frame) must specify `ncol` and a vector `colnames` of length `ncol`. Also, `modes` must have length 1 (so that all entries in the data frame have the same mode) or length `ncol`. An array must have a complete `dim` argument, a vector giving the number of dimensions for each index, and a `dimnames` argument, which is a collapsed vector; the first element is the name of the first dimension, followed by each index label in that dimension; each dimension is appended to end of the vector.

As indicated earlier, `PBSmodelling` can use this specialized data format as a convenient means of capturing data from other programs. For example, to export data from an external C program, write C code that generates a data file in PBS format, where component names in the file match the C variable names. Then read the resulting file into an R session with the function `readList`, and use `unpackList` to produce local or global R variables. At this point, both R and C share data with the same variable names. This method works well with programs written for [AD Model Builder](#), a package used extensively in fishery research and other fields. It uses reverse automatic differentiation (AD; Griewank 2000) for highly efficient calculation of maximum likelihood estimates.

To considerable extent, R has native support for reading and writing a variety of text files, including the functions `scan`, `cat`, `source`, `dump`, `dget`, `dput`, `read`, `write`, `read.table`, and `write.table`. External programs sometimes utilize R formats for their input data. For example, the program `WinBUGS` (Speigelhalter et al. 2004), which implements Bayesian inference using Gibbs sampling, uses data files written in a list format closely related to the R syntax produced by the `dput` function. If the file `myData.txt` has `dput` format, then either of the two R commands

```
myData <- dget("myData.txt");
myData <- eval(parse("myData.txt"));
```

produces a corresponding R list object named `myData`.

We should, however, add a word of caution here. When R saves array data in dput format, it converts the array to a vector by varying the indices from left to right. For example, a matrix with indices $[i, j]$ is saved as a vector in which i varies for each fixed j . In effect, the data are stored by column. This sometimes gives an unnatural visual appearance. In English, the eye reads naturally from left to right, then down. Matrices are normally displayed by row, with column index j varying for each fixed i . WinBUGS, supported by the R package BRugs (Thomas 2004), requires input data formatted in this visually meaningful way. More generally, WinBUGS reads arrays by varying the indices from right to left. The BRugs function bugsData writes data in this format, but users must take special care in reading WinBUGS data with the dget function.

4. Support Functions for Graphics and Analysis

As mentioned in the preface, we have devised a number of functions that make it easier for us to work in R. Some of them, such as plotBubbles, relate to techniques discussed in our published work (e.g., Richards et al. 1997; Schnute and Haigh 2007). Others just provide convenient utilities. For example, testCol("red") shows all colours in the palette colors() that contain the string "red". We also provide support for a few analytical methods, such as function minimization. This section gives a brief description of PBSmodelling support functions. See the help files for further information.

4.1. Graphics Utilities

In many of the graphical functions, we utilize a PBSmodelling function called evalCall. The functionality of evalCall is similar to that of do.call in the base package; however, we have geared our function towards rationalising arguments passed through the dots (...) argument so that no conflicts occur with formal arguments. This way the user can override predefined arguments in functions embedded within functions without the parent function having to recognize all the arguments in the embedded function.

resetGraph Reset various graphics parameters to defaults, with mfrow=c(1,1).
expandGraph Set various graphics parameters to make graphs fill out available space.

drawBars Draw a linear bar plot on the current graph.
genMatrix Generate a test matrix for use in plotBubbles.
plotACF Plot autocorrelation bars (ACF) from a data frame, matrix, or vector.
plotAsp Plot a graph with a prescribed aspect ratio, preserving xlim and ylim.
plotBubbles Construct a bubble plot for a matrix.
plotCsum Plot cumulative sum of a vector, with value added.
plotDens Plot density curves from a data frame, matrix, or vector.
plotFriedEggs Render a pairs plot as fried eggs (density contours) & beer (correlations).
..... *code courtesy of Dr. Steve Martell, (Sea State Inc., Seattle WA.)
plotTrace Plot trace lines from a data frame, matrix, or vector.

addArrows Call the arrows function using relative (0:1) coordinates.
addLegend Add a legend using relative (0:1) coordinates.
addLabel Add a panel label using relative (0:1) coordinates.

pickCol Pick a colour from a complete palette and get the hexadecimal code.
testAlpha Display various alpha transparency values.
testCol Display named colours available based on a set of strings.

testLty Display line types available.
 testLwd Display line widths.
 testPch Display plotting symbols and backslash characters.

4.2. Data Management

clearAll Function to clear all data in the global environment.
 pad0 Pad numbers with leading zeroes (string).
 show0 Show decimal places including zeroes (string).
 unpackList Unpack the objects in a list and make them available locally or globally.
 view View the first/last/random n rows of a data frame or matrix.

4.3. Function Minimization and Maximum Likelihood

Three functions in the `stat` package support function minimization in R: `nlm`, `nlmnb`, and `optim`. These tend to perform slowly compared with other software alternatives, due partly to R’s interpretive function evaluation. Nevertheless, for small problems they offer a convenient means of analysis, based entirely on code written in R. Our examples illustrate some of the possibilities. For large problems coded in other software, we still like to write independent code for a function in R, based only on the model documentation. If both versions of the software produce the same function values at selected values of the function arguments, then we have greater confidence that we have represented our model correctly in code. In that context, R serves as a valuable debugging tool.

`PBSmodelling` provides a support function `calcMin` that can use any method available in the `stat` package to find the vector $(\hat{x}_1, \dots, \hat{x}_n)$ of length n that minimizes the function $y = f(x_1, \dots, x_n)$. In practice, we usually apply this to the negative log likelihood for a statistical model, where the variables x_i are parameters. We define a new class `parVec`, which is a data frame with four columns:

- `val` – the actual value of parameter x_i ;
- `min` – a minimum allowable value of x_i ;
- `max` – a maximum allowable value of x_i ; and
- `active` – a logical value that determines whether or not the minimization algorithm should vary the value of x_i . If `active=F`, then x_i remains unchanged at the value `val`.

Internally, `calcMin` scales active variables x to surrogate variable s in the range $[0,1]$, where x and s are related by the inverse formulas (Schnute and Richards 1995, p. 2072):

$$x = x_{\min} + (x_{\max} - x_{\min}) \frac{1 - \cos(\pi s)}{2} = x_{\min} + (x_{\max} - x_{\min}) \sin^2 \left(\frac{\pi s}{2} \right), \quad (4.3a)$$

$$s = \frac{1}{\pi} \arccos \left(\frac{x_{\max} + x_{\min} - 2x}{x_{\max} - x_{\min}} \right) = \frac{2}{\pi} \arcsin \sqrt{\frac{x - x_{\min}}{x_{\max} - x_{\min}}}. \quad (4.3b)$$

All these formulas represent equivalent forms of a one-to-one relationship $x \leftrightarrow s$, where $x_{\min} \leq x \leq x_{\max}$ and $0 \leq s \leq 1$. Readers may find the second versions of (4.3a) and (4.3b) more intuitive (with a familiar “arc sine square root” transformation in (4.3b)), but the code uses the first versions for a possible improvement in computational efficiency by avoiding square and square root functions. The minimization algorithm works entirely with surrogate variables, which may have dimension smaller than n if some variables x_i are not active. The function `scalePar` scales an object x of class `parVec` to a vector s of surrogates via the formula (4.3b). Similarly, `restorePar` recovers x from s via (4.3a).

We also provide a convenient function `GT0` that restricts a numeric variable x to a positive value defined by

$$\text{GT0}(x, \varepsilon) = \begin{cases} x, & x \geq \varepsilon \\ \frac{\varepsilon}{2} \left[1 + \left(\frac{x}{\varepsilon} \right)^2 \right], & 0 < x < \varepsilon \\ \frac{\varepsilon}{2}, & x \leq 0 \end{cases} \quad (4.3c)$$

The notation `GT0` denotes “greater than zero”. This function preserves the value of x if $x \geq \varepsilon$, and for smaller values x it is always true that $\text{GT0}(x, \varepsilon) \geq \varepsilon/2$. The function (4.3c) also has a continuous first derivative that makes sense locally on a small scale of size ε . This property makes it useful for avoiding unrealistic numbers that might be negative or zero, particularly when the minimization algorithm uses derivatives of the objective function. In summary, `PBSmodelling` has four functions that facilitate function minimization.

`calcMin` Calculate the minimum of a user-defined function.
`scalePar` Scale parameters to surrogates in the range [0,1].
`restorePar` Restore actual parameters from surrogate values.
`GT0` Restrict a numeric variable to a positive value (“Greater Than 0”).

4.4. Handy Utilities

`calcFib` Calculate Fibonacci numbers (included to illustrate the use of C code).
`calcGM` Calculate the geometric mean of a vector of numbers.
`clearRcon` Clear the R console (code that executes ‘Ctrl L’).
`convSlashes` Convert pathway slashes from UNIX ‘/’ to DOS ‘\’. format.
`findPat` Find all strings that include any string in a vector of patterns.
`getYes` Prompt the user with a GUI to choose yes or no.
`isWhat` Identify an object by its class and attributes
`pause` Pause, typically between graphics displays.
`showAlert` Display a message in an alert window.
`showArgs` Show the arguments for a specified widget in Appendix A.
`showHelp` Display the Help Page for specified packages installed on user’s system.
`showPacks` Show packages required, but not installed on a user’s system.
`testWidgets` GUI to test all widgets listed in Appendix A.
`view` View the first/last/random n lines of a (potentially large) object.
`viewCode` View R code for all functions in a specified package on the user’s system.

5. Functions for Project Management

A project to design and write software typically involves keeping track of numerous component files that contain material at various stages of progress. Some contain input, such as source code, data, or documentation. Others contain various stages of output, such as compiled code, processed documents, graphs, and other analytic results. Specialized software, such as C compilers, text processors (like TeX), database utilities, and R itself play a role in converting the input to the output. Along the way, intermediate files often get created that ultimately need to be removed to give a clean result. GUI tools in `PBSmodelling` can assist a user in managing such projects.

For simplicity, we envisage a project as a collection of files in the current working directory that typically share a common prefix but also have various possible extensions, such as `.c`, `.h`, `.o`, `.so`, `.dll`, and `.exe`. We provide a GUI that illustrates a special case of project management. It allows a user to create and compile a C function, load it into R, run it, and compare the results with a similar function coded entirely in R. See the companion functions:

`loadC` Launch a GUI for compiling and loading C code.
`compileC` Compile a C file into a shared library object.

5.1. Project Options

Projects commonly involve specific paths and filenames associated with applications and binary libraries. To preserve information about these and other settings, `PBSmodelling` provides an S4 class `PBSoptions` for defining options, editing them in a GUI, and saving them to a local file. Instances of `PBSoptions` are independent of each other. We recommend that users create a distinct `PBSoptions` object for each distinct project.

Internally, an object of class `PBSoptions` contains (1) the options themselves as a (possibly empty) list, (2) a default file name in which to save the options, and (3) a default prefix for recognizing entries in a GUI that correspond to options. For example, the following code creates and displays a `PBSoptions` object called `myOpts`:

```
> # Create myOpts
> myOpts <- new("PBSoptions", filename="myOpts.txt",
+ initial.options=list(a="a",b="b"), gui.prefix="PBSopt")

> # Display myOpts
> myOpts
filename: myOpts.txt
gui.prefix: PBSopt
Options:
 $ a: chr "a"
 $ b: chr "b"
```

More generally, the `new` command (via the `initialize` method for class `PBSoptions`) first attempts to load previously saved values from the file `filename`. If the attempt fails or any options are missing, `new` assigns default options from `initial.options`. Users should generally save the newly created `PBSoptions` object in the global environment to facilitate the retrieval and modification of options from various functions in different scopes. For more details on object initialization, consult the `PBSoptions` class documentation.

The following functions allow users to retrieve or modify the values stored in a `PBSoptions` object:

```
getOption ..... retrieve options from the object;
setOption ..... add or modify options in the object;

getOptionPrefix..... retrieve the prefix that identifies widget variable names;
setOptionPrefix..... modify the prefix value;

getOptionFileName..... retrieve the default filename;
setOptionFileName..... modify the default filename.
```

Potentially, options can exist at three levels: a GUI window, internal R memory, or a file. They become active when they exist in internal memory as part of a `PBSoptions` object. In a GUI window with numerous entry fields, the `gui.prefix` identifies those fields that correspond to options. In the example above, where `gui.prefix="PBSopt"`, an entry field with `name=PBSoptCpath` would correspond to the option `Cpath` in a `PBSoptions` object. This naming convention allows options to be displayed and modified in a GUI. The following support functions allow a user to move options between a `PBSoptions` object and GUIs or files:

```
loadOptionsGUI..... load options from the object into a GUI;
saveOptionsGUI..... save options from a GUI in the object;

loadOptions ..... load options from a file into the object;
saveOptions ..... save options from the object into a file.
```

The structures and methods described above make it easy to prescribe options, modify them in a GUI, and save their values in files. A user typically develops a project in a directory where a particular file preserves the options between R sessions. More generally, files with distinct names can preserve distinct sets of options. An R function can automatically initialize the project by creating `PBSoptions` objects from the corresponding files.

From a technical perspective, `PBSoptions` objects have a single slot instance. This contains a hidden environment that is created on object initialization and preserved when objects are copied. Effectively, the class definition allows objects to be passed by reference, rather than by value. The methods can manipulate the original object and avoid the need for returning a new modified `PBSoptions` object.

5.2. Project Management Utilities

Sometimes projects have an association with an R package. For this reason, we include functions that can open files and examples from an R package installed on the user's computer:

```
openPackageFile..... Open a file from a package subdirectory
..... (deprecated, use openFile);
openExamples ..... Open files from the examples subdirectory of a package.
```

As discussed above, a project typically includes multiple files with the same prefix and a potential set of suffixes. (A suffix doesn't necessarily have to be a file extension. For example, you can use the prefix `foo` and the suffix `-bar.xxx` to match the file `foo-bar.xxx` where the extension is `.xxx`.) We provide a utility to open these files, provided that their extensions have associated applications. We also allow a user to search the current working directory for

potential prefixes, or to browse for a working directory and find such prefixes. Furthermore, a project can be “cleaned” by removing files with specified suffixes. See the functions:

openProjFiles open files with a common prefix;
findPrefix find a prefix based on names of existing files;
findSuffix find suffix of system files with specified prefix;
setwdGUI browse for a working directory;
cleanProj launch a GUI for project file deletion;
cleanWD launch a GUI to delete files from the current working directory.

6. Support for Lectures and Workshops

Speakers giving lectures and workshops about R often want their audience to experience the consequences of running some R code. Participants sometimes find themselves scrambling to copy code from the visual presentation, related web sites, or files distributed by the speaker. During this process, the actual point of the lecture can get lost. Focus shifts from R concepts to typing, other mechanical issues, and a struggle to keep up with the speaker’s activity.

PBSmodelling offers a potential solution to this problem that preserves an interactive spirit while ensuring that participants easily see the results from planned segments of R code. We encapsulate our approach in the two functions:

showRes display a string of R code and show results on the R console;
presentTalk present a talk on the R console, based on a talk description file.

The first provides a minor tool that sometimes comes in handy. The second implements a much more general idea. Just as a *window description file* defines a GUI window, a *talk description file* defines a talk that runs on the R console. A small GUI makes it easy to step through the talk interactively, with easy movement forward or backward. Planned results appear on the R console, and yet the console remains available for additional spontaneous code entry.

The author of a talk writes a text file that contemplates a sequence of actions, such as displaying text, running R code, and opening files. If audience members receive this file in advance, they can readily follow every step during the talk by simple mouse clicks on the GUI. The file also gives them an opportunity to review the concepts at a convenient later time. We anticipate R tutorials written as talk description files, and we may eventually add some to PBSmodelling.

For simplicity, our talk description files conform to the [XML specifications](#), and the R package XML is required to read them. We support the following five XML elements:

<talk> ... </talk>	to delimit an entire talk;
<section> ... </section>	to delimit a section within a talk;
<text> ... </text>	to delimit text that should appear in the R console;
<file> ... </file>	to delimit names of files that should be opened;
<code> ... </code>	to delimit code that should run in the R console.

Consistent with the standard format, each element has an initial tag in angle brackets, intermediate material (indicated here by . . .), and a final tag with a backslash character (/) prefixed to the initial tag. Each initial tag must be closed with a corresponding final tag.

Initial tags can include arguments, for which the values must appear in double or single quotes. Appendix B lists the complete syntax for all five tags. For example, the element

```
<code show="TRUE" print="TRUE" break="all"> plot(cars) </code>
```

would show the string `plot(cars)` on the R console, pause (i.e., `break`), generate the plot, print any related output on the R console, and then pause again.

As illustrated in Table 1, a talk description file must contain exactly one `<talk>` element as the root of a branching tree. The `<talk>` can contain one or more `<section>` branches. In turn, each `<section>` can contain any mixture of leaf nodes: `<text>`, `<file>`, or `<code>`. The `<talk>` and `<section>` elements play organizational roles, whereas the leaf nodes correspond to concrete actions. A `<text>` element specifies ordinary text, such as lecture notes, that should appear in the R console. A `<file>` element causes one or more files to open at this point of the talk. For example, it might be desirable to display a file of R code or open a PowerPoint file. A `<code>` element causes code to be displayed and run in the R console. Appendix B gives complete syntax details for talk description files.

Comments within a talk description file follow the standard XML format:

```
<!-- ... -->
```

where `...` denotes the text of the comment. XML has five standard reserved characters as shown in the list below, where “&” serves as an escape character that allows these characters to be interrupted as ordinary text.

Character	Escaped Character	Description
<	<	Less-than character starts an element tag
>	>	Greater-than character ends an element tag
&	&	Ampersand is used for escaping characters
"	"	Used for argument values in a leading tag
'	'	Used for argument values in a leading tag

If `<text>` or `<code>` elements contain numerous characters that must be escaped, then the syntax:

```
<![CDATA[...]]>
```

allows any raw character data (`...`) to be included.

Table 4. A talk description file *swisstalk.xml* designed for use with the *PBSmodelling* function *presentTalk*. This talk examines method dispatch for the *summary* function and illustrates how it applies to the *swiss* data set, which has class *data.frame*.

File: swisstalk.xml

```
<!-- We will use xml comments. There must be only ONE document root. -->
<talk name="Swiss">

  <!-- SECTION 1. summary method -->
  <section name="Methods" button="TRUE">

    <text break="T">
    This short talk examines the "summary" method
    and applies it to the "swiss" dataset.</text>
    <text break="T">
    The talk itself comes from a talk description file.
    The next step should open that file (swisstalk.xml),
    provided that a program is associated with xml files. </text>

    <file name="swisstalk" button="TRUE">swisstalk.xml</file>

    <text break="F">
    "summary" is a function (class function). </text>

    <code break="print">
    isWhat(summary) # isWhat() from PBSmodelling</code>
    <text break="F">"summary" is generic:</text>
    <code break="print">summary</code>
    <text break="F">"summary" has many methods:</text>
    <code break="print"> methods(summary)</code>

  </section>

  <!-- SECTION 2. The "swiss" data -->
  <section name="Data" button="TRUE">

    <text break="F"> "swiss" is a data frame (class data.frame):</text>
    <code> isWhat(swiss)</code>
    <text break="F"> You can read about the data here:</text>
    <code> help(swiss) # open the help file</code>
    <text break="F"> Apply "summary" to Swiss:</text>
    <code break="print"> summary(swiss)</code>
    <text break="F"> Print the first 3 records:</text>
    <code break="print"> head(swiss,3)</code>
    <text break="F"> Display the data with the "plot" method . . .</text>
    <code print="F"> plot(swiss,gap=0)</code>
    <text> THE END .. THANKS FOR WATCHING!</text>

  </section>
</talk>
```

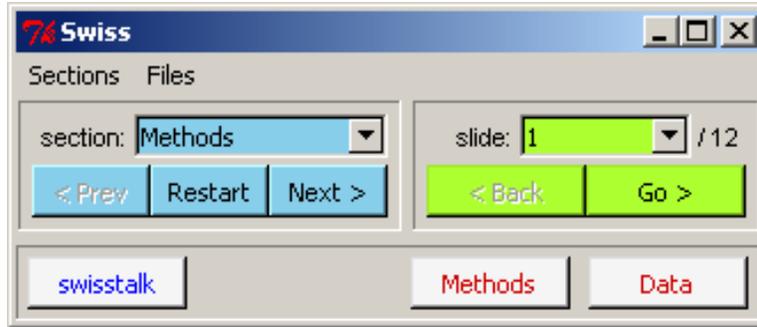


Figure 5. The GUI generated by *presentTalk* from the talk description file in Table 4.

The “Swiss Talk” example in *PBSmodelling* allows a user to view the results from the short talk description file in Table 4. The first section (named “Methods”) starts with a brief text message in the R console. The next step displays the description file itself (*swisstalk.xml*), as an illustration of how *presentTalk* works. Then the audience sees aspects of R’s polymorphic function summary. The *isWhat* function (from *PBSmodelling*) shows its properties, and the *methods* function reveals the diverse ways in which *summary* has been overloaded. The second section (named “Data”) shows properties of the data frame *swiss*, as well as the consequences of applying *summary* and *plot* to this object. The talk closes with a classic message showing “THE END”.

The code elements supported by *presentTalk* give an author considerable scope for introducing breaks and other features into the presentation. Furthermore, the root `<talk>` block in the description file produces a corresponding GUI, similar to the one shown in Figure 5. This enables the speaker to move stepwise through the presentation, via the “Go” button, analogous to moving through slides in a conventional talk. After each step, the R console remains open for additional code written on the spur of the moment. Furthermore, the menu items (“Sections”, “Files”) allow for quick movement among sections, as well as spontaneous opening of files. For example, the speaker might choose to open and close the same file several times during a presentation. This can be programmed into the talk description or done spontaneously through the “Files” menu.

In addition to the automatic menu items, a user can add buttons to the GUI that accomplish similar purposes. For example, Figure 5 shows buttons that will move to the start of the sections “Methods” and “Data” or open the “*swisstalk*” description file. The “Back” button moves back to the previous slide, and individual slides can be selected via a droplist widget. The blue buttons allow movement among sections – “Prev” to the previous section, “Restart” to the start of the current section, and “Next” to the next section.

Code executed during a talk presentation potentially changes objects in the current global environment. Although the GUI allows quick jumps among slide and sections of talks, the speaker needs to remain aware of objects currently in the global environment. For example, if the first section of the talk creates objects needed by the second section, it makes no sense to skip to the second before the first has done its work. In some cases, it may help to start a talk or section with `<code> clearAll() </code>` to ensure that previous objects in the environment don’t conflict with those now being created. On the other hand, depending on the author’s intent, this could be entirely the wrong thing to do.

In practice, a speaker would present his or her talk from a laptop connected to a digital projector. In this context, it is almost essential to choose large fonts in the R console. When writing a talk, it helps to view it with font sizes and R console dimensions chosen with the final presentation in mind.

7. Examples

As mentioned in the Preface, `PBSmodelling` includes a variety of examples that illustrate applications based on this and other packages. Generally, each example contains documentation, R code, a window description file, and (if required) other supporting files. All relevant files appear in the R library directory `PBSmodelling\Examples`. An example named `xxx` typically has corresponding files `xxxDoc.txt` or `xxxDoc.pdf` (documentation), `xxx.r` (R code), and `xxxWin.txt` (a window description). In the GUI for each example, buttons labelled `Docs`, `R Code`, and `Window` open these files provided that suitable programs have been associated with the file extensions `*.txt`, `*.pdf`, and `*.r`. In particular, a suitable program (such as the Acrobat Reader) must be installed for reading `*.pdf` files, and you may need to associate a text file editor with `*.r`. On some systems, it may be necessary to use the function `setPBSext` to define these associations, as discussed earlier in Section 2.3.

Use the function `runExamples()` to view all examples currently available in `PBSmodelling`. This procedure copies all relevant files to a temporary directory located on the path defined by the environment variable `Temp`. It then opens a window in which radio buttons allow you to select any particular case. Closing the menu window causes the temporary files and related data to be cleaned up, and returns to the initial working directory.

There is also a similar function `runExample()` which can run a single example. Choices exist in the `PBSmodelling/examples` directory and currently include:

`"CalcVor"`, `"CCA"`, `"FishRes"`, `"FishTows"`, `"LinReg"`, `"LissFig"`, `"MarkRec"`, `"PopSim"`, `"RanProp"`, `"RanVars"`, `"SGM"`, `"SineNorm"`, `"sudokuSolver"`, `"TestFuns"`, `"vonB"`, `"ypr"`

Alternatively, you can copy all the files from `PBSmodelling/examples` to a directory of your choice and open R in that working directory. To run example `xxx`, type `source("xxx.r")` on the R command line. For instance, `source("LissFig.r")` creates a window (from the description file `LissFigWin.txt`) that can be used to draw the Lissajous figures described in Section 2.1. The built-in example also includes a history widget for collecting settings that the user wishes to retain.

The examples documented here illustrate only some of those available in version 1 of `PBSmodelling`. For instance, we also include a `TestFuns` GUI that we have used as a tool for debugging various functions in the package. In future versions, we plan to add more examples that illustrate important modelling concepts and provide convenient supplementary materials for university courses in fisheries, biology, ecology, statistics, and mathematics. The function `runExamples()` should always represent the complete list currently available, and the `Docs` button for each case should link to the appropriate documentation.

The nine examples presented in this section illustrate some of the possibilities available in `PBSmodelling`, although the documentation may be somewhat out of date. For example, the figures in this report may not correctly represent current versions of the GUIs and their associated graphical output. Use the `Docs` button to read the most current information for each example.

7.1. Random Variables

7.1.1. RanVars – random variables

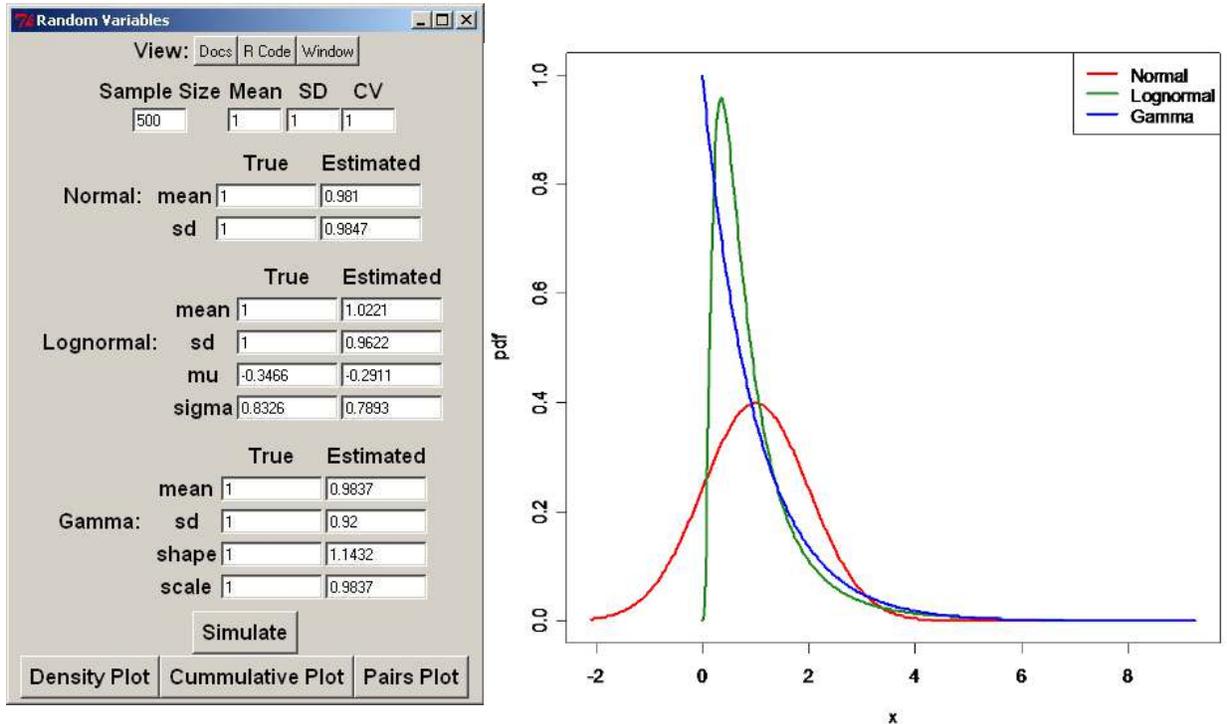


Figure 6. *RanVars* GUI (left) and density plot (right). Simulations are based on 500 random draws with mean=1 and SD=1.

The *RanVars* example draws samples from three continuous random distributions (normal, lognormal, and gamma) with a common mean μ and standard deviation σ . The documentation («Docs» button) shows relevant formulas that connect distribution parameters with the moments μ and σ . Estimated parameter values from a simulation (invoked by «Simulate») are displayed in the GUI alongside the true values (Figure 6). We use only the straightforward moment formulas in the documentation, without sample bias correction formulas like those described by Aitchison and Brown (1969). Three buttons at the bottom of the GUI portray the data visually as density curves, cumulative proportions, and paired scatter plots.

7.1.2. RanProp – random proportions

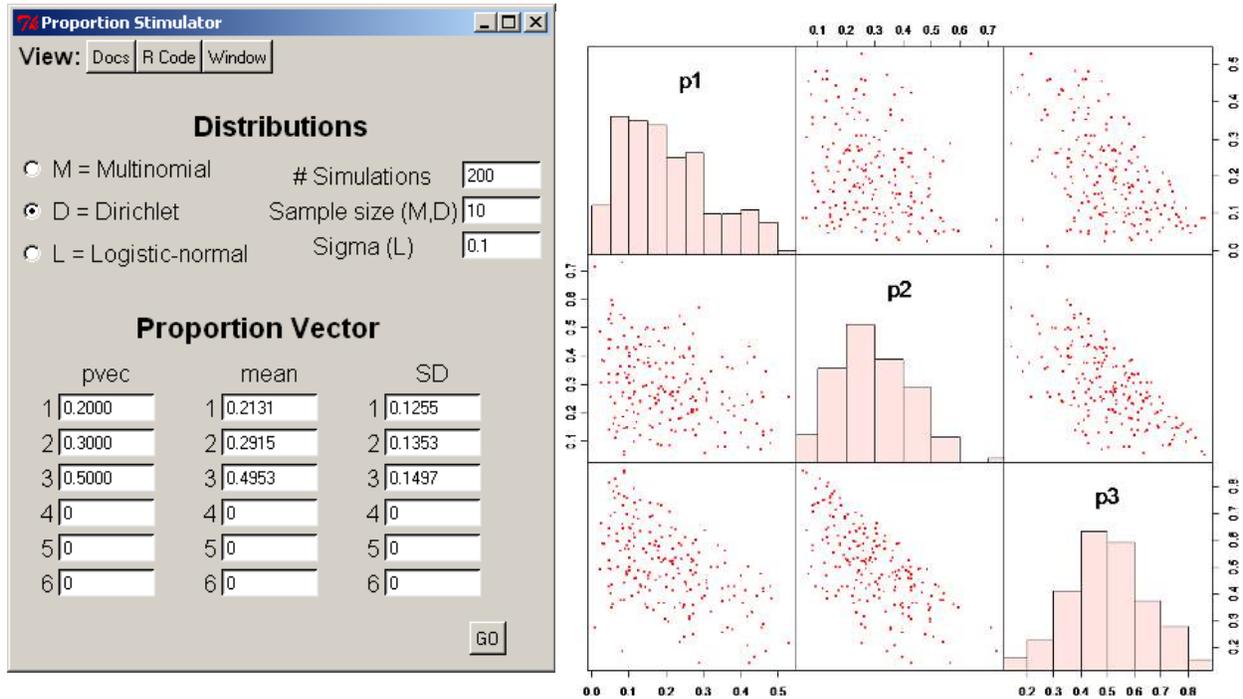


Figure 7. *RanProp* GUI (left) and pairs plot (right). Simulations are based on 200 random draws where $n = 10$ for the multinomial and Dirichlet distributions and $\sigma = 0.1$ for the logistic-normal distribution. The pairs plot portrays results for the Dirichlet.

The *RanProp* example simulates up to five random proportions drawn from one of three distributions – multinomial, Dirichlet, and logistic-normal. The observed proportion means and standard deviations are reported in the GUI (Figure 7), and a graphical display renders the points as a paired scatter plot. After defining options in the GUI, including the vector “pvec” of true underlying proportions, press «Go». Schnute and Haigh (2007) provide further technical details about these three distributions.

7.1.3. SineNorm – sine normal

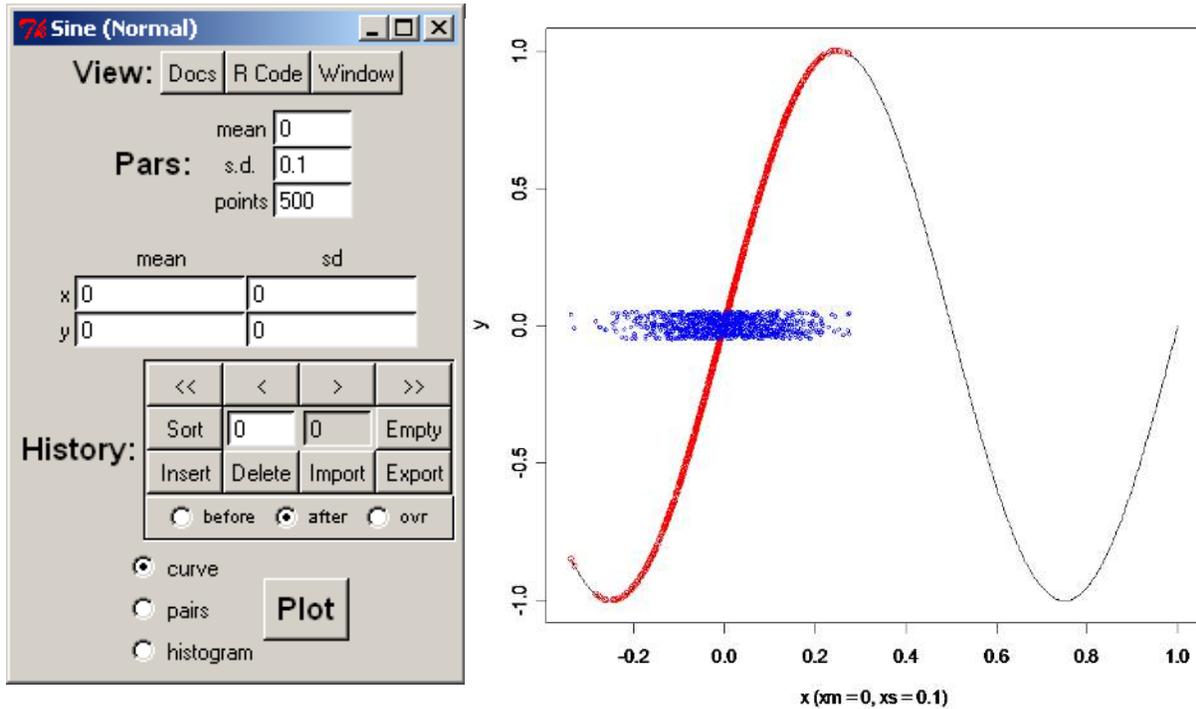


Figure 8. *SineNorm* GUI (left) and plot (right). Simulations are based on 500 random draws of $y = \sin(2\pi x)$, where x is normal with mean $\mu = 0$ and standard deviation $\sigma = 0.1$. Blue points portray jittered values of x , and red points show corresponding values of y .

The SineNorm example illustrates a somewhat unconventional random variable $y = \sin(2\pi x)$, where x is normal. The GUI allows you to specify the mean μ and standard deviation σ of x . If $\mu = 0$ and σ is small, the transformation is nearly linear, so that y is approximately normal. If σ is large, the transformation concentrates y near -1 and 1. Figure 8 illustrates the transformation when σ has the moderate value 0.1. Try $\sigma = 10$ to see how values y tend to occur near the peaks and troughs of the sine function, where the slope is relatively flat.

7.1.4. CalcVor – calculate Voronoi tessellations

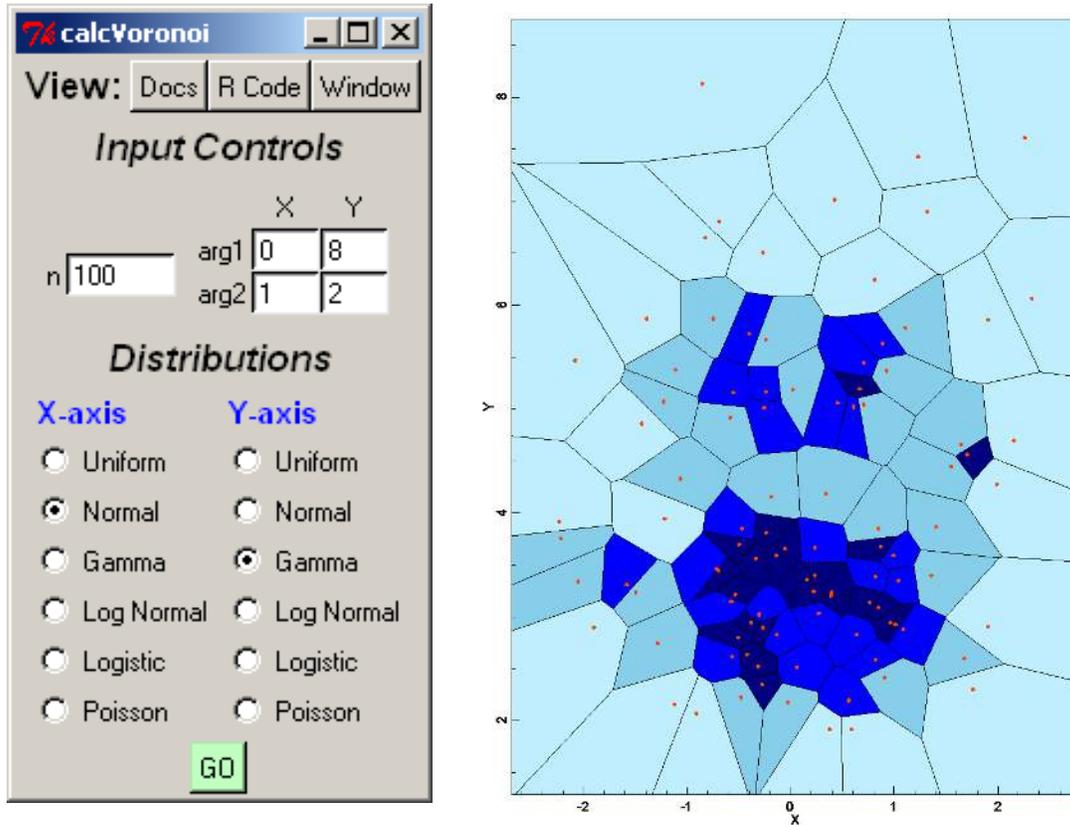


Figure 9. *CalcVor* GUI (left) and plot (right). Tessellation of random points (red) that are normally distributed on the x-axis ($mean=0, sd=1$) and gamma-distributed on the y-axis ($shape=8, rate=2$).

The CalcVor example calls PBSmapping's calcVoronoi function, which calculates the Voronoi (Dirichlet) tessellation for a set of points using the deldir function in the CRAN package deldir. The GUI accepts two arguments for each random distribution represented on each axis. The underlying functions and their arguments are:

Distribution	Function	Argument 1	Argument 2
Uniform	runif	min	max
Normal	rnorm	mean	sd
Gamma	rgamma	shape	rate
Log normal	rlnorm	meanlog	sdlog
Logistic	rlogis	location	scale
Poisson	rpois	lambda	---

7.2. Statistical Analyses

7.2.1. LinReg – linear regression

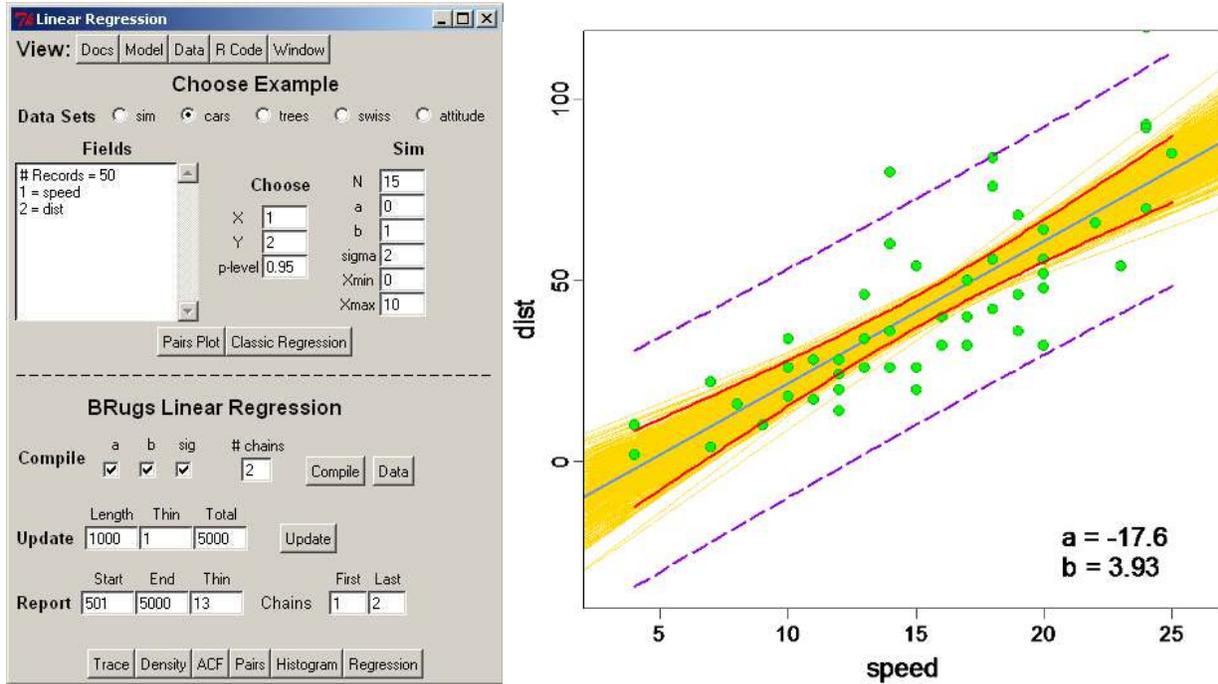


Figure 10. *LinReg* GUI (left) and regression plot (right). The linear regression uses the *cars* dataset ($n=50$) to predict *dist* vs. *speed*. The plot shows observations (green circles), fitted line (solid blue line), the 95% confidence limits of the fitted model (solid red lines), the 95% CL of the data (dashed purple lines), and the fits using the Bayes posterior estimates of (a,b) (gold lines).

The example *LinReg* estimates parameters in a linear regression $y = a + bx$ using either simulated data or data objects that come with the R-package. We compare a classical frequentist regression with results from Bayesian analysis, using the *BRugs* package to interface with the program *WinBUGS*. After selecting various data options, «Pairs Plot» shows a pairs plot (x, y) and «Classic Regression» adds confidence limits (at «p-level») from regression theory. Red and violet curves show bounds for a prediction or a new observation, respectively, each conditional on x . If the data came from simulation, a blue line portrays the truth, with specified values a and b , that must be estimated from the data.

A corresponding Bayesian analysis uses the *WinBUGS* model shown by pressing «Model». Choose parameters to monitor (normally all of them): the intercept a , the slope b , and the predictive standard deviation σ . After specifying a number of sample chains for the MCMC sample, press «Compile» to compile the model with these settings. «Update» generates samples in «Length» increments. Additional buttons at the bottom of the GUI allow you to explore the MCMC output. Posterior samples of (a,b) correspond to sample lines. The «Regression» button illustrates these in relationship to confidence limits from a frequentist analysis (Figure 10).

7.2.2. MarkRec – mark-recovery

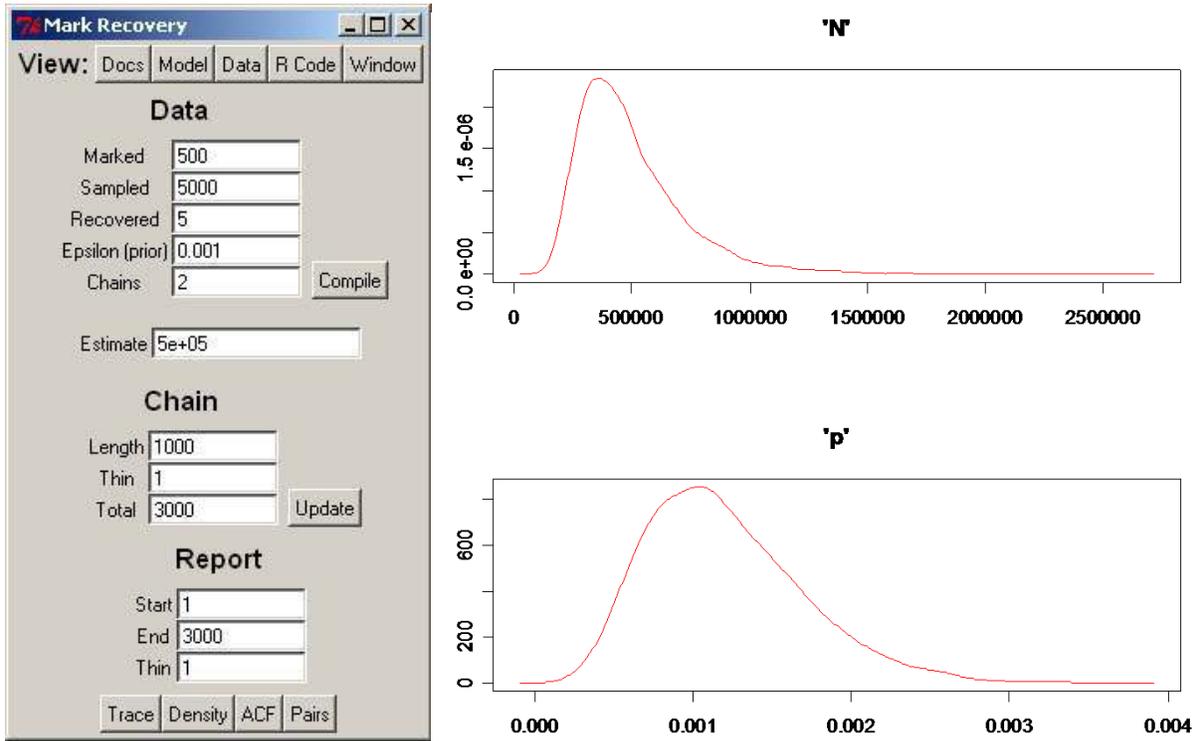


Figure 11. MarkRec GUI (left) and density plots (right). A low recovery of marked fish can lead to fat tails in N due to occasional large spikes in the population estimate.

The example MarkRec performs a Bayesian analysis of a mark-recovery experiment in which M fish are marked and allowed to disperse randomly in the population. Later, a sample of size S is removed from the population and R marks are recovered. Both the total population N and the marked proportion p are unknown, where

$$p = \frac{M}{N} \cong \frac{R}{S}.$$

In one version of the theory, R is binomially distributed with probability p in a sample of size S , and the above approximation suggests the estimate

$$\hat{N} = \frac{S}{R} M = \frac{M}{R} S.$$

When recoveries are low ($R \approx 0$), the posterior distribution of N exhibits a fat tail (Figure 11).

As in LinReg, «Model» shows the MarkRec model for WinBUGS, which (deliberately) includes an illegitimate prior that depends on the data. By increasing an initially small quantity ε , this fake prior allows the tail of N values to be arbitrarily clipped. Schnute (2006) gives some historical perspective to this analysis, in the context of work by W.E. Ricker.

7.2.3. CCA – catch-curve analysis

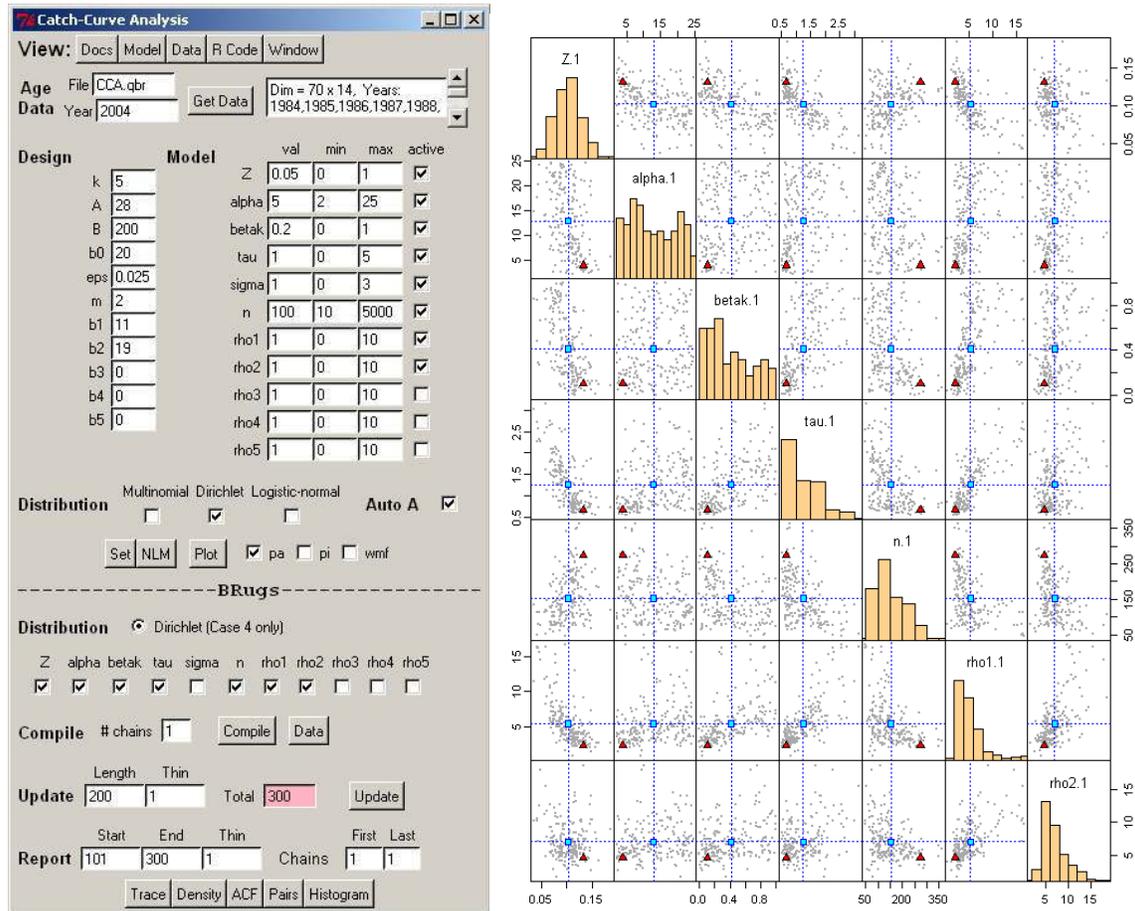


Figure 12. CCA GUI (left) and parameter pairs plot (right). Comparison of Bayes posterior distribution of CCA model parameter estimates from chain 1 (N=100). Symbols indicate means (blue squares) and modes (red triangles). Diagonal shows parameter estimate distributions.

The example CCA illustrates a catch-curve model proposed by Schnute and Haigh (2007). It incorporates effects of survival, selectivity, and recruitment anomalies on age structure data from a single year. After making various model choices, press «Set», «NLM» (which may take several seconds), and «Plot» to view the maximum likelihood estimates and their relationship with the data. A WinBUGS model («Model») allows us to calculate posterior distributions. (See the last few lines of “Model”.) As in MarkRec, select parameters to monitor, specify a number of chains, and «Compile» the model. «Update»s may be slow, but eventually they produce interesting posterior samples (Figure 12). «Docs» gives details of the deterministic model, and the Dirichlet distribution is used to describe error in the observed proportion.

We include this example to illustrate a somewhat realistic WinBUGS model that can be used to estimate parameters for a population dynamics model. Further information can be found in Schnute and Haigh (2007). PBSmodelling includes the data for this example as the matrix CCA.qbr.

7.3. Other Applications

7.3.1. FishRes – fishery reserve

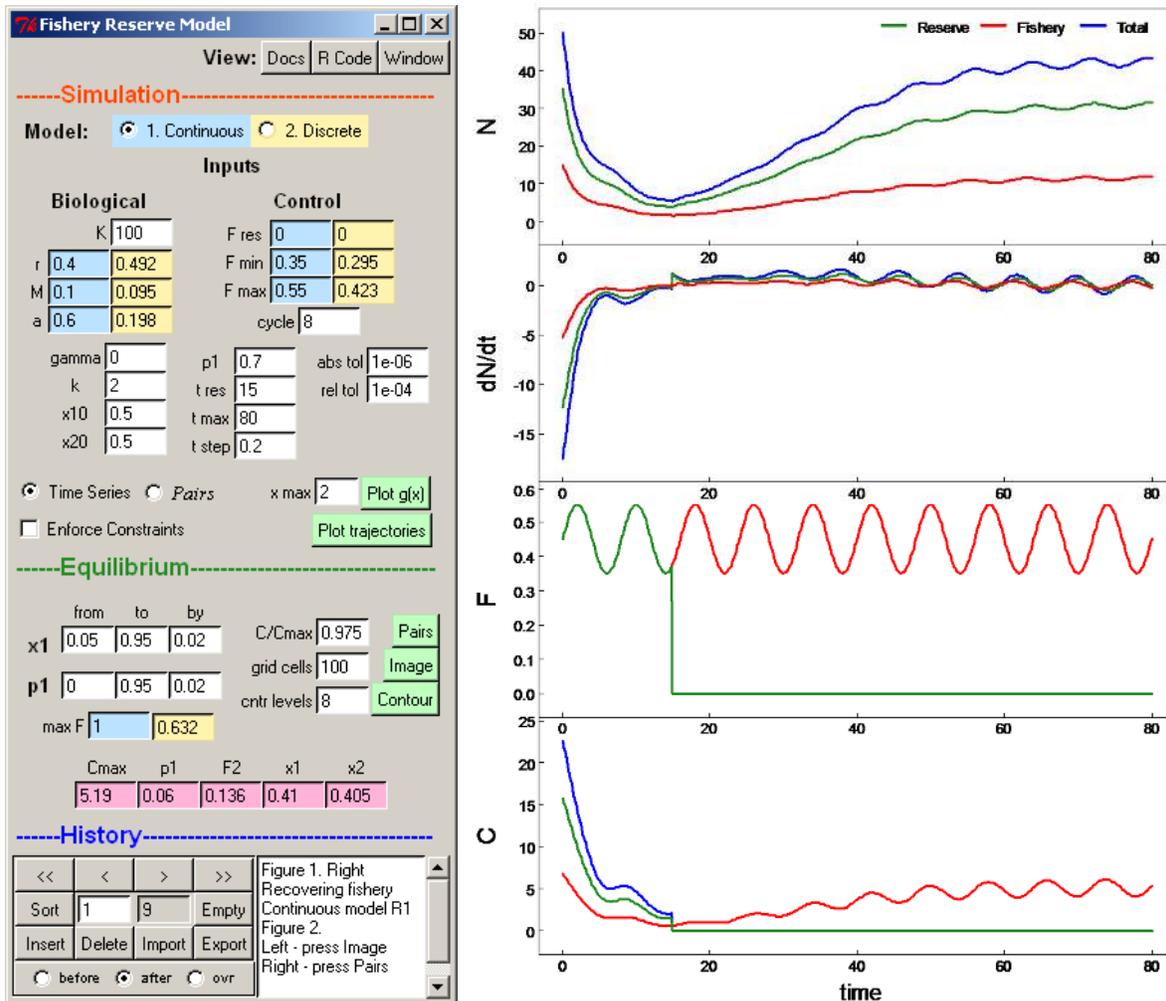


Figure 13. *FishRes* – Recovery of a heavily fished population after establishing a reserve. The GUI (left) shows all input values (parameters and controls). The selected continuous time model uses input values common to both models (white background) and values specific to the continuous model (blue background). Corresponding values are computed for the discrete model (yellow background). Output trajectories (right) trace various results (N = population, dN/dt = instantaneous change in population, F = instantaneous fishing mortality, C = instantaneous catch) for the reserve and fishery. Fishing mortality follows a sinusoid determined by F_{\min} , F_{\max} , and the cycle length n .

The example *FishRes* (Figure 13) models a fish population associated with a marine reserve in continuous or discrete time (delay differential or difference equations, respectively). For details see Schnute et al. (2007), which can be viewed by pressing the «Docs» button in the GUI. The R packages *akima*, *PBSddesolve*, and *deSolve* are required.

7.3.2. FishTows – fishery tows

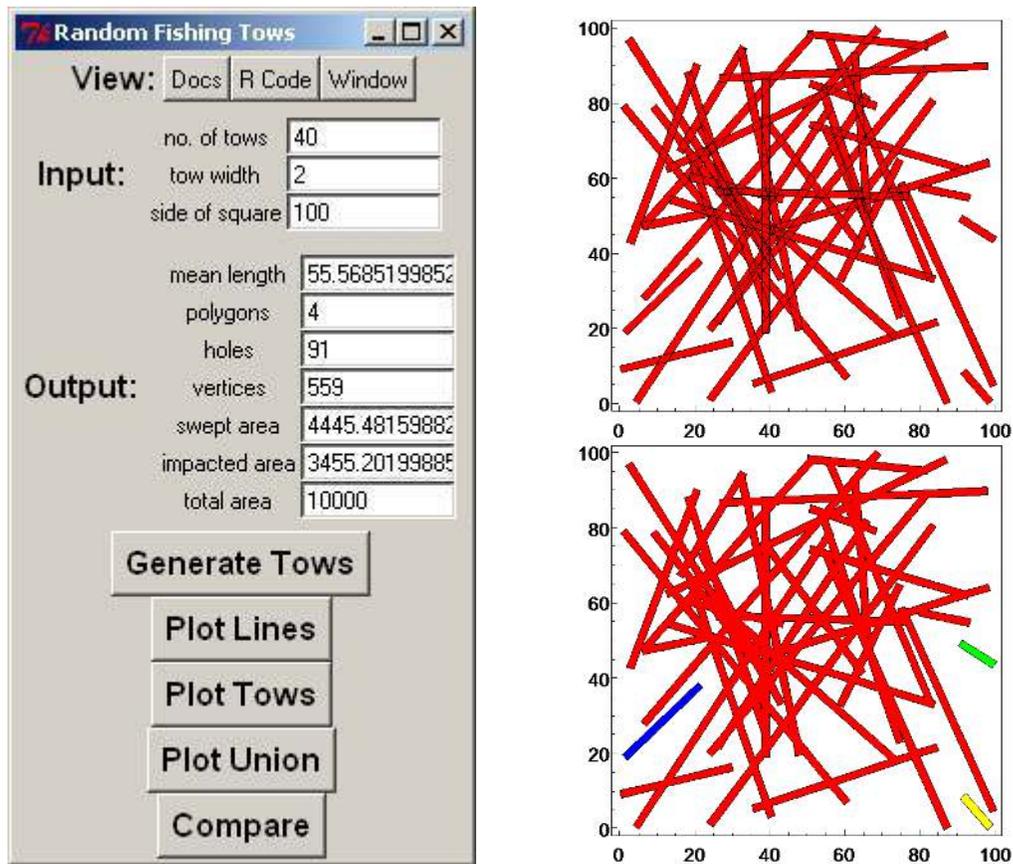


Figure 14. *FishTows* GUI (left) and simulated tow track (right). Tow track plots show 40 random tows in a square with side length 100. Each tow has width 2, and the rectangle encompasses 10,000 square units. Top: The individual rectangles, with 160 vertices, have areas that sum to 4,445 square units. Bottom: The union includes a complex polygon (red) and three isolated rectangles (blue, green, yellow) that cover only 3,455 square units. The complex polygon (red) has 547 vertices and 91 holes.

The example *FishTows* provides a simulator of fishery tow tracks using the *PBSmapping* package. The example demonstrates the difference between swept area and area impacted by trawls that often cover the same ground repeatedly. This application can be regarded an exotic random number generator, where tows initially join two points picked from a uniform random distribution within a square of a given side length. Three parameters (the number of tows, the tow width, the side length) determine several random variables, including the mean tow length, the areas swept and impacted, the numbers of polygons and holes in the union set of tows, and the number of vertices in the union. Each of these would also have a variance and an overall distribution generated by many runs of this example.

References

- Aitchison, J., and Brown, J.A.C. 1969. The lognormal distribution, with special reference to its uses in economics. Cambridge University Press. Cambridge, UK. xviii + 176 p.
- Chambers, J.M. 2008. Software for data analysis: Programming with R. Springer Science + Business Media, LLC. New York, NY. xiv + 498 p.
- Daalgard, P. 2001. [A primer on the R Tcl/Tk package](#). R News 1 (3): 27–31, Sep 2001.
- Daalgard, P. 2002. [Changes to the R Tcl/Tk package](#). R News 2 (3): 25–27, Dec 2002.
- Griewank A. (2000) Evaluating derivatives: principles and techniques of algorithmic differentiation. Frontiers in Applied Mathematics 19. Society for Industrial and Applied Mathematics
- Kronlund, A.R., Cox, S.P., and Cleary, J.S. 2012. [Management strategy evaluation in R \(mseR\): user's guide and simulation exercises](#). Can. Tech. Rep. Fish. Aq. Sci. 3001: vii + 52 p.
- Ligges, U. 2003. [R Help Desk: Package Management](#). R News 3(3): 37–39, Dec 2003.
- Ligges, U, and Murdoch, D. 2005. [R Help Desk: Make 'R CMD' work under Windows – an example](#). R News 5(2): 27–28, Nov 2005.
- Mittertreiner, A. and Schnute, J. 1985. [SIMPLEX: a manual and software package for easy nonlinear parameter estimation and interpretation in fishery research](#). Can. Tech. Rep. Fish. Aq. Sci. 1384: xi + 90 p.
- Ousterhout, J.K. 1994. Tcl and the Tk toolkit. Addison-Wesley, Boston, MA. 458 p.
- R Development Core Team (RDCT) (2011a). [R: A language and environment for statistical computing](#). R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.
- R Development Core Team (RDCT) (2011b). Writing R extensions. Version 2.13.1 (2011-07-08). R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-11-9.
- Raymond, E.S. 2000. [The Cathedral and the Bazaar](#), vers.3.0. Thyrus Enterprises, OPL v.2.
- Richards, L.J., Schnute, J.T., and Olsen, N. 1997. [Visualizing catch-age analysis: a case study](#). Can. J. Fish. Aq. Sci. 54: 1646–1658.
- Schnute, J. 1982. [A manual for easy nonlinear parameter estimation in fishery research with interactive microcomputer programs](#). Can. Tech. Rep. Fish. Aq. Sci. 1140: xvi + 115 p.
- Schnute, J.T. 2006. [Curiosity, recruitment, and chaos: a tribute to Bill Ricker's inquiring mind](#). Environmental Biology of Fishes 75: 95–110.
- Schnute, J.T., Boers, N.M., and Haigh, R. 2003. [PBS software: maps, spatial analysis, and other utilities](#). Can. Tech. Rep. Fish. Aq. Sci. 2496: viii + 82 p.
- Schnute, J.T., Boers, N.M., and Haigh, R. 2004. [PBS Mapping 2: user's guide](#). Can. Tech. Rep. Fish. Aq. Sci. 2549: viii + 126 p.
- Schnute, J.T., and Haigh, R. 2007. [Compositional analysis of catch curve data with an application to *Sebastes maliger*](#). ICES Journal of Marine Science 64: 218-233.

- Schnute, J.T., Haigh, R., and Couture-Beil, A. 2007. [Mathematical models of fish populations in marine reserves](#). Report on a Collaborative Project between Malaspina University-College and the Pacific Biological Station. February 2007, 24 pp.
- Schnute, J.T., and Richards, L.J. 1995. [The influence of error on population estimates from catch-age models](#). Can. J. Fish. Aq. Sci. 52: 2063-2077.
- Spiegelhalter, D., Thomas, A., Best, N., and Lunn, D. 2005. [WinBUGS User Manual](#). Also see [MRC Biostatistics Unit](#) at Cambridge University, UK.
- Thomas, A. 2004. BRugs User Manual (the R interface to BUGS), version 1.0. Dept. of Mathematics & Statistics, University of Helsinki.

Appendix A. Widget descriptions

This appendix lists PBSmodelling widgets in alphabetical order, except for “Window” which must exist before other widgets can be placed within it. Following a Tcl/Tk standard, we use a recursive grid design that allows grids within grids for flexible widget arrangement. The grid widget makes this possible. Furthermore, the notebook widget allows parts of a window to be selected by tabs, as in many other GUI applications.

For each widget, we include a brief description, a usage line showing the default arguments, a detailed list of arguments, and an illustrated example. In specifying a widget, the user can arrange named arguments in any order. If arguments are not named, they must appear in the order specified by the argument list, similar to named arguments in an R function.

The particular argument `sticky` needs a bit of explanation. It must be a string with zero or more of the characters N, E, W, or S that ‘stick’ the widget to the top, right, left, or bottom of its grid cell. These letters can have either upper or lower case and can appear in any order. The empty string (`sticky=""`) puts the widget in a central position of the cell. A string of length 1 binds the widget to the corresponding side (north, east, west, or south). The combinations NE, SE, SW, or NW will bind the widget to one of the corners. The combinations NS or EW will stretch the widget vertically or horizontally to the limits of its grid cell, while NEWS will stretch the widget in all directions to fill the cell.

Window

Description

Create a new window. Windows are palettes upon which widgets are placed. Each open window has a unique name. The function `closeWin` closes all windows unless a specific name (or vector of names) is provided by the user. Also, if `createWin` opens a window with a name already in use, the older window is closed before the new window is opened.

Usage

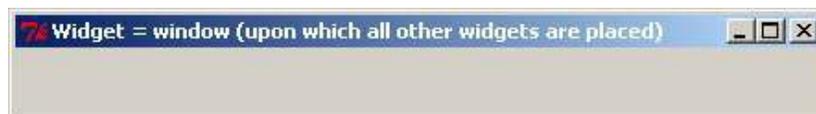
```
type=window name="window" title="" vertical=TRUE bg="#D4D0C8" fg="#000000"  
onclose="" remove=FALSE
```

Arguments

name unique name identifying an open window
title text to display in the window’s title line
vertical if TRUE, arrange widgets vertically, top to bottom, within the window
bg background colour for window
fg colour for label fonts
onclose name of function called when user closes the window by pressing 
remove if TRUE, remove from .PBSmod on closing

Example

```
window title="Widget = window (upon which all other widgets are placed)"
```



Grid

Description

Creates space for a rectangular block of widgets. Spaces must be filled. Widgets can be any combination of available widgets, including grid.

Usage

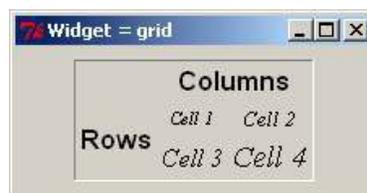
```
type=grid nrow=1 ncol=1 toptitle="" sidetitle="" topfont="" sidefont=""
topfg=NULL sidefg=NULL fg="black" topbg=NULL sidebg=NULL bg="" byrow=TRUE
borderwidth=1 relief="flat" sticky="" padx=0 pady=0
```

Arguments

nrow number of rows in the grid
ncol number of columns in the grid
toptitle title to place above grid
sidetitle title to place on the left side of the grid
topfont font for top labels – specify family (Times, Helvetica, or Courier), size (point size), & style (bold, italic, underline, overstrike), in any order
sidefont font for side labels – specify family (Times, Helvetica, or Courier), size (point size), & style (bold, italic, underline, overstrike), in any order
topfg colour for top title font
sidefg colour for side title font
fg colour for both top and side title fonts if topfg and sidefg are NULL
topbg background colour of the top title
sidebg background colour of the side title
bg backgr. col. of grid incl. top & side titles when topbg & sidebg are NULL
byrow if TRUE, create widgets across rows, otherwise down columns
borderwidth..... width of the border around the grid
relief type of border around the grid, where valid styles are:
 raised, sunken, flat, ridge, groove, solid
sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35
padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately
pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Example

```
grid 2 2 relief=groove toptitle=Columns sidetitle=Rows \  
topfont="Helvetica 12 bold" sidefont="Helvetica 12 bold"  
label text="Cell 1" font="times 8 italic"  
label text="Cell 2" font="times 10 italic"  
label text="Cell 3" font="times 12 italic"  
label text="Cell 4" font="times 14 italic"
```



Menu

Description

A menu grouping. Submenus can either be menu or menuitem.

Usage

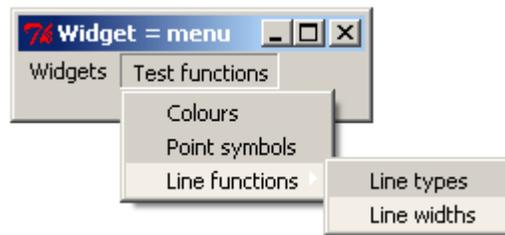
```
type=menu nitems=1 label font="" fg="" bg=""
```

Arguments

nitems number of items or submenus to include in the menu
label text to display as the menu label (required)
font font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
fg colour for menu fonts (only applicable for sub-menus)
bg background colour for menu (only applicable for sub-menus)

Example (assuming that the R functions have been defined)

```
window title="Widget = menu"  
menu nitems=1 label="Widgets"  
  menuitem label="Show arguments" func=showArgs  
menu nitems=3 label="Test functions"  
  menuitem label="Colours" func=testCol  
  menuitem label="Line types" func=testLty  
  menu nitems=2 label="Line functions"  
    menuitem label="Line widths" func=testLwd  
    menuitem label="Point symbols" func=testPch
```



Menuitem

Description

One of nitems following a menu command.

Usage

```
type=menuitem label font="" fg="" bg="" function action="menuitem"
```

Arguments

label text to display as the menu item label (required)

font font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
fg colour for menu item fonts
bg background colour for menu items
function R function to call when the menu item is clicked (required)
action string value associated whenever this widget is engaged

Button

Description

A button linked to an R function that runs a particular analysis and generates a desired output, perhaps including graphics.

Usage

```
type=button text="Calculate" font="" fg="black" bg="" disablefg=NULL width=0  
name=NULL function="" action="button" sticky="" padx=0 pady=0
```

Arguments

text text to display on the button
font font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
fg colour for label fonts
bg background colour for widget
disablefg colour for label fonts when state is disabled
width button width, the default 0 will adjust the width to the minimum required
name unique name to identify button for use with `setWidgetState`
function R function to call when the button is pushed (i.e., clicked by the mouse)
action string value associated whenever this widget is engaged
sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35
padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately
pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Example

```
window title="Widget = button"  
button text="Push Me"
```



Check

Description

A check box to turn a variable off or on, with corresponding values FALSE or TRUE (0 / 1).

Usage

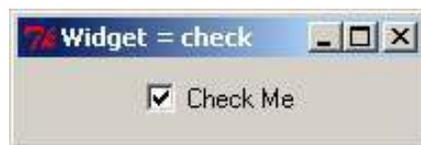
```
type=check name mode="logical" checked=FALSE text="" font="" fg="black" bg=""  
disablefg=NULL function="" action="check" edit=TRUE sticky="" padx=0 pady=0
```

Arguments

name name of R variable altered by this check box (required)
mode R mode for the associated variable, where valid modes are
 logical or numeric
checked if TRUE, the box is checked initially and the variable is set to TRUE or 1
text identifying text placed to the right of this check box
font font for labels – specify family (Times, Helvetica, or Courier), size (as
 point size), and style (bold, italic, underline, overstrike), in any
 order
fg colour for label fonts
bg background colour for widget
disablefg colour for label fonts when state is disabled
function R function to call when the check box is changed
action string value associated whenever this widget is engaged
edit if TRUE, the box's state can be modified by the user; if FALSE, the box is
 read-only
sticky option for placing the widget in its available space, as discussed in the
 introductory paragraphs for Appendix A on page 35
padx space used to pad the widget on the left and right; two values can be
 used to specify padding on the left and right separately
pady space used to pad the widget on the top and bottom; two values can be
 used to specify padding on the top and bottom separately

Example

```
window title="Widget = check"  
check name=junk checked=T text="Check Me"
```



Data

Description

An aligned set of entry fields for all components of a data frame. The data widget can accept a variety of modes. The user must keep in mind that rowlabels and collabels should conform to R naming conventions (no spaces, no special characters, etc.). If mode is logical, fields appear as a set of check boxes that can be turned on/off using mouse clicks.

Usage

```
type=data nrow ncol names modes="numeric" rowlabels="" collabels="" rownames="X"
colnames="Y" font="" fg="black" bg="" entryfont="" entryfg="black"
entrybg="white" noeditfg="black" noeditbg="gray" values="" byrow=TRUE
function="" enter=TRUE action="data" edit=TRUE width=6 borderwidth=0
sticky="" padx=0 pady=0
```

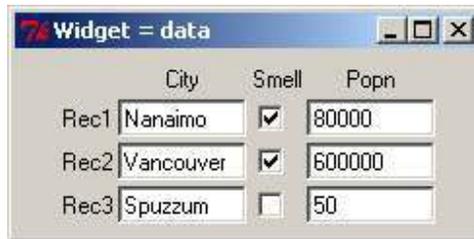
Arguments

nrow number of rows (required)
ncol number of columns(required)
names either one name or a set of nrow*ncol names used to store the data frame in R (required)
modes R modes for the data frame, where valid modes are:
 numeric, integer, complex, logical, character
rowlabels one of NULL, a single label, or a vector of nrow labels. The NULL label displays no labels and minimizes space. A single label displays a label to the left of the widget, and numbers each row (an empty label "" only numbers each row). A vector of nrow labels is used to specify a label for each row.
collabels one of NULL, a single label, or a vector of ncol labels. The NULL label displays no labels and minimizes space. A single label displays a label above the widget, and numbers each column (an empty label "" only numbers each column). A vector of ncol labels is used to specify a label for each column.
rownames string scalar or vector of length nrow to name the rows of the data frame
colnames string scalar or vector of length ncol to name the columns of the data frame
font font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
fg colour for label fonts
bg background colour for widget
entryfont font of entries appearing in input/output boxes
entryfg font colour of entries appearing in input/output boxes
entrybg background colour of input/output boxes
noeditfg font colour of entries appearing in input/output boxes when edit=FALSE
noeditbg background colour of input/output boxes when edit=FALSE
values default values (either one value for all data frame components or a set of nrow*ncol values)
byrow if TRUE and nrow*ncol names are used, interpret the names by row; otherwise by column. Similarly, interpret nrow*ncol initial values.
function R function to call when any entry in the data frame is changed

enter if TRUE, call the function only after the «Enter» key is pressed
action string value associated whenever this widget is engaged
edit if TRUE, the values can be modified by the user; if FALSE, the values are read-only
width character width to reserve for the each entry in the data frame
borderwidth..... a non-negative value specifying the amount of space to use for drawing a border (or margin) around the widget; the background colour of the space is determined by the bg value
sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35
padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately
pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Example

```
window title="Widget = data"  
data nrow=3 ncol=3 names=Census byrow=FALSE \  
  modes="character logical numeric" width=10 \  
  rowlabels="Rec1 Rec2 Rec3" collabels="City Smell Popn" \  
  values="Nanaimo Vancouver Spuzzum T T F 80000 600000 50"
```



Droplist

Description

A field in which a scalar variable (number or string) can be selected from a drop-down list.

Usage

```
type=droplist name values=NULL choices=NULL labels=NULL selected=1 add=FALSE  
font="" fg="black" bg="white" function="" enter=TRUE action="droplist"  
edit=TRUE mode="character" width=20 sticky="" padx=0 pady=0
```

Arguments

name name (required) of the R variable that will receive the selected choices from either values or choices
values vector of values to populate the drop-down selection; if NULL the values are taken from the R object named in choices
choices name of an R character vector object where elements will be the choices to populate the drop-down selection; if NULL the values are taken from the character vector specified by names

labels if supplied, labels is a vector with the same length as values, and is used as the contents of the drop-down list; however, values are return by `getWinVal`

selected the index of the pre-selected item in drop-down list

add if TRUE, the user can type in any text in addition to selecting a pre-defined item

font font for drop-down list items – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order

fg colour for drop-down list items

bg background colour for widget

function R function to call when the entry is changed

enter if TRUE, call the function only after the «Enter» key is pressed when add=TRUE; enter=FALSE, is not implemented.

action string value associated whenever this widget is engaged

edit if TRUE, the selected item can be changed by the user; if FALSE, the selected value is read-only and no other items can be selected

mode R mode for the value entered, where valid modes are: numeric, integer, complex, logical, character

width character width to reserve for the droplist

sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35

padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately

pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Note

To facilitate retrieving the index of the selected item, two additional variables are created by suffixing ".id" and ".values" to the given name. The "name.id" variable is only returned by `getWinVal`; the "name.values" variable can be retrieved with `getWinVal`, and can be set with `setWinVal` to change the selectable values dynamically after window creation.

Limitation: when `setWinVal` is used to modify the droplist "name.values", the labels are reset to NULL

Example

```
window title="Widget = droplist"  
droplist name=junk values="one two 'thirty three'" mode=character selected=3  
width=30  
droplist name=punk choices=state.name
```



Entry

Description

A field in which a scalar variable (number or string) can be altered.

Usage

```
type=entry name value="" width=20 label=NULL font="" fg="" bg="" entryfont=""  
entryfg="black" entrybg="white" noeditfg="black" noeditbg="gray" edit=TRUE  
password=FALSE function="" enter=TRUE action="entry" mode="numeric"  
sticky="" padx=0 pady=0
```

Arguments

name name of R variable corresponding to this entry (required)
value default value to display in the entry
width character width to reserve for the entry
label text to display above the entry box
font font for labels – specify family (Times, Helvetica, or Courier), size
(point size), & style (bold, italic, underline, overstrike), in any order
fg colour for label fonts
bg background colour for widget
entryfont font of entries appearing in input/output boxes
entryfg font colour of entries appearing in input/output boxes
entrybg background colour of input/output boxes
noeditfg font colour of input/output boxes when edit=FALSE
noeditbg background colour of input/output boxes when edit=FALSE
edit if TRUE, the entry value can be modified by the user; otherwise, the value
is read-only
password if TRUE, the value displayed in the GUI is masked with asterisks (****) to
protect sensitive information; otherwise, the value is displayed as normal
text
function R function to call when the entry is changed
enter if TRUE, call the function only after the «Enter» key is pressed
action string value associated whenever this widget is engaged
mode R mode for the value entered, where valid modes are:
numeric, integer, complex, logical, character
sticky option for placing the widget in its available space, as discussed in the
introductory paragraphs for Appendix A on page 35
padx space used to pad the widget on the left and right; two values can be
used to specify padding on the left and right separately
pady space used to pad the widget on the top and bottom; two values can be
used to specify padding on the top and bottom separately

Example

```
window title="Widget = entry"  
entry name=junk value="Enter something here" width=20 mode=character
```



History

Description

Allows the user to manage a temporary archive (history) of widget settings (records) through a panel of buttons:

<<	Go directly to the first record of the history.
<	Go to the previous record in the history.
>	Go to the next record in the history.
>>	Go directly to the last record in the history.
Sort	Sort the order of the records in the history.
<i>n</i>	Display window (white background) shows the current record.
<i>N</i>	Display window (grey background) shows total number of records in the history.
Empty	Remove all records from the history.
Insert	Add a new record (current widget settings) to the history, either before, after or overtop the current record.
Delete	Remove the current record from the history.
Import	Import a previously saved history (text file) to the history, either before or after the current record.
Export	Export the history to a text file.

Usage

```
type=history name="default" function="" import="" fg="black" bg=""  
    entryfg="black" entrybg="white" text=NULL textsize=0 sticky="" padx=0  
    pady=0
```

Arguments

name	name of history archive
function	R function to call when the history record counter is changed
import	file name of a saved history to load when the widget is called
fg	colour for label fonts
bg	background colour for widget
entryfg	font colour of entries appearing in input/output boxes
entrybg	background colour of input/output boxes
text	embed a text box for captions in the widget; the location of the text box is controlled by one of the following values: N, E, S, W or NULL for none
textsize	size of text box to display; if text=N or S, textsize controls the height; if text=E or W, the width is adjusted
sticky	option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35
padx	space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately
pady	space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Example

```
window title="Widget = history"  
vector length=3 names="alpha beta gamma" values="2 5 15"  
    history padx=20 pady=5
```



Image

Description

Embeds a graphics image in the current window. Support for GIF files only.

Usage

```
type=image file=NULL varname=NULL subsample=1 sticky="" padx=0 pady=0
```

Arguments

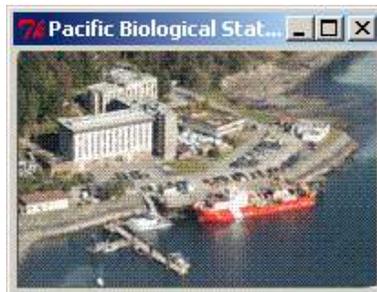
- file filename and path (if required) of GIF image to embed
- varname interpret the value of an R variable, identified by varname, as the filename of the image to embed; only one of the file or varname arguments can be supplied
- subsample reduce the size of the image by subsampling every subsampleth pixel, where subsample is an integer less than the width of the image
- sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35
- padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately
- pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Note

Image only supports GIF file formats

Example

```
window title="Pacific Biological Station"  
image file="pbs.gif"
```



Include

Description

Includes the specified window description file in the current window description file.

Usage

```
type=include file=NULL name=NULL
```

Arguments

```
file ..... file to include  
name ..... indirectly include a file by interpreting the value of an R variable, identified  
              by name, as the file to be included
```

Note

The window widget definition from the included file is ignored.

Example

```
window title="include - parent"  
label "hello world"  
include file=child.txt
```

```
# child.txt contents:  
window title="include - child"  
vector name="a b c d e"
```



Label

Description

Creates a text label. If the text argument is left blank, label emulates the null widget.

Usage

```
type=label text="" name="" mode="character" font="" fg="black" bg="" sticky=""  
justify="left" anchor="center" wraplength=0 width=0 padx=0 pady=0
```

Arguments

```
text ..... text to display in the label  
name ..... name of R variable corresponding to the label value; if name="", label is  
              static and cannot be changed with setWinVal  
mode ..... R mode for the label value where valid modes are:  
              numeric, integer, complex, logical, character
```

font font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order

fg colour for label fonts

bg background colour for widget

sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35

justify if there are multiple lines, then text is aligned to the left, center, or right.

anchor if a width is specified, anchor the text to the one of n, ne, e, se, s, sw, w, nw, or center locations of the widget. "w" for example, would anchor the text on the left side of the widget.

wraplength maximum number of characters to fit per line; text which is longer is split over multiple lines.

width width of the label widget

padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately

pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Example

```
window title="Widget = label"  
label text="Information Label"
```



Matrix

Description

An aligned set of entry fields for all components of a matrix. If the mode is logical, the matrix appears as a set of check boxes that can be turned on or off using mouse clicks.

Usage

```
type=matrix nrow ncol names rowlabels="" collabels="" rownames="" colnames=""  
font="" fg="black" bg="" entryfont="" entryfg="black" entrybg="white"  
noeditfg="black" noeditbg="gray" values="" byrow=TRUE function=""  
enter=TRUE action="matrix" edit=TRUE mode="numeric" width=6 borderwidth=0  
sticky="" padx=0 pady=0
```

Arguments

nrow number of rows (required)

ncol number of columns(required)

names either one name or a set of nrow*ncol names used to store the matrix in R (required)

rowlabels one of NULL, a single label, or a vector of nrow labels. The NULL label displays no labels and minimizes space. A single label displays a label to the left of the widget, and numbers each row (an empty label "" only

numbers each row). A vector of nrow labels is used to specify a label for each row.

collabels one of NULL, a single label, or a vector of ncol labels. The NULL label displays no labels and minimizes space. A single label displays a label above the widget, and numbers each column (an empty label "" only numbers each column). A vector of ncol labels is used to specify a label for each column.

rownames string scalar or vector of length nrow to name the rows of the matrix

colnames string scalar or vector of length ncol to name the columns of the matrix

font font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order

fg colour for label fonts

bg background colour for widget

entryfont font of entries appearing in input/output boxes

entryfg font colour of entries appearing in input/output boxes

entrybg background colour of input/output boxes

noeditfg font colour of entries appearing in input/output boxes when edit=FALSE

noeditbg background colour of input/output boxes when edit=FALSE

values default values (either one value for all matrix components or a set of nrow*ncol values)

byrow if TRUE and nrow*ncol names are used, interpret the names by row; otherwise by column. Similarly, interpret nrow*ncol initial values.

function R function to call when any entry in the matrix is changed

enter if TRUE, call the function only after the «Enter» key is pressed

action string value associated whenever this widget is engaged

edit if TRUE, matrix value can be modified by the user; if FALSE, the matrix is read-only

mode R mode for the matrix, where valid modes are:
 numeric, integer, complex, logical, character

width character width to reserve for the each entry in the matrix

borderwidth..... width of the border around the matrix widget

sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35

padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately

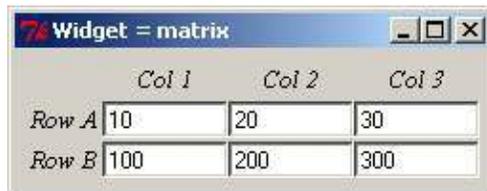
pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Example

```

window title="Widget = matrix"
matrix nrow=2 ncol=3 rowlabels="'Row A' 'Row B'" \
collabels="'Col 1' 'Col 2' 'Col 3'" names="a b c d e f" \
values="10 20 30 100 200 300" font="times 10 italic"

```



Notebook

Description

Creates a notebook widget comprising a set of pages that can be selected by tabs, where each page is visible when the corresponding tab is selected.

The number of notebook pages is determined by the length of the `tabs` vector argument. Each page of the notebook is specified by the widgets that immediately follow the declaration of a notebook widget. For example, if the notebook has three pages, `tabs` is a character vector of length three. Each of three widgets that follow `notebook` will be assigned to the pages in sequence. A collection of widgets can be embedded on each page by the use of `grid`.

A bug in an underlying Tcl/Tk library for `notebook` prevents combining font family and font style specifications for the `font` argument. For example, `font="Times italic 12"` cannot be specified but `font="Times 12"` or `font="italic"` can be specified. Specifying `font` with combinations of family and style will not cause an error but will not have the desired effect.

Usage

```
type= notebook tabs name=NULL selected=1 tabpos="top" font="" fg="" bg="" width=0
      height=0 homogeneous=FALSE arcradius=2 tabbevelsize=0 function=NULL
      action="notebook" sticky="we" padx=0 pady=0
```

Arguments

`tabs` a character vector of names for each tab – the length of the vector determines the number of tabs to use

`name` if specified, the index of the raised tab can be queried with `getWinVal`; other tabs can be raised programmatically with `setWinVal`.

`selected` default page to display

`tabpos` position tabs on the "top" or "bottom" of the notebook widget

`font` font for tab labels – specify family (Times, Helvetica, or Courier), size (as point size), or style (bold, italic, underline, overstrike)

`fg` colour of arrow used to horizontally scroll tabs (only applicable when the horizontal space required to display all tabs exceeds the width of the notebook widget)

`bg` background colour of the notebook page (but not tabs)

`width` width of the notebook

`height` height of the notebook

`homogeneous`..... if TRUE, all tabs have the same width, otherwise, each tab has a width determined by the length of the tab name

`arcradius` an integer in the range 0 to 8, specifying the rounding effect of the tab corners

`tabbevelsize`..... an integer in the range 0 to 8, specifying the amount of bevel the tabs should have; 0 effectively draws a rectangle, otherwise tabs have a trapezoidal look

`function` R function to call when a page is raised by selecting a tab

`action` string value associated whenever this widget is engaged

`sticky` option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35

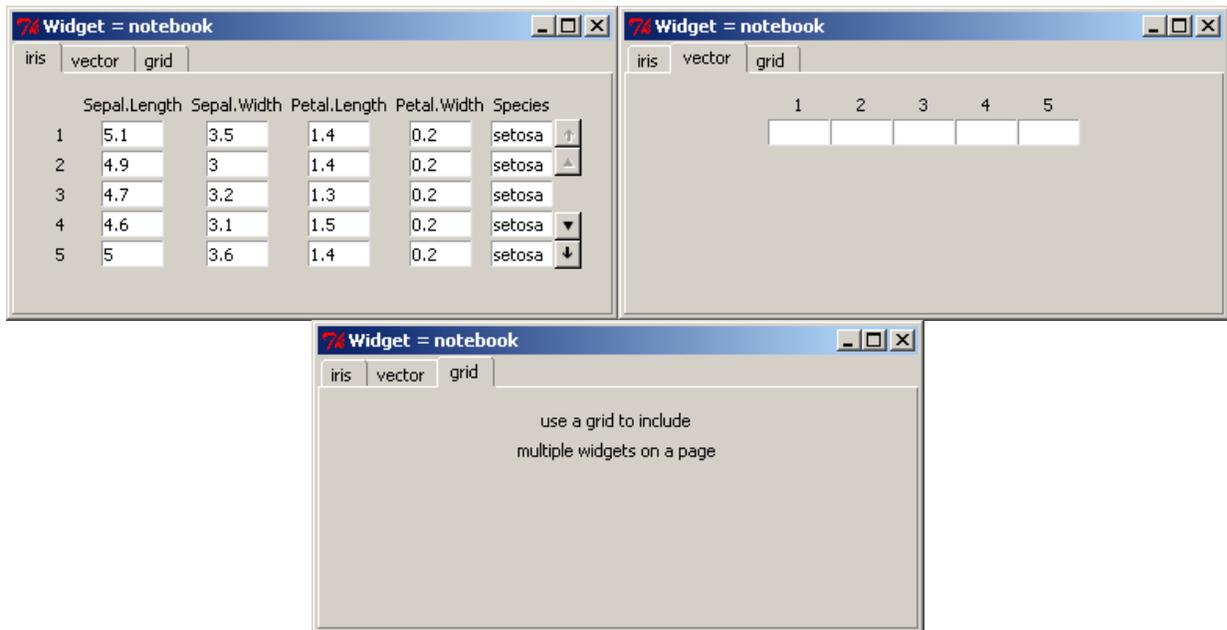
padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately
pady space used to pad the widget on the top and bottom

Note

Colour support is limited due to the underlying Tk/Tcl library implementation. It is not possible to specify colours for the tab font or background.

Example

```
window title="Widget = notebook"  
notebook tabs="iris vector grid" name=nb width=380 height=150  
  object iris rowshow=5  
  vector length=5 name=vec  
  grid 2 1  
    label "use a grid to include"  
    label "multiple widgets on a page"
```



Null

Description

Creates a null widget, useful for padding a grid with blank cells that appear as empty space.

Usage

```
type=null bg="" padx=0 pady=0
```

Arguments

bg background colour
padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately
pady space used to pad the label on the top and bottom

Example

```
grid 2 2 relief=raised toptitle=Top sidetitle=Side \  
topfont="Courier 10 bold" sidefont="courier 10 bold"  
label text="Here" font="courier 8"  
null  
null  
label text="There" font="courier 8"
```



Object

Description

A widget that represents the R-object specified – a vector becomes a vector widget, a matrix becomes a matrix widget, and a data frame becomes a data widget.

Usage

```
type=object name rowshow=0 font="" fg="black" bg="" entryfont="" entryfg="black"  
entrybg="white" noeditfg="black" noeditbg="gray" vertical=FALSE  
collabels=TRUE rowlabels=TRUE function="" enter=TRUE action="data"  
edit=TRUE width=6 borderwidth=0 sticky="" padx=0 pady=0
```

Arguments

name name of object (vector, matrix, or data frame) to convert to a widget (required)
rowshow number of rows to display on the screen; if rowshow=0 or rowshow>=rows(name) then all rows will be displayed

font font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order

fg colour for label fonts

bg background colour for widget

entryfont font of entries appearing in input/output boxes

entryfg font colour of entries appearing in input/output boxes

entrybg background colour of input/output boxes

noeditfg font colour of entries appearing in input/output boxes when edit=FALSE

noeditbg background colour of input/output boxes when edit=FALSE

vertical only applicable when the R-object is a vector; if TRUE , display the vector as a vertical column with labels on the left; otherwise display it as a horizontal row with labels above

collabels if TRUE, display the object’s column names, if FALSE, no column labels are displayed

rowlabels if TRUE, display the object’s row names, if FALSE, no row labels are displayed

function R function to call when any entry in the vector is changed

enter if TRUE, call the function only after the «Enter» key is pressed

action string value associated whenever this widget is engaged

edit if TRUE, the object’s values can be changed by the user; otherwise, the values are read-only

width character width to reserve for the each entry in the vector

borderwidth..... width of the border around the text box

sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35

padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately

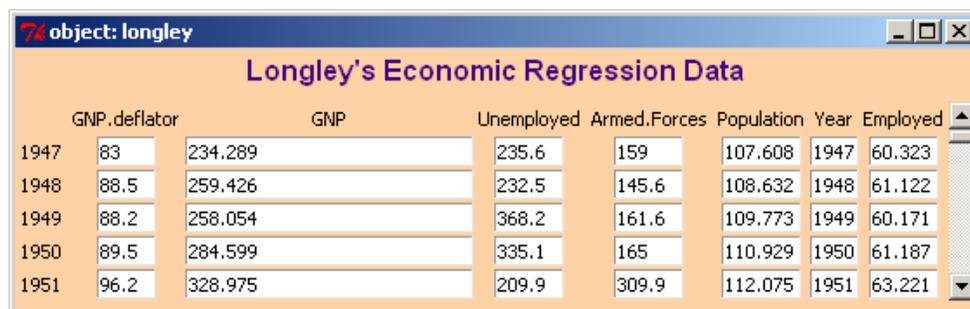
pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Note

When scrolling is enabled, the up, down, page up, and page down keys can be used to scroll. The keys are only enabled when some entry box in the object is selected.

Example

```
window bg="#ffd2a6" title="Object: longley"  
label text="Longley\'s Economic Regression Data" font="bold 12" \  
fg="#400080" pady=0 sticky=S  
object name=longley rowshow=5 width="5 27 6 6 7 4 6" pady=5
```



	GNP.deflator	GNP	Unemployed	Armed.Forces	Population	Year	Employed
1947	83	234.289	235.6	159	107.608	1947	60.323
1948	88.5	259.426	232.5	145.6	108.632	1948	61.122
1949	88.2	258.054	368.2	161.6	109.773	1949	60.171
1950	89.5	284.599	335.1	165	110.929	1950	61.187
1951	96.2	328.975	209.9	309.9	112.075	1951	63.221

Progressbar

Description

A progress indicator widget. The progressbar status can be animated by updating the widget state using `setWinVal`.

Usage

```
type=progressbar name value=0 maximum=100 style="normal" width=NULL
  height=NULL vertical=FALSE fg=NULL bg=NULL relief="sunken"
  borderwidth=2 sticky="" padx=0 pady=0
```

Arguments

name name of the progressbar
value initial value of the widget variable
maximum the maximum value of the widget variable (must be greater than zero)
style one of normal, incremental, infinite or nonincremental_infinite

if normal, a bar is displayed within a framed area with length proportional to value scaled to maximum. Updates of the widget state using `setWinVal` adjust the bar length to the passed positive value

if incremental, the widget behaves like the normal style with one exception: the positive value passed with `setWinVal` is added to the bar length rather than being used to set the bar length

if infinite, a bar segment is displayed within a framed area. Updating of the bar value using `setWinVal` advances the bar segment first from left to right and then from right to left by the specified positive integer increment.

if nonincremental_infinite, the widget behaves like the infinite style with one exception: the positive integer value passed with `setWinVal` is used to set the position of the bar segment. The bar segment moves from left to right if variable value (modulo maximum) is less than maximum/2 and from right to left if value is greater than maximum/2

width the width of the progressbar widget
height the height of the progressbar widget
vertical if TRUE, orient the progressbar vertically starting at the bottom for zero, moving upwards until maximum is reached, otherwise, orient the widget horizontally and move from left to right
fg foreground colour of the progress indicator
bg background colour for widget
relief type of border around the text, where valid styles are:
 raised, sunken, flat, ridge, groove, solid
borderwidth..... width of the border around the widget
sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35
padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately

pad space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Note

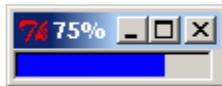
Animation of the progressbar widget to indicate the state of ongoing calculations can be achieved by repeated calls to `setWinVal` that reference the progressbar name.

If the value set with `setWinVal` is negative the progressbar is not displayed (i.e., it is hidden by drawing it "flat" using the background color), if the value is 0, the progressbar is reinitialized. Positive values work as described by the `style` argument and `infinite` and `nonincremental_infinite` styles only accept integers.

Run the `testWidgets()` function and select "progressbar" for an example of how to programmatically manipulate the progressbar widget for each of the `style` options.

Example

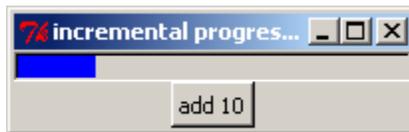
```
window title="75%"  
progressbar name=status fg=blue value=0.75 maximum=1.0
```



```
window title="infinite"  
progressbar name=status style=infinite fg=blue value=80
```



```
window title="incremental progressbar"  
progressbar name=status style=incremental \  
value=20 maximum=100 fg=blue width=200  
button text="add 10" function=doAction \  
action=setWinVal(c(status=10))
```



Radio

Description

One of a set of mutually exclusive radio buttons for making a particular choice. Buttons with the same value for name act collectively to define a single choice among the alternatives.

Usage

```
type=radio name value text="" font="" fg="black" bg="" function="" action="radio"
edit=TRUE mode="numeric" selected=FALSE sticky="" padx=0 pady=0
```

Arguments

name name of R variable altered by this radio button, where radio buttons with the same name define a mutually exclusive set (required)

value value of the variable when this radio button is selected (required)

text identifying text placed to the right of this radio button

font font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order

fg colour for label fonts

bg background colour for widget

function R function to call when this radio button is selected

action string value associated whenever this widget is engaged

edit if TRUE, the selected radio options can be changed; otherwise, the radio values are read-only

mode R mode for the value associated with this button, where valid modes are: numeric, integer, complex, logical, character

selected if TRUE, the radio button is selected (switched on)

sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35

padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately

pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Example

```
window title="Widget = radio"
grid 1 4
  radio name=junk value=0 text="None"
  radio name=junk value=1 text="Option A"
  radio name=junk value=2 text="Option B"
  radio name=junk value=3 text="Option C"
```



Slide

Description

A slide bar that sets the value of a variable. This widget only accepts integer values.

Usage

```
type=slide name from=0 to=100 value=NA showvalue=FALSE orientation="horizontal"  
font="" fg="black" bg="" function="" action="slide" sticky="" padx=0 pady=0
```

Arguments

name name of the numeric R variable corresponding to this slide bar (required)
from minimum value of the variable (must be an integer)
to maximum value of the variable (must be an integer)
value initial slide value, where the default is the specified from value
showvalue if TRUE, display the current slide value above the slide bar
orientation..... direction for orienting the slide bar: horizontal or vertical
font font for labels – specify family (Times, Helvetica, or Courier), size (as
point size), and style (bold, italic, underline, overstrike), in any
order
fg colour for label fonts
bg background colour for widget
function R function to call when the slide value is changed
action string value associated whenever this widget is engaged
sticky option for placing the widget in its available space, as discussed in the
introductory paragraphs for Appendix A on page 35
padx space used to pad the widget on the left and right; two values can be
used to specify padding on the left and right separately
pady space used to pad the widget on the top and bottom; two values can be
used to specify padding on the top and bottom separately

Example

```
window title="Widget = slide"  
slide name=junk from=1 to=1000 value=225 showvalue=T
```



SlidePlus

Description

An extended slide bar that also displays a minimum, maximum, and current value. This widget accepts real numbers.

Usage

```
type=slideplus name from=0 to=1 by=0.01 value=NA font="" fg="black" bg=""  
  entryfont="" entryfg="black" entrybg="white" function="" enter=FALSE  
  action="slideplus" sticky="" padx=0 pady=0
```

Arguments

name name of the numeric R variable corresponding to this slide bar (required)
from minimum value of the variable
to maximum value of the variable
by minimum amount for changing the variable's value
value initial slide value, where the default is the specified from value
font font for min/max labels – family (Times, Helvetica, or Courier), size
(point size), & style (bold, italic, underline, overstrike), in any order
fg colour for min/max label fonts
bg background colour for widget
entryfont font for entry widgets – family (Times, Helvetica, or Courier), size
(point size), & style (bold, italic, underline, overstrike), in any order
entryfg colour for entry widget fonts
entrybg background colour for entry widgets
function R function to call when the slide value is changed
enter if TRUE, call the function only after the «Enter» key is pressed while the
focus is within one of the text boxes
action string value associated whenever this widget is engaged
sticky option for placing the widget in its available space, as discussed in the
introductory paragraphs for Appendix A on page 35
padx space used to pad the widget on the left and right; two values can be
used to specify padding on the left and right separately
pady space used to pad the widget on the top and bottom; two values can be
used to specify padding on the top and bottom separately

Note

To facilitate retrieving and setting the minimum and maximum values, two additional variables are created by suffixing ".max" and ".min" to the given name.

Example

```
window title="Widget = slideplus"  
slideplus name=junk from=0 to=1 by=0.01 value=0.75
```



Spinbox

Description

A field in which a scalar variable can be incremented or decremented by a fixed value within a range of values.

Usage

```
type=spinbox name from to by=1 value=NA label="" font="" fg="black" bg=""  
entryfont="" entryfg="black" entrybg="white" function="" enter=TRUE  
edit=TRUE action="spinbox" width=20 sticky="" padx=0 pady=0
```

Arguments

name name of the R variable containing the text (required)
from minimum value of the variable
to maximum value of the variable
by minimum amount for changing the variable's value
value initial value; if NA, set the initial value to from
label text to display to the right of this spinbox
font font for labels – specify family (Times, Helvetica, or Courier), size (point size), & style (bold, italic, underline, overstrike), in any order
fg colour for label fonts
bg background colour for label
entryfont font for labels – specify family (Times, Helvetica, or Courier), size (point size), & style (bold, italic, underline, overstrike), in any order
entryfg colour for spinbox entry value and arrows
entrybg background colour for spinbox
function R function to call when the slide value is changed
enter if TRUE, call the function only after the <Enter> key is pressed while the focus is within the text box
edit if TRUE, value can be changed by user; otherwise, the value is read-only
action string value associated whenever this widget is engaged
width character width to reserve for the entry
sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35
padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately
pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Note

The values of the spinbox can be adjusted up/down with up/down arrows on the keyboard.

Example

```
window title="Widget = spinbox"  
spinbox name=spun from=0 to=100 by=12.5 value=50 label="Showcase showdown"  
bg=lightyellow font=bold entryfg=purple
```



Table

Description

A spreadsheet-like widget that can display and edit data in tabular format.

Usage

```
type=table name rowshow=0 font="" fg="black" bg="white" rowlabels="" collabels=""  
function="" action="table" edit=TRUE width=10 sticky="" padx=0 pady=0
```

Arguments

name name of object (vector, matrix, or data frame) to convert to a widget (required)

rowshow number of rows to display on the screen; if rowshow=0 then the table height is maximized and the number is determined automatically

font font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order

fg colour for label fonts

bg background colour for widget

rowlabels a vector of nrow labels used to label rows; if rowlabels="", then the object's row names are used; if NULL, no labels are displayed

collabels a vector of ncol labels used to label columns; if collabels="", then the object's column names are used; if NULL, no labels are displayed

function R function to call when any entry in the vector is changed

action string value associated whenever this widget is engaged

edit if TRUE, the object's values can be changed by the user; otherwise, the values are read-only

width character width to reserve for the each entry; if a vector of widths is given, then each element corresponds to a different column

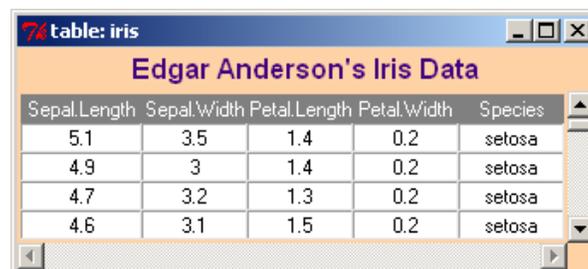
sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35

padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately

pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Example

```
window bg="#ffd2a6" title="table: iris"  
label text="Longley\'s Economic Regression Data" font="bold 12" \  
fg="#400080" pady=0 sticky=S  
table name=iris rowshow=5 rowlabels=NULL
```



Text

Description

An information text box that can display messages, results, or whatever the user desires. The displayed information can be either fixed or editable.

Usage

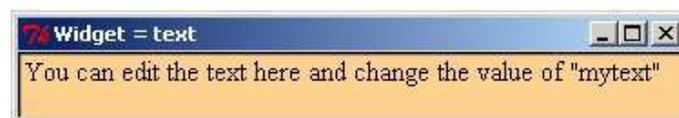
```
type=text name height=8 width=30 edit=FALSE scrollbar=TRUE fg="black" bg="white"
mode="character" font="" value="" borderwidth=1 relief="sunken" sticky=""
padx=0 pady=0
```

Arguments

name name of the R variable containing the text (required)
height text box height
width text box width
edit if TRUE, the user can edit the value stored in name
scrollbar if TRUE, a scroll bar is added to the right of the text box
fg colour for label fonts
bg background colour specified in hexadecimal format; e.g.,
 rgb(255,209,143,maxColorValue=255) yields "#FFD18F"
mode R mode for the value associated with this widget, where valid modes are:
 numeric, integer, complex, logical, character
font font for labels – specify family (Times, Helvetica, or Courier), size (as
 point size), and style (bold, italic, underline, overstrike), in any
 order
value default value to display in the text
borderwidth..... width of the border around the text box
relief type of border around the text, where valid styles are:
 raised, sunken, flat, ridge, groove, solid
sticky option for placing the widget in its available space, as discussed in the
 introductory paragraphs for Appendix A on page 35
padx space used to pad the widget on the left and right; two values can be
 used to specify padding on the left and right separately
pady space used to pad the widget on the top and bottom; two values can be
 used to specify padding on the top and bottom separately

Example

```
window title="Widget = text"
text name=mytext height=2 width=55 bg="#FFD18F" font="times 11"
borderwidth=1 relief="sunken" edit=TRUE \
value="You can edit text here & change value of \"mytext\""
```



Vector

Description

An aligned set of entry fields for all components of a vector. If the mode is logical, the vector appears as a set of check boxes that can be turned on or off using mouse clicks.

Usage

```
type=vector names length=0 labels="" values="" vecnames="" font="" fg="black"
bg="" entryfont="" entryfg="black" entrybg="white" noeditfg="black"
noeditbg="gray" vertical=FALSE function="" enter=TRUE action="vector"
edit=TRUE mode="numeric" width=6 borderwidth=0 sticky="" padx=0 pady=0
```

Arguments

names either one name (for a whole vector) or a vector of names for individual variables used to store the values in R (required)

length required only if a single name is given for a vector of length greater than 1

labels one of "", NULL, a single label, or a vector of length labels. The "" label uses the value of names as labels, if names only contains a single name, then elements are numbered. The NULL label displays no labels and minimizes space. A single label displays a label for the entire widget, and numbers elements. A vector of labels displays a label for each element of the array.

values default values (either one value for all vector components or a vector of length values)

vecnames string vector of length length to name the scalars or vector

font font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order

fg colour for label fonts

bg background colour for widget

entryfont font of entries appearing in input/output boxes

entryfg font colour of entries appearing in input/output boxes

entrybg background colour of input/output boxes

noeditfg font colour of entries appearing in input/output boxes when edit=FALSE

noeditbg background colour of input/output boxes when edit=FALSE

vertical if TRUE, display the vector as a vertical column with labels on the left; otherwise display it as a horizontal row with labels above

function R function to call when any entry in the vector is changed

enter if TRUE, call the function only after the «Enter» key is pressed

action string value associated whenever this widget is engaged

edit if TRUE, the vector's values can be changed by the user; otherwise, the values are read-only

mode R mode for the vector, where valid modes are:
numeric, integer, complex, logical, character

width character width to reserve for the each entry in the vector

borderwidth a non-negative value specifying the amount of space to use for drawing a border (or margin) around the widget; the background colour of the space is determined by the bg value

sticky option for placing the widget in its available space, as discussed in the introductory paragraphs for Appendix A on page 35

padx space used to pad the widget on the left and right; two values can be used to specify padding on the left and right separately

pady space used to pad the widget on the top and bottom; two values can be used to specify padding on the top and bottom separately

Example

```
window title="Widget = vector"  
vector length=4 names="a b g d" labels="alpha beta gamma delta" \  
values="100 0.05 1 5" font="times italic" width=6  
vector length=5 mode=logical names=chosen labels=choose \  
values="F T F T T"
```



Appendix B. Talk Description Files

This appendix specifies the structure and syntax for talk description files discussed in Section 6. Formally, such a file contains the five code elements listed there. A valid file must have one root `<talk>` element that contains one or more `<section>` elements. One `<talk>` element defines the root and the name of the corresponding GUI. This can include one or more `<section>` elements. Each `<section>` contains a mixture of the three *primitive* elements `<text>`, `<file>`, and `<code>`. These primitive elements occur in isolation; they cannot contain any other elements. Thus, we support only two levels of nesting: sections within a talk and primitives within a section. (Think of a talk root with section branches and primitive leaves.)

When `presentTalk()` runs a description file, it produces a control GUI like the one shown in Figure 5. Any declared `<section>`s, or `<file>`s automatically generate menu items in the GUI. These links can also appear as buttons within columns of the GUI’s lower section. By default, `<section>` buttons appear in the first column, and `<file>` buttons in the second column, although an author can overwrite these defaults. In this way, a talk description file allows an author to design both the talk’s content and the GUI used to present it.

Some tags allow the presentation to break at specified places. Specifically, a break produces a message in the R console indicating that the speaker must press the “Go” button in the GUI to continue on to the next step of the presentation. During a break, the speaker can spontaneously type code into the R console to illustrate points of immediate interest. A conceptual *slide* consists of all material between one break and the next.

We end this appendix with a precise description of the purpose and syntax for each code element. Instead of alphabetical order, we use the more logical order: `<talk>`, `<section>`, `<text>`, `<file>`, and `<code>`. In particular, we identify the arguments (also called *attributes* in the XML literature) that are supported in the initial tag.

```
<talk> ... </talk>
```

Description

A code element that constitutes a talk

Usage

```
<talk name=(required)>
```

Arguments

name A string giving the name of the talk (required). It appears as the title of the control GUI. It must start with a letter and contain only alphanumeric characters and underscores.

Notes

A file must have exactly one `<talk>` element that contains at least one `<section>` element.

`<section> ... </section>`

Description

A code element that defines a section of a talk

Usage

```
<section name=(required) button="FALSE" col="1">
```

Arguments

name A string giving the name of the section (required). It appears in the control GUI as a menu item (under “Sections”) and possibly also as a button. It must start with a letter and contain only alphanumeric characters and underscores.

button A Boolean variable (TRUE or FALSE) that determines whether or not the GUI should add a button that selects the section, in addition to access by the menu.

col If a button is used, the column within which to place it in lower section of the GUI.

Notes

A `<talk>` must include at least one `<section>`, and each section must have a unique name. Although a `<talk>` tag is commonly followed by a `<section>` tag (the first section), this may not always be true. See the description of `<file>` below.

`<text> ... </text>`

Description

A primitive that specifies text to be printed (displayed) on the R console

Usage

```
<text break="TRUE">
```

Arguments

break A Boolean value (TRUE or FALSE) that specifies whether or not to break the presentation after displaying the text specified.

Notes

Line breaks in the description file correspond to line breaks in the displayed text. Keep lines short enough that they will fit into the R console with the large font size required for presentation (Section 6).

`<file> ... </file>`

Description

A primitive that specifies files to be opened by the operating system with `openFile()`

Usage

```
<file name=(required) button="FALSE" col="2" break="TRUE">
```

Arguments

- name A string giving the name for this group of files (required). It appears in the control GUI as a menu item (under “Files”) and possibly also as a button. It must start with a letter and contain only alphanumeric characters and underscores.
- button A Boolean variable (TRUE or FALSE) that determines whether or not the GUI should add a button that opens this group of files, in addition to the available menu item.
- col If a button is used, the column within which to place it in lower section of the GUI.
- break A Boolean value (TRUE or FALSE) that specifies whether or not to break the presentation after opening the group of files.

Notes

File names between <file> and </file> must appear as individual strings (separated by spaces or line breaks) that are suitable arguments for `openFile()`. Files without explicit paths are presumed to lie in the user’s working directory. As usual, the operating system must have an associated application or the `PBSmodelling` options must be set to associate extensions and applications (Sections 2.3 and 5.1 above).

Although a speaker may commonly introduce only one file at a time, it can sometimes be convenient to open several files in a single step. For example, they may all appear in a single text editor window, with tabs for selecting individual files.

If a <file> element appears between <talk> and the talk’s first <section>, the file group name will be added to the talk’s GUI. However, because the segment doesn’t belong to any section, it will not cause files to be opened at this point. The feature allows files to become part of a talk without having to open them at an explicit point.

<code> ... </code>

Description

A primitive that specifies R code to be executed on the R console

Usage

`<code show="TRUE" print="TRUE" break="print">`

Arguments

- show A Boolean value (TRUE or FALSE) that specifies whether or not to show the code snippet in the R console. If shown, each line of the intended code will be prefixed by the usual R command prompt “> ”.
- print A Boolean value (TRUE or FALSE) that specifies whether or not to print the results of running the R code.
- break A string (show, print, all, or none) describing where to introduce breaks in the code segment:
 - show – break only after showing the R code;
 - print – break only after printing the results;
 - all – break after showing the R code and again after printing the results;
 - none – do not break during this code segment.

Notes

The text between `<code>` and `</code>` normally consists of valid R code, although a speaker may choose to demonstrate the consequences of invalid code.

Line breaks in the text correspond to individual lines of R code. Keep lines short enough that they will fit into the R console with the large font size required for presentation, as discussed in Section 6.

Implementing a `<code>` element involves **a two-step process**. First, if `show=TRUE`, the code is shown on the R console. Second, regardless of argument settings, the code is executed. If `print=TRUE`, the results are printed on the R console. Notice particularly that **code execution takes place in the second step**.

The `break` argument acts independently from the `show` and `print` arguments. For example, an author might use both `print=FALSE` and `break=print` if the R calculation takes notable time and produces extensive output that should be suppressed. In this case, the `break` would indicate that the calculation is complete. Similarly, the arguments `show=FALSE` and `break=show` allow an author to suppress the display of a large block of R code, but still to introduce a break before the code is executed.

Reminder: XML characters must be ‘escaped’ (i.e., ‘<’ becomes ‘<’). Since this is ugly, users will probably want to wrap code with `<![CDATA[. . .]]>`

Appendix C. Building PBSmodelling and Other Packages

The R project defines a standard for creating a package of functions, data, and documentation. You can obtain a comprehensive guide to “Writing R Extensions” (R Development Core Team 2011b, `R-exts.pdf`) from the CRAN web site or the R GUI (see the References above). Ligges (2003) and Ligges and Murdoch (2005) provide useful introductions. We have designed `PBSmodelling` and a very simple enclosed package `PBStr` as prototypes for package development. This Appendix summarizes the steps needed to:

- C.1. install the required software;
- C.2. build `PBSmodelling` from source materials;
- C.3. write source materials for a new package and compile them;
- C.4. include C code in a package.

Our discussion applies only to package development on a computer running Microsoft Windows 2000, XP, or (maybe) later. We particularly highlight issues that have proved troublesome for us. The R library directory `PBSmodelling\PBStools` contains batch files that can assist the process. For example, you might locate this directory as `C:\Utils\R\R-3.4.1\library\PBSmodelling\PBStools`.

C.1. Installing Required Software

Building R packages requires a few pieces of free software. Duncan Murdoch currently maintains their availability and [installation instructions for Windows](#).

Users should periodically check this website for changes to the various software packages. We recommend installing each package on a path that does *not* include spaces. For example, avoid using `C:\Program Files`, even if that happens to be part of a package’s default path. In this appendix, we use `C:\Utils` as a root directory for all required software. The list below gives a brief summary of the required software (Murdoch provides links to these products).

1. [R](#) itself, currently version R-3.4.1 (install to `C:\Utils\R\R-3.4.1`), although builders of packages destined for CRAN should use the latest [R-devel build for Windows](#), which is updated daily. We assume that R is already installed from the CRAN web site and that it runs correctly on your computer. (See ‘Upgrading to the latest version of R’ below.) We also assume that the package `PBSmodelling` is installed in R.
2. [Rtools installer](#): Command line tools, GCC compilers, etc. (install to `C:\Utils\Rtools\`). Download and run the file `Rtools34.exe`. This version has dropped support for the gcc 4.6.3 toolchain and now uses the gcc 4.9.3 toolchain produced by Jeroen Ooms. The installation should create the subdirectories `\bin` for command line programs and `\mingw_32` and `\mingw_64` for the “Minimalist GNU” C compilers (32-bit and 64-bit) for Windows. These tools are *essential*. DO NOT plan to use programs with the same name in an installation of Cygwin or any other UNIX emulator that happens to be installed on your computer.
3. [MiKTeX](#): a LaTeX and pdftex package (install to `C:\Utils\MiKTeX`). This processor for TeX and LaTeX files helps typeset the Rd help files within a package. Download the “basic” installation file, and install these components only. You can add more LaTeX packages from the Internet later, as required. (MiKTeX often does this automatically.) Take some time to investigate the MiKTeX package manager (`mpm.exe` or go to the “Programs” menu and select “MiKTeX 2.9”, “Browse Packages”).

We recommend enhancing MiKTeX slightly, so that it can independently process the LaTeX files produced from R documentation files.

- a) Create a new subdirectory `\R` under the MiKTeX's directory for storing LaTeX styles and font definitions (e.g., `C:\Utils\MiKTeX\tex\latex`).
- b) Copy into it all files from `\texmf` in the R installation tree (e.g., `C:\WinApps\R\R-3.4.1\share\texmf`). These should include `Rd.sty`.
- c) Go to the "Start" menu, select "Programs" then "MiKTeX 2.9", and run the program "Settings". In the "General" tab, click the button marked "Refresh FNDB". This refreshes MiKTeX's file name database, so that it recognizes files in the new `\R` subdirectory.

Every user has a preferred editor; however, if you are still using Notepad.exe, you may wish to explore the freely available, open-source software called [Notepad++](#). Notepad++ is described as "a free (as in 'free speech' and also as in 'free beer') source code editor ... that supports several languages." Alternatively, the text editor [WinEdt](#) provides a convenient GUI for editing LaTeX files and operating MiKTeX. Combined with the R package `RWinEdt`, it can also serve as an editor and interface for R. However, it is available only as shareware that requires a fee for long-term use, unlike any other software mentioned here.

Upgrading to the latest version of R

1. Download the new `R-x.y.z` binary from a local CRAN mirror, such as the one at SFU: <http://cran.stat.sfu.ca/bin/windows/base/>
2. Uninstall the old version `R-a.b.c` («Start», «Programs», «R», «Uninstall R-a.b.c»). If you cannot find an uninstall program in the «Programs» menu, use the Control Panel in the usual way (slightly different between Windows XP and Windows VISTA).
3. Install the new version `R-x.y.z` to a new folder. Our default would be:
`C:\Utils\R\R-x.y.z\`
4. Find the library files for both versions of R in the directories:
`C:\Utils\R\R-a.b.c\library\`
`C:\Utils\R\R-x.y.z\library\`
Copy all subdirectories (packages) from version `a.b.c` to version `x.y.z`; but press «Shift»«No» to avoid overwriting packages just installed as part of the new version. You want to copy the optional packages, but not those that come with the standard installation.
5. Run the new GUI for `R-x.y.z`. From the menu, click «Packages», «Update packages ...», select a local mirror, and wait for any installed packages to be updated. To stay current, repeat this update step every week or two.
6. Remove the old R installation directory (`C:\Utils\R\R-a.b.c\`).

At the time of writing, the program to uninstall `R-a.b.c` has a small bug, because it does not actually remove all of the packages that come with the base distribution.

PBStools for building R packages

After the above pieces of software are installed, you're ready to start building R packages. For this purpose, create a new directory (e.g., `D:\Rdevel\`) that will contain your packages. Within the R library directory (`C:\Utils\R\R-3.4.1\library\`), find the subdirectory `PBSmodelling\PBStools` (not to be confused with the R package `PBStools`). Copy all the batch files there into your new packages directory. You should have these 11 files:

- `RPaths.bat`, `RPathCheck.bat` related to the installation;

- `unpackPBS.bat`, `checkPBS.bat`, `buildPBS.bat`, `packPBS.bat`, related to PBSmodelling;
- `Runpack.bat`, `Rcheck.bat`, `Rbuild.bat`, `Rpack.bat`, `RmakePDF.bat` related to the construction of new packages.

IMPORTANT: You need to change `RPaths.bat` so that it reflects the paths you chose in the above six installations. For example, your version of this batch file might contain the lines

```
set R_PATH=C:\Utils\R\R-3.4.1\bin\i386
or
set R_PATH=C:\Utils\R\R-3.4.1\bin\x64
set TOOLS_PATH=C:\Utils\Rtools\bin
set GCC_PATH=C:\Utils\Rtools\gcc-4.6.3\bin
set TEX_PATH=C:\Utils\MiKTeX\miktex\bin
```

Notice that each path, except `R_Path`, ends in a `bin` subdirectory.

Hopefully, your installation is now complete. In your new packages directory, run `RPathCheck.bat` from a command line or double-click the icon. This script verifies that a few essential files lie on the indicated paths. If everything is correct, you should see the message “All program paths look good”. Otherwise, you’ll see a warning about software that doesn’t appear on your specified paths.

If you view all the batch files with a text editor, you will see that they don’t use your system `PATH` environment variable. Instead, each one defines a new local path appropriate for building R packages (via `RPathCheck.bat`). A `SETLOCAL` command ensures that this change doesn’t alter your system’s permanent environment.

C.2. Building PBSmodelling

Once all the required software is installed, the batch files discussed above make it fairly easy to build PBSmodelling. We assume that you have already created the directory discussed in Appendix C.1, say `D:\Rdevel`, for building R packages and that it contains the relevant eight batch files. In particular, `RPaths.bat` should reflect your installation paths and `RPathCheck.bat` should report the message that “All program paths look good”. Then follow these steps:

1. On the [CRAN](#) web site, go to “Packages” on the left and find [PBSmodelling](#). Download the file `PBSmodelling_x.y.z.tar.gz` into `D:\Rdevel`. Then rename this file (or copy it and rename the copy) so that the version number is removed. You should now have the file `PBSmodelling.tar.gz` in `D:\Rdevel`.
2. In the development directory `D:\Rdevel`, double-click the icon for `unpackPBS.bat` or type the command `unpackPBS` in a corresponding command window. This should extract the contents of `PBSmodelling.tar.gz`, preserving directory structure, into a subdirectory `\PBSmodelling` with five subdirectories: `\data`, `\inst`, `\man`, `\R`, and `\src`.
3. Our batch file uses the command `tar -xvzf PBSmodelling.tar.gz`, where `tar.exe` appears in the `\Rtools` directory (Section C.1, step 3). The command line parameters specify a verbose (`v`) extraction (`x`) of the given file (`f`), after filtering with `gzip(z)`.

If you use other software for this extraction, please ensure that it is configured to handle UNIX files correctly. For example, “WinZip” has an option to extract a “TAR file with smart CR/LF conversion”. This must be turned off.

4. In the base directory `D:\Rdevel`, double-click the icon for `checkPBS.bat` or type the command `checkPBS` in a corresponding command window. If all software is installed

correctly and `D:\Rdevel\PBSmodelling` correctly represents the contents of the `.tar.gz` file, you should see a series of DOS messages reporting “OK” to various tests. A distinct pause might accompany the message: “checking whether package 'PBSmodelling' can be installed ...”.

5. You might also encounter a delay as MiKTeX downloads the LaTeX package `lmodern`, part of a larger package `lm`. If this is really slow, you can abort the process and install `lm` with the MiKTeX package manager, as discussed in step 5 of Section C.1. Choose a remote server near you. You only need to do this once. When it’s finished, run `checkPBS.bat` again.
6. Examine the new directory `D:\Rdevel\PBSmodelling.Rcheck` created by the check process in step 2. The text files `00check.log` and `00install.out` show detailed results.
7. In the base directory `D:\Rdevel`, double-click the icon for `buildPBS.bat` or type the command `buildPBS` in a corresponding command window. This creates the file `D:\Rdevel\PBSmodelling.zip`, which could be used to install `PBSmodelling` from a local zip file.
8. Again in the base directory `D:\Rdevel`, double-click the icon for `packPBS.bat` or type the command `packPBS` in a corresponding command window. This creates a new package distribution file `PBSmodelling_x.y.z.tar.gz` that replaces the one downloaded from CRAN in step 1.
9. Finally, type the command `RmakePDF PBSmodelling` in a command window for `D:\Rdevel`. This generates an indexed documentation file `PBSmodelling.pdf`. See Appendix D.2 for further details about the use of this file for producing this report.

If these steps all work without problems, you can feel confident that the requisite software is installed correctly and that you understand the basic steps needed to build R packages.

C.3. Creating a New R Package

R packages require a special directory structure. The R function `package.skeleton` automatically creates this structure, but (without further work) it does not produce a package that can be compiled. Although `PBSmodelling` has the requisite structure, it is perhaps too complicated to serve as a convenient prototype. For this reason, we include a small subset `PBStry` that illustrates the key details. You can make a new package simply by editing the files in `PBStry`. You need a suitable editor (e.g., Notepad++, WinEdt, or [UltraEdit](#)) to view and change various text files.

1. Start by locating the file `PBStry_x.y.z.tar.gz` in the R library directory `\PBSmodelling\PBStools`. Copy this file into your development directory (`D:\Rdevel`), and rename it (or copy and rename the copy) to obtain the file `PBStry.tar.gz`.
2. Remove any previous traces of `PBStry` in your development directory, such as subdirectories `PBStry`, `PBStry.Rcheck`, and `.Rd2pdf$`, along with the documentation file `PBStry.pdf`.
3. Follow steps similar to those in Section C.2 to unpack, check, build, re-package, and document `PBStry`. You must now use a DOS command window in `D:\Rdevel` to issue the five commands
`Runpack PBStry`
`Rcheck PBStry`
`Rbuild PBStry`
`Rpack PBStry`

RmakePDF PBSty

which invoke the batch files Runpack.bat, Rcheck.bat, Rbuild.bat, Rpack.bat and RmakePDF.bat. The first command should give you a new subdirectory \PBSty, along with its five subdirectories: \data, \inst, \man, \R, and \src.

4. Use your editor to open the file DESCRIPTION in the root directory \PBSty. This file, essential in every R package, contains key information in a special format (RDCT 2011b, Section 1.1.1). The following example illustrates a minimal set of required fields.
5. Package: MyPack
Version: 1.00
Date: 2008-12-31
Title: My R Package
Author: User of PBS Modelling
Maintainer: User of PBS Modelling
Depends: R (>= 2.6.0)
Description: My customized R functions
License: GPL (>= 2)
6. The package name in DESCRIPTION must agree with the directory name in which this file lies. For example, if you change PBSty to MyPack in DESCRIPTION and rename the directory from \PBSty to \MyPack, you have effectively changed the package name. Similarly, if you change the version to 1.01, you have effectively changed the version number that appears in the file names for distributing your package.
7. The subdirectory \PBSty\R contains all R code used by the package. For example, PBSty includes seven R functions (calcFib, calcFib2, calcGM, calcSum, findPat, pause, and view). The seven files could be combined into a single file (such as PBSty.R), but we use separate files here for clarity. The functions all have relatively simple code, hopefully comprehensible to users with limited R experience. Five of them come from PBSmodelling. Three of them (calcFib, calcFib2, calcSum) call compiled C code, as we discuss more completely in Section C.4 below.
8. By convention, the distinct file zzz.r defines code for initializing the package. In this case the function .First.lib, calls library.dynam to load a dynamic link library (PBSty.dll) created from compiled C code during the build process.
9. When a version number changes, the DESCRIPTION file must be changed accordingly. We also like to make a corresponding change in zzz.r, so that the version number appears on the R console when the library is loaded. PBSty illustrates this possibility for zzz.r.
10. The subdirectory \PBSty\data contains all data objects that come with the package. Here, the binary file QBR.rda holds a matrix of quillback rockfish (*Sebastes maliger*) sample data used in the CCA example above (Section 7.2.3). The same data matrix is called CCA.qbr.h1 in PBSmodelling.
11. If you want to add data to a new package, first create the object (e.g., myData) in R and then execute the command:
save(myData, file="myData.rda")
The object name must match the prefix in the file name, and the suffix must be .rda. Include the resulting file in your package's \data subdirectory.
12. The subdirectory \PBSty\man contains a documentation file for every object in the package. PBSty has six functions and one data set, so the \man subdirectory has seven corresponding R documentation files (*.Rd). An additional file PBSty.Rd documents the package as a whole. Rd files use a rather complex scripting language (RDCT 2011b,

Section 2) that can be converted to help files in several formats (PDF, HTML, text). For many packages, the examples in `PBSty` may provide adequate prototypes. They represent three distinct cases: functions (e.g., `calcGM.Rd`, `findPat.Rd`), data sets (`QBR.Rd`), and complete packages (`PBSty.Rd`).

13. The subdirectory `\PBSty\src` contains source code for C code to be compiled into the dynamic link library `PBSty.dll`. We include sample files to calculate Fibonacci numbers iteratively (`fib.c`, `fib2.c`) and to add the components of a numeric vector (`sum.c`). In Section C.4, we discuss the linkage between R code and compiled C functions.
14. Finally, the subdirectory `\PBSty\inst` contains files that are to be included directly in the R library tree for `PBSty` when the package is installed. The file `PBSty-Info.txt` briefly describes the context and purpose of the trial package.

If you have successfully followed the steps above, you have actually built two R packages, `PBSmodelling` and `PBSty`. Furthermore, you're reasonably familiar with the contents of `PBSty`. You can use the files in that small package as prototypes for writing your own R package, which might contain R code in the subdirectory `\R.data` in `\data`, C source code in `\src`, and R documentation in `\man`.

The larger package `PBSmodelling` offers more prototypes and uses a somewhat different style. The main directory includes the required `DESCRIPTION` file, plus a second file `NAMESPACE` that lists all objects available to a user of the package. Effectively, the namespace mechanism distinguishes between objects provided by the package and other (hidden) objects required for the implementation, but not intended for public use. Our `NAMESPACE` file contains the rather cryptic instruction: `exportPattern("^[^\\.]*")`. The R string `"^[^\\.]*"` translates to the regular expression `^[^\\.]*` that designates any pattern not starting with a period (`.`). We also export "dot" objects (with names in R that start with a period) using explicit patterns, for example: `exportPattern("^[^\\.]*.add")`. The `NAMESPACE` file must also import functions required from other packages. Because `PBSmodelling` relies on `tcltk`, the file includes the command: `import(tcltk)`.

In `PBSty`, without a namespace, the file `zzz.r` defines the initializing function `.First.lib`, as mentioned in step 8 above. By contrast, the namespace protocol in `PBSmodelling` requires a different name for the initializing function: `.onLoad` in `zzz.r`. Note that all R packages now require a `NAMESPACE` file.

In summary, we recommend building a new package by editing, adding, and deleting prototype files in `PBSty`. Our batch files can facilitate tests and debugging. For more advanced work, particularly packages with a namespace protocol, look at `PBSmodelling`. Have a current version of `RDCT` (2011b) available, and consult that manual when necessary. We find it useful to keep the PDF file open and to use Acrobat's search feature (Ctrl-F) to find topics of interest.

C.4. Embedding C Code

R provides two functions, `.C()` and `.Call()`, for invoking compiled C code. `PBSty` includes two simple examples that use `.C()`, probably the method of choice for simple packages. The `.Call()` function uses a more complex interface that offers better support for R objects, and another example illustrates that calling convention.

Table C1. *C representations of R data types.*

R Object	C Type
logical	int *
integer	int *
double	double *
complex	Rcomplex * ¹
character	char **

¹ Rcomplex is defined in Complex.h.

Calling C functions from R using .C()

The .C() calling convention uses the following key concepts:

- R must allocate the appropriate length and type of variables before calling a C function.
- R objects are transformed into an equivalent C type (Table C1), and a pointer to the value is passed into the C function. All values are returned by modifying the original values passed in.
- A C function called by .C() must have return type void, because values are returned only by accessing the predefined R function arguments.
- C code written for the shared DLL must not contain a main function.
- Within a C function, dynamically allocated memory must be de-allocated by the programmer before the function returns. Otherwise a memory leak will likely occur.
- .C() returns a list similar to the '...' list of arguments passed in, but reflecting any changes made by the C code. (See the help file for .C)

The function calcFib in PBStry illustrates an application of these concepts (Table C2). The R function uses C code to calculate the first n Fibonacci numbers iteratively, where a vector holds the last len numbers calculated. After ensuring that n and len satisfy obvious constraints, the R code creates a return array retArr of the appropriate length. The .C call passes n, len, and retArr by reference to the C function fibonacci. On exit, the vector out contains a list corresponding to the input variables n, len, and retArr, so that the third component out[[3]] holds the modified vector of values calculated by fibonacci. We encourage you also to examine a second example in PBStry, associated the files calcSum.R and sum.c.

Table C2. Two text files associated with a `.C()` call in `PBSty`. R code in the first file calls C code in the second.

File 1: calcFib.R

```
calcFib <- function(n, len=1) {  
  if (n<0) return(NA);  
  if (len>n) len <- n;  
  retArr <- numeric(len);  
  out <- .C("fibonacci", as.integer(n), as.integer(len),  
           as.numeric(retArr), PACKAGE="PBSty")  
  x <- out[[3]]  
  return(x) }
```

File 2: fib.c

```
void fibonacci(int *n, int *len, double *retArr) {  
  double xa=0, xb=1, xn=-1; int i,j;  
  /* iterative loop */  
  for(i=0;i<=*n;i++) {  
    /* initial conditions: fib(0)=0, fib(1)=1 */  
    if (i <= 1) { xn = i; }  
    /* fib(n) = fib(n-1) + fib(n-2) */  
    else {xn = xa + xb; xa = xb; xb = xn; }  
    /* save results if iteration i is within the range from n-len to n */  
    j = i - *n + *len - 1;  
    if (j >= 0) retArr[j] = xn;  
  } /* end loop */  
} /* end function */
```

Table C3. .Call() example adapted from PBSty, with two associated text files. R code in the first file calls C code in the second.

File 1: calcFib2.R

```
calcFib2 <- function(n, len=1) {  
  out <- .Call("fibonacci2", as.integer(n), as.integer(len),  
              PACKAGE="PBSmodelling")  
  return(out) }
```

File 2: fib2.c

```
#include <R.h>  
#include <Rdefines.h>  
SEXP fibonacci2(SEXP sexp_n, SEXP sexp_len) {  
  /* ptr to output vector that we will create */  
  SEXP retVals;  
  double *p_retVals, xa=0, xb=1, xn;  
  int n, len, i, j;  
  /* convert R variables into C 'int's */  
  len = INTEGER_VALUE(sexp_len);  
  n = INTEGER_VALUE(sexp_n);  
  /* Allocate space for the output vector */  
  PROTECT(retVals = NEW_NUMERIC(len));  
  p_retVals = NUMERIC_POINTER(retVals);  
  /* iterative loop */  
  for(i=0; i<=n; i++) {  
    /* initial conditions: fib(0)=0, fib(1)=1 */  
    if (i <= 1) { xn = i; }  
    /* fib(n) = fib(n-1) + fib(n-2) */  
    else { xn = xa + xb; xa = xb; xb = xn; }  
    /* save results if iteration i is within the range from n-len to n */  
    j = i - n + len - 1;  
    if (j >= 0) p_retVals[j] = xn;  
  } /* end loop */  
  UNPROTECT(1);  
  return retVals;  
} /* end fibonacci2 */
```

Calling C functions from R using .Call()

The .C() convention requires a fairly simple conversion of R objects into C types (Table C1). By contrast, .Call() provides extra structure that enables C to handle R objects directly (RDCT 2011b, Section 4.7). This function uses “S-expression” SEXP types defined in rinternals.h, a file in the \include directory of the R installation. An SEXP pointer can reference any type of R object. The .Call() convention uses the following key concepts:

- C functions called by R must accept only SEXP typed arguments. These arguments should be treated as read only.
- Similarly, C functions called by R must have SEXP return types.
- The Programmer must protect R objects from the R garbage collector, and must release protected objects before the function terminates. R provides macros for this task.
- C code written for the shared DLL must not contain a main function.
- Within a C function, dynamically allocated memory must be de-allocated by the programmer before the function returns. Otherwise a memory leak will likely occur.

The function `calcFib2` in Table C3 illustrates an application of these concepts. As before, the R function uses C code to calculate the first `n` Fibonacci numbers iteratively, where a vector holds the last `len` numbers calculated. (To save space, we've removed R code that checks constraints on `n` and `len`). The simple `.Call` to `fibonacci2` looks very natural. Input values `n` and `len` produce the output vector `out`, where the C code must somehow determine what `out` should be. Not surprisingly, it requires more complicated C code to make this happen.

The C function `fibonacci2` (Table C3) first loads header files that include the required definitions from R. All input and output variables belong to type `SEXP`. Other internal variables have the standard C types `double` and `int`. Functions like `INTEGER_VALUE()` convert R types into C types. The `SEXP` vector `retVals` of return values is created by the R constructor `NEW_NUMERIC()` and then protected from garbage collection by `PROTECT()`. After all required variables are defined and type cast correctly, the iterative loop of calculations follows the earlier example in Table B2. Finally, the only protected vector `retVals` is released by `UNPROTECT(1)`, and the standard closing command `return retVals` returns the output vector from `fibonacci2`.

Obviously, it takes some time and effort to become familiar with the specialized R types, constructors, and conversion functions. For this reason, it's probably easier at first to use `.C()`, rather than `.Call()`.

Appendix D. PBSmodelling Functions and Data

Section 1 of this appendix summarises the functions currently available in PBSmodelling. Additionally, there are possible hidden or ‘dot’ functions (not presented here) that reside in R’s NAMESPACE. These can be seen either using the triple colon convention on the command line (e.g., PBSmodelling::: .function) or through our function viewCode, which gathers function code for a specified package installed on the user’s computer. (R also provides a utility called fixInNamespace() for modifying NAMESPACE objects.) Section 2 of this appendix details how a user can generate a standard R manual for PBSmodelling, that includes a Table of Contents, help pages for all objects, and an index. The manual itself is also appended.

D.1. Objects in PBSmodelling

addArrows.....	Add Arrows to a Plot Using Relative (0:1) Coordinates
addHistory.....	Create Structures for a New History Widget
addLabel.....	Add a Label to a Plot Using Relative (0:1) Coordinates
addLegend.....	Add a Legend to a Plot Using Relative (0:1) Coordinates
backHistory.....	Create Structures for a New History Widget
calcFib.....	Calculate Fibonacci Numbers by Several Methods
calcGM.....	Calculate the Geometric Mean, Allowing for Zeroes
calcMin.....	Calculate the Minimum of a User-Defined Function
chooseWinVal.....	Choose and Set a String Item in a GUI
cleanProj.....	Launch a GUI for Project File Deletion
cleanWD.....	Launch a GUI for File Deletion
clearAll.....	Remove all R Objects From the Global Environment
clearHistory.....	Create Structures for a New History Widget
clearPBSext.....	Clear File Extension Associations
clearRcon.....	Clear the R Console
clearWinVal.....	Remove all Current Widget Variables
clipVector.....	Clip a Vector at One or Both Ends
closeWin.....	Close GUI Window(s)
compileC.....	Compile a C File into a Shared Library Object
compileDescription.....	Convert and Save a Window Description as a List
convSlashes.....	Convert Slashes from UNIX to DOS
createVector.....	Create a GUI with a Vector Widget
createWin.....	Create a GUI Window
declareGUIoptions.....	Declare Option Names that Correspond with Widget Names
doAction.....	Execute Action Created by a Widget
drawBars.....	Draw a Linear Barplot on the Current Plot
evalCall.....	Evaluate a Function Call
expandGraph.....	Expand the Plot Area by Adjusting Margins
exportHistory.....	Export a Saved History
findPat.....	Search a Character Vector to Find Multiple Patterns
findPrefix.....	Find a Prefix Based on Names of Existing Files
findProgram.....	Locates a program in the PATH environment variable
findSuffix.....	Find a Prefix Based on Names of Existing Files
firstHistory.....	Create Structures for a New History Widget
focusRgui.....	Focus on the RGui Window
focusWin.....	Set the Focus on a Particular Window
forwHistory.....	Create Structures for a New History Widget

genMatrix..... Generate Test Matrices for plotBubbles
getChoice..... Choose One String Item from a List of Choices
getGUIoptions..... Get PBS Options for Widgets
getOptions..... get and set user options
getOptionsFileName..... get and set filename used for saving and loading of options
getOptionsPrefix..... get and set GUI prefix of options class
getPBSext..... Get a Command Associated With a File Name
getPBSoptions..... Retrieve A User Option
getWinAct..... Retrieve the Last Window Action
getWinFun..... Retrieve Names of Functions Referenced in a Window
getWinVal..... Retrieve Widget Values for Use in R Code
getYes..... Prompt the User to Choose Yes or No
GT0..... Restrict a Numeric Variable to a Positive Value
importHistory..... Import a History List from a File
initHistory..... Create Structures for a New History Widget
isWhat..... Identify an Object and Print Information
jumpHistory..... Create Structures for a New History Widget
lastHistory..... Create Structures for a New History Widget
lisp..... List Objects in .PBSmodEnv Workspace
loadC..... Launch a GUI for Compiling and Loading C Code
loadOptions..... save and load options to and from disk
loadOptionsGUI..... load and save options values to and from a GUI
lucent..... Convert Solid Colours to Translucence
openExamples..... Open Example Files from a Package
openFile..... Open a File with an Associated Program
openUG..... Open Package User Guide
packList..... Pack a List with Objects
pad0..... Pad Numbers with Leading Zeroes
parseWinFile..... Convert a Window Description File into a List Object
pause..... Pause Between Graphics Displays or Other Calculations
pickCol..... Pick a Colour From a Palette and get the Hexadecimal Code
plotACF..... Plot Autocorrelation Bars From a Data Frame, Matrix, or Vector
plotAsp..... Construct a Plot with a Specified Aspect Ratio
plotBubbles..... Construct a Bubble Plot from a Matrix
plotCsum..... Plot Cumulative Sum of Data
plotDens..... Plot Density Curves from a Data Frame, Matrix, or Vector
plotFriedEggs..... Render a Pairs Plot as Fried Eggs and Beer
plotSidebars..... Plot Table as Horizontal Sidebars
plotTrace..... Plot Trace Lines from a Data Frame, Matrix, or Vector
presentTalk..... Run an R Presentation
promptWriteOptions..... Prompt the User to Write Changed Options
readList..... Read a List from a File in PBS Modelling Format
readPBSoptions..... Read PBS Options from an External File
resetGraph..... Reset par Values for a Plot
restorePar..... Get Actual Parameters from Scaled Values
rmHistory..... Create Structures for a New History Widget
runDemos..... Interactive GUI for R Demos
runExample..... Run a Single GUI Example Included with PBS Modelling
runExamples..... Run GUI Examples Included with PBS Modelling
saveOptions..... save and load options to and from disk
saveOptionsGUI..... load and save options values to and from a GUI

scalePar.....	Scale Parameters to [0,1]
selectDir.....	Display Dialogue: Select directory
selectFile.....	Display Dialogue: Open or Save File
setFileOption.....	Set a PBS File Path Option Interactively
setGUIoptions.....	Set PBS Options from Widget Values
setOptions.....	get and set user options
setOptionsFileName.....	get and set filename used for saving and loading of options
setOptionsPrefix.....	get and set GUI prefix of options class
setPathOption.....	Set a PBS Path Option Interactively
setPBSext.....	Set a Command Associated with a File Name Extension
setPBSoptions.....	Set A User Option
setwdGUI.....	Browse for Working Directory and Optionally Find Prefix
setWidgetColor.....	Update Widget Colour
setWidgetState.....	Update Widget State
setWinAct.....	Add a Window Action to the Saved Action Vector
setWinVal.....	Update Widget Values
show0.....	Convert Numbers into Text with Specified Decimal Places
showAlert.....	Display a Message in an Alert Window
showArgs.....	Display Expected Widget Arguments
showHelp.....	Display HTML Help Pages for Packages in Browser
showPacks.....	Show Packages Required But Not Installed
showRes.....	Show Results of Expression Represented by Text
showVignettes.....	Display Vignettes for Packages
sortHistory.....	Sort an Active or Saved History
tcall.....	Call Objects From Temporary Work Environment
testAlpha.....	Test Various Alpha Transparency Values
testCol.....	Display Named Colours Available Based on a Set of Strings
testLty.....	Display Line Types Available
testLwd.....	Display Line Widths
testPch.....	Display Plotting Symbols and Backslash Characters
testWidgets.....	Display Sample GUIs and their Source Code
tget.....	Get Objects From Temporary Work Environment
tprint.....	Print Objects From Temporary Work Environment
tput.....	Put Objects Into Temporary Work Environment
unpackList.....	Unpack List Elements into Variables
updateGUI.....	Update Active GUI With Local Values
view.....	View First/Last/Random n Elements/Rows of an Object
viewCode.....	View Package R Code
writeln.....	Write a List to a File in PBS Modelling Format
writePBSoptions.....	Write PBS Options to an External File

D.2. PBSmodelling Manual

The following pages show the standard R manual for PBSmodelling, including help pages for all objects, a table of contents, and an index. This manual also appears on the CRAN web site for [PBSmodelling](#).

To generate the pages that follow, the user should first ensure that R's style and font files have been copied to MiKTeX (see steps 5a-c in Section C.1). This enhancement may be necessary for the successful creation of a PDF manual.

Next we provide a batch file `RmakePDF.bat` to assist the user in building the manual. This method uses R's temporary latex output `Rd2.tex` and alters it using system and MiKTeX commands (e.g., `sed`, `latex`, `makeindex`, `dvips`, `ps2pdf`). The final result yields a PDF manual with letter-size (8.5" × 11") pages rather than A4, and page numbering beginning at a specified odd number to ensure that the next page becomes the front of a two-sided copy. Currently, R CMD `Rd2pdf` has no way to start page numbering at an arbitrary integer. (Note that the control file `Rd2dvi` has been removed by the CRAN mandarins.)

Method: On a command line, type the command:

```
RmakePDF PBSmodelling 83
```

which automatically generates the PDF manual `PBSmodelling.pdf` from the package's `*.Rd` files. Page numbering for this PDF begins with the number specified by the second argument of the above command. If the argument is not supplied, it defaults to 1.

The batch file issues the following command:

```
%R_PATH%\R CMD Rd2pdf --no-clean --no-preview %1
```

This command creates a temporary directory called `.Rd2pdfNNNN\` (where `NNNN` comprise an arbitrary set of four integers) containing `Rd2.tex` with the initial lines:

```
\documentclass[letterpaper]{book}
\usepackage[times,inconsolata,hyper]{Rd}
\usepackage{makeidx}
\usepackage[utf8,latin1]{inputenc}
\makeindex{}
\topmargin -0.25in \oddsidemargin 0in \evensidemargin 0in
\textheight 9in \textwidth 6.5in
\begin{document}
\setcounter{page}{83}
```

where a boldface red font indicates new lines added by `RmakePDF.bat` to the file `Rd2.tex` in the temporary folder. The altered tex file is saved as `PBSmodelling.tex` in the current working directory, and the following MiKTeX commands are issued:

```
latex -interaction=nonstopmode %1.tex
makeindex %1.idx
latex -interaction=nonstopmode %1.tex
makeindex %1.idx
latex -interaction=nonstopmode %1.tex
latex -interaction=nonstopmode %1.tex
dvips -q %1.dvi
ps2pdf %1.ps
```

The repetitive calls to `latex` and `makeindex` are a byproduct of a non-dynamic system where index references need to be updated several times. You should now have the PDF manual called `PBSmodelling.pdf`, which can be appended to the first 82 pages of this report.

The technique presented in this appendix can be applied to any package to produce a manual based on the `*.Rd` files. Readers may wish to go further and append their manual to more detailed instructions to produce a comprehensive User's Guide such as this one.

Page left blank intentionally