Examples for gWidgets

John Verzani, `gWidgetsRGtk@gmail.com`

April 8, 2007

Abstract:

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, there are two available toolkits. The GTK toolkit is implemented via the **gWidgetsRGtk2** package which in turn uses the **RGtk2** package. JAVA can be used via the **gWidgetsrJava** package, which in turn calls **rJava**. The **gWidgetsRGtk2** implementation is much more complete, as the **gWidgetsrJava** package is lacking many features and must run from within a JGR (`http://www.rosuda.org`) session.

The API is inteneded to faciliatate 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.

Contents

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 improvement by Philippe Grosjean, Michael Lawrence, Simon Urbanek and John Verzani. This document focuses on the more complete toolkit implementation provided by the **gWidgetsRGtk2** package. [Occasional differences with **gWidgetsrJava** are pointed out in braces.] 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 still in the formative stages and likely will change as more people use it and offer suggestions for improvement.

This document supplements the man pages by providing more detailed examples. The man pages contain more specific information. See the page `gWidgets-package` for a listing of the available man pages.

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

2 Installation

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

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

2.1 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 (2.8.0 or higher). 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`

```
> 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.

For Mac OS X, this author has installed GTK libraries from source on an older 10.3.9 machine using Apple's X11 server. It may be possible in 10.4 to install using fink or Darwin ports, or it may be possible to run the native GTK libraries. None of these has been tested.

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

2.2 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 packages are provided as source, not the default "mac.binary." Use the `type=` argument, as follows:

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

On occasion, newer versions are available from gWidgets's website. To install from here add the `repos=` argument, as follows:

```
> install.packages("gWidgetsRGtk2",dep = TRUE, repos = "http://www.math.csi.cuny.edu/
```

[The **gWidgetsrJava** package is installed similarly. However, it requires **rJava** and **JGR** and friends to work properly. These should install through the dependencies, so

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

should do it.]

3 Loading gWidgets

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

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

```
[1] TRUE
```

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 **gWidgetsrJava** use "rJava" in the `options()` call.]

Both the **gWidgets** and **gWidgetsRGtk2** package use S4 methods and classes and load much faster under the newer **methods** package accompanying R version 2.4.0 or greater.

[For **gWidgetsrJava** the interactive features only work if run from within a `JGR` session. For this, one starts `JGR()`, then uses its console to load **gWidgets**.]

4 Hello world

We begin by showing how to make various widgets which display the ubiquitous "Hello world" message. First though we define a function allowing us to comment code within Sweave.

```
> Comment = function(...) invisible(...)
```

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 ediable text:

```
> 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())
```

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

Next, a drop list, or combo box, again for selecting just one of several, although in this case an option can be give for the user to edit the value.

```
> obj = gdroplist(c("hello", "world"), container = gwindow())
```

A drop list can also allow its value to be entered,

```
> obj = gdroplist(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()`.

4.1 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 = gdroplist(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. The button looks like a button. A label is used to show text which may perhaps be edited. A droplist 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.

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()`. 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`.

Figure 2: Hello world example

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.

The message is an instance of a dialog. in the **gWidgets** API dialogs are usually modal, meaning nothing can be done until they are dismissed. (This can be annoying if a dialog appears under another window and can't be seen!)

5 Making a confirmation dialog

Let's see how we might use widgets to create our own confirmation dialog. 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)
```

```
+     add(group, gimage("info", dirname = "stock", size = "dialog"))
### A group for the message and buttons
+     innner.group = ggroup(horizontal = FALSE, container = group)
+     add(innner.group, glabel(message), expand = TRUE)
### A group to organize the buttons
+     button.group = ggroup(container = innner.group)
+     addSpring(button.group)        # Push buttons to right
+     obj = gbutton("ok", handler = handler, container = button.group)
+     obj = 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, with one of the `addhandler` 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

Figure 3: Confirmation dialog

```
> 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
```

6 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.

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

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, a `[` method for referring to the widgets comprising the notebook pages, and a `length()` method to return the number of notebook pages.

7 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`.

7.1 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 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$action, list(svalue(h$obj)))
+     })
+ }
```

The `action=` argument parameterizes the action. The default above calls `print()` on the selected file 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...")
```

7.2 browseEnv()

The browseEnv() function creates a table in a web browser listing the current objects in the global environement (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 (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.

```
> 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$action, list(svalue(h$obj))))
+         })
+ }
```

As a final refinement, we add a droplist box to filter by the unique values of "Class." We leave as an excercise 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$action, list(svalue(h$obj))))
+         })
+ }
```

The `gvarbrowser()` function constructs a widget very similar to this, only it uses `gtree()` to allow further display of list-like objects.

8 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.

We use the `ggraphics()` constructor to create a new plot device. For RGtk2, this uses the **cairoDevice** package also developed by Michael Lawrence.

[In **gWidgetsrJava** the **JavaGD** package is used for a JAVA device. This implementation does not currently allow the device to be embedded in a widget, so the `gimage()` constructor below would make a widget in a separate window.]

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.

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

```
> 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 = gdroplist(availKernels, handler = updatePlot)
> bandwidthAdjust = gslider(from = 0, to = 2, by = 0.01, value = 1,
+     handler = updatePlot)
> sampleSize = gradio(c(50, 100, 200, 300), handler = updatePlot)
```

And 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, with a default using the left-hand side.

```
> window = gwindow("gWidgetsDensity")
> BigGroup = ggroup(cont = window)
> group = ggroup(horizontal = FALSE, container = BigGroup)
> tmp = gframe("Distribution", container = group)
> add(tmp, distribution)
> 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.

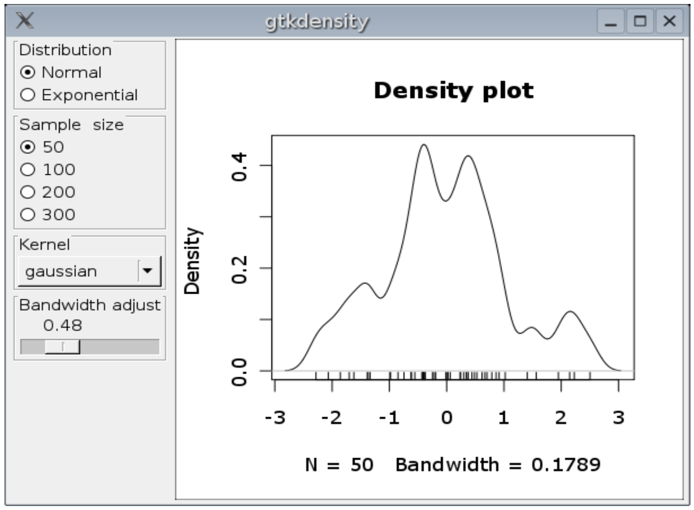

Figure 4: The gWidgetsDensity example in action.

9 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 (especially with Sweave's formatting), 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-mails; 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.

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, ...) {
### Define main widgets, store in a list for ease of use
+     widgets = list()
+     widgets$to = gdroplist(c(), editable = TRUE)
+     widgets$from = glabel(FROM, editable = TRUE)
+     widgets$subject = gedit()
+     widgets$text = gtext()
### 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))
+             source(draft)
+         if (is.list(draft))
+             sapply(c("to", "from", "subject", "text"), function(i) svalue(widgets
+     }
+     sendIt = function(...) {       # Helper functions
+         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)
+         unlink(tmp)                # Close window, delete file
+         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]]) <- sva
+         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)
+     svalue(group) <- 0             # Remove border
### Menubar is defined by a list
+     menubarlist = list()
+     menubarlist$File$Save$handler = saveDraft
+     menubarlist$File$Send$handler = sendIt
+     menubarlist$File$Quit$handler = function(...) dispose(window)
+     menubarlist$File$Quit$icon = "quit"
+     menubarlist$Help$About$handler = aboutMail
+     add(group, gmenu(menubarlist))
### Toolbar is also defined by a list
+     toolbarlist = list()
+     toolbarlist$Send$handler = sendIt
+     toolbarlist$Send$icon = "connect"
+     toolbarlist$Save$handler = saveDraft
+     toolbarlist$Save$icon = "save"
+     add(group, gtoolbar(toolbarlist))
### Put headers in a glayout() container
+     tbl = glayout(container = group)
### To: field. Looks for buddyList
+     tbl[1, 1] = glabel("To:")
+     tbl[1, 2] = widgets$to
+     if (exists("buddyList"))
+         widgets$to[] <- buddyList
### From: field. Click to edit value
+     tbl[2, 1] = glabel("From:")
+     tbl[2, 2] = widgets$from
### Subject: field. Handler updates window title
```

```
+     tbl[3, 1] = glabel("Subject:")
+     tbl[3, 2] = widgets$subject
+     addhandlerkeystroke(widgets$subject, handler = function(h,
+         ...) svalue(window) = paste("Compose mail:", svalue(h$obj),
+         collapse = ""))
### Layout needs to be finalized
+     visible(tbl) <- TRUE
### Add text box for message, but first some space
+     addSpace(group, 5)
+     add(group, widgets$text, expand = TRUE)
+ }                                     # That's it.
```

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

```
> Rmail()
```

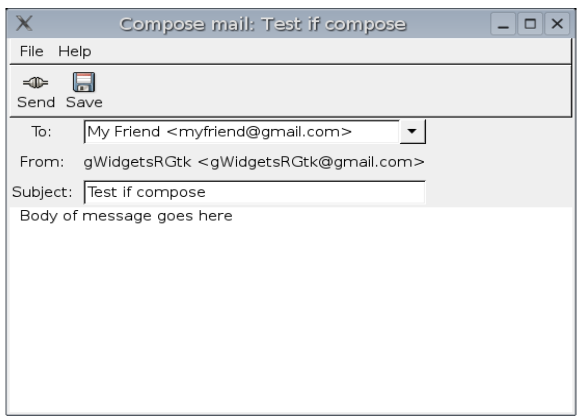

Figure 5: Widget for composing an e-mail message

The `Rmail()` function uses a few tricks. A droplist is used to hold the "To:" field. This is done so that a "buddy list" can be added if present. The `[<-` method for drop lists 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.

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

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" is loosely interpreted: i.e., for buttons, its 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.

10 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 **gWidgetsrJava** drag and drop is not fully implemented. 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.]

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.

10.1 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()
```

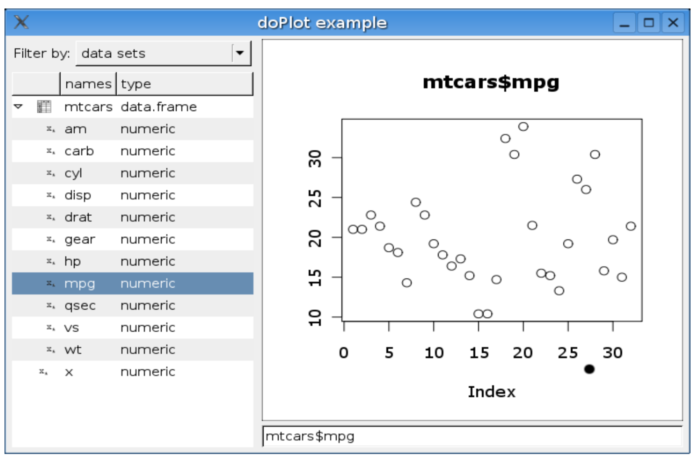

Figure 6: Dialog produced by `doPlot()` example

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()`). 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 return 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.

10.2 DND from the data frame editor

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

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 propogate 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$dropdata
+         plotWidget(gd)
### this makes the dynamic part:
### - we put a change handler of the column that we get the data from
### - we store the handler id, so that we can clean up the handler when this
###   window is closed
### The is.gdataframecolumn function checks if the drop value
###   comes from the data frame editor (gdf)
+         if (is.gdataframecolumn(h$dropdata)) {
+             view.col = h$dropdata
### Put change handler on column to update plotting widget
+             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(0.75))
+ }
```

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

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

11 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, 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 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()
```

Next, we begin with an initial plot device.

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

The `add()` method is used to add new widgets, in this case a graphics device. The label goes on the tab.

Now we define and add a toolbar.

```
> tblist = list()
> tblist$Quit$handler = function(h, ...) dispose(win)
> tblist$Quit$icon = "quit"
> tblist$tmp1$separator = TRUE
> tblist$New$handler = function(h, ...) add(nb, ggraphics(), label = "plot")
> tblist$New$icon = "new"
> tblist$Delete$handler = function(h, ...) dispose(nb)
> tblist$Delete$icon = "delete"
> add(group, gtoolbar(tblist))
```

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

Finally we add the notebook.

```
> add(group, nb, expand = TRUE)
```

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()`.

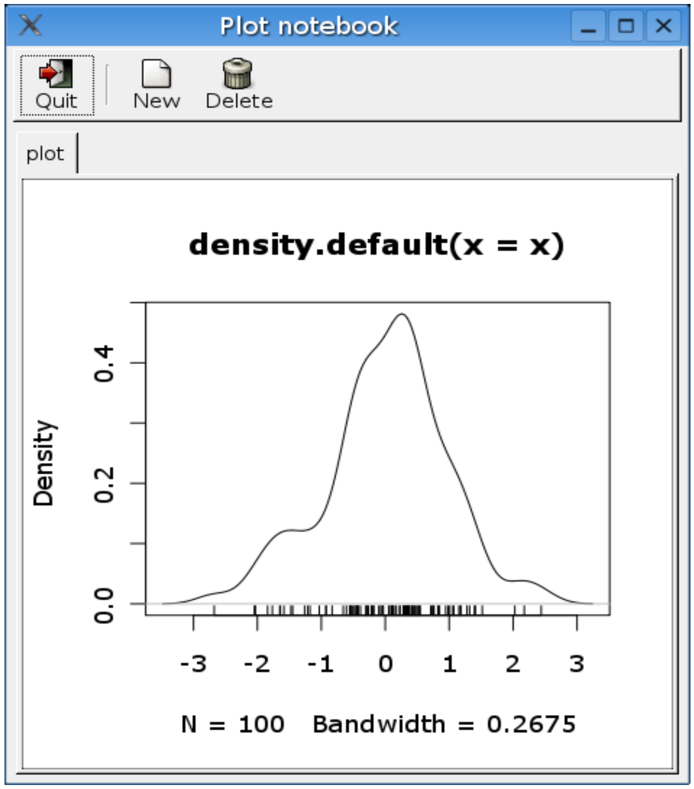

Figure 7: Notebook widget for holding multiple plot devices provided by `ggraphics()`

12 The tree widget

The `gtree()` constructor is used to present tree-like data. A familiar example of such data is the directory structure of your computer. To describe a tree, `gtree()` has the idea of a node which consists of a path back to a root node. This node can have offspring which will be determined by a function (`offspring()`) which takes the current path, 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.

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=`.

[The `tree()` widget is not implemented in **gWidgetsrJava**. The default JAVA tree only holds a single item, so a new table-tree class needs to be written.]

To illustrate, we create a file system browser using `gtree()`.

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:

```
> offspring = function(path, user.data = NULL) {
+     if (length(path) > 0)
+         directory = paste(getwd(), "/", paste(path, sep = "/",
+             collapse = ""), sep = "", collapse = "")
+     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
```

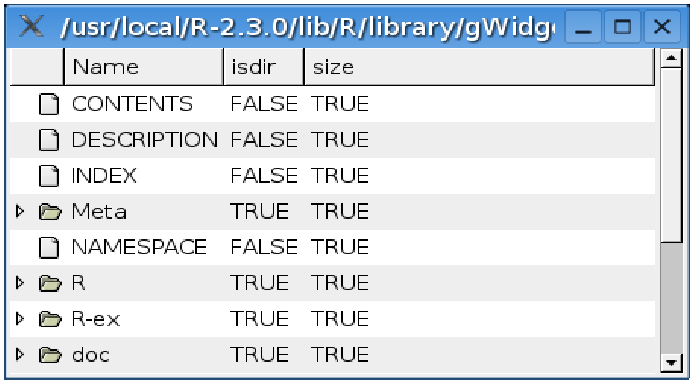

Figure 8: Illustration of a file browser using `gtree()` constructor.

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 is 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.

13 Popup menus

A popup menu "pops" up a menu after a mouse click, typically a right mouse click. Implemented here 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:

```
> group = ggroup(container = gwindow("Click on button to change"))
> glabel("Hello ", container = group)
```

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

```
> world = gbutton("world", container = group)
> 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.

14 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.

14.1 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"))

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

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: gGenericWidgetRGtk 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.

14.2 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
+     }
### Make widgets for arguments from formals
+     widgets = sapply(lst, getWidget)
### 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))
### Put widgets into a layout container
+     tbl = glayout()
```

```
+     for (i in 1:length(widgets)) {
+         tbl[i, 1] = glabel(names(lst)[i])
+         tbl[i, 2] = widgets[[i]]
+     }
### Finalize the layout container
+     visible(tbl) <- TRUE
### Main group
+     gp = ggroup(horizontal = TRUE, container = window)
### Arrange widgets with an output area
+     add(gp, tbl)
+     gseparator(horizontal = FALSE, container = gp)
+     outputArea = gtext()
+     add(gp, outputArea, expand = 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(...) {
+         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) {
+     switch(class(x), numeric = gedit(x, coerce.with = as.numeric),
+         character = gdroplist(x, active = 1), logical = gdroplist(c(TRUE,
+             FALSE), active = 1 + (x == FALSE)), name = gedit(""),
+         "NULL" = gedit("NULL"), list = gListOfWidgets(x, name = ""),
+         call = getWidget(eval(x)), gedit())
+ }
```

This function defines a separate widget to handle the case where an argument expects a list. It is written in the `gWidgetsRGtk` style including an `svalue()` method below. The `tag()` method stores a value in the widget, similar to setting an attribute. In this case, the list of widgets stored is consulted by the following `svalue()` method.

```
> gListOfWidgets = function(lst, name = "", container = NULL, ...) {
+     gp = gframe(text = name, container = container, horizontal = FALSE,
+         ...)
+     obj = list(ref = gp)
+     class(obj) = c("gListOfWidgets", "gComponent", "gWidget")
+     widgetList = lapply(lst, getWidget)
+     tag(obj, "widgetList") <- widgetList
+     tbl = glayout(container = gp)
+     for (i in 1:length(widgetList)) {
+         tbl[i, 1] = glabel(names(widgetList)[i])
+         tbl[i, 2] = widgetList[[i]]
+     }
+     visible(tbl) <- TRUE
+     return(obj)
+ }
```

The methods below (`svalue()`, `svalue<-()` and `addhandlerchanged()`) map the same method to each component of the list using `sapply()`.

```
> svalue.gListOfWidgets = function(obj, ...) {
+     lst = lapply(tag(obj, "widgetList"), svalue)
+     return(lst)
+ }
> "svalue<-.gListOfWidgets" = function(obj, ..., value) {
+     if (!is.list(value))
+         return(obj)
```

```
+     widgetList = getdata(obj, "widgetList")
+     sapply(names(value), function(x) svalue(widgetList[[x]]) <- value[[x]])
+     return(obj)
+ }
> addhandlerchanged.gListOfWidgets = function(obj, handler = NULL,
+     action = NULL, ...) {
+     widgetList = getdata(obj, "widgetList")
+     sapply(widgetList, function(x) try(addhandlerchanged(x, handler,
+         action), silent = TRUE))
+ }
```

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)

Define length for x of class:[1] "gEditRGtk"
attr(,"package")
[1] "gWidgetsRGtk2"
Define length for x of class:[1] "gEditRGtk"
attr(,"package")
[1] "gWidgetsRGtk2"
Define length for x of class:[1] "gDroplistRGtk"
attr(,"package")
[1] "gWidgetsRGtk2"
Define length for x of class:[1] "gEditRGtk"
attr(,"package")
[1] "gWidgetsRGtk2"
Define length for x of class:[1] "gDroplistRGtk"
attr(,"package")
[1] "gWidgetsRGtk2"
Define length for x of class:[1] "gDroplistRGtk"
attr(,"package")
[1] "gWidgetsRGtk2"
Define length for x of class:[1] "gEditRGtk"
attr(,"package")
[1] "gWidgetsRGtk2"
```

```
Define length for x of class:[1] "gEditRGtk"
attr(,"package")
[1] "gWidgetsRGtk2"
```

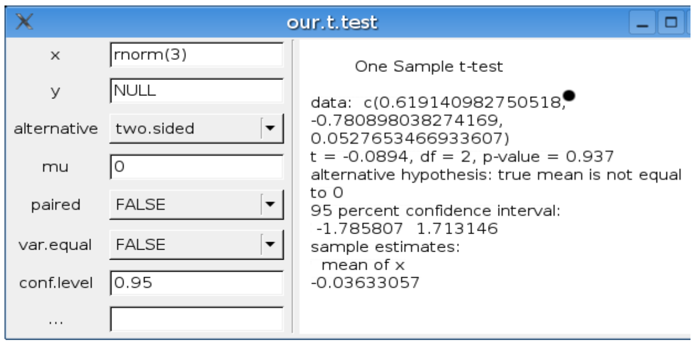

Figure 9: Illustration of `gfunction()`