DSC 2003 Working Papers
(Draft Versions)
http://www.ci.tuwien.ac.at/Conferences/DSC-2003/

Putting RGtk to Work

Jim Robison-Cox*

Abstract

The RGtk package provides one way to build statistical applications in R which can be controlled by the user through a graphical user interface. RGtk is an adaptation of the GTK library of the Free Software Foundation (Gnu Project). In this paper an example GUI is built to illustrate usage of the RGtk package. The basic concepts of widgets and callbacks are demonstrated so that those familiar with programming in R will be able to build graphical applications.

1 Motivation

Graphical applications are useful for many purposes, including avoidance of the command line interface for repetitive analyses by non-statisticians, and for development of pedagogical tools. In this paper, the motivation for development will be a particular teaching tool. Any of the computer simulation modules described in Mills (2002) would be appropriate for a demonstration. I have chosen one which goes beyond the statistical capabilities of a spreadsheet, using a resistant regression routine and local smoother. The R statistical program (Ihaka and Gentleman, 1996) will be used for the underlying computation. For the particular unit to be taught, suppose that we are introducing ideas of fitting points to a linear model, and would like students to understand that choices are made in deciding which line is "best-fitting". We want them to see that the least squares line, in particular, is influenced by movement of a single point.

The intended audience for this demonstration is a group of neophytes in statistics; people who would be uncomfortable with a command-line interface. In fact, even if programming skill could be assumed, the ideal pedagogical technique would be graphical rather than command-line oriented so that the student's attention

would be on the picture rather than on the code used to invoke the demonstration. In order to use R from graphical interface rather than its native command line interface, we are changing the types of events to which our program must respond. The R engine expects command line inputs, and, if so instructed, can also process mouse clicks in a plotting region with the `locator` function. In contrast, the GUI approach requires detection of any of possibly many user inputs, either by keystrokes in a text window or by mouse actions. The GUI must translate inputs into calls to R, and output from R commands must be used to update the display.

A variety of methods are available for producing graphical interfaces to R programs. There has been a flurry of interest in GUI's for R in the Fall of 2002 on the R-Help mailing list (`https://www.stat.math.ethz.ch/pipermail/r-help/` continued on `http://www.sciviews.org/_rgui/`) but no agreement on a favorite graphical toolkit. For instance with the tcltk package, one of the earliest unix GUI's for R, Peter Dalgaard (2002) has provided `tkcanvas` which meets many of the objectives for the proposed regression demo. Many other graphical toolkits are being used to develop GUIs with R, for instance, ObveRsive (`http://obversive.sourceforge.net/`), a recent addition, uses the FOX graphical toolkit. Without claiming that gtk (from GIMP ToolKit, `http://www.gtk.org/`) is the best graphical toolkit for use with R, I would note that it is GPL, is available for many platforms including Gnu/Linux, Unix, MSWindows, and MacOSX. Another advantage, as implemented in RGtk by Duncan Temple Lang is `http://www.omegahat.org/RGtk/index.html` that the code makes use of R object classes, and is written entirely in R, avoiding languages such as C. Like most toolkits, it contains a full set of graphical tools, and, unfortunately, the help system is not fully developed.

The remainder of this paper is a brief tutorial describing how the regression demo can be built in RGtk.

2 GTK

2.1 Widgets

The basic building blocks to be combined into a GUI are called widgets. Those of particular interest to the R programmer include menus, buttons, sliders, and text boxes for user input; text boxes and plotting areas for user output. Widgets are typically created, given the desired attributes, and then packed into windows. With RGtk, control over widgets utilizes the class attributes of objects to determine which GTK function to call. For example, in the code below, a simple text widget is created, and text is inserted into the widget.

```
> textx <- gtkText()     ## Creates a widget by calling S_gtk_text_new
> attributes(textx)      ## Check its attributes
$class
[1] "GtkText"     "GtkWidget"   "GtkObject"
> textx$Insert(chars=xtxt, length= nchar(xtxt))            ## or
> gtkTextInsert(textx, chars=xtxt, length= nchar(x))
             ## both call the function S_gtk_text_insert
```

The last two lines perform the same action because `textx$Insert` is interpreted by utilizing the class attribute of `textx`, GtkText, and is translated into a call to `gtkTextInsert`. Similarly one could create an object of class `GtkMenu` called `menu1` and use `menu1$Insert()` to invoke the `gtkMenuInsert` function, a binding to `S_gtk_menu_insert`. The code which follows uses the shortened forms wherever possible.

The process of developing a graphical application involves:

1. Creating appropriate widgets,
2. Displaying the widgets,
3. Capturing user-initiated events, and
4. Updating output based on user inputs.

These four steps will be illustrated as the regression demo is built.

2.2 Widget Creation

One starts by creating a window for display. It's easiest to display the window after all its internal constituents have been built, so the show argument is initially set to `FALSE`.

```
wRegDemo <- gtkWindow(show=FALSE)     ## Create Display Window
wRegDemo$SetTitle("Regression Demo")  ## Give it a title
```

Within the window we want several components. These will be "packed" into boxes. Boxes can be either vertical or horizontal stacks of widgets. We will use one of each.

```
box1 <- gtkVBox(show =FALSE) ## Create a Vertical box
wRegDemo$Add(box1)           ## Add it to the display window
box2 <- gtkHBox(show=FALSE)  ## Create a Horizontal box
```

To use only basic widgets available on all platforms, the x and y values for the regression will initially be displayed in text boxes which the user can edit. Separate text boxes are used for each variable.

```
textx <- gtkText()           ## Create a text box
textx$SetEditable(TRUE)      ## Make it user-editable
x <- sort(rnorm(10, 100,10)) ## create some data and a data frame
reg.frame <- data.frame( y = round(x + rnorm(10,0,4)), x=round(x))
xtxt <- paste(c("x", as.character(reg.frame$x)), collapse="\n")
textx$Insert(chars=xtxt, length=  nchar(xtxt))
                             ## Insert the x values into the text box
```

The response (y) values are generated from the x's and inserted as character values into their own text box.

```
texty <- gtkText()
texty$SetEditable(TRUE)
ytxt <- paste(c("y",as.character( reg.frame$y)), collapse="\n")
texty$Insert(chars=ytxt, length=  nchar(ytxt) )
```

Finally, we need a drawing region, so we will create a drawing area widget. In order to use the drawing area as our graphics device, we also need the gtkDevice package.

```
require(gtkDevice)
drawArea <- gtkDrawingArea()  ## Create the widget
drawArea$SetUsize(300,300)    ## specify the size
asGtkDevice(drawArea)         ## set as graphics device for R
```

Note: Use of the DrawingArea device gives R control of the drawing area. The individual points and elements of the graph are not widgets and are not clickable using GTK event control.

2.3 Arranging and Displaying Widgets

For this simple example, the text and drawing area widgets will be displayed side-by-side in the horizontal box, `box2`. The `PackStart` function places them into `box2` from left to right (`PackEnd` would place them from right to left).

```
box2$PackStart(textx);     textx$Show()
box2$PackStart(texty);     texty$Show()
box2$PackStart(drawArea);  drawArea$Show()
```

The horizontal box is ready to be placed into the vertical box, but first a label will be added to give the user minimal instructions.

```
label1 <-  gtkLabel( "Change numerical values to move points.",
                    show=FALSE)
label1$SetJustify("left")
label1$Show()
box1$PackStart(label1)
box1$PackStart(box2)
box2$Show()
box1$Show()
```

The only visual missing from our simple demo is the plot of the points. Since this will be repeated when points are changed, it needs to be a function. The data plotted will always come from the data frame assigned above, so the function needs no arguments.

```
redraw <- function(){
  ## function to plot points and draw regression line
  plot(y ~ x, data = reg.frame,
```

```
        xlab = "x",ylab="y", pch = 16, col=4)
    abline(lm(y~x, data = reg.frame))
}
 redraw()
```

Finally we change the attribute of the display window to make everything appear, as shown in Figure 1.

```
wRegDemo$Show()
```

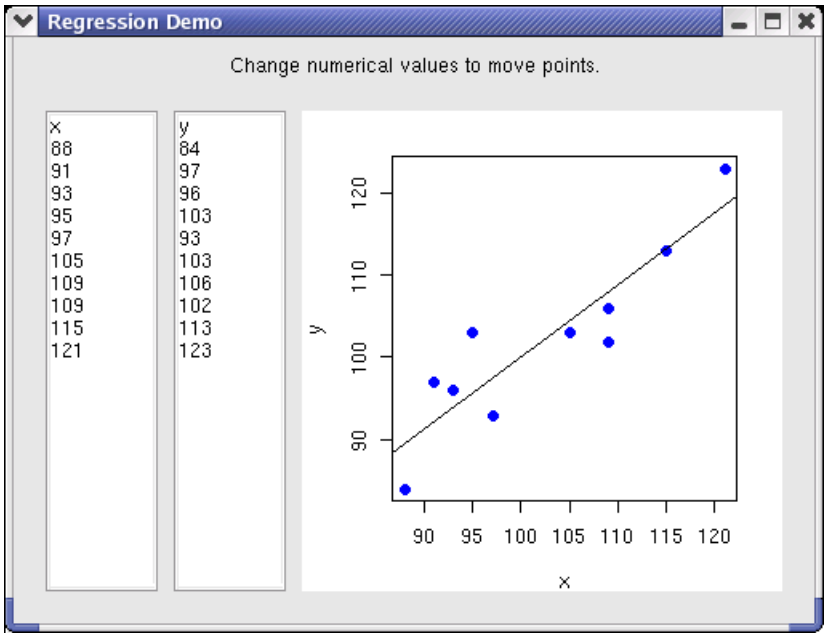

Figure 1: Screen-shot of the Simple Regression Demonstration

2.4 Capturing User Interaction and Updating

The next task is to build callbacks which will register user inputs and update the displayed output. In this case, the user needs to be able to change the values of x and y coordinates. When such changes occur, the demo should redraw the points and the regression line. Definition of a callback requires specification of the widget involved, the action to capture, and the function to be invoked. For the demo, callbacks are needed for each of the two text windows. When the user inserts text the data frame, `reg.frame`, will be updated and the plot redrawn (no action is taken when text is deleted).

```
textx$AddCallback("insert-text", ## Add Callback when x is changed
      function(...) {            ##need a timer here to slow input?
```

```
          tmp <- textx$GetChars(start=0, end=textx$GetLength())
          reg.frame$x <<- as.numeric(unlist(strsplit(tmp,"\n"))[-1])
          redraw()
        })
texty$AddCallback("insert-text", ## callback for when y is changed
          function(...) {          ##need a timer here to slow input
            tmp <- texty$GetChars(start=0, end=texty$GetLength())
            reg.frame$y <<- as.numeric(unlist(strsplit(tmp,"\n"))[-1])
            redraw()
          })
```

The above implementation is clumsy in that if a user is inserting a number which is more than a single character, the callback is activated for each digit, rather than at the end of the inputs. It also suffers from jumpiness, in that the range of x and y plotted depend on the values specified, so when points change, the plotting area shifts. The first extension we will consider removes these problems.

3 Extensions

3.1 RGtkExtra

Another set of bindings provided by Duncan Temple Lang, the RGtkExtra package (`http://www.omegahat.org/RGtkExtra/`) , allows one to use a spreadsheet widget based on the `gtk+extra` library (`http://gtkextra.sourceforge.net/`). The package also provides icon lists and directory trees, but has the disadvantage of being unavailable for MSWindows platforms. Installing the `gtk+extra` library is non-trivial, requiring several other libraries, as documented at `http://gtkextra.sourceforge.net/`.

Using the data sheet, the predictor and response can be displayed as a two-column spreadsheet. When the user changes a cell and presses enter (or up/down arrow), a callback performs the same updates as before, but now data points will not be moved out of the viewing area. The following code was inserted before definition of the text boxes for x and y, and the text box definitions used above were wrapped into an `else` statement.

```
if(require(RGtkExtra)){
  ## Use  code from dataViewer in RgtkViewers package by D. Temple Lang.
  sheet <- gtkSheetNew(rows=nrow(reg.frame), cols=2, show = FALSE)
   sheet$ColumnButtonAddLabel(0, "x")  ## First column label
   sheet$ColumnButtonAddLabel(1, "y")  ## Second column label
   sheet$ShowColumnTitles()
   for (j in 1:2) {         ## Set initial values
     for (i in 1:nrow(reg.frame)) {
          sheet$SetCellText(i - 1, j - 1, as.character(reg.frame[i,j]))
   } }
  sheet$AddCallback("set-cell", function(sheet, i, j) {
```

```
      ## Function to replot points and redraw the regression line
      ## If input is off the "page", it moves points
      ##   to the edge of the plotting region.
      newVal <- as.numeric(sheet$CellGetText(i, j))
      if (reg.frame[i + 1, j + 1] != newVal){
        if(j == 0){              ## the 'x' column
          if(newVal < xlimits[1]){
            newVal <- ceiling(xlimits[1])
            sheet$SetCellText(i, j, as.character(newVal))
          }
          else if(newVal > xlimits[2]){
             newVal <- floor(xlimits[2])
            sheet$SetCellText(i, j, as.character(newVal))
       }  }
       else if(j == 1){
        if(newVal < ylimits[1]){
            newVal <- ceiling(ylimits[1])
            sheet$SetCellText(i, j, as.character(newVal))
          }
          else if(newVal > ylimits[2]){
             newVal <- floor(ylimits[2])
            sheet$SetCellText(i, j, as.character(newVal))
        }  }
        reg.frame[i+1,j+1] <<- newVal
        redraw()
      }
    })
    box2$PackStart(sheet)
    sheet$Show()
 }
else{
     ##  text box code from version 1 goes here
 }
```

Notes: The cell indices returned by gtk are 0-based, so add 1 to use them as indices in R. One callback for the entire sheet is used, rather than one for x, and another for y.

The demo using a data sheet is shown in Figure 2.

3.2 More Widgets

Several other types of widgets will be added to illustrate various capabilities of RGtk. First, check buttons will be added to allow the user to choose which lines will be plotted (least squares and/or a robust fit). Each check box has its own callback, which will change the redraw function.

```
  ## Button to choose lm fit
```

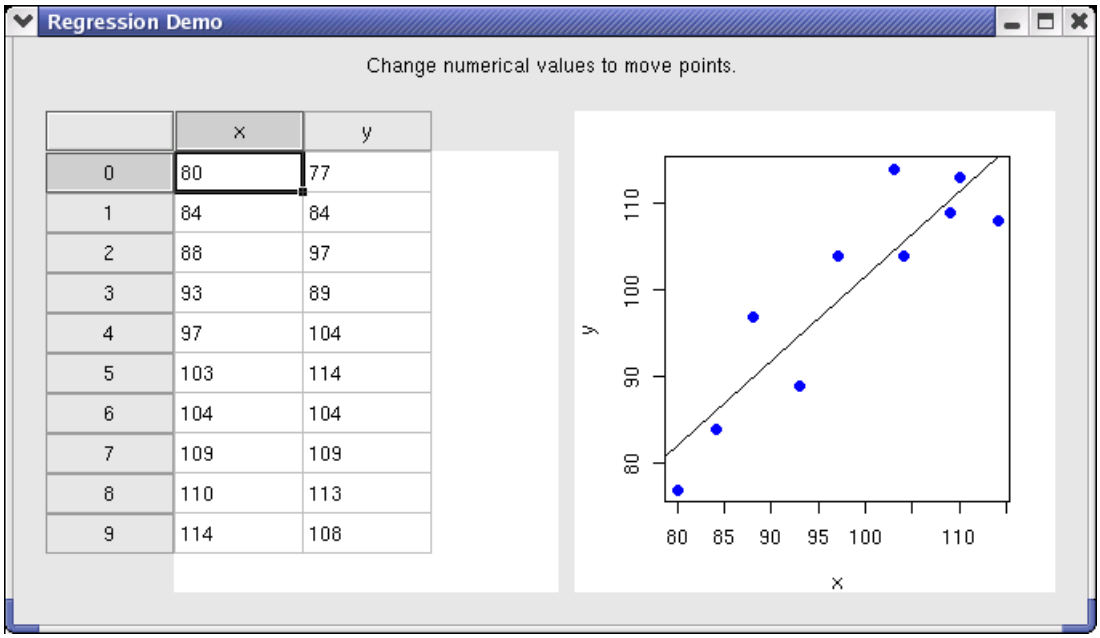

Figure 2: Screen-shot of Regression Demo Using Data Sheet

```
  checkLM <- gtkCheckButtonNewWithLabel("Show Least Squares Fit (Black)",FALSE)
  checkLM$SetUsize(20,20)
  ## Button to choose resistant fit
  require(MASS)                      ## (Venables and Ripley, 2002)
  checkRLM <- gtkCheckButtonNewWithLabel("Show Resistant Fit (Red)", FALSE)
  checkRLM$SetUsize(20,20)
  checkRLM$SetActive(FALSE)  ## Set lm  fit to be drawn initially
  checkLM$SetActive(TRUE)    ## Set rlm fit to be undrawn initially

  ## Add callbacks for the toggled buttons
  checkLM$AddCallback("toggled", quote(redraw()))  ## redraw if toggled
  checkLM$Show()
  checkRLM$AddCallback("toggled", quote(redraw()))  ## redraw if toggled
  checkRLM$Show()
## redefine the redraw function to include robust fit and lowess smoother
  loess.span <- .5
  redraw <- function(){
     ## function to plot points and draw regression line
     plot(y ~ x, data = reg.frame,  xlim=xlimits, ylim=ylimits,
          xlab = "x",ylab="y", pch = 16, col=4)
     if( checkLM$GetActive() )  abline(lm(y~x, data = reg.frame), col=1)
     if( checkRLM$GetActive() )  abline(rlm(y~x, data = reg.frame), col=2)
```

```
    lines(lowess(reg.frame$x, reg.frame$y, loess.span), col=4)
}
```

The callbacks are activated whenever the check boxes are "toggled", meaning whenever the user clicks them on or off. Their actions are to invoke the `redraw` function which now assess the state of each check box in order to add – or not – the appropriate line to the plot.

Next, to illustrate the use of a sliding scale, a loess smoother will be added. (No justification will be made for pedagogical appropriateness.) Whenever the user changes the scale, the callback will redraw the plot. The `gtkAdjustment` function sets up the initial value of the scale, the lower and upper limits, and the increments of change per mouse-click.

```
## Define the range of values for loess.span and the increments of movement
 # This allows the span to vary from 0 to 1 with increments of .1
 smoothness <-  gtkAdjustment(loess.span, 0, 1.1, .1, .1, .25)
 hscale <- gtkHScale (smoothness)
 hscale$SetUsize ( 100, 20)
 # Create a callback for when the slider thumb is moved.
 smoothness$AddCallback("value-changed",
    function(adj) {  ##need a timer here to slow input for dragging
       tmp <- adj$GetValue()
       loess.span <<- tmp
       redraw()
    })
```

In order to pack the new widgets and a label into the display window, I will use a table which is more efficient than several boxes.

```
 label2 <-  gtkLabel( paste(
     "Smoothness of the Lowess smoother goes from 0 to 1.",
     "Move the slider to set the smoothness",sep="\n"), show=FALSE)
# Create table to hold Check Buttons and loess-smoothness slider
table1 <- gtkTable(3,2, homo=TRUE, show=FALSE)
table1$Attach(checkLM,  left.attach=0, right.attach=1,
  top.attach=0, bottom.attach=1)                  ## Top Left Cell
table1$Attach(checkRLM,  left.attach=0, right.attach=1,
  top.attach=1, bottom.attach=2)                  ## Bottom Left Cell
 table1$Attach(label2,  left.attach=1, right.attach=2,
  top.attach=0, bottom.attach=1)                  ## Top Right Cell
 label2$Show()
table1$Attach(hscale,  left.attach=1, right.attach=2,
  top.attach=1, bottom.attach=2)                  ## Bottom Right Cell
 hscale$Show()
```

Finally, text boxes are added to output the equations of the lines. Unlike those used for x and y inputs, these are not set as editable. Because the code adds nothing new, it is not shown here. The final demo is shown in Figure 3.

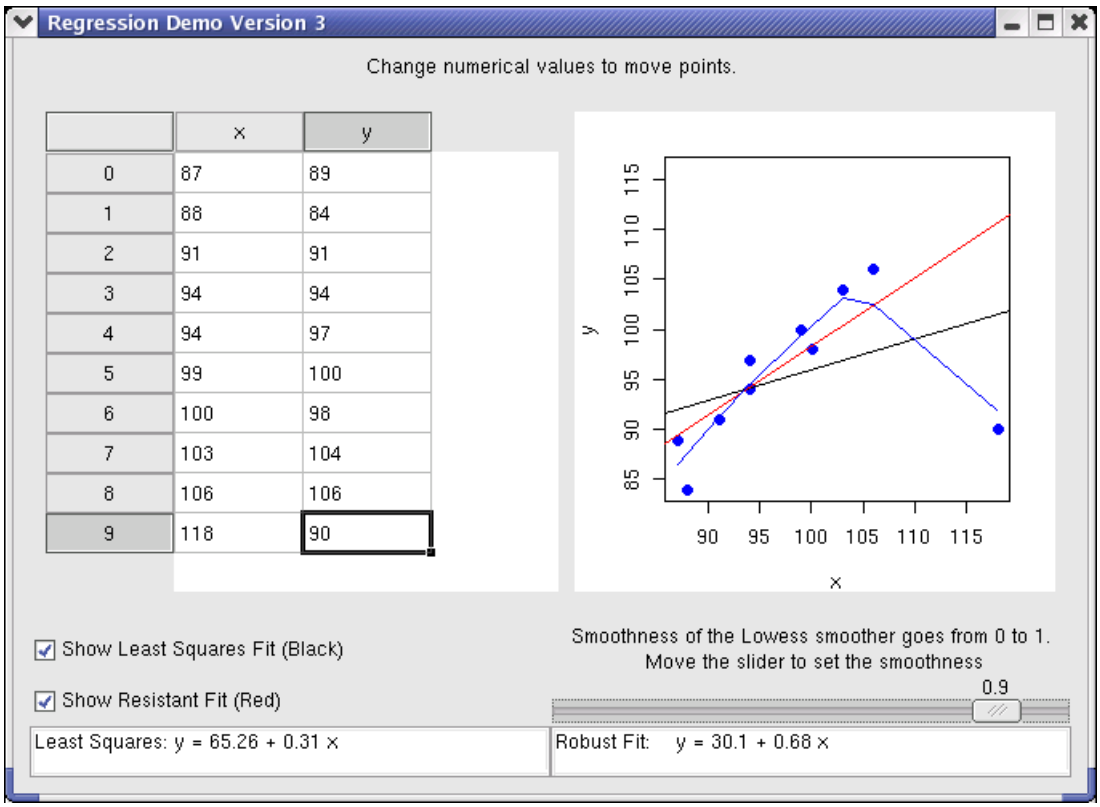

Figure 3: Screen-shot of the Final Regression Demonstration

4 Getting Help

RGtk and RGtkExtra do not have help pages for each function. Current best advice is to locate the desired gtk functions from gtk help pages, and then find the corresponding R function.

Help is available for gtk functions on the web. The gtk site offers a reference manual, `http://developer.gnome.org/doc/API/gtk/index.html`, and a tutorial, `http://www.gtk.org/tutorial/`. Some of the widgets are not currently documented. The developers' advice is to read the headers of those widgets to find out what inputs are needed and what outputs are available. That advice (and the tutorial in general) assumes one can read and understand C code, but other choices are available. Bindings have been created to translate Python to gtk. The pygtk tutorial is much easier to translate into R than the C API version. It's online at `http://www.moeraki.com/pygtktutorial/pygtktutorial/` For Perl users, a useful tutorial is `http://personal.riverusers.com/~swilhelm/gtkperl-tutorial/`.

Once the appropriate gtk functions have been found, one may list the corresponding R functions using `ls()` and find their arguments. In the following example, `search()` was used to determine that `package:RGtk` was in the fourth position. All functions associated with "CheckButton" are listed, and arguments for one of the functions are shown.

```
> ls(pos=4,patt="CheckButton")
[1] "gtkCheckButton"              "gtkCheckButtonNew"
[3] "gtkCheckButtonNewWithLabel"
> args(gtkCheckButtonNewWithLabel)
function (label, show = TRUE, .flush = TRUE)
```

Acknowledgments: Thanks go to Duncan Temple Lang for not only developing the RGtk bindings, but also for generously answering many questions.

References

Dalgaard, Peter. (2002) "Changes to the R-Tcl/Tk package", *R News* **2**:3, 25–27. `http://CRAN.R-project.org/doc/Rnews/`

Ihaka, Ross, and Gentleman, Robert. (1996) "R: A language for data analysis and graphics", *Journal of Computational and Graphical Statistics*, **5**(3): 299-314.

Mills, Jamie D. (2002). "Using Computer Simulation Methods to Teach Statistics: A Review of the Literature" *Journal of Statistics Education* **10**:1.

Temple Lang, Duncan, (2002). "The RGtk package"
`<http://www.omegahat.org/RGtk/index.html>` Nov. 30, 2002.

Temple Lang, Duncan, (2002). "The RGtkExtra package"
`<http://www.omegahat.org/RGtkExtra/index.html>` Nov. 21, 2002.

Venables, William N., and Ripley, Brian D. (2002) *Modern Applied Statistics with S*, Springer, New York.