PBS Modelling 0.60: User's Guide
(Draft Report)

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

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

2006

**Canadian Technical Report of
Fisheries and Aquatic Sciences xxxx**

Fisheries and Oceans Canada   Pêches et Océans Canada

Canada

Canadian Technical Report of
Fisheries and Aquatic Sciences

Technical reports contain scientific and technical information that contributes to existing knowledge but which is not normally appropriate for primary literature. Technical reports are directed primarily toward a worldwide audience and have an international distribution. No restriction is placed on subject matter and the series reflects the broad interests and policies of the Department of Fisheries and Oceans, namely, fisheries and aquatic sciences.

Technical reports may be cited as full publications. The correct citation appears above the abstract of each report. Each report is abstracted in *Aquatic Sciences and Fisheries Abstracts* and indexed in the Department's annual index to scientific and technical publications.

Numbers 1 - 456 in this series were issued as Technical Reports of the Fisheries Research Board of Canada. Numbers 457 - 714 were issued as Department of the Environment, Fisheries and Marine Service Technical Reports. The current series name was changed with report number 925.

Technical reports are produced regionally but are numbered nationally. Requests for individual reports will be filled by the issuing establishment listed on the front cover and title page. Out-of-stock reports will be supplied for a fee by commercial agents.

Rapport technique canadien des
sciences halieutiques et aquatiques

Les rapports techniques contiennent des renseignements scientifiques et techniques qui constituent une contribution aux connaissances actuelles, mais que ne sont pas normalement appropriés pour la publication dans un journal scientifique. Les rapports techniques sont destinés essentiellement à un public international et ils sont distribués à cet échelon. Il n'y a aucune restriction quant au sujet; de fait, la série reflète la vaste gamme des intérêts et des politiques du ministère des Pêches et des Océans, c'est-à-dire les scences halieutiques et aquatiques.

Les rapports techniques peuvent être cités comme des publications complètes. Le titre exact paraît au-dessus du résumé de chaque rapport. Les rapports techniques sont résumés dans la revue *Résumés des sciences aquatiques et halieutiques*, et ils sont classés dans l'index annual des publications scientifiques et techniques du Ministère.

Les numéros 1 à 456 de cette série ont été publiés à titre de rapports techniques de l'Office des recherches sur les pêcheries du Canada. Les numéros 457 à 714 sont parus à titre de rapports techniques de la Direction générale de la recherche et du développement, Service des pêches et de la mer, ministère de l'Environnement. Les numéros 715 à 924 ont été publiés à titre de rapports techniques du Service des pêches et de la mer, ministère des Pêches et de l'Environnement. Le nom actuel de la série a été établi lors de la parution du numéro 925.

Les rapports techniques sont produits à l'échelon regional, mais numérotés à l'échelon national. Les demandes de rapports seront satisfaites par l'établissement auteur dont le nom figure sur la couverture et la page du titre. Les rapports épuisés seront fournis contre rétribution par des agents commerciaux.

Canadian Technical Report of

Fisheries and Aquatic Sciences xxxx

2006

PBS Modelling 0.60: User's Guide
(Draft Report)

by

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

Fisheries and Oceans Canada

Science Branch, Pacific Region

Pacific Biological Station

3190 Hammond Bay Road

Nanaimo, British Columbia

V9T 6N7

CANADA

Cat. No. Fs97-6/2549E ISSN xxxx-xxxx

9 8 7 6 5 4 3 2 1 (First printing – MMM DD, YYYY)

Correct citation for this publication:

Schnute, J.T., Couture-Beil, A., and Haigh, R. 2006. PBS Modelling 1: user's guide v.0.60 (draft version). Can. Tech. Rep. Fish. Aquat. Sci. xxxx: xxx + xxx p.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Schnute, J.T., Couture-Beil, A., and Haigh, R. 2006. PBS Modelling 1: user’s guide v.0.60 (draft version). Can. Tech. Rep. Fish. Aquat. Sci. xxxx: xxx + xxx p.

This draft report describes the R package *PBS Modelling*, 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. *PBS Modelling* 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, *PBS Modelling* 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 other R libraries, such as *PBS Mapping*, *odesolve*, and *BRugs*. Users interested in building new packages can use *PBS Modelling* and a simpler enclosed package *PBS Try* as prototypes. An appendix describes this process completely, including the use of C code for efficient calculation.

RÉSUMÉ

Schnute, J.T., Couture-Beil, A. et Haigh, R. 2006. PBS Modelling 1: Guide de l’utilisateur v.0.60 (version provisoire). Can. Tech. Rep. Fish. Aquat. Sci. xxx: xxx + xxx p.

Google translation: Ce projet de rapport décrit le paquet PBS de R modelant, qui contient le logiciel pour faciliter la conception, l'essai, et l'opération des modèles d'ordinateur. Les initiales PBS se rapportent à la station biologique Pacifique, un laboratoire important de pêche sur la côte Pacifique du Canada dans Nanaimo, Colombie britannique. Au commencement conçu pour des scientifiques de pêche, ce paquet a la large application potentielle dans beaucoup de domaines scientifiques. PBS modelant des foyers en particulier sur les outils qui le rendent facile à construire et éditer une interface utilisateur graphique adaptée aux besoins du client (GUI) s'approprient pour un problème particulier. Bien que notre paquet dépende fortement de l'interface de R à Tcl/Tk, un utilisateur n'a pas besoin de savoir Tcl/Tk. En plus des outils de conception de GUI, modeler de PBS fournit des utilités à l'échange d'informations supplémentaires parmi les composants modèles, les analyses statistiques spécialisées par conduite, et les graphiques de produit utiles dans la pêche modelant et l'analyse de données. Les exemples mettent en application des idées classiques de la littérature de pêche, aussi bien que nos propres papiers édités. Les exemples fournissent également des calibres pour concevoir des analyses adaptées aux besoins du client en utilisant d'autres bibliothèques de R, telles que tracer de PBS, odesolve, et BRugs. Les utilisateurs intéressés à de nouveaux paquets de bâtiment peuvent employer modeler de PBS et un essai inclus plus simple du paquet PBS comme prototypes. Une annexe décrit ce processus complètement, y compris l'utilisation du code de C pour le calcul efficace..

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. Furthermore, the associated network of contributed libraries on CRAN (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.

At various times, I've used software in Basic, 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.

PBS Modelling 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 the Tcl/Tk library into convenient tools fully documented here. A user need not learn Tcl/Tk to use this package. Everything required appears in Appendix B. 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 5. The code for all of them appears in the R library directory `PBSmodelling\Examples`. We demonstrate the power of other R libraries, such as `BRugs` (to perform Bayesian posterior sample with the application `WinBUGS`), `odesolve` (to solve differential equations numerically), and `PBSmapping` (to draw maps and perform spatial analyses).

Third, *PBS Modelling* serves as a prototype for building a new R package, as summarized in Appendix A. We illustrate two methods of calling C code (`.C` and `.Call`), and discuss many other technical issues encountered while building this library.

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 *PBS Modelling* inspires interest in interactive models that demonstrate applications in many fields. Enjoy!

Jon Schnute, August 2006

1. Introduction

This draft report describes software to facilitate the design, testing, and operation of computer models. The package *PBS Modelling* is distributed as a freely available library 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 library *PBS Mapping* (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, often pronounced "gooey") facilitates this exploratory process. *PBS Modelling* 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 libraries.

In addition to GUI design tools, *PBS Modelling* 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 libraries discussed here. In part, *PBS Modelling* provides a (very incomplete) guide to the variety of analyses possible with the R framework. We anticipate many revisions of our library, as we find time to include more examples.

PBS Modelling 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). In our library, 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 B). A user of *PBS Modelling* does not need to know Tcl/Tk.

Computer models typically involve a variety of components, such as code, data, a user interface, and documentation. Figure 1 illustrates the tangled relationships that sometimes accompany computer model design. *PBS Modelling* 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.

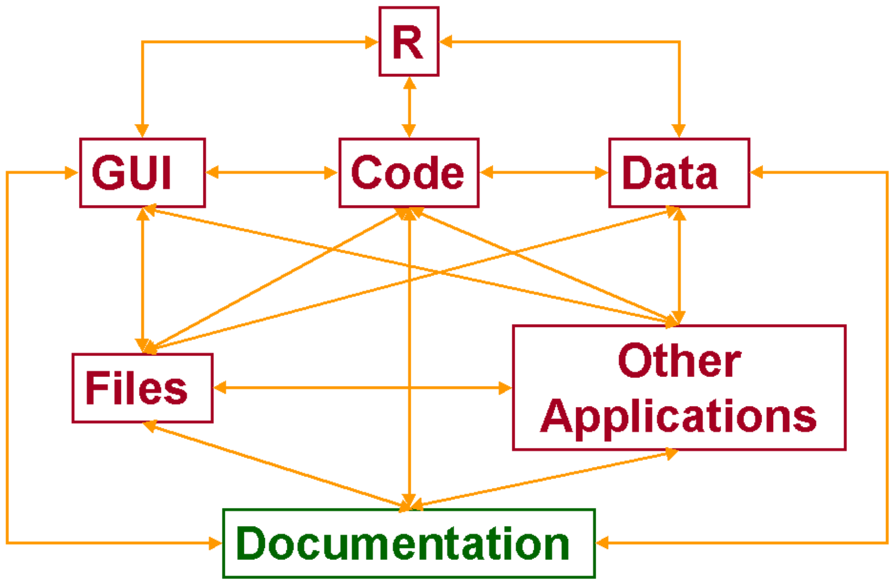


Figure 1. Tangled relationships among computer model components.

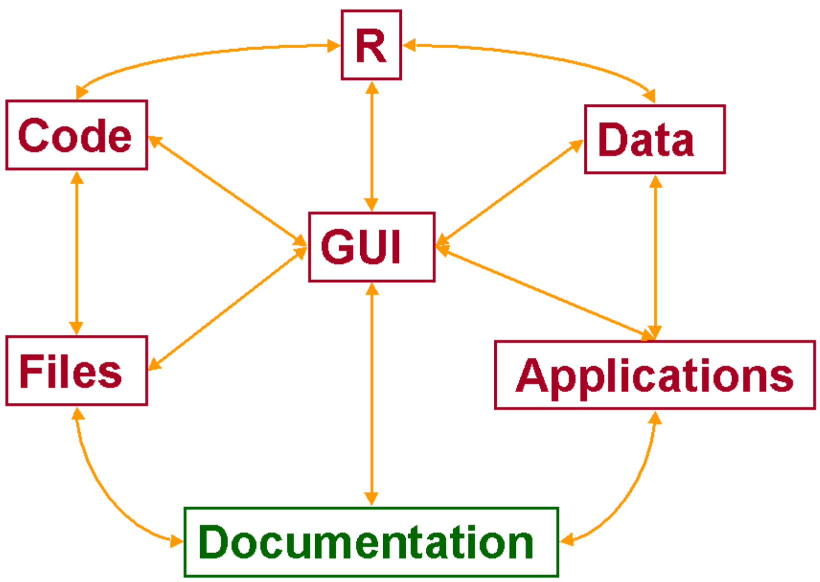


Figure 2. Computer model components organized with a graphical user interface (GUI).

In *PBS Modelling*, 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 code written in languages other than R. For example, the R *BRugs* library (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. In Section 5, we highlight briefly some of the examples in our initial release, although we expect the list to expand in future versions.

Appendix A describes the process of building *PBS Modelling* in a Windows environment. A simple enclosed package *PBS Try* gives a prototype for building any R package, including the use of C code to speed calculations. Appendix B gives the complete syntax for all visual components (called *widgets*) available for a writing a window description file to specify a customized GUI. Appendix C shows the help files included with the library.

To use *PBS Modelling*, 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 zip file from the authors of this report or directly from the CRAN web site http://cran.r-project.org/.

2. GUI tools for model exploration

The practical task of writing appropriate code for the R Tcl/Tk package can sometimes become a bit daunting, particularly if the GUI window requires extensive design and change. For a restricted set of Tk components (called *widgets*), *PBS Modelling* 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

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

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 *PBS Modelling* 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 *PBS Modelling* 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:

$$m = 2,\ n = 3,\ \phi = 0,\ k = 1000\,. \tag{2}$$

The GUI allows different Lissajous figures to be drawn easily. Simply change parameter values in the four entry boxes, and click on <Plot>.

The 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 is actually drawn (i.e., the R function `drawLiss` is called) when the user presses a `button` that contains the text “Plot”. In general, we use the symbols <…> to designate a button or keystroke, such as the <Plot> button or the radio buttons <lines> and <points>.

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 *PBS Modelling* 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.)

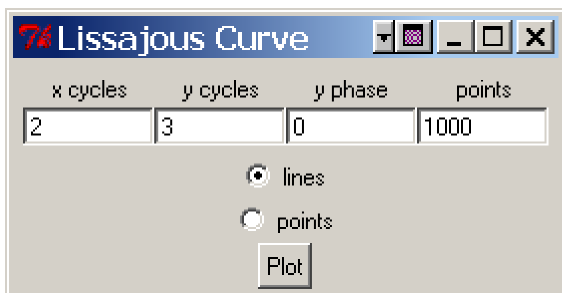


Figure 3. GUI generated by the description file `LissajousCurve.txt` in Table 1. It contains five widgets: the window titled “Lissajous Curve”, a vector of four entries, two linked radio buttons (<lines> and <points>), and a <Plot> button.

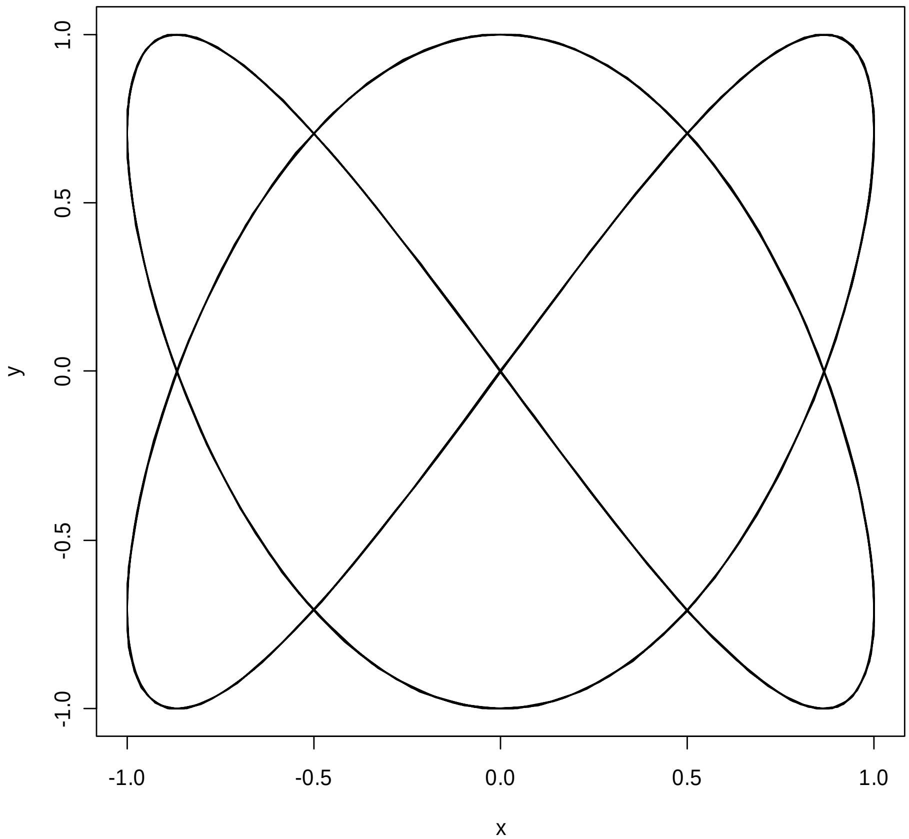


Figure 4. Default graph for the “Lissajous Curve” project, obtained by clicking the <Plot> button in Figure 3. The *x* variable goes through two cycles while the *y* variable goes through 3 cycles. A line graph is drawn through 1000 points generated by the algorithm (1).

2.2. Window description file

A window description file currently supports the following 17 widgets:

1. `window` – an entire new window;
2. `menu` – a menu grouping;
3. `menuitem` – an item in a menu;
4. `grid` – a rectangular block of 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 off or on, with corresponding values 0 or 1;
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. `slide` – a slide bar that sets the value of a variable;
16. `slideplus` – an extended slide bar that also displays a minimum, maximum, and current value;
17. `history` – a device for archiving parameter values corresponding to different model choices, so that a "slide show" of interesting choices can be preserved.

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 17 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. 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

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

Although `createWin` normally builds a GUI from a description file, it will also accept a vector of strings equivalent to such a file. Thus, a file of R source code can define a GUI directly, without the need for a separate description file. Table 2 illustrates how this can be done in a simple case. 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, *PBS Modelling* uses a `list` object in the process of generating a GUI from a description file. 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. *PBS Modelling* provides three 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;
- `clearWinVal` clears global values associated with the current window.

Some models associate a particular action with a single parameter vector. In such cases the function `createVector` generates a GUI directly, without the need for a corresponding description file.

Several functions support file display and manipulation from a GUI:

- `openFile` opens a file using the default program for the file extension;
- `promptOpenFile` shows the current directory for choosing a file to open;
- `promptSaveFile` shows the current directory for naming a file to save.

PBS Modelling 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.

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

- `initPBShistory` initializes data structures for holding a collection of history data;
- `addPBShistory` saves the current window settings to the current history record;
- `rmPBShistory` removes the current record from the history;
- `backPBShistory` and `forwPBShistory` move backward and forward among the history records;
- `jumpPBShistory` moves to a specified record in the history;
- `exportPBShistory` and `importPBShistory` save and load histories from files;
- `clearPBShistory` removes all records from the current collection.

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

2.4. Internal data for windows

PBS Modelling uses two list variables `PBS.win` and `PBS.options` in the global environment to store information relevant to its current settings. In particular, `PBS.win` has four components that contain data about the current GUI window, where

- `$vars` is a vector with the names of all R variables declared in the description file;
- `$funs` is a vector with the names of all R functions declared in the description file;
- `$actions` is a vector with action names (optionally declared in the description file) that indicate recent actions taken by the user in the current GUI;
- `$windowname` is the name of the currently active window.

The functions `showVars`, `showFuns`, `showActions`, and `showWin` can also be used to return these character vectors. If multiple windows are present, `PBS.win` automatically gets updated to the data for the window currently selected.

After using `createWin` to produce a GUI, the vectors `PBS.win$vars` and `PBS.win$funs` provide useful summaries of names declared in the current project. Furthermore, the vector `PBS.win$actions` 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.

If a widget invokes the function `openFile`, the associated `action` should be the file name. By definition, `openFile` has the default argument `PBS.win$actions[1]`.

Currently, `PBS.options` acts primarily to store default program names associated with file extensions. 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:

`PBS.options$openFile=list()`.

The default can, however, be overwritten by specifying explicit list components, such as:

```
html='"c:/Program Files/Mozilla Firefox/firefox.exe" %f'
```

where `%f` denotes the file name in the string passed to the operating system. On Unix platforms, it may be essential to specify default this way. Future versions of our library may include other options, such as default width for a data entry field or the maximum number of `actions`.

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. *PBS Modelling* 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 colnames="" byrow=TRUE
$$ data modes=numeric ncol colnames byrow=TRUE
$$ array mode=numeric dim fromright=TRUE
```

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`.

Table 3. Sample data file for *PBS Modelling*. 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
```

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.

As indicated earlier, *PBS Modelling* 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.

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 this 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 2006). Others just provide convenient utilities. For example, `testCol("red")` shows all colours in the palette `colors()` that contain the string `"red"`.

4.1. Graphics utilities

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

`drawBars`................Draw a linear bar plot on the current graph
`plotCsum`................Plots cumulative sum of a vector, with value added

`plotBubbles`.........Construct a bubble plot for a matrix
`genMatrix`.............Generate a test matrix for use in `plotBubbles`

`addArrows`.............Calls `arrows` function using relative (0:1) coordinates
`addLegend`.............Adds a legend using relative (0:1) coordinates
`addLabel`................Adds a panel label using relative (0:1) coordinates

`testCol`...................Display colours available
`testLty`...................Display line types available
`testLwd`...................Display line widths
`testPch`...................Display plotting symbols
`testGr`.....................GUI to do some/most/all of the above
`pickCol`...................Pick a color from a complete palette

4.2. Data management

`clearAll`................Function to clear all data in the global environment

`pad0`..........................Pads numbers with leading zeroes (string)
`show0`........................Shows decimal places including zeroes (string)
`view`..........................Views first n rows of a data.frame or matrix

`comparePars`.........Find the difference between two par vectors ***** not implemented
`compareLists`.......Find the difference between two lists

4.3. Function minimization and maximum likelihood

***** These concepts have not been implemented in *PBS Modelling 0.60*.

<u>New data type:</u>
`parVec` – a data frame with columns `val`, `min`, `max`, `active` (logical), and possible row names;

`scalePar` scales parameters to [0,1]
`restorePar` get actual parameters from scaled values

`calcMin(pvec,func,tol)` calculates the minimum of `func`, starting at `pvec`

4.4. Handy utilities

`calcGM`.....................Calculates the geometric mean of a vector of numbers
`findPat`...................Find all strings that include any string in a vector of patterns
`pause`........................Pause, typically between graphics displays
`showArgs`................Show the arguments for a specified widget in Appendix B
`testWidgets`.........GUI to test all widgets listed in Appendix B

`view`.........................View of first few lines of a (potentially large) matrix or data frame

5. Examples

As mentioned in the Preface, *PBS Modelling* includes a variety of examples that illustrate applications based on this and other libraries. Generally, each example comprises R-code, a window description file, documentation, and other supporting files. All relevant code appears in the R library directory `PBSmodelling\Examples`, where example `xxx` typically has corresponding files `xxx.r`, `xxxWin.txt`, and `xxxDoc.txt` or `xxxDoc.pdf`. The function `runExamples()` brings up a window that runs the examples in a temporary directory located on the path defined by the environment variable `Temp`.

Alternatively, you can copy all the files from `PBSmodelling\Examples` to a directory of your choice and open R in this working directory. To run example `xxx`, type `source("xxx.r")` on the R command line. For instance, `source("LissFig.r")` creates Lissajous figures (described earlier) and includes the history widget for collecting settings that the user wishes to retain. Sourcing `LissFig.r` invokes the windows description file `LissFigWin.txt`, which produces the GUI.

These examples work correctly only if a user's computer has been set to associate ".txt" and "r" with suitable text editors. Similarly, the Acrobat Reader must be installed, so that "*.pdf" is associated with that program. In the descriptions below, we often refer to GUI elements in quotation marks. For example, "Model" usually refers to the button labelled "Model". In some cases, this becomes shorthand for "Press the button labelled Model".

5.1. Random variables

5.1.1. `RanVars` – Random variables

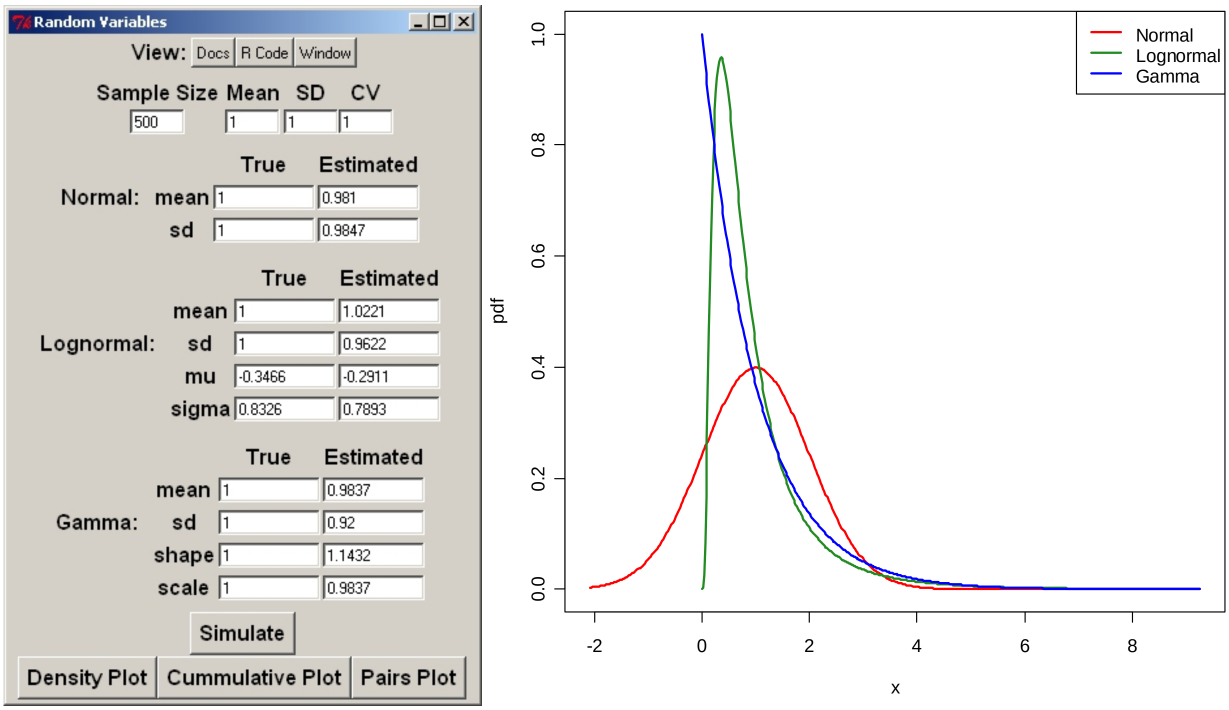


Figure 5. `RanVars` GUI (left) and density plot (right). Simulations are based on 500 random draws with mean =1 and SD = 1.

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 5). We use only the straightforward formulas in the documentation, without 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.

5.1.2. `RanProp` – Random proportions

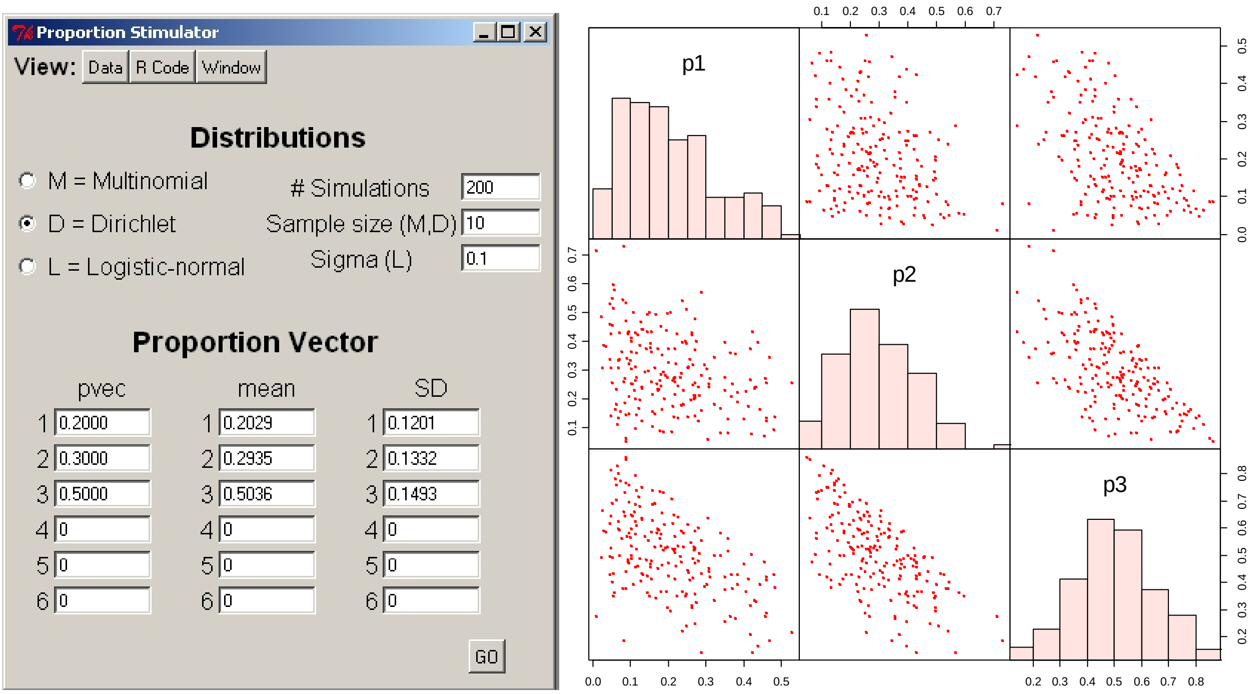


Figure 6. `RanProp` GUI (left) and pairs plot (right). Simulations are based on 200 random draws where $n = 10$ for the multinomial and Dirichlet distributions and $\sigma = 0.1$ for the logistic-normal distribution. The pairs plot portrays results for the Dirichlet.

The `RanProp` example simulates up to five random proportions drawn from one of three distributions – multinomial, Dirichlet, and logistic-normal. The observed proportion means and standard deviations are reported in the GUI (Figure 6), 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 (2006) provide further technical details about these three distributions.

***** This example still needs a documentation file.

5.1.3. `SineNorm` – Sine normal

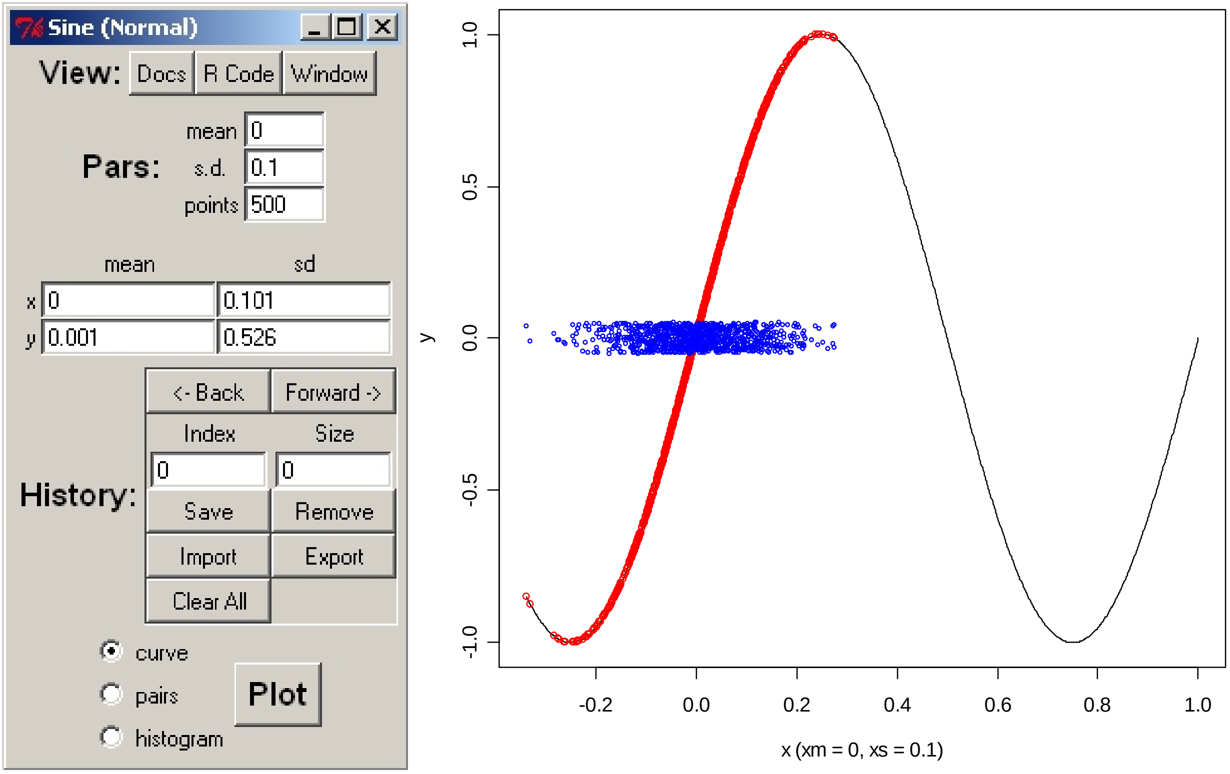


Figure 7. `SineNorm` GUI (left) and plot (right). Simulations are based on 500 random draws of $y = \sin(2\pi x)$, where *x* is normal with mean $\mu = 0$ and standard deviation $\sigma = 0.1$. Blue points portray jittered values of *x*, and red points show corresponding values of *y*.

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 7 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.

5.2. Statistical analyses

5.2.1. `LinReg` – Linear regression

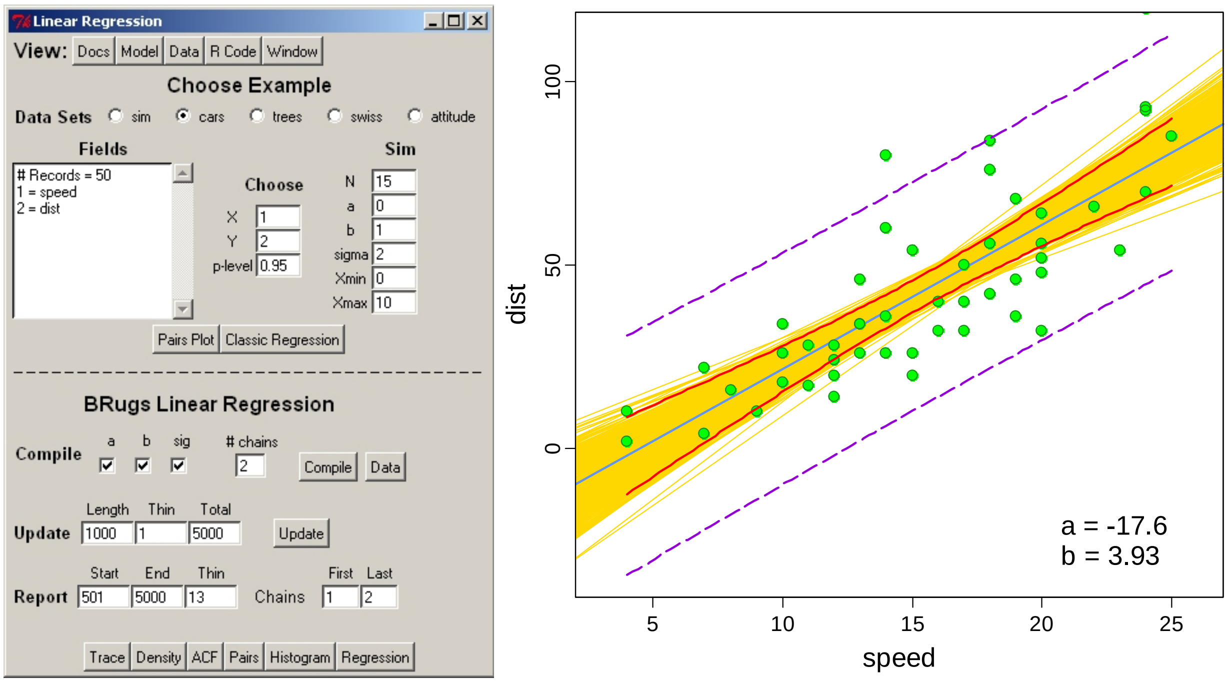


Figure 8. `LinReg` GUI (left) and regression plot (right). The linear regression uses the `cars` dataset (n=50) to predict `dist` vs. `speed`. The plot shows observations (green circles), fitted line (solid blue line), the 95% confidence limits of the fitted model (solid red lines), the 95% CL of the data (dashed purple lines), and the fits using the Bayes posterior estimates of (a,b) (gold lines).

The example `LinReg` estimate parameters in a linear regression $y = a + bx$ using either simulated data or data objects that come with the R-package. We compare classical frequentist regression with results from Bayesian analysis, using the BRugs library 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 8).

5.2.2. `MarkRec` – Mark-recovery

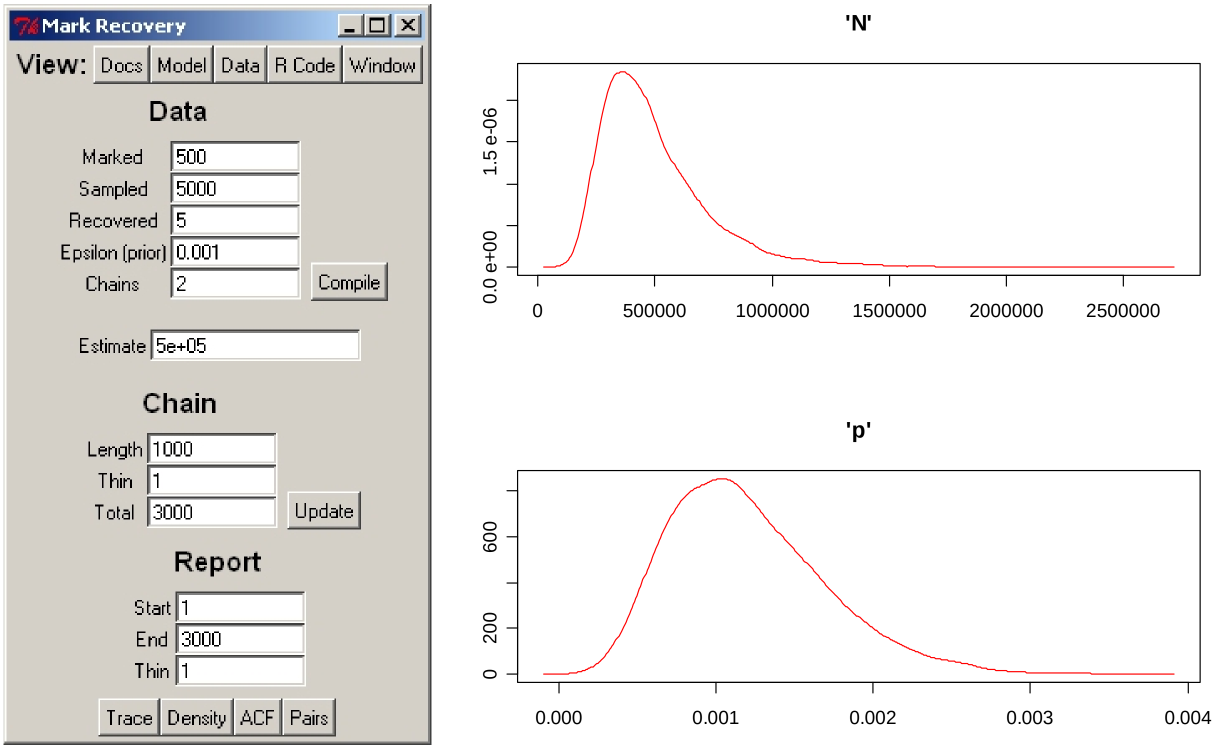


Figure 9. `MarkRec` GUI (left) and density plots (right). A low recovery of marked fish can lead to fat tails in *N* due to occasional large spikes in the population estimate.

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 9).

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.

5.2.3. CCA – Catch-curve analysis

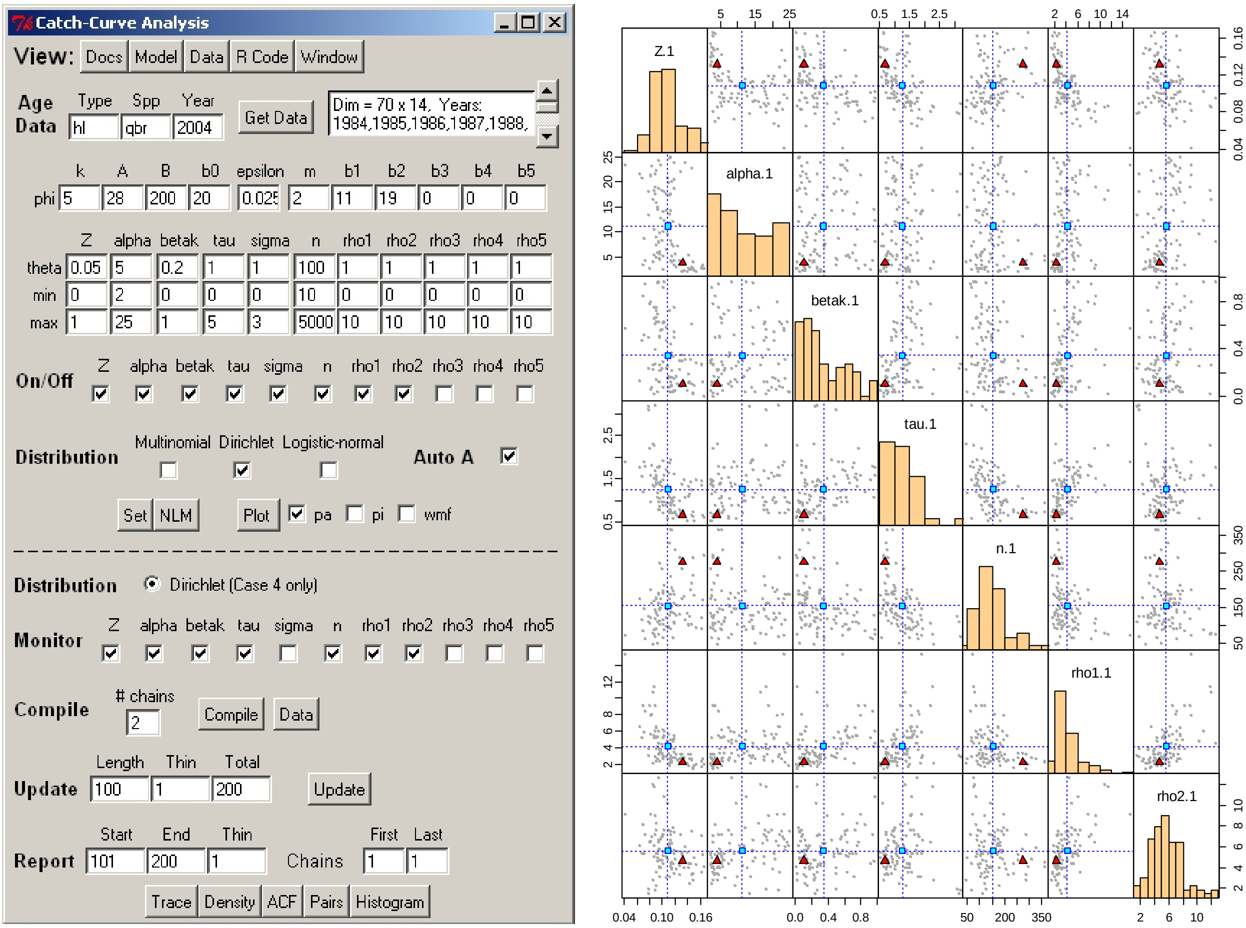


Figure 10. CCA GUI (left) and parameter pairs plot (right). Comparison of Bayes posterior distribution of CCA model parameter estimates from chain 1 (N=100). Symbols indicate means (blue squares) and modes (red triangles). Diagonal shows parameter estimate distributions.

The example CCA illustrates a catch-curve model proposed by Schnute and Haigh (2006). 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 10). “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. We will provide further information when the paper (Schnute and Haigh 2006) is published. *PBS Modelling* includes the data for this example as the matrix CCA.qbr.hl.

5.3. Other applications

5.3.1. `FishRes` – Fishery reserve

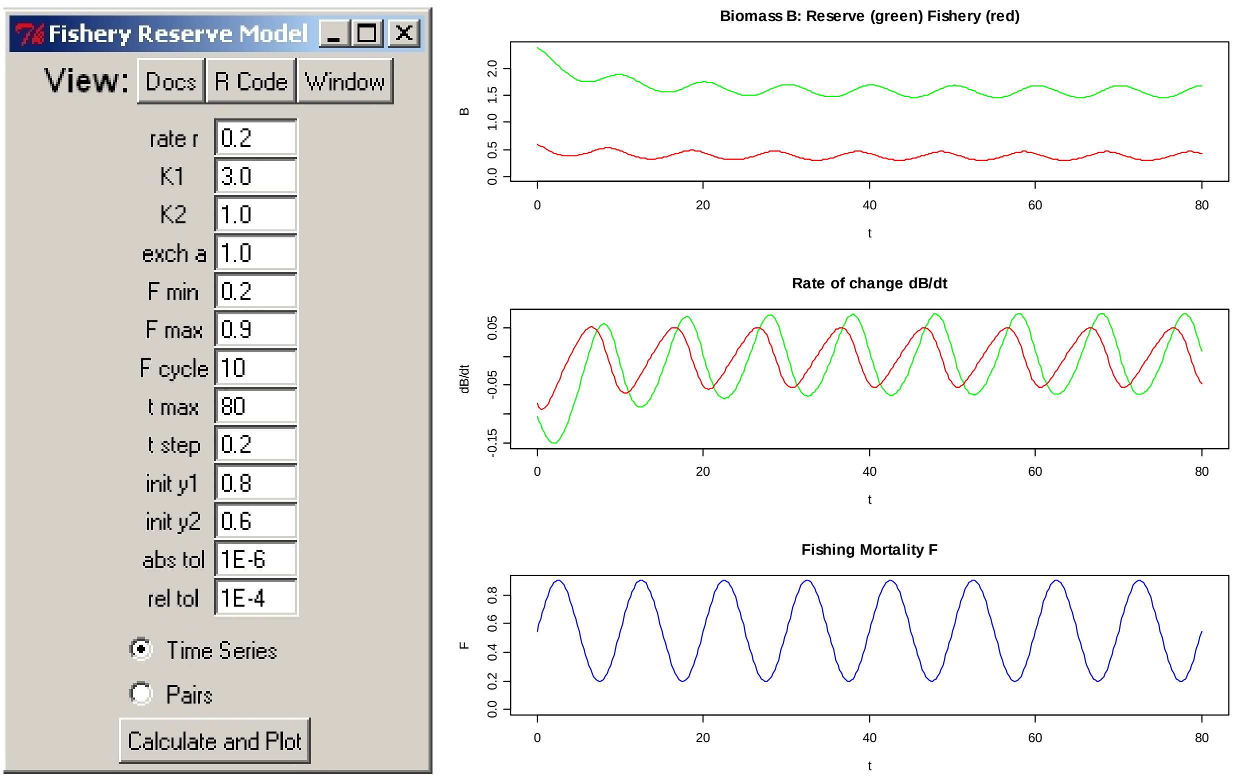


Figure 11. `FishRes` GUI (left) and time series (right) of population biomass, rates of biomass change, and fishing mortality.

The example `FishRes` models a fish population associated with a marine reserve using differential equations, which are solved numerically with the `odesolve` library. The dynamic equations:

$$\frac{dB_1}{dt} = rB_1\left(1 - \frac{B_1}{K_1}\right) + a\left(\frac{B_2}{K_2} - \frac{B_1}{K_1}\right)$$

$$\frac{dB_2}{dt} = rB_2\left(1 - \frac{B_2}{K_2}\right) - a\left(\frac{B_2}{K_2} - \frac{B_1}{K_1}\right) - F(t)B_2$$

$$F(t) = F_{\min} + \frac{F_{\max} - F_{\min}}{2}\left(1 + \sin\frac{2\pi t}{n}\right)$$

describe the biomass B_i in region $i = 1, 2$, where region 1 is a reserve and region 2 experiences a periodic fishing mortality rate *F*, with minimum and maximum values $F_{\min}$ and $F_{\max}$. The two regions have a common growth rate *r*, but different carrying capacities K_i. A parameter *a* determines the movement rate from the region of higher density to the other region (Figure 11).

5.3.2. `FishTows` – Fishery tows

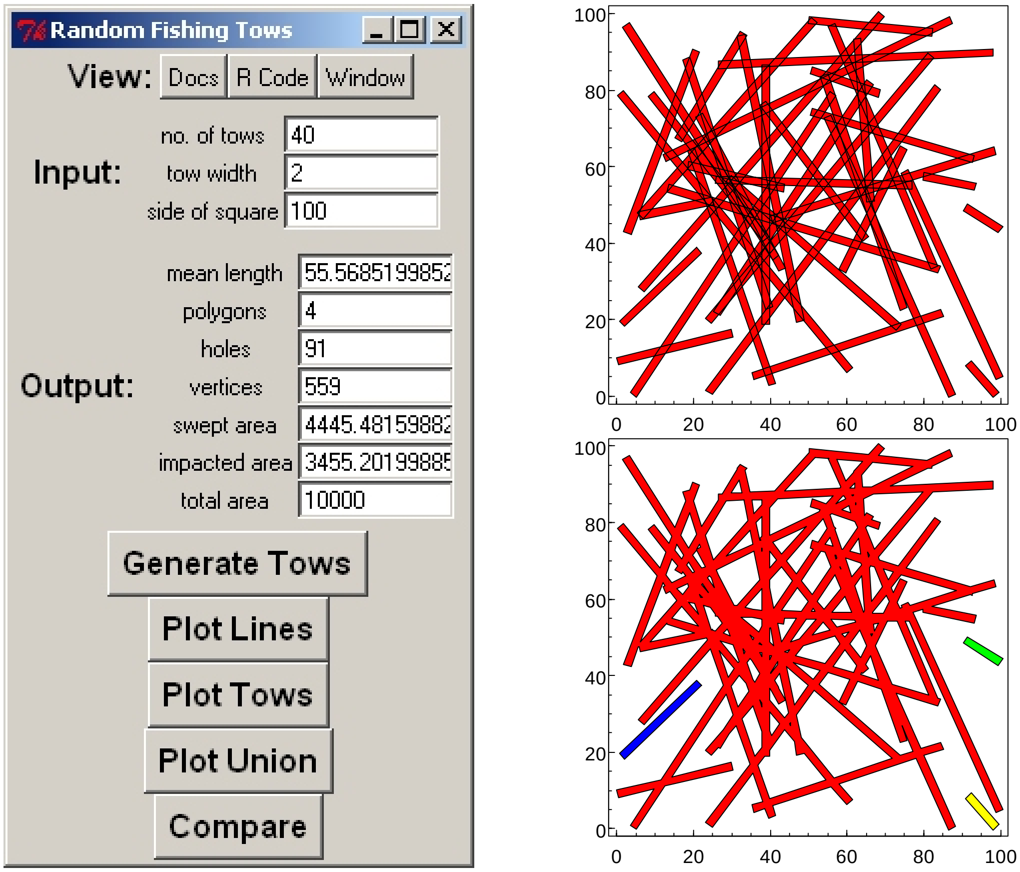


Figure 12. `FishTows` GUI (left) and simulated tow track (right). Tow track plots show 40 random tows in a square with side length 100. Each tow has width 2, and the rectangle encompasses 10,000 square units. *Top*: The individual rectangles, with 160 vertices, have areas that sum to 4,445 square units. *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` library. 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 J.A.C. Brown. 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/

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

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. 2006. Compositional analysis of catch curve data with an application to *Sebastes maliger*. ICES Journal of Marine Science (in review).

Spiegelhalter, D., A. Thomas, N. Best, and D. Lunn. 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. 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:

A.1. install the required software;
A.2. build PBS Modelling from source materials;
A.3. write source materials for a new package and compile them;
A.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-2.3.1\library\PBSmodelling\PBStools`.

A.1. Installing required software

Building R packages requires six pieces of free software. Even if some of this software is already installed, it may be helpful (or even essential) to update to the latest versions. 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 shows versions available at the time of writing this report, along with suggested paths.

1. **R** itself, currently version 2.3.1 (`C:\Utils\R\R-2.3.1`). 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. We also assume that the package `PBSmodelling` is installed in R.

2. A set of **UNIX tools** for building R packages in Windows (no version number shown, `C:\Utils\Rtools`). Obtain the file `tools.zip` from the web site http://www.murdoch-sutherland.com/Rtools/. Uncompress this file in `C:\Utils`, and (if you wish) rename the base directory from `\tools` to `\Rtools`. 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. **Perl**, currently version 5.8.8.817 (`C:\Utils\Perl`). Obtain this from the Active State web site: http://www.activestate.com/Products/ActivePerl/Download.html.

4. **MinGW**, currently version 5.0.3 (`C:\Utils\MinGW`), the minimalist GNU C compiler for Windows. Obtain a small installation file from http://www.mingw.org/, and use this as a package manager to download and install at least the basic components.

   Alternatively, you may wish to obtain **Dev-C++**, currently version 4.9.9.2 (`C:\Utils\DevCpp`). Obtain a complete installation file from http://www.bloodshed.net/. This package includes a distribution of MinGW and a convenient integrated development environment (IDE) for compiling and testing C programs. Based on our current experience, Dev-C++ works as an adequate substitute for MinGW in building R packages.

5. **MiKTeX**, currently version 2.5 (`C:\Utils\MiKTeX`), available from http://www.miktex.org/. This processor for TeX and LaTeX files helps typeset help files within a package. If you don't have version 2.5 or later, take this opportunity to upgrade. 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").

   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.

6. The Microsoft **HTML Help Workshop** (no version shown, `C:\Utils\HHW`). You need the installation file `HtmlHelp.exe`, currently at http://www.microsoft.com/office/ork/xp/appndx/appa06.htm.
   If Microsoft no longer makes it available there, you can obtain the file from http://www.murdoch-sutherland.com/Rtools/. 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.)

After these six pieces of software are installed, you're ready to start building R packages. For this purpose, create a new directory that can contain your packages; such as `D:\Rdevel\`. Within the R library directory, which could be `C:\Utils\R\R-2.3.1\library\`, find the subdirectory `PBSmodelling\PBStools`. Copy all the batch files there into your new packages directory. You should have these ten files:

- `definePaths.bat`, `checkPaths.bat` related to the installation;
- `checkPBS.bat`, `buildPBS.bat`, `packPBS.bat`, `unpackPBS.bat` related to *PBS Modelling*;
- `check.bat`, `build.bat`, `pack.bat`, `unpack.bat` related to the construction of new packages.

You need to change `definePaths.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-2.3.1\bin
set TOOLS_PATH=C:\Utils\Rtools\bin
set PERL_PATH=C:\Utils\Perl\bin
set MINGW_PATH=C:\Utils\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 you new packages directory, run `checkPaths.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.

You may wish to inspect all the batch files with a text editor. They don't use your system PATH environment variable. Instead, each one defines a new local path appropriate for building R packages (via `checkPaths.bat`). A `SETLOCAL` command ensures that this change doesn't alter your system's permanent environment.

A.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 A.1, say `D:\Rdevel`, for building R packages and that it contains the relevant eight batch files. In particular, `definePaths.bat` should reflect your installation paths and `checkPaths.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`.

   Our batch file uses the command `tar -xzvf PBSmodelling.tar.gz`, where `tar.exe` appears in the `\Rtools` directory (section A.1, step 2). 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.

3. 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 ...”.

   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 A.1. Choose a remote server near you. You only need to do this once. When it’s finished, run `checkPBS.bat` again.

4. Examine the new directory `E:\Rdevel\PBSmodelling.Rcheck` created by the `check` process in step 2. The text files `00check.log` and `00install.out` show detailed results.

5. 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.

6. Again in the base directory `D:\Rdevel`, double-click the icon for `packagePBS.bat` or type the command `packagePBS` 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.

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

A.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., WinEdt or the 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, such as `D:\Rdevel`, and rename it (or copy and rename the copy) to obtain the file `PBStry.tar.gz`.

2. Follow steps similar to those in section A.2 to unpack, check, build, and re-package `PBStry`. You must now use a DOS command window in `D:\Rdevel` to issue the four commands
```
unpack PBStry
check PBStry
build PBStry
pack PBStry
```
which invoke the batch files `unpack.bat`, `check.bat`, `build.bat`, and `package.bat`. The first command should give you a new subdirectory `\PBStry`, along with its five sudirectories: `\data`, `\inst`, `\man`, `\R`, and `\src`.

3. 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.

```
Package: MyPack
Version: 1.00
Date: 2006-08-31
Title: My R Package
Author: User of PBS Modelling
Maintainer: User of PBS Modelling
Depends: R (>= 2.3.1)
Description: My customized R functions
License: GPL version 2 or newer (recommended)
```

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.

4. The subdirectory `\PBStry\R` contains all R code used by the package. For example, `PBStry` includes six R functions (`calcFib`, `calcGM`, `calcSum`, `findPat`, `pause`, and We `view`). The six 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`. Two of them (`calcFib`, `calcSum`) call compiled C code, as we discuss more completely in section A.4 below.

   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.

   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`.

5. 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 5.2.3). The same data matrix is called CCA.qbr.hl in PBSmodelling.

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.

6. 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).
7. 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) and to add the components of a numeric vector (sum.c). In section A.4, we discuss the linkage between R code and compiled C functions.
8. 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, see the file PBSMfunctions.txt in the subdirectory \inst or the R library directory for PBSmodelling.) The namespace file must also import functions required from other packages. Because PBSmodelling relies heavily 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 4 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.

A.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. PBSmodelling includes a simple example to illustrate this calling convention.

Calling C functions from R using .C()

Table A1. 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.

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 A.1), 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 A2. 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 A2). 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 the second example in `PBStry` , associated the files `calcSum.R` and `sum.c`.

Table A3. `.Call()` example adapted from `PBSmodelling`, with two associated text files. R code in the first file calls C code in the second.

File 1: callFib.R

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

} **File 2: fib2.c**

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

Calling C functions from R using `.Call()`

The `.C()` convention requires a fairly simple conversion of R objects into C types (Table A.1). 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 `callFib` in Table A3 (adapted from `.fibCall` in `PBSmodelling`) 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. The code in `callFib` assumes that `n` and `len` already satisfy the necessary constraints. 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 A3) 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 A2. 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 B. Widget descriptions

This appendix lists PBS Modelling widgets in alphabetical order. 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.

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="" width=0 function=""
  action="button" sticky="" padx=0 pady=0
```

Arguments

`text` ....text to display on the button
`font` ....font for button text - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
`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` ....value for `PBS.win$action` when pressing this button
`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
`pady` ....space used to pad the widget on the top and bottom

Example

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

Check

Description

A check box used to turn a variable off or on, with corresponding values `FALSE` or `TRUE`.

Usage

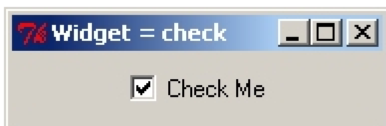
```
type=check name checked=FALSE text="" font="" function=""
  action="check" sticky="" padx=0 pady=0
```

Arguments

`name`....................name of R variable altered by this check box (required)
`checked`............if `TRUE`, the box is checked initially and the variable is set to 1
`text`....................identifying text placed to the right of this check box
`font`....................font for check text - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
`function`..........R function to call when the check box is changed
`action`...............value for `PBS.win$action` when altering this check 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
`pady`....................space used to pad the widget on the top and bottom

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= font="" values=
  byrow=TRUE function="" enter=TRUE action="data" width=6
  sticky="" padx=0 pady=0
```

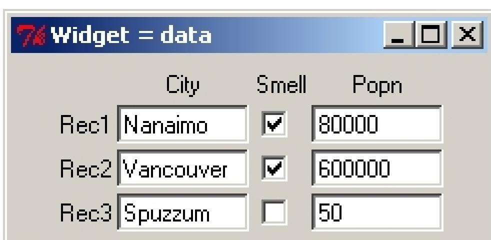
Arguments

`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`........either one label or a vector of `nrow` labels used to label rows of this data frame in the display

collabels........either one label or a vector of ncol labels used to label columns of this data frame in the display
font....................font for labels - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
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...............value for PBS.win$action when changing any component of this data frame
width.................character width to reserve for the each entry in the data frame
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
pady....................space used to pad the widget on the top and bottom

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"
```



Entry

Description

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

Usage

```
type=entry name value="" width=20 label="" font="" function=""
  enter=TRUE action="entry" mode="numeric" sticky="" padx=0
  pady=0
```

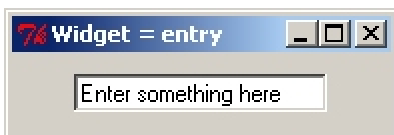
Arguments

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 label - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
`function`..........R function to call when the entry is changed
`enter`..................if `TRUE`, call the function only after the <Enter> key is pressed
`action`...............value for `PBS.win$action` when making this entry
`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
`pady`.....................space used to pad the widget on the top and bottom

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="" byrow=TRUE borderwidth=1 relief="flat"
   sticky="" padx=0 pady=0
```

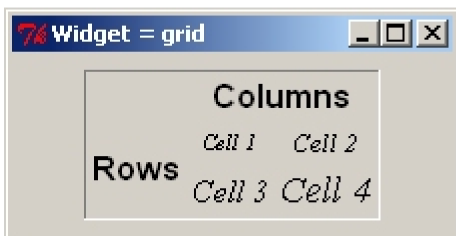
Arguments

`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 of grid - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
`sidefont`..........font for left of grid - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
`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
`pady` ....space used to pad the widget on the top and bottom

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 save and recover previous combinations of widget settings. A desired realization can be saved to a hidden stack using the `Add` button. Each saved realization has an `Index` within the stack of total saves `Size`. The user can scroll `back` and `forward` through the stack, invoking saved widget settings. Realizations no longer desired can be deleted using the `Remove` button.

Usage

```
type=history name="default" archive=TRUE sticky="" padx=0
  pady=0
```

Arguments

`name` ....name of history archive
`archive` ....if TRUE, user can import previous sessions or export the current session
`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
`pady` ....space used to pad the widget on the top and bottom

Example

```
window title="Widget = history"
history archive=TRUE
```

Label

Description

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

Usage

```
type=label text="" font="" sticky="" padx=0 pady=0
```

Arguments

`text` ....text to display in the label
`font` ....font for label - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
`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
`pady` ....space used to pad the widget on the top and bottom

Example

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

Matrix

Description

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

Usage

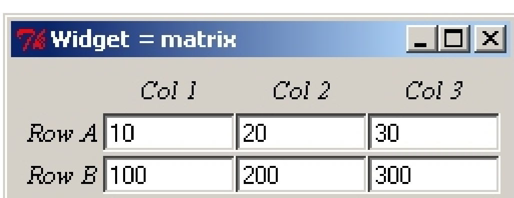
```
type=matrix nrow ncol names rowlabels= collabels= rownames=""
  colnames="" font="" values="" byrow=TRUE function=""
  enter=TRUE action="matrix" mode="numeric" width=6 sticky=""
  padx=0 pady=0
```

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`........either one label or a vector of `nrow` labels used to label rows of this matrix in the display
`collabels`........either one label or a vector of `ncol` labels used to label columns of this matrix in the display
`font`....................font for labels - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
`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`...............value for `PBS.win$action` when changing any component of this matrix
`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
`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
`pady`....................space used to pad the widget on the top and bottom

Example

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



Menu

Description

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

Usage

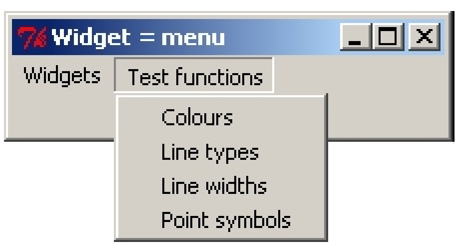
```
type=menu nitems=1 label font=""
```

Arguments

`nitems` ....number of items or submenus to include in the menu
`label`..................text to display as the menu label (required)
`font` ....font for menus - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order

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=4 label="Test functions"
     menuitem label="Colours" func=testCol
     menuitem label="Line types" func=testLty
     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="" function action="menuitem"
```

Arguments

`label`..................text to display as the menu item label (required)
`font` ....font for submenus - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
`function` ....R function to call when the menu item is clicked (required)
`action` ....value for `PBS.win$action` when selecting this menu item

Null

Description

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

Usage

```
type=null padx=0 pady=0
```

Arguments

`padx` ....space used to pad the label on the left and right
`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"
```

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="" function=""
  action="radio" mode="numeric" sticky="" padx=0 pady=0
```

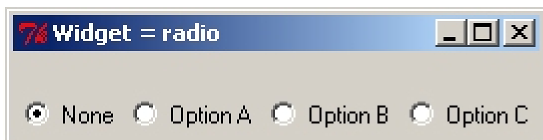
Arguments

`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 radio buttons - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
`function` ....R function to call when this radio button is selected
`action` ....value for `PBS.win$action` when this radio button is selected

`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
`pady` ....space used to pad the widget on the top and bottom

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

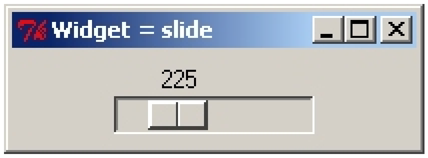
```
type=slide name from=0 to=100 value=NA showvalue=FALSE
  orientation="horizontal" function="" action="slide"
  sticky="" padx=0 pady=0
```

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`
`function` ....R function to call when the slide value is changed
`action` ....value for `PBS.win$action` when moving the slide bar
`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
`pady` ....space used to pad the widget on the top and bottom

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 function=""
  enter=FALSE action="slideplus" sticky="" padx=0 pady=0
```

Arguments

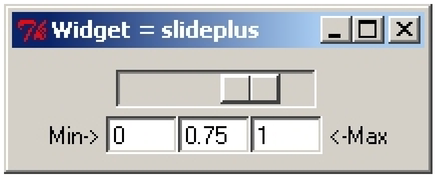
`name`....................name of the numeric R variable corresponding to this slide bar (required)
`from`....................minimum value of the variable
`to`.........................maximum value of the variable
`by`.........................minimum amount for changing the variable's value
`value`..................initial slide value, where the default is the specified `from` value
`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`...............value for `PBS.win$action` when changing the slide value, either by moving the slide bar or changing the value in the entry 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
`pady`....................space used to pad the widget on the top and bottom

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
```



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

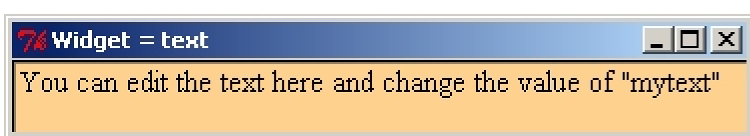
```
type=text name height=8 width=30 bg="white" mode="character"
  font="" value="" borderwidth=1 relief="sunken" edit=TRUE
```

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`
`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 text - 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`

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 the text here and change the 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

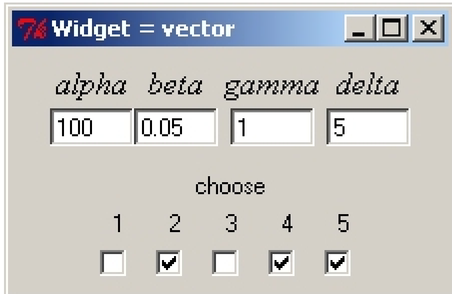
```
type=vector names length=0 labels="" values="" font=""
  vertical=FALSE function="" enter=TRUE action="vector"
  mode="numeric" width=6 sticky="" padx=0 pady=0
```

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`...............either one label of a vector of `length` labels used to label the vector in the display
`values`...............default values (either one value for all vector components or a vector of `length` values)
`font`....................font for vector labels - specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
`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`...............value for `PBS.win$action` when changing any component of this vector
`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
`pady`....................space used to pad the widget on the top and bottom

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"
```



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

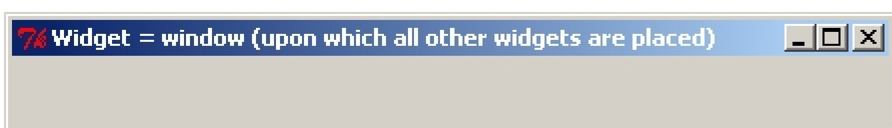
```
type=window name="window" title="" vertical=TRUE
```

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

Example

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



Appendix C. *PBS Modelling* functions and data

This appendix lists the functions currently made available by *PBS Modelling*, along with a list of their dependencies on hidden "dot" functions. For more information about the hidden functions, see the file `PBSMfunctions.txt` in the R library directory for `PBSmodelling`.

C.1. Objects in *PBS Modelling*

CCA.qbr.hl	Dataset: sampled counts of quillback rockfish (*Sebastes maliger*)
addArrows	Add arrows to a plot using relative (0:1) coordinates
addLabel	Add a label to a plot using relative (0:1) coordinates
addLegend	Add a legend to a plot using relative (0:1) coordinates
calcFib	Calculate Fibonacci numbers by several methods
calcGM	Calculate the geometric mean, allowing for zeros
clearAll	Remove all R objects from the global environment
clearWinVal	Remove all current widget variables
closeWin	Close GUI window(s)
compareLists	Compare two non-recursive lists
compileDescription	Convert and save a window description as a list
createVector	Create a GUI with a vector widget
createWin	Create a GUI window
drawBars	Draw a linear barplot on the current plot
expandGraph	Expand the plot area by adjusting margins
exportPBShistory	Export a saved history
findPat	Search a character vector to find multiple patterns
genMatrix	Generate test matrices for plotBubbles
getWinVal	Retreive widget values for use in R code
importPBShistory	Import a history list from a file
initPBShistory	Create structures of a new history widget
openFile	Open a file with the associated program
pad0	Pad numbers with leading zeroes
parseWinFile	Convert a window description file into a `list`
pause	Pause between graphics displays or other calculations
pickCol	Display an interactive colour selection palette
plotBubbles	Construct a bubble plot from a matrix
plotCsum	Plot cumulative sum of data
promptOpenFile	Display an “Open File” dialog
promptSaveFile	Display a “Save File” dialog
readList	Read a `list` from a file in *PBS Modelling* format
resetGraph	Reset `par` values for a plot
runExamples	Run GUI examples included with *PBS Modelling*
setWinVal	Update widget values
show0	Convert numbers into text with specified decimal places
showArgs	Display expected widget arguments
testCol	Display colour palette
testLty	Display line types
testLwd	Display line widths
testPch	Display plotting symbols
testWidgets	Display sample GUIs and their source code
unpackList	Unpack `list` elements into variables
view	Display first `n` rows of an object
writeList	Write a `list` to a file in *PBS Modelling* format

C.2. Function dependencies

This appendix documents function dependencies within *PBS Modelling*. 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. *PBS Modelling* enforces this standard through the `NAMESPACE` protocol discussed above in section A.3.

<u>.addslashes</u>

<u>.autoConvertMode</u>

<u>.buildgrid</u>
.createTkFont
.createWidget

<u>.catError</u>

<u>.convertMatrixListToDataFrame</u>
.getMatrixListSize
.setMatrixElement

<u>.convertMatrixListToMatrix</u>
.getMatrixListSize
.setMatrixElement

<u>.convertMode</u>
.convertMode

<u>.convertPararmStrToList</u>
.catError
.trimWhiteSpace

<u>.convertPararmStrToVector</u>
.catError
.trimWhiteSpace

<u>.convertVecToArray</u>
.getArrayPts
.mapArrayToVec

<u>.createTkFont</u>
.convertPararmStrToVector

<u>.createWidget</u>
.createWidget.button
.createWidget.check
.createWidget.data
.createWidget.entry
.createWidget.grid
.createWidget.history
.createWidget.label
.createWidget.matrix
.createWidget.null
.createWidget.radio
.createWidget.slide
.createWidget.slideplus
.createWidget.text
.createWidget.vector

<u>.createWidget.button</u>
.createTkFont
.extractData

<u>.createWidget.check</u>
.createTkFont
.extractData

<u>.createWidget.data</u>
.createWidget.grid
.stopWidget

<u>.createWidget.entry</u>
.createWidget.grid
.extractData

<u>.createWidget.grid</u>
.buildgrid

<u>.createWidget.history</u>
.createWidget.grid
initPBShistory

<u>.createWidget.label</u>
.createTkFont

<u>.createWidget.matrix</u>
.createWidget.grid
.stopWidget

.createWidget.null

.createWidget.radio
.createTkFont
.extractData

.createWidget.slide
.extractData

.createWidget.slideplus
.extractData

.createWidget.text
.createTkFont

.createWidget.vector
.createWidget.grid
.stopWidget

.extractData
.getPBS.win

.extractFuns

.extractVar
.convertMatrixListToDataFrame
.convertMatrixListToMatrix
.matrixHelp
.PBSdimnameHelper

.fibC

.fibCall

.fibClosedForm

.fibR

.getArrayPts

.getMatrixListSize
.getMatrixListSize

.getParamFromStr
.catError
.convertPararmStrToList
.isReallyNull
.searchCollection
.stripSlashes
.stripSlashesVec
.trimWhiteSpace

.getParamOrder

.getPBS.win
.extractVar

.getReadListParamOrder

.hash
.isReallyNull

.inCollection

.initPBSoptions

.isReallyNull

.mapArrayToVec

.matrixHelp
.matrixHelp

.parsegrid
.parsegrid

.parsemenu
.parsemenu

.PBSdimnameHelper

.readList.P
.catError
.readList.P.convertData
.stripComments
.trimWhiteSpace

.readList.P.convertData
.autoConvertMode
.catError
.convertMode
.convertPararmStrToVector
.convertVecToArray
.getParamFromStr
.getReadListParamOrder

.searchCollection

.setMatrixElement
.setMatrixElement

importPBShistory
.updatePBShistory
promptOpenFile
readList

initPBShistory

jumpPBShistory
.updatePBShistory
getWinVal
setWinVal

openFile
.initPBSoptions
.isReallyNull
openFile

pad0

parseWinFile
.getParamFromStr
.parsegrid
.parsemenu
.stripComments
.trimWhiteSpace

pause

pickCol

plotBubbles

plotCsum
addLabel
resetGraph

promptOpenFile
.trimWhiteSpace

promptSaveFile
promptOpenFile

readList
.readList.P

resetGraph

rmPBShistory
.updatePBShistory
setWinVal

runExamples
closeWin
createWin
getWinVal
setWinVal

setWinVal
.isReallyNull
.setWinValHelper

show0

showArgs
.getParamOrder

testCol

testLty

testLwd
resetGraph

testPch
resetGraph

testWidgets
closeWin
createWin
getWinVal
setWinVal

unpackList

view

writeList
.writeList.P

`addArrows` *Add Arrows to a Plot Using Relative (0:1) Coordinates*

Description

Calls `arrows` function using relative (0:1) coordinates.

Usage

```
addArrows(x1, y1, x2, y2, ...)
```

Arguments

`x1`	x-coordinate at base of arrow
`y1`	y-coordinate at base of arrow
`x2`	x-coordinate at tip of arrow
`y2`	y-coordinate at tip of arrow
`...`	additional paramaters for arrows

Details

Lines will be drawn from `(x1[i],y1[i])` to `(x2[i],y2[i])`

See Also

`addLabel`

`addLegend`

Examples

```
tt=seq(from=-5,to=5,by=0.01)
plot(sin(tt), cos(tt)*(1-sin(tt)), type="l")
addArrows(0.2,0.5,0.8,0.5)
addArrows(0.8,0.95,0.95,0.55, col="#FF0066")
```

`addLabel` *Add Label to a Plot Using Relative (0:1) Coordinates*

Description

Places a label in a plot using relative (0:1) coordinates

Usage

```
addLabel(x, y, txt, ...)
```

Arguments

`x`	x-axis coordinate in the range (0,1); can step outside
`y`	y-axis coordinate in the range (0,1); can step outside
`txt`	desired label at (x,y)
`...`	additional arguments passed to text function

See Also

`addArrows`

`addLegend`

Examples

```
resetGraph()
addLabel(0.75,seq(from=0.9,to=0.1,by=-0.10),c('a','b','c'), col="#0033AA")
```

`addLegend` *Add Legend to a Plot Using Relative (0:1) Coordinates*

Description

Places a legend in a plot using relative (0:1) coordinates.

Usage

```
addLegend(x, y, ...)
```

Arguments

`x`	x-axis coordinate in the range (0,1); can step outside
`y`	y-axis coordinate in the range (0,1); can step outside
`...`	arguments used by legend, such as “lines”, “text”, or “rectangle”

See Also

`addArrows`

`addLabel`

`calcFib` *Several Methods to Calculate Fibonacci Numbers*

Description

Computes Fibonacci numbers using four different methods: 1) iteratively using R code, 2) via the closed function in R code, 3) iteratively in C using the `.C` function, and 4) iteratively in C using the `.Call` function.

Usage

```
calcFib(n, len=1, method="C")
```

Arguments

`n`	nth fibonacci number to calculate
`len`	A vector of length `len` showing previous fibonacci numbers
`method`	Select method to use: `C`, `Call`, `R`, `closed`

Value

Vector of the last `len` Fibonacci numbers calculated.

`calcGM` *Calculate the Geometric Mean*

Description

Calculate the geometric mean of a numeric vector, possibly excluding zeros and/or adding an offset to compensate for zero values.

Usage

```
calcGM(x, offset = 0, exzero = TRUE)
```

Arguments

`x`	vector of numbers
`offset`	value to add to all components, including zeroes
`exzero`	if T, exclude zeroes (but still add the offset)

Value

geometric mean of the modified vector `x + offset`

Note

`NA` values are automatically removed from `x`

Examples

```
calcGM(c(0,1,100))
calcGM(c(0,1,100),offset=0.01,exzero=FALSE)
```

`CCA.qbr.hl` *Dataset of Sampled Counts of Quillback Rockfish (Sebastes maliger)*

Description

Count of sampled fish-at-age for quillback rockfish (*Sebastes maliger*) in Johnstone Strait, British Columbia, from 1984 to 2004.

Usage

```
data(CCA.qbr.hl)
```

Format

A matrix with 70 rows (ages) and 14 columns (years). Attributes "syrs" and "cyrs" specify years of survey and commercial data, respectively.

`[,c(3:5,9,13,14)]`	Counts-at-age from research survey samples
`[,c(1,2,6:8,10:12)]`	Counts-at-age from commercial fishery samples

All elements represent sampled counts-at-age in year. Zero-value entries indicate no observations.

Details

Handline surveys for rockfish have been conducted in Johnstone Strait (British Columbia) and adjacent waterways (126°37'W to 126°53'W, 50°32'N to 50°39'N) since 1986. Yamanaka and Richards (1993) describe surveys conducted in 1986, 1987, 1988, and 1992. In 2001, the Rockfish Selective Fishery Study (Berry 2001) targeted quillback rockfish *Sebastes maliger* for experiments on improving survival after capture by hook and line gear. The resulting data subsequently have been incorporated into the survey data series. The most recent survey in 2004 essentially repeated the 1992 survey design. Fish samples from surveys have been supplemented by commercial handline fishery samples taken from a larger region (126°35'W to 127°39'W, 50°32'N to 50°59'N) in the years 1984-1985, 1989-1991, 1993, 1996, and 2000 (Schnute and Haigh 2006).

Note

Years 1994, 1997-1999, and 2002-2003 do not have data.

Source

Fisheries and Oceans Canada - GFBio database:
`http://www-sci.pac.dfo-mpo.gc.ca/sa-mfpd/statsamp/StatSamp_GFBio.htm`

References

Berry, M.D. 2001. Area 12 (Inside) Rockfish Selective Fishery Study. Science Council of British Columbia, Project Number FS00- 05.

Schnute, J.T., and Haigh, R. 2006. Compositional analysis of catch curve data with an application to *Sebastes maliger*. ICES Journal of Marine Science (in revision).

Yamanaka, K.L. and Richards, L.J. 1993. 1992 Research catch and effort data on nearshore reef-fishes in British Columbia Statistical Area 12. Canadian Manuscript Report of Fisheries and Aquatic Sciences 2184, 77 pp.

Examples

```
# Plot age proportions (blue bubbles = survey data, red = commercial)
data("CCA.qbr.hl", package="PBSmodelling")
```

```
z <- CCA.qbr.hl; cyr <- attributes(z)$cyrs;
z <- apply(z,2,function(x){x/sum(x)}); z[,cyr] <- -z[,cyr];
x <- as.numeric(dimnames(z)[[2]]); xlim <- range(x) + c(-.5,.5);
y <- as.numeric(dimnames(z)[[1]]); ylim <- range(y) + c(-1,1);
plotBubbles(z,xval=x,yval=y,powr=.5,size=0.15,lwd=1,clrs=c("blue","red"),
            xlim=xlim,ylim=ylim,xlab="Year",ylab="Age",cex.lab=1.5)
```

`clearAll` *Remove All R Objects*

Description

Generic function to clear .RData in R

Usage

```
clearAll()
```

`clearWinVal` *Remove All Widget Variables*

Description

Removes all global variables that share a name in common with any widget variable name as defined in PBS.win$vars. Use this with caution.

Usage

```
clearWinVal()
```

See Also

`getWinVal`

`closeWin` *Close GUI windows*

Description

The closeWin function closes (destroys) one or more windows made with createWin.

Usage

```
closeWin(name=names(.PBS.tclHash))
```

Arguments

`name`	a vector of window names to close, as defined in the window description file's WINDOW widget.

See Also

`createWin`

`compareLists` *Compares two non-recursive lists*

Description

Displays any differences between two non-recursive lists.

Usage

```
compareLists(a,b,verbose=FALSE)
```

Arguments

`a`	list a
`b`	list b
`verbose`	be verbose

`compileDescription` *Converts and Saves a Window Description Into a List*

Description

compileDescription converts a Window Description File into an equivalent Window Description List. The list is complete - meaning all default values have been added to the list.

Usage

```
compileDescription(descFile, outFile)
```

Arguments

`descFile`	filename of markup file.
`outFile`	filename indicating where to save outputed R source code.

Details

The Window Description File is converted into a list, which is then converted into R code and saved to the outfile.

See Also

`parseWinFile`

`createWin`

`createVector` *Create a GUI with a Vector Widget*

Description

createVector creates a basic window containing a vector and a submit button. This provides a quick way to create a window without the need for a window description file.

Usage

```
createVector(vec, vectorLabels=NULL, func="", windowname="vectorwindow")
```

Arguments

`vec` a vector of strings representing widget variables. If it is named, the names are used as the variable names, and the values are used as the default value of the widget

`vectorLabels` option vector of strings to be used as labels above each widget.

`func` string value of function name to be called when new data is entered or when "GO" is pressed.

`windowname` window name required if multiple vector windows are created.

See Also

`createWin`

Examples

```
#user defined function which is called on new data
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="p");

  invisible(NULL);
}

#create the vector window
createVector(c(m=2, n=3, phi=0, k=1000),
             vectorLabels=c("x cycles","y cycles", "y phase", "points"),
             func="drawLiss"
             )
```

`createWin` *Create a GUI Window*

Description

The createWin function takes a window markup file, and creates a window based on the markup file.

Usage

```
createWin(fname, astext=FALSE)
```

Arguments

`fname`	filename of markup file or list returned from parseWinFile.
`astext`	if true, fname is intrupted as a vector of strings. with each element representing a line of the source of a window description file

Details

The markup file contains a single widget per line. Widgets can span multiple lines by including a backslash ('\') as the last character of the line, which then ignores the newline.

For more details about widget types, and the markup file see the pdf located in the installation directory.

It is possible to use a Window Description List produced by compileDescription rather than a filename for the fname argument.

Another alternative is to set astext=TRUE and pass in a vector of characters for fname. This vector of characters represents the file contents. Each element of the vector is equivalent to a new line in the window description file.

Value

PBS.win contains window information such as: present values, present variable names, present variable values, and triggered action values. This information is encapsulated in a list which is set as the global PBS.win variable, which is also returned.

PBS.win

`vars`	Current widget values
`funs`	Functions required by Window
`names`	Variable names: names(PBS.win$vars)
`action`	Action that triggered a function call

See Also

`parseWinFile`

`getWinVal`

`setWinVal`

`closeWin`

`compileDescription`

`createVector`

`initPBShistory` for an example of using astext=TRUE

Examples

```
#see file testWidgets\LissWin.txt in PBSmodelling package directory
#
# window title="Lissajous Curve"
# grid 1 2
#   label text=Pars: font=bold
#   vector length=4 names="m n phi k" \
#     labels="'x cycles' 'y cycles' 'y phase' points" \
#     values="2 3 0 1000" vertical=T
# grid 1 2
#   label text=History: font=bold
#   history
# grid 1 2
#   grid 2 1
#     radio name=ptype text=lines  value="l" mode=character
#     radio name=ptype text=points value="p" mode=character
# button text=Plot func=drawLiss font=bold
#

# Calculate and draw the Lissajous figure
drawLiss <- function()
{
  getWinVal(scope="L");
  ti <- 2*pi*(0:k)/k;
  x <- sin(2*pi*m*ti);
  y <- sin(2*pi*(n*ti+phi));

  plot(x,y,type=ptype);
  invisible(NULL);
}
## Not run:
require(PBSmodelling);
createWin(system.file("testWidgets/LissWin.txt",package="PBSmodelling"))
## End(Not run)
```

`drawBars` *Draw a Linear Barplot on the Current Plot*

Description

Draws a linear barplot on the current graph.

Usage

```
drawBars(x, y, width, base = 0, ...)
```

Arguments

`x`	x-coordinates
`y`	y-coordinates
`width`	bar width, computed if missing
`base`	y-value of the base of each bar
`...`	further graphical parameters (see 'par') may also be supplied as arguments

Examples

```
plot(0:10,0:10,type="n")
drawBars(x=1:9,y=9:1,col="deepskyblue4",lwd=3)
```

`expandGraph` *Expand Plot Area by Adjusting Margins*

Description

Tries to maximize the area of multiple plots by minimizing margins.

Usage

```
expandGraph(mar=c(4,3,1.2,0.5), mgp=c(1.6,.5,0),...)
```

Arguments

`mar`	numerical vector of the form 'c(bottom, left, top, right)' specifying the margins of the plot
`mgp`	numerical vector of the form 'c(axis title, axis labels, axis line)' specifying the margins for axis title, axis labels, and axis line
`...`	Additional graphical parameters to be passed to par()

See Also

`resetGraph`

`par`

Examples

```
resetGraph()
expandGraph(mfrow=c(2,1))

tt=seq(from=-10, to=10, by=0.05)

plot(tt,sin(tt), xlab="this is the x label",  ylab="this is the y label",
     main="main title", sub="sometimes there is a \"sub\" title")
plot(cos(tt),sin(tt*2), xlab="cos(t)", ylab="sin(2 t)", main="main title",
     sub="sometimes there is a \"sub\" title")
```

`exportPBShistory` *Export Saved History*

Description

Exports the current history List.

Usage

```
exportPBShistory(hisname="", fname="")
```

Arguments

`hisname`	Name of the history list to export. If it is set to `""`, the value from `PBS.win$action[1]` will be used instead.
`fname`	Where to save the history to. If it is set to `""`, a save file window will be displayed.

See Also

`importPBShistory`

`initPBShistory`

`promptSaveFile`

`findPat` *Search a Vector With Multiple Patterns*

Description

Searches all patterns in pat from vec, and returns the matched elements in vec.

Usage

```
findPat(pat, vec)
```

Arguments

`pat`	character vector of patterns to match in vec
`vec`	character vector where matches are sought

Value

A character vector of all matched strings.

Examples

```
#find all strings with a vowel, or that start with a number
findPat(c("[aeoiy]", "^[0-9]"), c("hello", "WRLD", "11b"))
```

`genMatrix` *Generate Test Matrices for plotBubbles*

Description

Generates a test matrix of random numbers (mean = mu and standard deviation = sigma), primarily for plotBubbles.

Usage

```
genMatrix(m,n,mu=0,sigma=1)
```

Arguments

`m`	number of rows
`n`	number of columns
`mu`	mean of normal distribution
`sigma`	standard deviation of normal distribution

Value

An m by n matrix with normally distributed random values.

See Also

`plotBubbles`

Examples

```
plotBubbles(genMatrix(20,6))
```

`getWinVal` *Retreive Widget Values*

Description

An optional scope argument allows the function to create local or global variables based on the list that is returned.

Usage

```
getWinVal(v=NULL, scope="", asvector=FALSE, windowname="")
```

Arguments

`v`	vector of variable names to be retrieved. If NULL, it is set to every widget.
`scope`	If `""`, do not set any variables. If `"L"` create variables local to the parent frame that called the function. If `"G"` create global variables.
`asvector`	return a vector instead of a list. WARNING: if a widget variable is a true vector or matrix, this will not work.
`windowname`	Which window to select values from. If `""` is given, it will use the most recently active window determined from `PBS.win$windowname`.

Value

A list (or vector) with named components, based on the variable name as the key, and the value of the associated widget as the value.

See Also

`parseWinFile`

`setWinVal`

`clearWinVal`

`importPBShistory` *Import a history list from a file*

Description

Imports a history list from a file, and places it as the history list identified by hisname.

Usage

```
importPBShistory(hisname="", fname="")
```

Arguments

`hisname`	The imported list is placed into the history list identified by hisname. If it is set to `""`, the value from `PBS.win$action[1]` will be used instead.
`fname`	Which file to import. If it is set to `""`, an open file window will be displayed.

See Also

`exportPBShistory`

`initPBShistory`

`promptOpenFile`

`initPBShistory` *Customized History Widget Functions*

Description

The functions: initPBShistory, rmPBShistory, addPBShistory, forwPBShistory, backPBShistory, jumpPBShistory are made available to those who would like to use history, without using the history widget. In other words, these functions allow users to create a custom history widget.

Usage

```
initPBShistory(hisname, indexname=NULL, sizename=NULL, overwrite=TRUE)
rmPBShistory(hisname="", index="")
addPBShistory(hisname="")
forwPBShistory(hisname="")
backPBShistory(hisname="")
jumpPBShistory(hisname="", index="")
clearPBShistory(hisname="")
```

Arguments

- `hisname` — The name of the history "list" to manipulate. If it is omitted, the function will use the value of `PBS.win$actions[1]` as the history name. This allows functions to be called directly from the window description file. With the exception of `initPBShistory` which must be called before `createWin()`.
- `indexname` — The name of the index entry widget in the window description file. If it is `NULL`, then the current index feature will be disabled.
- `sizename` — The name of the current size entry widget. If it is `NULL`, then the current size feature will be disabled.
- `index` — Index to the history item. if it is left as `""`, then value will be extracted from the widget identified by `indexname`
- `overwrite` — If set to true, history (matching `hisname`) will be cleared. Otherwise, the two will be merged.

Details

PBS Modelling includes a pre-built history widget designed to collect interesting choices of GUI variables so that they can be redisplayed later, rather like a slide show.

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

To create a customized history, each button must be described separately in the window description file rather than making reference to the history widget.

The history "List" must be initialized before any other functions may be called. The use of a unique history name (`hisname`) is used to associate a unique history session with the supporting functions.

The `indexname` and `sizename` arguments correspond to the given names of entry widgets in the window description file which will be used to display the current index, and total size of the list. the `indexname` entry widget can also be used by `jumpPBShistory` to retrieve an index to jump to.

See Also

`importPBShistory`

`exportPBShistory`

Examples

```
#Example of creating a custom history widget that saves values
#whenever the "plot" button is pressed, and updates the plot when
#back or next is pushed. A custom history is needed to achieve this
#functionality since the normal history widget does not update plots
require(PBSmodelling)
```

```
#create a window Description to be used with createWin using astext=TRUE
#PS: watch out for escaping special characters which
#    must be done twice (first for R, then PBSmodelling)

winDesc <- '
window title="Custom History"
vector names="a b k" labels="a b points" font="bold" values="1 1 1000" function=myPlot
grid 1 3
  button function=myHistoryBack text="<- Back"
  button function=myPlot text="Plot"
  button function=myHistoryForw text="Next ->"

grid 2 2
  label "Index"
  entry name="myHistoryIndex" width=5
  label "Size"
  entry name="myHistorySize" width=5
'
#convert text into vector with each line represented as a new element
winDesc <- strsplit(winDesc, "\n")[[1]]

#custome functions required to update plots after restoring history values
myHistoryBack <- function() {
  backPBShistory("myHistory")
  myPlot(saveVal=FALSE) #show the plot with saved values
}
myHistoryForw <- function() {
  forwPBShistory("myHistory")
  myPlot(saveVal=FALSE) #show the plot with saved values
}

myPlot <- function(saveVal=TRUE) {

  #save all data whenever plot is called (directly)
  if (saveVal)
    addPBShistory("myHistory")

  getWinVal(scope="L")
  tt <- 2*pi*(0:k)/k;
  x <- (1+sin(a*tt));  y <- cos(tt)*(1+sin(b*tt));
  plot(x, y)
}

initPBShistory("myHistory", "myHistoryIndex", "myHistorySize")
createWin(winDesc, astext=TRUE)
```

`openFile` *Open Files With Associated Program*

Description

openFile attempts to open a file, based off the command set in the `PBS.options$openfile` list. If `PBS.options$openfile[[extension]]` is set openFile will replace all occurrences of `"%f"` with the absolute path of the filename, and then execute the command. Otherwise, if no command is set, `shell.exec()` will be used.

Usage

```
openFile(fname)
```

Arguments

`fname`	filename to be opened.

Examples

```
## Not run:
#setup firefox to open .html files
PBS.options$openfile$html='"c:/Program Files/Mozilla Firefox/firefox.exe" %f'
openFile("foo.html")
## End(Not run)
```

`pad0` *Pads Numbers with leading zeroes*

Description

Takes numbers, converts them to integers then text, and pads them with leading zeroes.

Usage

```
pad0(x, n, f = 0)
```

Arguments

`x`	Vector of numbers
`n`	Length of padded integer
`f`	Factor of 10 to expand x by

Value

A character vector representing x with leading zeros.

parseWinFile *Convert Window Description File into a List*

Description

Parses a Window Markup file into the list format expected by `createWin()`

Usage

```
parseWinFile(fname, astext=FALSE)
```

Arguments

`fname`	file name of window markup file.
`astext`	if true, fname is interpreted as a vector of strings. with each element representing a line of the source of a window description file

Value

A list representing a parsed window description file that can be directly passed to createWin.

Note

All widgets are encapsulated into a 1 column by N row grid.

See Also

`createWin`

`compileDescription`

Examples

```
## Not run:
x<-parseWinFile(system.file("examples/LissFigWin.txt",package="PBSmodelling"))
createWin(x)
## End(Not run)
```

`pause` *Pause*

Description

Pause, typically between graphics displays

Usage

```
pause(s = "Press <Enter> to continue")
```

Arguments

`s`	prompt text

`PBSmodelling` *PBS Modelling*

Description

PBS Modelling provides software to facilitate the design, testing, and operation of computer models. It focuses particularly on tools that make it easy to construct and edit a customized graphical user interface (GUI). Although it depends heavily on the R interface to the Tcl/Tk package, a user does not need to know Tcl/Tk.

The package contains examples that illustrate models built with other R packages, including PBS Mapping, odesolve, and BRugs. It also serves as a convenient prototype for building new R packages, along with instructions and batch files to facilitate that process.

The root library directory of PBSmodelling includes a complete user guide `PBSmodelling-UG.pdf`. To use this package effectively, please consult the guide.

PBS Modelling comes packaged with several examples which can be accessed with the `runExamples()` function.

`pickCol` *Display Interactive Colour Selection Palette*

Description

Display an interactive colour chooser. If returnValue is true, then the equivalent hex colour value is returned, otherwise if returnValue is faluse, an intermediate GUI is used to display the hex value and no value is returned to R.

Usage

```
pickCol(returnValue=TRUE)
```

Arguments

`returnValue` If true only display the colour chooser and return the hex value. Otherwise use an intermediate GUI to display the hex value

Value

A hexidecimal colour value.

See Also

`testCol`

`plotBubbles` *Construct a Bubble Plot from a Matrix*

Description

Constructs a bubble plot for a matrix z.

Usage

```
plotBubbles(z, xval = FALSE, yval = FALSE, rpro = FALSE,
cpro = FALSE, rres = FALSE, cres = FALSE, powr = 1,
clrs = c("black", "red"), size = 0.2, lwd = 2, debug = FALSE, ...)
```

Arguments

`z`	input matrix
`xval`	x-values for the columns of z. if xval=TRUE, first row contains x-values for the columns
`yval`	y-values for the rows of z. if yval=TRUE, first column contains y-values for the rows
`rpro`	if rpro=TRUE, convert rows to proportions
`cpro`	if cpro=TRUE, convert columns to proportions
`rres`	if rres=TRUE, use row residuals (subtract row means)
`cres`	if cres=TRUE, use column residuals (subtract column means)
`powr`	power transform. radii proportional to z^{powr}. powr=0.5 gives bubble areas proportional to z
`clrs`	colours used for positive and negative values
`size`	size (inches) of the largest bubble
`lwd`	line width for drawing circles
`debug`	display debug information if true
`...`	additional arguments for symbols function

See Also

`genMatrix`

Examples

```
plotBubbles(genMatrix(20,6))
```

`plotCsum` *Plots Cumulative Frequency of Data*

Description

Plots cumulative frequency of data

Usage

```
plotCsum(x, add = FALSE, ylim = c(0, 1), xlab = "Measure",
ylab = "Cumulative Proportion", ...)
```

Arguments

`x`	vector of values
`add`	if TRUE, add cumulative frequency curve to current plot
`ylim`	limits for y-axis
`xlab`	label for x-axis
`ylab`	label for y-axis
`...`	additional arguments for plot.

Examples

```
x <- rgamma(n=1000,shape=2)
plotCsum(x)
```

`promptOpenFile` *Display Open File Dialog*

Description

Displays the default open file prompt provided by the Operating System.

Usage

```
promptOpenFile(initialfile="", filetype=list(c("*", "All Files")), open=TRUE)
```

Arguments

`initialfile`	file name of text file containing the list.
`filetype`	a list of character vectors indicating what filetypes to look for. Each vector is of length one or two, and specifies the file extension, or `"*"` (for all filetypes). The second element is an optional name for the file type describing the file type.
`open`	If True display Open file prompt, if False display Save File prompt.

Value

The filename and path of file selected by user.

See Also

`promptSaveFile`

Examples

```
## Not run:

#open a filename, and return it line by line in a vector
scan(promptOpenFile(),what=character(),sep="\n")

#illustrates how to set filetype.
promptOpenFile("intial_file.txt", filetype=list(c(".txt", "text files"),
               c(".r", "R files"), c("*", "All Files")))
## End(Not run)
```

`promptSaveFile` *Display Save File Dialog*

Description

Displays the default save file prompt provided by the Operating System.

Usage

```
promptSaveFile(initialfile="", filetype=list(c("*", "All Files")), save=TRUE)
```

Arguments

`initialfile`	file name of text file containing the list.
`filetype`	a list of character vectors indicating what filetypes to look for. Each vector is of length one or two, and specifies the file extension, or `"*"` (for all filetypes). The second element is an optional name for the file type describing the file type.
`save`	If True display Save file prompt, if False display Open File prompt.

Value

The filename and path of file selected by user.

See Also

`promptOpenFile`

Examples

```
## Not run:

#illustrates how to set filetype.
promptSaveFile("intial_file.txt", filetype=list(c(".txt", "text files"),
               c(".r", "R files"), c("*", "All Files")))
## End(Not run)
```

`readList` *Read a List From a File*

Description

Reads in a list which was saved to a file by writeList. It can support a native R list, or PBSmodelling P format. It is detected automatically.

For information about `"P"` format, see writeList.

Usage

```
readList(fname)
```

Arguments

`fname` file name of text file containing the list.

See Also

`writeList`

`unpackList`

`resetGraph` *Reset plot par Values*

Description

Reset the plot par() to default values to ensure the plot takes up the whole canvas.

Usage

```
resetGraph()
```

See Also

`resetGraph`

`runExamples` *Display PBS Modelling Examples*

Description

Displays an interactive demo GUI to display PBS Modelling examples.

The example source files can be found in the PBSmodelling/examples directory located in the R library.

Usage

```
runExamples()
```

Details

Some examples make use of external packages which must be installed in order to work correctly.

LinReg, MarkRec, and CCA Require the Brugs package.

FishRes requires the odesolve package.

FishTows requires the PBSmapping package.

`setWinVal` *Update Widget Values*

Description

setWinVal updates a widget with a new value. The vars argument expects a list or vector with named elements. Every element name corresponds to the widget name which will be updated with the supplied element value.

The vector and matrix widgets can be updated in several ways. If more than one name is given for the names argument then each element is treated as if it simply an entry widget. If however, a single name is given, and the value returned by getWinVal is a vector or matrix, the whole widget can be updated by passing an appropriately sized vector or widget. Alternatively each element can be updated by appending the index in square braces to the end of the name.

Usage

```
setWinVal(vars, windowname)
```

Arguments

`vars` a list or vector with named components.

`windowname` Which window to select values from. If `""` is given, it will use the most recently active window determined from `PBS.win$windowname`.

Details

The data widget can also be updated in the same fashion as the matrix; however, when updating a single element, a "d" must be added after the brackets. This is due to the internal coding of PBS Modeling. Example: `"foo[1,1]d"`

See Also

`getWinVal`

`createWin`

Examples

```
winDesc <- c(
            "vector length=3 name=vec",
            "matrix nrow=2 ncol=2 name=mat",
            "slideplus name=foo"
            );
createWin(winDesc, astext=TRUE)
setWinVal(list(vec=1:3, "mat[1,1]"=123, foo.max=1.5, foo.min=0.25, foo=0.7))
```

`show0` *Convert Numbers Into Text With Specified Decimal Places*

Description

Return character representation of number with specified decimal places.

Usage

```
show0(x, n, add2int = FALSE)
```

Arguments

`x` Number as scalar or vector

`n` Number of decimal places to show, include zeroes

`add2int` If TRUE, add zeroes on the end of integers

Examples

```
frame()

#do not show decimals on integers
addLabel(0.25,0.75,show0(15.2,4))
addLabel(0.25,0.7,show0(15.1,4))
addLabel(0.25,0.65,show0(15,4))

#show decimals on integers
addLabel(0.25,0.55,show0(15.2,4,TRUE))
addLabel(0.25,0.5,show0(15.1,4,TRUE))
addLabel(0.25,0.45,show0(15,4,TRUE))
```

`showArgs` *Display Expected Widget Arguments*

Description

Displays the order and default values of widget arguments. The list can be shortened by specifying a single widget name. Otherwise all widgets are displayed.

Usage

```
showArgs(widget="")
```

Arguments

`widget` Only displays information about this one widget

`testCol` *Display Colour Palette*

Description

Provides a testing bed for displaying colours on a graph. Colours can be specified in any of 3 different ways: 1) by a color name, 2) by a hexidecimal color code created by `rgb()` or 3) by an index into the color palette.

Usage

```
testCol(cnam=colors()[1:20])
```

Arguments

`cnam`	vector of colour names to display

See Also

`pickCol`

Examples

```
testCol(c("sky","fire","sea","wood"))

testCol(c("plum","tomato","olive","peach","honeydew"))

testCol(rainbow(63))

#display all colours set in the colour palette
testCol(1:length(palette()))

#they can even be mixed
testCol(c("#9e7ad3", "purple", 6))
```

`testLty` *Display Line Types*

Description

Displays line types available

Usage

```
testLty(newframe = TRUE)
```

Arguments

`newframe`	if true, create a new blank frame, otherwise overlay current frame

`testLwd` *Display Line Widths*

Description

User can specify particular ranges for lwd. Colours can also be specified and are internally repeated as necessary.

Usage

```
testLwd(lwd=1:20, col=c("black","blue"), newframe=TRUE)
```

Arguments

`lwd`	line widths to test
`col`	colours to test
`newframe`	if true, create a new blank frame, otherwise overlay current frame

Examples

```
testLwd(3:15,col=c("salmon","aquamarine","gold"))
```

`testPch` *Display Plotting Symbols*

Description

Allows the user to specify particular ranges (increasing continuous integer) for pch.

Usage

```
testPch(pch=1:100, ncol=10, grid=TRUE, newframe=TRUE, bs=FALSE)
```

Arguments

`pch`	symboles to view
`ncol`	number of columns (can only be 2, 5, or 10). Most sensibly this is set to 10.
`grid`	if T, grid is plotted for visual aid
`newframe`	if T,reset the graph, otherwise overlay on top of the current graph
`bs`	if T, shows backslash characters used in text statements. (e.g. `30\272C` = 30°C)

Examples

```
testPch(123:255)
testPch(1:25,ncol=5)
testPch(41:277,bs=TRUE)
```

`testWidgets` *Displays Sample GUIs and Source Code*

Description

Displays an interactive demo GUI to provide several sample GUIs along with window description source code. It is possible to modify the sample source code in the provided text box which can then be recreated with the button below.

The window description source files can be found in the PBSmodelling/testWidgets directory located in the R library.

Usage

```
testWidgets()
```

Details

The following are the widgets and default values supported by PBS Modelling. See Appendix B for a detailed description.

```
button text="Calculate" font="" width=0 function="" action="button"
sticky="" padx=0 pady=0

check name checked=FALSE text="" font="" function="" action="check"
sticky="" padx=0 pady=0

data nrow ncol names modes="numeric" rowlabels="" collabels=""
rownames="X" colnames="Y" font="" values="" byrow=TRUE function=""
enter=TRUE action="data" width=6 sticky="" padx=0 pady=0

entry name value="" width=20 label="" font="" function="" enter=TRUE
action="entry" mode="numeric" sticky="" padx=0 pady=0

grid nrow=1 ncol=1 toptitle="" sidetitle="" topfont="" sidefont=""
byrow=TRUE borderwidth=1 relief="flat" sticky="" padx=0 pady=0

history name="default" archive=TRUE sticky="" padx=0 pady=0

label text="" font="" sticky="" padx=0 pady=0
```

```
matrix nrow ncol names rowlabels="" collabels="" rownames="" colnames=""
font="" values="" byrow=TRUE function="" enter=TRUE action="matrix"
mode="numeric" width=6 sticky="" padx=0 pady=0

menu nitems=1 label font=""

menuitem label font="" function action="menuitem"

null padx=0 pady=0

radio name value text="" font="" function="" action="radio"
mode="numeric" sticky="" padx=0 pady=0

slide name from=0 to=100 value=NA showvalue=FALSE orientation="horizontal"
function="" action="slide" sticky="" padx=0 pady=0

slideplus name from=0 to=1 by=0.01 value=NA function="" enter=FALSE
action="slideplus" sticky="" padx=0 pady=0

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

vector names length=0 labels="" values="" font="" vertical=FALSE
function="" enter=TRUE action="vector" mode="numeric" width=6 sticky=""
padx=0 pady=0 window name="window" title="" vertical=TRUE
```

See Also

createWin, showArgs

---

unpackList *Unpack List Elements Into Variables*

Description

Function to make local or global variables (depending on the scope) from the named components of a list.

Usage

```
unpackList(x, scope="L")
```

Arguments

`x`	list to unpack.
`scope`	If "L" create variables local to the parent frame that called the function. If "G" create global variables.

Value

A character vector of unpacked variable names.

See Also

`readList`

Examples

```
x<-list(a=21,b=23)
unpackList(x)
print(a)
```

`view` *Display First n Rows of an Object*

Description

Views first n rows of a data.frame or matrix or first n elements of a vector or list. All other objects are simply reflected.

Usage

```
view(obj, n = 5)
```

Arguments

`obj`	Object to view
`n`	First n elements of the obj to view

`writeList` *Write a List to a File*

Description

Writes an ASCII text representation in either "D" format, or "P" format. The "D" format makes use of dput and dget and produces an R representation of the list. The "P" format attempts to represents a simple list in an easy to read format.

Usage

```
writeList(x, fname, format="D", comments="")
```

Arguments

`fname`	file name of text file containing the list
`x`	list to write out
`format`	format of file to create. "P" or "D"
`comments`	vector of character strings to be used as initial comments in the file

Details

The "D" format is equivalent to using dput and dget, which supports all R objects.

The "P" format only supports named lists of vectors, matrices, and data-frames. Nested lists are not supported. In the simplest form, the "P" format consists of a named list element prefixed with a dollar sign ($) on a single line with data on the following line(s). All following data will be until a new variable name is found (denoted by $), or the end of a file is reached.

If multiple lines of data are used, then the data is treated as a matrix, and all rows must have the same amount of values (separated by whitespace).

It is possible to specify more advanced options by including a line with two dollar signs ($$) that follows on the next immediate line after the variable name declaration. These options allow names for vectors, and colnames, as well as data.frames objects. For complete details see the PBS modelling PDF.

Note

"P" format only supports a list of vectors, or matrices, but cannot support sub-lists. However "D" format supports all R objects

See Also

`readList`

`dput`