PBSmodelling 2.20: User's Guide

Jon T. Schnute, Alex Couture-Beil, Rowan Haigh, and Anisa Egeli

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

2009

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

Fisheries and Oceans Canada | Pêches et Océans Canada

Canada

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

Last update: July 28, 2009

Correct citation for this publication:

Schnute, J.T., Couture-Beil, A., Haigh, R., and Egeli, A. 2009. PBSmodelling 2.20: user's guide revised from Canadian Technical Report of Fisheries and Aquatic Science 2674: vi + 173 p. Last updated July 28, 2009

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Schnute, J.T., Couture-Beil, A., Haigh, R., and Egeli, A. 2009. PBSmodelling 2.20: user's guide revised from Can. Tech. Rep. Fish. Aquat. 2674: vi + 173 p. Last updated July 28, 2009.

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`, 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 2006a) 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` (to solve differential equations numerically), `ddesolve` (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* (http://www.cbc.ca/vinylcafe/), “We may not be big, but we’re small.”

Jon Schnute, December 2006; revised October 2008.

Update for Version 2.20

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 Farser University, added the new programming code that contributes to this significant upgrade.

The scope of our R packages has grown considerably over the last few years. We continue to find PBSmodelling useful in a variety of contexts. For further information, see the Google Code web sites referenced at http://code.google.com/p/pbs-software/.

Jon Schnute, July 2009

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 2006a). 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.

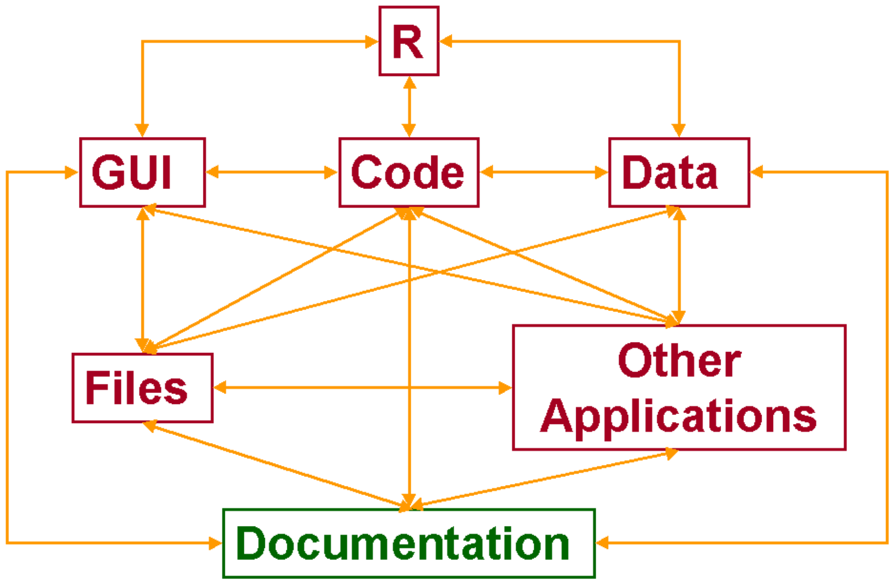


Figure 1. Tangled relationships among computer model components.

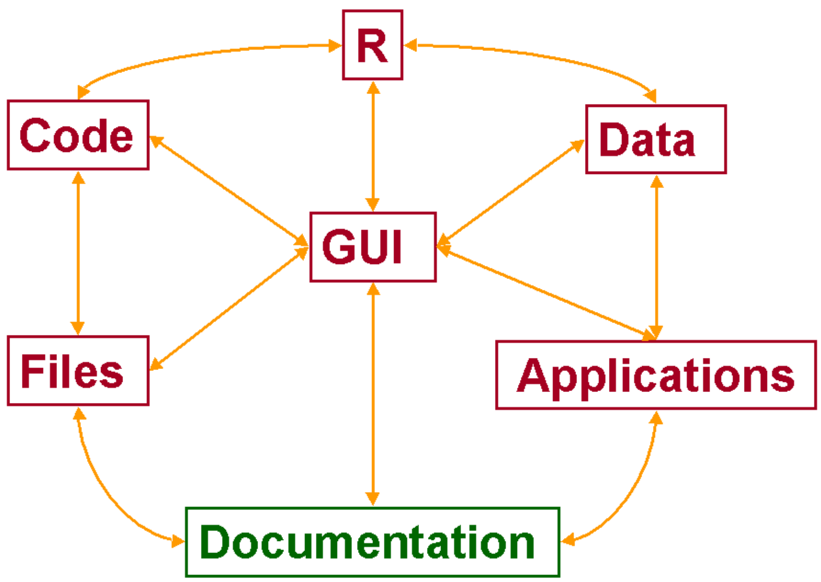


Figure 2. Computer model components organized with a graphical user interface (GUI).

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 http://cran.r-project.org/.

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 (http://mathworld.wolfram.com/LissajousCurve.html), named after one of its inventors Jules-Antoine Lissajous, represents the dynamics of the system

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

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 1, 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:

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

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

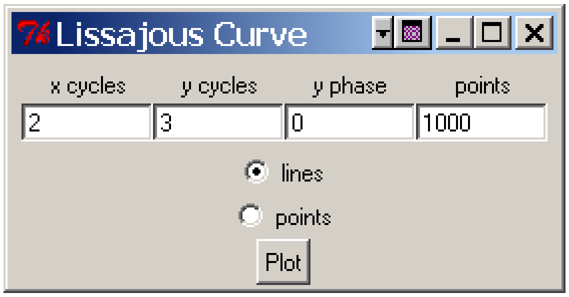


Figure 3. GUI generated by the description file `LissajousCurve.txt` in Table 1. It contains five widgets: the window titled “Lissajous Curve”, a vector of four entries, two linked radio buttons ($\langle$lines$\rangle$ and $\langle$points$\rangle$), and a $\langle$Plot$\rangle$ button.

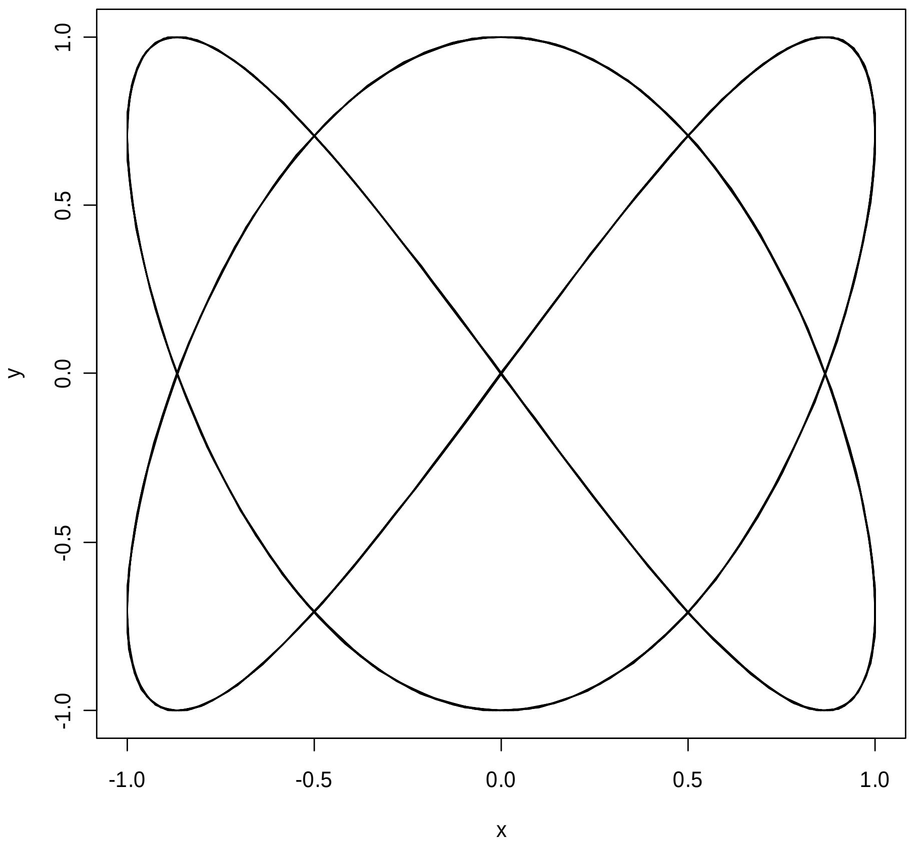


Figure 4. Default graph for the “Lissajous Curve” project, obtained by clicking the $\langle$Plot$\rangle$ button in Figure 3. The *x* variable goes through two cycles while the *y* variable goes through 3 cycles. A line graph is drawn through 1,000 points generated by the algorithm (1).

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`, `phi`, 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 (\) 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”. In, we use the symbols ⟨…⟩ to designate a button or keystroke, such as the ⟨Plot⟩ button or the radio buttons ⟨lines⟩ and ⟨points⟩. These symbols are not to be confused with *talk description file* tags (<>) used later (Section 6).

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 22 widgets:

1. `window` – an entire new window;
2. `menu` – a menu grouping;
3. `menuitem` – an item in a menu;
4. `grid` – a rectangular block for placing widgets;
5. `label` – a text label;
6. `button` – a button linked to an R function that runs a particular analysis and generates a desired output, perhaps including graphics;
7. `check` – a check box used to turn a variable on or off, with corresponding values `TRUE` or `FALSE`;
8. `radio` – one of a set of mutually exclusive radio buttons for making a particular choice;
9. `null` – a blank widget that can occupy an empty space in a grid;
10. `entry` – a field in which a scalar variable (number or string) can be altered;
11. `text` – an entry box that supports multiple lines of text;
12. `vector` – an aligned set of entry fields for all components of a vector;
13. `matrix` – an aligned set of entry fields for all components of a matrix;
14. `data` – an aligned set of entry fields for all components of a data frame, where columns can have different modes;
15. `object` – an aligned set of entry fields defined by an existing R-object (vector, matrix, or data frame);
16. `slide` – a slide bar that sets the value of a variable;

17. `slideplus` – an extended slide bar that also displays a minimum, maximum, and current value;
18. `history` – a device for archiving parameter values corresponding to different model choices, so that a “slide show” of interesting choices can be preserved;
19. `droplist` – an entry widget with a drop down list of values;
20. `table` – a spreadsheet widget with scrollbars for large tabular data;
21. `spinbox` – an entry widget for a numeric value within a given range which can be changed with the up and down arrows;
22. `include` – a pseudo widget which embeds a specified window description file within the current window description file.

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 22 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. illustrates how this can be done in a simple case. 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.

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",
  "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)
```

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 six 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 (up 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.

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.

Two functions provide support for selecting a file from a GUI:

- `promptOpenFile` shows the current directory for choosing a file to open;
- `promptSaveFile` shows the current directory for naming a file to save.

Files can be opened in programs external from R depending on their file extension:

- `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;
- `exportHistory` 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.

2.4. Internal data

`PBSmodelling` uses the hidden list variable `.PBSmod` in the global environment 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 `tktoplevel()`.
- `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`. For example, from the R console, type `runExamples()` and select the example “CalcVor”. Then type the command `str(.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.

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* (http://otter-rsch.ca/admodel.htm), 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

`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) and beer (correlations).
....................................(Code courtesy of Dr. Steve Martell, Fisheries Science Centre, UBC.)
`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.
`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 *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`, `nlminb`, 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 $(\hat{x}_1, \ldots, \hat{x}_n)$ of length *n* that minimizes the function $y = f(x_1, \ldots, x_n)$. 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):

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

$$s = \frac{1}{\pi}\operatorname{acos}\left(\frac{x_{\max} + x_{\min} - 2x}{x_{\max} - x_{\min}}\right) = \frac{2}{\pi}\operatorname{asin}\sqrt{\frac{x - x_{\min}}{x_{\max} - x_{\min}}}\,. \tag{4.3b}$$

All these formulas represent equivalent forms of a one-to-one relationship $x \leftrightarrow s$, where $x_{\min} \le x \le x_{\max}$ and $0 \le s \le 1$. 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` *x* 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

$$\texttt{GT0}(x,\varepsilon) = \begin{cases} x, & x \ge \varepsilon \\ \dfrac{\varepsilon}{2}\left[1 + \left(\dfrac{x}{\varepsilon}\right)^2\right], & 0 < x < \varepsilon \\ \dfrac{\varepsilon}{2}, & x \le 0 \end{cases}\,. \tag{4.3c}$$

The notation `GT0` denotes "greater than zero". This function preserves the value of *x* if $x \ge \varepsilon$, and for smaller values *x* it is always true that $\texttt{GT0}(x,\varepsilon) \ge \frac{\varepsilon}{2}$. 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 only to illustrate the use of C code).
`calcGM`..................Calculate the geometric mean of a vector of numbers.
`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.
`testWidgets`......GUI to test all widgets listed in Appendix A.
`view`.......................View the first/last/random *n* lines of a (potentially large) object.

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 like a C compiler. To preserve information about these settings, `PBSmodelling` allows options (including the associations with file extensions for `openFile` mentioned earlier) to be saved in a local file with the default name `PBSoptions.txt`. To avoid conflict with R's `options()`, we use the hidden list `.PBSmod$.options` (mentioned in Section 2.4), and we provide the support functions:

`writePBSoptions`...........Write PBS options to an external file.

`readPBSoptions`.............Read PBS options from an external file.
`promptWriteOptions`....Prompt the user to save changed options.

Options can also be set within a GUI window. This requires declaring which widgets correspond to options, as well as synchronizing (getting and setting) the current options with values shown in the window. These tasks can be accomplished with:

`declareGUIoptions`.......Declare option names that correspond with widget names.
`getGUIoptions`................Get PBS options for widgets.
`setGUIoptions`................Set PBS options from widget values.

Potentially, the options can exist at three levels: within a Window, within internal memory, or within a file. They become active when they exist in internal memory as part of `.PBSmod`. Our support functions allow them to be altered in GUIs and preserved in files. Different project directories can have files that specify different options. Even within a single directory, files with different names can hold different possible options.

Some options correspond to directory paths or particular files. We provide interactive GUIs that prompt for these choices with a file exploration window:

`setPathOption`................Set a PBS path option interactively.
`setFileOption`................Set a PBS file path option interactively.

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.
`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.
`setwdGUI`............................Browse for working directory and find prefix.
`cleanProj`..........................Launch a GUI for file deletion.

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. Sometimes participants find themselves scrambling to copy code from the visual presentation, files distributed by speaker, or related web sites. During this process, the actual intended content can get lost. Focus shifts from R concepts to typing and other mechanical issues.

`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. The author of a talk can write 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. The files also give 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`.

Table 4 illustrates the format of a talk description file. It uses a mark-up style, in which tagged lines (delineated with <>) indicate starting points for description segments. Currently, `presentTalk` supports the five tags `<talk>`, `<section>`, `<text>`, `<file>`, and `<code>`. A single file can contain one or more talks and each talk can contain one or more sections. Possibly after initial comments (marked as usual with #), the first significant line in the file is tagged `<talk>`, normally followed by the start of a `<section>`. Lines tagged as `<text>` are displayed as ordinary text in the R console. These correspond to lecture notes, comparable to what might otherwise appear on a slide. A `<file>` line indicates that one or more files should be opened at that point. For example, it might be desirable to display a file of R code or open a PowerPoint file that supplements the examples in the R console. Lines tagged as `<code>` are displayed and run in the R console. Appendix B gives complete details of the options available for talk description files.

Table 4. A talk description file `SwissTalk.txt` 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.txt

```
<talk name="Swiss" button=FALSE>

# SECTION 1. The "summary" method
<section name="Methods" button=TRUE>

# State the talk's purpose in text
<text>
This short talk examines the "summary" method
and applies it to the "swiss" dataset.
The talk itself comes from a talk description file ...

# Show the description file
<file name="swissTalk" button=TRUE>
  swissTalk.txt

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

# SECTION 2. The "swiss" data
<section name="Data" button=TRUE>
<text break=F> "swiss" is a data frame (class data.frame):
<code> isWhat(swiss)
<text break=F> You can read about the data here:
<code> help(swiss) # open the help file
<text break=F> Apply "summary" to Swiss:
<code break=print> summary(swiss)
<text break=F> Print the first 3 records:
<code break=print> head(swiss,3)
<text break=F> Display the data with the "plot" method . . .
<code print=F> plot(swiss,gap=0)
<text> THE END .. THANKS FOR WATCHING!
```

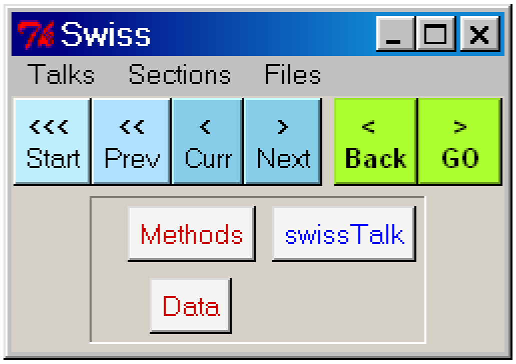


Figure 5. The GUI generated by `presentTalk` from the talk description file in Table 4.

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 by showing the description file itself (`SwissTalk.txt`), 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 tag lines for `presentTalk` give the author considerable scope for introducing breaks and other features into the presentation. Furthermore, each `<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. After each step, the R console remains open for additional code written on the spur of the moment. Furthermore, the menu items (`Talks`, `Sections`, `Files`) allow for quick movement among talks and/or 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 tag segment. The blue buttons allow movement among sections – "Start" to the first section of the talk, "Prev" to the previous section, "Curr" 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 talks 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. Partly for this reason, we emphasize that **`presentTalk` will allow *only one* talk to operate at a time. Each talk has its own GUI, named from the `<talk>` tag line.** If you use the GUI to switch from one talk to another, the first will be terminated, the second started from the beginning, and the global environment left unchanged. In some cases, it may help to start a talk or section with `<code> clearAll()` 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.

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. If this seems rather primitive, please wait for improvements in future versions.

7.1. Random variables

7.1.1. `RanVars` – Random variables

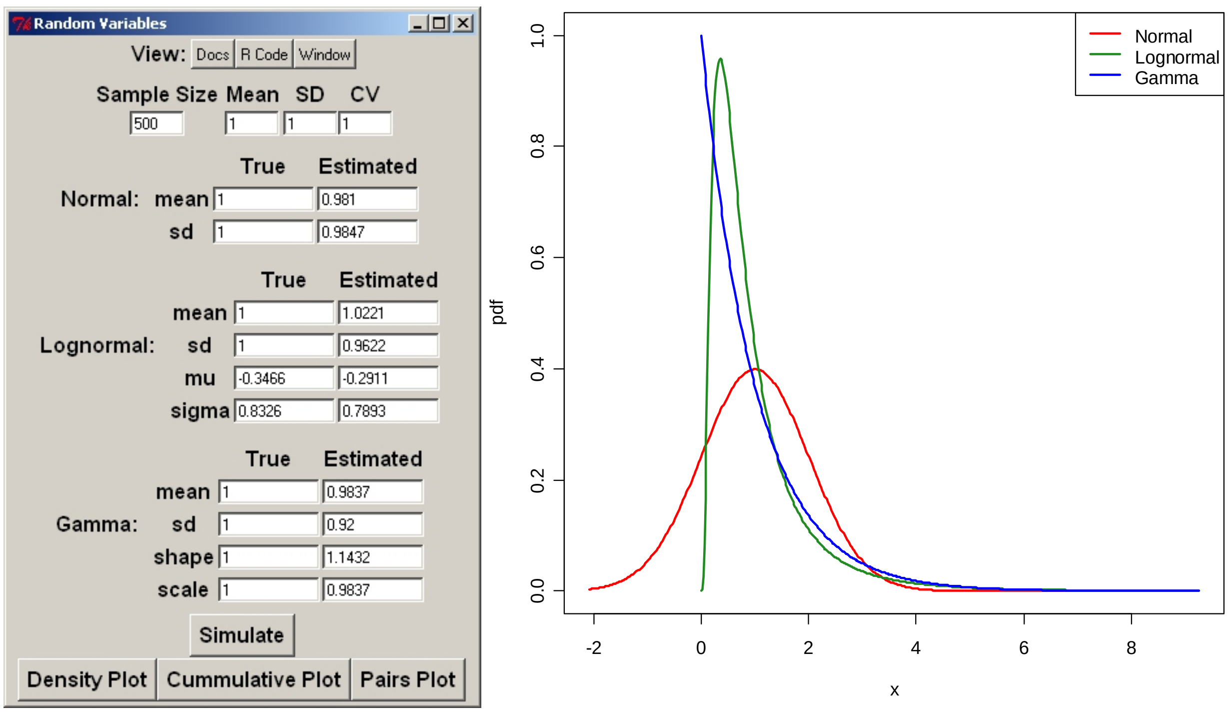


Figure 6. `RanVars` GUI (left) and density plot (right). Simulations are based on 500 random draws with mean =1 and SD = 1.

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

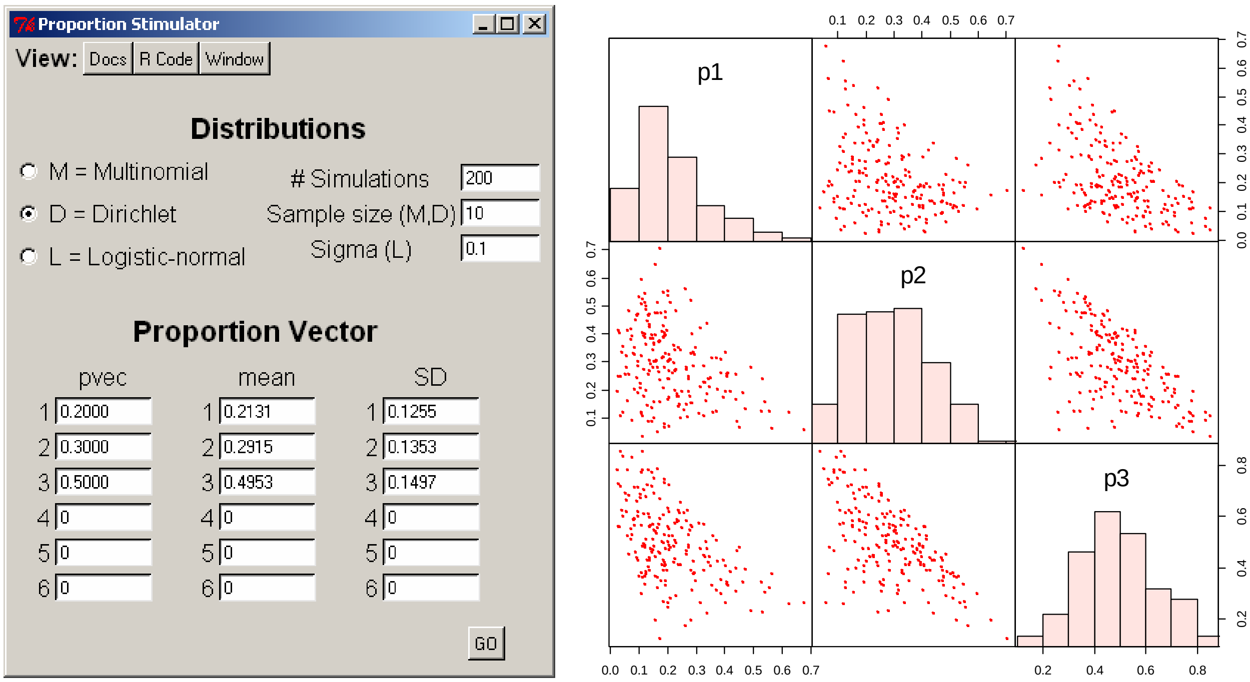


Figure 7. `RanProp` GUI (left) and pairs plot (right). Simulations are based on 200 random draws where n = 10 for the multinomial and Dirichlet distributions and σ = 0.1 for the logistic-normal distribution. The pairs plot portrays results for the Dirichlet.

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

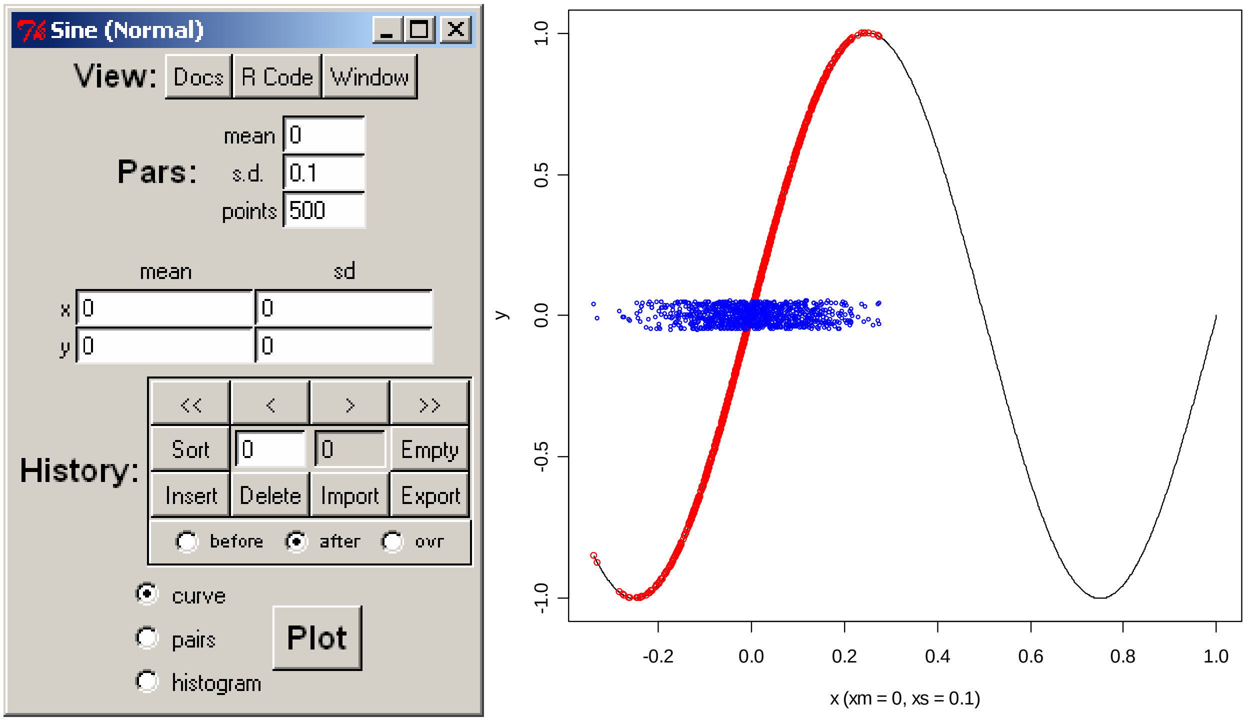


Figure 8. `SineNorm` GUI (left) and plot (right). Simulations are based on 500 random draws of $y = \sin(2\pi x)$, where *x* is normal with mean $\mu = 0$ and standard deviation $\sigma = 0.1$. Blue points portray jittered values of *x*, and red points show corresponding values of *y*.

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

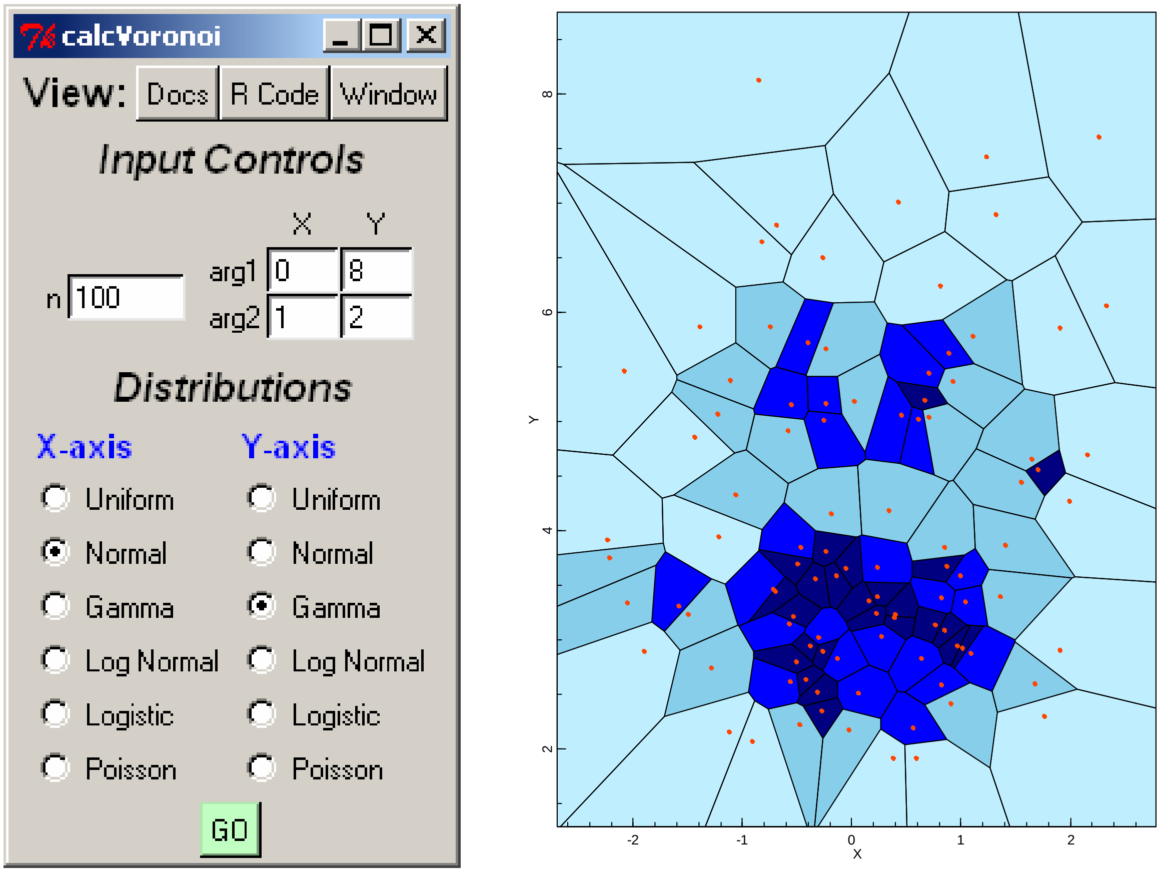


Figure 9. `CalcVor` GUI (left) and plot (right). Tessellation of random points (red) that are normally distributed on the x-axis (`mean=0`, `sd=1`) and gamma-distributed on the y-axis (`shape=8`, `rate=2`).

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

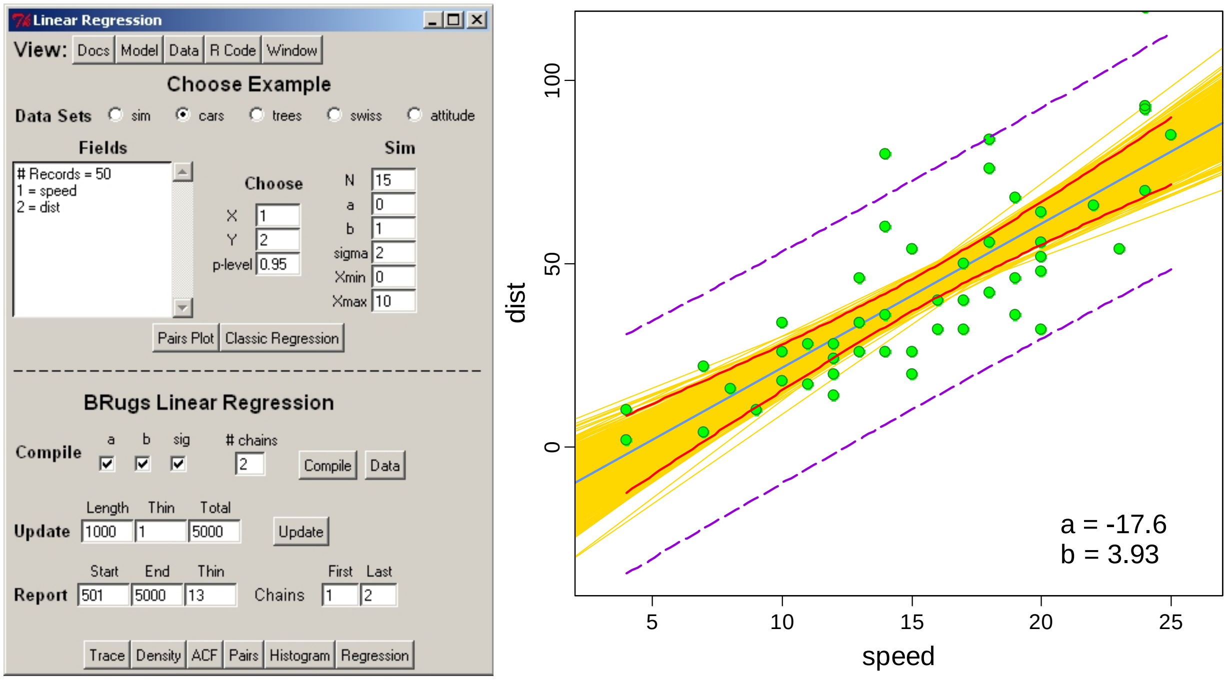


Figure 10. `LinReg` GUI (left) and regression plot (right). The linear regression uses the `cars` dataset (*n*=50) to predict `dist` vs. `speed`. The plot shows observations (green circles), fitted line (solid blue line), the 95% confidence limits of the fitted model (solid red lines), the 95% CL of 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

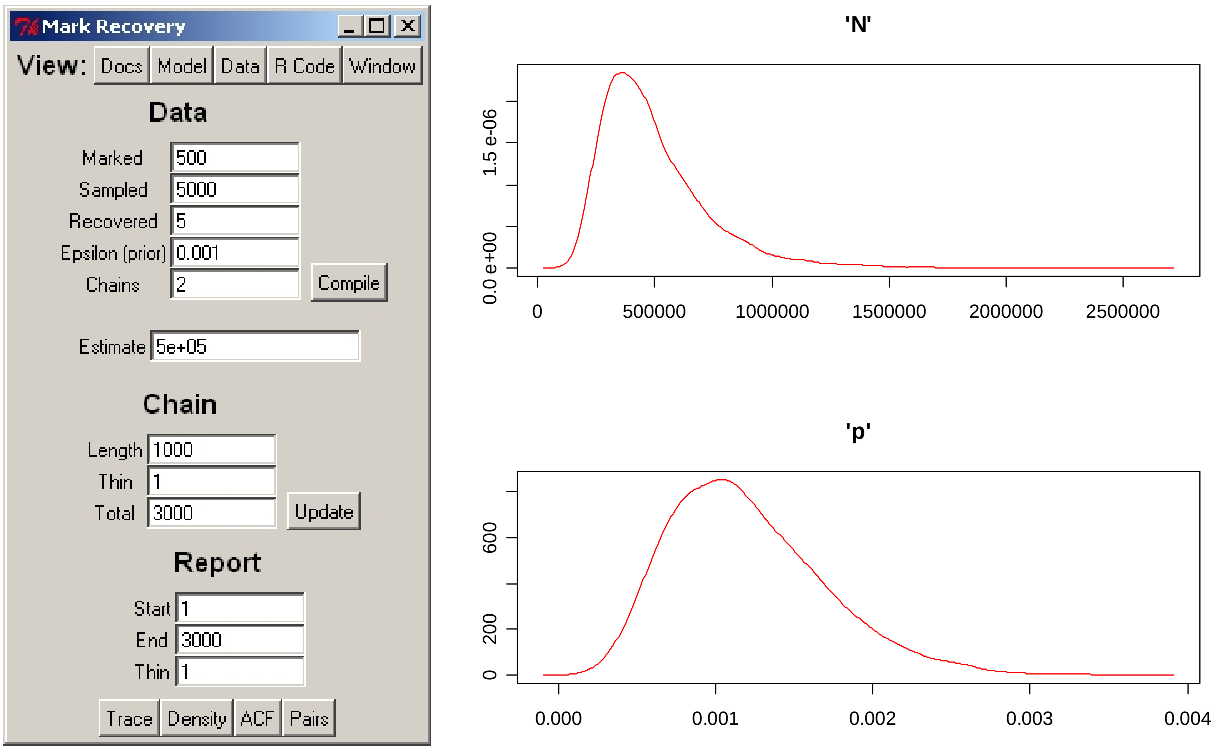


Figure 11. `MarkRec` GUI (left) and density plots (right). A low recovery of marked fish can lead to fat tails in *N* due to occasional large spikes in the population estimate.

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

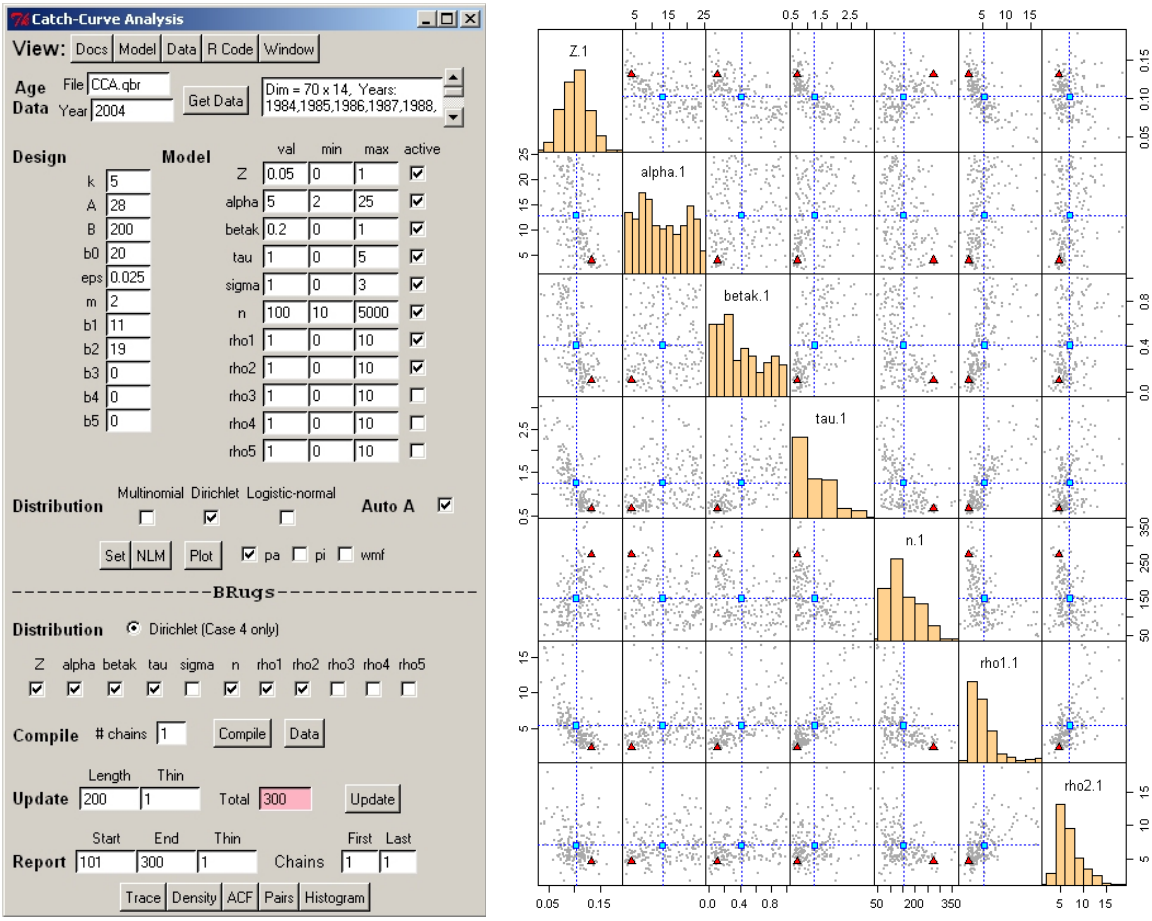


Figure 12. `CCA` GUI (left) and parameter pairs plot (right). Comparison of Bayes posterior distribution of CCA model parameter estimates from chain 1 (*N*=100). Symbols indicate means (blue squares) and modes (red triangles). Diagonal shows parameter estimate distributions.

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

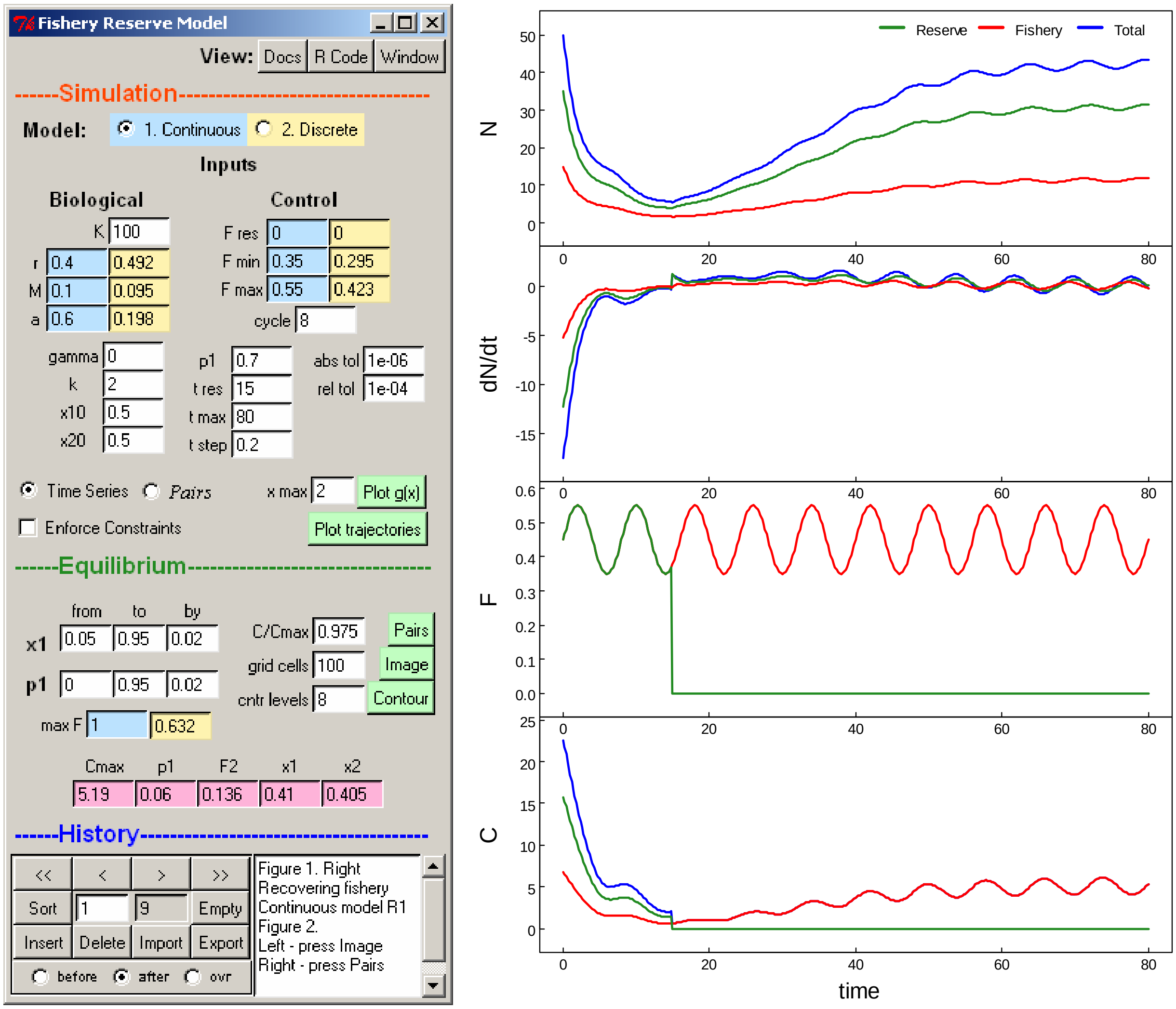


Figure 13. `FishRes` – Recovery of a heavily fished population after establishing a reserve. The GUI (left) shows all input values (parameters and controls). The selected continuous time model uses input values common to both models (white background) and values specific to the continuous model (blue 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 `odesolve` are required.

7.3.2. `FishTows` – Fishery tows

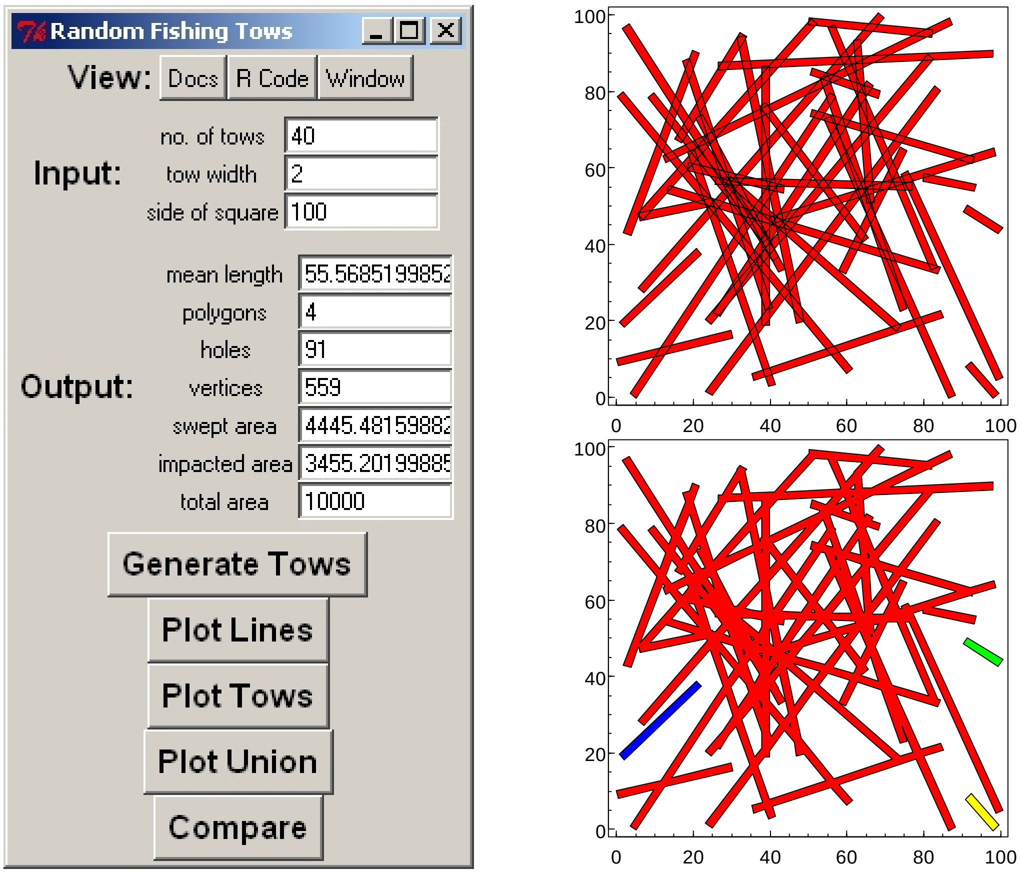


Figure 14. `FishTows` GUI (left) and simulated tow track (right). Tow track plots show 40 random tows in a square with side length 100. Each tow has width 2, and the rectangle encompasses 10,000 square units. *Top*: The individual rectangles, with 160 vertices, have areas that sum to 4,445 square units. *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.

Daalgard, P. 2001. A primer on the R Tcl/Tk package. *R News* 1 (3): 27–31, September 2001. URL: http://CRAN.R-project.org/doc/Rnews/

Daalgard, P. 2002. Changes to the R Tcl/Tk package. *R News* 2 (3): 25–27, December 2002. URL: http://CRAN.R-project.org/doc/Rnews/

Griewank A. (2000) Evaluating derivatives: principles and techniques of algorithmic differentiation. Frontiers in Applied Mathematics 19. Society for Industrial and Applied Mathematics

Ligges, U. 2003. R Help Desk: Package Management. *R News* 3 (3), 37–39. URL: http://CRAN.R-project.org/doc/Rnews/

Ligges, U, and Murdoch, D. 2005. R Help Desk: Make `'R CMD'` work under Windows – an example. *R News* 5 (2), 27–28. URL: http://CRAN.R-project.org/doc/Rnews/

Mittertreiner, A., and Schnute, J. 1985. Simplex: a manual and software package for easy nonlinear parameter estimation and interpretation in fishery research. Canadian Technical Report of Fisheries Aquatic Sciences 1384: xi+90 p.

Ousterhout, J.K. 1994. Tcl and the Tk toolkit. Addison-Wesley, Boston, MA. 458 p.

RDCT: R Development Core Team (2006a). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0. URL http://www.R-project.org. (Available in the current R GUI from “Help”, “Manuals in PDF”, “R Reference Manual”)

RDCT: R Development Core Team (2006b). Writing R extensions. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-11-9. URL http://www.R-project.org. (Available in the current R GUI from “Help”, “Manuals in PDF”, “Writing R extensions”)

Richards, L.J., Schnute, J.T., and Olsen, N. 1997. Visualizing catch-age analysis: a case study. Canadian Journal of Fisheries and Aquatic Sciences 54: 1646–1658.

Schnute, J. 1982. A manual for easy nonlinear parameter estimation in fishery research with interactive microcomputer programs. . Canadian Technical Report of Fisheries and Aquatic Sciences 1140. xvi+115 pp.

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. Canadian Technical Report of Fisheries and Aquatic Sciences 2496. viii+82 pp.

Schnute, J.T., Boers, N.M., and Haigh, R. 2004. PBS Mapping 2: user's guide. Canadian Technical Report of Fisheries and Aquatic Sciences 2549. viii+126 pp.

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. Available at http://icesjms.oxfordjournals.org/content/vol64/issue2/index.dtl, reference number doi:10.1093/icesjms/fsl024.

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. (File `FishResDoc.pdf` available in the package `PBSmodelling`.)

Schnute, J.T., and Richards, L.J. 1995. The influence of error on population estimates from catch-age models. Canadian Journal of Fisheries and Aquatic Sciences, 52: 2063–2077.

Spiegelhalter, D., Thomas, A., Best, N., and Lunn, D. 2004. WinBUGS User Manual, version 2.0. Available at http://mathstat.helsinki.fi/openbugs/.

Thomas, N. 2004. BRugs User Manual (the R interface to BUGS), version 1.0. Available at http://mathstat.helsinki.fi/openbugs/.

Appendix A. Widget descriptions

This appendix lists `PBSmodelling` widgets in alphabetical order, except for “Window” which needs to exist before the other widgets can be placed. Details for each widget include a description, usage, 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.

Window

Description

Create a new window. Windows are used as a palette 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=""
```

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 

Example

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

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="" width=0
     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
`width` .... button width, the default 0 will adjust the width to the minimum required
`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 available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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="" 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
`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 available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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 or 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" entryfg="black"
     entrybg="grey" 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=F`

`noeditbg` .... background colour of input/output boxes when `edit=F`

`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

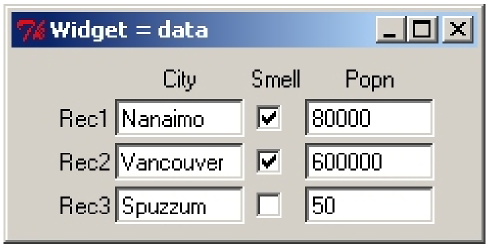
`sticky` .... option for placing the widget in available space; valid choices are: `N, NE, E, SE, S, SW, W, NW`

`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 available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
- `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

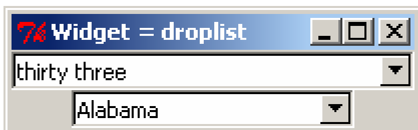
Note

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
```

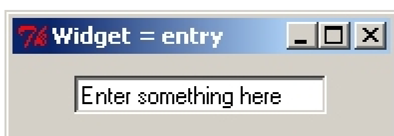
Arguments

`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 (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

`entryfg`............font colour of input/output boxes when `edit=F`
`entrybg`............background colour of input/output boxes when `edit=F`
`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 available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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
```



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
```

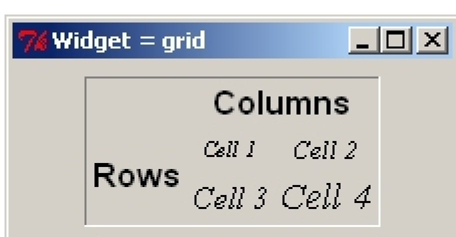
Arguments

`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 (as point size), and style (`bold`, `italic`, `underline`, `overstrike`), in any order
`sidefont` ....font for side labels – specify family (`Times`, `Helvetica`, or `Courier`), size (as point size), and 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 color of the top title
`sidebg` ....background color of the side title
`bg`.........................background colour of grid including top and side titles when `topbg` and `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 available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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"
```



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.
n	Display window (white background) shows the current record.
N	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 available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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
```

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" wraplength=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 available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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=NULL collabels=NULL
     rownames="" colnames="" font="" fg="black" bg="" entryfont=""
     entryfg="black" entrybg="white" entryfg="black" entrybg="grey"
     values="" byrow=TRUE function="" enter=TRUE action="matrix"
     edit=TRUE mode="numeric" width=6 borderwidth=0 sticky=""
     padx=0 pady=0
```

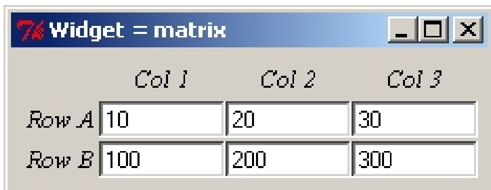
Arguments

`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=F`
`noeditbg` ....background colour of input/output boxes when `edit=F`
`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 available space; valid choices are:
`N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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"
```



Menu

Description

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

Usage

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

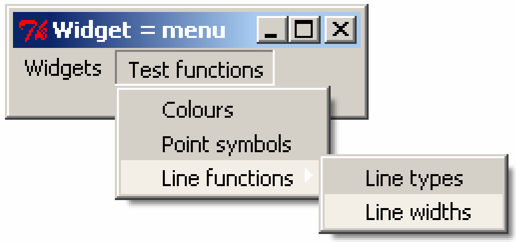
Arguments

`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

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="grey" 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

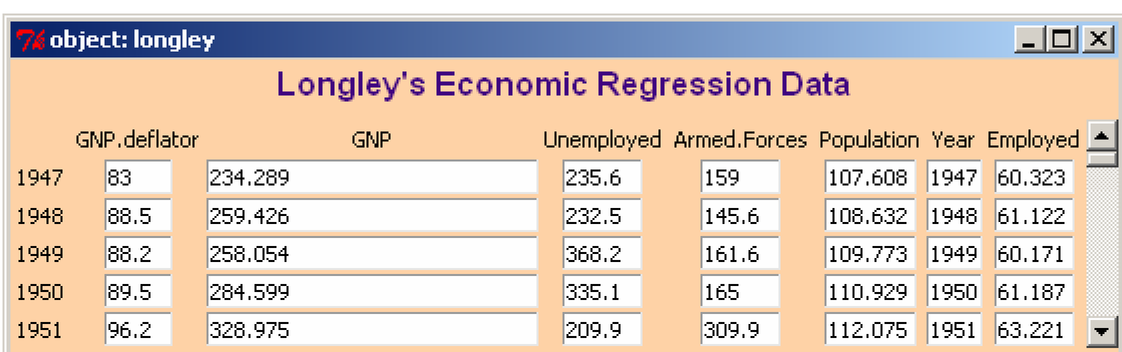
`noeditfg` ....font colour of entries appearing in input/output boxes when `edit=F`
`noeditbg` ....background colour of input/output boxes when `edit=F`
`vertical` ....only applicable when the R-object is a vector; if `TRUE` , display the vector as a vertical column with labels on the left; otherwise 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 available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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
```



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

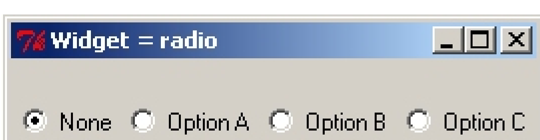
```
type=radio name value text="" font="" fg="black" bg="" function=""
     action="radio" edit=TRUE mode="numeric" sticky="" padx=0
     pady=0
```

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`
`sticky`...............option for placing the widget in available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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 available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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 – specify family (`Times`, `Helvetica`, or `Courier`), size (as point size), and 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 – specify family (`Times`, `Helvetica`, or `Courier`), size (as point size), and 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` and the slide value is changed via the entry box, call the function only after the ⟨Enter⟩ key is pressed
`action`...............string value associated whenever this widget is engaged
`sticky`...............option for placing the widget in available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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

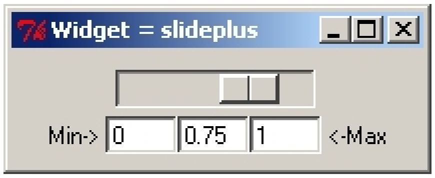
Note

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="droplist" 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 (as point size), and 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 (as point size), and 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` and the slide value is changed via the entry box, call the function only after the ⟨Enter⟩ key is pressed
`action`...............string value associated whenever this widget is engaged
`width`.................character width to reserve for the entry
`sticky`...............option for placing the widget in available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`

`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 and down with the up and 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

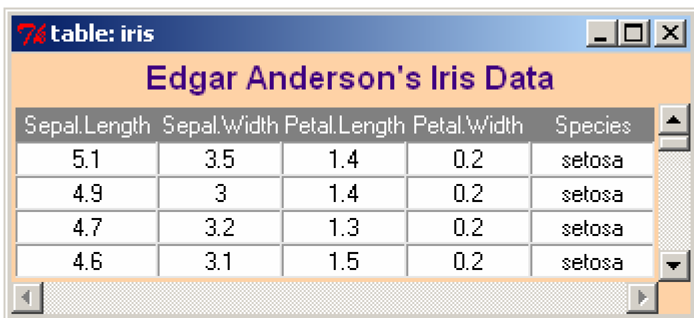
`collabels` ....a vector of `ncol` labels used to label columns; if `collabels=""`, then the object's column names are used; if `NULL`, no labels are displayed

`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 available space; valid choices are: `N, NE, E, SE, S, SW, W, NW`
`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
```

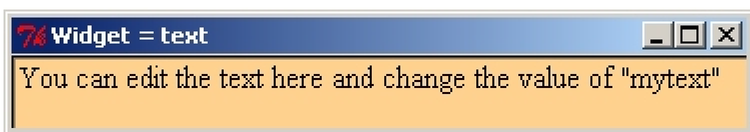
Arguments

`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 available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
`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"
     entryfg="black" entrybg="grey" vertical=FALSE function=""
     enter=TRUE action="vector" mode="numeric" width=6 sticky=""
     padx=0 pady=0
```

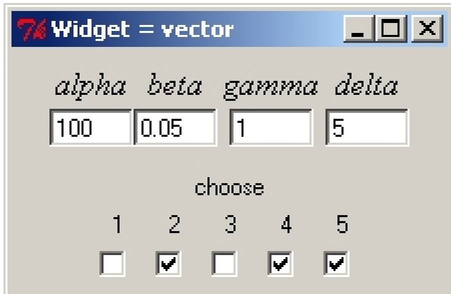
Arguments

- `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=F`
- `noeditbg`..........background colour of input/output boxes when `edit=F`
- `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
- `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
- `sticky`...............option for placing the widget in available space; valid choices are: `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`
- `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 tag lines (marked `<>`) with intervening text. We define a file *segment* as a tag line along with all the text down to (but not including) the next tag line. The last segment ends at the end of the file. Similarly, we define a *block* in the description file as a group of contiguous segments. A file contains segments of `<text>`, R `<code>`, and `<file>` names. These are combined to give `<section>` blocks, which in turn make up `<talk>` blocks. A valid file must have at least one `<talk>` line, and each `<talk>` line must be followed by at least one `<section>` line.

When `presentTalk()` calls a description file, it produces a control GUI like the one shown in Figure 5. Any declared `<talk>`s, `<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, `<talk>` buttons appear in the first column, `<section>` buttons in the second column, and `<file>` buttons in the third 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. The `name`s of menu items and buttons must always consist of alphanumeric characters and underscores. Furthermore, a `name` must begin with a letter.

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.

We end this appendix with a precise description of the purpose and syntax for each tag line. Instead of alphabetical order, we use the more logical order: `<talk>`, `<section>`, `<text>`, `<code>`, and `<file>`.

<talk>

Description

Starts a description block that constitutes a talk. The block ends at the next `<talk>` line or the end of the file.

Usage

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

Arguments

`name`....................A string giving the name of the talk (required). It appears as the title of the control GUI, a menu item (under "Talks"), and possibly also as a button.

`button` .... A Boolean variable (`TRUE` or `FALSE`) that determines whether or not the GUI should add a button that selects the talk, 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 file must have at least one `<talk>` line, and each `<talk>` line must be followed by at least one `<section>` line. Each `<talk>` block in a file must have a unique `name`. **Different talks have distinct associated control GUIs, and `presentTalk` allows *only one* presentation at a time.**

`<section>`

Description

Starts a description block that constitutes a section of a talk. The block ends at the next `<section>` line, `<talk>` line, or the end of the file.

Usage

```
<section name=(required) button=FALSE col=2>
```

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.
`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

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

`<text>`

Description

Starts a description segment that represents text to be printed 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>`

Description

Starts a description segment that names files to be opened by the operating system with `openFile()`.

Usage

```
<file name=(required) button=FALSE col=3 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.
`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 in the description segment 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>` segment 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>`

Description

Starts a description segment that represents 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 in this segment normally consists of valid R code, although a speaker may choose to demonstrate the consequences of invalid code.

Line breaks in the description file 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>` segment 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.

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 2006b, `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 `PBStry` 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-2.8.0\library\PBSmodelling\PBStools`.

C.1. Installing required software

Building R packages requires four pieces of free software. Duncan Murdoch currently maintains their availability and installation instructions at:
http://www.murdoch-sutherland.com/Rtools/
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 2.7.2 (`C:\Utils\R\R-2.8.0`). We assume that R is already installed from the CRAN web site http://cran.r-project.org/ 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, MinGW compilers, ActivePerl text scripting, etc. (`C:\Utils\Rtools\`). Download and run the file `Rtools28.exe`. The installation should create the subdirectories `\bin` for command line programs, `\MinGW` for the minimalist GNU C compiler for Windows, and `\perl` for the ActivePerl scripting language. 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. The Microsoft **HTML Help Workshop** (`C:\Utils\HHW\`). Run the installation file `HtmlHelp.exe`. After installation, we think you can safely ignore a message that “This computer already has a newer version of HTML Help”. (If anyone has different information, please let us know.)

4. **MiKTeX**: a LaTeX and pdftex package (`C:\Utils\MiKTeX`). The link takes the user to http://www.miktex.org/. This processor for TeX and LaTeX files helps typeset 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 2.5”, “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-2.8.0\share\texmf`). These should include `Rd.sty`.

   c) Go to the “Start” menu, select “Programs” then “MiKTeX 2.5”, 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 **Tinn-R** available at http://sourceforge.net/projects/tinn-r. **Tinn-R** is described as a “simple but efficient replacement for the basic code editor provided by Rgui”. Alternatively, the text editor **WinEdt** (available from http://www.winedt.com/) 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.x`; 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-2.8.0\library\`), find the subdirectory `PBSmodelling\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-2.8.0\bin
set TOOLS_PATH=C:\Utils\Rtools\bin
set PERL_PATH=C:\Utils\Rtools\perl\bin
set MINGW_PATH=C:\Utils\Rtools\MinGW\bin
set TEX_PATH=C:\Utils\MiKTeX\miktex\bin
set HTMLHELP_PATH=C:\Utils\HHW
```

Notice that each path, except the last, 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 http://cran.r-project.org/, go to "Packages" on the left and find `PBSmodelling`. Download the file `PBSmodelling_x.xx.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 sudirectories: `\data`, `\inst`, `\man`, `\R`, and `\src`.

3. Our batch file uses the command `tar -xzvf 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.xx.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.3 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., UltraEdit, WinEdt, or Notepad) to view and change various text files.

1. Start by locating the file `PBStry_x.xx.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 `.Rd2dvi$`, 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 PBStry
   ```
   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 `\PBStry`, along with its five sudirectories: `\data`, `\inst`, `\man`, `\R`, and `\src`.

4. Use your editor to open the file `DESCRIPTION` in the root directory `\PBStry`. This file, essential in every R package, contains key information in a special format (RDCT 2006b, 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 `PBStry` to `MyPack` in `DESCRIPTION` and rename the directory from `\PBStry` 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 `\PBStry\R` contains all R code used by the package. For example, `PBStry` includes seven R functions (`calcFib`, `calcFib2`, `calcGM`, `calcSum`, `findPat`, `pause`, and `view`). The seven files could be combined into a single file (such as `PBStry.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 (`PBStry.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. `PBStry` illustrates this possibility for `zzz.R`.

10. The subdirectory `\PBStry\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.hl` 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 `\PBStry\man` contains a documentation file for every object in the package. `PBStry` has six functions and one data set, so the `\man` subdirectory has seven

corresponding R documentation files (`*.Rd`). An additional file `PBStry.Rd` documents the package as a whole. Rd files use a rather complex scripting language (RDCT 2006b, Section 2) that can be converted to help files in several formats (PDF, HTML, text). For many packages, the examples in `PBStry` may provide adequate prototypes. They represent three distinct cases: functions (e.g., `calcGM.Rd`, `findPat.Rd`), data sets (`QBR.Rd`), and complete packages (`PBStry.Rd`).

13. The subdirectory `\PBStry\src` contains source code for C code to be compiled into the dynamic link library `PBStry.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 `\PBStry\inst` contains files that are to be included directly in the R library tree for `PBStry` when the package is installed. The file `PBStry-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 `PBStry`. Furthermore, you're reasonably familiar with the contents of `PBStry`. 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 don't export "dot" objects, whose names in R start with a period. (For more complete information on these functions, see Appendix D.2.) The `NAMESPACE` file must also import functions required from other packages. Because `PBSmodelling` relies on `tcltk`, the file includes the command: `import(tcltk)`.

In `PBStry`, 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`.

In summary, we recommend building a new package by editing, adding, and deleting prototype files in `PBStry`. 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 (2006b) 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. `PBStry` 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 illustrate 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`.

<u>Calling C functions from R using `.C()`</u>

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`)

Table C2. Two text files associated with a `.C()` call in `PBStry`. 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="PBStry")
  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 */
```

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 C3. `.Call()` example adapted from `PBStry`, 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 */
```

<u>Calling C functions from R using `.Call()`</u>

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 2006b, 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

This appendix documents the objects currently available in PBSmodelling, along with a list of function dependencies for exported functions and hidden "dot" functions. The latter are hidden through R's NAMESPACE but can be seen through the triple colon convention (e.g., PBSmodelling:::.addslashes). R also provides a function called fixInNamespace() for modifying NAMESPACE objects. The final section 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........................Add current window settings to the current history record
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.....................Move back one step in the saved values for a history widget
calcFib...............................Calculate Fibonacci numbers by several methods
calcGM................................Calculate the geometric mean, allowing for zeroes
calcMin...............................Calculate the minimum of user-defined function
CCA.qbr..............................Data: sampled counts of quillback rockfish (*Sebastes maliger*)
chooseWinVal..................Choose and set a string item in a GUI
cleanProj..........................Launch a GUI for file deletion
clearAll............................Remove all R objects from the global environment
clearHistory...................Clear saved values for a history widget
clearPBSext.....................Clear file extension associations
clearWinVal.....................Remove all current widget variables
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
createVector..................Create a GUI with a vector widget
createWin..........................Create a GUI window
declareGUIoptions.......Declare option names that correspond with widget names
drawBars............................Draw a linear barplot on the current plot
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
firstHistory...................Jump to the first history record
focusWin............................Set the focus on a particular window
forwHistory.....................Move forward one step in the saved values for a history widget
genMatrix..........................Generate test matrices for plotBubbles
getChoice..........................Choose one string item from a list of choices
getGUIoptions................Get PBS options for widgets

`getPBSext`	Get a command associated with a filename
`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 by its class, and attributes
`jumpHistory`	Jump to a particular history record
`lastHistory`	Jump to the last history record
`loadC`	Launch a GUI for compiling and loading C code
`openExamples`	Open example files from a package
`openFile`	Open a file with the associated program
`openPackageFile`	Open a file from a package subdirectory
`openProjFiles`	Open files with a common prefix
`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 pairs plots as fried eggs and beer
`plotTrace`	Plot trace lines from a data frame, matrix, or vector
`presentTalk`	Run a talk in R from a talk description file
`promptOpenFile`	Display an “Open File” dialogue
`promptWriteOptions`	Prompt the user to save changed options
`promptSaveFile`	Display a “Save File” dialogue
`readList`	Read a `list` from a file in `PBSmodelling` format
`readPBSoptions`	Read PBS options from an external file
`resetGraph`	Reset `par` values for a plot
`restorePar`	Get actual parameters from scaled values
`rmHistory`	Remove a record from the history
`runDemos`	Run a GUI to access demos from any R package installed
`runExamples`	Run GUI examples included with `PBSmodelling`
`scalePar`	Scale parameters to [0,1]
`setFileOption`	Set a PBS file path option interactively
`setGUIoptions`	Set PBS options from widget values
`setPathOption`	Set a PBS path option interactively

Dot functions (and two list objects: `.pFormatDefs` and `.widgetDefs`)

`.addslashes`	Escape special characters from a string
`.autoConvertMode`	Convert `x` into a numeric mode
`.buildgrid`	Attach child widgets to a grid
`.catError`	Display parsing errors
`.catError2`	Display parsing error (from C code)
`.CGUIchooseSection`	Choose a section from a talk control GUI
`.CGUIgo`	Continue the execution of a talk
`.cleanLoadC`	Launch a GUI for cleaning C junk files
`.convertMatrixListToDataFrame`	Convert a list into a data frame
`.convertMatrixListToMatrix`	Convert a list to a matrix (or a higher dimensional array)
`.convertMode`	Convert a variable into a mode without showing any warnings
`.convertPararmStrToList`	Convert a string representing a widget into a vector
`.convertPararmStrToVector`	Convert a string representing data into a vector
`.convertVecToArray`	Convert a vector to an array
`.createTkFont`	Creates a usable *Tk* font from a given string
`.createWidget`	Call the appropriate sub-function (below) to create a given widget
`.createWidget.button`	
`.createWidget.check`	
`.createWidget.data`	
`.createWidget.entry`	
`.createWidget.grid`	
`.createWidget.history`	
`.createWidget.label`	
`.createWidget.matrix`	
`.createWidget.null`	
`.createWidget.object`	
`.createWidget.radio`	
`.createWidget.slide`	
`.createWidget.slideplus`	
`.createWidget.text`	
`.createWidget.vector`	
`.dClose`	Function to execute on closing `runDemos()`
`.doClean`	Do cleaning for `cleanProj`
`.extractData`	Receive events from *Tk*, and extract data for `getWinAct`
`.extractFuns`	Extract a list of called functions
`.extractVar`	Extract values from the `tclvar ptrs` of a window
`.fibC`	Call Fibonacci C code via `C`
`.fibCall`	Call Fibonacci C code via `Call`
`.fibClosedForm`	Close form equation for Fibonacci numbers

`.fibR`	Calculate Fibonacci numbers in R using iteration
`.getArrayPts`	Return all possible indices of an array
`.getHome`	Get home drive (Windows) or user home (Unix)
`.getMatrixListSize`	Determine the minimum required size of the required array
`.getParamFromStr`	Convert a string representing a widget into a list including default values as defined in `widgetDefs.r`
`.getPrefix`	Get value of widget named “prefix”
`.guiCompileC`	Get parameters from GUI and call `compileC`
`.guiDyn`	Load or unload `lib` based on information from GUI
`.guiSource`	Source an R file as indicated in window description file
`.inCollection`	Find a needle in a haystack (may be removed in future)
`.initPBSoptions`	Initialization function when `PBSmodelling` is loaded
`.isReallyNull`	Test if a key exists in a list
`.libName`	Append `.dll` for Windows or `.so` for Unix
`.loadCRunComparison`	Run a comparison between R and C functions from `loadC` GUI
`.makeCleanVec`	Make descriptions of vectors for `cleanProj`
`.makeTCGUI`	Create a talk control GUI
`.map.add`	Save a new value for a given key, if no current value is set
`.map.get`	Returns a value associated with a key
`.map.getAll`	Return all values of the map
`.map.init`	Initialize the data structure that holds the map(s); a map is another name for hash table (implemented using an R list)
`.map.set`	Save a value, even if a current one exists
`.mapArrayToVec`	Determine the index to use for a vector, given the indices for an element of a higher dimensional array
`.matrixHelp`	Store an element in matrix list (or a higher dimensional array list)
`.mergeLists`	Merge two lists
`.mergeVectors`	Merge two vectors, ensuring values are unique
`.openFileFromGUI`	Open a file from a GUI
`.optionsNotUpdated`	Determine if there are uncommitted options in widget values
`.parsegrid`	Create a branch in the parse tree for children widgets of a grid
`.parsemenu`	Create a branch in the parse tree for children widgets of a menu
`.parseTalk`	Parse a talk description file
`.PBSdimnameHelper`	Add `dimnames` to an object
`.pFormatDefs`	A list defining accepted parameters (and default values) for `"P"` format of `readList` and `writeList`
`.readList.P`	Read a list in `P` format
`.readList.P.convertData`	Convert data into a proper mode
`.removeFromList`	Remove list components
`.runChunk`	Handle code, text, or file in a talk
`.runSection`	Run a section of a talk
`.runTalk`	Run a talk and launch a control GUI
`.searchCollection`	Search a haystack for a needle, or a similar longer needle

`.selectCleanBoxes`	Select checkboxes for `cleanProj`
`.setMatrixElement`	Assign values from a list into a matrix (or *n* dimensional array)
`.setWinValHelper`	Update widget values when `setWinVal` is called
`.setOption`	Set option for `setFileOption` or `setPathOption`
`.showLog`	Shows text in log window and/or creates log file
`.sortActHistory`	Use window action as history name
`.sortHelper`	Helper function to sort `history`
`.sortHelperActive`	Helper function to sort `history`
`.sortHelperFile`	Help `history` with input from and output to an archive file
`.stopWidget`	Display fatal post-parsing errors and halt
`.stripComments`	Remove comments from a string
`.stripExt`	Remove file extension from end of filename
`.stripSlashes`	Removes escape backslashes from a string
`.stripSlashesVec`	Convert a grouping of strings representing an argument into a vector of strings
`.trimWhiteSpace`	Remove leading and trailing white space
`.tryOpen`	Open file with "editor" option or alternatively, `openFile`
`.updateHistory`	Update widget values
`.updateFile`	Coordinate file transfers
`.validateWindowDescList`	Check for a valid `PBSmodelling` description list and set any missing default values
`.validateWindowDescWidgets`	Validate a single widget
`.viewPkgDemo`	Display a GUI to display something equivalent to R's `demo()`
`.widgetDefs`	A list defining widget parameters and default values
`.writeList.P`	Saves a list to disk using the `"P"` format

D.2. Function dependencies

This appendix documents function dependencies within `PBSmodelling`. All functions appear as underlined entries in alphabetic order. If a function depends on others, the list of dependencies appears below the underlined name. Following a standard in UNIX and R, functions whose name begins with a period (*dot functions*) are considered hidden from the user. `PBSmodelling` enforces this standard through `NAMESPACE` discussed in Section C.3.

```
.addslashes

.autoConvertMode

.buildgrid
   .createTkFont
   .createWidget

.catError

.CGUIchooseSection
   .runSection

.CGUIgo
   .presentTalk
   .runChunk
   .runSection

.cleanLoadC
   .getPrefix
   .libName
   cleanProj

.convertMatrixList
   ToDataFrame
   .getMatrixListSize
   .setMatrixElement

.convertMatrixList
   ToMatrix
   .getMatrixListSize
   .setMatrixElement

.convertMode

.convertPararmStr
   ToList
   .catError
   .trimWhiteSpace

.convertPararmStr
   ToVector
   .catError
   .trimWhiteSpace

.convertVecToArray
   .getArrayPts
   .mapArrayToVec

.createTkFont
   .convertPararmStr
      ToVector

.createWidget
   .isReallyNull

.createWidget.button
   .createTkFont
   .extractData

.createWidget.check
   .createTkFont
   .extractData
   .map.add

.createWidget.data
   .createWidget.grid
   .stopWidget

.createWidget.entry
   .createTkFont
   .createWidget.grid
   .extractData
    .map.add

.createWidget.grid
   .buildgrid
   .createTkFont

.createWidget.history
   .createWidget.grid
   initHistory

.createWidget.label
   .createTkFont

.createWidget.matrix
   .createWidget.grid
   .stopWidget

.createWidget.null

.createWidget.object
   .createWidget

.createWidget.radio
   .createTkFont
   .extractData
   .map.add

.createWidget.slide
   .createTkFont
   .extractData
   .map.add

.createWidget.slideplus
   .extractData
   .map.add
   .map.set

.createWidget.text
   .createTkFont
   .map.add

.createWidget.vector
   .createWidget.grid
   .stopWidget

.dClose
   getWinAct
   closeWin

.doClean
   .removeFromList
   getWinVal
   showAlert

.extractData
   setWinAct

.extractFuns

.extractVar
   .convertMatrixList
      ToDataFrame
   .convertMatrixList
      ToMatrix
   .convertMode
   .isReallyNull
   .map.getAll
   .matrixHelp
   .PBSdimnameHelper

.fibC

.fibCall

.fibClosedForm

.fibR

.getArrayPts

.getHome

.getMatrixListSize
   .getMatrixListSize

.getParamFromStr
   .catError
   .convertPararmStr
      ToList
   .isReallyNull
   .searchCollection
   .stripSlashes
   .stripSlashesVec
   .trimWhiteSpace

.getPrefix
   getWinVal
   showAlert
```

- .inCollection
- .initPBSoptions
- .isReallyNull
- .makeCleanVec
- .makeTCGUI
  - .CGUIchooseSection
  - .CGUIgo
  - .openFileFromGUI
  - .presentTalk
  - .runSection
- .map.add
  - .isReallyNull
  - .map.init
- .map.get
- .map.getAll
- .map.init
- .map.set
  - .isReallyNull
  - .map.init
- .mapArrayToVec
- .matrixHelp
  - .matrixHelp
- .mergeLists
- .mergeVectors
- .optionsNotUpdated
  - .initPBSoptions
  - getWinVal
- .parsegrid
  - .parsegrid
- .parsemenu
  - .parsemenu
- .parseTalk
- .PBSdimnameHelper
- .readList.P
  - .catError
  - .readList.P.convert Data
  - .stripComments
  - .trimWhiteSpace
- .readList.P.convertData
  - .autoConvertMode
  - .catError
  - .convertMode
  - .convertPararmStr ToVector
  - .convertVecToArray
  - .getParamFromStr
- .removeFromList
- .runChunk
  - .presentTalk
  - .runChunk
  - .runTalk
- .runSection
  - .runChunk
- .runTalk
  - .makeTCGUI
  - .runSection
- .searchCollection
- .selectCleanBoxes
  - .removeFromList
  - getWinAct
  - setWinVal
- .setMatrixElement
  - .setMatrixElement
- .setOption
  - .getHome
  - .initPBSoptions
  - setPBSoptions
  - setWinVal
- .setWinValHelper
  - .map.get
  - .setWinValHelper
- .showLog
  - createWin
  - setWinVal
- .sortActHistory
  - getWinAct
  - sortHistory
- .sortHelper
  - getWinAct
  - getWinVal
  - .sortHelperActive
  - .sortHelperFile
  - sortHistory
- .sortHelperActive
  - .updateHistory
- .sortHelperFile
  - readList
  - writeList
- .stopWidget
- .stripComments
  - .stripComments
- .stripExt
- .stripSlashes
  - .catError
- .stripSlashesVec
  - .catError
- .trimWhiteSpace
- .tryOpen
  - .initPBSoptions
  - openFile
  - showAlert
- .updateFile
  - getWinAct
  - getWinVal
  - promptOpenFile
  - promptSaveFile
  - setWinVal
- .updateHistory
  - setWinVal
- .validateWindowDescList
  - .validateWindow DescWidgets
- .validateWindow DescWidgets
- .viewPkgDemo
  - getWinAct
  - getWinVal
  - openFile
  - runDemos
- .viewPkgVignette
  - getWinAct()
  - getWinVal
  - openFile
  - showVignettes
- .writeList.P
  - .addslashes

```
addArrows

addLabel

addLegend

addHistory
  .updateHistory
  getWinAct
  getWinVal

backHistory
  .updateHistory
  getWinAct
  setWinVal

calcFib
  .fibC
  .fibCall
  .fibClosedForm
  .fibR

calcGM

calcMin
  restorePar
  scalePar
  show0

chooseWinVal
  getChoice
  setPBSoptions
  setWinVal

cleanProj
  .makeCleanVec
  createWin

clearAll

clearHistory
  .updateHistory
  getWinAct
  rmHistory

clearPBSext
  .initPBSoptions
  .removeFromList

clearWinVal
  getWinVal

closeWin
  .isReallyNull

compileDescription
  parseWinFile
  writeList

createVector
  createWin

createWin
  .createWidget
  .initPBSoptions
  .map.init
  .validateWindow
     DescList
  parseWinFile

declareGUIoptions
  .initPBSoptions
  .mergeVectors

drawBars

expandGraph

exportHistory
  getWinAct
  promptSaveFile
  writeList

findPat

findPrefix
  .stripExt
  getWinVal
  setWinVal

focusWin

forwHistory
  .updateHistory
  getWinAct
  setWinVal

genMatrix

getChoice
  createWin
  focusWin
  getPBSoptions
  setPBSoptions

getGUIoptions
  readPBSoptions
  setWinVal

getPBSext
  .isReallyNull

getPBSoptions

getWinAct

getWinFun

getWinVal
  .extractVar
  .isReallyNull

getYes

GT0

importHistory
  .updateHistory
  getWinAct
  promptOpenFile
  readList

initHistory

isWhat

jumpHistory
  .updateHistory
  getWinAct
  getWinVal
  setWinVal

openExamples
  .tryOpen
  setWinVal

openFile
  .initPBSoptions
  .isReallyNull
  getPBSext
  getWinAct
  openFile

openPackageFile
  openFile

openProjFiles
  .getPrefix
  .tryOpen
  showAlert

pad0

parseWinFile
  .getParamFromStr
  .parsegrid
  .parsemenu
  .stripComments
  .trimWhiteSpace

pause

pickCol

plotACF

plotAsp
```

plotBubbles

plotCsum
- addLabel
- resetGraph

plotDens

plotFriedEggs
- KernSmooth::bkde2D
- graphics::contour
- grDevices::
  contourLines

plotTrace

presentTalk
- .parseTalk
- .runTalk

promptOpenFile
- .trimWhiteSpace

promptWriteOptions
- .initPBSoptions
- .optionsNotUpdated
- getYes
- setGUIoptions
- writePBSoptions

promptSaveFile
- promptOpenFile

readList
- .readList.P

readPBSoptions
- .mergeLists
- readList

resetGraph

restorePar

rmHistory
- .updateHistory
- getWinAct
- setWinVal

runExamples
- closeWin
- createWin
- getWinAct
- getWinVal
- setWinAct
- setWinVal

scalePar

setFileOption
- .setOption

setGUIoptions
- .initPBSoptions
- getWinAct
- getWinVal
- setPBSoptions

setPathOption
- .setOption

setPBSext

setPBSoptions
- .initPBSoptions
- .removeFromList

setwdGUI
- findPrefix
- getWinAct

setWinAct

setWinVal
- .isReallyNull
- .setWinValHelper

show0

showAlert

showArgs

showHelp
- findPat
- openFile

showRes

showVignettes
- closeWin
- createWin

testCol

testLty

testLwd
- resetGraph

testPch
- resetGraph

testWidgets
- closeWin
- createWin
- getWinAct
- getWinVal
- setWinVal

unpackList

view

writeList
- .writeList.P

writePBSoptions
- .initPBSoptions
- writeList

D.3. 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:

http://cran.r-project.org/src/contrib/Descriptions/PBSmodelling.html

(Or from CRANS's root, locate "Packages" and find "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 is essential for the successful creation of a PDF manual.

Next we provide a choice of two methods that use the batch files RmakePDF.bat and RmakePDF2.bat to assist the user in building the manual. The first method alters a temporary TEX file *after* R's Perl script is run, and the PDF is built by calling MiKTeX commands. The second method modifies R's Perl script *before* it builds the TEX and PDF files. The final result of both methods yields a manual with letter (8.5″ × 11″) rather than A4 paper, and renumbering beginning on a specified page. This page number should be odd so that the next page becomes the front of a two-sided copy. Although the first method requires a redundant build of the document, it is possibly more robust to future changes in R's Perl script.

Method 1: On a command line, type the command:

```
RmakePDF PBSmodelling 89
```

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 uses R's Perl script by issuing the following command:

```
R CMD Rd2dvi --pdf --no-clean %1
```

This method creates a temporary directory called .Rd2dvi$\ containing Rd2.tex with the initial lines:

```
\nonstopmode{}
\documentclass[letter]{book}
\usepackage[times,hyper]{Rd}
\usepackage{makeidx}
\makeindex{}
\begin{document}
\setcounter{page}{89}
```

where a boldface red font indicates changes that `RmakePDF.bat` makes to the file `Rd2.tex`. The revised TEX file is then copied to `D:\Rdevel\PDFmodelling.tex` and the following MiKTeX commands are issued:

```
latex PBSmodelling
latex PBSmodelling
makeindex PBSmodelling
pdflatex PBSmodelling
```

(The second call to `latex` might not be needed, but it resolves a number of references. The `makeindex` command creates the table of contents.) You should now have the PDF manual called `PBSmodelling.pdf`, which can be appended to the first 88 pages of this report.

<u>Method 2</u>: On a command line, type the command:

```
RmakePDF2 PBSmodelling 89
```

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.

Essentially the script in `RmakePDF2.bat` modifies R's `Rd2dvi.sh` Perl script and saves it to the file `Rd2dvi4pbs.sh`, which sits in R's `bin\` directory. The batch file then issues the command:

```
R CMD Rd2dvi4pbs.sh --pdf --no-clean %1
```

which builds and creates the manual `PBSmodelling.pdf` in the `D:\Rdevel\` directory. The batch file also retains the temporary directory `.Rd2dvi$\` and copies the TEX file into the development directory. The PDF manual can be then be appended to this report (`PBSmodelling-UG.pdf`).

Once the user is satisfied with the results, he/she may wish to remove the temporary directory:

```
rm -rf .Rd2dvi$
```

The techniques 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