Examples for gWidgets

John Verzani, gWidgetsRGtk@gmail.com

July 13, 2011

Abstract:

[This was updated last for **gWidgets_0.0-41**.]

Examples for using the **gWidgets** package are presented. The **gWidgets** API is intended to be a cross platform means within an R session to interact with a graphics toolkit. Currently, the API can be used through:

- **gWidgetsRGtk2** for **RGtk2** package which is the most complete implementation.
- **gWidgetstcltk** for the **tcltk** package which is the easiest to install (for windows users) but not as fully implemented or polished.
- **gWidgetsQt** for the **qtbase** package. This package allows the Qt toolkit to be used from within R. It is being developed and is available from r-forge.
- **gWidgetsrJava** for the **rJava** package. This package may be deprecated in the near term if no interest is shown.

The **gWidgetsWWW** package is an independent implementation for web programming. It uses the EXT JS JavaScript libraries to provide R to web interactivity. It leverages the **rapache** package for server installs, and has a user-only web browser leveraged from the **Rpad** package. Unlike the other implementations, this package is stand alone and does not use **gWidgets** itself.

The API is intended to facilitate the task of writing basic GUIs, as it simplifies many of the steps involved in setting up widgets and packing them into containers. Although not nearly as powerful as any individual toolkit, the **gWidgets** API is suitable for many tasks or as a rapid prototyping tool for more complicated applications.

The examples contained herein illustrate that quite a few things can be done fairly easily with more complicated applications being pieced together in a straightforward manner. To see a fairly complicated application built using **gWidgets**, install the **pmg** GUI (`http://www.math.csi.cuny.edu/pmg`), which is on CRAN.

The **traitr** package implements a model-view-controller programming style for GUI development. It uses **gWidgets** to provide the GUI components.

1 Background

The `gWidgets` API is intended to be a cross-toolkit API for working with GUI objects. It is based on the iwidgets API of Simon Urbanek, with improvements suggested by Philippe Grosjean, Michael Lawrence, Simon Urbanek and John Verzani. This document focuses on the more complete toolkit implementation provided by the **gWidgetsRGtk2** package. [1] The GTK toolkit is interfaced via the **RGtk2** package of Michael Lawrence, which in turn is derived from Duncan Temple Lang's **RGtk** package. The excellent **RGtk2** package opens up the full power of the GTK2 toolkit, only a fraction of which is available though **gWidgetsRGtk2**.

The `gWidgets` API is mostly stable, but does occasionally have additional methods and constructors added. If a feature seems lacking, please let the author know.

2 Overview

The basic tasks in setting up a GUI involve: constructing the desired widgets, laying these widgets out into containers, and assigning handlers to respond to user-driven events involving the widgets. The **gWidgets** package is geared around these tasks, and also around providing a familiar means to interact with the widgets once constructed. The actual toolkits have much more flexibility during the layout of the GUI and afterwards in pragmatically interacting with the GUI. Each widget constructor in **gWidgets** includes the arguments `container` to specify a parent container to place it in; `handler`, to specify a handler to respond to the main event that the widget has; and `action`, which is used to pass extra information along to any handler. Each widget when constructed returns an object of S4 class which can be manipulated using generic functions, some new and some familiar.

[1] Occasional differences with **gWidgetsQt**, **gWidgetstcltk**, or **gWidgetsrJava** are pointed out in braces. The major differences from the **gWidgets** API are documented in the help files given by the package name, e.g. **gWidgetstcltk**

The basic widgets are likely familiar: buttons, labels, text-edit areas, etc. In addition, there are some R-specific compound widgets such as a variable browser, data frame viewer, and help viewer. In contrast to these, are a few basic dialogs, which draw their own window and are modal, hence do not return objects which can be manipulated.

The examples below introduce the primary widgets, containers, and dialogs in a not-so systematic manner, focusing instead on some examples which show how to use **gWidgets**. The man pages are also a source information about the widgets, see `?gWidgets` for more detail. For differences between the toolkit implementation, see the man page for the package name, for example, `?gWidgetstcltk`.

This document is a vignette. As such, the code displayed is available within an R session through the command `edit(vignette("gWidgets"))`.

3 Installation

In case you are reading this vignette without having installed gWidgets, here are some instructions focusing on installing **gWidgetsRGtk2**.

More details are found at the **gWidgets** website `http://www.math.csi.cuny.edu/pmg/gWidgets`.

Installing `gWidgets` with the `gWidgetsRGtk2` package requires two steps: installing the GTK libraries and installing the R packages.

3.1 Installing the GTK libraries

The `gWidgetsRGtk2` provides a link between `gWidgets` and the GTK libraries through the `RGTk2` package. `RGtk2` requires relatively modern versions of the GTK libraries (2.8.12 is suggested). These may need to be installed or upgraded on your system.

In case of **Windows** you can do this:

1. Download the files from http://gladewin32.sourceforge.net/modules/wfdownloads/visit.php?lid=
2. run the resulting file. This is an automated installer which will walk you through the installation of the Gtk2 libraries.

For Windows users, the following command, will do this and install `pmg`. (Likely you are better off with the previous method.)

```
> source("http://www.math.csi.cuny.edu/pmg/installPMG.R")
```

In Linux, you may or may not need to upgrade the GTK libraries depending on your distribution. If you have a relatively modern installation, you should be ready.

For Mac OS X there is also a packaged port of the GTK run time libraries available through at `http://r.research.att.com/`.

When the author used this quite some time ago, the **cairoDevice** package failed (this may be fixed by now). As such, the the macports (`http://www.macports.org`) project was used to install its `gtk2` package. Once macports is installed, the command

```
sudo port install gtk2
```

will install the libraries (although it may take quite a while).

There are more details on RGtk2 at RGtk2's home page.

3.2 Install the R packages

The following R packages are needed: `RGtk2`, `cairoDevice`, `gWidgets`, and `gWidgetsRGtk2`. Install them in this order, as some depend on others to be installed first. All can be downloaded from CRAN.

These can all be installed by following the dependencies for `gWidgetsRGtk2`. The following command will install them all if you have the proper write permissions:

```
install.packages("gWidgetsRGtk2", dep = TRUE)
```

It may be necessary to adjust the location where the libraries will be installed if you do not have the proper permissions.

For MAC OS X, the source packages might be desired if you installed via macports, not the default "mac.binary." Use the `type=` argument, as follows:

```
> install.packages("gWidgetsRGtk2", dep = TRUE,type = "source" )
```

[The **gWidgetsQt** package requires the **qtbase** package which is not yet on CRAN, but rather is installed through r-forge.]

[The **gWidgetstcltk** requires the 8.5 version of tcl/tk which comes with the windows version of R as of 2.7.0, but typically needs to be installed manually for linux and mac machines, as of this writing.]

[The **gWidgetsrJava** package is installed similarly. However, it requires **rJava**. These should install through the dependencies, so

```
> install.packages("gWidgetsrJava", dep = TRUE)
```

should do it. The **rJava** has some potential issues with the mac version of R that can arise if R is compiled from source.]

[The **gWidgetsWWW** package is altogether different. The package requires **rapache** to be installed to be used in server mode. As well, the apache web server must also be configured. The package can be used locally through a standalone web server.]

4 Loading gWidgets

We load the `gWidgets` package, using the `gWidgetsRGtk2` toolkit, below: following

```
> require(gWidgets)
> options("guiToolkit"="RGtk2")
```

When `gWidgets` is started, it tries to figure out what its default toolkit will be. In this examples, we've set it to be "gWidgetsRGtk2." If the option was not set and there is more than one toolkit available, then a menu asks the user to choose between toolkit implementations.

[For **gWidgetsQt** use "Qt", for **gWidgetsrJava** use "rJava" in the `options` call or "tcltk" for the **gWidgetstcltk** package.]

[The **gWidgetsWWW** package is different, as it doesn't use the **gWidgets** package for dispatch. It is loaded directly via `require(gWidgetsWWW, quietly=TRUE)` – using the `quietly` switch, as anything output to STDOUT is sent to the browser.]

5 Hello world

We begin by showing how to make various widgets which display the ubiquitous "Hello world" message.

Now to illustrate (Figure 1 shows a few) some of the basic widgets. These first widgets display text: a label, a button and a text area.

First a button:

```
>   obj <- gbutton("Hello world", container = gwindow())
```

Next a label:

```
>   obj <- glabel("Hello world", container = gwindow())
```

Now for single line of editable text:

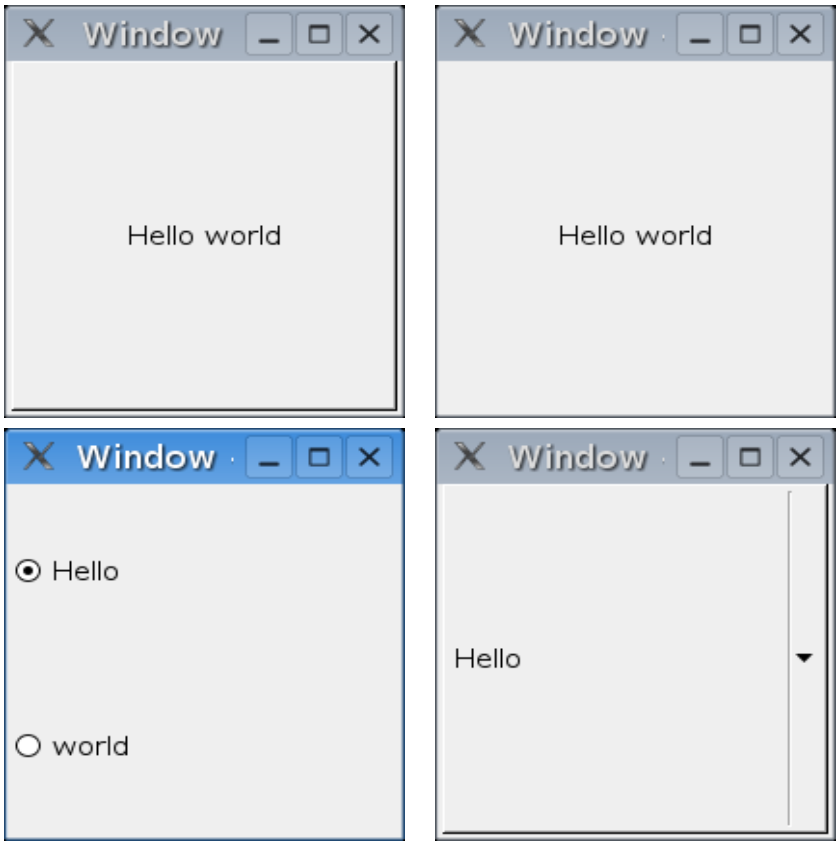

Figure 1: Four basic widgets: a button, a label, radio buttons, and a drop list.

```
>   obj <- gedit("Hello world", container = gwindow())
```

Finally, a text buffer for multiple lines of text:

```
>   obj <- gtext("Hello world", container = gwindow())
```

The following widgets are used for selection of a value or values from a vector of possible values.

First a radio group for selecting just one of several:

```
>   obj <- gradio(c("hello","world"), container=gwindow())
```

Next, combo box, (was called a droplist) again for selecting just one of several, although in this case an option can be give for the user to edit the value.

```
>   obj <- gcombobox(c("hello","world"), container=gwindow())
```

A combo box can also allow its value to be entered by typing,

```
>   obj <- gcombobox(c("hello","world"), editable=TRUE, container=gwindow())
```

For longer lists, a table of values can be used.

```
>   obj <- gtable(c("hello","world"), container=gwindow())
```

This widget is also used for displaying tabular data with multiple columns and rows (data frames). For this widget there is an argument allowing for multiple selections. Multiple selections can also be achieved with a checkbox group:

```
>   obj <- gcheckboxgroup(c("hello","world"), container=gwindow())
```

For selecting a numeric value from a sequence of values, sliders and spinbuttons are commonly used:

```
>   obj <- gslider(from=0, to = 7734, by =100, value=0,
+     container=gwindow())
>   obj <- gspinbutton(from=0, to = 7734, by =100, value=0,
+    container=gwindow())
```

Common to all of the above is a specification of the "value" of the widget, and the container to attach the widget to. In each case a top-level window constructed by `gwindow`.

5.1 Using containers

In this next example, we show how to combine widgets together using containers. (Figure 2.)

```
>   win <- gwindow("Hello World, ad nauseum", visible=TRUE)
>   group <- ggroup(horizontal = FALSE, container=win)
>   obj <- gbutton("Hello...",container=group,
+     handler = function(h,...) gmessage("world"))
>   obj <- glabel("Hello...", container =group,
+     handler = function(h,...) gmessage("world"))
>   obj <- gcombobox(c("Hello","world"), container=group)
>   obj <- gedit("Hello world", container=group)
>   obj <- gtext("Hello world", container=group, font.attr=list(style="bold"))
```

As before, the constructors `gbutton`, `glabel`, `gedit` and `gtext` create widgets of different types [2]. The button looks like a button. A label is used to show text which may perhaps be edited. [Editing text isn't implemented in all toolkits. and in personal experience seems to be confusing to users.] A combobox allows a user to select one of several items, or as illustrated can take user input. The `gedit` and `gtext` constructors both create widgets for inputting text, in the first case for single lines, and in the second for multiple lines using a text buffer.

Figure 2: Hello world example

[2] We refer to `gbutton` as a constructor, which it is, and also a widget class, which it technically isn't.

These widgets are packed into containers (see `?ggroup` or `?gwindow`). The base container is a window, created with the `gwindow` function. A top-level window only contains one widget, like a group, so we pack in a group container created with `ggroup`. (Top-level windows also may contain menubars, toolbars and statusbars.) The `ggroup` container packs in widgets from left to right or top to bottom. Imagine each widget as a block which is added to the container. In this case, we want the subsequent widgets packed in top to bottom so we used the argument `horizontal=FALSE`.

For the button and label widgets, a handler is set so that when the widget is clicked a message dialog appears showing "world." Handlers are used to respond to mouse-driven events. In this case the event of a widget being clicked. See `?gWidgetsRGtk-handlers` for details on handlers.

Note the handler has signature `(h,...)` where `h` is a list with components `obj` referring to the widget the handler is called on, `action` referring to the value passed to the `action=` argument and perhaps others (eg. `dropdata` for drag-and-drop events, and `x` and `y` for `ggraphics` click events.). The `...` in the signature, may contain information passed along by the underlying toolkit and is necessary but not generally employed.

The message is an instance of a "basic dialog" (see `?gWidgets-dialogs`). The dialogs in `gWidgets` are modal, meaning R's event loop is stopped until the dialog is answered. (This can be annoying if a dialog appears under another window and can't be seen!) As such, they don't return an object which has methods defined for it, as by the time they can be accessed they have been dismissed. Instead, they return a value like a logical or a string.

6 Making a confirmation dialog

Let's see how we might use widgets to create our own confirmation dialog, one which is not modal. We want to have an icon, a label for the message, and buttons to confirm or dismiss the dialog.

The `gimage` constructor allows images to be shown in a widget. In **gWidgetsRGtk** there are several stock images, which can be listed with `getStockIcons`. We will use "info" below.

First we define a function for making a dialog. This one uses nested group containers to organize the layout. (Alternately the `glayout` constructor could have been used in some manner.)

```
> confirmDialog <- function(message, handler=NULL)  {
```

```
+   window <- gwindow("Confirm")
+   group <- ggroup(container = window)
+   gimage("info", dirname="stock", size="dialog", container=group)
+
+   ## A group for the message and buttons
+   inner.group <- ggroup(horizontal=FALSE, container = group)
+   glabel(message, container=inner.group, expand=TRUE)
+
+   ## A group to organize the buttons
+   button.group <- ggroup(container = inner.group)
+   ## Push buttons to right
+   addSpring(button.group)
+   gbutton("ok", handler=handler, container=button.group)
+   gbutton("cancel", handler = function(h,...) dispose(window),
+           container=button.group)
+
+   return()
+ }
```

The key to making a useful confirmation dialog is attaching a response to a click of the "ok" button. This is carried out by a handler, which are added using the argument `handler=` for the constructor, or with one of the `addHandlerXXX` functions. The handler below prints a message and then closes the dialog. To close the dialog, the `dispose` method is called on the "ok" button widget, which is referenced inside the handler by `h$obj` below. In `gWidgets`, handlers are passed information via the first argument, which is a list with named elements. The `$obj` component refers to the widget the handler is assigned to.

Trying it out produces a widget like that shown in Figure 3

```
> confirmDialog("This space for rent", handler = function(h,...) {
+   print("what to do... [Change accordingly]")
+   ## In this instance dispose finds its parent window and closes it
+   dispose(h$obj)
+ })

NULL
```

Figure 3: Confirmation dialog

7 Methods

Widgets are interacted with by their methods. The main methods are `svalue` and `svalue<-` for getting and setting a widgets primary value.

The following silly example illustrates how clicking one widget can be used to update another widget.

```
> w <- gwindow("Two widgets")
> g <- ggroup(container = w)
> widget1 <- gbutton("Click me to update the counter", container=g,
+                    handler = function(h,...) {
+                      oldVal <- svalue(widget2)
+                      svalue(widget2) <- as.numeric(oldVal) + 1
+                    })
> widget2 <- glabel(0, container=g)
```

The value stored in a label is just the text of the label. This is returned by `svalue` and after 1 is added to the value, replaced back into the label. As text labels are of class "character," the value is coerced to be numeric.

There are other methods (see `?gWidgetsRGtk-methods`) that try to make interacting with a widget as natural (to an R user) as possible. For instance, a radio button has a selected value returned by `svalue`, but also a vector of possible values. These may be referenced using vector, `[`, notation. Whereas, a notebook container has a `names` method which refers to the tab labels, which may be set via `names<-` and a `length` method to return the number of notebook pages.

8 Adding a GUI to some common tasks

A GUI can make some command line tasks easier to perform. Here are a few examples that don't involve much coding in `gWidgets`.

8.1 `file.choose`

The `file.choose` function is great for simplifying a user's choice of a file from the file system. A typical usage might be

```
source(file.choose())
```

to allow a user to source a file with a little help from a GUI. However, in many UNIX environments, there is no GUI for `file.choose`, only a more convenient curses interface. With the `gfile` dialog, we can offer some improvement.

This dialog returns the name of the file selected, so that

```
source(gfile())
```

can replace the above.

More in keeping with the `gWidgets` style, though, would be to give a handler when constructing the file chooser. The `file` component of the handler argument gives the files name (not `svalue(h$obj)`, as `gfile` does not return an object to call `svalue` on). For example, the function below is written to give some flexibility to the process:

```
> fileChoose <- function(action="print", text = "Select a file...",
+                        type="open", ...) {
+   gfile(text=text, type=type, ..., action = action, handler =
+         function(h,...) {
+           do.call(h$action, list(h$file))
+         })
+ }
```

The `action=` argument parametrizes the action. The default above calls `print` on the selected file name, hence printing the name. However, other tasks can now be done quite simply. For example, to `source` a file we have:

```
> fileChoose(action="source")
```

Or to set the current working directory we have:

```
> fileChoose(action="setwd", type="selectdir", text="Select a directory...")
```

8.2 browseEnv

The `browseEnv` function creates a table in a web browser listing the current objects in the global environment (by default) and details some properties of them. This is an easy to use function, but suffers from the fact that it may have to open up a browser for the user if none is already open. This may take a bit of time as browsers are generally slow to load. We illustrate a means of using the `gtable` constructor to show in a table the objects in an environment.

The following function creates the data.frame we will display. Consult the code of `browseEnv` to see how to produce more details.

```
> lstObjects <- function(envir= .GlobalEnv, pattern) {
+   objlist <- ls(envir=envir, pattern=pattern)
+   objclass <- sapply(objlist, function(objName) {
+     obj <- get(objName, envir=envir)
+     class(obj)[1]
+   })
+   data.frame(Name = I(objlist), Class = I(objclass))
+ }
```

Now to make a table to display the results. We have some flexibility with the arguments, which is shown in subsequent examples:

```
> browseEnv1 <- function(envir = .GlobalEnv, pattern) {
+   listOfObjects <- lstObjects(envir=envir, pattern)
+   gtable(listOfObjects, container = gwindow("browseEnv1"))
+ }
```

Tables can have a double click handler (typically a single click is used for selection). To illustrate, we add a handler which calls `summary` (or some other function) on a double-clicked item [Qt's behaviour is OS dependent]. (The first `svalue` returns a character string, it is promoted to an object using `get`)

```
> browseEnv2 <- function(envir = .GlobalEnv, pattern, action="summary") {
+   listOfObjects <- lstObjects(envir=envir, pattern)
+   gtable(listOfObjects, container = gwindow("browseEnv2"),
+          action = action,
+          handler = function(h,...) {
+            print(do.call(h$action, list(get(svalue(h$obj)))))
+          })
+ }
```

As a final refinement, we add a combobox to filter by the unique values of "Class." We leave as an exercise the display of icons based on the class of the object.

```
> browseEnv3 <- function(envir = .GlobalEnv, pattern, action="summary") {
+   listOfObjects <- lstObjects(envir=envir, pattern)
+   gtable(listOfObjects,
+          container =gwindow("browseEnv3"),
+          filter.column = 2,
+          action = action,
+          handler = function(h,...) {
+            print(do.call(h$action, list(get(svalue(h$obj)))))
+          })
+ }
```

[The `filter` argument is not available with **gWidgetsrJava** or **gWidgetsWWW**.]

In **gwidgetsRGtk2** The `gvarbrowser` function constructs a widget very similar to this, only it uses a `gtree` widget to allow further display of list-like objects. [The `gtree` widget is not implemented in all of the toolkits.]

9 A gWidgetsDensity demo

We illustrate how to make a widget dynamically update a density plot. The idea comes from the `tkdensity` demo that accompanies the **tcltk** package due to, I believe, Martin Maechler.

We use the `ggraphics` constructor to create a new plot device. For RGtk2, this uses the **cairoDevice** package also developed by Michael Lawrence. (This package takes some work to get going under OS X, as the easy-to-install RGtk2 libraries don't seem to like the package.)

[In **gWidgetsrJava** the **JavaGD** package is used for a JAVA device. In theory this should be embeddable in a **gWidgets** container, but it isn't implemented, so a new window is created.]

[In **gWidgetstcltk** there is no embeddable graphics device. (The **tkrplot** is different.) The `ggraphics` call would just put in a stub.]

This demo consists of a widget to control a random sample, in this case from the standard normal distribution or the exponential distribution with rate 1; a widget to select the sample size; a widget to select the kernel; and a widget to adjust the default bandwidth. We use radio buttons for the first two, a drop list for the third and a slider for the latter.

The **gWidgetstcltk** package is a little different from the other two, as it requires that a container be non-null. This is because the underlying **tcltk** widgets need to have a parent container to be initialized. As such, the following won't work. A working example will be given next for comparison. This one is left here, as the separation of GUI building into: widget definition; widget layout; and assignment of handlers, or call backs, seems to lead to easier to understand code.

Proceeding, first we define the two distributions and the possible kernels.

```
> ## set up
> availDists <- c(Normal="rnorm", Exponential="rexp")
> availKernels <- c("gaussian", "epanechnikov", "rectangular",
+                   "triangular", "biweight", "cosine", "optcosine")
```

We then define the key function for drawing the graphic. This refers to widgets yet to be defined.

```
> updatePlot <- function(h,...) {
+   x <- do.call(availDists[svalue(distribution)],list(svalue(sampleSize)))
+   plot(density(x, adjust = svalue(bandwidthAdjust),
+                kernel = svalue(kernel)),main="Density plot")
+   rug(x)
+ }
```

Now to define the widgets.

```
> distribution <- gradio(names(availDists), horizontal=FALSE, handler=updatePlot)
> kernel <- gcombobox(availKernels, handler=updatePlot)
> bandwidthAdjust <- gslider(from=0,to=2,by=.01, value=1, handler=updatePlot)
> sampleSize <- gradio(c(50,100,200, 300), handler = updatePlot)
```

Now the layout. We use frames to set off the different arguments. A frame is like a group, only it has an option for placing a text label somewhere along the top. The position is specified via the `pos` argument, with a default using the left-hand side.

```
> ## now layout
> window <- gwindow("gWidgetsDensity")
> BigGroup <- ggroup(cont=window)
> group <- ggroup(horizontal=FALSE, container=BigGroup)
> tmp <- gframe("Distribution", container=group)
> add(tmp, distribution)
```

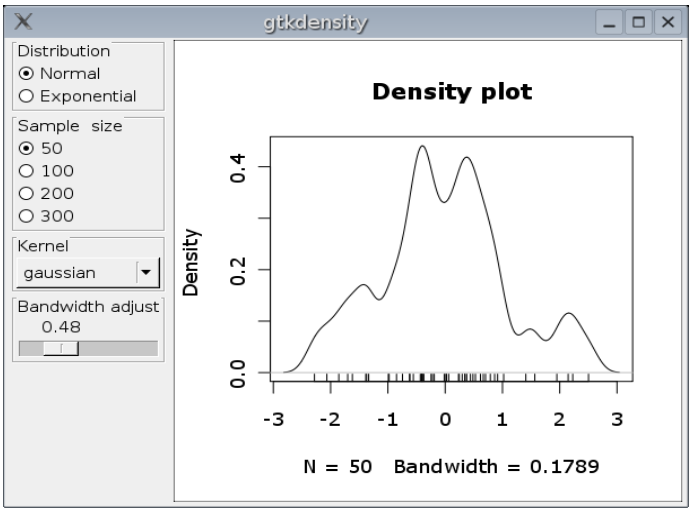

Figure 4: The gWidgetsDensity example in action.

```
> tmp <- gframe("Sample  size", container=group)
> add(tmp,sampleSize)
> tmp <- gframe("Kernel", container=group)
> add(tmp,kernel)
> tmp <- gframe("Bandwidth adjust", container=group)
> add(tmp,bandwidthAdjust, expand=TRUE)
```

Now to add a graphics device.

```
> add(BigGroup, ggraphics())
```

[Again, if using `gWidgetsrJava` this wouldn't place the device inside the `BigGroup` container.]

A realization of this widget was captured in Figure 4.

Now, as promised, we present a function that would work under any of the toolkits. The `ggraphics` call is not included, as this doesn't work with **gWidgetstcltk**. The key difference though is the containers are included in the creation of the widgets. The arguments that would be passed to `add` are included in the construction of the widget.

```
> gwtkdensity <- function() {
+
+   ## set up
```

```
+   availDists <- c(Normal = "rnorm", Exponential="rexp")
+   availKernels <- c("gaussian", "epanechnikov", "rectangular",
+                     "triangular", "biweight", "cosine", "optcosine")
+
+
+   updatePlot <- function(h,...) {
+     x <- do.call(availDists[svalue(distribution)],list(svalue(sampleSize)))
+     plot(density(x, adjust = svalue(bandwidthAdjust), kernel = svalue(kernel)))
+     rug(x)
+   }
+
+   ##The widgets
+   win <- gwindow("gwtkdensity")
+   gp <- ggroup(horizontal=FALSE, cont=win)
+
+   tmp <- gframe("Distribution", container=gp, expand=TRUE)
+   distribution <- gradio(names(availDists), horizontal=FALSE,
+                          cont=tmp,
+                          handler=updatePlot)
+
+
+   tmp <- gframe("Sample  size", container=gp, expand=TRUE)
+   sampleSize <- gradio(c(50,100,200, 300), cont=tmp,
+                        handler =updatePlot)
+
+
+   tmp <- gframe("Kernel", container=gp, expand=TRUE)
+   kernel <- gcombobox(availKernels, cont=tmp,
+                       handler=updatePlot)
+
+   tmp <- gframe("Bandwidth adjust", container=gp, expand=TRUE)
+   bandwidthAdjust <- gslider(from=0,to=2,by=.01, value=1,
+                              cont=tmp, expand=TRUE,
+                              handler=updatePlot)
+
+ }
```

10 Composing email

This example shows how to write a widget for composing an email message. Not that this is what R is intended for, but rather to show how a familiar widget is produced by combining various pieces from `gWidgets`. This example is a little lengthy, but hopefully straightforward due to the familiarity with the result of the task.

For our stripped-down compose window we want the following: a menubar to organize functions; a toolbar for a few common functions; a "To:" field which should have some means to store previously used e-mail addresses; a "From:" field that should be editable, but not obviously so as often it isn't edited; a "Subject:" field which also updates the title of the window; and a text buffer for typing the message.

The following code will create a function called `Rmail` (apologies to any old-time emacs users) which on many UNIX machines can send out e-mails using the `sendmail` command.

This is not written to work with `gWidgetstcltk`.

First we define some variables:

```
> FROM <- "gWidgetsRGtk <gWidgetsRGtk@gmail.com>"
> buddyList <- c("My Friend <myfriend@gmail.com>","My dog <mydog@gmail.com>")
```

Now for the main function. We define some helper functions inside the body, so as not to worry about scoping issues.

```
> Rmail <- function(draft = NULL, ...) {
+   ## We use a global list to contain our widgets
+   widgets <- list()
+
+
+   ## Helper functions
+   sendIt <- function(...) {
+     tmp <- tempfile()
+
+     cat("To:", svalue(widgets$to),"\n",file = tmp, append=TRUE)
+     cat("From:", svalue(widgets$from),"\n", file=tmp, append=TRUE)
+     cat("Subject:", svalue(widgets$subject),"\n", file=tmp, append=TRUE)
+     cat("Date:", format(Sys.time(),"%d %b %Y %T %Z"),"\n", file=tmp, append=TRUE)
+     cat("X-sender:", "R", file=tmp, append=TRUE)
+     cat("\n\n", file=tmp, append=TRUE)
+     cat(svalue(widgets$text), file=tmp, append=TRUE)
+     cat("\n", file=tmp, append=TRUE)
```

```
+
+     ## Use UNIX sendmail to send message
+     system(paste("sendmail -t  <", tmp))
+     ## Add To: to buddyList
+     if(exists("buddyList"))
+       assign("buddyList", unique(c(buddyList,svalue(widgets$to))), inherits=TRUE)
+
+     ## Close window, delete file
+     unlink(tmp)
+     dispose(window)
+   }
+
+   ## Function to save a draft to the file draft.R
+   saveDraft <- function(...) {
+     draft <- list()
+     sapply(c("to","from","subject","text"), function(i)
+            draft[[i]] <<- svalue(widgets[[i]])
+            )
+     dump("draft","draft.R")
+     cat("Draft dumped to draft.R\n")
+   }
+
+   ## A simple dialog
+   aboutMail <- function(...) gmessage("Sends a message")
+
+   ## Make main window from top down
+
+   window <- gwindow("Compose mail")
+   group <- ggroup(horizontal=FALSE, spacing=0, container = window)
+   ## Remove border
+   svalue(group) <- 0
+
+   actions <- list(save=gaction("Save", icon="save", handler=saveDraft),
+                   send=gaction("Send", icon="connect", handler=sendIt),
+                   quit=gaction("Quit", icon="quit", handler=function(...) dispose
+                   about=gaction("About", icon="about", handler=aboutMail))
+
+   ## Menubar is defined by the actions, but we nest under File menu
```

```
+   menubarlist <- list(File=actions)
+   gmenu(menubarlist, cont = window)
+
+   ## Toolbar is also defined by the actions
+   toolbarlist <-  actions[-4]
+   gtoolbar(toolbarlist, cont=window)
+
+
+   ## Put headers in a glayout() container
+   tbl <- glayout(container = group)
+
+
+   ## To: field. Looks for buddyList
+   tbl[1,1] <- glabel("To:",container = tbl)
+   tbl[1,2] <- (widgets$to <- gcombobox(c(), editable=TRUE, container=tbl))
+   if(exists("buddyList")) widgets$to[] <- buddyList
+
+   ## From: field. Click to edit value
+   tbl[2,1] <- glabel("From:", container = tbl)
+   tbl[2,2] <- (widgets$from <- glabel(FROM, editable=TRUE, container=tbl))
+
+   ## Subject: field. Handler updates window title
+   tbl[3,1] <- glabel("Subject:")
+   tbl[3,2] <- (widgets$subject <- gedit("",container=tbl))
+   addHandlerKeystroke(widgets$subject, handler = function(h,...)
+                       svalue(window) <- paste("Compose mail:",svalue(h$obj),colla
+
+   ## Add text box for message, but first some space
+   addSpace(group, 5)
+   widgets$text <- gtext("", container = group, expand=TRUE)
+
+
+   ## Handle drafts. Either a list or a filename to source")
+   ## The generic svalue() method makes setting values easy")
+   if(!is.null(draft)) {
+     if(is.character(draft))
+       sys.source(draft,envir=environment()) # source into right enviro
+     if(is.list(draft)) {
```

```
+       sapply(c("to","from","subject","text"), function(i) {
+         svalue(widgets[[i]]) <- draft[[i]]
+       })
+     }
+   }
+
+   ## That's it.
+
+ }
>
```

To compose an e-mail we call the function as follows. (The widget constructed looks like Figure 5.)

```
>   Rmail()
```

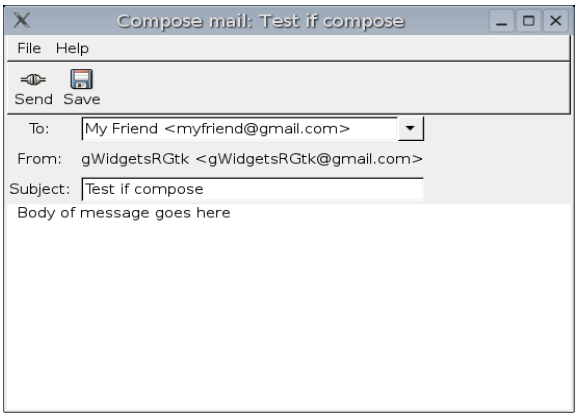

Figure 5: Widget for composing an e-mail message. (Needs a grammar checker.)

The `Rmail` function uses a few tricks. A combobox is used to hold the "To:" field. This is done so that a "buddy list" can be added if present. The `[<-` method for comboboxes make this straightforward. For widgets that have a collection of items to select from, the vector and matrix methods are defined to make changing values familiar to R users. (Although the use of the method `[<-` for `glayout` containers can cause confusion, as no `[` method exists due to the indexing referring to coordinates and not indexes.)

The "From:" field uses an editable label. Clicking in the label's text allows its value to be changed. Just hit ENTER when done. [3]

The handler assigned to the "Subject:" field updates the window title every keystroke. The title of the window is updated with the windows `svalue<-` method.

The `svalue` and `svalue<-` methods are the work-horse methods of `gWidgets`. The are used to retrieve the selected value of a widget or set the selected value of a widget. One advantage to have a single generic function do this is illustrated in the handling of a draft:

```
sapply(c("to","from","subject","text"), function(i)
  svalue(widgets[[i]]) <- draft[[i]])
```

(Another work-horse method is `addHandlerChanged` which can be used to add a handler to any widget, where "changed" – being a generic function – is loosely interpreted: i.e., for buttons, it is aliased to `addHandlerClicked`.

As for the `sendIt` function, this is just one way to send an e-mail message on a UNIX machine. There are likely more than 100 different ways clever people could think of doing this task, most better than this one.

To make portable code, when filling in the layout container, the widget constructors use the layout container as the parent container for the new widget.

11 An expanding group

As an example of the `add` and `delete` methods of a container, we show how one could create the `gexpandgroup` widget, were it not implemented by the underlying toolkit (eg. **gWidgetstcltk**). This container involves an icon to click on to show the container and a similar icon to hide the container. A label indicates to the user what is going on.

```
>   ## expand group
>   rightArrow <- system.file("images/1rightarrow.gif",package="gWidgets")
>   downArrow <- system.file("images/1downarrow.gif",package="gWidgets")
>   g <- ggroup(horizontal=FALSE,cont=T)
>   g1 <- ggroup(horizontal=TRUE, cont=g)
>   icon <- gimage(downArrow,cont=g1)
>   label <- glabel("Expand group example", cont=g1)
```

[3] Editable labels seemed like a good idea at the time – they were borrowed from gmail's interface – but in experience they seem to be confusing to users. YMMV.

```
>   g2 <- ggroup(cont=g, expand=TRUE)
>   expandGroup <- function() add(g,g2, expand=TRUE)
>   hideGroup <- function() delete(g,g2)
>   state <- TRUE                              # a global
>   changeState <- function(h,...) {
+     if(state) {
+       hideGroup()
+       svalue(icon) <- rightArrow
+     }  else {
+       expandGroup()
+       svalue(icon) <- downArrow
+     }
+     state <<- !state
+   }
> addHandlerClicked(icon, handler=changeState)
> addHandlerClicked(label, handler=changeState)
> gbutton("Hide by clicking arrow", cont=g2)

guiWidget of type: gButtonRGtk for toolkit: guiWidgetsToolkitRGtk2

>
```

As an alternative to using the global `state` variable, the `tag` function could be used. This is like `attr` only it stores the value with the widget – not a copy – so can be used within a handler to propogate changes outside the scope of the handler call. For instance, the `changeState` function could have been written as

```
> tag(g,"state") <- TRUE                              # a global
> changeState <- function(h,...) {
+   if(tag(g,"state")) {
+     hideGroup()
+     svalue(icon) <- rightArrow
+   }  else {
+     expandGroup()
+     svalue(icon) <- downArrow
+   }
+   tag(g,"state") <- !tag(g,"state")
+ }
```

Other alternatives to using global variables include using environments or the **proto** package for `proto` objects. These objects are passed not by copy, but by reference so changes within a function are reflected outside the scope of the function.

12 Drag and drop

GTK supports drag and drop features, and the `gWidgets` API provides a simple mechanism to add drag and drop to widgets. (Some widgets, such as text boxes, support drag and drop without these in GTK.) The basic approach is to add a drop source to the widget you wish to drag from, and add a drop target to the widget you want to drag to. You can also provide a handler to deal with motions over the drop target. See the man page `?gWidgetsRGtk-dnd` for more information.

[In **gWidgetsQt** drag and drop support is very limited.]

[In **gWidgetsrJava** drag and drop is provided through Java and not **gWidgets**. One can drag from widget to widget, but there is no way to configure what happens when a drop is made, or what is dragged when a drag is initiated.]

[In **gWidgetstcltk** drag and drop works but not from other applications.]

We give two examples of drag and drop. One where variables from the variable browser are dropped onto a graph widget. Another illustrating drag and drop from the data frame editor to a widget.

12.1 DND with plots

This example shows the use of the plot device, the variable browser widget, and the use of the drag and drop features of gWidgets (Figure 6).

```
> doPlot <-  function() {
+   ## Set up main group
+   mainGroup <- ggroup(container=gwindow("doPlot example"))
+
+   ## The variable browser widget
+   gvarbrowser(container = mainGroup)
+
+   rightGroup <- ggroup(horizontal=FALSE, container=mainGroup)
+
+   ## The graphics device
+   ggraphics(container=rightGroup)
+   entry <- gedit("drop item here to be plotted", container=rightGroup)
```

```
+   adddroptarget(entry,handler = function(h,...) {
+     do.call("plot",list(svalue(h$dropdata),main=id(h$dropdata)))
+   })
+ }

>  doPlot()
```

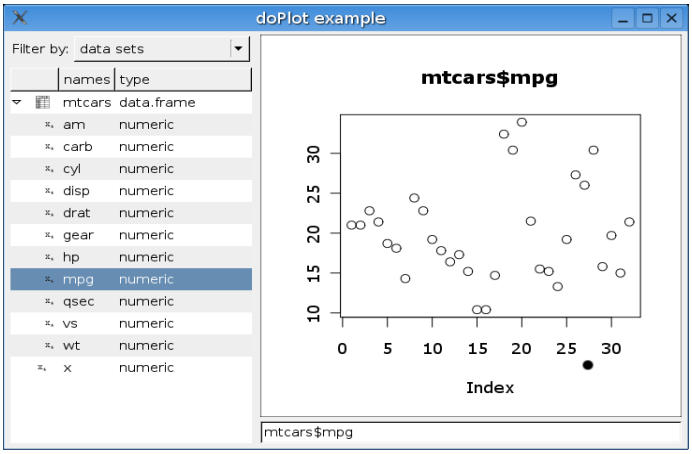

Figure 6: Dialog produced by `doPlot` example

The basic structure of using `gWidgets` is present in this example. The key widgets are the variable browser (`gvarbrowser`), the plot device (`ggraphics`), and the text-entry widget (`gedit`). (One can see an **RGtk2** bias here, as the other implementations do not currently have an embeddable graphics device.) These are put into differing containers. Finally, there is an handler given to the result of the drag and drop. The `do.call` line uses the `svalue` and `id` methods on a character, which in this instance returns the variable with that name and the name.

To use this widget, one drags a variable to be plotted from the variable browser over to the area below the plot window. The `plot` method is called on the values in the dropped variable.

The `gvarbrowser` produces a widget from which a drop *source* has been added. To drop from a different source the `addDropSource` method is used. The return value of the handler will be passed to the `dropdata` value of the handler for `addDropTarget`.

[The latter varies from toolkit to toolkit, in **gWidgetsrJava** both are ignored, in RGtk2 some widgets have built-in drag and drop which can be overridden, and in **gWidgetstcltk** everything must be done using the **gWidgets** interface, as drag and drop is not naturally implemented by the toolkit.]

12.2 DND from the data frame editor

[This example applies only to **gWidgetsRGtk**, not **gWidgetsrJava** or **gWidgetstcltk**]

The `gdf` constructor makes a widget for editing data frames. The columns of which can be dropped onto a widget. This is done by dragging the column header. The code below also adds a handler so that changes to the column propagate to changes in the widget where the column is dropped. (Careful, this has some issues: the handler needs to be removed if the widget is closed.)

```
> ## Drag a column onto plot to have a boxplot drawn.
> ## Changing the column values will redraw the graph.
>
> makeDynamicWidget <-  function() {
+
+   win <- gwindow("Draw a boxplot")
+   gd <- ggraphics(container = win)
+
+   adddroptarget(gd, targetType="object", handler=function(h,...) {
+     tag(gd,"data") <- h$dropdata
+     plotWidget(gd)
+
+     ## this makes the dynamic part:
+     ## - we put a change handler of the column that we get the data from
+     ## - we store the handler id, so that we can clean up the handler when this
+     ##   window is closed
+
+     ## The is.gdataframecolumn function checks if the drop value
+     ##   comes from the data frame editor (gdf)
+     if(gWidgetsRGtk2:::is.gdataframecolumn(h$dropdata)) {
+       view.col <- h$dropdata
+
+       ## Put change handler on column to update plotting widget
+       ## (use lower case, to fix oversight)
+       id <- addhandlerchanged(view.col, handler=function(h,...) plotWidget(gd))
+       ## Save drop handler id so that it can be removed when
+       ##   widget  is closed
+       dropHandlers <- tag(gd,"dropHandlers")
+       dropHandlers[[length(dropHandlers)+1]] <-
```

```
+         list(view.col = view.col,
+              id = id
+              )
+       tag(gd,"dropHandlers") <- dropHandlers
+     }
+   })
+
+   ## Remove drop handlers if widget is unrealized.
+   addHandlerUnrealize(gd, handler = function(h,...) {
+     dropHandlers <- tag(gd,"dropHandlers")
+     if(length(dropHandlers) > 0) {
+       for(i in 1:length(dropHandlers)) {
+         removehandler(dropHandlers[[i]]$view.col,dropHandlers[[i]]$id)
+       }
+     }
+   })
+ }
```

Next, we make the function that produces or updates the graphic. The data is stored in the tag-key "data". The use of `id` and `svalue` works for values which are either variable names or columns.

```
> plotWidget <- function(widget) {
+   data <- tag(widget, "data")
+   theName <- id(data)
+   values <- svalue(data)
+   boxplot(values, xlab=theName, horizontal=TRUE, col=gray(.75))
+ }
```

Now show the two widgets, the `gdf` function constructs the data frame editor widget.

```
> gdf(mtcars, container=TRUE)

guiWidget of type: gGridRGtk for toolkit: guiWidgetsToolkitRGtk2

> makeDynamicWidget()
```

13 Notebooks

The notebook is a common metaphor with computer applications, as they can give access to lots of information compactly on the screen. The `gnotebook` constructor produces a notebook widget. New pages are added via the `add` method (which is called behind the scenes when a widget is constructed), the current page is deleted through an icon [not implemented in **gWidgetsrJava**], or via the `dispose` method, and vector methods are defined, such as `names`, to make interacting with notebooks natural (the names refer to the tab labels.) One can also add pages when constructing a widget using the notebook as the container and passing in an extra argument `label`.

The following example shows how a notebook can be used to organize different graphics devices.

[In **gWidgetsRGtk2** the `ggraphicsnotebook` function produces a similar widget. However, this is not possible if using **gWidgetsrJava**, as the graphic devices can't currently be embedded in a notebook page.]

Our widget consists of a toolbar to add or delete plots and a notebook to hold the different graphics devices. The basic widgets are defined by the following:

First we make window and group containers to hold our widgets and then a notebook instance.

```
> win <- gwindow("Plot notebook")
> group <- ggroup(horizontal = FALSE, container=win)
> nb <- gnotebook(container = group, expand=TRUE)
```

(`expand=TRUE` causes the child widget – the notebook – to expand to take as much space as allotted.)

Next, we begin with an initial plot device.

```
> ggraphics(container = nb,  label="plot")
```

The `label` argument goes on the tab, as it is passed to the notebook's `add` method.

Now we define and add a toolbar.

```
> tblist <- list(quit=gaction("Quit", icon="quit", handler=function(...) dispose(wi
+                separator=gseparator(),
+                new=gaction("New", icon="new", handler=function(h,...) add(nb,ggra
+                delete=gaction("Delete", icond="delete", handler=function(...) dis
+                )
> gtoolbar(tblist, cont=group)

guiWidget of type: gToolbarRGtk for toolkit: guiWidgetsToolkitRGtk2
```

The `dispose` method is used both to close the window, and to close a tab on the notebook (the currently selected one).

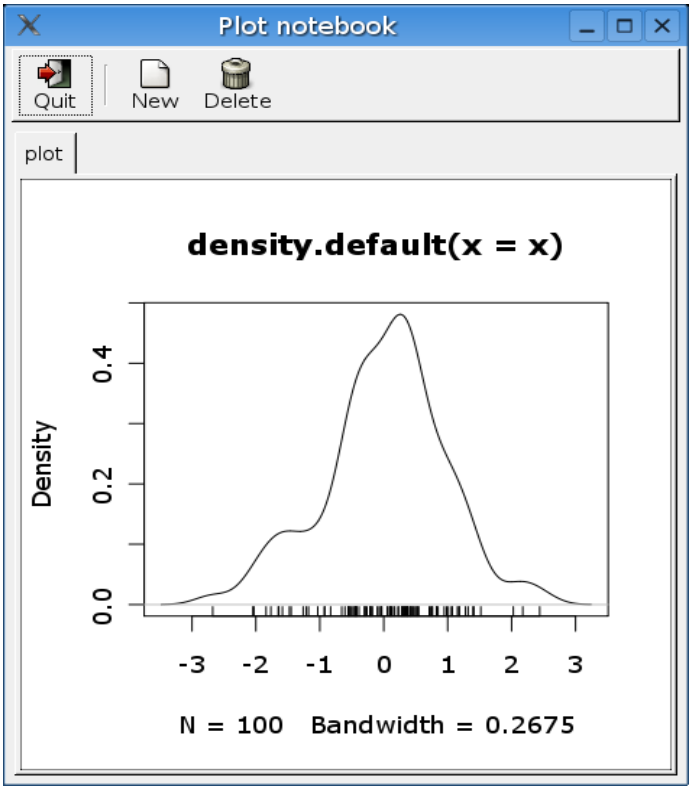

Figure 7: Notebook widget for holding multiple plot devices provided by `ggraphics`

That's it (Figure 7). There is one thing that should be added. If you switch tabs, the active device does not switch. This happens though if you click in the plot area. To remedy this, you can think about the `addHandlerChanged` method for the notebook, or just use `ggraphicsnotebook`.

14 The tree widget

The `gtree` constructor [this widget is only implemented in **gWidgetsRGtk2**.] is used to present tree-like data. A familiar example of such data is the directory structure of your computer. To display a tree, `gtree` needs to know what to display (the offspring) and which offspring have further offspring. idea of a node which consists of a path back to a root node. The offspring are determined by a function

(offspring) which takes the current path (ancestor information), and a passed in parameter as arguments. These offspring can either have subsequent offspring or not. This information must be known at the time of displaying the current offspring, and is answered by a function (hasOffspring) which takes as an argument the offspring. In our file-system analogy, offspring would list the files and directories in a given directory, and hasOffspring would be TRUE for a directory in this listing, and FALSE for a file. For decorations, a function icon.FUN can be given to decide what icon to draw for which listing. This should give a stock icon name.

The data presented for the offspring is a data frame, with one column determining the path. This is typically the first column, but can be set with chosencol=.

To illustrate, we create a file system browser using gtree. (This is for UNIX systems. change the file separator for windows.)

First to define the offspring function we use the file.info function. The current working directory is used as the base node for the tree:

```
>   ## function to find offspring
>   offspring <- function(path, user.data=NULL) {
+     if(length(path) > 0)
+       directory <- paste(getwd(),"/",paste(path,sep="/", collapse=""),sep="",coll
+     else
+       directory <- getwd()
+     tmp <- file.info(dir(path=directory))
+     files <- data.frame(Name=rownames(tmp), isdir=tmp[,2], size=as.integer(tmp[,1
+     return(files)
+   }
```

The offspring function is determined by the isdir column in the offspring data frame.

```
> hasOffspring <- function(children,user.data=NULL, ...) {
+   return(children$isdir)
+ }
```

Finally, an icon function can be given as follows, again using the isdir column.

```
> icon.FUN <- function(children,user.data=NULL, ...) {
+   x <- rep("file",length= nrow(children))
+   x[children$isdir] <- "directory"
+   return(x)
+ }
```

The widget is then constructed as follows. See Figure 8 for an example.

```
>   gtree(offspring, hasOffspring, icon.FUN = icon.FUN,
+         container=gwindow(getwd()))

guiWidget of type: gTreeRGtk for toolkit: guiWidgetsToolkitRGtk2
```

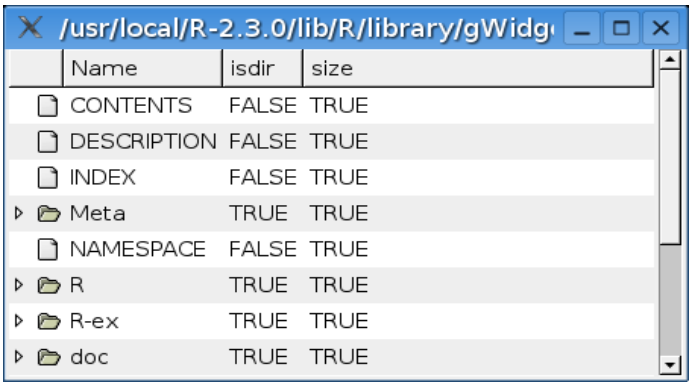

Figure 8: Illustration of a file browser using `gtree` constructor.

The presence of the `isdir` column may bug some. It was convenient when defining `hasOffspring` and `icon.FUN`, but by then had served its purpose. One way to eliminate it, is to use the default for the `hasOffspring=` argument which is to look for the second column of the data frame produced by `offspring`. If this column is of class `logical`, it is used to define `hasOffspring` and is then eliminated from the display. That is, the following would produce the desired file browser:

```
> gtree(offspring,  icon.FUN = icon.FUN,
+       container=gwindow(getwd()))

guiWidget of type: gTreeRGtk for toolkit: guiWidgetsToolkitRGtk2
```

Finally, the `handler=` argument (or `addHandlerDoubleclick`) could have been used to give an action to double clicking of an item in the tree.

15 Popup menus

A popup menu "pops" up a menu after a mouse click, typically a right mouse click. Implemented in **gWidgets** are the functions

`add3rdmousepopupmenu` for adding a popup on a right click

`addpopupmenu` for adding a popup on any click

The menu is specified using the syntax for `gmenu`.

A simple example would be something like:

```
> w <- gwindow("Click on button to change")
> g <- ggroup(cont = w)                    # abbreviate container
> glabel("Hello ", cont=g)

guiWidget of type: gLabelRGtk for toolkit: guiWidgetsToolkitRGtk2

> world <- gbutton("world", cont=g)
> lst <- list()
> lst$world$handler <- function(h,...) svalue(world) <- "world"
> lst$continent$handler <- function(h,...) svalue(world) <- "continent"
> lst$country$handler <- function(h,...) svalue(world) <- "country"
> lst$state$handler <- function(h,...) svalue(world) <- "state"
> addPopupmenu(world, lst)
```

Clicking on "world" with the mouse allows one to change the value in the label.

16 Defining layouts with `gformlayout`

The `gformlayout` constructor allows one to define the layout of a GUI using a list. The details of the list are in the man page, but the list has a relatively simple structure. Each widget is specified using a list. The widget type is specified through the `type` component as a string. These may be containers or widgets or the special container `fieldset`. For containers, the component `children` is used to specify the children to pack in. These again are specified using a list, and except for the `fieldset` container may again contain containers. The `name` argument is used to have the newly created component stored in a list of widgets for later manipulations. There are means to control whether a component is enabled or not based on a previously defined component, such as a checkbox. (This works much better under **gWidgetsRGtk2** than **gWidgetstcltk** as disabled parent containers do not disable their children in the latter.)

This example comes from the man page. It shows how the "fieldset" container is used with some of its arguments to modify the number of columns and adjust the default label placements.

We present the final list, `tTest` using some predefined lists that could be recycled for other GUIs if desired.

```
> ## layout a collection of widgets to generate a t.test
>
> ## widgets to gather the variable(s)
> varList <- list(type="fieldset",
+                 columns = 2,
+                 label = "Variable(s)",
+                 label.pos = "top",
+                 label.font = c(weight="bold"),
+                 children = list(
+                   list(name = "x",
+                        label = "x",
+                        type = "gedit",
+                        text = ""),
+                   list(name = "y",
+                        label = "y",
+                        type = "gedit",
+                        text = "",
+                        depends.on = "x",
+                        depends.FUN = function(value) nchar(value) > 0,
+                        depends.signal = "addHandlerBlur"
+                        )
+                        )
+                 )
> ## list for alternative
> altList <- list(type = "fieldset",
+                 label = "Hypotheses",
+                 columns = 2,
+                 children = list(
+                   list(name = "mu",
+                        type = "gedit",
+                        label = "Ho: mu=",
+                        text = "0",
+                        coerce.with = as.numeric),
```

```
+                   list(name = "alternative",
+                        type="gcombobox",
+                        label = "HA: ",
+                        items = c("two.sided","less","greater")
+                        )
+                   )
+                 )
> ## now make t-test list
> tTest <- list(type = "ggroup",
+               horizontal = FALSE,
+               children = list(
+                 varList,
+                 altList,
+                 list(type = "fieldset",
+                      label = "two sample test",
+                      columns = 2,
+                      depends.on = "y",
+                      depends.FUN = function(value) nchar(value) > 0,
+                      depends.signal = "addHandlerBlur",
+                      children = list(
+                        list(name = "paired",
+                             label = "paired samples",
+                             type = "gcombobox",
+                             items = c(FALSE, TRUE)
+                             ),
+                        list(name = "var.equal",
+                             label = "assume equal var",
+                             type = "gcombobox",
+                             items = c(FALSE, TRUE)
+                             )
+                        )
+                      ),
+                 list(type = "fieldset",
+                      columns = 1,
+                      children = list(
+                        list(name = "conf.level",
+                             label = "confidence level",
+                             type = "gedit",
```

```
+                               text = "0.95",
+                               coerce.with = as.numeric)
+                         )
+                       )
+                   )
+               )

> ## Code to call the layout
> w <- gwindow("t.test")
> g <- ggroup(horizontal = FALSE, cont = w)
> fl <- gformlayout(tTest, cont = g, expand=TRUE)
> bg <- ggroup(cont = g)
> addSpring(bg)
> b <- gbutton("run t.test", cont = bg)
> addHandlerChanged(b, function(h,...) {
+   out <- svalue(fl)
+   out$x <- svalue(out$x) # turn text into numbers
+   if(out$y == "") {
+     out$y <- out$paired <- NULL
+   } else {
+    out$y <- svalue(out$y)
+   }
+   ## easy, not pretty
+   print(do.call("t.test",out))
+ })
>
```

17 Making widgets from an R function

A common task envisioned for **gWidgets** is to create GUIs that make collecting the arguments to a function easier to remember or enter. Presented below are two ways to do so without having to do any programming, provided you are content with the layout and features provided.

17.1 Using `ggenericwidget`

The `ggenericwidget` constructor maps a list into a widget. The list contains two types of information: meta information about the widget, such as the name of the

function, and information about the widgets. This is specified using a list whose first component is the constructor, and subsequent components are fed to the constructor.

To illustrate, a GUI for a one sample t-test is given. The list used by `ggenericwidget` is defined below.

```
> lst <- list()
> lst$title <- "t.test()"
> lst$help <- "t.test"
> lst$variableTypes <- "univariate"
> lst$action <- list(beginning="t.test(",ending=")")
> lst$arguments$hypotheses$mu <-
+   list(type = "gedit",text=0,coerce.with=as.numeric)
> lst$arguments$hypotheses$alternative <-
+   list(type="gradio", items=c("'two.sided'","'less'","'greater'")
+        )
```

This list is then given to the constructor.

```
> ggenericwidget(lst, container=gwindow("One sample t test"))
```

Although this looks intimidating, due to the creation of the list, there is a function `autogenerategeneric` that reduces the work involved. In particular, if the argument to `ggenericwidget` is a character, then it is assumed to be the name of a function. From the arguments of this function, a layout is guessed.

For example, we could have done:

```
>   our.t.test <- stats:::t.test.default
>   ggenericwidget("our.t.test", container=gwindow("t-test"))
```

```
guiWidget of type: gGenericWidgetANY for toolkit: guiWidgetsToolkitRGtk2
```

As the arguments of this function are

```
> args(our.t.test)
```

```
function (x, y = NULL, alternative = c("two.sided", "less", "greater"),
    mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95,
    ...)
NULL
```

This widget has fields for selecting the alternative, the null, whether the data is paired, has equal variance assumption and a field to adjust the confidence level.

17.2 An alternative to ggenericwidget

This next example shows a different (although ultimately similar) way to produce a widget for a function. One of the points of this example is to illustrate the power of having common method names for the different widgets. Of course, the following can be improved. Two obvious places are the layout of the automagically generated widget, and and the handling of the initial variable when a formula is expected.

```
> ## A constructor to automagically make a GUI for a function
> gfunction <- function(f, window = gwindow(title=fName), ...) {
+
+   ## Get the function and its name
+   if(is.character(f)) {
+     fName <- f
+     f <- get(f)
+   } else if(is.function(f)) {
+     fName <- deparse(substitute(f))
+   }
+
+   ## Use formals() to define the widget
+   lst <- formals(f)
+
+
+
+   ## Hack to figure out variable type
+   type <- NULL
+   if(names(lst)[1] == "x" && names(lst)[2] == "y") {
+     type <- "bivariate"
+   } else if(names(lst)[1] == "x") {
+     type <- "univariate"
+   } else if(names(lst)[1] == "formula") {
+     type <- "model"
+   } else {
+     type + NULL
+   }
+
+
+   ##  Layout
+
```

```
+   w <- gwindow("create dialog")
+   g <- ggroup(horizontal = TRUE, cont=w)
+   ## Arrange widgets with an output area
+   ## Put widgets into a layout container
+   tbl <- glayout(container=g)
+   gseparator(horizontal=FALSE, container=g)
+   outputArea <- gtext(container=g, expand=TRUE)
+
+
+   ## Make widgets for arguments from formals()
+   widgets <- sapply(lst, getWidget, cont=tbl)
+
+   ## Layout widgets
+   for( i in 1:length(widgets)) {
+     tbl[i,1] <- names(lst)[i]
+     tbl[i,2] <- widgets[[i]]
+   }
+
+
+   ## Add update handler to each widget when changed
+   sapply(widgets, function(obj) {
+     try(addHandlerChanged(obj, function(h,...) update()), silent=TRUE)
+   })
+
+   ## Add drop target to each widget
+   sapply(widgets, function(obj)
+          try(adddroptarget(obj,
+                            handler=function(h,...) {
+                              svalue(h$obj) <- h$dropdata
+                              update()
+                            }),
+              silent=TRUE))
+
+
+
+   ## In case this doesn't get exported
+   svalue.default <- function(obj, ...) obj
+
```

```
+   ## Function used to weed out 'NULL' values to widgets
+   isNULL <- function(x)
+     ifelse(class(x) == "character" && length(x) ==1 && x == "NULL",
+            TRUE, FALSE)
+
+   ## Function called when a widget is changed
+   ## 2nd and 3rd lines trim out non-entries
+   update <- function(...) {
+     is.empty <- function(x) return(is.na(x) || is.null(x) || x == "" )
+     outList <- lapply(widgets,svalue)
+     outList <- outList[!sapply(outList,is.empty)]
+     outList <- outList[!sapply(outList,isNULL)]
+     outList[[1]] <- svalue(outList[[1]])
+     if(type == "bivariate")
+       outList[[2]] <- svalue(outList[[2]])
+
+     out <- capture.output(do.call(fName,outList))
+
+     dispose(outputArea)
+     if(length(out)>0)
+       add(outputArea, out)
+   }
+   invisible(NULL)
+ }
```

The `getWidget` function takes a value from `formals` and maps it to an appropriate widget. For arguments of type `call` the function recurses.

```
> getWidget <- function(x, cont=cont) {
+   switch(class(x),
+          "numeric" = gedit(x, coerce.with=as.numeric, cont=cont),
+          "character" = gcombobox(x, active=1, cont=cont),
+          "logical" = gcombobox(c(TRUE,FALSE), active = 1 + (x == FALSE), cont=con
+          "name" = gedit("", cont=cont),
+          "NULL" = gedit("NULL", cont=cont),
+          "call" = getWidget(eval(x), cont=cont), # recurse
+          gedit("", cont=cont)                    # default
+     )
+ }
```

We can try this out on the default `t.test` function. First we grab a local copy from the namespace, then call our function. The widget with an initial value for `x` is shown in Figure 9.

```
> our.t.test <- stats:::t.test.default
> gfunction(our.t.test)
```

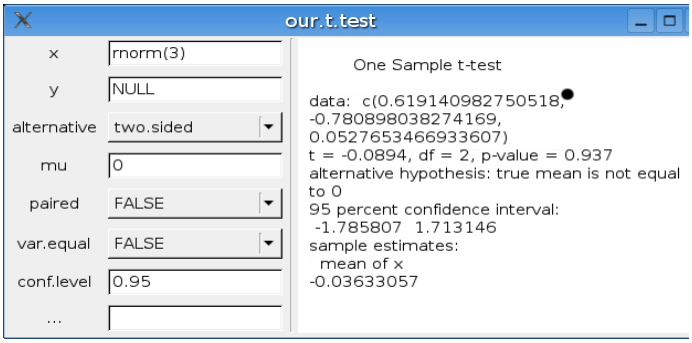

Figure 9: Illustration of `gfunction`