PBS Modelling 1.58: User's Guide

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

2007

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

Fisheries and Oceans Canada

Pêches et Océans Canada

Canada

Last update: August 28, 2007

Correct citation for this publication:

Schnute, J.T., Couture-Beil, A., and Haigh, R. 2007. PBS Modelling 1.58: user's guide revised
      from Canadian Technical Report of Fisheries and Aquatic Science 2674: vi + 123 p.
      Last updated August 28, 2007

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Schnute, J.T., Couture-Beil, A., and Haigh, R. 2007. PBS Modelling 1.58: user's guide revised from Can. Tech. Rep. Fish. Aquat. 2674: vi + 123 p. Last updated August 28, 2007.

This 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., and Haigh, R. 2006. PBS Modelling 1.58: guide de l'utilisateur révisé de Can. Tech. Rep. Fish. Aquat. Sci. 2674: vi + 123 p. Dernier mis à jour August 28, 2007.

Le présent rapport décrit la trousse R *PBS Modelling* qui contient des logiciels permettant de rendre plus aisés la conception, les essais et l'utilisation des modèles numériques. L'acronyme PBS fait référence à la *Pacific Biological Station* (Station biologique du Pacifique), un grand laboratoire axé sur l'étude des pêches sur la côte canadienne du Pacifique à Nanaimo, en Colombie-Britannique. Initialement conçue pour les chercheurs travaillant sur les pêches, cette trousse peut être utilisée dans de nombreux domaines scientifiques. *PBS Modelling* contient principalement des outils qui facilitent la construction et la modification d'une interface graphique (GUI) sur mesure, adaptée à un problème particulier. Bien que cette trousse s'appuie sur l'interface R pour Tcl/Tk, ses utilisateurs n'ont pas besoin de connaître le langage Tcl/Tk. En plus d'offrir des outils de conception de GUI, *PBS Modelling* propose des logiciels permettant d'échanger plus facilement des données entre les composants de divers modèles, d'effectuer des analyses statistiques spécialisées et de produire des graphiques utiles pour la modélisation et l'analyse des données concernant les pêches. Les exemples fournis mettent en application des idées classiques glanées dans les articles publiés sur les pêches ainsi que dans nos propres articles publiés. Ces exemples fournissent également des modèles dont l'utilisateur peut s'inspirer pour concevoir des protocoles d'analyse sur mesure à l'aide d'autres librairies R telles que *PBS Mapping*, *odesolve* et *BRugs*. Les utilisateurs qui désirent construire de nouvelles trousses peuvent utiliser *PBS Modelling* et la trousse simple *PBS Try* comme prototypes. Ce procédé est décrit de manière détaillée en annexe, y compris l'utilisation du langage C pour les calculs.

Preface

After working with fishery models for more than 30 years, I've used a great variety of computer software and hardware. Currently, the free distribution of R (R Development Core Team 2006a) provides an excellent platform for software development in an environment designed to support multiple computers and operating systems. Furthermore, an associated network of contributed libraries on the Comprehensive R Archive Network (CRAN: http://cran.r-project.org/) gives access to a wealth of algorithms from many users in various fields. This disciplined system allows users, like the authors of this package, to distribute software that extends the utility of R in new directions.

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

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 A. You might want to start by running the function `testWidgets()`. Co-author Rowan Haigh likes the subtitle: "modelling the world with gooey substances."

Second, we want to demonstrate interesting analyses related to our work in fishery management and other fields. The function `runExamples()` illustrates some of these, as described further in Section 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), `ddesolve` (to solve delay differential equations), 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 B. 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.

As with our earlier package *PBS Mapping*, Rowan and I employed a bright student who could learn quickly and implement creative ideas. Dr. Jim Uhl (Computing Science) and Dr. Lev Idels (Mathematics), both from Malaspina University-College (MUC) here in Nanaimo, drew my attention to the student Alex Couture-Beil, who has strong credentials in both fields. Rowan and I gave him a few initial specifications, and he quickly got ahead of us by extending our ideas in new and useful directions. *PBS Modelling* version 1 represents the result of an evolutionary process, as we experimented with design concepts that would support our modelling goals. Users familiar with the earlier version 0.60 (posted on CRAN in August, 2006) may need to revise their code slightly to make it work with this version.

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

Fishery management depends on models with a great range of complexity, starting from some fairly simple ideas. Unfortunately from a coding perspective, "industrial strength" models can't run exclusively in R. Algorithms with high computational requirements don't run fast enough in R for practical application, due to interpretive code and other technical limitations. Examples in *PBS Modelling* often illustrate ideas at the simple end of the spectrum, although the package can certainly be used to manage external software designed to deal with greater complexity.

Scientifically, I like to work from both ends of the spectrum. The behaviour of a complex model sometimes mimics a much simpler model, and it helps to become well versed in some of the simpler cases. I appreciate the motto of Canadian storyteller and humorist Stuart McLean, who hosts a CBC radio broadcast *The Vinyl Cafe* (http://www.cbc.ca/vinylcafe/), "We may not be big, but we're small."

Jon Schnute, December 2006

This page has been left intentionally left blank for printing purposes.

1. Introduction

This 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, 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 A). A user of *PBS Modelling* does not need to know Tcl/Tk.

Computer models typically involve a variety of components, such as code, data, documentation, and a user interface. Figure 1 illustrates the tangled relationships that sometimes accompany computer model design. *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. In addition to its interactive role, the GUI becomes an archival tool that reminds the developer how components, functions, and data tie together. Consequently, it facilitates the process of restarting a project at a future date, when details of the design may have been forgotten.
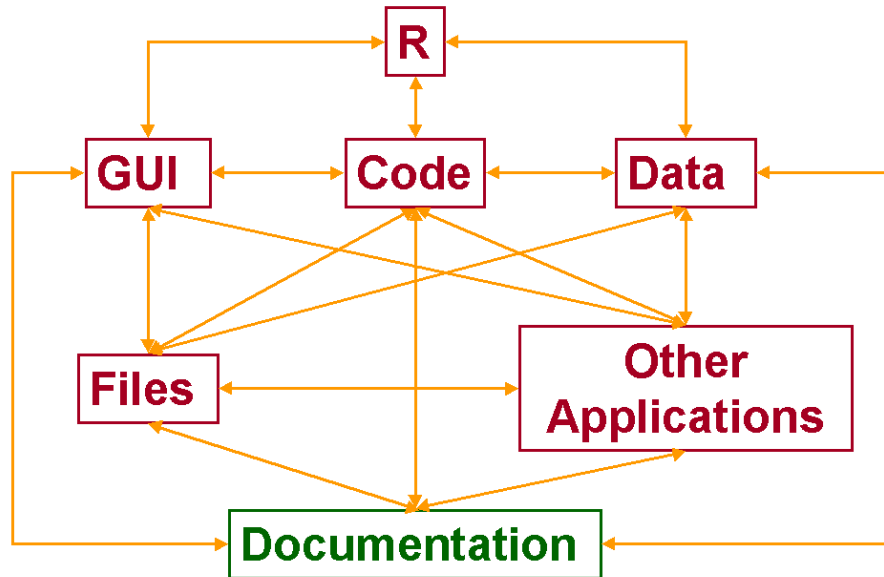
Figure 1. Tangled relationships among computer model components.
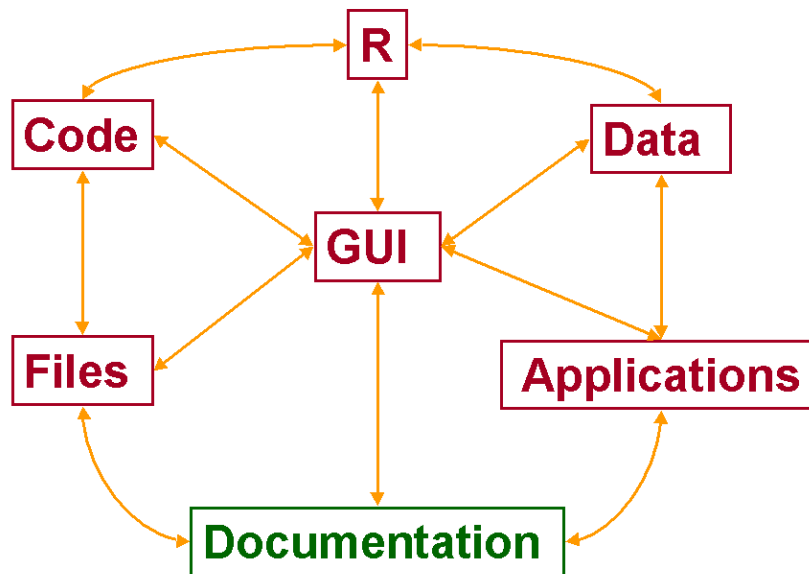
Figure 2. Computer model components organized with a graphical user interface (GUI).

In *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 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. This guide explains the context and general purpose of all functions in *PBS Modelling*. Consult the help files for complete technical details.

Appendix A gives the complete syntax for all visual components (called *widgets*) available for writing a window description file to construct a customized GUI. Appendix B 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 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 compressed file from the authors of this report or directly from the CRAN web site http://cran.r-project.org/.

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

2. GUI tools for model exploration

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

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

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

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

File 1: `LissajousCurve.txt`

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

File 2: `LissajousCurve.r`

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

This analysis can be accomplished with the R code and window description file shown in Table 1. Assume that these two files reside in the current working directory and that *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 any of the four entry boxes, and click <Plot>.
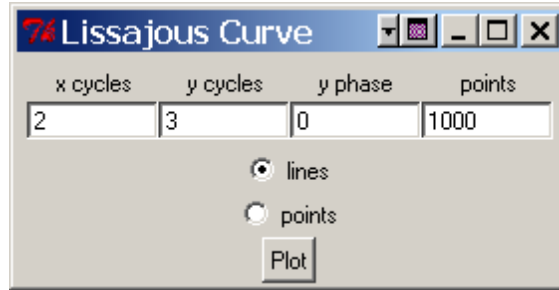
Figure 3. GUI generated by the description file `LissajousCurve.txt` in Table 1. It contains five widgets: the window titled "Lissajous Curve", a vector of four entries, two linked radio buttons (<lines> and <points>), and a <Plot> button.
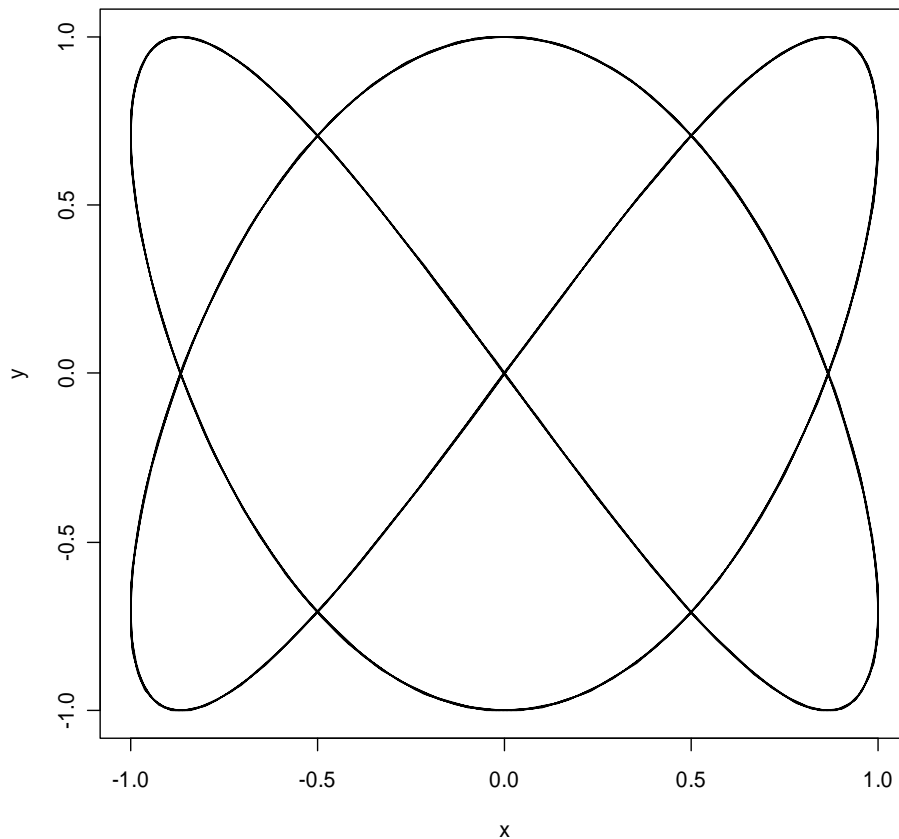
Figure 4. Default graph for the "Lissajous Curve" project, obtained by clicking the <Plot> button in Figure 3. The *x* variable goes through two cycles while the *y* variable goes through 3 cycles. A line graph is drawn through 1000 points generated by the algorithm (1).

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 (Figure 4) is actually drawn (i.e., the R function `drawLiss` is called) when the user presses a `button` that contains the text "Plot". In 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 by writing them to the user's global environment `.GlobalEnv`.)

2.2. Window description file

A window description file currently supports the following 18 widgets:

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

17. `slideplus` – an extended slide bar that also displays a minimum, maximum, and current value;

18. `history` – a device for archiving parameter values corresponding to different model choices, so that a "slide show" of interesting choices can be preserved.

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 18 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. Window names must use only letters and numbers; they cannot contain a period (dot) or any other punctuation. When running a program with multiple windows, only one window will be current (i.e., selected by the user) at any particular time. Normally, a user selects a window by clicking on it, but the function `focusWin` allows program control of the window currently in focus. Thus, activity in one window might be used to shift the focus to another.

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

Although `createWin` normally builds a GUI from a description file, it will also accept a vector of strings equivalent to such a file. Thus, a file of R source code can define a GUI directly, without the need for a separate description file. illustrates how this can be done in a simple case. To see the character vectors equivalent to a given description file (say, `winDesc.txt`), type the R command:

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

In particular, if the description file includes a backslash or double quote character, the corresponding R string must represent it as `\ \` or `\ "`, respectively. Despite this alternative of embedding window descriptions in R source files, we recommend writing separate files to define GUIs, except perhaps for very simple models.

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

<div align="center">

File: Simple.r

</div>

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

R programs need to share data with a GUI window. *PBS Modelling* provides six functions that deal with values of R variables named in a description file:

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

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

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

description file could give a unique action name to each potential action, so that the vector would give an unambiguous record of user actions.

Two functions provide support for selecting a file from a GUI:
- `promptOpenFile` shows the current directory for choosing a file to open;
- `promptSaveFile` shows the current directory for naming a file to save.

Files can be opened in programs external from R depending on their file extension:
- `openFile` opens a file using the default program for the file extension;
- `setPBSext` overrides the default program associated with an extension;
- `getPBSext` shows the overridden file extension and associated program.

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

On a Windows platform, the native R function `shell.exec` (called by `openFile`) automatically chooses a default from the registry. For this reason, our distribution specifies an empty list:

```
getPBSext() returns list().
```

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

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

where `%f` denotes the file name in the string passed to the operating system. On Unix platforms, it may be essential to specify defaults 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`.

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. The `history` widget defines and uses the list `PBS.history` in the global environment to store a saved history.

Normally, a user would invoke a `history` widget simply by including a reference to it in the description file. However, *PBS Modelling* includes some support functions for customized applications:
- `initHistory` initializes data structures for holding a collection of history data;
- `addHistory` saves the current window settings to the current history record;
- `rmHistory` removes the current record from the history;
- `backHistory` and `forwHistory` move backward and forward between successive history records;
- `firstHistory` and `lastHistory` move to the first and last records in the history;
- `jumpHistory` moves to a specified record in the history;
- `exportHistory` and `importHistory` save and load histories from files;

- `clearHistory` removes all records from the current collection.

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

2.4. Internal data

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

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

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

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

Users can explore the contents of `.PBSmod` with the R structure command `str`. For example, from the R console, type `runExamples()` and select the example "CalcVor". Then type the command `str(.PBSmod,2)` to shows the list structure to a depth of 2. This reveals

all the list components discussed above. Further details appear by exploring the structure to depths 3, 4, or more. Notice also how the contents change as different examples are selected.

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

Table 3. Sample data file for *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
```

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 rownames="" colnames="" byrow=TRUE
$$ data modes=numeric ncol rownames="" colnames byrow=TRUE
$$ 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.

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. This method works well with programs written for *AD Model Builder* (http://otter-rsch.ca/admodel.htm), a package used extensively in fishery research and other fields. It uses reverse automatic differentiation (AD; Griewank 2000) for highly efficient calculation of maximum likelihood estimates.

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

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

produces a corresponding R list object named `myData`.

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

4. Support functions for graphics and analysis

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

4.1. Graphics utilities

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

`drawBars` ....Draw a linear bar plot on the current graph.
`genMatrix`..............Generate a test matrix for use in `plotBubbles`.
`plotACF`..................Plot autocorrelation bars (ACF) from a data frame, matrix, or vector.
`plotAsp`..................Plot a graph with a prescribed aspect ratio, preserving `xlim` and `ylim`.
`plotBubbles`.........Construct a bubble plot for a matrix.
`plotCsum` ....Plot cumulative sum of a vector, with value added.

`plotDens` ....Plot density curves from a data frame, matrix, or vector.
`plotTrace` ....Plot trace lines from a data frame, matrix, or vector.

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

`pickCol` ....Pick a colour from a complete palette and get the hexadecimal code.
`testCol` ....Display named colours available based on a set of strings.
`testLty` ....Display line types available.
`testLwd` ....Display line widths.
`testPch` ....Display plotting symbols and backslash characters.

4.2. Data management

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

4.3. Function minimization and maximum likelihood

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

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

- `val` – the actual value of parameter x_i;
- `min` – a minimum allowable value of x_i;
- `max` – a maximum allowable value of x_i; and

- `active` – a logical value that determines whether or not the minimization algorithm should vary the value of x_i. If `active=F`, then x_i remains unchanged at the value `val`.

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

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

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

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

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

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

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

In summary, *PBS Modelling* has four functions that facilitate function minimization.
`calcMin`................Calculate the minimum of a user-defined function.
`scalePar` ....Scale parameters to surrogates in the range [0,1].
`restorePar`.........Restore actual parameters from surrogate values.
`GT0` ....Restrict a numeric variable to a positive value ("Greater Than 0").

4.4. Handy utilities

```
calcFib................Calculate Fibonacci numbers (included only to illustrate the use of C code).
calcGM.................Calculate the geometric mean of a vector of numbers.
findPat................Find all strings that include any string in a vector of patterns.
pause....................Pause, typically between graphics displays.
showArgs ..............Show the arguments for a specified widget in Appendix A.
testWidgets ......GUI to test all widgets listed in Appendix A.
view.......................View the 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 contains documentation, R code, a window description file, and (if required) other supporting files. All relevant files appear in the R library directory `PBSmodelling\Examples`. An example named `xxx` typically has corresponding files `xxxDoc.txt` or `xxxDoc.pdf` (documentation), `xxx.r` (R code), and `xxxWin.txt` (a window description). In the GUI for each example, buttons labelled `Docs`, `R Code`, and `Window` open these files **provided that suitable programs have been associated with the file extensions `*.txt`, `*.pdf`, and `*.r`.** In particular, the Acrobat Reader must be installed for reading `*.pdf` files, and you may need to associate a text file editor with `*.r`. On some systems, it may be necessary to use the function `setPBSext` to define these associations, as discussed earlier in Section 2.3.

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

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

The examples documented here illustrate only some of those available in version 1 of *PBS Modelling*. For instance, we also include a `TestFuns` GUI that we have used as a tool for debugging various functions in the package. In future versions, we plan to add more examples that illustrate important modelling concepts and provide convenient supplementary materials for university courses in fisheries, biology, ecology, statistics, and mathematics. The function

`runExamples()` should always represent the complete list currently available, and the `Docs` button for each case should link to the appropriate documentation.

The nine examples presented in this section illustrate some of the possibilities available in *PBS Modelling*, although the documentation may be somewhat out of date. For example, the figures in this report may not correctly represent current versions of the GUIs and their associated graphical output. Use the `Docs` button to read the most current information for each example. If this seems rather primitive, please wait for improvements in future versions.

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 moment formulas in the documentation, without sample bias correction formulas like those described by Aitchison and Brown (1969). Three buttons at the bottom of the GUI portray the data visually as density curves, cumulative proportions, and paired scatter plots.

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 (2007) provide further technical details about these three distributions.

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.

5.1.4. `CalcVor` – Calculate Voronoi tessellations

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

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

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

5.2. Statistical analyses

5.2.1. `LinReg` – Linear regression

Figure 9. `LinReg` GUI (left) and regression plot (right). The linear regression uses the `cars` dataset (*n*=50) to predict `dist` vs. `speed`. The plot shows observations (green circles), fitted line (solid blue line), the 95% confidence limits of the fitted model (solid red lines), the 95% CL of the data (dashed purple lines), and the fits using the Bayes posterior estimates of (*a*,*b*) (gold lines).

The example `LinReg` estimates parameters in a linear regression $y = a + bx$ using either simulated data or data objects that come with the R-package.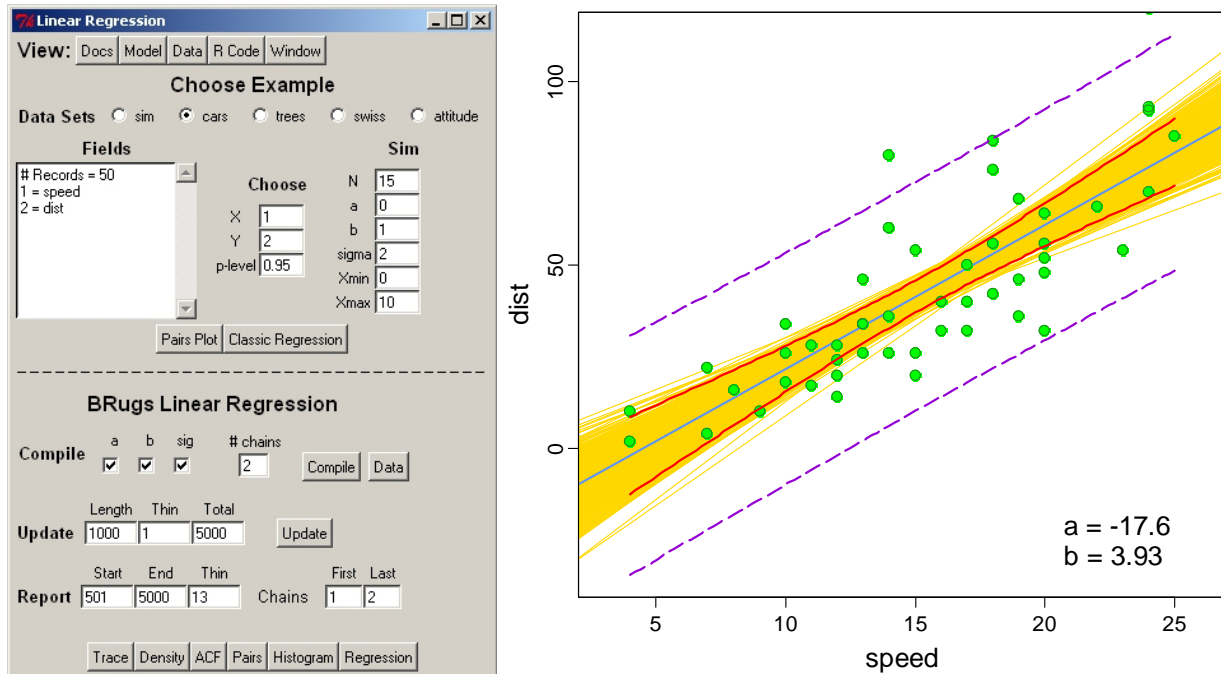 We compare a classical frequentist regression with results from Bayesian analysis, using the BRugs library to interface with the program WinBUGS. After selecting various data options, "Pairs Plot" shows a pairs plot (x, y) and "Classic Regression" adds confidence limits (at "p-level") from regression 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 9).

5.2.2. `MarkRec` – Mark-recovery

Figure 10. `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

$$\hat{N} = \frac{S}{R} M = \frac{M}{R} S.$$
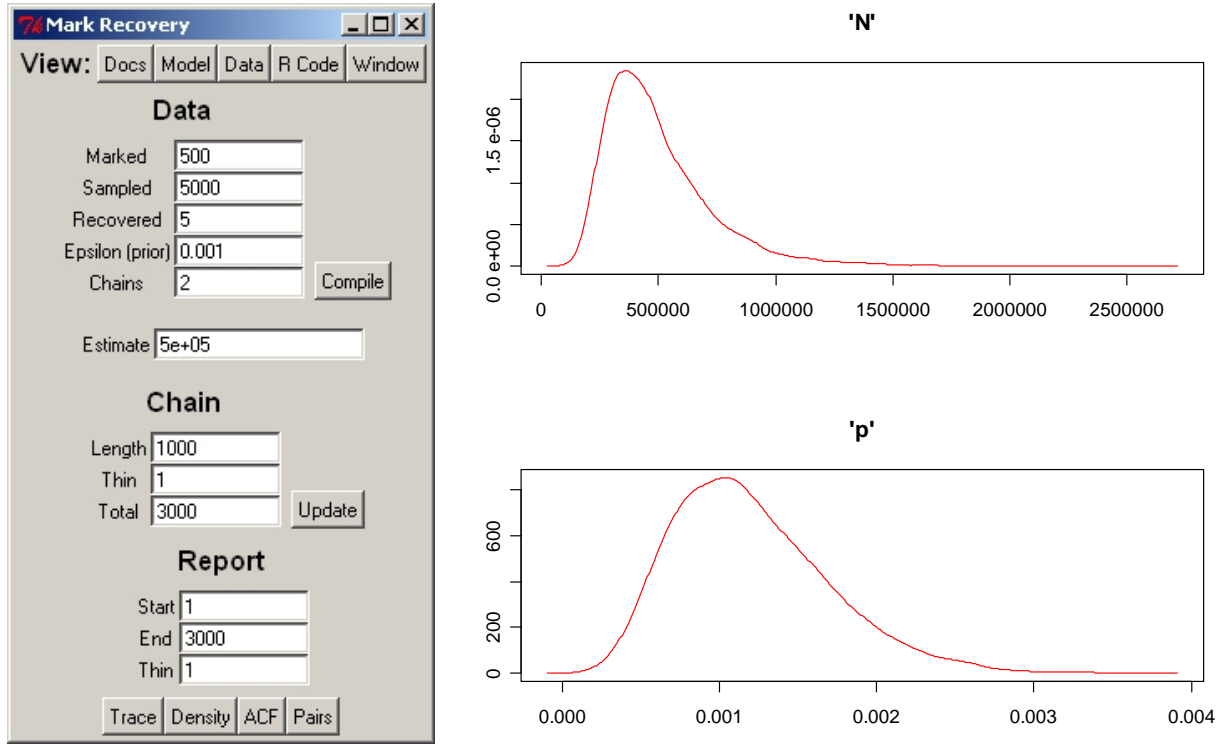
When recoveries are low ($R \approx 0$), the posterior distribution of *N* exhibits a fat tail (Figure 10).

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
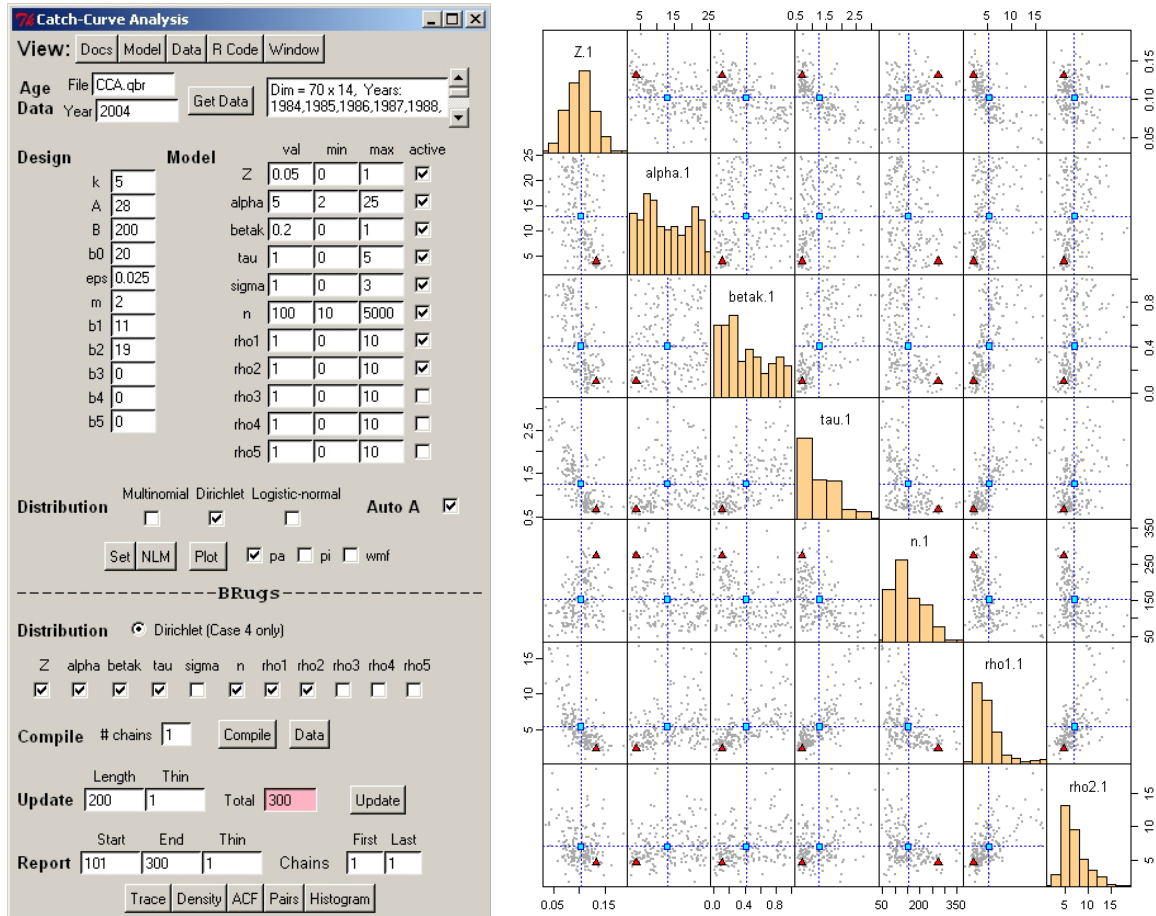
Figure 11. CCA GUI (left) and parameter pairs plot (right). Comparison of Bayes posterior distribution of CCA model parameter estimates from chain 1 (*N*=100). Symbols indicate means (blue squares) and modes (red triangles). Diagonal shows parameter estimate distributions.

The example CCA illustrates a catch-curve model proposed by Schnute and Haigh (2007). It incorporates effects of survival, selectivity, and recruitment anomalies on age structure data from a single year. After making various model choices, press "Set", "NLM" (which may take several seconds), and "Plot" to view the maximum likelihood estimates and their relationship with the data. A WinBUGS model ("Model") allows us to calculate posterior distributions. (See the last few lines of "Model".) As in MarkRec, select parameters to monitor, specify a number of chains, and "Compile" the model. "Update"s may be slow, but eventually they produce interesting posterior samples (Figure 11). "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 2007) is published. *PBS Modelling* includes the data for this example as the matrix CCA.qbr.

5.3. Other applications

5.3.1. `FishRes` – Fishery reserve
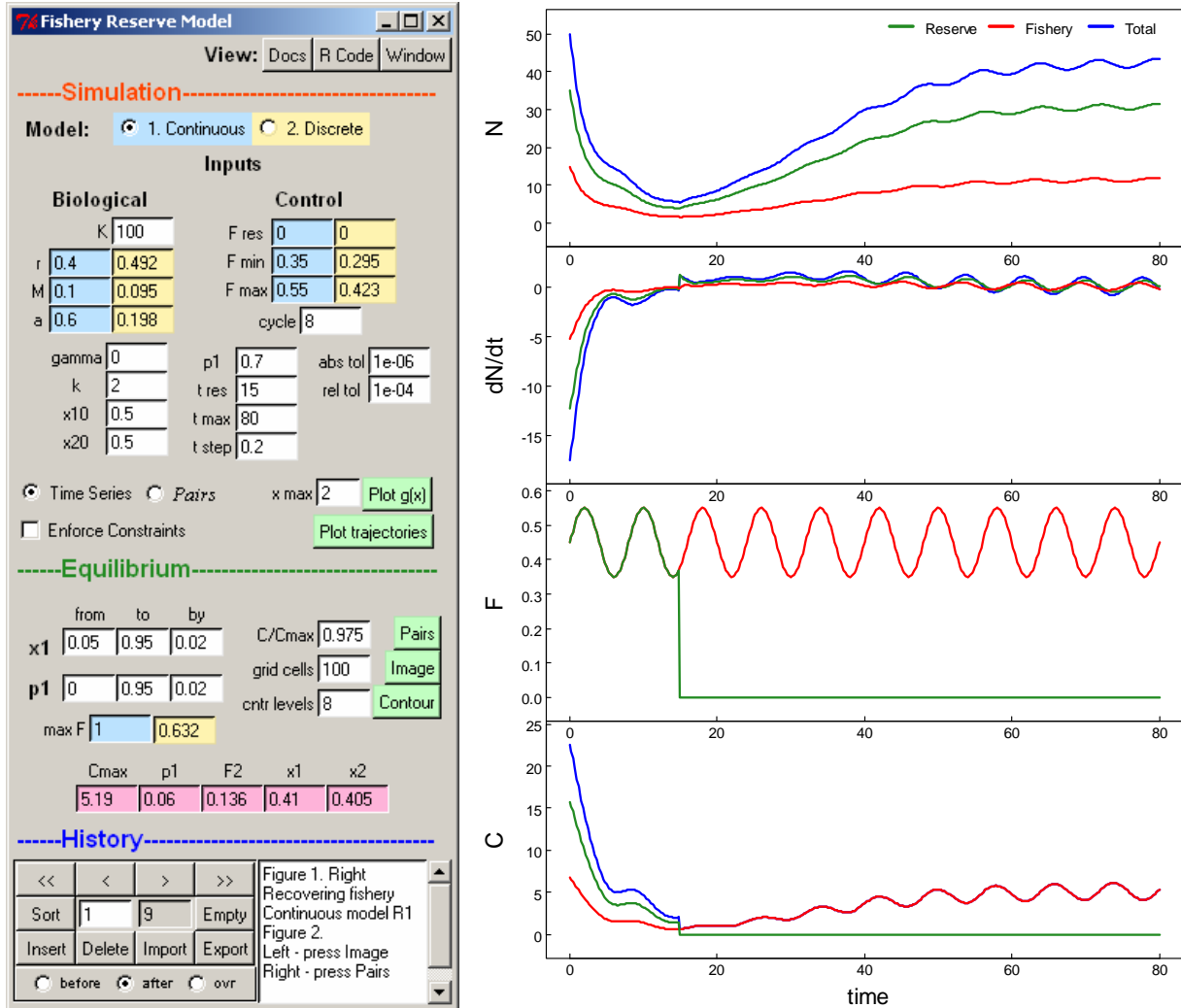
Figure 12. `FishRes` – Recovery of a heavily fished population after establishing a reserve. The GUI (left) shows all input values (parameters and controls). The selected continuous time model uses input values common to both models (white background) and values specific to the continuous model (blue background). Corresponding values are computed for the discrete model (yellow background). Output trajectories (right) trace various results (N = population, dN/dt = instantaneous change in population, F = instantaneous fishing mortality, C = instantaneous catch) for the reserve and fishery. Fishing mortality follows a sinusoid determined by F_{min}, F_{max}, and the cycle length n.

The example `FishRes` (Figure 12) models a fish population associated with a marine reserve in continuous or discrete time (delay differential or difference equations, respectively). For details see Schnute et al. (2007), which can be viewed by pressing the Docs button in the GUI. The R libraries `akima`, `ddesolve`, and `odesolve` are required.

5.3.2. `FishTows` – Fishery tows

Figure 13. `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 Brown, J.A.C. 1969. The lognormal distribution, with special reference to its uses in economics. Cambridge University Press. Cambridge, UK. xviii+176 p.

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

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

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

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

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

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

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

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

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

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

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

Schnute, J.T. 2006. Curiosity, recruitment, and chaos: a tribute to Bill Ricker's inquiring mind. Environmental Biology of Fishes 75: 95-110.

Schnute, J.T., Boers, N.M., and Haigh, R. 2003. PBS software: maps, spatial analysis, and other utilities. Canadian Technical Report of Fisheries and Aquatic Sciences 2496. viii+82 pp.

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

Schnute, J.T., and Haigh, R. 2007. Compositional analysis of catch curve data with an application to *Sebastes maliger*. ICES Journal of Marine Science 64: 218-233. Available at http://icesjms.oxfordjournals.org/content/vol64/issue2/index.dtl, reference number doi:10.1093/icesjms/fsl024.

Schnute, J.T., Haigh, R., and Couture-Beil, A. 2007. Mathematical models of fish populations in marine reserves. Report on a Collaborative Project between Malaspina University-College and the Pacific Biological Station. February 2007, 24 pp. (File `FishResDoc.pdf` available in the package `PBSmodelling`.)

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

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

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

Appendix A. Widget descriptions

This appendix lists 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="" fg="black" bg="" width=0
   function="" action="button" sticky="" padx=0 pady=0
```

Arguments

text .... text to display on the button
font .... font for labels – specify family (Times, Helvetica, or Courier),
                         size (as point size), and style (bold, italic, underline,
                         overstrike), in any order
fg .... colour for label fonts
bg .... background colour for widget
width .... button width, the default 0 will adjust the width to the minimum required
function .... R function to call when the button is pushed (i.e., clicked by the mouse)
action .... string value associated whenever this widget is engaged
sticky .... option for placing the widget in available space; valid choices are:
                         N, NE, E, SE, S, SW, W, NW
padx .... space used to pad the widget on the left and right
pady .... space used to pad the widget on the top and bottom

Example

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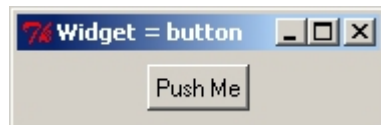
Check

Description

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

Usage

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

Arguments

name ....name of R variable altered by this check box (required)
mode ....R mode for the associated variable, where valid modes are
                `logical` or `numeric`
checked.............if `TRUE`, the box is checked initially and the variable is set to `TRUE` or `1`
text ....identifying text placed to the right of this check box
font ....font for labels – specify family (`Times`, `Helvetica`, or `Courier`),
                size (as point size), and style (`bold`, `italic`, `underline`,
                `overstrike`), in any order
fg.........................colour for label fonts
bg.........................background colour for widget
function ....R function to call when the check box is changed
action...............string value associated whenever this widget is engaged
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.
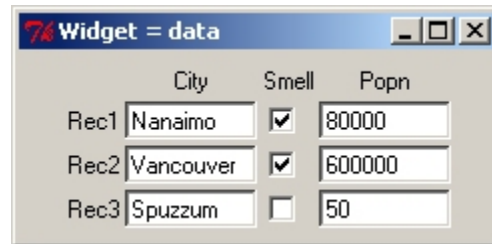
Usage

```
type=data nrow ncol names modes="numeric" rowlabels=""
   collabels="" rownames="X" colnames="Y" font="" fg="black"
   bg="" entryfont="" entryfg="black" entrybg="white"
   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

rownames ....string scalar or vector of length `nrow` to name the rows of the data frame

colnames ....string scalar or vector of length `ncol` to name the columns of the data frame

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

fg.........................colour for label fonts

bg.........................background colour for widget

entryfont........font of entries appearing in input/output boxes

entryfg.............font colour of entries appearing in input/output boxes

entrybg.............background colour of input/output boxes

values...............default values (either one value for all data frame components or a set of `nrow*ncol` values)

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

function ....R function to call when any entry in the data frame is changed

enter..................if `TRUE`, call the function only after the <Enter> key is pressed

action...............string value associated whenever this widget is engaged

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="" fg="" bg=""
  entryfont="" entryfg="black" entrybg="white" function=""
  enter=TRUE action="entry" mode="numeric" sticky="" padx=0
  pady=0
```

Arguments

name	name of R variable corresponding to this entry (required)
value	default value to display in the entry
width	character width to reserve for the entry
label	text to display above the entry box
font	font for labels – specify family (Times, Helvetica, or Courier), size (as point size), and style (bold, italic, underline, overstrike), in any order
fg	colour for label fonts
bg	background colour for widget
entryfont	font of entries appearing in input/output boxes
entryfg	font colour of entries appearing in input/output boxes
entrybg	background colour of input/output boxes
function	R function to call when the entry is changed
enter	if TRUE, call the function only after the <Enter> key is pressed
action	string value associated whenever this widget is engaged
mode	R mode for the value entered, where valid modes are: numeric, integer, complex, logical, character

```
sticky ...............option for placing the widget in available space; valid choices are:
                    N, NE, E, SE, S, SW, W, NW
padx ...................space used to pad the widget on the left and right
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
```
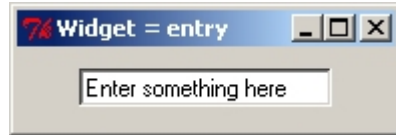

Grid

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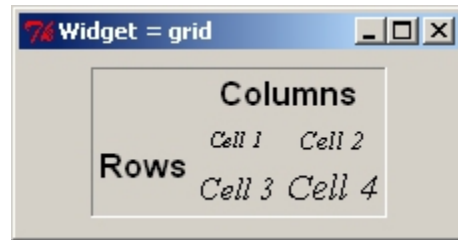

History

Description

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

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

Usage

```
type=history name="default" function="" import="" sticky=""
   padx=0 pady=0
```

Arguments

name....................name of history archive
function ....R function to call when the history record counter is changed

```
import ...............file name of a saved history to load when the widget is called
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"
vector length=3 names="alpha beta gamma" values="2 5 15"
history padx=20 pady=5
```


Label

Description

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

Usage

```
type= label text="" font="" fg="black" bg="" sticky="" padx=0
   pady=0
```

Arguments

```
text....................text to display in the label
font....................font for labels – specify family (Times, Helvetica, or Courier),
              size (as point size), and style (bold, italic, underline,
              overstrike), in any order
fg........................colour for label fonts
bg........................background colour for widget
sticky...............option for placing the widget in available space; valid choices are:
              N, NE, E, SE, S, SW, W, NW
padx....................space used to pad the widget on the left and right
pady....................space used to pad the widget on the top and bottom
```

Example

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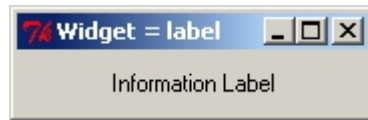

Matrix

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="" fg="black" bg=""
   entryfont="" entryfg="black" entrybg="white" values=""
   byrow=TRUE function="" enter=TRUE action="matrix"
   mode="numeric" width=6 sticky="" padx=0 pady=0
```
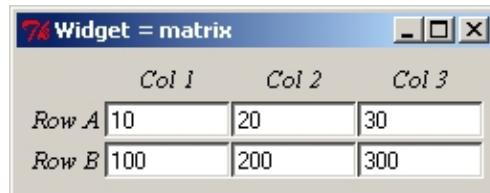
Arguments

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

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

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

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

fg........................colour for label fonts

bg........................background colour for widget

entryfont........font of entries appearing in input/output boxes

entryfg.............font colour of entries appearing in input/output boxes

entrybg.............background colour of input/output boxes

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

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

```
function ..........R function to call when any entry in the matrix is changed
enter.................if TRUE, call the function only after the <Enter> key is pressed
action...............string value associated whenever this widget is engaged
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.
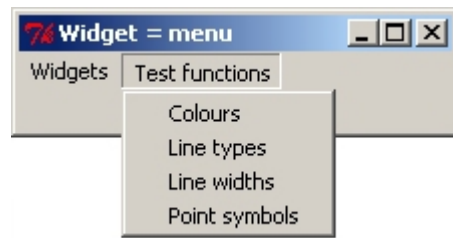
Usage

```
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 labels – 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 labels – 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..............string value associated whenever this widget is engaged

Null

Description

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

Usage

```
type=null 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"
```


Object

Description

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

Usage

```
type=object name font="" fg="black" bg="" entryfont=""
    entryfg="black" entrybg="white" vertical=FALSE byrow=TRUE
    function="" enter=TRUE action="data" width=6 sticky=""
    padx=0 pady=0
```

Arguments

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

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

fg.........................colour for label fonts

bg........................background colour for widget

entryfont........font of entries appearing in input/output boxes

entryfg............font colour of entries appearing in input/output boxes

entrybg.............background colour of input/output boxes

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

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

enter..................if `TRUE`, call the function only after the <Enter> key is pressed

action...............string value associated whenever this widget is engaged

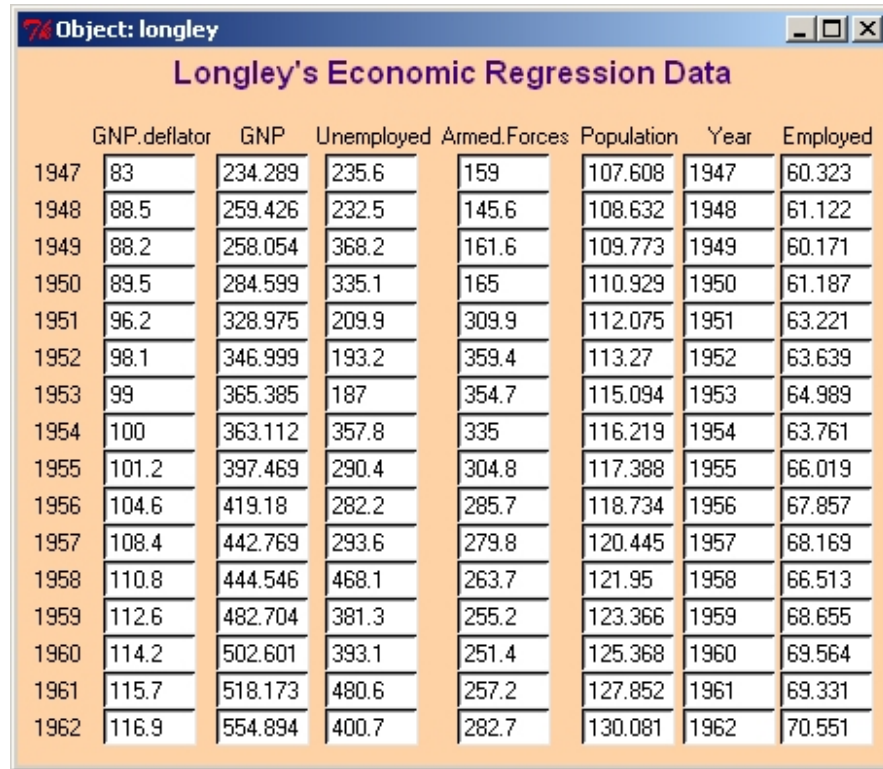width..................character width to reserve for the each entry in the vector

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

	GNP.deflator	GNP	Unemployed	Armed.Forces	Population	Year	Employed
1947	83	234.289	235.6	159	107.608	1947	60.323
1948	88.5	259.426	232.5	145.6	108.632	1948	61.122
1949	88.2	258.054	368.2	161.6	109.773	1949	60.171
1950	89.5	284.599	335.1	165	110.929	1950	61.187
1951	96.2	328.975	209.9	309.9	112.075	1951	63.221
1952	98.1	346.999	193.2	359.4	113.27	1952	63.639
1953	99	365.385	187	354.7	115.094	1953	64.989
1954	100	363.112	357.8	335	116.219	1954	63.761
1955	101.2	397.469	290.4	304.8	117.388	1955	66.019
1956	104.6	419.18	282.2	285.7	118.734	1956	67.857
1957	108.4	442.769	293.6	279.8	120.445	1957	68.169
1958	110.8	444.546	468.1	263.7	121.95	1958	66.513
1959	112.6	482.704	381.3	255.2	123.366	1959	68.655
1960	114.2	502.601	393.1	251.4	125.368	1960	69.564
1961	115.7	518.173	480.6	257.2	127.852	1961	69.331
1962	116.9	554.894	400.7	282.7	130.081	1962	70.551

Radio

Description

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

Usage

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

Arguments

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

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

text ....identifying text placed to the right of this radio button

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

fg .... colour for label fonts

bg .... background colour for widget

function .... R function to call when this radio button is selected

action .... string value associated whenever this widget is engaged

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" font="" fg="black" bg=""
   function="" action="slide" sticky="" padx=0 pady=0
```
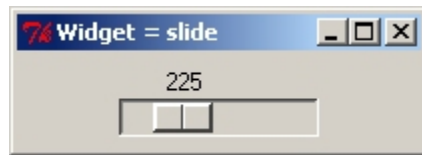
Arguments

name .... name of the numeric R variable corresponding to this slide bar (required)

from .... minimum value of the variable (must be an integer)

to .... maximum value of the variable (must be an integer)

value .... initial slide value, where the default is the specified `from` value

showvalue .... if `TRUE`, display the current slide value above the slide bar

orientation ... direction for orienting the slide bar: `horizontal` or `vertical`

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

fg .... colour for label fonts

bg .... background colour for widget

function .... R function to call when the slide value is changed

action .... string value associated whenever this widget is engaged

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

padx .... space used to pad the widget on the left and right

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 .... string value associated whenever this widget is engaged

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

padx .... space used to pad the widget on the left and right
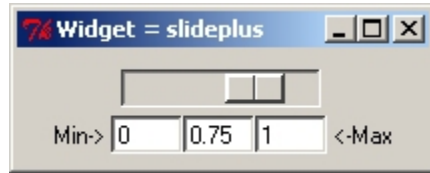
pady ....space used to pad the widget on the top and bottom

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 edit=FALSE scrollbar=TRUE
   fg="black" bg="white" mode="character" font="" value=""
   borderwidth=1 relief="sunken" sticky="" padx=0 pady=0
```

Arguments

name ....name of the R variable containing the text (required)
height ....text box height
width ....text box width
edit ....if TRUE, the user can edit the value stored in `name`
scrollbar ....if TRUE, a scroll bar is added to the right of the text box
fg ....colour for label fonts
bg ....background colour specified in hexadecimal format; e.g.,
                 `rgb(255,209,143,maxColorValue=255)` yields "#FFD18F"
mode ....R mode for the value associated with this widget, where valid modes are:
                 `numeric, integer, complex, logical, character`
font ....font for labels – specify family (`Times`, `Helvetica`, or `Courier`),
                 size (as point size), and style (`bold`, `italic`, `underline`,
                 `overstrike`), in any order
value ....default value to display in the text
borderwidth ...width of the border around the text box

```
relief ...............type of border around the text, where valid styles are:
                    raised, sunken, flat, ridge, groove, solid
sticky ...............option for placing the widget in available space; valid choices are:
                    N, NE, E, SE, S, SW, W, NW
padx ...................space used to pad the widget on the left and right
pady ...................space used to pad the widget on the top and bottom
```

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\""
```
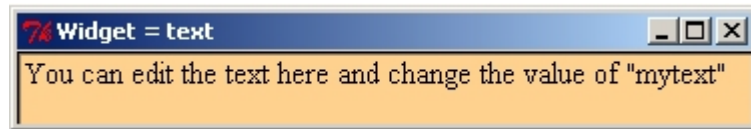

Vector

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="" vecnames=""
   font="" fg="black" bg="" entryfont="" entryfg="black"
   entrybg="white" 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 ...............labels for the vector display – either one label, a vector of length labels,
                    or NULL for no labels (default " " labels with names and, if number of
                    specified names is one, numbered elements)
values ...............default values (either one value for all vector components or a vector of
                    length values)
vecnames ..........string vector of length length to name the scalars or vector
font ...................font for labels – specify family (Times, Helvetica, or Courier),
                    size (as point size), and style (bold, italic, underline,
                    overstrike), in any order
```

```
fg.........................colour for label fonts
bg.........................background colour for widget
entryfont........font of entries appearing in input/output boxes
entryfg.............font colour of entries appearing in input/output boxes
entrybg.............background colour of input/output boxes
vertical ..........if TRUE , display the vector as a vertical column with labels on the left;
                  otherwise display it as a horizontal row with labels above
function ..........R function to call when any entry in the vector is changed
enter..................if TRUE, call the function only after the <Enter> key is pressed
action...............string value associated whenever this widget is engaged
mode....................R mode for the vector, where valid modes are:
                  numeric, integer, complex, logical, character
width..................character width to reserve for the each entry in the vector
sticky...............option for placing the widget in available space; valid choices are:
                  N, NE, E, SE, S, SW, W, NW
padx....................space used to pad the widget on the left and right
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"
```
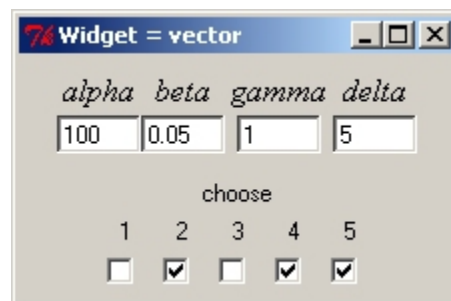

Window

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 bg="#D4D0C8"
    fg="#000000" onclose=""
```

Arguments

name....................unique name identifying an open window

title..................text to display in the window's title line

vertical ....if TRUE, arrange widgets vertically, top to bottom, within the window

bg.........................background colour for window

fg.........................colour for label fonts

onclose.............name of function called when user closes the window by pressing

Example

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

Appendix B. 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:

B.1. install the required software;
B.2. build PBS Modelling from source materials;
B.3. write source materials for a new package and compile them;
B.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-2.5.0\library\PBSmodelling\PBStools`.

B.1. Installing required software

Building R packages requires five pieces of free software. Duncan Murdoch currently maintains their availability and installation instructions at:
http://www.murdoch-sutherland.com/Rtools/
Users should periodically check this website for changes to the various software packages. We recommend installing each package on a path that does *not* include spaces. For example, avoid using `C:\Program Files`, even if that happens to be part of a package's default path. In this appendix, we use `C:\Utils` as a root directory for all required software. The list below gives a brief summary of the required software (Murdoch provides links to these products).

1. **R** itself, currently version 2.5.0 (`C:\Utils\R\R-2.5.0`). We assume that R is already installed from the CRAN web site http://cran.r-project.org/ and that it runs correctly on your computer. We also assume that the package `PBSmodelling` is installed in R.

2. **ActivePerl**: text scripting language (`C:\Utils\Perl\`).

3. **Rtools installer**: Command line tools (`C:\Utils\Rtools\`) and MinGW compilers, etc. Download and run the file `Rtools.exe`. The installation should create the subdirectories `\bin` for command line programs and `\MinGW` for the minimalist GNU C compiler for Windows. These tools are *essential*. DO NOT plan to use programs with the same name in an installation of Cygwin or any other UNIX emulator that happens to be installed on your computer.

4. The Microsoft **HTML Help Workshop** (`C:\Utils\HHW\`). Run the installation file `HtmlHelp.exe`. After installation, we think you can safely ignore a message that "This

computer already has a newer version of HTML Help". (If anyone has different information, please let us know.)

5. **MiKTeX**: a LaTeX and pdftex package (`C:\Utils\MiKTeX`). The link takes the user to http://www.miktex.org/. This processor for TeX and LaTeX files helps typeset help files within a package. Download the "basic" installation file, and install these components only. You can add more LaTeX packages from the Internet later, as required. (MiKTeX often does this automatically.) Take some time to investigate the MiKTeX package manager (`mpm.exe` or go to the "Programs" menu and select "MiKTeX 2.5", "Browse Packages").

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

a) Create a new subdirectory `\R` under the MiKTeX's directory for storing LaTeX styles and font definitions (e.g., `C:\Utils\MiKTeX\tex\latex`).

b) Copy into it all files from `\texmf` in the R installation tree (e.g., `C:\WinApps\R\R-2.5.0\share\texmf`). These should include `Rd.sty`.

c) Go to the "Start" menu, select "Programs" then "MiKTeX 2.5", and run the program "Settings". In the "General" tab, click the button marked "Refresh FNDB". This refreshes MiKTeX's file name database, so that it recognizes files in the new `\R` subdirectory.

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.

PBStools for building R packages

After these five pieces of software are installed, you're ready to start building R packages. For this purpose, create a new directory (e.g., `D:\Rdevel\`) that will contain your packages. Within the R library directory (`C:\Utils\R\R-2.5.0\library\`), find the subdirectory `PBSmodelling\PBStools`. Copy all the batch files there into your new packages directory. You should have these 11 files:

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

IMPORTANT: 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-2.5.0\bin
set TOOLS_PATH=C:\Utils\Rtools\bin
set PERL_PATH=C:\Utils\Perl\bin
set MINGW_PATH=C:\Utils\Rtools\MinGW\bin
```

```
set TEX_PATH=C:\Utils\MiKTeX\miktex\bin
set HTMLHELP_PATH=C:\Utils\HHW
```

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

Hopefully, your installation is now complete. In your new packages directory, run `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.

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

B.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 B.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`.

3. Our batch file uses the command `tar -xzvf PBSmodelling.tar.gz`, where `tar.exe` appears in the `\Rtools` directory (section B.1, step 3). The command line parameters specify a verbose (`v`) extraction (`x`) of the given file (`f`), after filtering with `gzip` (`z`).

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

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

.tar.gz file, you should see a series of DOS messages reporting "OK" to various tests. A distinct pause might accompany the message: "checking whether package 'PBSmodelling' can be installed ...".

5. You might also encounter a delay as MiKTeX downloads the LaTeX package `lmodern`, part of a larger package `lm`. If this is really slow, you can abort the process and install `lm` with the MiKTeX package manager, as discussed in step 5 of section B.1. Choose a remote server near you. You only need to do this once. When it's finished, run `checkPBS.bat` again.

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

7. In the base directory `D:\Rdevel`, double-click the icon for `buildPBS.bat` or type the command `buildPBS` in a corresponding command window. This creates the file `D:\Rdevel\PBSmodelling.zip`, which could be used to install `PBSmodelling` from a local zip file.

8. Again in the base directory `D:\Rdevel`, double-click the icon for `packPBS.bat` or type the command `packPBS` in a corresponding command window. This creates a new package distribution file `PBSmodelling_x.xx.tar.gz` that replaces the one downloaded from CRAN in step 1.

9. Finally, type the command `makePDF PBSmodelling` in a command window for `D:\Rdevel`. This generates an indexed documentation file `PBSmodelling.pdf`. See Appendix C.3 for further details about the use of this file for producing this report.

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

B.3. Creating a new R package

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

1. Start by locating the file `PBStry_x.xx.tar.gz` in the R library directory `\PBSmodelling\PBStools`. Copy this file into your development directory (`D:\Rdevel`), and rename it (or copy and rename the copy) to obtain the file `PBStry.tar.gz`.

2. Remove any previous traces of `PBStry` in your development directory, such as subdirectories `PBStry`, `PBStry.Rcheck`, and `.Rd2dvi`, along with the documentation file `PBStry.pdf`.

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

4. Use your editor to open the file `DESCRIPTION` in the root directory `\PBStry`. This file, essential in every R package, contains key information in a special format (RDCT 2006b, section 1.1.1). The following example illustrates a minimal set of required fields.

5.
```
Package: MyPack
Version: 1.00
Date: 2006-12-31
Title: My R Package
Author: User of PBS Modelling
Maintainer: User of PBS Modelling
Depends: R (>= 2.3.0)
Description: My customized R functions
License: GPL version 2 or newer (recommended)
```

6. The package name in `DESCRIPTION` must agree with the directory name in which this file lies. For example, if you change `PBStry` to `MyPack` in `DESCRIPTION` and rename the directory from `\PBStry` to `\MyPack`, you have effectively changed the package name. Similarly, if you change the version to `1.01`, you have effectively changed the version number that appears in the file names for distributing your package.

7. The subdirectory `\PBStry\R` contains all R code used by the package. For example, `PBStry` includes seven R functions (`calcFib`, `calcFib2`, `calcGM`, `calcSum`, `findPat`, `pause`, and `view`). The seven files could be combined into a single file (such as `PBStry.R`), but we use separate files here for clarity. The functions all have relatively simple code, hopefully comprehensible to users with limited R experience. Five of them come from `PBSmodelling`. Three of them (`calcFib`, `calcFib2`, `calcSum`) call compiled C code, as we discuss more completely in section B.4 below.

8. By convention, the distinct file `zzz.R` defines code for initializing the package. In this case the function `.First.lib`, calls `library.dynam` to load a dynamic link library (`PBStry.dll`) created from compiled C code during the build process.

9. When a version number changes, the `DESCRIPTION` file must be changed accordingly. We also like to make a corresponding change in `zzz.R`, so that the version number appears on the R console when the library is loaded. `PBStry` illustrates this possibility for `zzz.R`.

10. The subdirectory `\PBStry\data` contains all data objects that come with the package. Here, the binary file `QBR.rda` holds a matrix of quillback rockfish (*Sebastes maliger*) sample data used in the `CCA` example above (section 5.2.3). The same data matrix is called `CCA.qbr.hl` in `PBSmodelling`.

11. If you want to add data to a new package, first create the object (e.g., `myData`) in R and then execute the command:
    `save(myData,file="myData.rda")`
    The object name must match the prefix in the file name, and the suffix must be `.rda`. Include the resulting file in your package's `\data` subdirectory.

12. The subdirectory `\PBStry\man` contains a documentation file for every object in the package. `PBStry` has six functions and one data set, so the `\man` subdirectory has seven corresponding R documentation files (`*.Rd`). An additional file `PBStry.Rd` documents the package as a whole. Rd files use a rather complex scripting language (RDCT 2006b, section 2) that can be converted to help files in several formats (PDF, HTML, text). For many packages, the examples in `PBStry` may provide adequate prototypes. They represent three distinct cases: functions (e.g., `calcGM.Rd`, `findPat.Rd`), data sets (`QBR.Rd`), and complete packages (`PBStry.Rd`).

13. The subdirectory `\PBStry\src` contains source code for C code to be compiled into the dynamic link library `PBStry.dll`. We include sample files to calculate Fibonacci numbers iteratively (`fib.c`, `fib2.c`) and to add the components of a numeric vector (`sum.c`). In section B.4, we discuss the linkage between R code and compiled C functions.

14. Finally, the subdirectory `\PBStry\inst` contains files that are to be included directly in the R library tree for `PBStry` when the package is installed. The file `PBStry-Info.txt` briefly describes the context and purpose of the trial package.

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

The larger package `PBSmodelling` offers more prototypes and uses a somewhat different style. The main directory includes the required `DESCRIPTION` file, plus a second file `NAMESPACE` that lists all objects available to a user of the package. Effectively, the namespace mechanism distinguishes between objects provided by the package and other (hidden) objects required for the implementation, but not intended for public use. Our `NAMESPACE` file contains the rather cryptic instruction: `exportPattern("^[^\\.]")`. The R string `"^[^\\.]"` translates to the regular expression `^[^\.]` that designates any pattern not starting with a period (`.`). We don't export "dot" objects, whose names in R start with a period. (For more complete information on these functions, see Appendix C2.) The `NAMESPACE` file must also import functions required from other packages. Because `PBSmodelling` relies on `tcltk`, the file includes the command: `import(tcltk)`.

In `PBStry`, without a namespace, the file `zzz.R` defines the initializing function `.First.lib`, as mentioned in step 8 above. By contrast, the namespace protocol in `PBSmodelling` requires a different name for the initializing function: `.onLoad` in `zzz.R`.

In summary, we recommend building a new package by editing, adding, and deleting prototype files in `PBStry`. Our batch files can facilitate tests and debugging. For more advanced work, particularly packages with a namespace protocol, look at `PBSmodelling`. Have a current version of RDCT (2006b) available, and consult that manual when necessary. We find it useful to keep the PDF file open and to use Acrobat's search feature (Ctrl-F) to find topics of interest.

B.4. Embedding C code

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

Table B1. C representations of R data types.

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

[1] `Rcomplex` is defined in `Complex.h`.

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

The `.C()` calling convention uses the following key concepts:
- R must allocate the appropriate length and type of variables before calling a C function.
- R objects are transformed into an equivalent C type (Table B.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 B2. 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 B2). The R function uses C code to calculate the first n Fibonacci numbers iteratively, where a vector holds the last `len` numbers calculated. After ensuring that n and `len` satisfy obvious constraints, the R code creates a return array `retArr` of the appropriate length. The `.C` call passes n, `len`, and `retArr` by reference to the C function `fibonacci`. On exit, the vector `out` contains a list corresponding to the input variables n, `len`, and `retArr`, so that the third component `out[[3]]` holds the modified vector of values calculated by `fibonacci`. We encourage you also to examine a second example in `PBStry`, associated the files `calcSum.R` and `sum.c`.

Table B3. `.Call()` example adapted from `PBStry`, with two associated text files. R code in the first file calls C code in the second.

File 1: calcFib2.R

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

File 2: fib2.c

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

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

The `.C()` convention requires a fairly simple conversion of R objects into C types (Table B.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 calcFib2 in Table B3 illustrates an application of these concepts. As before, the R function uses C code to calculate the first n Fibonacci numbers iteratively, where a vector holds the last len numbers calculated. (To save space, we've removed R code that checks constraints on n and len). The simple .Call to fibonacci2 looks very natural. Input values n and len produce the output vector out, where the C code must somehow determine what out should be. Not surprisingly, it requires more complicated C code to make this happen.

The C function fibonacci2 (Table B3) first loads header files that include the required definitions from R. All input and output variables belong to type SEXP. Other internal variables have the standard C types double and int. Functions like INTEGER_VALUE() convert R types into C types. The SEXP vector retVals of return values is created by the R constructor NEW_NUMERIC() and then protected from garbage collection by PROTECT(). After all required variables are defined and type cast correctly, the iterative loop of calculations follows the earlier example in Table B2. Finally, the only protected vector retVals is released by UNPROTECT(1), and the standard closing command return retVals returns the output vector from fibonacci2.

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

Appendix C. *PBS Modelling* functions and data

This appendix documents the objects currently available in *PBS Modelling*, along with a list of function dependencies for exported functions and hidden "dot" functions. The latter are hidden through R's `NAMESPACE` but can be seen through the triple colon convention (e.g., `PBSmodelling:::.addslashes`). R also provides a function called `fixInNamespace()` for modifying `NAMESPACE` objects. The final section of this appendix details how a user can generate a standard R manual for *PBS Modelling*, that includes a Table of Contents, help pages for all objects, and an index. The manual itself is also appended.

C.1. Objects in *PBS Modelling*

`addArrows`	Add arrows to a plot using relative (0:1) coordinates
`addHistory`	Add current window settings to the current history record
`addLabel`	Add a label to a plot using relative (0:1) coordinates
`addLegend`	Add a legend to a plot using relative (0:1) coordinates
`backHistory`	Move back one step in the saved values for a history widget
`calcFib`	Calculate Fibonacci numbers by several methods
`calcGM`	Calculate the geometric mean, allowing for zeroes
`calcMin`	Calculate the minimum of user-defined function
`CCA.qbr`	Dataset: sampled counts of quillback rockfish (*Sebastes maliger*)
`chooseWinVal`	Choose and set a string item in a GUI
`clearAll`	Remove all R objects from the global environment
`clearHistory`	Clear saved values for a history widget
`clearWinVal`	Remove all current widget variables
`closeWin`	Close GUI window(s)
`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
`exportHistory`	Export a saved history
`findPat`	Search a character vector to find multiple patterns
`firstHistory`	Jump to the first history record
`focusWin`	Set the focus on a particular window
`forwHistory`	Move forward one step in the saved values for a history widget
`genMatrix`	Generate test matrices for `plotBubbles`
`getChoice`	Choose one string item from a list of choices
`getPBSext`	Get a command associated with a filename
`getPBSoptions`	Retrieve a user option
`getWinAct`	Retrieve the last window action
`getWinFun`	Retrieve names of functions referenced in a window
`getWinVal`	Retrieve widget values for use in R code
`GT0`	Restrict a numeric variable to a positive value

importHistory	Import a history list from a file
initHistory	Create structures for a new `history` widget
jumpHistory	Jump to a particular history record
lastHistory	Jump to the last history record
openFile	Open a file with the associated program
pad0	Pad numbers with leading zeroes
parseWinFile	Convert a window description file into a list object
pause	Pause between graphics displays or other calculations
pickCol	Pick a colour from a palette and get the hexadecimal code
plotACF	Plot autocorrelation bars from a data frame, matrix, or vector
plotAsp	Construct a plot with a specified aspect ratio
plotBubbles	Construct a bubble plot from a matrix
plotCsum	Plot cumulative sum of data
plotDens	Plot density curves from a data frame, matrix, or vector
plotTrace	Plot trace lines from a data frame, matrix, or vector
promptOpenFile	Display an "Open File" dialogue
promptSaveFile	Display a "Save File" dialogue
readList	Read a `list` from a file in *PBS Modelling* format
resetGraph	Reset `par` values for a plot
restorePar	Get actual parameters from scaled values
rmHistory	Remove a record from the history
runDemos	Run GUI to access demos from any R package installed
runExamples	Run GUI examples included with *PBS Modelling*
scalePar	Scale parameters to [0,1]
setPBSext	Set a command associated with a filename extension
setPBSoptions	Set a user option
setWinAct	Add a window action to the saved action vector
setWinVal	Update widget values
show0	Convert numbers into text with specified decimal places
showArgs	Display expected widget arguments
sortHistory	Sort the history records
testCol	Display named colours available based on a set of strings
testLty	Display line types available
testLwd	Display line widths
testPch	Display plotting symbols and backslash characters
testWidgets	Display sample GUIs and their source code
unpackList	Unpack `list` elements into variables
vbdata	Dataset: Length-at-age data for a von Bertalanffy curve
vbpars	Dataset: Initial parameters for a von Bertalanffy curve
view	Display first *n* rows of an object
writeList	Write a `list` to a file in *PBS Modelling* format

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

`.addslashes`	Escape special characters from a string
`.autoConvertMode`	Convert `x` into a numeric mode
`.buildgrid`	Attach child widgets to a grid
`.catError`	Display parsing errors
`.catError2`	Display parsing error (from C code)
`.convertMatrixListToDataFrame`	
	Convert a list into a data frame
`.convertMatrixListToMatrix`	
	Convert a list to a matrix (or a higher dimensional array)
`.convertMode`	Convert a variable into a mode without showing any warnings
`.convertPararmStrToList`	
	Convert a string representing a widget into a vector
`.convertPararmStrToVector`	
	Convert a string representing data into a vector
`.convertVecToArray`	Convert a vector to an array
`.createTkFont`	Creates a usable TK font from a given string
`.createWidget`	Call the appropriate sub-function (below) to create a given widget
`.createWidget.button`	
`.createWidget.check`	
`.createWidget.data`	
`.createWidget.entry`	
`.createWidget.grid`	
`.createWidget.history`	
`.createWidget.label`	
`.createWidget.matrix`	
`.createWidget.null`	
`.createWidget.object`	
`.createWidget.radio`	
`.createWidget.slide`	
`.createWidget.slideplus`	
`.createWidget.text`	
`.createWidget.vector`	
`.dClose`	Function to execute on closing `runDemos()`
`.extractData`	Receive events from TK, and extract data for `getWinAct`
`.extractFuns`	Extract a list of called functions
`.extractVar`	Extract values from the `tclvar ptrs` of a window
`.fibC`	Call Fibonacci C code via `C`
`.fibCall`	Call Fibonacci C code via `Call`
`.fibClosedForm`	Close form equation for Fibonacci numbers
`.fibR`	Calculate Fibonacci numbers in R using iteration
`.getArrayPts`	Return all possible indices of an array
`.getMatrixListSize`	Determine the minimum required size of the required array

`.getParamFromStr`	Convert a string representing a widget into a list including default values as defined in `widgetDefs.r`
`.inCollection`	Find a needle in a haystack (may be removed in future)
`.initPBSoptions`	Initialization function when `PBSmodelling` is loaded
`.isReallyNull`	Test if a key exists in a list
`.map.add`	Save a new value for a given key, if no current value is set
`.map.get`	Returns a value associated with a key
`.map.getAll`	Return all values of the map
`.map.init`	Initialize the data structure that holds the map(s); a map is another name for hash table (implemented using an R list)
`.map.set`	Save a value, even if a current one exists
`.mapArrayToVec`	Determine the index to use for a vector, given the indices for an element of a higher dimensional array
`.matrixHelp`	Store an element in matrix list (or a higher dimensional array list)
`.parsegrid`	Create a branch in the parse tree for children widgets of a grid
`.parsemenu`	Create a branch in the parse tree for children widgets of a menu
`.PBSdimnameHelper`	Add `dimnames` to an object
`.pFormatDefs`	A list defining accepted parameters (and default values) for `"P"` format of `readList` and `writeList`
`.readList.P`	Read a list in `P` format
`.readList.P.convertData`	Convert data into a proper mode
`.searchCollection`	Search a haystack for a needle, or a similar longer needle
`.setMatrixElement`	Assign values from a list into a matrix (or a higher dimensional array)
`.setWinValHelper`	Update widget values when `setWinVal` is called
`.sortActHistory`	Use window action as history name
`.sortHelper`	Helper function to sort history
`.sortHelperActive`	Helper function to sort history
`.sortHelperFile`	Help history with input from and output to an archive file
`.stopWidget`	Display fatal post-parsing errors and halt
`.stripComments`	Remove comments from a string
`.stripSlashes`	Removes escape backslashes from a string
`.stripSlashesVec`	Convert a grouping of strings representing an argument into a vector of strings
`.trimWhiteSpace`	Remove leading and trailing white space
`.updateHistory`	Update widget values
`.updateFile`	Coordinate file transfers
`.validateWindowDescList`	Check for a valid *PBS Modelling* description list and set any missing default values
`.validateWindowDescWidgets`	Validate a single widget
`.viewPkgDemo`	Display a GUI to display something equivalent to R's `demo()`
`.widgetDefs`	A list defining widget parameters and default values
`.writeList.P`	Saves a list to disk using the `"P"` format

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 NAMESPACE discussed in section B.3.

.addslashes

.autoConvertMode

.buildgrid
   .createTkFont
   .createWidget

.catError

.convertMatrixList
  ToDataFrame
  .getMatrixListSize
  .setMatrixElement

.convertMatrixList
  ToMatrix
  .getMatrixListSize
  .setMatrixElement

.convertMode

.convertPararmStr
  ToList
  .catError
  .trimWhiteSpace

.convertPararmStr
  ToVector
  .catError
  .trimWhiteSpace

.convertVecToArray
  .getArrayPts
  .mapArrayToVec

.createTkFont
  .convertPararmStr
    ToVector

.createWidget
  .isReallyNull

.createWidget.button
  .createTkFont
  .extractData

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

.createWidget.data
  .createWidget.grid
  .stopWidget

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

.createWidget.grid
  .buildgrid
  .createTkFont

.createWidget.history
  .createWidget.grid
  initHistory

.createWidget.label
  .createTkFont

.createWidget.matrix
  .createWidget.grid
  .stopWidget

.createWidget.null

.createWidget.object
  .createWidget

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

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

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

.createWidget.text
  .createTkFont
  .map.add

.createWidget.vector
  .createWidget.grid
  .stopWidget

.dClose
  getWinAct
  closeWin

.extractData
  setWinAct

.extractFuns

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

.fibC

.fibCall

.fibClosedForm

.fibR

.getArrayPts

.getMatrixListSize
  .getMatrixListSize

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

.inCollection

.initPBSoptions

.isReallyNull

.map.add
  .isReallyNull
  .map.init

.map.get

.map.getAll

.map.init

.map.set
   .isReallyNull
   .map.init

.mapArrayToVec

.matrixHelp
   .matrixHelp

.parsegrid
   .parsegrid

.parsemenu
   .parsemenu

.PBSdimnameHelper

.readList.P
   .catError
   .readList.P.convert
     Data
   .stripComments
   .trimWhiteSpace

.readList.P.convertData
   .autoConvertMode
   .catError
   .convertMode
   .convertPararmStr
     ToVector
   .convertVecToArray
   .getParamFromStr

.searchCollection

.setMatrixElement
   .setMatrixElement

.setWinValHelper
   .map.get
   .setWinValHelper

.sortActHistory
   getWinAct
   sortHistory

.sortHelper
   getWinAct
   getWinVal
   .sortHelperActive
   .sortHelperFile
   sortHistory

.sortHelperActive
   .updateHistory

.sortHelperFile
   readList
   writeList

.stopWidget

.stripComments
   .stripComments

.stripSlashes
   .catError

.stripSlashesVec
   .catError

.trimWhiteSpace

.updateFile
   getWinAct
   getWinVal
   promptOpenFile
   promptSaveFile
   setWinVal

.updateHistory
   setWinVal

.validateWindowDescList
   .validateWindow
     DescWidgets

.validateWindow
  DescWidgets

.viewPkgDemo
   getWinAct
   getWinVal
   openFile
   runDemos

.writeList.P
   .addslashes

addArrows

addLabel

addLegend

addHistory
    .updateHistory
   getWinAct
   getWinVal

backHistory
    .updateHistory
   getWinAct
   setWinVal

calcFib
    .fibC
    .fibCall
    .fibClosedForm
    .fibR

calcGM

calcMin
   restorePar
   scalePar
   show0

chooseWinVal
   getChoice
   setPBSoptions
   setWinVal

clearAll

clearHistory
    .updateHistory
   getWinAct
   rmHistory

clearWinVal
   getWinVal

closeWin
    .isReallyNull

compileDescription
   parseWinFile
   writeList

createVector
   createWin

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

drawBars

expandGraph

exportHistory
   getWinAct
   promptSaveFile
   writeList

findPat

focusWin

forwHistory
    .updateHistory
   getWinAct
   setWinVal

genMatrix

getChoice
   createWin
   focusWin
   getPBSoptions
   setPBSoptions

getPBSext
    .isReallyNull

getPBSoptions

getWinAct

getWinFun

getWinVal
    .extractVar
    .isReallyNull

GT0

importHistory
    .updateHistory
   getWinAct
   promptOpenFile
   readList

initHistory

jumpHistory
    .updateHistory
   getWinAct
   getWinVal
   setWinVal

openFile
    .initPBSoptions
    .isReallyNull
   getPBSext
   getWinAct
   openFile

pad0

parseWinFile
    .getParamFromStr
    .parsegrid
    .parsemenu
    .stripComments
    .trimWhiteSpace

pause

pickCol

plotACF

plotAsp

plotBubbles

plotCsum
   addLabel
   resetGraph

plotDens

plotTrace

promptOpenFile
    .trimWhiteSpace

promptSaveFile
   promptOpenFile

readList
    .readList.P

resetGraph

restorePar

rmHistory
    .updateHistory
   getWinAct
   setWinVal

C.3. *PBS Modelling* manual

The following pages show the standard R manual for *PBS Modelling*, including help pages for all objects, a table of contents, and an index. This manual also appears on the CRAN web site:

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

(Or from CRANS's root, locate "Packages" and find "PBSmodelling".)

To generate the pages that follow, the user should first ensure that R's style and font files have been copied to MiKTeX (see steps 5a-c in Section B1). This enhancement is essential for the successful creation of a PDF manual.

Next we provide a choice of two methods that use the batch files `makePDF.bat` and `makePDF2.bat` to assist the user in building the manual. The first method alters a temporary TEX file *after* R's Perl script is run, and the PDF is built by calling MiKTeX commands. The second method modifies R's Perl script *before* it builds the TEX and PDF files. The final result of both methods yields a manual with letter ($8.5'' \times 11''$) rather than A4 paper, and renumbering beginning on a specified page. This page number should be odd so that the next page becomes the front of a two-sided copy. Although the first method requires a redundant build of the document, it is possibly more robust to future changes in R's Perl script.

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

```
makePDF PBSmodelling 67
```

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

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

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

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

```
\nonstopmode{}
\documentclass[letter]{book}
\usepackage[times,hyper]{Rd}
\usepackage{makeidx}
\makeindex{}
\begin{document}
\setcounter{page}{67}
```

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

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

(The second call to `latex` might not be needed, but it resolves a number of references. The `makeindex` command creates the table of contents.) You should now have the PDF manual called `PBSmodelling.pdf`, which can be appended to the first 66 pages of this report.

Method 2: On a command line, type the command:

```
makePDF2 PBSmodelling 67
```

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

Essentially the script in `makePDF2.bat` modifies R's `Rd2dvi.sh` Perl script and saves it to the file `Rd2dvi4pbs.sh`, which sits in R's `bin\` directory. The batch file then issues the command:

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

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

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

```
rm -rf .Rd2dvi
```

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

Package 'PBSmodelling'

August 28, 2007

Version 1.58

Date 2007-08-28

Title PBS Modelling

Author Jon T. Schnute <SchnuteJ@pac.dfo-mpo.gc.ca>, Alex Couture-Beil <alex@mofo.ca>, and Rowan Haigh <HaighR@pac.dfo-mpo.gc.ca>

Maintainer Jon Schnute <SchnuteJ@pac.dfo-mpo.gc.ca>

Depends R (>= 2.3.0)

Suggests PBSmapping, odesolve, BRugs

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, ddesolvem, 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.

License GPL version 2 or newer

R topics documented:

`CCA.qbr` *Dataset: 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)
```

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 2007).

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. 2007. Compositional analysis of catch curve data with an application to *Sebastes maliger*. ICES Journal of Marine Science (in press).

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", package="PBSmodelling")
z <- CCA.qbr; 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)
```

GT0   *Restrict a Numeric Variable to a Positive Value*

Description

Restrict a numeric value x to a positive value using a differentiable function. GT0 stands for "greater than zero".

Usage

```
GT0(x,eps=1e-4)
```

Arguments

x	vector of values
eps	minimum value greater than zero.

Details

```
if (x >= eps)..........GT0 = x
if (0 < x < eps).......GT0 = (eps/2) * (1 + (x/eps)^2)
if (x <= 0)............GT0 = eps/2
```

See Also

scalePar, restorePar, calcMin

Examples

```
plotGT0 <- function(eps=1,x1=-2,x2=10,n=1000,col="black") {
        x <- seq(x1,x2,len=n); y <- GT0(x,eps);
        lines(x,y,col=col,lwd=2); invisible(list(x=x,y=y)); }

testGT0 <- function(eps=c(7,5,3,1,.1),x1=-2,x2=10,n=1000) {
        x <- seq(x1,x2,len=n); y <- x;
        plot(x,y,type="l");
        mycol <- c("red","blue","green","brown","violet","orange","pink");
        for (i in 1:length(eps))
                plotGT0(eps=eps[i],x1=x1,x2=x2,n=n,col=mycol[i]);
        invisible(); };

testGT0()
```

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.

PBSmodelling contains examples that illustrate models built uisng other R packages, including PBSmapping, odesolve, ddesolve, 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 interesting examples accessed through the function runExamples(). Additionally, users can view *PBS Modelling* widgets through the function testWidgets(). More generally, a user can run any available demos in his/her locally installed packages through the function runDemos().

---

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

Description

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

Usage

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

Arguments

x1	x-coordinate (0:1) at base of arrow.
y1	y-coordinate (0:1) at base of arrow.
x2	x-coordinate (0:1) at tip of arrow.
y2	y-coordinate (0:1) at tip of arrow.
...	additional paramaters for the function `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 a Label to a Plot Using Relative (0:1) Coordinates*

Description

Place 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 the function text.

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

Description

Place 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 the function legend, such as lines, text, or rectangle.

See Also

addArrows, addLabel

Examples

```
resetGraph(); n <- sample(1:length(colors()),15); clrs <- colors()[n]
addLegend(.2,1,fill=clrs,leg=clrs,cex=1.5)
```

| calcFib | *Calculate Fibonacci Numbers by Several Methods* |

Description

Compute 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, Allowing for Zeroes* |

Description

Calculate the geometric mean of a numeric vector, possibly excluding zeroes 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 TRUE, 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)
```

---

calcMin	*Calculate the Minimum of a User-Defined Function*

Description

Minimization based on the R-stat functions nlm, nlminb, and optim. Model parameters are scaled and can be active or not in the minimization.

Usage

```
calcMin(pvec, func, method="nlm", trace=0, maxit=1000, reltol=1e-8,
        steptol=1e-6, temp=10, repN=0, ...)
```

Arguments

pvec	Initial values of the model parameters to be optimized. pvec is a data frame comprising four columns ("val","min","max","active") and as many rows as there are model parameters. The "active" field (logical) determines whether the parameters are estimated (T) or remain fixed (F).
func	The user-defined function to be minimized (or maximized). The function should return a scalar result.
method	The minimization method to use: one of nlm, nlminb, Nelder-Mead, BFGS, CG, L-BFGS-B, or SANN. Default is nlm.
trace	Non-negative integer. If positive, tracing information on the progress of the minimization is produced. Higher values may produce more tracing information: for method "L-BFGS-B" there are six levels of tracing. Default is 0.
maxit	The maximum number of iterations. Default is 1000.
reltol	Relative convergence tolerance. The algorithm stops if it is unable to reduce the value by a factor of reltol * (abs(val) + reltol) at a step. Default is 1e-8.
steptol	A positive scalar providing the minimum allowable relative step length. Default is 1e-6.
temp	Temperature controlling the "SANN" method. It is the starting temperature for the cooling schedule. Default is 10.
repN	Reports the parameter and objective function values on the R-console every repN evaluations. Default is 0 for no reporting.
...	Further arguments to be passed to the optimizing function chosen: nlm, nlminb, or optim. Beware of partial matching to earlier arguments.

Details

See `optim` for details on the following methods: `Nelder-Mead`, `BFGS`, `CG`, `L-BFGS-B`, and `SANN`.

Value

A list with components:

`Fout`	The output list from the optimizer function chosen through `method`.
`iters`	Number of iterations.
`evals`	Number of evaluations.
`cpuTime`	The user CPU time to execute the minimization.
`elapTime`	The total elapsed time to execute the minimization.
`fminS`	The objective function value calculated at the start of the minimization.
`fminE`	The objective function value calculated at the end of the minimization.
`Pstart`	Starting values for the model parameters.
`Pend`	Final values estimated for the model parameters from the minimization.
`AIC`	Akaike's Information Criterion
`message`	Convergence message from the minimization routine.

Note

Some arguments to `calcMin` have no effect depending on the `method` chosen.

See Also

scalePar, restorePar, calcMin, GT0
In the `stats` package: nlm, nlminb, and optim.

Examples

```
Ufun <- function(P) {
        Linf <- P[1]; K <- P[2]; t0 <- P[3]; obs <- afile$len;
        pred <- Linf * (1 - exp(-K*(afile$age-t0)));
        n <- length(obs); ssq <- sum((obs-pred)^2 );
        return(n*log(ssq)); };
afile <- data.frame(age=1:16,len=c(7.36,14.3,21.8,27.6,31.5,35.3,39,
        41.1,43.8,45.1,47.4,48.9,50.1,51.7,51.7,54.1));
pvec <- data.frame(val=c(70,0.5,0),min=c(40,0.01,-2),max=c(100,2,2),
        active=c(TRUE,TRUE,TRUE),row.names=c("Linf","K","t0"),
        stringsAsFactors=FALSE);
alist <- calcMin(pvec=pvec,func=Ufun,method="nlm",steptol=1e-4,repN=10);
print(alist[-1]); P <- alist$Pend;
resetGraph(); expandGraph();
xnew <- seq(afile$age[1],afile$age[nrow(afile)],len=100);
ynew <- P[1] * (1 - exp(-P[2]*(xnew-P[3])) );
plot(afile); lines(xnew,ynew,col="red",lwd=2);
addLabel(.05,.88,paste(paste(c("Linf","K","t0"),round(P,c(2,4,4)),
        sep=" = "),collapse="\n"),adj=0,cex=0.9);
```

---

chooseWinVal *Choose and Set a String Item in a GUI*

Description

Prompts the user to choose one string item from a list of choices displayed in a GUI, then sets a specified variable in a target GUI.

Usage

```
chooseWinVal(choice, varname, winname="window")
```

Arguments

choice   vector of strings from which to choose

varname   variable name to which `choice` is assigned in the target GUI

winname   window name for the target GUI

Details

`chooseWinVal` activates a `setWinVal` command through an `onClose` function created by the `getChoice` command and modified by `chooseWinVal`.

Value

No value is returned directly. The choice is written to the PBS options workspace, accessible through `getPBSoptions("getChoice")`. Also set in PBS options is the window name from which the choice was activated.

Note

Microsoft Windows users may experience difficulties switching focus between the R console and GUI windows. The latter frequently disappear from the screen and need to be reselected (either clicking on the task bar or pressing <Alt><Tab>. This issue can be resolved by switching from MDI to SDI mode. From the R console menu bar, select <Edit> and <GUI preferences>, then change the value of "single or multiple windows" to SDI.

See Also

getChoice, getWinVal, setWinVal

Examples

```
## Not run:
dfnam <-
        c("airquality","attitude","ChickWeight","faithful","freeny",
        "iris","LifeCycleSavings","longley","morley","Orange",
        "quakes","randu","rock","stackloss","swiss","trees")

wlist <- c(
        "window name=choisir title=\"Test chooseWinVal\"",
        "label text=\"Press <ENTER> in the green entry box
        \nto choose a file, then press <GO>\" sticky=W pady=5",
        "grid 1 3 sticky=W",
        "label text=File: sticky=W",
        "entry name=fnam mode=character width=23 value=\"\"
        func=chFile entrybg=darkolivegreen1 pady=5",
        "button text=GO bg=green sticky=W func=test",
        "")

chFile <- function(ch=dfnam,fn="fnam")
        {chooseWinVal(ch,fn,winname="choisir")};

#-- Example 1 GUI test
test <- function() {
        getWinVal(winName="choisir",scope="L")
        if (fnam!="" && any(fnam==dfnam)) {
                file <- get(fnam);
                pairs(file,gap=0); }
        else {
                resetGraph();
                addLabel(.5,.5,"Press <ENTER> in the green entry box
                \nto choose a file, then press <GO>", col="red",cex=1.5)}};

#-- Example 2 Non-GUI test
#To try the non-GUI version, type 'test2()' on the command line
test2 <- function(fnames=dfnam) {
  frame();resetGraph()
  again <- TRUE;
  while (again) {
    fnam <- sample(fnames,1); file <- get(fnam);
    flds <- names(file);
    xfld <- getChoice(paste("Pick x-field from",fnam),flds,gui=F);
    yfld <- getChoice(paste("Pick y-field from",fnam),flds,gui=F)
    plot(file[,xfld],file[,yfld],xlab=xfld,ylab=yfld,
      pch=16,cex=1.2,col="red");
    again <- getChoice("Plot another pair?",gui=F) }
  }
require(PBSmodelling)
createWin(wlist,astext=T); test();
## End(Not run)
```

---

clearAll   *Remove all R Objects From the Global Environment*

Description

Generic function to clear .RData in R

Usage

```
clearAll(hidden=TRUE, verbose=TRUE)
```

Arguments

hidden   if TRUE, remove variables that start with a dot (.)

verbose   if TRUE, report all removed items

---

clearWinVal   *Remove all Current Widget Variables*

Description

Remove all global variables that share a name in common with any widget variable name defined in names(getWinVal()). Use this function with caution.

Usage

```
clearWinVal()
```

See Also

[getWinVal](#)

---

closeWin *Close GUI Window(s)*

Description

Close (destroy) one or more windows made with `createWin`.

Usage

```
closeWin(name)
```

Arguments

name   a vector of window names that indicate which windows to close. These names
        appear in the Window Description File(s) on the line(s) defining WINDOW wid-
        gets. If `name` is ommitted, all active windows will be closed.

See Also

createWin

---

compileDescription *Convert and Save a Window Description as a List*

Description

Convert a Window Description File (ASCII markup file) to an equivalent Window Description List.
The output list (an ASCII file containing R-source code) is complete, i.e., all default values have
been added.

Usage

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

Arguments

descFile   file name of markup file.
outFile   file name of output file containing R source code.

Details

The Window Description File `descFile` is converted to a list, which is then converted to R code,
and saved to `outFile`.

See Also

parseWinFile, createWin

---

createVector *Create a GUI with a Vector Widget*

Description

Create 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. The values in vec become the default values for the widget. If vec is named, the names are used as the variable names.
vectorLabels	an optional vector of strings to use as labels above each widget.
func	string name of function to call when new data are entered in widget boxes or when "GO" is pressed.
windowname	unique window name, required if multiple vector windows are created.

See Also

createWin

Examples

```
## Not run:
#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");
## End(Not run)
```

---

createWin   *Create a GUI Window*

Description

Create a GUI window with widgets using instructions from a Window Description (markup) File.

Usage

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

Arguments

fname   file name of Window Description File or list returned from parseWinFile.

astext   logical; if TRUE, interpret fname as a vector of strings with each element representing a line in a Window Description File.

Details

Generally, the markup file contains a single widget per line. However, widgets can span multiple lines by including a backslash ('\') character at the end of a line, prompting the suppression of the newline character.

For more details on widget types and markup file, see "PBSModelling-UG.pdf" in the installation directory.

It is possible to use a Window Description List produced by compileDescription rather than a file name for fname.

Another alternative is to pass a vector of characters to fname and set astext=T. This vector represents the file contents where each element is equivalent to a new line in the Window Description File.

Note

Microsoft Windows users may experience difficulties switching focus between the R console and GUI windows. The latter frequently disappear from the screen and need to be reselected (either clicking on the task bar or pressing <Alt><Tab>. This issue can be resolved by switching from MDI to SDI mode. From the R console menu bar, select <Edit> and <GUI preferences>, then change the value of "single or multiple windows" to SDI.

See Also

parseWinFile, getWinVal, setWinVal

closeWin, compileDescription, createVector

initHistory for an example of using astext=TRUE

Examples

```
## Not run:
#see file testWidgets\LissWin.txt in PBSmodelling package directory

# 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); };

createWin(system.file("testWidgets/LissWin.txt",package="PBSmodelling"));
## End(Not run)
```

---

drawBars   *Draw a Linear Barplot on the Current Plot*

Description

Draw a linear barplot on the current plot.

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 the Plot Area by Adjusting Margins*

Description

Optimize the plotting region(s) 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](#)

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")
```

---

exportHistory   *Export a Saved History*

Description

Export the current history list.

Usage

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

Arguments

hisname	name of the history list to export. If set to `""`, the value from `getWinAct()[1]` will be used instead.
fname	file name where history will be saved. If it is set to `""`, a **Save As** window will be displayed.

See Also

importHistory, initHistory, promptSaveFile

---

findPat	*Search a Character Vector to Find Multiple Patterns*

Description

Use all available patterns in `pat` to search in `vec`, and return 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"))
```

---

focusWin   *Set the Focus on a Particular Window*

Description

Bring the specified window into focus, and set it as the active window. `focusWin` will fail to bring the window into focus if it is called from the R console, since the R console returns focus to itself once a function returns. However, it will work if `focusWin` is called as a result of calling a function from the GUI window. (i.e., pushing a button or any other widget that has a function argument).

Usage

```
focusWin(winName, winVal=TRUE)
```

Arguments

winName   name of window to focus

winVal   if TRUE, associate winName with the default window for setWinVal and
             getWinVal

Examples

```
## Not run:
focus <- function() {
  winName <- getWinVal()$select;
  focusWin(winName);
  cat("calling focusWin(\"", winName, "\")\n", sep="");
  cat("getWinVal()$myvar = ", getWinVal()$myvar, "\n\n", sep=""); };

#create three windows named win1, win2, win3
#each having three radio buttons, which are used to change the focus
for(i in 1:3) {
  winDesc <- c(
    paste('window name=win',i,' title="Win',i,'"', sep=''),
    paste('entry myvar ', i, sep=''),
    'radio name=select value=win1 text="one" function=focus mode=character',
    'radio name=select value=win2 text="two" function=focus mode=character',
    'radio name=select value=win3 text="three" function=focus mode=character');
  createWin(winDesc, astext=TRUE); };
## End(Not run)
```

---

genMatrix   *Generate Test Matrices for plotBubbles*

Description

Generate a test matrix of random numbers (`mu` = mean and `signa` = standard deviation), 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))
```

---

getChoice   *Choose One String Item from a List of Choices*

Description

Prompts the user to choose one string item from a list of choices displayed in a GUI. The simplest case `getChoice()` yields `TRUE` or `FALSE`.

Usage

```
getChoice(question="Make a choice: ", choice=c("Yes","No"),
          winname="getChoice", horizontal=TRUE, radio=FALSE,
          qcolor="blue", gui=TRUE, quiet=FALSE)
```

Arguments

question	question or prompting statement
choice	vector of strings from which to choose
winname	window name for the getChoice GUI
horizontal	if TRUE, display the choices horizontally, else vertically
radio	if TRUE, display the choices as radio buttons, else as buttons
qcolor	colour for question
gui	if TRUE, getChoice is functional when called from a GUI, else it is functional from command line programs
quiet	if TRUE, don't print the choice on the command line

Details

The user's choice is stored in .PBSmod$options$getChoice (or whatever winname is supplied).

getChoice generates an onClose function that returns focus to the calling window (if applicable) and prints out the choice.

Value

If called from a GUI (gui=TRUE), no value is returned directly. Rather, the choice is written to the PBS options workspace, accessible through getPBSoptions("getChoice") (or whatever winname was supplied).

If called from a command line program (gui=FASLE), the choice is returned directly as a string scalar (e.g., answer <- getChoice(gui=F)).

Note

Microsoft Windows users may experience difficulties switching focus between the R console and GUI windows. The latter frequently disappear from the screen and need to be reselected (either clicking on the task bar or pressing <Alt><Tab>. This issue can be resolved by switching from MDI to SDI mode. From the R console menu bar, select <Edit> and <GUI preferences>, then change the value of "single or multiple windows" to SDI.

See Also

chooseWinVal, getWinVal, setWinVal

Examples

```
## Not run:
#-- Example 1
getChoice("What do you want?", c("Fame","Fortune","Health",
   "Beauty","Lunch"),qcolor="red",gui=F)

#-- Example 2
getChoice("Who`s your daddy?",c("Homer Simpson","Pierre Trudeau",
```

```
    "Erik the Red"),horiz=F,radio=T,gui=F)
## End(Not run)
```

---

getPBSext   *Get a Command Associated With a Filename*

Description

Display all locally defined file extensions and their associated commands, or search for the command associated with a specific file extension `ext`.

Usage

```
getPBSext(ext)
```

Arguments

ext   optional string specifying a file extension (suffix)

Value

Command associated with file extension.

Note

These file associations are not saved from one *PBS Modelling* session to the next.

See Also

[setPBSext](#), [openFile](#)

---

getPBSoptions   *Retreive A User Option*

Description

Get a previously defined user option.

Usage

```
getPBSoptions(option)
```

Arguments

option   name of option to retrieve. If omitted, a list containing all options is returned.

Value

Value of the specified option, or NULL if the specified option is not found.

See Also

getPBSext

---

getWinAct   *Retreive the Last Window Action*

Description

Get a string vector of actions (latest to earliest).

Usage

```
getWinAct(winName)
```

Arguments

winName   name of window to retrieve action from

Details

When a function is called from a GUI, a string descriptor associated with the action of the function is stored internaly (appended to the first position of the action vector). A user can utilize this action as a type of argument for programming purposes. The command getWinAct()[1] yields the latest action.

Value

String vector of recorded actions (latest first).

---

getWinFun   *Retrieve Names of Functions Referenced in a Window*

Description

Get a vector of all function names referenced by a window.

Usage

```
getWinFun(winName)
```

Arguments

winName   name of window, to retrieve its function list

Value

A vector of function names referenced by a window.

---

getWinVal   *Retreive Widget Values for Use in R Code*

Description

Get a list of variables defined and set by the GUI widgets. An optional argument scope directs the function to create local or global variables based on the list that is returned.

Usage

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

Arguments

v   vector of variable names to retrieve from the GUI widgets. If NULL, v retrieves all variables from all GUI widgets.

scope   scope of the retrieval. The default sets no variables in the non-GUI environment; scope="L" creates variables locally in relation to the parent frame that called the function; and scope="G" creates global variables(pos=1).

asvector   return a vector instead of a list. WARNING: if a widget variable defines a true vector or matrix, this will not work.

winName   window from which to select GUI widget values. The default takes the window that has most recently received new user input.

Value

A list (or vector) with named components, where names and values are defined by GUI widgets.

See Also

parseWinFile, setWinVal, clearWinVal

---

importHistory   *Import a History List from a File*

Description

Import a history list from file fname, and place it into the history list hisname.

Usage

```
importHistory(hisname="", fname="", updateHis=TRUE)
```

Arguments

hisname	name of the history list to be populated. The default ("") uses the value from getWinAct()[1].
fname	file name of history file to import. The default ("") causes an open-file window to be displayed.
updateHis	if true, update the history widget to reflect the change in size and index.

See Also

exportHistory, initHistory, promptOpenFile

---

initHistory   *Create Structures of a New History Widget*

Description

PBS history functions (below) are available to those who would like to use the package's history functionality, without using the pre-defined history widget. These functions allow users to create customized history widgets.

Usage

```
initHistory(hisname,indexname=NULL,sizename=NULL,modename=NULL,func=NULL,overwrite=
rmHistory(hisname="", index="")
addHistory(hisname="")
forwHistory(hisname="")
backHistory(hisname="")
lastHistory(hisname="")
firstHistory(hisname="")
jumpHistory(hisname="", index="")
clearHistory(hisname="")
```

Arguments

hisname	name of the history "list" to manipulate. If it is omitted, the function uses the value of getWinAct()[1] as the history name. This allows the calling of functions directly from the Window Description File (except initHistory, which must be called before createWin()).
indexname	name of the index entry widget in the Window Description File. If NULL, then the current index feature will be disabled.
sizename	name of the current size entry widget. If NULL, then the current size feature will be disabled.
modename	name of the radio widgets used to change addHistory's mode. If NULL, then the default mode will be to insert after the current index.
index	index to the history item. The default ("") causes the value to be extracted from the widget identified by indexname.
func	name of user supplied function to call when viewing history items.
overwrite	if TRUE, history (matching hisname) will be cleared. Otherwise, the imported history will be merged with the current one.

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 Window Description File. However, PBS Modelling includes support functions (above) 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 jumpHistory to retrieve a target index.

See Also

importHistory, exportHistory

Examples

```
## Not run:
# Example of creating a custom history widget that saves values
# whenever the "Plot" button is pressed. The user can tweak the
# inputs "a", "b", and "points" before each "Plot" and see the
# "Index" increase. After sufficient archiving, the user can review
# scenarios using the "Back" and "Next" buttons.
# A custom history is needed to achieve this functionality since
# the packages pre-defined history widget does not update plots.
```

```
# To start, create a Window Description to be used with createWin
# using astext=TRUE. P.S. Watch out for special characters which
# must be "escaped" 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 to vector with each line represented as a new element
winDesc <- strsplit(winDesc, "\n")[[1]]

# Custom functions to update plots after restoring history values
myHistoryBack <- function() {
        backHistory("myHistory");
        myPlot(saveVal=FALSE); # show the plot with saved values
}
myHistoryForw <- function() {
        forwHistory("myHistory");
        myPlot(saveVal=FALSE); # show the plot with saved values
}
myPlot <- function(saveVal=TRUE) {
        # save all data whenever plot is called (directly)
        if (saveVal) addHistory("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);
}

initHistory("myHistory", "myHistoryIndex", "myHistorySize")
createWin(winDesc, astext=TRUE)
## End(Not run)
```

---

openFile   *Open a File with the Associated Program*

Description

Open a file using the program associated with its extension defined by the Windows shell. Non-windows users, or users wishing to overide the default application, can specify a program associa-

tion using `setPBSext`.

Usage

```
openFile(fname)
```

Arguments

fname	file name of file to open.

Note

If a command is registered with `setPBSext`, then `openFile` will replace all occurrences of `"%f"` with the absolute path of the filename, before executing the command.

See Also

`getPBSext`, `setPBSext`

Examples

```
## Not run:
# Set up firefox to open .html files
setPBSext("html", '"c:/Program Files/Mozilla Firefox/firefox.exe" file://%f')
openFile("foo.html")
## End(Not run)
```

`pad0`	*Pad Numbers with Leading Zeroes*

Description

Convert numbers to integers then text, and pad them with leading zeroes.

Usage

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

Arguments

x	vector of numbers
n	number of text characters representing a padded integer
f	factor of 10 transformation on x before padding

Value

A character vector representing x with leading zeroes.

Examples

```
resetGraph(); x <- pad0(x=123,n=10,f=0:7);
addLabel(.5,.5,paste(x,collapse="\n"),cex=1.5);
```

parseWinFile	*Convert a Window Description File into a List Object*

Description

Parse a Window Description (markup) File into the list format expected by createWin().

Usage

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

Arguments

fname	file name of the Window Description File.
astext	if TRUE, fname is interpreted as a vector of strings, with each element representing a line of code in a Window Description File.

Value

A list representing a parsed Window Description File that can be directly passed to createWin.

Note

All widgets are forced 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 Between Graphics Displays or Other Calculations*

Description

Pause, typically between graphics displays. Useful for demo purposes.

Usage

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

Arguments

s   text issued on the command line when `pause` is invoked.

---

pickCol *Pick a Colour From a Palette and get the Hexadecimal Code*

Description

Display an interactive colour palette from which the user can choose a colour.

Usage

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

Arguments

returnValue   If `TRUE`, display the full colour palette, choose a colour, and return the hex value to the R session. If `FALSE`, use an intermediate GUI to interact with the palette and display the hex value of the chosen colour.

Value

A hexidecimal colour value.

See Also

[testCol](#)

Examples

```
## Not run:
junk<-pickCol(); resetGraph(); addLabel(.5,.5,junk,cex=4,col=junk);
## End(Not run)
```

---

plotACF   *Plot Autocorrelation Bars From a data frame, matrix, or vector*

Description

Plot autocorrelation bars (ACF) from a data frame, matrix, or vector.

Usage

```
plotACF(file, lags=20,
        clrs=c("blue","red","green","magenta","navy"), ...)
```

Arguments

file	data frame, matrix, or vector of numeric values.
lags	maximum number of lags to use in the ACF calculation.
clrs	vector of colours. Patterns are repeated if the number of fields exceeed the length of `clrs`.
...	additional arguments for `plot` or `lines`.

Details

This function is designed primarily to give greater flexibility when viewing results from the R-package `BRugs`. Use `plotACF` in conjuction with `samplesHistory("*",beg=0,plot=FALSE)` rather than `samplesAutoC` which calls `plotAutoC`.

Examples

```
resetGraph(); plotACF(trees,lwd=2,lags=30);
```

---

plotAsp   *Construct a Plot with a Specified Aspect Ratio*

Description

Plot `x` and `y` coordinates using a specified aspect ratio.

Usage

```
plotAsp(x, y, asp=1, ...)
```

Arguments

x	vector of x-coordinate points in the plot.
y	vector of y-coordinate points in the plot.
asp	y/x aspect ratio.
...	additional arguments for `plot`.

Details

The function `plotAsp` differs from `plot(x,y,asp=1)` in the way axis limits are handled. Rather than expand the range, `plotAsp` expands the margins through padding to keep the aspect ratio accurate.

Examples

```
x <- seq(0,10,0.1)
y <- sin(x)
par(mfrow=2:1)
plotAsp(x,y,asp=1,xlim=c(0,10),ylim=c(-2,2), main="sin(x)")
plotAsp(x,y^2,asp=1,xlim=c(0,10),ylim=c(-2,2), main="sin^2(x)")
```

---

plotBubbles	*Construct a Bubble Plot from a Matrix*

Description

Construct a bubble plot for a matrix `z`.

Usage

```
plotBubbles(z, xval=FALSE, yval=FALSE, dnam=FALSE, rpro=FALSE,
   cpro=FALSE, rres=FALSE, cres=FALSE, powr=1, size=0.2, lwd=2,
   clrs=c("black","red","blue"), hide0=FALSE, debug=FALSE, ...)
```

Arguments

z	input matrix
xval	x-values for the columns of `z`. if `xval=TRUE`, the first row contains x-values for the columns.
yval	y-values for the rows of `z`. If `yval=TRUE`, the first column contains y-values for the rows.
dnam	logical; if `TRUE`, attempt to use `dimnames` of input matrix `z` as `xval` and `yval`. The `dimnames` are converted to numeric values and must be strictly inreasing or decreasing. If successful, these values will overwrite previously specified values of `xval` and `yval` or any default indices.
rpro	logical; if `TRUE`, convert rows to proportions.

cpro	logical; if TRUE, convert columns to proportions.
rres	logical; if TRUE, use row residuals (subtract row means).
cres	logical; if TRUE, use column residuals (subtract column means).
powr	power transform. Radii are proportional to z^powr. Note: powr=0.5 yields bubble areas proportional to z.
size	size (inches) of the largest bubble.
lwd	line width for drawing circles.
clrs	colours (3-element vector) used for positive, negative, and zero values, respectively.
hide0	logical; if TRUE, hide zero-value bubbles.
debug	logical; if TRUE, display debug information.
...	additional arguments for symbols function.

Details

The function plotBubbles essentially flips the z matrix visually. The columns of z become the x-values while the rows of z become the y-values, where the first row is displayed as the bottom y-value and the last row is displayed as the top y-value. The function's original intention was to display proportions-at-age vs. year.

See Also

genMatrix

Examples

```
plotBubbles(round(genMatrix(40,20),0),clrs=c("green","grey","red"));

data(CCA.qbr)
plotBubbles(CCA.qbr,cpro=TRUE,powr=.5,dnam=TRUE,size=.15,
    ylim=c(0,70),xlab="Year",ylab="Quillback Rockfish Age")
```

---

plotCsum	*Plot Cumulative Sum of Data*

Description

Plot the cumulative frequency of a data vector or matrix, showing the median and mean of the distribution.

Usage

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

Arguments

x	vector or matrix of numeric values.
add	logical; if TRUE, add the cumulative frequency curve to a current plot.
ylim	limits for the y-axis.
xlab	label for the x-axis.
ylab	label for the y-axis.
...	additional arguments for the plot function.

Examples

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

---

plotDens	*Plot Density Curves from a data frame, matrix, or vector*

Description

Plot the density curves from a data frame, matrix, or vector. The mean density curve of the data combined is also shown.

Usage

```
plotDens(file, clrs=c("blue","red","green","magenta","navy"), ...)
```

Arguments

file	data frame, matrix, or vector of numeric values.
clrs	vector of colours. Patterns are repeated if the number of fields exceeed the length of clrs.
...	additional arguments for plot or lines.

Details

This function is designed primarily to give greater flexibility when viewing results from the R-package BRugs. Use plotDens in conjuction with samplesHistory("*",beg=0,plot=FALSE) rather than samplesDensity which calls plotDensity.

Examples

```
z <- data.frame(y1=rnorm(50,sd=2),y2=rnorm(50,sd=1),y3=rnorm(50,sd=.5))
plotDens(z,lwd=3)
```

---

plotTrace   *Plot Trace Lines from a data frame, matrix, or vector*

Description

Plot trace lines from a data frame or matrix where the first field contains x-values, and subsequent
fields give y-values to be traced over x. If input is a vector, this is traced over the number of
observations.

Usage

```
plotTrace(file, clrs=c("blue","red","green","magenta","navy"), ...)
```

Arguments

file   data frame or matrix of x and y-values, or a vector of y-values.

clrs   vector of colours. Patterns are repeated if the number of traces (y-fields) exceeed
                the length of `clrs`.

...   additional arguments for `plot` or `lines`.

Details

This function is designed primarily to give greater flexibility when viewing results from the R-
package `BRugs`. Use `plotTrace` in conjuction with `samplesHistory("*",beg=0,plot=FALSE)`
rather than `samplesHistory` which calls `plotHistory`.

Examples

```
z <- data.frame(x=1:50,y1=rnorm(50,sd=3),y2=rnorm(50,sd=1),y3=rnorm(50,sd=.25))
plotTrace(z,lwd=3)
```

---

promptOpenFile   *Display an "Open File" Dialogue*

Description

Display the default **Open** prompt provided by the Operating System.

Usage

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

Arguments

`initialfile`	file name of the text file containing the list.
`filetype`	a list of character vectors indicating file types made available to users of the GUI. Each vector is of length one or two. The first element specifies either the file extension or `"*"` for all file types. The second element gives an optional descriptor name for the file type. The supplied `filetype` list appears as a set of choices in the pull-down box labelled "Files of type:"".
`open`	logical; if `TRUE` display **Open** prompt, if `FALSE` display **Save As** prompt.

Value

The file name and path of the file selected by the 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 a "Save File" Dialogue*

Description

Display the default **Save As** prompt provided by the Operating System.

Usage

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

Arguments

`initialfile`	file name of the text file containing the list.
`filetype`	a list of character vectors indicating file types made available to users of the GUI. Each vector is of length one or two. The first element specifies either the file extension or `"*"` for all file types. The second element gives an optional descriptor name for the file type. The supplied `filetype` list appears as a set of choices in the pull-down box labelled "Files of type:".
`save`	logical; if `TRUE` display **Save As** prompt, if `FALSE` display **Open** prompt.

Value

The file name and path of the file selected by the 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 in PBS Modelling Format*

Description

Read in a list previously saved to a file by `writeList`. At present, only two formats are supported - R's native format used by the `dput` function or an ad hoc `PBSmodelling` format. The function `readList` detects the format automatically.

For information about the `PBSmodelling` format, see [writeList](#).

Usage

```
readList(fname)
```

Arguments

fname   file name of the text file containing the list.

See Also

[writeList](#), [unpackList](#)

---

resetGraph   *Reset par Values for a Plot*

Description

Reset `par()` to default values to ensure that a new plot utilizes a full figure region. This function helps manage the device surface, especially after previous plotting has altered it.

Usage

```
resetGraph()
```

See Also

resetGraph

---

restorePar   *Get Actual Parameters from Scaled Values*

Description

Restore scaled parameters to their original units. Used in minimization by `calcMin`.

Usage

```
restorePar(S,pvec)
```

Arguments

S	scaled parameter vector.
pvec	a data frame comprising four columns - `c("val","min","max","active")` and as many rows as there are model parameters. The `"active"` field (logical) determines whether the parameters are estimated (`TRUE`) or remain fixed (`FALSE`).

Details

Restoration algorithm: $P = P_{min} + (P_{max} - P_{min})(sin(\frac{\pi S}{2}))^2$

Value

Parameter vector converted from scaled units to original units specified by `pvec`.

See Also

scalePar, calcMin, GT0

Examples

```
pvec <- data.frame(val=c(1,100,10000),min=c(0,0,0),max=c(5,500,50000),
        active=c(TRUE,TRUE,TRUE))
S    <- c(.5,.5,.5)
P    <- restorePar(S,pvec)
print(cbind(pvec,S,P))
```

---

runDemos	*Interactive GUI for R demos*

Description

An interactive GUI for accessing demos from any R package installed on the user's system. `runDemos` is a convenient alternative to Rś `demo` function.

Usage

```
runDemos(package)
```

Arguments

package   display demos from a particular package (optional)

Details

If the argument `package` is not specified, the function will look for demos in all packages installed on the user's system.

Note

The `runDemos` GUI attempts to retain the user's objects and restore the working directory. However, pre-existing objects will be overwritten if their names co-incide with names used by the various demos. Also, depending on conditions, the user may lose working directory focus. We suggest that users run this demo from a project where data objects are not critical. **— USER BEWARE —**

See Also

[runExamples](#) for examples specific to PBSmodelling.

---

runExamples   *Run GUI Examples Included with PBS Modelling*

Description

Display an interactive GUI to demonstrate PBS Modelling examples.

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

Usage

```
runExamples()
```

Details

Some examples use external packages which must be installed to work correctly:

`BRugs` - `LinReg`, `MarkRec`, and `CCA`;

`odesolve`/`ddesolve` - `FishRes`;

`PBSmapping` - `FishTows`.

Note

The examples are copied from `PBSmodelling/examples` to R's current temporary working directory and run from there.

See Also

runDemos

---

scalePar   *Scale Parameters to [0,1]*

Description

Scale parameters for function minimization by `calcMin`.

Usage

```
scalePar(pvec)
```

Arguments

pvec   a data frame comprising four columns - `c("val","min","max","active")` and as many rows as there are model parameters. The `"active"` field (logical) determines whether the parameters are estimated (`TRUE`) or remain fixed (`FALSE`).

Details

Scaling algorithm: $S = \frac{2}{\pi} asin \sqrt{\frac{P - P_{min}}{P_{max} - P_{min}}}$

Value

Parameter vector scaled between 0 and 1.

See Also

[restorePar](), [calcMin](), [GT0]()

Examples

```
pvec <- data.frame(val=c(1,100,10000),min=c(0,0,0),max=c(5,500,50000),
        active=c(TRUE,TRUE,TRUE))
S    <- scalePar(pvec)
print(cbind(pvec,S))
```

---

setPBSext	*Set a Command Associated with a Filename Extension*

Description

Set a command with an associated extension (suffix), for use in openFile. The command must specify where the target file name is inserted by indicating a %f.

Usage

```
setPBSext(ext, cmd)
```

Arguments

ext	string of specifying the extension suffix.
cmd	command string to associate with the extension.

Note

These values are not saved from one PBS Modelling session to the next.

See Also

[getPBSext](), [openFile]()

---

setPBSoptions   *Set A User Option*

Description

Some options may be set by the user.

Usage

```
setPBSoptions(option, value)
```

Arguments

option   name of the option to set.

value   new value to assign this option.

See Also

[getPBSoptions](#)

---

setWinAct   *Add a Window Action to the Saved Action Vector*

Description

Append a string value specifying an action to the first position of an action vector.

Usage

```
setWinAct(winName, action)
```

Arguments

winName   window name where action is taking place.

action   string value describing an action.

Details

When a function is called from a GUI, a string descriptor associated with the action of the function is stored internaly (appended to the first position of the action vector). A user can utilize this action as a type of argument for programming purposes. The command `getWinAct()[1]` yields the latest action.

Sometimes it is useful to "fake" an action. Calling `setWinAct` allows the recording of an action, even if a button has not been pressed.

setWinVal   *Update Widget Values*

Description

Update a widget with a new value.

Usage

```
setWinVal(vars, winName)
```

Arguments

vars   a list or vector with named components.

winName   window from which to select GUI widget values. The default takes the window
                that has most recently received new user input.

Details

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`, `matrix`, and `data` widgets can be updated in several ways. If more than one
name is specified for the `names` argument of these widgets, each element is treated like an `entry`
widget.

If however, a single `name` describes any of these three widgets, the entire widget can be updated by
passing an appropriately sized object.

Alternatively, any element can be updated by appending its index in square brackets to the end of
the `name`. The `data` widget is indexed differently than the `matrix` widget by adding "d" after
the brackets. This tweak is necessary for the internal coding (bookkeeping) of PBS Modelling.
Example: `"foo[1,1]d"`.

See Also

[getWinVal](#), [createWin](#)

Examples

```
## Not run:
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))
## End(Not run)
```

---

show0 *Convert Numbers into Text with Specified Decimal Places*

Description

Return a character representation of a number with added zeroes out to a specified number of decimal places.

Usage

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

Arguments

x	numeric data (scalar, vector, or matrix).
n	number of decimal places to show, including zeroes.
add2int	If TRUE, add zeroes on the end of integers.

Value

A scalar/vector of strings representing numbers. Useful for labelling purposes.

Note

This function does not round or truncate numbers. It simply adds zeroes if n is greater than the available digits in the decimal part of a number.

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

Display the order and default values of **all** widget arguments. The list can be shortened by specifying
a single widget name.

Usage

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

Arguments

widget   If specified, information about this one widget only is displayed. The default
                displays information about all widgets.

Value

A text stream to the R console. Cannot be directed to a file or other device.

---

sortHistory   *Sort an Active or Saved History*

Description

Utility to sort history. When called without any arguments, an interactive GUI is used to pick which
history to sort. When called with `hisname`, sort this active history widget. When called with
`file` and `outfile`, sort the history located in `file` and save to `outfile`.

Usage

```
sortHistory(file="", outfile=file, hisname="")
```

Arguments

file   file name of saved history to sort.

outfile   file to save sorted history to.

hisname   name of active history widget and window it is located in, given in the form
                `WINDOW.HISTORY`.

Details

After selecting a history to sort (either from given arguments, or interactive GUI) the R data editor window will be displayed. The editor will have one column named ïewẅhich will have numbers 1,2,3,...,n. This represents the current ordering of the history. You may change the numbers around to define a new order. The list is sorted by reassigning the index in row i as index i.

For example, if the history had three items 1,2,3. Reordering this to 3,2,1 will reverse the order; changing the list to 1,2,1,1 will remove entry 3 and create two duplicates of entry 1.

See Also

importHistory, initHistory

---

testCol	*Display Named Colours Available Based on a Set of Strings*

Description

Display colours as patches in a plot. Useful for programming purposes. Colours can be specified in any of 3 different ways: (i) by colour name, (ii) by hexidecimal colour code created by rgb(), or (iii) by an index to the color() palette.

Usage

```
testCol(cnam=colors()[sample(length(colors()),15)])
```

Arguments

cnam   vector of colour names to display. Defaults to 15 random names from the color palette.

See Also

pickCol

Examples

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

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

testCol(substring(rainbow(63),1,7))

#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 Available*

Description

Display line types available.

Usage

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

Arguments

newframe   if TRUE, create a new blank frame, otherwise overlay current frame.

Note

Quick representation of first 20 line types for reference purposes.

---

testLwd   *Display Line Widths*

Description

Display line widths. 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 display. Ranges can be specified.

col   colours to use for lines. Patterns are repeated if `length(lwd) > length(col)`

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 and Backslash Characters*

Description

Display plotting symbols. User can specify particular ranges (increasing continuous integer) for `pch`.

Usage

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

Arguments

pch	symbol codes to view.
ncol	number of columns in display (can only be 2, 5, or 10). Most sensibly this is set to 10.
grid	logical; if TRUE, grid lines are plotted for visual aid.
newframe	logical; if TRUE reset the graph, otherwise overlay on top of the current graph.
bs	logical; if TRUE, show 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 their Source Code*

Description

Display an interactive GUI to demonstrate the available widgets in PBS Modelling. A `text` window displays the Window Description File source code. The user can modify this sample code and recreate the test GUI by pressing the button below.

The Window Description Files can be found in the directory `PBSmodelling/testWidgets` located in the R directory `library`.

Usage

```
testWidgets()
```

Details

Following are the widgets and default values supported by PBS Modelling. See Appendix A in "PBSModelling-UG.pdf" for detailed descriptions.

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

check name mode=logical checked=FALSE text="" font="" fg="black" bg=""
   function="" action="check" sticky="" padx=0 pady=0

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

entry name value="" width=20 label="" font="" fg="" bg=""
   entryfont="" entryfg="black" entrybg="white" 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" function="" import="" sticky="" padx=0 pady=0

label text="" font="" fg="black" bg="" sticky="" justify="left"
   wraplength=0 padx=0 pady=0

matrix nrow ncol names rowlabels="" collabels="" rownames=""
   colnames="" font="" fg="black" bg="" entryfont="" entryfg="black"
   entrybg="white" 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

object name font="" fg="black" bg="" entryfont=""
   entryfg="black" entrybg="white" vertical=FALSE function=""
   enter=TRUE action="data" width=6 sticky="" padx=0 pady=0

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

slide name from=0 to=100 value=NA showvalue=FALSE
   orientation="horizontal" font="" fg="black" bg="" 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 scrollbar=TRUE
   fg="black" bg="white" mode="character" font="" value=""
   borderwidth=1 relief="sunken" sticky="" padx=0 pady=0

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

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

See Also

[createWin](), [showArgs]()

---

unpackList	*Unpack List Elements into Variables*

Description

Make local or global variables (depending on the scope specified) from the named components of a list.

Usage

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

Arguments

x	named 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);
```

---

vbdata   *Dataset: Length-at-Age Data for a von Bertalanffy Curve*

Description

Lengths-at-age for freshwater mussels (*Anodonta kennerlyi*).

Usage

```
data(vbdata)
```

Format

A data frame with 16 rows and 2 columns `c("age","len")`.

Details

Data for demonstartion of the von Bertalanffy model used in the `calcMin` example.

Source

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

---

vbpars   *Dataset: Initial Parameters for a von Bertalanffy Curve*

Description

Starting parameter values for `Linf`, `K`, and `t0` for von Bertalanffy minimization using length-at-age data ([vbdata](#)) for freshwater mussels (*Anodonta kennerlyi*).

Usage

```
data(vbpars)
```

Format

A matrix with 3 rows and 3 columns `c("Linf","K","t0")`. Each row contains the starting values, minima, and maxima, respectively, for the three parameters.

Details

Data for demonstration of the von Bertalanffy model used in the `calcMin` example.

Source

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

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

Description

View the first n rows of a data frame or matrix or the 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 rows (matrix/data frame) or elements (vector/list) of `obj` to view.

`writeList`	*Write a List to a File in PBS Modelling Format*

Description

Write 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 represents a simple list in an easy-to-read, ad hoc `PBSmodelling` format.

Usage

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

Arguments

`x`	R list object to write to an ASCII text file.
`fname`	file name of the text file containing the list.
`format`	format of the file to create: `"D"` or `"P"`.
`comments`	vector of character strings to use as initial-line comments in the file.

Details

The `"D"` format is equivalent to using R's `base` functions `dput` and `dget`, which support all R objects.

The `"P"` format only supports named lists of vectors, matrices, and data frames. Scalars are treated like vectors. Nested lists are not supported.

The `"P"` format writes each named element in a list using the following conventions: (i) $ followed by the name of the data object to denote the start of that object's description; (ii) $$ on the next line to describe the object's structure - object type, mode(s), names (if vector), rownames (if matrix or data), and colnames (if matrix or data); and (iii) subsequent lines of data (one line for vector, multiple lines for matrix or data).

Multiple rows of data for matrices or data frames must have equal numbers of entries (separated by whitespace).

For complete details, see "PBSmodelling-UG.pdf" in R's directory `library/PBSmodelling`.

See Also

[readList](), [openFile]()

Examples

```
## Not run:
test <- list(a=10,b=euro,c=view(WorldPhones),d=view(USArrests))
writeList(test,"test.txt",format="P",
        comments=" Scalar, Vector, Matrix, Data Frame")
openFile("test.txt")
## End(Not run)
```

Index